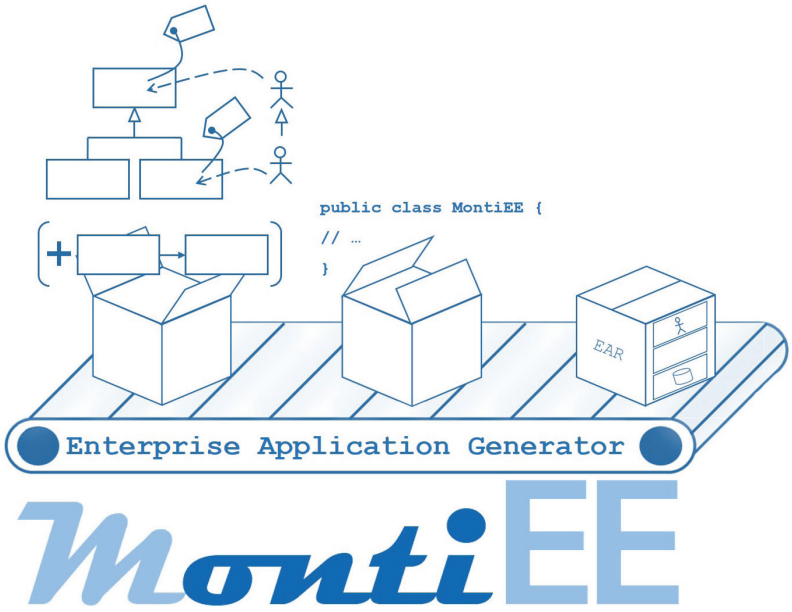
SE | RWTH AACHEN UNIVERSITY

Markus Look

Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE



Aachener Informatik-Berichte,
Software Engineering

Band 27

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker Diplom-Wirtschaftsinformatiker
Markus Look
aus Düren-Lendersdorf

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Wilhelm Hasselbring

Tag der mündlichen Prüfung: 15. Dezember 2016

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

[Loo17] M. Look:
Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.
Shaker Verlag, ISBN 978-3-8440-5131-5. Aachener Informatik-Berichte, Software Engineering, Band 27. 2017.
www.se-rwth.de/publications/

Kurzfassung

Um Geschäftsprozesse eines Unternehmens in der IT abzubilden, werden Enterprise Applikationen eingesetzt. Wesentliche Funktionalitäten dieser sind Speicherung und Bereitstellung von Daten für Benutzer mit unterschiedlichen Rollen und heterogenen Clients. Enterprise Applikationen können aus mehreren Servern mit unterschiedlichen Aufgaben bestehen und von unterschiedlichen Clients verwendet werden. Kommunikation findet zwischen Applikationsservern, zwischen diesen und Datenbanken, aber auch zwischen mehreren Clients und einzelnen Applikationsservern statt. Sowohl die Kommunikationstechnologie als auch der Inhalt muss den Beteiligten bekannt sein. Die Kommunikation ist meistens durch den Client initiiert, so dass die konkreten Clients dem Applikationsserver unbekannt sind. Der Applikationsserver kann unter Verwendung verschiedener Kommunikationstechnologien und –formate clientspezifische Schnittstellen bereitstellen.

Im Software Engineering werden Modelle zu unterschiedlichen Zwecken, wie zur Spezifikation und zur Codegenerierung, eingesetzt. Durch häufig wiederkehrende Aufgaben bei der Entwicklung und der Verwendung einer Schichtenarchitektur eignet sich der Einsatz von Generatoren gut zur Umsetzung von Enterprise Applikationen. Im Rahmen dieser Dissertation wird mit MontiEE die modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen durch Generatoren unterstützt. MontiEE enthält eine Sprachfamilie zur Modellierung sowie verschiedene Generatoren zur Umsetzung von Enterprise Applikationen. MontiEE unterstützt die Modellierung der Datenpersistenz, der Mehrbenutzerfähigkeit und heterogener Clients. Durch die Generatoren kann von benötigten technologiespezifischen Informationen abstrahiert werden, so dass sich vollständig auf die Entwicklung der Geschäftslogik fokussiert werden kann. Darüber hinaus werden Sprachen zur Modellierung der Evolution und Generatoren zur Datenmigration zur Verfügung gestellt.

Die wichtigsten Ergebnisse dieser Arbeit sind:

- Eine Methodik zur Ableitung domänenspezifischer Sprachen zur schematischen Anreicherung von Modellen mit technologiespezifischen Informationen.
- Eine Sprachfamilie zur Modellierung von Mehrbenutzerfähigkeit, zur Berücksichtigung heterogener Clients und zur Modellierung von Evolution.
- Generatoren zur Generierung weiter Teile, wie Persistenz, Kommunikations- und Migrationsinfrastruktur, von Enterprise Applikationen.
- Eine Methodik zur Entwicklung von Enterprise Applikationen mit MontiEE.

MontiEE wurde in verschiedenen Fallstudien eingesetzt, in denen gezeigt werden konnte, dass MontiEE insgesamt ein umfassendes Framework zur agilen Modellierung, Generierung und Evolution mehrbenutzerfähiger Enterprise Applikationen ist. MontiEE unterstützt sowohl die Entwicklung als auch die Weiterentwicklung von Enterprise Applikationen.

Abstract

To reflect a company's business processes in IT Enterprise Applications are put in place. The major functionality of such applications is persisting and offering data to different users with different roles using heterogeneous clients. Enterprise Applications may consist of several servers. Clients may use these servers which are fulfilling different tasks. Communication takes place between application servers, between application servers and databases as well as between multiple clients and a single application server. The communication technology as well as the content has to be known to all participants. Usually, communication is initiated by clients unknown to the application server. The application server can offer specific interfaces to various clients, enabling different communication technologies and formats.

Software Engineering uses models for different purposes, such as specification and code generation. Due to repeatedly reoccurring tasks during the development of enterprise applications and the utilisation of a tiered architecture the use of code generators does fit the implementation of enterprise applications well. In this dissertation MontiEE supports the model-driven, agile development of multi-user Enterprise Applications through code generators. MontiEE contains a language family for modelling as well as various generators for implementing Enterprise Applications. MontiEE supports modelling data persistency, multi-user capability, and heterogeneous clients. Because of the code generators an abstraction is made possible so that the developer can completely focus on implementing the business logic. Furthermore, languages for modelling evolution and generators for data migration are provided.

The main contributions of this thesis are:

- A methodology for deriving domain specific languages for a systematic enrichment of models with technology specific information.
- A language family for modelling multi-user capability, for considering heterogeneous clients, and for modelling evolution.
- Generators for generating major parts of Enterprise Applications, such as persistency, communication and migration infrastructure.
- A methodology for developing Enterprise Applications with MontiEE.

MontiEE has been used in different case studies showing that overall MontiEE offers a comprehensive framework for the agile modelling and generating of multi-user Enterprise Applications. MontiEE supports development as well as advancement of Enterprise Applications.

Danksagung

An dieser Stelle möchte ich mich bei all den Menschen bedanken, die mich während meiner Promotion begleitet und unterstützt haben. Sie alle haben wesentlich zum Gelingen beigetragen.

Mein erster Dank gilt Prof. Dr. Bernhard Rumpe für die Möglichkeit der Promotion am Lehrstuhl Software Engineering der RWTH Aachen. Seine Betreuung, seine konstruktiven Diskussionen und seine wertvollen Ratschläge formten die Arbeit maßgeblich. Neben der abwechslungsreichen und spannenden akademischen Arbeit wurden mir auch tiefe Einblicke und vielschichtige Erfahrungen in der Praxis, bei der Erstellung von Projektanträgen, der Durchführung dieser und der Leitung der großartigen Arbeitsgruppe *Energie* ermöglicht.

Darüber hinaus möchte ich Prof. Dr. Wilhelm Hasselbring für die Bereitschaft zur Übernahme der Zweitkorrektur danken. Auch Prof. Dr. Ir. Joost-Pieter Katoen möchte ich für die Leitung sowie Prof. Dr. Urlik Schroeder für die Mitarbeit in meinem Prüfungskommitee danken.

Auch meinen Kollegen am Lehrstuhl möchte ich großen Dank aussprechen. Die vielen Diskussionen, spannenden Forschungsvorhaben und gemeinsamen Errungenschaften, aber auch gemeinsame Freizeitaktivitäten haben mir immer viel Freude bereitet. Wir haben viel erlebt, viel gearbeitet und viel gelacht. Es war mir eine Freude, die letzten Jahre mit euch zusammen arbeiten und auch feiern zu dürfen. Daher möchte ich Kai Adam, Vincent Bertram, Marita Breuer, Lennart Bucher, Arvid Butting, Gereon Bürvenich, Robert Eikermann, Timo Greifenberg, Dr. Arne Haber, Lars Hermerschmidt, Dr. Christoph Herrmann, Andreas Horst, Katrin Hölldobler, Oliver Kautz, Thomas Kurpick, Evgeny Kusmenko, Achim Lindt, Ben Mainz, Klaus Müller, Antonio Navarro Pérez, Dr. Pedram Mir Seyed Nazari, Sebastian Oberhoff, Jerome Pfeiffer, Dr. Claas Pinkernell, Dimitri Plotnikov, Manuel Pützer, Dr. Dirk Reiss, Dr. Holger Rendel, Dr. Jan Oliver Ringert, Alexander Roth, Dr. Martin Schindler, Christoph Schulze, Brian Sinkovec, Minh Tran, Galina Volkova, Max Voß, Michael von Wenckstern und Dr. Andreas Wortmann herzlich für die letzten Jahre danken.

Ganz besonders erwähnen möchte ich die Mitglieder der Arbeitsgruppe *Energie*. Es hat stets sehr viel Spaß gemacht, gemeinsam mit euch Projekte zu bearbeiten und neue zu akquirieren. Ebenfalls hervorheben möchte ich die vielen hilfsbereiten Freunde und Kollegen, die sich bereit erklärt haben, die Arbeit Korrektur zu lesen und Verbesserungsvorschläge einzubringen. Danke Andreas, Claas, Dimitri, Ingrid, Katrin, Robert, Ruth, Timo.

Darüber hinaus möchte ich mich bei den Kollegen bedanken, die immer ein offenes Ohr für mich hatten, stets meinen unterschiedlichsten Anliegen Aufmerksamkeit geschenkt haben und mir mit Rat zur Seite standen. Auch allen beteiligten Hiwis, Studenten, Abschlussarbeitern, die ich betreuend unterstützen durfte, möchte ich danken. Ihr alle,

Kollegen, Freunde und Studenten, habt mir sehr geholfen.

Großen Dank möchte ich meinen Eltern, Ingrid und Norbert, die mich fortwährend unterstützt haben, aussprechen. Sie haben mir diese Promotion und den Weg dorthin erst ermöglicht und waren jederzeit für mich da. Bei meinen Schwiegereltern Petra und Alfred möchte ich mich ebenfalls bedanken. Auch auf ihre Unterstützung konnte ich immer bauen.

Besonderer Dank gilt meiner Frau Ruth Maria, die mich stets unterstützt und motivert hat. Ihre Unterstützung, ihr Zuspruch und ihre Geduld haben maßgeblich zum Erfolg dieser Arbeit beigetragen.

Aachen, Februar 2017
Markus Look

Inhaltsverzeichnis

Teil I

Grundlagen

Kapitel 1

Einführung

Um Geschäftsprozesse eines Unternehmens in der IT abzubilden und dadurch deren Effizienz und Qualität zu verbessern, werden Enterprise Applikationen eingesetzt. Wesentliche Funktionalitäten dieser sind Speicherung und Bereitstellung von Daten für Benutzer mit unterschiedlichen Rollen und heterogenen Clients. Zur Umsetzung einer Enterprise Applikation existieren erprobte Architekturen, Technologien und Entwicklungsmethodiken, die die Umsetzung ermöglichen oder erleichtern [SSN01, Fow03, RH06, Kaj12].

Software sowie die Architektur von Software haben sich in den letzten Jahren und Jahrzehnten stark verändert. Heutzutage existierende Software hat sich von Desktopanwendungen hin zu Webanwendungen gewandelt. Eine Desktopanwendung ist dabei eine monolithische Anwendung, die alle Funktionen, die für eine Aufgabe benötigt werden, bereitstellt. Sie läuft unabhängig auf einem einzelnen Computer und wird von einem einzelnen Benutzer an eben diesem Rechner verwendet. Ein Austausch von Daten ist nur in Form eines manuellen Dateiaustausches oder eines geteilten Speicherplatzes möglich. Dabei wurden die Daten meistens kopiert und dem anderen Nutzer übergeben. Die Verwendung einer solchen Anwendung als Enterprise Applikation im Unternehmen ist nur mit großem Aufwand möglich.

Zur Vereinfachung der Kollaboration wurden die ersten Formen von Client-Server Systemen und Architekturen entwickelt. Diese besitzen einen zentralen Server, zu dem sich die einzelnen Clients verbinden und der die Geschäftsdaten in Datenbanken vorhält. Mit Hilfe von Transaktionskonzepten kann die Konsistenz der Daten gesichert werden. Als Datenbanken kommen meistens relationale Datenbanken zum Einsatz. Die Server stellen den Clients, welche meist noch als Desktopanwendungen umgesetzt wurden, ein Konglomerat verschiedener Funktionalitäten über proprietäre Protokolle zur Verfügung. Dadurch entsteht eine starke Kopplung zwischen dem Server und seinen Clients. Zudem führt ein Server unterschiedliche, nicht zusammengehörige Funktionalitäten aus [TMD09]. Zur Bewältigung der immer größer werdenden Zahl der Clients wurde begonnen, eine Skalierung durch Replikation der Server zu erreichen. Ferner wurde, um die monolithischen Server abzulösen, begonnen, Systeme nach ihrer Funktion zu modularisieren. Dies hat zur Folge, dass die unterschiedlichen Funktionalitäten unabhängig voneinander auf einem Server erreichbar sind. Durch die Anwendung von Cloud-Computing Technologien kann Skalierung elastisch, also bei Bedarf, erfolgen. Gleichzeitig wird diese Funktionalität den Clients über offene Schnittstellen, beispielsweise in Form von Webservices, bereitgestellt. Dadurch konnte die starke Kopplung gelöst werden. Auch die Clients haben sich weiterentwickelt. Die Verfügbarkeit und Kapazität unterschied-

licher Endgeräte, wie Computer, Tablet-PCs oder Smartphones, sind gestiegen. Zudem ist der Bedarf an Software, die überall verfügbar ist, stark gestiegen. Dies bedeutet, dass die von Services angebotene Funktionalität von unterschiedlichen Clients genutzt wird und die angebotenen Services komponiert werden. Dabei wird auf native Clients und Webanwendungen gesetzt. Zur Umsetzung von Enterprise Applikationen auf Applikationsservern wird eine solche modularisierte und lose gekoppelte Architektur von Servern und Clients benötigt, die den unterschiedlichen Nutzern verschiedene Funktionalitäten auf heterogenen Plattformen bereitstellt. Solche Systeme sind weit verbreitet und stellen den de facto Standard der Implementierung dar.

Heute hat sich das Verständnis einer solchen Architektur bedingt durch die allgegenwärtige Digitalisierung weiterentwickelt. Viele Prozesse des täglichen Lebens durchlaufen eine digitale Transformation hin zu digitalisierten Prozessen. Dies hat zur Folge, dass immer mehr Daten verfügbar werden und zur Verarbeitung genutzt werden können. Diese anfallenden Daten können zur Steuerung und Optimierung anderer digitaler Prozesse verwendet werden. Ebenso hat Digitalisierung zur Folge, dass immer neue Technologien benötigt werden, um die anfallenden Daten zu bearbeiten und gleichzeitig performant zu bleiben. Dadurch wurden neue Datenbankparadigmen, wie NoSQL Datenbanken, neue Kommunikationsarchitekturen und Darstellungsmöglichkeiten, wie Responsive Webdesign, geschaffen. Neuerdings werden Microservices [New15] als Architekturgrundlage verwendet. Einer solchen Architektur liegt die Idee zu Grunde, dass nicht mehr die Funktionalität im Sinne einer technischen Funktionalität, wie Skalierung, sondern domänenspezifische Funktionalität, die die Prozessfunktionalität abbildet, im Vordergrund steht [Eva04]. Dadurch werden Microservices geschaffen, die einzelne Prozesse innerhalb eines Unternehmens abbilden. Die einzelnen Services können durch flexible Komposition eine übergeordnete Funktionalität anbieten. Sie lassen sich unabhängig entwickeln, skalieren und von unterschiedlichen Clients verwenden, da sie eine domänenspezifische Funktionalität erfüllen und somit eine für den Nutzer erfahrbare Funktionalität bieten.

Enterprise Applikationen sind oft Systeme, die einer Client-Server Architektur folgen und viele heterogene Clients besitzen. Sie verwenden verschiedene Datenbanken und werden auf unterschiedlichen Applikationsservern ausgeführt. Die Entwicklung von Enterprise Applikationen ist aufwendig, komplex und folgt eigenen Paradigmen und Entwurfsmustern. Diese Komplexität resultiert sowohl aus der Domäne als auch aus dem Technologiestack [Fow03]. Dies erfordert vom Entwickler tiefgehende Kenntnis über alle eingesetzten Technologien, da ein Großteil der aufzubringenden Entwicklungsleistung in die Konfiguration und die Entwicklung technologiespezifischer Teile abseits der eigentlichen Funktionalität fließt. Für viele technische Funktionalitäten existieren Frameworks, die dazu gedacht sind, die Entwicklung zu vereinfachen. Auch wenn diese Frameworks eine gewisse Abstraktion bieten, müssen auch diese korrekt verwendet und konfiguriert werden. Der Entwickler muss spezifisches Wissen über die Verwendung des Frameworks besitzen. Die benötigten technologiespezifischen Teile und die Konfiguration der Technologie oder der Frameworks findet darüber hinaus als weitere Hürde nicht im Quellcode, sondern meist in XML- oder Property-Dateien statt. Dies führt, obwohl sich diese gegenseitig beeinflussen, zu einer Trennung von Quellcode und Konfiguration. Häufig

werden Teile des Quellcodes aus den Konfigurationsartefakten referenziert. Gleichzeitig referenzieren die Konfigurationsartefakte zusätzlich noch Artefakte der Infrastruktur, die ebenfalls konfiguriert werden müssen. Auch diese Referenzen müssen übereinstimmen. Dies führt während der Entwicklung leicht zu Inkonsistenzen. Frameworks, wie die Java Persistence API (JPA), verwenden zur Konfiguration Annotationen im Quellcode [JPA16]. Dies reduziert die Trennung von Konfiguration und Quellcode, ist allerdings plattformspezifisch. Auch wenn diese Annotationen die Situation der Inkonsistenzen ein wenig verbessern, beruht eine Konfiguration dennoch auf frei wählbaren Namen, die aber über das gesamte System hinweg konsistent sein müssen. Es existieren Ansätze, die diese Konfiguration vor dem Entwickler nahezu vollständig verbergen. Sie nehmen ihm aber auch gleichzeitig die Möglichkeit der Konfiguration. Gerade bei der Vielzahl der heute zu entwickelnden Applikationen und ihren unterschiedlichen Anforderungen muss der Entwickler Einfluss auf dieses Verhalten haben, da er die Applikation spezifisch an die Situation anpassen können muss.

Eine Reduktion dieses Aufwands führt zu einer schnelleren, effizienteren Entwicklung und zu einer Abstraktion von technologiespezifischen Teilen. Dadurch können Entwickler sich auf die Umsetzung der gewünschten Funktionalität konzentrieren und müssen keine Detailkenntnis über die jeweiligen Technologien haben. Die Erstellung der Konfiguration, die Erstellung der technologiespezifischen Teile und die Entwicklung von Enterprise Applikationen beinhalten viele Aufgaben, die sehr ähnlich und analog erfüllt werden können, aber dennoch stets manuell gemacht werden müssen. Ein Großteil dieser Aufgaben lässt sich mit Hilfe modellgetriebener Techniken und Analysen sowie dem zielgerichteten Einsatz von Codegeneratoren verbessern. Dadurch wird der Technologiestack beherrschbarer und die Zeit besser genutzt, so dass die Fokussierung auf das Wesentliche, die Entwicklung der Systemfunktionalität, unterstützt wird.

Im Rahmen dieser Arbeit wird MontiEE zur modellgetriebenen Entwicklung von Enterprise Applikationen vorgestellt. MontiEE enthält eine Sprachfamilie zur Modellierung solcher Applikationen sowie verschiedene Generatoren zur Umsetzung von Enterprise Applikationen. Dadurch wird eine Modellierung ermöglicht, die von technologiespezifischen Details abstrahiert, aber dennoch eine applikationsspezifische Konfiguration ermöglicht. Gleichzeitig unterstützt die im Rahmen dieser Arbeit mit MontiEE entwickelte Methodik die Berücksichtigung der Persistenz von Daten, vieler Benutzer und heterogener Clients. Durch die Generatoren kann ein Großteil der Enterprise Applikation generiert werden. Der generierte Teil enthält nahezu alle technologiespezifischen Artefakte sowie die technologiespezifischen Repräsentationen der modellierten fachlichen Objekte der Domäne und der Clients. Der Entwickler einer Enterprise Applikation kann sich vollständig auf die Entwicklung der Geschäftslogik fokussieren und muss keine tiefgehende Kenntnis der Technologie haben.

Im nächsten Abschnitt werden die wichtigsten Ziele dieser Arbeit näher erläutert. Daran anschließend werden die wichtigsten Ergebnisse und der Aufbau der Arbeit vorgestellt.

1.1 Wichtigste Ziele der Arbeit

Im Rahmen dieser Arbeit wird MontiEE als Framework zur modellgetriebenen Entwicklung von Enterprise Applikationen vorgestellt. Das wichtigste Ziel der Arbeit lässt sich mit nachfolgender Forschungsfrage ausdrücken:

> *Wie lässt sich die modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen durch Generatoren unterstützen?*

Die vorliegende Arbeit hat sich demnach zum Ziel gesetzt, eine Entwicklungsmethodik und eine geeignete Umsetzung für die Entwicklung von Enterprise Applikationen durch Generatoren zu schaffen. Enterprise Applikationen sind Systeme, die Geschäftsprozesse unterstützen und meistens in einem komplexen Technologiestack zur Erfüllung ihrer Aufgaben implementiert sind. Dazu verarbeiten und speichern sie Daten persistent und stellen sie unterschiedlichen Benutzern mit unterschiedlichen Befugnissen zur Verfügung. Enterprise Applikationen werden meistens von mehreren Benutzern gleichzeitig verwendet und müssen mit unterschiedlichen Arten von Clients kommunizieren können. Die Möglichkeit, Enterprise Applikationen mit unterschiedlichen Arten von Clients zu verwenden, und eine hohe Verfügbarkeit machen sie ubiquitär. Diese hohe Verfügbarkeit stellt gleichzeitig Anforderungen an die Infrastruktur, aber auch die Weiterentwicklung solcher Systeme. Innerhalb eines agilen Entwicklungsprozesses, wie in der Forschungsfrage gefordert, muss auch die Weiterentwicklung und Evolution agil sein, darf aber dennoch nicht dazu führen, dass die Enterprise Applikation nicht erreichbar ist. Darüber hinaus wird im Rahmen dieser Arbeit die Eignung eines modellgetriebenen Ansatzes untersucht. Der letzte Teil der Forschungsfrage zielt auf die Untersuchung der Möglichkeiten zur Unterstützung der Entwicklung von Enterprise Applikationen durch Generatoren. Dazu werden im Rahmen dieser Arbeit unterschiedliche Generatoren auf einer festgelegten Zielplattform umgesetzt und somit die Möglichkeiten zur Unterstützung gezeigt.

Die vorgestellte Forschungsfrage lässt sich unter Berücksichtigung der Eigenschaften von Enterprise Applikationen noch weiter untergliedern, deren Beantwortung eine Antwort auf die übergeordnete Forschungsfrage ermöglicht:

RQ1: Welche Modelle werden zur Abstraktion von dem komplexen Technologiestack benötigt und wie kann der Abstraktionsgrad erhalten bleiben?

RQ2: Wie lassen sich die Daten, die eine Enterprise Applikation speichert, verarbeitet und wieder zur Verfügung stellt, modellieren und wie lässt sich diese Funktionalität aus Modellen generieren?

RQ3: Welche Modelle eignen sich zur Modellierung verschiedener Benutzer mit unterschiedlichen Funktionen im Geschäftsprozess und wie lässt sich die Bereitstellung der Daten für die verschiedenen Benutzer generieren?

RQ4: Welche Modelle eignen sich zur Modellierung unterschiedlicher Arten von Clients, die die Enterprise Applikation verwenden, und wie lassen sich für diese clientspezifische Teile des Quellcodes generieren?

RQ5: Wie können die Evolution von Enterprise Applikationen modelliert und die Daten der Enterprise Applikation migriert werden?

Diese fünf Forschungsfragen beschäftigen sich einzeln mit den Eigenschaften von Enterprise Applikationen. Dabei haben sie stets einen modellgetriebenen und einen generativen Anteil. Für alle diese Eigenschaften wird zunächst untersucht, welche Modelle sich zur Modellierung eignen. Daran anschließend wird die Möglichkeit der generativen Umsetzung der Eigenschaften auf Basis der Modelle untersucht. Die Zweiteilung zwischen Modell und Generierung, aber auch die Teilung in die unterschiedlichen Eigenschaften finden sich auch im Aufbau dieser Arbeit wieder. Innerhalb der einzelnen Kapitel werden die Konzepte zur Beantwortung der Teilforschungsfragen vorgestellt, damit die übergeordnete Forschungsfrage beantwortet werden kann. Nachfolgend werden die wichtigsten Ergebnisse der Arbeit vorgestellt. Daran anschließend wird der Aufbau der Arbeit präsentiert.

1.2 Wichtigste Ergebnisse der Arbeit

MontiEE enthält eine Sprachfamilie und zugehörige Generatoren, die die Modellierung und Generierung von Enterprise Applikationen unterstützen. Die einzelnen Sprachen und Generatoren beantworten Teilaspekte der Forschungsfragen und führen zur übergeordneten Forschungsfragestellung hin.

Der übergeordneten Forschungsfragestellung und den Teilforschungsfragen folgend, wurden im Rahmen dieser Arbeit mit MontiEE verschiedene Ergebnisse erreicht, die die modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen durch Generatoren unterstützen. Die wichtigsten Ergebnisse von MontiEE im Rahmen dieser Arbeit sind:

- Eine Methodik zur Ableitung sprachspezifischer Taggingsprachen aus beliebigen DSLs. Durch Taggingsprachen lassen sich Modelle einer DSL mit technologiespezifischen Informationen anreichern, ohne die Modelle selbst zu belasten. Damit werden die Modelle lesbar gehalten und gleichzeitig in den unterschiedlichsten Kontexten wiederverwendbar gemacht.
- Ein Schema zur Anreicherung des Domänenmodells der Enterprise Applikationen um technologiespezifische Informationen zur Konfiguration und Verwendung des komplexen Technologiestacks.
- Rechte-, Rollen- und Zuordnungssprachen zur Modellierung der Mehrbenutzerfähigkeit und der unterschiedlichen Rechte der Rollen.
- Eine Sichtensprache zur Berücksichtigung heterogener Clients, die die Modellierung des Domänenmodells der Clients ermöglicht.
- Eine klassendiagrammspezifische Deltasprache zur Modellierung der Evolution einer Enterprise Applikation.

- Eine allgemeine Infrastruktur für Generatoren zur Berücksichtigung technologiespezifischer Informationen.
- Generatoren für die bei der persistenten Datenhaltung benötigten Komponenten, wie Entitäten, DAOs und Datenbankinitialisierungen.
- Generatoren für die Kommunikationsinfrastruktur mit heterogenen Clients unter Berücksichtigung unterschiedlicher Befugnisse verschiedener Clients.
- Generatoren für die API von der Kommunikationsfassade zur Geschäftslogik und von der Geschäftslogik zur Persistenz.
- Generatoren für die Evolutionsinfrastruktur zur Weiterentwicklung der Enterprise Applikation. Dabei unterliegen Klassendiagramme einer Evolution und persistent gespeicherte Daten werden analog zu dieser Evolution migriert.
- Eine Methodik des MDD mit MontiEE.

Nachdem die wichtigsten Ergebnisse der Arbeit vorgestellt wurden, wird nachfolgend der Aufbau der Arbeit erläutert.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit stellt das Framework MontiEE vor, evaluiert es und grenzt es gegenüber verwandten Frameworks ab. Dazu ist die vorliegende Arbeit wie folgt strukturiert:

Kapitel 2 stellt die Grundlagen der modellgetriebenen Entwicklung vor. Dabei stehen Domänenspezifische Sprachen (DSLs), deren Erstellung, Verwendung und Nutzen in Abschnitt 2.1 besonders im Vordergrund. Daran anschließend wird das Framework MontiCore zur Erstellung domänenspezifischer Sprachen in Abschnitt 2.2 vorgestellt. MontiCore bietet neben der Sprachdefinition noch Möglichkeiten zur Kontextbedingungsprüfung und zur Codegenerierung. In Abschnitt 2.3 wird darauf aufbauend die Sprachfamilie UML/P, die basierend auf der Unified Modelling Language (UML) eine Reihe DSLs zur Modellierung von Systemen bereitstellt, vorgestellt.

Kapitel 3 stellt die Entwicklung von Enterprise Applikationen in den Vordergrund. Dazu werden in Abschnitt 3.1 im Rahmen dieser Arbeit verwendete Definitionen solcher Systeme, ihrer Einsatzgebiete und Herausforderungen angegeben. In Abschnitt 3.2 werden typische Architekturen und verwendete Technologien solcher Systeme vorgestellt. Dabei werden insbesondere die Kommunikation mit Clients und mit Datenbankservern sowie typische Implementierungsframeworks und -muster beschrieben. Abschnitt 3.4 stellt verwandte Frameworks vor. Generelle verwandte Arbeiten werden in den einzelnen Kapiteln vorgestellt, sodass hier auf Frameworks, die den gleichen Ansatz verfolgen, fokussiert wird. In Abschnitt 3.5 wird ein durchgängiges Szenario vorgestellt, anhand dessen Anforderungen sowohl an Enterprise Applikationen als auch an MontiEE vorgestellt werden.

Kapitel 4 stellt den Teil der Sprachfamilie MontiEE vor, der zur Modellierung der Persistenz von Enterprise Applikationen bereitgestellt wird. Dazu wird in Abschnitt 4.2 ein Klassendiagramm als durchgängiges Beispiel vorgestellt. Dieses Klassendiagramm stellt die Modellierung des Szenarios aus Abschnitt 3.5 dar. Die nachfolgenden Abschnitte stellen die im Rahmen dieser Arbeit geschaffenen DSLs vor. Dabei werden stets die konkrete Syntax, die abstrakte Syntax und Kontextbedingungen vorgestellt. In Abschnitt 4.3 werden die Taggingsprachen, die die Anreicherung eines Modells mit zusätzlichen Informationen ermöglichen, vorgestellt. Sowohl die Tagdefinitionssprache zum Taggen von Modellelementen als auch die Tagschemasprache zur Definition von Tagtypen werden hier erläutert. Dabei wird sowohl eine allgemeine Methodik zur Ableitung sprachspezifischer Taggingsprachen als auch eine klassendiagrammspezifische Variante der Tagdefinitionssprache und der Tagschemasprache vorgestellt. Mit Hilfe dieser klassendiagrammspezifischen Variante können Informationen, die für die Modellierung der Persistenz benötigt werden, modelliert werden.

Kapitel 5 stellt den Teil der Sprachfamilie MontiEE vor, der zur Modellierung der Kommunikation zwischen Clients und Server von Enterprise Applikationen bereitgestellt wird. Dazu wird in Abschnitt 5.2 eine Sichtensprache zur Modellierung von Sichten auf Klassendiagramme vorgestellt. Diese Sichten stellen das Domänenmodell der Clients dar. Zudem wird eine Transformation einer modellierten Sicht in ein korrektes Klassendiagramm vorgestellt. In Abschnitt 5.3 werden Sprachen zur Modellierung von Rechten und Rollen sowie deren Zuordnung erläutert, mit deren Hilfe sich die Zugriffskontrolle des Systems modellieren lässt. Darüber hinaus wird eine Methodik vorgestellt, die mögliche Rechte auf Basis des Klassendiagramms ableitet.

Kapitel 6 stellt eine Sprache zur Modellierung der Evolution von Enterprise Applikationen vor. Dazu wird eine existierende Methodik verwendet, die die Ableitung einer klassendiagrammspezifischen Deltasprache ermöglicht. Basierend auf der Methodik wird die Sprache um Operationen erweitert, die ein Refactoring von Klassendiagrammen basierend auf Basisoperationen ermöglichen. Modelle der Sprache werden zur Ableitung der Datenmigration verwendet.

Kapitel 7 stellt die auf Basis der Sprachfamilie MontiEE geschaffenen Generatoren zur Generierung der Persistenz vor. In Abschnitt 7.2 wird ein MontiEE-spezifisches Tagschema vorgestellt, welches die von den Generatoren verwendeten Tagtypen definiert. In Abschnitt 7.3 wird die Generierung und Befüllung der Taginfrastruktur, die von allen MontiEE-Generatoren verwendet wird, erläutert. Abschnitt 7.4 stellt den Entity-Generator, der aus Klassendiagrammen und der Tagdefinitionen Entitäten erzeugt, vor. Abschnitt 7.5 stellt den Data Access Object (DAO)-Generator vor, der aus Klassendiagrammen und der Tagdefinition DAOs erzeugt. In Abschnitt 7.6 wird der Structured Query Language (SQL)-Generator vorgestellt, der aus den gleichen Informationen wie die vorherigen Generatoren ein SQL-Schema erzeugt.

Kapitel 8 stellt die auf Basis der Sprachfamilie MontiEE geschaffenen Generatoren zur Generierung der Kommunikationsinfrastruktur vor. In Abschnitt 8.2 wird der Data Transfer Object (DTO)-Generator vorgestellt, der aus Klassendiagrammen, Sichten und der Tagdefinition DTOs zur Kommunikation mit den Clients erstellt. In Ab-

schnitt 8.3 wird der BusinessAPI-Generator, der die Schnittstelle zwischen der Kommunikationsfassade für Clients und der Geschäftslogik sowie die Schnittstelle zwischen der Geschäftslogik und den DAOs generiert, vorgestellt. In Abschnitt 8.4 wird der Facade-Generator, der aus Klassendiagrammen, Rechte- und Rollen- und Mappingdiagrammen sowie aus der Tagdefinition Kommunikationsfassaden für verschiedene Clients generiert, vorgestellt.

Kapitel 9 stellt die auf Basis der Sprachfamilie MontiEE geschaffenen Generatoren zur Generierung der Evolutionsinfrastruktur vor. Dabei werden Modelle der Deltasprache zur Generierung einer Evolutions- und Migrationsinfrastruktur verwendet. Die Evolution des Klassendiagramms wie auch die Migration der Daten auf Basis von Objektdiagrammen werden beschrieben.

Kapitel 10 stellt die Methodik des Model-Driven Development (MDD) mit MontiEE vor. Dazu werden zunächst die Möglichkeiten von MontiEE auf Basis zweier Aktivitätsdiagramme vorgestellt. Dann wird die Verwendung und Konfiguration von MontiEE in Abschnitt 10.2 erläutert. Anschließend wird in Abschnitt 10.3 das Szenario aus Abschnitt 3.5 exemplarisch mit MontiEE modelliert und umgesetzt. In Abschnitt 10.4 wird das Deployment des Generats von MontiEE gefolgt von der Vorstellung des geschaffenen Toolings in Abschnitt 10.5 erläutert. Abschließend wird in Abschnitt 10.6 die Anwendung von MontiEE in Forschungsprojekten gezeigt und über Erfahrungen bei der Integration in bestehende sowie neuentwickelte Systeme berichtet.

Kapitel 11 fasst diese Arbeit zusammen und gibt einen Ausblick auf offene Fragestellungen.

Kapitel 2

Grundlagen der modellgetriebenen Entwicklung

Nachdem im vorherigen Kapitel Motivation, Ziele, Ergebnisse und der generelle Aufbau der vorliegenden Arbeit erläutert wurden, werden in diesem Kapitel zunächst die wichtigsten in der Arbeit verwendeten Begriffe vorgestellt und definiert. Die verwendeten Techniken entstammen allesamt der agilen Entwicklung [BBB+01], dem Model-Driven Engineering (MDE) und der generativen Programmierung.

Diese Entwicklungsparadigmen haben den Begriff eines Modells gemein. Im Rahmen der vorliegenden Arbeit wird ein Modell dabei als

> *"eine abstrahierende Darstellung eines Originals, welches sie im Hinblick auf einen pragmatischen Zweck beschreibt [Sta73]"*

verstanden. Dabei lässt sich das Original im Software Engineering als das Softwaresystem verstehen, welches sich möglicherweise noch in Entwicklung befindet [Wei12]. Modelle haben unterschiedliche Verwendungszwecke. Sie können zur Kommunikation, zur Analyse, aber auch zur Codegenerierung verwendet werden. Die Verwendung und der Einsatz von Modellen variiert in den verschiedenen Entwicklungsparadigmen.

Dabei wird in der Literatur zumeist zwischen *Model-Based Engineering (MBE)*, *Model-Driven Engineering (MDE)*, *Model-Driven Development (MDD)* und *Model-Driven Architecture (MDA)* unterschieden [SVC06, BCW12, Sch12].

Der Zusammenhang dieser Begriffe ist in Abbildung 2.1 dargestellt. MDA ist eine standardisierte, definierte Menge von Modellierungssprachen und Transformationen, so dass MDA die spezifischste Variante darstellt. Etwas weniger spezifisch ist MDD, das Modelle dazu verwendet, die Implementierung zu spezifizieren und Code zu generieren. Dabei haben die Modelle einen starken Implementierungsfokus. Eine etwas weiter gefasste Definition stellt MDE dar, da hier Modelle auch für andere Zwecke, wie beispielsweise Evolution des Systems, eingesetzt werden. Aber auch diese Modelle stellen primäre Artefakte in der Umsetzung des Systems dar. Der am weitesten gefasste Begriff ist MBE, bei dem Modelle zwar wichtige Artefakte darstellen, aber hauptsächlich der Veranschaulichung, der Kommunikation und der Dokumentation dienen. Das MDE-Paradigma kann oftmals erfolgreich in Entwicklungsprozessen angewendet werden, um diese zu vereinfachen. Die erfolgreiche Anwendung von MDE konnte in verschiedenen Studien gezeigt werden [KR05, Sta06, WWM+07, HRW11, HWRK11, WHR14].

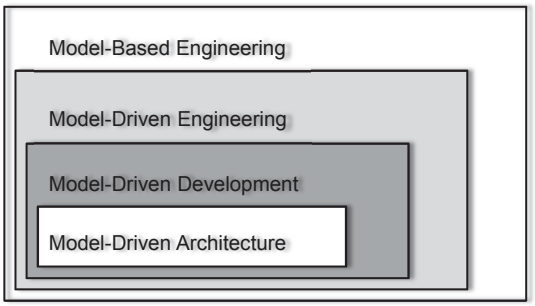


Abbildung 2.1: Darstellung der Beziehungen zwischen den unterschiedlichen Paradigmen: Model-Driven Architecture (MDA), Model-Driven Development (MDD), Model-Driven Engineering (MDE) und Model-Based Engineering (MBE) in Anlehnung an [BCW12].

Diese Arbeit ordnet sich dabei primär in die Domäne des MDD ein, enthält aber Teile des MDE. Dies bedeutet, dass im Rahmen dieser Arbeit Modelle verwendet werden, die der Implementierung eines Systems dienen und dabei mit Hilfe von Codegeneratoren in Code umgesetzt werden. Zudem werden Modelle eingesetzt, um die Evolution des Systems zu unterstützen.

Modelle sind im Allgemeinen in einer geeigneten Modellierungssprache modelliert. Dabei lassen sich *General Purpose Modeling Languages (GPMLs)* und *Domänenspezifische Modellierungssprachen (DSMLs)* unterscheiden. Eine GPML ist dabei, analog zu einer *General Purpose Language (GPL)* wie Java, eine Modellierungssprache, die nicht auf eine spezielle Problemstellung zugeschnitten ist, sondern zur Modellierung unterschiedlicher Dinge eingesetzt wird. Demgegenüber ist eine DSML, analog zu einer DSL, eine Modellierungssprache, die zum Modellieren innerhalb einer Domäne eingesetzt wird und dazu auch domänenspezifische Elemente enthält. Im Rahmen dieser Arbeit werden die Begriffe DSML, DSL und Modellierungssprache synonym verwendet, da diese nicht klar differenzierbar sind. DSLs stellen einen wichtigen Teil dieser Arbeit dar und werden in Abschnitt 2.1 detailliert eingeführt.

Der bekannteste Repräsentant einer GPML ist dabei die *Unified Modelling Language (UML)* [OMG15c] der *Object Management Group (OMG)*. Die UML definiert sowohl Strukturdiagramme als auch Verhaltensdiagramme. Es existieren sieben Struktur- und sieben Verhaltensdiagramme. Innerhalb der Strukturdiagramme sind Klassendiagramme und Objektdiagramme und innerhalb der Verhaltensdiagramme Sequenz- und Aktivitätsdiagramme zu nennen, da diese im Rahmen dieser Arbeit verwendet werden. Die weiteren Diagrammgruppen, neben Struktur- und Verhaltensdiagrammen, und die enthaltenen Diagrammarten werden hier nicht erläutert, sondern können in [OMG15c] gefunden werden.

Klassendiagramme beschreiben die interne Struktur eines Systems und sind sehr implementierungsnah. Sie modellieren einzelne Klassen, Interfaces und deren Beziehungen untereinander. Objektdiagramme hingegen stellen einen Systemzustand zu einer System-

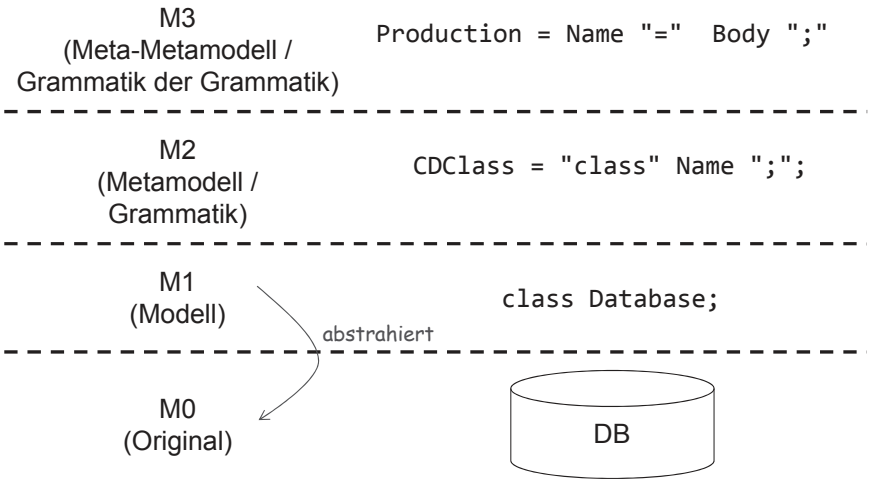


Abbildung 2.2: Darstellung der verschiedenen Modellebenen. Die Ebene M0 zeigt dabei ein Original, welches durch das Modell der Ebene M1 abstrahiert wird. Auf Ebene M2 ist das Metamodell, also die Grammatik, der Sprache dargestellt, welches die Syntax des Modells der M1 Ebene definiert und das Modell erst ermöglicht. Auf Ebene M3 ist das Meta-Metamodell, also die Grammatik der Grammatik, dargestellt, welches die Syntax der Grammatik auf M2 definiert.

zeit dar. Sie modellieren die im System vorhandenen Instanzen der im Klassendiagramm modellierten Klassen. Ein Objektdiagramm wird dabei als konform zu einem Klassendiagramm bezeichnet, wenn es nicht gegen die statische Struktur des Klassendiagramms verstößt. Allerdings kann ein Objektdiagramm auch ohne ein zugehöriges Klassendiagramm verwendet werden, da beide Sprachen keine formale Beziehung untereinander haben. In Abschnitt 2.3 werden Klassen- und Objektdiagramme detaillierter vorgestellt. Aktivitätsdiagramme modellieren Aktivitäten und Transitionen zwischen verschiedenen Aktivitäten. Zudem können Entscheidungen modelliert werden. Aktivitätsdiagramme dienen dazu, Verhalten detailliert zu spezifizieren. Sie werden im Rahmen dieser Arbeit nur dediziert in Kapitel 10 zur Illustration eingesetzt.

Darüber hinaus definiert die UML die *Object Constraint Language (OCL)*. Die OCL kann verwendet werden, um inhaltliche Bedingungen an Instanzen eines Klassendiagramms, die über die Struktur hinausgehen, zu formulieren.

Modellierungssprachen können auf zwei verschiedene Arten definiert werden: durch Angabe eines Metamodells [Kü05] oder durch Angabe einer Grammatik [GKR+08]. Im Rahmen dieser Arbeit werden Sprachen mit Hilfe einer Grammatik, wie sie in Abschnitt 2.2 vorgestellt wird, definiert. Es werden typischerweise verschiedene Modellebenen unterschieden [BCW12]: M0, M1, M2, M3.

Abbildung 2.2 zeigt die unterschiedlichen Ebenen. Die Ebene M0 bezeichnet das Original, welches der Definition eines Modells aus [Sta73] folgend abstrahierend im Modell dargestellt wird. In Abbildung 2.2 ist das Original eine Datenbank. M1 bezeichnet die

nächsthöhere Ebene, nämlich das konkrete Modell selbst. Dabei stellt das Original eine Instanz des Modells dar. Das Modell der Datenbank ist die Klasse `Database`. Die nächste Ebene ist die Ebene M2. Sie bezeichnet die Grammatik der Modellierungssprache. In Abbildung 2.2 wird der Teil der Grammatik gezeigt, der es ermöglicht, das Modell in seiner dargestellten Form aufzuschreiben. Das Modell stellt also eine Instanz der Grammatik einer Modellierungssprache dar. Die Grammatik selbst ist aber ebenfalls eine Modellierungssprache zur Modellierung von Grammatiken. Die Sprache der Grammatik wird durch die Grammatik der Grammatik auf der Ebene M3 definiert. Die UML basiert auf einem Metamodell und einem Meta-Metamodell, welches in [OMG15c] dargestellt ist. Im Rahmen dieser Arbeit werden die Begriffe Grammatik und Metamodell synonym verwendet, auch wenn sich das Metamodell typischerweise auf die abstrakte Syntax einer Sprache bezieht. Details dazu werden in Abschnitt 2.2 dargestellt. Gleichzeitig werden im Rahmen dieser Arbeit nur die Ebenen M1 und M2 betrachtet und es wird ein grammatikbasierter Ansatz zur Definition von Modellierungssprachen verwendet.

Neben MDD werden auch Techniken der generativen Programmierung eingesetzt, welche sich wie folgt definieren lässt:

> *"Generative Programming is about automating the manufacture of intermediate and end-products [...]. This requires modeling product families, [...] specifying the mapping from product specifications to concrete assemblies of implementation components, and implementing this mapping using generators. The products that we want to automatically generate range from classes or procedures to whole subsystems or systems [CE00]."*

Dies bedeutet, dass bei der generativen Programmierung, wie sie typischerweise in MDD zum Einsatz kommt, Modelle mit Hilfe von Codegeneratoren in ausführbaren Code überführt werden. Dabei können einzelne Teile oder ganze Systeme entstehen. Im Rahmen dieser Arbeit wird ein Großteil des Quellcodes von Enterprise Applikationen generiert und um handgeschriebenen Code erweitert [GHK+15a, GHK+15b]. In den Kapiteln 4, 5 und 6 werden die Modellierungssprachen und Modelle, in den Kapiteln 7, 8 und 9 die Generatoren und das Mapping detailliert vorgestellt.

Generierung stellt sich als Spezialfall einer Transformation [Wei12] dar und erzeugt dabei ein neues Modell oder ein ausführbares Programm. Dabei lassen sich im Wesentlichen *Model-to-Model (M2M)* und *Model-to-Text (M2T)* Transformationen unterscheiden. M2T Transformationen werden auch als *Model-to-Code* Transformationen bezeichnet. Allerdings stellt dies eine Einschränkung auf Code dar und umfasst nicht die Generierung von Konfigurationsdateien oder beliebiger Textfragmente. Ein guter Überblick über Eigenschaften von Transformationen und eine detaillierte Klassifizierung wird in [CH03, CH06] gegeben. Diese Übersicht klassifiziert unterschiedliche Transformationsansätze nach ihren jeweiligen Features. Innerhalb der M2T Transformationen existieren visitorbasierte und templatebasierte Ansätze [CH03], die allerdings nicht disjunkt sind. Visitorbasierte Ansätze verwenden Visitoren [GHJV95, HMSNRW16], um den abstrakten Syntaxbaum eines Modells zu traversieren und je nach besuchtem Knoten des abstrakten Syntaxbaums Text zu generieren. Templatebasierte Ansätze hingegen erhalten einen Knoten

als Eingabe und führen ein Template zur Generierung von Text aus. Diese Templates sind dabei ebenfalls in einer Templatesprache ausgedrückt. Es existiert eine Vielzahl verfügbarer Templatesprachen und Templateengines, wobei sich diese in ihrer Funktion und der GPL, in der sie umgesetzt sind, unterscheiden. Zwei Vertreter stellen die Templateengines Velocity [GC03, Vel15] und Freemarker [Fre13, For13], deren Erweiterung und Verwendung in Abschnitt 2.2 genauer erläutert wird, dar.

Neben den M2T Ansätzen existieren innerhalb der M2M Ansätze ebenfalls verschiedene Arten der Transformation. Bekannte Ansätze sind dabei relationale Ansätze, die mit Hilfe einer Transformationsrelation zwischen dem abstrakten Syntaxbaum der Quelle und dem abstrakten Syntaxbaum des Ziels das neue Modell erstellen. Einen bekannten Ansatz stellt hier Query View Transformation (QVT) [Kur08] dar. Weiterhin gibt es graphbasierte Ansätze, die mit Hilfe von Mustern zur Teilgraphersetzung arbeiten. Bekannte Umsetzungen sind dabei AGG [Tae03], PROGRES [Sch90], VIATRA3 [VB07, BDH+15], eMoflon [ALPS11] und GReAT [AKS03]. Weitere Transformationssprachen, die einen hybriden Ansatz darstellen, sind die weitverbreitete *Atlas Transformation Language (ATL)* [JAB+06] und *Extensible Stylesheet Language Transformations (XSLT)* [Kay00], die eine Transformationssprache für die *Extensible Markup Language (XML)* [RS01, XML06] darstellen. Ein weiterer Ansatz wird in [RW11, Wei12] verfolgt. Hier wird eine Methodik zur Erstellung von Transformationssprachen in konkreter Syntax definiert. Während andere Transformationssprachen meist auf der Ebene der abstrakten Syntax arbeiten und Transformationen auf dieser Ebene definiert werden, reicht es in dem Ansatz [RW11, Wei12] aus, die konkrete Syntax einer Sprache zu kennen. Eine Erweiterung des Ansatzes wird in [HRW15] gegeben. Eine Anwendung auf Komponenten und Konnektor Architekturen wird in [HHRW15] gezeigt.

Im Rahmen dieser Arbeit wird ein templatebasierter M2T Ansatz auf Basis einer erweiterten Freemarker Templateengine, die in Abschnitt 2.2 präsentiert wird, verwendet. Die Wahl der Templateengine resultiert aus der Wahl der *Language Workbench* MontiCore und den dort verfügbaren hochintegrierten Prozessen und Codegenerierungsmechanismen, die in Abschnitt 2.2 erläutert werden. Eine allgemeine Einführung zu Codegeneratoren ist in [CE00] gegeben. Im Rahmen dieser Arbeit wird der Begriff des Generierens oder Erzeugens von Quellcode anstelle des Begriffs der Transformation in Quellcode verwendet. Zudem wird im Rahmen dieser Arbeit in Abschnitt 6.2 eine Delta Sprache [HHK+13, HHK+15] zur Modellierung von Evolution, die eine reduzierte und auf den Anwendungsfall der Weiterentwicklung speziell zugeschnittene Form einer Transformationssprache ist, dargestellt.

In diesem Abschnitt wurde eine grundlegende Einordnung dieser Arbeit in das Gebiet des MDD gezeigt. Dazu wurden wichtige Definitionen präsentiert und gängige Praktiken und Technologien kurz vorgestellt. Im nächsten Abschnitt 2.1 wird noch einmal spezifisch auf DSLs eingegangen. Dazu werden sie genauer vorgestellt und ihr Nutzen, ihre Qualitätsmerkmale, aber auch ihre Kosten tiefergehend diskutiert.

2.1 Domänenspezifische Sprachen

Nachdem zuvor die grundlegenden Begriffe eingeführt wurden, wird in diesem Abschnitt tiefergehend auf DSLs eingegangen. Eine DSL wird in [Fow10] als

> *"a computer programming language of limited expressiveness focused on a particular domain"*

definiert. Dies zeigt zwei Charakteristika einer DSL. Zum einen stellt eine DSL eine Programmier- und Modellierungssprache dar, zum anderen besitzt sie meist eine verringerte Ausdrucksfähigkeit, die spezifisch auf eine Domäne zugeschnitten ist. In [Fow10] wird zwischen internen und externen DSLs unterschieden. Dabei ist eine externe DSL eine eigenständige Sprache in einem eigenständigen Artefakt. Eine interne DSL hingegen ist eine in einer GPL verwendete Sprache. Dabei kann die Sprache vollständig oder in Form eines Application Programming Interface (API) als eine interne DSL innerhalb der Sprache realisiert sein. In [CE00] werden zwei andere Arten von DSLs beschrieben: *fixed, seperate DSL* und *embedded DSL*. Dabei entspricht die erste Variante einer externen DSL, wohingegen bei der zweiten eine ganze Sprache oder Teile einer Sprache in eine andere eingebettet werden. Zusätzlich identifiziert [CE00] *modularly composable DSLs*, die in [VBD+13] als *modulare Sprachen* beschrieben werden. Diese DSLs können beliebig zusammen verwendet werden, so dass der Grad der Wiederverwendung deutlich erhöht werden kann. Im Rahmen dieser Arbeit werden mit MontiCore, das in Abschnitt 2.2 vorgestellt wird, externe DSLs erstellt, die modular komponierbar sind. Dabei wurde MontiCore gewählt, da die im Rahmen dieser Arbeit entwickelten Sprachen auf der ebenfalls mit MontiCore entwickelten UML/P, die in Abschnitt 2.3 präsentiert wird, aufbauen und gleichzeitig die von MontiCore bereitgestellten Mechanismen zur Sprachaggregation, -einbettung und -vererbung verwendet werden. Dadurch ermöglicht MontiCore eine hochgradig kompositionale, agile Entwicklung neuer Sprachen unter starker Wiederverwendung bereits existierender Sprachen. MontiCore ist eine *Language Workbench*, die in Abschnitt 2.2 vorgestellt wird. Im Rahmen dieser Arbeit wird zwischen technischen DSLs und DSLs der Anwendungsdomäne, wie in [VBD+13] vorgestellt, nicht unterschieden. Als Beispiele bekannter DSLs werden in [Fow10] verschiedene DSLs genannt: GraphViz [EGK+02], die Hibernate Query Language (HQL) [ML05] und die Extensible Application Markup Language (XAML) [Mac06], aber auch die Structured Query Language (SQL) [Mol06] und die Hypertext Markup Language (HTML) [SS11].

Neben diesen werden DSLs in weiteren Domänen, wie zur Modellierung von Gebäuden und technischen Anlagen [Pin14], aber auch zur Modellierung von Robotern und deren Verhalten [RRW14], eingesetzt. Die Menge der möglichen Beispiele ist nahezu unerschöpflich, aber bereits an der hier vorgenommenen Auswahl lässt sich erkennen, dass diese Sprachen unterschiedliche Domänen abdecken.

In [Fow10] werden verschiedene Vor- und Nachteile von DSLs diskutiert. Generell wird davon ausgegangen, dass eine DSL die Entwicklungsproduktivität erhöht, da es möglich ist, Domänenkonzepte deutlich effizienter auszudrücken. Gleichzeitig kann ein solches Modell deutlich leichter gelesen und gewartet werden. Hinzu kommt, dass die

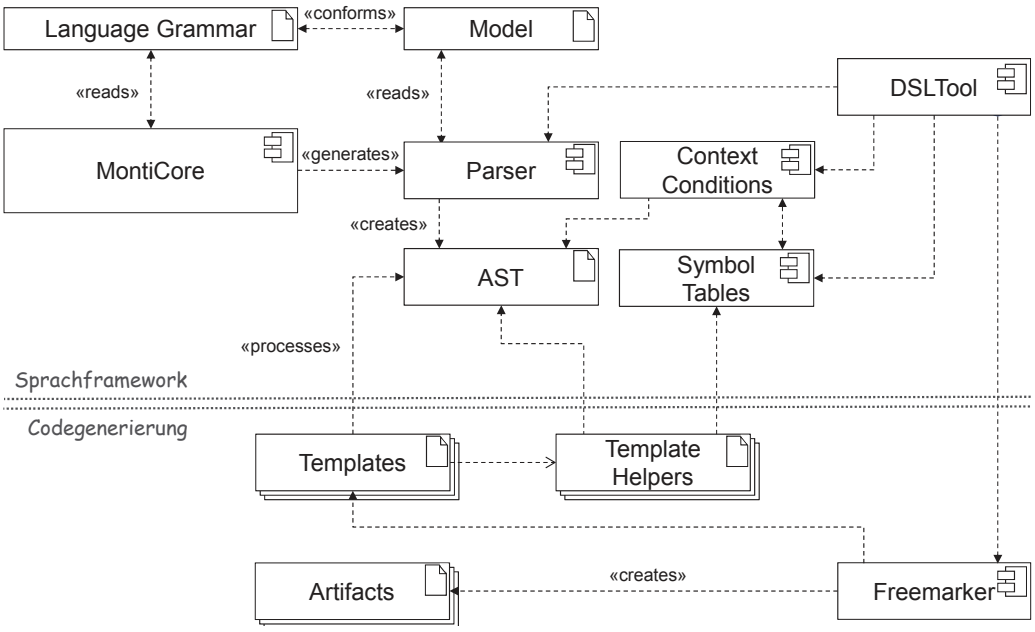


Abbildung 2.3: Darstellung der Komponenten und der Funktionsweise der Language Workbench MontiCore. Abbildung aus [RRW13].

Modelle als Eingabe für einen Codegenerator, der technisches Wissen hinzufügen kann, verwendet werden können. Demgegenüber steht der Aufwand des Erstellens einer DSL [VBD+13]. Die Sprache muss im Rahmen des Sprachdesigns entworfen, die Semantik der Sprache definiert und die Generatoren umgesetzt werden. Der Aufwand des Sprachdesigns lässt sich durch geeignete Guidelines [KKP+09, VBD+13] reduzieren. Neben den Entwicklungskosten sind auch die Kosten zur Einführung und Integration in bestehende Entwicklungsprozesse zu nennen.

Eine DSL ermöglicht es aber auch, mit Domänenexperten effizienter zu kommunizieren und die Zieltechnologie einfacher auszutauschen, da die Sprache selbst davon typischerweise abstrahiert. Demgegenüber steht wieder, dass eine Sprache von den Anwendern erlernt werden muss, was zu einem gewissen Einarbeitungsaufwand führt. Ein weiterer Nachteil ist, dass neue Nutzer innerhalb eines Unternehmens neu angelernt werden müssen und nicht bereits im Rahmen ihrer Ausbildung damit vertraut werden.

Alles in allem überwiegen die Vorteile einer DSL in dem Moment, in dem häufig repetitive Aufgabe, die durch eine geeignete Abstraktion vereinfacht werden können, erfüllt werden müssen, so dass der initiale Aufwand vom Nutzen der häufigen Verwendung überwogen wird. Gleichzeitig überwiegt der Nutzen auch bei häufigen Technologiewechseln. Zwar benötigt dies eine Anpassung der Codegeneratoren, ermöglicht aber gleichzeitig, dass alle entwickelten Systeme ohne Modelländerung migriert werden können. Die Verwendung und Entwicklung eines Codegenerators erhöht dabei zunächst die Komplexität, ermöglicht es aber, dass im Wesentlichen der Codegenerator und nicht das Generat selbst gewartet und getestet werden muss.

Eine gute Übersicht über die unterschiedlichen Vor- und Nachteile der Verwendung einer DSL und eine Entscheidungshilfe, ob die Umsetzung einer DSL sinnvoll ist, wird in [MHS05] gegeben. Zudem wird dort eine Vielzahl existierender DSLs für unterschiedliche Anwendungszwecke gezeigt und wie der Aufwand zur Erstellung einer DSLs reduziert werden kann. Zudem wird in [DSM15] eine Menge an Fallstudien, die die Produktivitätssteigerung bei Einsatz einer DSL messen, gezeigt. Es wird dabei davon ausgegangen, dass die Produktivität um den Faktor fünf bis zehn gesteigert wird.

In [Wil03] wird empfohlen, DSLs für ca. 80 Prozent des Systems einzusetzen. Dies resultiert aus der Abstraktion und der Notwendigkeit der hochspezifischen Anpassung eines Systems. Die hochspezifischen Teile können dabei nur schwer mit Hilfe einer DSL modelliert werden, so dass diese manuell umgesetzt werden sollten.

In [HR04] werden die Begriffe *konkrete Syntax*, die beschreibt, wie eine Sprache textuell oder graphisch aussieht, und *abstrakte Syntax*, die die interne Repräsentation der Sprache darstellt [Kra10], unterschieden. Dies haben alle Formen von DSLs gemein. Hinzu kommen Kontextbedingungen und die Semantik der Sprache. Im nächsten Abschnitt wird die Language Workbench MontiCore vorgestellt, die es erlaubt, die genannten Teile, konkrete und abstrakte Syntax sowie Kontextbedingungen einer DSL zu definieren und aus diesen durch die Einbindung von Codegeneratoren Code zu generieren. Neben MontiCore existieren andere Language Workbenches, wie XText [EB10], Spoofax [KV10] oder MetaEdit+ [KLR96]. Weitere Language Workbenches und eine gute Übersicht sind in [ESV+13] gegeben. Im Rahmen dieser Arbeit wird, wie bereits zuvor begründet, die Language Workbench MontiCore, die im folgenden Abschnitt 2.2 vorgestellt wird, verwendet.

2.2 Die Language Workbench MontiCore

MontiCore [GKR+08, Kra10] stellt eine Language Workbench zur Erstellung domänenspezifischer Sprachen dar und bietet darüber hinaus Unterstützung bei der Prüfung von Kontextbedingungen, Einbindung von Transformationen und der Codegenerierung. Im Rahmen dieser Arbeit wird MontiCore in der Version 3 verwendet. MontiCore verwendet textuelle Grammatiken, welche zur Definition einer domänenspezifischen Sprache verwendet werden. Sie werden in Abschnitt 2.2.1 vorgestellt. MontiCore setzt ein eigenständiges, Erweiterte Backus-Naur Form (EBNF) ähnliches Grammatikformat ein, welches die Definition kontextfreier Sprachen ermöglicht. Innerhalb der Grammatik sind sowohl die Definition der konkreten als auch der abstrakten Syntax der entstehenden domänenspezifischen Sprache vereint. Auf Basis dieser Grammatiken generiert MontiCore, wie in Abschnitt 2.2.2 präsentiert wird, Lexer, Parser, die Klassen des abstrakten Syntaxbaums und weitere Infrastrukturen zur Verwendung und Integration domänenspezifischer Sprachen im Entwicklungsprozess.

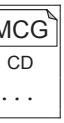
Eine Übersicht über die Bestandteile von MontiCore ist in Abbildung 2.3 dargestellt. Die primären Artefakte stellen Grammatiken, welche zunächst in Abschnitt 2.2.1 vorgestellt werden, dar. Im Anschluss daran werden die Abstract Syntax Tree (AST)-Klassengenerierung in Abschnitt 2.2.2, Kontextbedingungen in Abschnitt 2.2.4 und die Code-

generierungsinfrastruktur soweit vorgestellt, wie sie für das weitere Verständnis dieser Arbeit benötigt werden.

2.2.1 MontiCore Grammatiken

MontiCore verwendet Grammatiken zur Sprachdefinition. Listing 2.4 zeigt eine MontiCore Grammatik ohne ihre enthaltenen Elemente. Die hier vorgestellte Grammatik stellt eine stark gekürzte und nicht vollständige Variante einer textuellen UML/P Klassendiagrammsprache, welche in Abschnitt 2.3 vorgestellt wird, dar. Sie wurde an dieser Stelle aber sowohl gekürzt als auch vereinfacht, um die benötigten MontiCore Konzepte der Sprachentwicklung in einer einfachen und prägnanten Art einzuführen. Eine umfassendere Version und Referenzen zu vollständigen Versionen werden in Abschnitt 2.3 gegeben. In MontiCore beginnt eine Grammatik mit dem Schlüsselwort `grammar` gefolgt von einem Namen.

```
1 grammar CD extends mc.umlp.common.Common {
2   // weitere Elemente der Grammatik
3 }
```



Listing 2.4: Auszug des generellen Aufbaus einer MontiCore Grammatik ohne enthaltene Elemente.

Das folgende Schlüsselwort `extends` und der anschließende Name wird für das Konzept der Sprachvererbung, welches später erläutert wird, verwendet. Danach folgen eine öffnende geschweifte Klammer, weitere Elemente der Grammatik und eine schließende geschweifte Klammer. Die Elemente einer Grammatik können im Wesentlichen die folgenden vier sein:

- Optionen
- Token
- Produktionen
- Konzepte

Dabei dienen Optionen der Konfiguration des Parsergenerators und ermöglichen weitergehende Einstellungen wie Parser oder Lexer Lookaheads oder auch Kommentarzeichen. Konzepte werden für unterschiedliche Aufgaben verwendet. Eine ihrer Aufgaben ist die Konfiguration der Editorgenerierung, die von MontiCore angeboten wird. Dabei ist es möglich, einen textuellen Eclipse-Editor mit Syntaxhighlighting, Autovervollständigung und Hyperlinking zu generieren. Identifier werden im Rahmen dieser Arbeit nicht weiter verwendet. Eine detaillierte Erklärung der Optionen, Identifier und Konzepte ist in [GKR+06, Kra10] gegeben. Zum besseren Verständnis dieser Arbeit wird hier genauer auf Produktionsregeln als Elemente der Grammatik eingegangen.

```
1  Definition =                                                    MCG
2    "classdiagram" Name                                           CD
3      "{"                                                         ...
4         (Class | Interface)*
5      "}";
```

Listing 2.5: Produktionsregel des Nichtterminals `Definition` in der Syntax einer MontiCore Grammatik.

Listing 2.5 zeigt eine MontiCore Produktionsregel. Produktionsregeln bestehen aus einem Regelkopf und einem Rumpf. MontiCore verwendet kontextfreie Grammatiken, so dass der Regelkopf immer aus einem Nichtterminal, das durch den Rumpf definiert wird, besteht. Der Rumpf besteht aus Terminalen, welche atomare Elemente der Grammatik darstellen und Nichtterminalen, welche einen Verweis auf weitere Produktionsregeln oder Tokens, die während des Parsens als Ersetzungsschritte verwendet werden, darstellen.

In Listing 2.5 wird die Produktionsregel des Nichtterminals `Definition` gezeigt. Diese Produktionsregel soll die Startregel für die hier vorgestellte Sprache zur Modellierung von Klassendiagrammen darstellen. Der Regelkopf ist dabei die linke Seite, der Regelrumpf die rechte des Gleichheitszeichens. Jede `Definition` beginnt mit dem Terminal `classdiagram` als Schlüsselwort der konkreten Syntax. Terminale stehen in Anführungszeichen und können Zeichenketten, Schlüsselwörter, Konstanten oder Konstantengruppen sein. Danach folgt in der abstrakten Syntax ein Name und eine öffnende geschweifte Klammer in der konkreten Syntax. An dieser Stelle können entweder Klassen oder Interfaces beliebig oft als Teil der abstrakten Syntax stehen. Die Alternative, ob Klassen oder Interfaces, wird dabei durch "|" und die Kardinalität durch den "∗" definiert. Generell können Nichtterminale Kardinalitäten besitzen, die als "∗", "+", "?" dargestellt werden. Dabei steht "∗" für beliebig viele, "+" für beliebig viele, aber mindestens eins und "?" für eins oder keins. Zudem können Klammern zur Strukturierung verwendet werden, um Gruppierungen zu bilden.

```
1  token Name =                                                    MCG
2    ( 'a'..'z' | 'A'..'Z' | '_' | '$' )                          CD
3    ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*;             ...
```

Listing 2.6: Definition des Tokens `Name` in der Syntax einer MontiCore Grammatik.

Neben Produktionsregeln können auch Tokendefinitionen als Nichtterminale verwendet werden. Listing 2.6 zeigt die Definition des Tokens `Name`. Der Token ist dabei analog zu einem erlaubten Namen in Java definiert.

Darüber hinaus bietet MontiCore die Möglichkeit, Variablennamen vor einem Nichtterminal zu spezifizieren. Dies führt dazu, dass innerhalb der abstrakten Syntax an dieser Stelle ebenfalls eine Variable mit diesem Namen, der ohne explizite Angabe automatisch

```
1  Class =                                                          MCG
2    "class" Name                                                   CD
3    ( "extends"                                                    ...
4         superclass:RefName ("," superclass:RefName)* )?
5    ( "implements"
6         interfaces:RefName ("," interfaces:RefName)* )?
7    ( "{" ClassElement* "}" | ";" );
```

Listing 2.7: Produktionsregel des Nichtterminals `Class` in der Syntax einer MontiCore Grammatik. Gezeigt ist die Möglichkeit zur Definition von Listen mit einem Trennzeichen sowie die Möglichkeit zur Definition von Kardinalitäten.

abgeleitet wird, generiert wird. Zudem können auch Listenproduktionsregeln mit einem Trennzeichen angegeben werden. Listing 2.7 zeigt die Produktionsregel für eine Klasse, welche mit dem Schlüsselwort `class` und einem Namen beginnt. Danach folgen dann die Liste der Superklassen, die die zu modellierende Klasse erweitert, und die Liste der Interfaces, die von dieser Klasse implementiert werden. Durch das dort abgebildete Konstrukt mit Nichtterminalen, die den gleichen Variablennamen haben, wird dies ermöglicht. Innerhalb einer Klasse können dann beliebig viele `ClassElement` Elemente definiert werden. Das Nichtterminal stellt ein weiteres Nichtterminal dar, welches aber eine spezielle Rolle innerhalb der Grammatik hat. MontiCore erlaubt die Definition von Interface-Nichtterminalen. Dabei können solche Nichtterminale wie normale Nichtterminale verwendet werden, ohne dass ihre konkrete Produktionsregel bereits bekannt ist. Andere Nichtterminale können dann das Interface-Nichtterminal implementieren und können somit überall dort verwendet werden, wo das Interface-Nichtterminal verwendet wurde.

```
1  interface ClassElement;                                          MCG
2                                                                   CD
3  Attribute implements ClassElement =                              ...
4    Modifier? Type Name ("=" Value)? ";";
5
6  Method implements ClassElement =
7    Modifier? ReturnType Name "(" ParameterList? ")"
8    ( "{" Body "}" | ";" );
```

Listing 2.8: Definition des Interface-Nichtterminals `ClassElement` und dessen Implementierungen durch die `Attribute` und `Method` Produktionen.

Listing 2.8 zeigt die Definition des Interface-Nichtterminals `ClassElement`. Zudem zeigt Listing 2.8 zwei Produktionsregeln, die das Interface implementieren: `Attribute` und `Method`. Dies hat zur Folge, dass Attribute und Methoden innerhalb einer Klasse in einem Klassendiagramm modelliert werden können. Attribute haben einen optionalen

Modifikator, einen Typ, einen Namen und einen optionalen Wert. Beendet wird eine Attributdefinition mit einem Semikolon. Methoden haben einen optionalen Modifikator, einen Rückgabetypen, einen Namen, eine optionale Parameterliste und eine Implementierung oder ein schließendes Semikolon. Die hier verwendeten Nichtterminale, mit Ausnahme des Nichtterminals `Body`, welches später aufgegriffen wird, werden hier nicht weiter präsentiert.

Neben der Möglichkeit Interface-Nichtterminale zu implementieren, existiert auch die Möglichkeit der Vererbung zwischen Produktionsregeln. Dabei wird das Schlüsselwort `extends` analog zum Schlüsselwort `implements` in Grammatiken verwendet. Der semantische Unterschied dieser beiden Möglichkeiten liegt darin, dass eine `implements` Beziehung sich immer auf ein Interface-Nichtterminal bezieht, wohingegen sich eine `extends` Beziehung immer auf eine konkret definierte Produktionsregel analog zur Semantik der Verwendung beider Konzepte in gängigen GPLs, wie Java, bezieht.

Die bisher vorgestellten Konzepte zeigen nur einen Auszug der Möglichkeiten des Sprachentwurfs mit MontiCore. Allerdings sind diese Konzepte im Rahmen dieser Arbeit ausreichend und werden bei Bedarf minimal an den geeigneten Stellen ergänzt. So existieren beispielsweise Konstrukte, die die Sprachkomposition auf unterschiedliche Arten ermöglichen. Diese Konzepte werden dann in Abschnitt 2.2.3 vorgestellt. Nachdem die Grammatik vorgestellt wurde, werden die weiteren Elemente aus Abbildung 2.3 beginnend mit der Generierung der AST-Klassen, vorgestellt.

2.2.2 Generierung der AST-Klassen

Auf Basis der Grammatik wird mit Hilfe des Parser Generators ANTLR [Par07] der Parser generiert und zur Verfügung gestellt. Zusätzlich zum generierten Parser werden die Klassen des AST generiert. Die AST-Klassen werden dabei schematisch generiert. Für jedes normale Nichtterminal wird eine AST-Klasse generiert. Die Attribute werden auf Basis der rechten Seite der Produktion mitgeneriert. Für die Kardinalitäten "+" und "$*$" werden zudem spezielle Listenklassen generiert. Für jedes Interface-Nichtterminal wird ein Interface generiert. Die generierten Klassen heißen dabei so wie der Name des Nichtterminals mit dem Präfix "AST". Die Listenklassen erhalten zudem das Suffix "List".

Zusätzlich kann bereits innerhalb der Grammatik Einfluss auf die abstrakte Syntax genommen werden. Dazu werden die Schlüsselwörter `astextends`, `astimplements` oder `ast` verwendet. Diese arbeiten auf Ebene der abstrakten Syntax und bestimmen, ob eine AST-Klasse von einer anderen Klasse erbt oder ein Interface implementiert. Mit Hilfe des Schlüsselworts `ast` lassen sich zudem Codeteile in der Grammatik definieren, die in die generierte AST-Klasse übernommen werden. Darüber hinaus gibt es die Möglichkeit eine Produktionsregel mit "/" als Prototyp zu markieren, was dazu führt, dass der Name der generierten AST-Klasse sich ändert. Die reguläre, erwartete AST-Klasse kann dann vom Sprachentwickler manuell geschrieben werden und muss von der Prototypklasse erben. Dadurch kann handgeschriebener Code hinzugefügt werden. Der von MontiCore generierte Parser instanziiert diese AST-Klassen zur weiteren Verarbeitung auf Basis der von ihm geparsten Modelle. Genauere Informationen zu den hier nur kurz beschriebenen Möglichkeiten lassen sich in [GKR$^+$06, Kra10] finden.

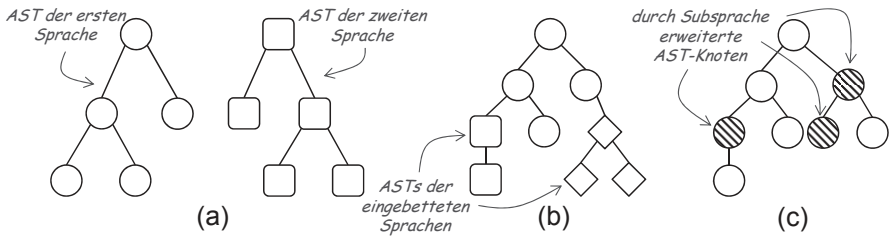


Abbildung 2.9: Schematische Darstellung der Mechanismen der Sprachkomposition. Dabei zeigt (a) Sprachkomposition, (b) Spracheinbettung und (c) Sprachvererbung. Abbildung abgeändert aus [LNPR+13, HLMSN+15a, HLMSN+15b].

2.2.3 Sprachintegrationsmechanismen in MontiCore

Die bisher vorgestellten Möglichkeiten beziehen sich auf die Erstellung einer DSL und die Modellierung eines Modells in einer Sprache. Darüber hinaus bietet MontiCore mit Hilfe verschiedener Mechanismen [Völ11] die Möglichkeit zur Sprachintegration. Dies erlaubt es, unterschiedliche Sprachen zur Modellierung verschiedener Sachverhalte zu verwenden. Mit Hilfe von Referenzen zwischen Elementen und Konzepten der jeweiligen Modelle können diese adressiert werden. Somit kann die jeweils am besten geeignete Modellierungssprache, um Teilbereiche zu modellieren, eingesetzt werden. Details der verschiedenen Mechanismen sind in [LNPR+13, HLMSN+15b, HLMSN+15a] erläutert und werden hier lediglich kurz vorgestellt. MontiCore bietet drei verschiedene Möglichkeiten zur Integration verschiedener Modellierungssprachen, welche in Abbildung 2.9 dargestellt sind und nachfolgend erläutert werden.

Sprachaggregation

Die erste Möglichkeit ist *Sprachaggregation*, welche es erlaubt, Elemente zwischen zwei verschiedenen Modellen verschiedener Sprachen zu referenzieren. Auf Ebene des ASTs existieren dabei für jede Sprache eigene AST-Klassen und jedes Modell stellt, wie in Abbildung 2.9(a) dargestellt, eine Instanz der jeweiligen Klassen dar. Die Referenzierung erfolgt über Namen. Ein referenzierter Knoten des ASTs wird von einem anderen AST dadurch referenziert, dass sein Name innerhalb des referenzierenden ASTs als Name verwendet wird. Zum späteren Auflösen und weiteren Verwenden des Knotens wird das Symbol der Symboltabelle des referenzierten Knotens verwendet.

Spracheinbettung

Die zweite Möglichkeit ist *Spracheinbettung*. Diese Möglichkeit erlaubt es Sprachen, ineinander einzubetten. Auf die Produktionsregel einer Methode in Listing 2.8 zurückblickend lassen sich dadurch beispielsweise verschiedene Sprachen zur Definition der Methodenimplementierung verwenden. Dazu muss das entsprechende Nichtterminal mit Hilfe des Schlüsselworts `external` als extern markiert werden.

```
1  external Body;                                                    MCG
2                                                                    CD
3                                                                    ...
```

Listing 2.10: Definition der externen Produktion `Body` in der Syntax einer MontiCore Grammatik.

Listing 2.10 zeigt die Markierung des Nichtterminals `Body` als ein externes Nichtterminal und definiert somit einen Erweiterungspunkt in der Grammatik. An dieser Stelle können Nichtterminale anderer Sprachen eingebettet werden. Die Verbindung erfolgt über ein externes Artefakt, welches hier nicht genauer vorgestellt wird. Auf Ebene der ASTs führt dies zu der Situation in Abbildung 2.9(b): Der AST der einbettenden Sprache enthält Knoten des ASTs der eingebetteten Sprachen. Eine Referenzierung zwischen den Elementen erfolgt ebenfalls über Namen und kann mit Hilfe der Symboltabelle oder direkt auf dem AST erfolgen.

Sprachvererbung

Die dritte Möglichkeit ist *Sprachvererbung*. Diese Möglichkeit erlaubt es, Produktionsregeln einer übergeordneten Sprache wiederzuverwenden, zu überschreiben oder zu verfeinern. Dies wird, wie in Listing 2.4 dargestellt, durch das Schlüsselwort `extends` signalisiert. Dadurch stehen der Klassendiagrammgrammatik alle Produktionsregeln der übergeordneten Sprache `Common` zur Verfügung und können in der Klassendiagrammsprache verwendet werden. Auf Ebene des AST führt dies dazu, dass Knoten der übergeordneten Sprache im AST enthalten sind.

Es sei an dieser Stelle angemerkt, dass die zuvor beschriebenen Möglichkeiten zur Definition von Interface-Nichtterminalen und Vererbung auf Ebene der Produktionsregeln auch über mehrere Grammatiken hinweg einsetzbar sind und somit ebenfalls mit Sprachvererbung funktionieren. Es ist also möglich, ein Interface-Nichtterminal in einer übergeordneten Sprache zu definieren und dieses in der Subsprache zu implementieren. Genauso ist es möglich, dass eine Produktionsregel der Subsprache von einer Produktionsregel der übergeordneten Sprache erbt. Im Rahmen dieser Arbeit stellt die MontiEE Sprachfamilie eine Sprachaggregation, in der sich Elemente der jeweiligen Modelle gegenseitig referenzieren, dar. Einzelne Sprachen, wie die Tagdefinitions-, die Tagschema- und die Deltasprache, die in den Kapiteln 4 und 6 präsentiert werden, verwenden Sprachvererbung zur Bildung klassendiagrammspezifischer Varianten. Spracheinbettung wird bei der Modellierung von Queries innerhalb einer Tagdefinition, wie in Abschnitt 7.2 präsentiert, verwendet. Diese Konzepte basieren stark auf den von MontiCore zur Verfügung gestellten Mechanismen und bedingen so dessen Nutzung.

2.2.4 Kontextbedingungen in MontiCore

Neben der Definition der Grammatiken und der Sprachintegration bietet MontiCore ebenfalls die Möglichkeiten Symboltabellen für Sprachen zu erstellen und Kontextbedin-

gungen zu prüfen [Sch12, Völ11]. Dazu müssen für jede Sprache verschiedene Komponenten definiert werden, die den Aufbau und die Verwendung der Symboltabelle ermöglichen. Die Symboltabelle beschreibt dabei die Schnittstelle des Modells, also die Informationen, die nach außen hin zur Verfügung gestellt werden.

Die Symboltabelle wird vor allem zur Modellkomposition verwendet. Sie ermöglicht dabei das aus der komponentenbasierten Softwareentwicklung bekannte Konzept der Modularität. Jedes Modell ist dabei ein eigenes Modul und kann, solange sich seine Schnittstelle nicht ändert, intern weiterentwickelt und gleichzeitig von anderen Modellen benutzt werden. Zudem ermöglicht die Symboltabelle die Prüfung von Kontextbedingungen auch über Modellgrenzen hinweg. MontiCore bietet ein Framework zur Definition von Kontextbedingungen, die entweder den AST oder die Symboltabelle benutzen [Sch12, Völ11]. Dabei unterscheidet [Sch12] sprachinterne Intra- und Inter-Modell-Bedingungen sowie sprachübergreifende Intra- und Inter-Modell-Bedingungen. Sprachinterne Intra-Modell-Bedingungen sind dabei Kontextbedingungen, die sich auf eine Sprache und ein Modell beschränken. Diese Bedingungen prüfen innerhalb eines Modells die Konsistenz zwischen Elementen der gleichen Sprache und können direkt auf dem aktuellen AST berechnet werden. So können solche Bedingungen beispielsweise die Existenz referenzierter Elemente innerhalb des Modells sichern und werden z.B. bei der Prüfung der Verwendung von Variablennamen verwendet, um sicherzustellen, dass diese vorher definiert wurden.

Sprachinterne Inter-Modell-Bedingungen sind Kontextbedingungen, die innerhalb einer Sprache die Konsistenz zwischen verschiedenen Modellen sicherstellen. Dies unterstützt die Sprachaggregation und die Aggregation mehrerer Modelle. Dabei müssen diese Bedingungen auf Basis der Symboltabelle des aggregierten Modells geprüft werden. Die dritte Art, sprachübergreifende Intra-Modell-Bedingungen, sind Kontextbedingungen innerhalb eines Modells mit mehreren durch das Konzept der Spracheinbettung eingebetteten Sprachen. Diese können entweder über den AST des Modells und den dort vorhandenen verschiedenen AST-Knoten oder aber über die Symbole des eingebetteten Modells geprüft werden. Die vierte Variante, also sprachübergreifende Inter-Modell-Bedingungen, prüfen Beziehungen zwischen zwei Modellen unterschiedlicher Sprachen. Diese müssen gegen die Symboltabelle des anderen Modells geprüft werden.

Bei der Verarbeitung von Modellen, dem Parsen, der Transformation und der anschließenden Codegenerierung verwendet MontiCore den Begriff der Sprachfamilien. Einzelne Sprachen werden in Familien zusammengefasst. Jede Sprache besitzt dabei eigene Kontextbedingungen: sprachinterne Intra- und Inter-Modell-Bedingungen. Die Sprachfamilie besitzt ebenfalls Kontextbedingungen, die sprachübergreifenden Intra- und Inter-Modell-Bedingungen. Bei der Verarbeitung werden die Modelle immer einzeln, der Reihe nach verarbeitet. Die Symboltabelle wird aufgebaut, verschiedene Transformationen werden ausgeführt und die Codegenerierung wird für jedes Modell ausgeführt. Im Rahmen dieser Arbeit werden Kontextbedingungen der geschaffenen Sprachfamilie zur Konsistenzsicherung innerhalb einzelner Modelle oder zwischen verschiedenen Modellen vorgestellt. Darüber hinaus werden Kontextbedingungen, die vor Ausführung der Codegenerierung geprüft werden, definiert. Im weiteren Verlauf werden an unterschiedlichen Stellen eigene Vorverarbeitungsschritte und Transformationen genauer erläutert werden.

Nachdem alle Elemente der Language Workbench aus Abbildung 2.3 erläutert wurden, wird anschließend die von MontiCore angebotene Codegenerierung, um ein genaueres Verständnis der in der Arbeit entwickelten Generatoren zu ermöglichen, erläutert.

MontiCore verwendet zur Codegenerierung die Templateengine `Freemarker` [Fre13, For13]. Freemarker stellt eine M2T Transformation dar, welche den AST, oder Teile des ASTs in Text transformiert. Dabei wird eine Menge von Templates zur Erstellung des gewünschten Texts durchlaufen. Im Rahmen dieser Arbeit wird gültiger Java-Quellcode generiert. Dabei wird MontiCore so konfiguriert, dass es für einen Typ eines AST-Knotens Starttemplates ausführt. Für einen Typ können mehrere Starttemplates registriert werden. Zudem können auch mehrere Typen mit unterschiedlichen Starttemplates registriert werden. Die Templateengine führt für alle AST-Knoten des registrierten Typs die registrierten Starttemplates aus. Innerhalb der Templates wird der Teilgraph des AST traversiert und weitere Subtemplates referenziert und durchlaufen. Freemarker selbst bietet eine Templatesprache an, die bereits verschiedene Direktiven, sogenannte `built-ins`, beinhaltet [Fre13, For13]. So können einfache Schleifenoperationen, Bedingungen und Stringoperationen direkt mit Hilfe der `built-ins` umgesetzt werden. Freemarker erlaubt es aber auch, auf Java-Objekte zuzugreifen und Methoden dieser Objekte auszuführen. Die MontiCore Codegenerierung stellt dabei zwei vordefinierte Java-Objekte zur Verfügung: Den `AST`, der gerade durch Freemarker in Text transformiert wird, und den `TemplateOperator`.

CD

```
1 class Person {
2   String realName;
3 }
```

```
...
```

Listing 2.11: Modell der zuvor eingeführten Grammatik. Modelliert ist eine Klasse `Person` mit einen Attribut `realName`.

Listing 2.11 zeigt einen Ausschnitt aus einem Modell der zuvor eingeführten Klassendiagrammsprache. Es modelliert eine einzelne Klasse `Person`, die ein Attribut `realName` besitzt. Dieser Auszug wird mit Hilfe der MontiCore Codegenerierung in validen Java-Code überführt.

Der Auszug 2.11 modelliert lediglich eine Klasse und ein einzelnes Attribut. Der resultierende Quellcode hingegen soll auch Getter- und Setter-Methoden für das jeweilige Attribut beinhalten. Mit Hilfe einer Menge von Templates wird die Eingabe in den gewünschten Quellcode transformiert.

Listing 2.12 zeigt ein Freemarker Template zur Transformation der modellierten Klasse des Klassendiagramms in eine valide Java-Klasse. Es beginnt mit dem statischen Teil, `public class`, welcher in allen generierten Klassen vorkommt. Es sei angemerkt, dass dieses Template keine inneren Klassen oder Klassen mit anderen Modifikatoren generieren kann. Danach wird auf das Objekt `ast` zugegriffen, welches den entsprechenden Knoten repräsentiert. In Listing 2.7 wurde gezeigt, dass eine modellierte Klasse im Klas-

```
1 public class ${ast.getName()} {                                    FTL
2                                                                     ...
3   ${op.defineValue("elementHelper",
4                    op.instantiate("helpers.ElementHelper(ast)")}
5
6   ${op.includeTemplate("AttributeTemplate.ftl",
7                        elementHelper.getAttributes())}
8 }
```

Listing 2.12: Definition eines Templates in Freemarker Syntax zur Generierung von Java-Code. Das Template verarbeitet die Eingabe aus Listing 2.11.

sendiagramm immer einen Namen hat. Die zugehörige AST-Klasse, die von MontiCore generiert wird, beinhaltet demnach eine Methode, die den Namen `getName` besitzt. Im Template kann auf das Objekt zugegriffen und die angebotene API verwendet werden. Hier handelt es sich um eine zum Verständnis des Konzepts vereinfachte Darstellung. Als nächste Anweisung ist im Template die Möglichkeit der Wertedefinition und der Instanziierung neuer Typen dargestellt. Hierzu bietet der `TemplateOperator`, der innerhalb des Templates durch `op` repräsentiert wird, die Methoden `defineValue` und `instantiate` an. Die `defineValue`-Methode bietet die Möglichkeit, einen beliebigen Wert, dargestellt durch das zweite Argument der Methode, unter einem Schlüssel, dargestellt durch das erste Argument der Methode, zu speichern. Das zweite Argument ist dabei eine Instanz der Klasse `helpers.ElementHelper`, die mit dem Argument `ast` instanziiert wird. Die Instanziierung übernimmt die `instantiate`-Methode des `TemplateOperator`.

Zudem ist gezeigt, dass der zuvor definierte Wert, `elementHelper`, direkt über seinen Schlüssel verwendet werden kann. Dies bietet dem Templateentwickler die Möglichkeit, aus dem Template und seiner angebotenen Funktionalität wieder in die Touring-vollständige GPL zurückzukehren und somit komplexe Berechnungen, die in der Templatesprache nicht möglich wären, auszuführen. In Listing 2.12 berechnet der instanziierte Helper die Menge aller modellierten Attribute einer Klasse. Zur Erinnerung sei angemerkt, dass eine Klasse in der Grammatik eine Menge von `ClassElement` beinhaltet, welche ein Interface-Nichtterminal darstellt und keine Unterscheidung zwischen Attributen oder Methoden bietet. Der Helper, programmiert in der GPL, kann den AST vollständig traversieren und die benötigten Informationen zurückliefern.

Als weitere Methode bietet der `TemplateOperator` die Methode `includeTemplate` an. Diese erlaubt den Aufruf weiterer Subtemplates. Der erste Parameter der Methode ist der vollqualifizierte Name des aufzurufenden Templates und der zweite Parameter ist der AST-Knoten, für den das Subtemplate aufgerufen wird. Hierbei sei angemerkt, dass es durch Methodenüberladung auch möglich ist, eine Liste von AST-Knoten zu übergeben. Dies bedeutet, dass das entsprechende Subtemplate für alle Elemente der Liste einmalig aufgerufen wird.

```
1  private ${ast.getType()} ${ast.getName()};                    FTL
2                                                                 ...
3  public ${ast.getType()} get${ast.getName()}()
4  {
5    return ${ast.getName()};
6  }
7
8  public void set${ast.getName()}
9    (${ast.getType()} ${ast.getName()})
10 {
11   this.${ast.getName()} = ${ast.getName()};
12 }
```

Listing 2.13: Definition eines Templates in Freemarker Syntax zur Generierung von Getter- und Setter-Methoden. Das Template verarbeitet die Eingabe aus Listing 2.11.

Das aufgerufene Subtemplate ist in Listing 2.13 dargestellt. Im Wesentlichen zeigt sich, dass auf die vom entsprechenden AST-Knoten bereitgestellten Methoden zugegriffen werden kann. Es sei allerdings angemerkt, dass der AST-Knoten ein anderer ist als im Template zuvor. Während er im vorherigen Template einer modellierten Klasse entsprach, entspricht er in diesem Template einem Attribut. Durch den Aufruf der `includeTemplate`-Methode wird die AST-Variable der Subtemplates neu belegt und kann somit dort anders verwendet werden. Das in Listing 2.13 gezeigte Template generiert also aus dem modellierten Attribut immer ein privates Attribut und passende Getter- und Setter-Methoden im Quellcode, so dass der Zugriff auf das Attribut ermöglicht wird. Ebenso wäre auch die Verwendung des `TemplateOperator` in diesem Template wieder möglich. Allerdings ändert sich dieses Objekt auch im Subtemplate nicht, so dass auf im übergeordneten Template gesetzte Werte zugegriffen werden kann. Andersherum ist dies nicht möglich.

Listing 2.14 zeigt den resultierenden Java-Code der modellierten Klasse und der vorgestellten Templates. Die `public` Java-Klasse `Person` beinhaltet ein `private` Attribut vom Typ `String` mit Namen `realName`. Zudem wurden `public` Getter- und Setter-Methoden generiert, die den Zugriff auf das Attribut kapseln.

Die Codegenerierung zusammenfassend lässt sich festhalten, dass MontiCore zwei ausgezeichnete Objekte `ast` und `op`, die innerhalb eines Templates verwendet werden können, anbietet. Das `ast` Objekt stellt dabei die Verbindung zum AST und somit zum Modell sicher. Das `op` Objekt dient der Steuerung des Templateflusses. Weiterführende Informationen und eine Diskussion der Entwurfsentscheidungen ist in [Sch12] zu finden.

Im Rahmen dieser Arbeit sind, bedingt durch die Wahl von MontiCore, alle Generatoren durch Freemarker Templates, die den Quellcode erzeugen, umgesetzt. Alles in allem stellt MontiCore ein Werkzeug zur Definition domänenspezifischer Sprachen sowie eine Infrastruktur zur Prüfung von Kontextbedingungen, Erstellung von Symboltabellen und

```
1  public class Person {                                          Java
2                                                                  ...
3    private String realName;
4
5    public String getRealName(){
6      return realName;
7    }
8
9    public void setRealName(String realName){
10     this.realName = realName;
11   }
12 }
```

Listing 2.14: Generierter Java-Code als Resultat der Eingabe aus Listing 2.11 und den beiden Templates aus Listing 2.12 und Listing 2.13.

Codegenerierung dar. Seine einzelnen Komponenten sind in Abbildung 2.3 dargestellt. MontiCore verwendet ein eigenes Grammatikformat, welches an die EBNF angelehnt ist, und erweitert es um Möglichkeiten, wie z.B. die Definition von Interface-Nichtterminalen und Produktionsregelvererbung. Darüber hinaus bietet MontiCore Möglichkeiten zur Sprachintegration. Die Codegenerierung erfolgt mit Hilfe von Freemarker Templates und bietet Möglichkeiten zum Rückgriff auf das Modell sowie zur Steuerung des Templateflusses. Im nächsten Abschnitt 2.3 wird die Sprachfamilie UML/P, auf die die in der Arbeit vorgestellten Ergebnisse aufbauen, vorgestellt.

2.3 Die Sprachfamilie UML/P

Basierend auf der zuvor vorgestellten Language Workbench MontiCore wurde die Sprachfamilie UML/P [Sch12, Rum11, Rum12] entworfen und implementiert. Die UML/P dient dabei zur Modellierung vollständiger Softwaresysteme und deren Tests. Daher umfasst die Sprachfamilie struktur- und verhaltensmodellierende Sprachen sowie eine Aktionssprache, eine Testfallsprache und eine Bedingungssprache. Zur Modellierung von Struktur können Klassendiagramme (CDs) eingesetzt werden. Zur Modellierung des Verhaltens können Zustandsdiagramme (SCs), Sequenzdiagramme (SDs) und die Aktionssprache Java/P eingesetzt werden. Darüber hinaus können durch Klassendiagramme modellierte Systeme mit Hilfe der OCL/P durch geeignete Bedingungen eingeschränkt werden. Der Zustand von Systemen kann durch Objektdiagramme (ODs) dargestellt und Testfälle (TCs) können mit Hilfe der Testfallsprache modelliert werden.

In [Sch12] wurde eine Werkzeuginfrastruktur zur Entwicklung mit der UML/P geschaffen, die auf der UML/P Definition aus [Rum11, Rum12] basiert. Diese Werkzeuginfrastruktur umfasst eine Integration der verschiedenen Sprachen und eine Menge von prototypischen Codegeneratoren zur Umsetzung einfacher Java-Systeme. Zur Modellierung wurde dazu die Java/P, die in ihrer konkreten Syntax als auch ihrer Ausdrucksmächtig-

keit Java 6 gleicht, in CDs eingebettet, so dass Methodenrümpfe in einem CD modelliert werden können. Durch Sprachaggregation kann zusätzliche Applikationslogik in einem SC modelliert werden, welches das CD referenziert. Die Testfallsprache bettet Java/P oder SDs als Testtreiber ein und verwendet ODs zur Formulierung des Systemzustands vor und nach Testausführung. Mit Hilfe einer Reihe von Intra- und Intermodellkontextbedingungen, sowohl sprachintern als auch sprachübergreifend, kann die Konsistenz der Modelle gesichert werden. Für diese Modelle existieren dann verschiedene Codegeneratoren, die einen spezifischen Anwendungsfall der UML/P umsetzen können. Im weiteren Verlauf der Arbeit werden die Sprachen CD und OD stärker verwendet und in den folgenden Abschnitten 2.3.1 und 2.3.2 detaillierter vorgestellt. Im weiteren Verlauf der Arbeit werden diese Sprachen erweitert und um weitere Sprachen sowie Konzepte ergänzt.

2.3.1 Klassendiagramme

Klassendiagramme (CDs) sind Teil der strukturellen Sprachen und werden zur Modellierung des Systems und seiner Struktur verwendet. In der UML/P sind Klassendiagramme als textuelle Sprache realisiert. Eine zur Erläuterung aufbereitete und gekürzte Version der Grammatik der CD Sprache wurde bereits in Abschnitt 2.2 vorgestellt. Die hier vorgestellten Modelle sind korrekte UML/P Modelle basierend auf der vollständigen Grammatik der UML/P. Eine vollständige Version der Grammatik ist in [Sch12] zu finden.

```
1 package montiee;                                                    CD
2                                                                     ...
3 classdiagram SocNet {
4   // weitere Elemente des Klassendiagramms
5 }
```

Listing 2.15: Auszug des generellen Aufbaus eines textuellen Klassendiagramms ohne enthaltene Elemente.

Listing 2.15 zeigt die Grundstruktur eines Klassendiagramms. Jedes Klassendiagramm beginnt mit der optionalen Angabe eines Paketnamens zur hierarchischen Strukturierung. Mit Hilfe des Schlüsselworts `import` können weitere Typen in das Klassendiagramm importiert werden. Danach beginnt das eigentliche Klassendiagramm mit dem Schlüsselwort `classdiagram` gefolgt vom Namen des Klassendiagramms. In Listing 2.15 ist der Name des Klassendiagramms `SocNet`, welches sich im Paket `montiee` befindet. Das dargestellte Klassendiagramm wird im weiteren Verlauf der Arbeit als Anwendungsbeispiel weiter erläutert und aufgebaut werden, dient an dieser Stelle aber in seiner gekürzten, sich aufbauenden Version zunächst zur Erläuterung der Klassendiagrammsprache. Innerhalb des Klassendiagrammblocks können mehrere Elemente modelliert werden: Klassen, Interfaces, Enumerationen und Assoziationen.

```
1  abstract class Profile {                                    CD
2    String userName;                                          ...
3  }
4
5  class Person extends Profile {
6    String realName;
7  }
8
9  class Commercial extends Profile;
```

Listing 2.16: Modellierte Elemente des Klassendiagramms. Modelliert sind die Klassen `Profile`, `Person` und `Commercial` sowie die Vererbungsbeziehung zwischen diesen.

Listing 2.16 zeigt drei modellierte Klassen darunter die bereits in Listing 2.11 verwendete und hier erweiterte Klasse `Person`. Alle Klassen beginnen dabei mit einem optionalen Modifikator und dem Schlüsselwort `class` gefolgt von einem Namen. Klassen können durch das Schlüsselwort `abstract` als abstrakt markiert werden und können von anderen Klassen erben. Dies wird durch das Schlüsselwort `extends` modelliert. Darüber hinaus können Klassen auch Interfaces, was durch das Schlüsselwort `implements` modelliert wird, implementieren oder Interfaces können von Interfaces erben. Klassen haben entweder einen Rumpf oder werden direkt mit einem Semikolon beendet.

Als weitere mögliche Elemente beginnen Interfaces mit dem Schlüsselwort `interface`, Enumerationen mit `enum` und Assoziationen, wie in Listing 2.17 dargestellt mit dem Schlüsselwort `association`. Neben Assoziationen können ebenfalls Kompositionen mit dem Schlüsselwort `composition` und Aggregationen mit dem Schlüsselwort `aggregation` modelliert werden. Alle Elemente besitzen nach dem jeweiligen Schlüsselwort einen Namen, der bei Assoziationen optional ist. Hierbei sei angemerkt, dass die Modellierung von Mehrfachvererbung sowohl bei Klassen als auch bei Interfaces erlaubt ist. Innerhalb des Rumpfes können Methoden oder Attribute modelliert werden. Attribute bestehen dabei aus einem Typ und einem Namen, wie in Listing 2.16 durch das Attribut `userName` von Typ `String` dargestellt. Attribute können zusätzlich eine Sichtbarkeit, wie `public (+)`, `protected (#)` oder `private (-)`, haben. Neben den Sichtbarkeiten sind auch weitere Modifikatoren, wie beispielsweise `final`, möglich. Die Modellierung von Methoden wurde der Implementierung von Methoden in Java nachempfunden. Jede Methode hat eine Sichtbarkeitsangabe, bzw. einen Modifikator, einen Rückgabetyp, einen Namen, eine Parameterliste und einen Rumpf. Die Angabe des Rumpfes ist optional. Wird ein Rumpf angegeben, so wird dieser mit Hilfe der eingebetteten Aktionssprache Java/P modelliert. Eine Prüfung, dass in Interfaces nur Methoden ohne Rumpf, oder in abstrakten Klassen nur abstrakte Methoden ohne Rumpf, bzw. konkrete Methoden mit Rumpf modelliert werden, wird über geeignete Kontextbedingungen abgefangen.

```
1  association follows [*] Person -> Commercial [*];                CD
2                                                                    ...
```

Listing 2.17: Modellierte Assoziation des Klassendiagramms.

Listing 2.17 zeigt Assoziationen zwischen modellierten Klassen des sozialen Netzwerks mit Hilfe der UML/P. Assoziationen werden, wie auch in der UML explizit als eigenständige Elemente innerhalb des Klassendiagramms und nicht in eine Klasse eingebettet modelliert. Assoziationen bestehen dabei neben dem Schlüsselwort `association` und dem optionalen Namen aus einer linken und einer rechten Assoziationsseite sowie einer Richtung. Die linke sowie auch die rechte Assoziationsseite besitzen dabei eine Referenz auf die Klassen, die durch die Assoziation verbunden werden. Darüber hinaus können Kardinalitäten auf beiden Seiten der Assoziation angegeben werden. Die möglichen Kardinalitäten beinhalten ”0”, ”1”, ”∗” und ”+” zur Modellierung der entsprechenden Objekte, die in Relation zueinander stehen. Ein ”+” steht dabei für beliebig viele, aber mindestens ”1”, und ein”∗” für beliebig viele. Neben diesen in der Praxis am häufigsten vorkommenden Kardinalitäten können aber auch beliebige natürliche Zahlen zur Modellierung verwendet werden. Des Weiteren können Intervalle der Form ”*n..m*” angegeben werden, wobei auch hier das Intervall ”0..1” das in der Praxis am meisten genutzte ist. Hinzu kommt die optionale Möglichkeit zur Angabe von Rollennamen und Qualifikatoren. Die Navigationsrichtung wird durch Angabe von ”− >”, ”< −”, ”−” und ”< − >” modelliert, wobei ”−” für eine unterspezifizierte Navigationsrichtung steht. Listing 2.17 zeigt eine unidirektionale Assoziation namens `follows` zwischen den Klassen `Person` und `Commercial`, wobei durch die Kardinalitäten ausgedrückt wird, dass viele Personen vielen kommerziellen Profilen folgen können.

Die Elemente des Klassendiagramms können über ihren Namen, da dieser innerhalb eines Klassendiagramms eindeutig ist, referenziert werden. Das Klassendiagramm selbst spannt also einen Namensraum auf. Der Klassendiagrammname hingegen muss innerhalb des Pakets, welches ebenfalls einen Namensraum aufspannt, eindeutig sein. Ein Element des Klassendiagramms lässt sich also eindeutig durch die vollqualifizierte Angabe über Paket-, Klassendiagramm- und Elementnamen, bzw. der Konkatenation mehrerer Elementnamen, wie beispielsweise bei Attributen, identifizieren. Dies ist allerdings für Assoziationen und Methoden nicht korrekt. Eine Methode wird eindeutig über ihren Namen und ihre Parameterliste identifiziert, eine Assoziation über den Assoziationsnamen, die beiden Referenzen zu den beteiligten Klassen und den Rollennamen. Eine vollständige Übersicht über die Merkmale der Klassendiagramme findet sich in [Rum11, Rum12, Sch12]. Dort wird ebenfalls eine Übersicht über die Menge der vorhandenen Kontextbedingungen sowie die Semantik der Sprachen gegeben.

2.3.2 Objektdiagramme

Im Gegensatz zu Klassendiagrammen modellieren Objektdiagramme keine Systemstrukturen, sondern Systemzustände zu einem gegebenen Zeitpunkt. Sie repräsentieren somit die im objektorientierten System vorhandenen Objekte und deren Verbindungen untereinander. Ein Objektdiagramm gilt als konsistent zu einem Klassendiagramm, wenn es die vom Klassendiagramm vorgeschriebene Struktur nicht verletzt. Dies betrifft vorhandene Objekte, Attribute und ihre Belegungen sowie die Einhaltung von Kardinalitäten. In der objektorientierten Programmierung hat eine Klasse viele Instanzen in Form von Objekten, daher findet sich in der Literatur auch häufig der Ausdruck, dass das Objektdiagramm eine Instanz des Klassendiagramms sei. Aus modellierungssprachtheoretischer Sicht ist dieser Begriff aber doppeldeutig verwendet, da konkrete Modelle der Ebene M0 Instanzen von Grammatiken der Ebene M1 sind. Zwischen Objekt- und Klassendiagrammen besteht aber keine inhärente Beziehung.

```
1 package montiee;                                                    OD
2                                                                     ...
3 objectdiagram Profiles {
4   // weitere Elemente des Objektdiagramms
5 }
```

Listing 2.18: Auszug des generellen Aufbaus eines textuellen Objektdiagramms ohne enthaltene Elemente.

Dennoch existiert sicherlich eine Konformitätsbeziehung zwischen beiden Modellen. Im Rahmen dieser Arbeit wird ein Objektdiagramm als eine Repräsentation des Zustands eines objektorientierten Systems verstanden. Dieser ist durch die Menge der im System vorhandenen Objekte und deren Attributbelegungen, zu einem Zeitpunkt definiert. Er kann zu einem Klassendiagramm konform sein kann. Generell wird ein Objektdiagramm, welches konform zu einem Klassendiagramm, ist als ein Diagramm verstanden, welches eine gültige Instanz des Klassendiagramms darstellt und keinen modellierten Elementen widerspricht. Solche Bedingungen können Attribute, Attributwerte, vorhandene Typen, vorhandene Assoziationen und insbesondere auch Kardinalitäten, Assoziations- und Rollennamen sein. Wie auch die Klassendiagrammsprache der UML/P ist die Objektdiagrammsprache der UML/P ebenfalls textuell umgesetzt worden. Die Sprache der Objektdiagramme beginnt wie auch die Klassendiagramme mit einer Paketangabe gefolgt vom Schlüsselwort `objectdiagram` und einem Namen. Listing 2.18 zeigt einen Auszug eines Objektdiagramms, dass konform zu dem zuvor vorgestellten Klassendiagramm ist. Das Objektdiagramm modelliert einen möglichen Systemzustand. Das hier vorgestellte Objektdiagramm liegt im Paket `montiee` und hat den Namen `Profiles`. Es stellt einen Auszug des Systemzustands des sozialen Netzwerks zu einem Zeitpunkt dar. Die Elemente, die innerhalb des Objektdiagramms modelliert werden können, sind Objekte und Links.

```
OD
...
bob:Person {
  userName="Bob.Doe";
  realName = "Bob Doe";
}

alice:Person {
  userName="Alice.Doe";
  realName = "Alice Doe";
}

soccer:Commercial {
  userName="Soccer F.C"
}
```

Listing 2.19: Modellierte Elemente des Objektdiagramms. Modelliert sind die Objekte `bob`, `alice` und `soccer`.

Listing 2.19 zeigt drei modellierte Objekte. Ein Objekt beginnt mit der Angabe eines optionalen Namens und eines Typs. Der Name dient als eindeutiger Identifikator innerhalb des Diagramms und kann zur Referenzierung verwendet werden. Ist kein Name modelliert, wird das Objekt als anonymes Objekt bezeichnet. Der Typ des Objekts referenziert die Klasse für die dieses Objekt eine Instanz darstellt. Dabei ist es innerhalb des Diagramms nicht zwingend erforderlich immer den konkretesten Typ eines Objekts zu modellieren, sondern es können auch polymorphe Supertypen verwendet werden.

In Listing 2.19 sind zwei Personen und ein kommerzielles Profil als Objekte modelliert. Die `realName` Attribute der Personen sind mit dem String ”Bob Doe” bzw. ”Alice Doe” belegt. Zusätzlich haben beide Objekte ein `userName` Attribut, welches im Klassendiagramm innerhalb der `Profile` Klasse modelliert wurde und durch die Vererbung auch der Klasse `Person` bekannt ist. Neben den beiden Personen existiert ein kommerzielles Profil, dessen `userName` Attribut ebenfalls belegt ist. Das Objekt repräsentiert eine Fußballmannschaft, die den Namen ”Soccer F.C” trägt.

Listing 2.20 zeigt einen zugehörig modellierten Link. Links haben nach ihrem Schlüsselwort einen optionalen Namen, der die Assoziation des Klassendiagramms benennt, die von diesem Link instanziiert wird. Daran anschließend können ein oder mehrere Objektnamen, ein ”−” wieder gefolgt von einem oder mehreren Objektnamen modelliert werden. Semantisch bedeutet dies, dass jedes Objekt der linken Seite einen Link zu allen Objekten der rechten Seite hat. Die Objekte `bob` und `alice` aus Listing 2.19 haben einen Link zu dem kommerziellen Profil. Dies bedeutet, dass Bob und Alice der Fußballmannschaft semantisch folgen.

Der gezeigte Link ist konform zu der Assoziation des Klassendiagramms aus Listing 2.17, die modelliert, dass viele Personen vielen kommerziellen Profilen folgen können. Im Rahmen dieser Arbeit werden Objektdiagramme als Serialisierungsformat zur Datenmigration, wie in Kapitel 9 beschrieben, verwendet. Diese eignen sich als Serialisierungsformat, da sie eine kompakte Repräsentation von Systemzuständen darstellen. Zudem

```
1  link follows bob, alice - soccer;                                    OD
2                                                                       ...
```

Listing 2.20: Modellierte Links des Objektdiagramms.

dienen sie in der UML/P als Eingabe für modellbasierte Testfälle, die im Zuge einer Systemevolution, analog zu den zu migrierenden Daten, wie in [Plo12] beschrieben, ebenfalls angepasst werden müssen. Die im Rahmen dieser Arbeit entwickelte Möglichkeit zur Datenmigration kann dadurch auch auf das modellbasierte Testen, wie es von der UML/P beschrieben wird, angewendet werden.

2.4 Zusammenfassung

In diesem Kapitel wurden zunächst die grundlegenden Begriffe und Definitionen, die im Rahmen dieser Arbeit verwendet werden, vorgestellt. Dazu wurde zunächst der Modellbegriff eingeführt und eine Einordnung in MDD vorgenommen. Darauf aufbauend wurden gängige Elemente des MDD, wie Transformationen, vorgestellt. Eine tiefergehende Einführung in DSLs wurde in Abschnitt 2.1 präsentiert. Dort wurden die Elemente einer DSL, die unterschiedlichen Modellebenen, aber auch eine Kosten-Nutzen Betrachtung solcher Sprachen erläutert. Daran anschließend wurde die Language Workbench MontiCore vorgestellt und ihre Wahl begründet. Das Grammatikformat und die Möglichkeiten zur Komposition von DSLs wurden beleuchtet. Darüber hinaus wurden weitere Elemente, wie Kontextbedingungen, eingeführt. Daran anschließend wurde eine kurze Einführung in die Codegenerierung mit Freemarker gegeben. Auf Basis von MontiCore und der Erklärung des Grammatikformats wurde die Sprachfamilie UML/P vorgestellt. Insbesondere wurden die in der Sprachfamilie enthaltenen Sprachen der Klassendiagramme und der Objektdiagramme vorgestellt.

Im Rahmen dieser Arbeit wird die Sprachfamilie MontiEE vorgestellt, die zum einen Klassendiagramme und Objektdiagramme zur Systemspezifikation verwendet, zum anderen aber weitere DSLs zur Modellierung spezifischer Sachverhalte im Kontext von Enterprise Applikationen einführt. Diese DSLs werden ebenfalls mit MontiCore entworfen und geeignete Kontextbedingungen werden definiert. Auf Basis dieser Sprachen werden dann Modelle und eine Modellierungsmethodik zur Entwicklung von Enterprise Applikationen gezeigt, die Codegeneratoren zur Generierung großer Teile des Systems verwendet. Diese Codegeneratoren sind mit Freemarker umgesetzt. In Abschnitt 3.4 werden verwandte Arbeiten vorgestellt. In Abschnitt 3.5 werden dann beteiligte Rollen bei der Systementwicklung und Anforderungen dieser Rollen auf Basis eines durchgängigen Szenarios gezeigt.

Im nächsten Kapitel werden die Grundlagen von Enterprise Applikationen und die verwendeten Technologien vorgestellt. Daran anschließend folgt die Vorstellung der Sprachfamilie MontiEE und der umgesetzten Codegeneratoren sowie des MDD mit MontiEE.

Kapitel 3

Entwicklung von Enterprise Applikationen

Nachdem zuvor die Grundlagen und wichtigsten Begriffe des MDD beschrieben wurden, werden in diesem Kapitel Enterprise Applikationen und deren Entwicklung vorgestellt. Dazu wird zunächst der Begriff Enterprise Applikation, wie er im Rahmen dieser Arbeit verwendet wird, eingegrenzt und darauf aufbauend der typische Aufbau und die typische Architektur sowie benötigte Komponenten beschrieben und präsentiert.

Zunächst wird mit einer Begriffsklärung, die das der Arbeit zugrunde liegende Verständnis darstellt, begonnen.

3.1 Enterprise Applikationen

In der Literatur lässt sich für den Begriff Enterprise Applikationen keine allgemeingültige Definition finden, so dass häufig auf eine Beschreibung ihrer Eigenschaften zurückgegriffen wird.

Martin Fowler [Fow03] charakterisiert Enterprise Applikationen daher wie folgt:

> „I can't give a precise definition, but I can give some indication of my meaning. I'll start with examples. Enterprise applications include payroll, patient records, shipping tracks, cost analysis, credit scoring, insurance, supply chains, accounting, customer service, and foreign exchange trading. Enterprise applications don't include automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games."

Dieses Zitat macht deutlich, dass eine genaue Definition, da es sich um heterogene Systeme mit den unterschiedlichsten Anwendungsdomänen handelt, schwierig ist. Historisch gesehen werden solche Enterprise Applikationen auch als Informationssysteme bezeichnet [Fow03], wobei sich der Begriff weiterentwickelt hat. Auch wenn die Definition von Enterprise Applikationen über die Anwendungsdomäne nicht möglich ist, haben diese Systeme aber viele Eigenschaften gemeinsam. Ein Teil dieser gemeinsamen Eigenschaften lässt sich in der historischen Definition eines Informationssystems wie sie [Kaj12] wiedergibt, nach der ein Informationssystem:

> „a collection of people, procedures, and equipment designed, constructed, operated, and maintained to collect, record, process, store, retrieve, and display information"

darstellt. Dies zeigt, dass ein Informationssystem im Wesentlichen dazu dient, Informationen darzustellen, zu speichern, zu laden und anzuzeigen. Auch zeigt diese Definition, dass es auch unterschiedliche Rollen und Technologien gibt, die zusammen mit der Anwendungsfunktionalität das Informationssystem bilden. Darüber hinaus wird in [SSN01] der Begriff eines Enterprise Informationssystem verwendet. Ein solches Enterprise Informationssystem (EIS) wird dabei wie folgt beschrieben:

> „[A]n application or enterprise system that provides the information infrastructure for an enterprise. An EIS consists of one or more applications deployed on an enterprise system. An EIS provides a set of services to its users. Services exposed to clients may be at different levels of abstraction—including the system level, data level, function level, and business object or process level."

Diese Definition ist der eines Informationssystems ähnlich, erweitert diese allerdings. Sie nimmt stärker Bezug auf das Unternehmen und die Bereitstellung von Diensten für unterschiedliche Nutzer. Dabei zeigt diese Definition, dass es sich um unterschiedliche Anwendungen, die von Nutzern verwendet werden, handelt und die es erlauben, Funktionen auszuführen, Daten zu betrachten oder Geschäftsprozesse zu befolgen. Nutzer können also auf die unterschiedlichen Anwendungen zugreifen und Geschäftslogik ausführen sowie gespeicherte Daten einsehen oder manipulieren. Dies zeigt einen Teil der überlappenden Eigenschaften, die von der Anwendungsdomäne unabhängig sind.

Diese und weitere Eigenschaften werden auch in [Fow03] zur genaueren Einschränkung und Definition aufgegriffen. Die Eigenschaften von Enterprise Applikationen werden so beschrieben, dass solche Anwendungen persistente Daten behandeln, eine große Menge Daten verarbeiten, die Daten mehreren Nutzern parallel zur Verfügung stellen, eine Vielzahl von Elementen der Graphischen Benutzerschnittstelle GUI enthalten, mit anderen Enterprise Applikationen kommunizieren und Geschäftslogik beinhalten. Dies entspricht dem Verständnis der Eigenschaften von Enterprise Applikationen in dieser Arbeit.

Diese Eigenschaften sowie die Definitionen zuvor zeigen deutlich, dass es im Wesentlichen darum geht, unterschiedlichen Nutzern eine Menge an Daten zur Bearbeitung zur Verfügung zu stellen und sie bei der Ausführung von Geschäftsprozesse zu unterstützen. Es wird auch deutlich, dass Daten meistens in einer Datenbank gespeichert werden müssen, dass sie meist über eine Graphische Benutzerschnittstelle (GUI) veränderbar sein müssen und meistens mehrere Benutzer gleichzeitig auf das Mehrbenutzersystem zugreifen. Dazu wird aber auch ein Datenaustausch zwischen den beteiligten Clients und den Servern benötigt. Aus diesem Grund werden solche Systeme auch Informationssysteme genannt, da sie Informationen bereitstellen und der Nutzer diese manipuliert. Gleichzeitig bringen diese gemeinsamen Eigenschaften Herausforderungen an die Persistenz, die Skalierbarkeit, die Performance und die Behandlung von parallel arbeitenden Benutzern mit. Im Rahmen dieser Arbeit wird dieser Definition folgend, ebenfalls die Entwicklung einer solchen Enterprise Applikation unterstützt.

Der Begriff EIS wird im Rahmen der Enterprise Architecture [SH93] und des Enterprise Modelling [FG98] auch weiter gefasst verstanden. Zur besseren Abgrenzung werden

kurz die Konzepte der Enterprise Architecture und des Enterprise Modelling aufgezählt und Gemeinsamkeiten zu den zuvor vorgestellten Enterprise Applikationen und Informationssystemen hervorgehoben.

Dabei umfasst Enterprise Architecture das gesamte Unternehmen und berücksichtigt Personen, Strategien und Möglichkeiten. Zur Erfassung aller Bereiche einer solchen Architektur existieren verschiedene Frameworks. Eine Übersicht über verschiedene Frameworks ist in [Sch06] gegeben. Die beiden bekanntesten sind das Zachman [Z+87] und das TOGAF Framework [Tog15]. Beide Frameworks definieren Bereiche und Teilbereiche, die Teil der Enterprise Architecture sind. Zachman stellt dies tabellarisch dar und benennt für jede Zelle der Tabelle die notwendigen Schritte zur Umsetzung einer Enterprise Architecture. Dabei stellen die Spalten die Fragen *Was*, *Wie*, *Wo*, *Wer*, *Wann* und *Warum* dar, wohingegen die Zeilen Geschäftskonzepte, Geschäftslogik, Technologien und Komponenten umfassen. Die vollständige Übersicht der Tabelle kann in ihrer aktuellsten Version in [Zac15] eingesehen werden. Das TOGAF Framework stellt dem verschiedene Schritte, die allesamt durchlaufen werden müssen, gegenüber. Eine Auswahl dieser Schritte umfasst die Erstellung der gewünschten Unternehmensstruktur, die Analyse der aktuellen Struktur, aber auch die Erfassung der Technologischen Architektur. Ebenfalls Teil des TOGAF Frameworks ist der Punkt *Information System Architecture*, der sich in die Punkte *Application Architecture* und *Data and Information Architecture* unterteilt [Sno14]. Dabei umfassen diese Teile Informationen über Kommunikation zwischen beteiligten Systemen und Funktionen, aber auch das Domänenmodell, das letztendlich als Datenmodell dient.

Das Themenfeld des Enterprise Modelling bezieht sich dabei auf die vollständige Modellierung eines Unternehmens im Sinne der zuvor genannten Frameworks. Es werden dazu Modelle für die jeweiligen Schritte oder Tabellenzellen verwendet. Das breite Themengebiet der Enterprise Architecture wird im Rahmen dieser Arbeit nicht weiter betrachtet, da die Ausrichtung, wie zuvor genannt, eine andere ist. Der Unterpunkt der Information System Architecture verhält sich kompatibel zur Verwendung der Konzepte dieser Arbeit und muss nicht gesondert betrachtet werden.

Ein weiteres breites Themenfeld, in das sich diese Arbeit einordnet und aus denen Konzepte wiederverwendet werden, ist das Themenfeld des *Web Engineerings*. In diesem Themenfeld geht es um die Entwicklung moderner Webanwendungen. Diese umfassen Web 2.0 Anwendungen, mobile Anwendungen, aber auch Rich Internet Applications (RIAs) [RPSO07]. Auch diese Anwendungen teilen viele zuvor genannte Eigenschaften von Enterprise Applikationen. Sie verändern sich schnell, werden von vielen unterschiedlichen Nutzern auf unterschiedlichen Geräten verwendet, präsentieren ihren Inhalt in einer GUI, benötigen einen inkrementellen Entwicklungsprozess und haben Anforderungen an Sicherheit und Privatsphäre. Während sich einige der hier genannten Eigenschaften auf das Frontend, also den Teil, der nutzerseitig ausgeführt wird, beziehen, verwenden die meisten Webanwendungen ein Backend. Dieses Backend besitzt die gleichen Eigenschaften wie eine Enterprise Applikation. Das Frontend einer Webanwendung kann dabei als die GUI der Enterprise Applikation verstanden werden, wohingegen der Server einer solchen Anwendung die Enterprise Applikation selbst ist, auch wenn Teile

der GUI serverseitig vorgehalten sein können. Aber auch hier spielen Fragestellungen nach dem Datenmodell und der Kommunikation eine zentrale Rolle.

Auch sind verschiedene Frameworks zur Modellierung von Websystemen oder Enterprise Applikationen entstanden. Diese umfassen unterschiedliche Frameworks, wie beispielsweise: Hera [HBFV03, HSB+08], die Web Semantics Design Method (WSDM) [TCP08], die Object-Oriented Hypermedia Design Method (OOHDM) [SR95, RS08], Netsilon [MSFB05], MontiWIS [RR13, Rei16], WebDSL [Vis08], UML-based Web Engineering (UWE) [KKZB08], WebML [CFB00, BCFM00], Object-Oriented Web Solution (OOWS) [PFPA06] und Eclipse Scout [Zim16]. Eine detaillierte Vorstellung sowie eine Abgrenzung gegenüber diesen bestehenden Frameworks wird in Abschnitt 3.4 gegeben.

Im Rahmen dieser Arbeit wird eine Enterprise Applikation als eine Anwendung verstanden, die die oben genannten Eigenschaften beinhaltet. Besonderer Fokus wird dabei auf die persistenten Daten, das parallele Bereitstellen von Daten sowie die Kommunikation zwischen Server und unterschiedlichen Clients gelegt. Die Clients sind dabei vormerklich als Webanwendungen anzusehen, können aber auch andere Enterprise Applikationen sein. Der Fokus liegt auf der Entwicklung des Servers und der Bereitstellung der Funktionalität des Servers sowie der Kommunikation mit heterogenen Clients. Dabei werden Konzepte des Web-Engineerings und der Entwicklung von Enterprise Applikationen verwendet.

Im Rahmen dieser Arbeit wird die Entwicklung solcher Systeme durch DSLs und Generatoren unterstützt, so dass der komplexe Entwicklungsprozess des Servers vereinfacht werden kann. Nachdem zuvor eine Einordnung des Begriffs der Enterprise Applikation vorgenommen wurde, wird im nächsten Abschnitt die typische Architektur sowie typische Komponenten einer Enterprise Applikation gegeben.

3.2 Architektur von Enterprise Applikationen

Wie bereits zuvor erläutert, besteht eine wesentliche Aufgabe einer Enterprise Applikation darin, dem Nutzer Informationen bereitzustellen. Die Informationen müssen persistent gespeichert sein und die Nutzer können unterschiedliche Clients verwenden. Zur Speicherung der Informationen wird eine Datenbank verwendet, wohingegen die Clients in der Regel Webanwendungen oder Rich-Clients sind. Sowohl mit der Datenbank als auch mit den unterschiedlichen Clients und mit anderen Enterprise Applikationen muss die Enterprise Applikationen kommunizieren können. Im Rahmen dieses Kapitels werden die typische Architektur einer Enterprise Applikationen sowie Kommunikationsmechanismen und Kommunikationsinhalte vorgestellt.

Die Architektur von Enterprise Applikationen ist typischerweise als Schichtenarchitektur angelegt. Sie besteht im Wesentlichen aus drei verschiedenen Schichten, die wiederum unterteilt sein können. Zumeist werden sie als Präsentationsschicht, Domänenschicht und Datenquellschicht bezeichnet [Fow03].

Dabei ist die Präsentationsschicht dafür verantwortlich, dass Informationen und Daten angezeigt sowie übertragen werden. Sie kümmert sich um einen Teil der GUI und die Kommunikation mit unterschiedlichen Clients. Die Domänenschicht enthält die Anwen-

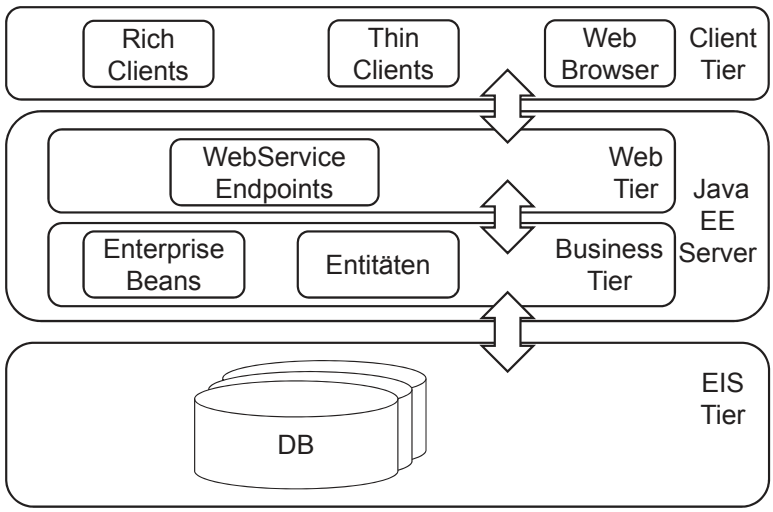


Abbildung 3.1: Charakteristische Architektur einer Enterprise Applikation im JEE Umfeld. Abbildung in Anlehnung an [Ora10].

dungslogik und die Domänenobjekte. Die Datenquellschicht kapselt die Kommunikation mit Datenbanken.

Im Rahmen dieser Arbeit wird eine Enterprise Applikation mit Hilfe der Java Enterprise Edition (JEE) [Hef10] umgesetzt und die dort typischen Komponenten und Technologien werden verwendet, da die JEE den etablierten de-facto Industriestandard zur Entwicklung portabler, robuster, skalierbarer und sicherer Serveranwendungen darstellt [Hef10]. Auf Konzeptebene ist dies aber analog zur Entwicklung von Enterprise Applikationen für andere Zielplattformen.

Abbildung 3.1 zeigt einen typischen Aufbau einer Enterprise Applikation mit den zuvor beschriebenen Schichten am Beispiel eines typischen JEE Technologiestacks. Es sind drei Schichten dargestellt, wobei die mittlere Schicht unterteilt ist. Die oberste Schicht, der *Client Tier*, ist mit der zuvor beschriebenen Präsentationsschicht gleichzusetzen. In Abbildung 3.1 ist zu erkennen, dass dort im Wesentlichen Webseiten, Applets oder vollwertige Applikationen mit der darunter liegenden Schicht kommunizieren. Die darunter liegende Schicht wird auf einem Applikationsserver ausgeführt und besteht aus zwei Schichten: dem *Web Tier* und dem *Business Tier*. Der *Web Tier* stellt dabei den serverseitigen Teil der Präsentationsschicht dar, da dieser die direkte Kommunikation mit den Clients übernimmt und die Daten zur Anzeige aufbereitet. Der *Business Tier* hingegen stellt die Domänenschicht dar und beinhaltet die Anwendungslogik sowie die Domänenobjekte. Die Anwendungslogik ist dabei in Enterprise JavaBeans (EJB) gekapselt. Diese sind in *Session Beans* oder *Message-Driven Beans* unterteilt. Die persistenten Daten sind als *Persistence Entities* umgesetzt. Eine genauere Erklärung der Beans wird in Abschnitt 3.3 gegeben. Die Datenquellschicht befindet sich zum Teil auf dem Applikationsserver, aber auch im *EIS Tier*, der unterschiedliche Datenbanken beinhalten kann.

Dabei wird die Kommunikation mit dem konkreten Datenbanksystem vom Applikationsserver abstrahiert.

Es ist anzumerken, dass die Enterprise Applikation als solche auch auf mehreren Servern mit unterschiedlichen Aufgaben und unterschiedlichen Clients bestehen kann. So können mehrere Applikationsserver existieren, die auf einen gemeinsamen Datenbankserver, oder aber auf mehrere Datenbankserver, zugreifen. Eine häufig verwendete Implementierung eines Applikationsservers im JEE Umfeld ist der Glassfish Applikationsserver [Hef10] oder der JBoss Applikationsserver [MD05]. Im Rahmen dieser Arbeit werden die JEE als Industriestandard und ein Glassfish Applikationsserver verwendet. Die Wahl des Glassfish Applikationsservers resultiert aus seiner hohen Verfügbarkeit als Open-Source Produkt und daraus, dass keine spezifischen JBoss Applikationsserver Funktionalitäten benötigt werden. Dennoch sind die beiden Applikationsserver austauschbar, da im Rahmen dieser Arbeit die JEE verwendet wird und keine spezifischen Applikationsserver Funktionalitäten eingesetzt werden.

Nachdem die generelle Architektur erläutert wurde, wird in den folgenden Abschnitten die Kommunikation näher betrachtet. Gemäß dem Sender-Empfänger-Modell [Gra72] sind bei einer Kommunikation unterschiedliche Merkmale zu beachten. Diese Merkmale sind der Sender und der Empfänger einer Nachricht, die Nachricht selbst, ihre En- und Dekodierung, der Kommunikationskanal sowie Störquellen und Rückmeldungen. Für diese Arbeit wird die Kommunikation zwischen Clients als Sender und Applikationsserver als Empfänger sowie zwischen Datenbanksystem als Sender und Applikationsserver als Empfänger betrachtet. Zudem werden das Format der gesendeten Nachrichten, also die En- und Dekodierung einer Nachricht, sowie der Kommunikationskanal präsentiert. Die Merkmale Störquellen und Rückmeldung sind zumeist Teil der verwendeten Technologie des Kommunikationskanals und werden hier nicht näher betrachtet.

Im folgenden Abschnitt 3.2.1 wird zunächst die Kommunikation des *Client Tiers* mit dem *Web Tier* betrachtet.

3.2.1 Clientkommunikation

Wie bereits zuvor beschrieben, findet Kommunikation auf allen Ebenen der Architektur statt. Diese Kommunikation kann zwischen mehreren Applikationsservern, zwischen einem Applikationsserver als Sender und mehreren Datenbanken als Empfänger, aber auch zwischen mehreren Clients als Sender und einem Applikationsserver als Empfänger erfolgen. In diesem und im darauffolgenden Abschnitt 3.2.2 werden die Kommunikationsmöglichkeiten zwischen Clients und Server, aber auch zwischen Server und Datenbanksystem vorgestellt. Ebenso wird die Möglichkeit zur Kommunikation zwischen verschiedenen Applikationsservern kurz angerissen. Die Integration unterschiedlicher Applikationsservern ist durch drei Dimensionen bedingt: Verteiltheit, Heterogenität und Autonomie [Has00, CHKT05]. Durch eine vereinheitliche Kommunikation kann eine Integration von Enterprise Applikationen angestrebt werden. Daher geht dieser Abschnitt auf die Kommunikation mit Clients ein.

Die Clientkommunikation ist meistens durch den Client initiiert, so dass die konkreten einzelnen Clients dem Applikationsserver unbekannt sind, auch wenn ihm die

unterschiedlichen Arten bekannt sind. Der Applikationsserver antwortet meist nur auf Anfragen der Clients, auch wenn einige neuere Frameworks, wie Websockets [WSM13] oder Asynchronous JavaScript and XML (AJAX) [Hol08], auch eine Kommunikation vom Applikationsserver als Sender zum Client als Empfänger, durch Simulation von Polling, unterstützen. Zur Kommunikation der Clients mit dem Applikationsserver kommen unterschiedlichste Technologien zum Einsatz.

Zunächst werden die unterschiedlichen Kommunikationskanäle beschrieben. Im Anschluss daran werden En- und Dekodierung der Nachricht beschrieben und daran anschließend die Nachricht selbst genauer erläutert. Für den Kommunikationskanal kommen unterschiedliche Konzepte zum Einsatz, die auf unterschiedliche Arten einen *Remote Procedure Call (RPC)* umsetzen. Alle diese Konzepte basieren auf tieferen Ebenen auf Kommunikationsprotokollen, wie dem *User Datagram Protocol (UDP)* oder *Transmission Control Protocol (TCP)*. Über dem *TCP* liegt das *Hypertext Transfer Protocol (HTTP)* oder das *Hypertext Transfer Protocol Secure (HTTPS)*. Die Kommunikationsprotokolle werden im Rahmen dieser Arbeit nicht weiter betrachtet. Stattdessen werden höherliegende Kommunikationskanäle kurz erläutert.

Die bekanntesten Konzepte für Kommunikationskanäle sind die *Common Object Request Broker Architecture (CORBA)*, das *Distributed Component Object Model (DCOM)*, und die *Remote Method Invocation (RMI)*. Ebenfalls sehr häufig verwendet sind Webservice-Konzepte wie das *Simple Object Access Protokoll (SOAP)* und der *Representational State Transfer (REST)* [Ses97, ACKM04, HM15]. Darüber hinaus gewinnen *Message Oriented Middlewares (MOMs)* immer mehr an Bedeutung. Sie verwenden eine nachrichtenbasierte Architektur, in der einzelne Komponenten Nachrichten erzeugen, diese global bekannt machen und andere Komponenten auf diese Nachrichten reagieren können. Hinter all diesen verschiedenen Technologien bildet RPC im Wesentlichen das grundlegende Paradigma und wird von ihnen auf unterschiedliche Art und Weise umgesetzt.

CORBA stellt dabei ein komplexes Framework dar, welches dem Grundgedanken der strikten Trennung zwischen Interface und Implementierung folgt. Möchte der Client eine Funktion auf dem Server ausführen, ruft er diese lokal auf dem ihm bekannten Objekt aus, welches als Proxy fungiert. Das Proxyobjekt kapselt den Aufruf sowie die benötigten Parameter und leitet ihn an den entsprechenden Applikationsserver weiter. Dort wird die entsprechende Objektimplementierung über den *Object Request Broker (ORB)* gesucht und die Funktion ausgeführt. Rückgabewerte von Funktionen durchlaufen ebenso diesen Prozess. Dem Client ist das Interface der Implementierung in Form der *Interface Definition Language (IDL)* bekannt. Diese beschreibt die zur Verfügung stehenden Funktionen plattformunabhängig. Jede Programmiersprache, die CORBA unterstützt, besitzt ein eigenes Mapping auf die entsprechenden Interface Konzepte der IDL, so dass der Entwickler mit diesen Konstrukten arbeitet. Im JEE Umfeld stehen dem Entwickler also die kompilierten Java-Interfaces zur Verfügung. Generell unterstützt CORBA nur synchrone Kommunikation, führt zu einer hohen Kopplung der beteiligten Systeme und benötigt ein tiefes Verständnis des Frameworks. DCOM stellt die Microsoft spezifische Variante von CORBA dar und bietet eine ähnliche Funktionalität, ist aber auf die Microsoft Systemlandschaft eingeschränkt. RMI stellt eine Java-spezifische Implementierung von

RPC dar. In seinen Grundzügen ähnelt es der CORBA Funktionalität, ist aber in seiner Komplexität, auch durch die plattformabhängigkeit, deutlich reduziert. Allerdings ist es an Java-basierte Systeme gebunden. Diese zuvor genannten Technologien setzen häufig auch Konzepte für das Auffinden von Implementierungen, Transaktionskonzepte und Authentisierung und Autorisierung um.

Demgegenüber existieren Webservices [ACKM04], die eine lose Kopplung versprechen. Zum einen existieren RESTful Webservices, die über einen *Unified Resource Locator (URL)* adressiert werden und unterschiedliche Ressourcen bereitstellen. Diese Ressourcen können über einen *Uniform Resource Identifier (URI)* verwendet werden. Als mögliche Operationen stehen die gängigen HTTP Operationen *GET*, die eine Ressource liest, *POST* und *PUT*, die eine neue Ressource anlegen oder verändern, und *DELETE*, die eine Ressource löscht, zur Verfügung. Auf eine Diskussion der genauen Unterschiede zwischen *POST* und *PUT* wird an dieser Stelle verzichtet und auf [GT02] verwiesen. Webservices, die diese Funktionalitäten anbieten, werden auch CRUD-Webservices genannt [WPR10]. Generell muss jede REST-Nachricht alle Informationen enthalten, die für die Bearbeitung der Ressource notwendig ist, da es sich um eine zustandslose Kommunikation handelt. Neben RESTful Webservices sind SOAP-Webservices zu nennen, die die Kommunikation standardisiert vorgeben. Sie geben dabei sowohl die Kommunikationssprache als auch das Format der Kommunikation vor. Auch hier können wiederum die gängigen HTTP-Operationen verwendet werden, welche in einer SOAP-Nachricht gekapselt werden [WPR10]. Innerhalb einer SOAP-Nachricht werden die aufzurufende Methode, ihre Parameter sowie die Objekte übergeben. Dadurch lässt sich RPC-artig eine entfernte Funktionalität aufrufen. Analog zur IDL bei CORBA existiert auch bei Webservice eine Beschreibung der angebotenen Funktionalität und der erwarteten Datenstrukturen. Diese Beschreibung ist typischerweise in der *Webservice Description Language (WSDL)* ausgedrückt [ACKM04]. Diese ist unter einer URL zu finden und kann von Clients verwendet werden, um clientspezifische Stubs zu generieren. WSDL-Beschreibungen sind in XML-Syntax und verwenden das XML-Schema Typsystem. Demgegenüber ist bei der IDL, wie sie CORBA einsetzt, das Typsystem der Zielsprache ausschlaggebend. Innerhalb eines WSDL-Dokuments werden die erlaubten Datentypen, Nachrichten und Operationen beschrieben, die hier nicht näher betrachtet werden. Durch die Verwendung einer unabhängigen Beschreibungssprache wird Interoperabilität zwischen unterschiedlichen Systemen erreicht. Die bekanntesten und etabliertesten JEE Implementierungen dieser Funktionalität sind die *Java API for XML Web Services (JAX-WS)* und die *Java API for RESTful Web Services (JAX-RS)* [Bur09], die daher im Rahmen dieser Arbeit verwendet werden.

Nachdem die unterschiedlichen Möglichkeiten für Kommunikationskanäle kurz erläutert wurden, werden die En- und Dekodierung von Daten sowie das verwendete Datenformat beschrieben. Alle Kommunikationskanäle haben gemeinsam, dass sie Daten von einem Sender zu einem Empfänger kommunizieren. Dazu senden sie Nachrichten, die die Daten enthalten. Meistens handelt es sich dabei um Objekte oder ganze Objektgraphen, die serverseitig gespeichert oder verändert werden sollen oder um solche, die vom Client, damit dieser sie anzeigen kann, angefragt werden. Zur Übertragung werden

diese Objektgraphen auf eine bestimmte Art und Weise, die dem Empfänger bekannt sein muss, enkodiert, also serialisiert und können dann beim Empfänger wieder dekodiert, also deserialisiert werden. Die Serialisierung und Deserialisierung wird im Rahmen von Webservices auch häufig als Marshalling und Unmarshalling bezeichnet. Im Rahmen dieser Arbeit wird der Begriff der Serialisierung und Deserialisierung als Synonym der zuvor genannten Begriffe verwendet.

```
1 <Person realName="Bob Doe" userName="Bob.Doe">                    XML
2   <Commercials>                                                   ...
3     <Commercial userName="Soccer F.C"></Commercial>
4   </Commercials>
5 </Person>
```

Listing 3.2: Darstellung eines in XML serialisierten Objektgraphen.

Hauptaufgabe der Serialisierung ist es, Objektgraphen der zu übertragenden Objekte in eine clientseitig wiederverwendbare Form zu transformieren, so dass diese auf der Clientseite wieder in einen Objektgraphen deserialisiert werden können. Hierbei ist sicherlich die Performanz ein entscheidender, aber nicht alleiniger Faktor für die Wahl einer Serialisierung. Weitere Faktoren sind die Lesbarkeit für den Menschen, Plattformunabhängigkeit, Möglichkeiten zur Versionierung oder die Verwendung eines Schemas. Dabei haben alle Formen verschiedene Vor- und Nachteile.

Serialisierungsformate können textbasiert, also für den Menschen lesbar, oder binär sein. Die bekanntesten Formen einer textbasierten Serialisierung stellen die *JavaScript Object Notation (JSON)* [Bas15] und XML [XML06, RS01] dar. Diese textbasierten Ansätze sind im Wesentlichen plattformunabhängig, da sie mit Hilfe eines geeigneten Parsers theoretisch auf beliebigen Plattformen serialisiert, bzw. deserialisiert werden können. Bei Ansätzen, die eine plattformspezifsche, binäre Serialisierung einsetzen, ist dies nicht gegeben. Listing 3.2 zeigt dabei einen Ausschnitt eines serialisierten Objektgraphen, welcher konform zu dem Klassendiagramm aus Listing 2.16 ist. Dieser Objektgraph besteht aus einem Personenobjekt mit den Attributen `realName` und `userName`. Diese sind mit den Werten `Bob Doe` und `Bob.Doe` belegt. Gleichzeitig besitzt das Personenobjekt einen Link zu einem kommerziellen Profil, dessen Attribute ebenfalls dargestellt sind. Dabei fällt auf, dass Links mittels einer Verschachtelung dargestellt werden, was wiederum zu Problemen bei bidirektionalen Links führt. Gleichzeitig zeigt sich, dass der Typ eines Objekts jeweils durch ein XML-Starttag und ein XML-Endetag angegeben werden muss. Kann ein Objekt Links zu mehreren weiteren Objekten haben, müssen zusätzlich Gruppierungstags, wie `Commercials` in Listing 3.2, angegeben werden.

Listing 3.3 zeigt eine Serialisierung des gleichen Objektgraphens wie Listing 3.2, allerdings in JSON Notation. Es wird deutlich, dass die Markierungen für Anfang und Ende eines Objekts reduziert wurden. Zudem werden Mengen von Objekten durch eckige Klammern dargestellt. Dies führt insgesamt dazu, dass die JSON Notation deutlich kompakter ist als die XML-Notation. In [NPRI09] wurde gezeigt, dass durch die Vielzahl

vorhandener Strukturinformationen in der XML-Serialisierung die JSON Serialisierung, Übertragung und Deserialisierung schneller ist und gleichzeitig weniger Ressourcen verbraucht.

```
{                                                                    JSON
  "Person": {                                                        ...
    "realName":"Bob Doe",
    "userName":"Bob.Doe",
    "commercials": [
      {
        "userName":"Soccer F.C"
      }
    ]
  }
}
```

Listing 3.3: Darstellung eines in JSON serialisierten Objektgraphen.

Demgegenüber verwenden Technologien wie CORBA proprietäre, binäre Serialisierungsformate, die CORBA spezifisch sind. Auch Java selbst bietet ein binäres Serialisierungsformat, welches vom Programmierer verwendet werden kann, aber Nachteile in Bezug auf die Deserialisierung unterschiedlicher Versionen, der Datenmigration und der Plattformunabhängigkeit mit sich bringt, an. Im Rahmen dieser Arbeit werden XML und JSON zur Serialisierung aufgrund ihrer Lesbarkeit und ihrer Plattformunabhängigkeit eingesetzt. Daneben existieren Protocol Buffers [Pro15], Thrift [Thr16] und Apache Avro [Avr16], die eine binäre Serialisierung vornehmen und im Wesentlichen Plattformunabhängig sind.

In [MLG+10] wurde gezeigt, dass Protocol Buffers im Vergleich zu XML und SOAP die Größe der versendeten Nachrichten um bis zu 75%, als auch die Verarbeitungszeit um bis zu 59% reduzieren. Auch wenn diese Ergebnisse in einem spezifischen Anwendungsfall erzielt wurden, lässt sich verallgemeinernd davon ausgehen, dass binäre Serialisierungen effizienter sind. Dies zeigen auch die Ergebnisse in [Mae12]. Die Plattformunabhängigkeit bringt aber gleichzeitig auch die Notwendigkeit der Erstellung eines Schemas mit, wodurch zusätzlicher Erstellungs- aber auch Wartungsaufwand entsteht. Auf Basis dieser externen Datenstruktur wird die benötigte Serialisierungs- und Deserialisierungsinfrastruktur für unterschiedliche Plattformen generiert. Dazu zeigt Listing 3.4 ein exemplarisches Protocol Buffer Schema für die Serialisierung von Objektgraphen, die zu dem Klassendiagramm aus Listing 2.16 konform sind. Dazu wird jede Klasse als sogenannte `message` definiert. Innerhalb einer solchen Nachricht können Attribute als `required` oder `optional` definiert werden. Dies legt fest, ob die jeweiligen Attribute bei der Serialisierung belegt sein müssen oder nicht. Darüber hinaus werden Mengen von assoziierten Objekten als `repeated` markiert. Zudem werden die einzelnen Attribute nummeriert, wodurch eine Versionierung unterstützt wird. Dies erschwert allerdings die Wartbarkeit und Evolution solcher Schemas. Eine weiterführende Diskussion und Un-

terscheidung der unterschiedlichen Paradigmen ist in [Mae12] gegeben. Die Verwendung von Protocol Buffers stellt eine Erweiterung im Rahmen dieser Arbeit dar.

```
message Person{                                                    Proto
  required String realName = 1;                                    ...
  optional String userName = 2;
  repeated Commercial commercials = 3;
}

message Commercial {
  optional String userName = 1;
}
```

Listing 3.4: Exemplarisches Protocol Buffers Schema für einen Auszug des Klassendiagramms aus Listing 2.16

Neben den unterschiedlichen Paradigmen existieren aber auch Unterschiede in der eigentlichen Implementierung der verschiedenen Herangehensweisen. Im Java-Umfeld existieren beispielsweise zwei bekannte Implementierungen zur Serialisierung und Deserialisierung von Objektgraphen in JSON Syntax: Jackson [Jac16] und GSON [GSO16].

Bei der Serialisierung unterscheiden sich beide darin, dass Jackson die Serialisierung zyklischer Objektgraphen erlaubt. Dies wird dadurch ermöglicht, dass GSON Assoziationen zwischen Objekten dahingehend auflöst, dass sie in das Startobjekt verschachtelt werden. Jackson hingegen beschreibt alle Objekte auf der gleichen Hierarchiestufe und erstellt die Verknüpfungen auf Basis symbolischer Links zwischen diesen. Dies führt dazu, dass GSON keine Objektgraphen, die einen Zykel enthalten, serialisieren kann. Weiterhin führt dies dazu, das ein Objekt, dass zwei eingehende Links hat, zu zwei verschiedenen Objekten serialisiert und später auch wieder deserialisiert wird und somit nicht mehr dem Ausgangsobjektgraphen entspricht. Zudem führen gerade diese Unterschiede in der Implementierung dazu, dass die Unabhängigkeit der Serialisierungs- und der Deserialisierungstechnologie nicht mehr vollständig gewährleistet wird, da sie Designentscheidungen auf die gleiche Art und Weise abbilden müssen und sonst nicht kompatibel sind. Aus diesen Gründen wurde sich im Rahmen dieser Arbeit zur Verwendung von Jackson entschieden.

Im Rahmen dieser Arbeit werden in Kapitel 8 Generatoren vorgestellt, die die Generierung unterschiedlicher Fassaden ermöglichen. Dabei kann die Fassade als Webservice- oder aber als RPC-Fassade umgesetzt werden. Als Serialisierungsformat wird dabei ein binäres Format oder XML sowie JSON verwendet. Beide Serialisierungsformate wurden gewählt, da sie einen etablierten Kommunikationsstandard, der interoperabel und plattformunabhängig ist, darstellen. Zur Umsetzung wird auf bestehende Serialisierungsbibliotheken, wie Jackson und JAX-WS oder JAX-RS, zurückgegriffen. Die Verwendung von Protocol Buffers wird als Erweiterungsmöglichkeit offen gelassen. Darüber hinaus werden Objektdiagramme als textuelles Serialisierungsformat zur Evolution persistenter

Daten verwendet. Wie zuvor begründet, resultiert dies aus der Möglichkeit der Wiederverwendbarkeit der Evolutionsmechanismen beim modellbasierten Testen der UML/P. Die Evolution persistenter Daten wird in Kapitel 9 vorgestellt.

Im folgenden Abschnitt 3.2.2 werden die technischen Grundlagen der Kommunikation mit einer Datenbank vorgestellt. Neben den in diesem und im nächsten Abschnitt vorgestellten technischen Grundlagen werden in Abschnitt 3.3 verbreitete Muster zur Implementierung von Enterprise Applikationen vorgestellt.

3.2.2 Datenbankkommunikation

Neben der Kommunikation des Applikationsservers mit verschiedenen Clients muss dieser auch mit einer Datenbank kommunizieren. Die bereits vorgestellte Abbildung 3.1 zeigt die typische Architektur einer JEE Anwendung. Die unterste Schicht ist der *EIS Tier*. Hier sind die Datenbanken lokalisiert. Darüber liegt der in Abbildung 3.1 als JavaEE-Server dargestellte, Applikationsserver. Dieser Server besteht aus den drei zuvor erwähnten Schichten. Der *Business Tier* beinhaltet die Datenquellschicht und die Domänenschicht, während der *Web Tier* die Präsentationsschicht darstellt. Vorangegangen wurde die Kommunikation des *Client Tier* mit dem *Web Tier* beschrieben. In diesem Kapitel wird die Kommunikation des *Business Tier* mit dem *EIS Tier* beschrieben, also die Kommunikation der Geschäftslogik mit der Datenbank.

Dazu werden die unterschiedlichen Datenbankarten, die Kommunikation sowie Frameworks, die auf Seiten des Applikationsservers verwendet werden, kurz vorgestellt. Zur persistenten Speicherung der anfallenden Daten können diese in verschiedenen Datenbanken abgespeichert werden. Im Wesentlichen lassen sich objektorientierte und relationale Datenbanken unterscheiden [KE11]. Zudem existieren dateibasierte Datenbanken, wie Hierarchical Data Format 5 (HDF5), oder Not only SQL (NoSQL) Datenbanken, wie Apache Cassandra oder MongoDB [RW12]. Vor allem letztere gewinnen in letzter Zeit immer mehr an Bedeutung, da im Zuge der gegenwärtigen Digitalisierung BigData Ansätze immer wichtiger werden.

NoSQL Datenbanken eignen sich besonders zur Verarbeitung großer Datenmengen, die häufigen Lese- und Schreibzugriffen unterliegen. Die Effizienz dieser Datenbanken wird durch eine Relaxierung der Konsistenzbedingungen an die Datenbank und durch die Auslassung starrer Strukturen im Sinne des CAP-Theorems [Bre12] erreicht. Daher besitzen NoSQL Datenbanken auch keine Relationen wie relationale Datenbanken. Relationale Datenbanken eignen sich besonders, um strukturierte Daten zu speichern und diese durch Wahrung des ACID-Prinzips [HR83] konsistent vorzuhalten. Sie organisieren Daten in Tabellen und ermöglichen effiziente Joins zwischen einzelnen Tabellen, welche bei NoSQL Datenbanken sehr kostenintensiv sind. Die Möglichkeit zur Speicherung strukturierter, relationaler Daten hat relationale Datenbanksysteme bis zum Aufkommen von NoSQL Datenbanken zum de facto Standard gemacht. Durch die steigende Datenmenge wird ein hybrider Ansatz, der unstrukturierte Massendaten sowie strukturierte Daten in den jeweiligen am Besten geeigneten Datenbanksystemen speichert und Verbindungen zwischen diesen anbietet, favorisiert.

Gleichzeitig folgen die meisten Systeme einem objektorientierten Paradigma. Dies hat

zur Folge, dass die Objekte und Objektgraphen auf die Relationen der Datenbank abgebildet werden müssen. Hierzu wurden objektorientierte Datenbanken entwickelt, die sich aber nicht durchgesetzt haben. Im Rahmen dieser Arbeit wird PostgreSQL als relationale Datenbank verwendet. Diese wurde aufgrund der hohen Verfügbarkeit gewählt. Dabei wurde darauf geachtet, dass keine PostgreSQL spezifischen Funktionalitäten verwendet werden, so dass die Datenbank auch gegen ein Oracle Pendant ausgetauscht werden kann. Zudem werden Erweiterungspunkte aufgezeigt, die die Integration von NoSQL Datenbanken erlauben.

Zur Kommunikation mit einer Datenbank wird im Java-Umfeld die *Java Database Connectivity (JDBC)* API eingesetzt. JDBC stellt die auf Java basierende Abstraktion der Kommunikation mit Datenbanken dar, deren Verwendung sich in der Industrie durchgesetzt hat. Im Rahmen dieser Arbeit wird JDBC verwendet. Diese API wird von unterschiedlichen Treibern implementiert. Diese Treiber sind technologie- und herstellerspezifisch. Die Java-Applikation verwendet lediglich die API und muss sich somit nicht um Spezifika kümmern. JDBC öffnet eine Verbindung zum Datenbankserver, führt Operationen aus und schließt die Verbindung wieder. Zur Skalierung kann JDBC auch gleichzeitig mehrere Verbindungen, die in einem sogenannten *Connection Pool* organisiert sind und verwaltet werden, öffnen.

Der Applikationsserver verwendet JDBC zur Kommunikation mit dem Datenbankserver. Da JDBC aber hauptsächlich das Management der Verbindungen übernimmt und Operationen nur an das Datenbanksystem weiterleitet, verwendet der Applikationsserver ein Persistenz Framework, das Transaktionskontrolle, Persistenzmanagement und das objektrelationale Mapping übernimmt. Dieses Framework, im folgenden Object-Relational Mapper (ORM) genannt, implementiert die JPA, die einen Java-Standard für die Persistenz von Daten beschreibt und daher im Rahmen dieser Arbeit verwendet wird. Die bekanntesten Implementierungen sind Hibernate [Hib16] und Toplink Essentials [Top16]. Im Rahmen dieser Arbeit wird Hibernate exemplarisch verwendet. Es wurde darauf geachtet, dass keine spezifischen Hibernate Funktionalitäten verwendet wurden, sondern lediglich solche, die durch die JPA Spezifikation vorgegeben werden. Somit kann Hibernate einfach gegen Toplink ausgetauscht werden. Die Aufgabe des ORMs ist es, die Objekte des Systems auf die Relationen der Datenbank abzubilden und die notwendigen SQL-Operationen mit Hilfe der JDBC-Verbindung an den Datenbankserver zu kommunizieren. Für den Nutzer des Systems ist dies weitestgehend transparent, da diese Funktionalität in einem sogenannten *Entity Manager* gekapselt ist. Dieser bietet typische *create*, *read*, *update* und *delete* (CRUD) Funktionalität an. Der Nutzer übergibt dem *Entity Manager* einen Objektgraph, der persistiert werden soll. Der *Entity Manager* erzeugt die benötigten SQL-Operationen und übernimmt das Transaktionsmanagement. Dennoch benötigt der ORM weitere Informationen zur Konfiguration und zur Abbildung des Objektgraphen auf die Tabellen der Datenbank. Dazu kann Hibernate auf zwei verschiedene Arten konfiguriert werden: mittels einer externen XML-Datei oder mittels Java-Annotationen im Quellcode, die ebenfalls durch die JPA definiert und von Hibernate zum Teil erweitert werden. Im Rahmen dieser Arbeit wird die Konfiguration mit Hilfe von Annotationen verwendet. Mit Hilfe dieser Annotationen ist es dem ORM möglich,

Objektgraphen auf die Relationen der Datenbank abzubilden. Auch objektorientierte Konzepte, wie Vererbung, können so abgebildet werden. Eine Übersicht aller von der JPA definierten Annotationen findet sich in [JPA16]. Die durch Hibernate zusätzlich hinzugefügten Annotationen finden sich in [Hib16]. Eine Diskussion aller möglichen Annotationen würde den Rahmen der vorliegenden Arbeit übersteigen, so dass sich hier auf die wesentlichen Annotationen beschränkt wird. Generell haben die Annotationen, aber auch die externe XML-Datei das Problem, dass alle Angaben auf Strings basieren. Dies bedeutet, dass keine statische Analyse ermöglicht wird. Im Wesentlichen sind die Annotationen und ihre Einträge ein Freitext, der vom Entwickler semantisch richtig ausgefüllt werden muss. Um dieses Problem zu verringern, wird im Rahmen dieser Arbeit die Möglichkeit der Konfiguration des ORMs auf die Modellebene gehoben. Dazu wird in Abschnitt 4.3 eine Tagdefinitionssprache vorgestellt, die eine abstrakte Modellierung dieser Annotationen und eine Konsistenzsicherung in Form einer statischen Analyse ermöglicht. Sie basiert dazu auf einem Tagschema, welches MontiEE spezifische Tagtypen definiert. Dieses Schema wird in Abschnitt 7.2 vorgestellt. Dort wird gezeigt, wie von den JPA-Annotationen abstrahiert wird und welche im Rahmen von MontiEE umgesetzt wurden. Auch wird an dieser Stelle eine detaillierte Diskussion der Semantik der Annotationen vorgenommen.

3.3 Implementierung von Enterprise Applikationen

Bei der Implementierung von Enterprise Applikationen werden verschiedene Komponenten verwendet. Da im Rahmen dieser Arbeit die JEE, die einen etablierten Industriestandard darstellt, verwendet wird, werden diese Komponenten an dieser Stelle technologiespezifisch erläutert. Die JEE und der Applikationsserver stellen unterschiedliche Funktionalitäten bereit, die die Implementierung einer Enterprise Applikation unterstützen. Diese Funktionalitäten werden innerhalb sogenannter *Enterprise Beans* umgesetzt. Beans sind normale Java-Klassen, die durch spezielle Annotationen als solche markiert werden und somit dem Management des Applikationsservers unterliegen. Die Managementfunktionalitäten sind [KS06]: lose Kopplung, Abhängigkeitsmanagement, Lebenszyklusmanagement, deklarative Containerdienste, Portierbarkeit sowie Skalierung und Zuverlässigkeit.

Die lose Kopplung wird durch die Trennung von Implementierung und Interface sowie die Verwendung von Dependency Injection [Pra09] erreicht. Jede Bean benötigt ein entsprechend markiertes Interface. Gleichzeitig wird das Abhängigkeitsmanagement dadurch unterstützt, dass Beans ihre Abhängigkeiten nicht explizit bei Instanziierung angeben, sondern diese mit Hilfe von Dependency Injection injeziert bekommen. Das Management des Lebenszyklus wird dadurch erreicht, dass der Applikationsserver die Instanziierung und Deinstanziierung von Beans übernimmt. Als Containerdienste werden Transaktionsmanagement, Autorisierung, entfernter Zugriff und Nebenläufigkeitsmanagement angeboten. Portierbarkeit, Skalierung und Zuverlässigkeit werden durch eine mögliche automatische Replikation des Applikationsservers erreicht, auf der die Beans direkt weiterverwendet werden können. Aus diesem Grund kann das Singleton Mus-

ter im Rahmen einer Enterprise Applikation nicht verwendet werden, da das Singleton theoretisch auf unterschiedlichen Instanzen des Applikationsservers mehrfach instanziiert werden kann. Diese zusätzlichen Funktionalitäten werden im Rahmen dieser Arbeit an unterschiedlichen Stellen, wie in Kapiteln 7 und 8 präsentiert, zur Umsetzung des Systems vom generierten Quellcode verwendet.

Im Rahmen des JEE Technologiestacks werden die Kommunikationsfassaden, welche von Clients verwendet werden, auf dem Applikationsserver als sogenannte *Session Beans* umgesetzt. Session Beans haben einen *Session Context*, in dem zusätzliche Informationen über den Client enthalten sind. Dabei unterstützen sie synchrone Kommunikation. *Message-Driven Beans* stellen das asynchrone Pendant dar. Ebenso kann ein Client im Rahmen des JEE Technologiestacks mit Hilfe des *Java Naming and Directory Interface (JNDI)* eine Instanz der Bean, auf der er Methoden aufrufen kann, erhalten. Intern arbeitet der Client immer auf einem vom Applikationsserver bereitgestellten Proxyobjekt, welches zusätzliche Funktionalität integriert. Ein Kommunikationsaufruf beginnt mit dem Aufruf einer Methode und endet mit dem Rückgabewert. In dieser Zeit wird keine weitere Kommunikation ausgeführt. Session Beans können *Stateless* oder *Stateful* innerhalb des JEE Technologiestacks sein. Im Rahmen dieser Arbeit werden *Stateless Session Beans*, wie in Abschnitt 8.4 beschrieben, zur Implementierung der Clientkommunikation verwendet. Im Gegensatz zu *Stateful Session Beans* besitzen *Stateless Session Beans* keinen Zustand und müssen alle Operationen innerhalb eines Methodenaufrufs abgeschlossen haben. Sie sind die häufigste Form der Session Beans [KS06]. Darüber hinaus können Session Beans als *Remote Session Beans* oder *Local Session Beans* umgesetzt werden. Beide Formen von Session Beans werden in Abschnitt 8.4 benötigt. Dies bedeutet, dass Clients Instanzen der Bean auflösen können oder aber die Bean nur innerhalb des Applikationsservers in anderen Komponenten verwendet werden kann. Im Rahmen dieser Arbeit werden die Kommunikationsfassaden für Clients als *Remote Session Beans*, oder alternativ als Webservices umgesetzt. Bei der Umsetzung der Fassaden kommen das *Remote Facade* und das *Data Transfer Object (DTO)* Muster [Fow03] zum Einsatz. Das DTO Pattern wird in Abschnitt 8.2 genauer erläutert. In seinen Grundzügen wird das Muster dazu verwendet, Daten zu Clients zu transferieren und die dabei anfallenden Methodenaufrufe zu minimieren. Innerhalb der DTOs werden zusammengehörige Daten aggregiert oder für einen Client überflüssige Daten ausgelassen. Dadurch lässt sich einerseits die Last der Kommunikationsfassade minimieren, zum anderen aber auch ein für den Client spezifisches Datenmodell so umsetzen, dass dies vom Client gehandhabt werden kann.

Klassen, die die Objekte, die in einer Datenbank persistiert werden sollen, definieren, werden als Entitäten bezeichnet. Sie werden mit unterschiedlichen Annotationen zur Steuerung des ORMs annotiert und ebenfalls vom Applikationsserver verwaltet. Zur persistenten Speicherung von Objekten werden die Java-Klassen als `@Entity` markiert. Dadurch lassen sich diese über den *Entity Manager* in einer Datenbank speichern. Normalerweise existiert für jeden Typ eine Tabelle in der Datenbank, in die dann die Objekte des Typs als Zeilen gespeichert werden. Mit Hilfe der `@Table` Annotation kann der Name der Tabelle und des Schemas konfiguriert werden. Dies erfordert vom Entwickler

Detailwissen der zu Grunde liegenden Datenbank. Daher ist diese Annotation optional, wird aber dennoch zumeist einem von Hibernate automatisiert generierten Namen vorgezogen. Attribute werden als Spalten abgebildet. Entitäten haben dabei immer einen Primärschlüssel und sind nicht zwingend über ihre Objektidentität vergleichbar. Primärschlüssel können zusätzlich mit der `@ID` Annotation ausgezeichnet werden. Links zwischen Objekten werden ebenfalls auf die Tabellen der Datenbank abgebildet. Dazu werden Joinspalten oder Jointabellen verwendet. Dies kann mit Hilfe der `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany` Annotationen konfiguriert werden. Zudem können Queries definiert werden. Diese Queries werden als Annotationen direkt in der Klasse, deren Objekte das Query zurückgibt, definiert. Die Definition erfolgt mit Hilfe der `@Query` Annotation. Diese Annotation ermöglicht die Definition eigener Queries, die auf der Datenbank ausgeführt werden. Die Query Annotation besteht dazu aus einem beliebigen Namen und einer Definition des Queries. Das Query kann entweder in nativem SQL, oder aber in der HQL verfasst sein. Die HQL ist dabei ein an SQL angelehnter Dialekt, welcher von dem Mapping in die Datenbank abstrahiert. Während bei nativem SQL Tabellen- und Spaltennamen im Query angegeben werden müssen, können in der HQL Typ- und Attributnamen verwendet werden. Zudem enthalten die Entitäten keine Anwendungslogik, sondern sind reine Datencontainer, die persistiert werden können. Die Anwendungslogik wird in unterschiedlichen Beans implementiert, die Entitäten erzeugen, lesen, verändern oder löschen können. Im Rahmen dieser Arbeit werden die Entitäten des Applikationsservers generiert. Dies wird in Abschnitt 7.4 vorgestellt.

Ein Muster, welches bei der Implementierung von Enterprise Applikationen und zur transparenten Kommunikation mit der Datenbank verwendet wird, ist das *Data Access Object (DAO)* Muster. Ein DAO kapselt dabei für gewöhnlich die Kommunikation mit dem *Entity Manager*, welcher wiederum mittels JDBC mit der Datenbank kommuniziert. Dabei wird dieses Pattern eingesetzt, um den Zugriff auf die Datenbank technologieunabhängig und austauschbar zu halten. Allerdings wird dadurch verschattet, dass sich hinter dem Aufruf einer Methode des DAOs eine kostspielige Datenbankinteraktion verbirgt. Ein DAO kapselt dabei typischerweise CRUD- und Queryoperationen für die Domänenobjekte. Es empfiehlt sich zudem, die zuvor in der Annotation der Klasse definierten Queries hier zu kapseln. Es sei angemerkt, dass der Entity Manager verfügbare Queries nur anhand ihres Namens, welcher ein beliebiger String ist, identifiziert und ausführt. Kommt es hier also zu Inkonsistenzen, führt dies, weil ein Query nicht ausgeführt werden kann, zu einem Laufzeitfehler der Applikation. Ähnliche Probleme können beim Einsatz reflektiver Techniken auftreten. Die Entwurfsentscheidung, ob ein ”großes” DAO oder mehrere nach dem Prinzip des Separation-of-Concerns getrennte DAOs umgesetzt werden, obliegt dem Systemarchitekten. Generell ist die Implementierung der DAOs eine mechanische Aufgabe, die zu stark ähnlich aussehendem Code führt und sich daher für den Einsatz generativer Ansätze anbietet. Im Rahmen dieser Arbeit werden DAOs in Abschnitt 7.5 näher erläutert.

Die Herausforderung bei der Implementierung einer Enterprise Applikation liegt im komplexen Technologiestack und der Notwendigkeit von Annotationen oder XML-Dokumenten. Dabei gibt es eine Reihe spezifischer Einschränkungen oder Methoden zu beach-

ten, die zur Implementierung der Enterprise Applikation erforderlich sind. Gleichzeitig werden viele Annotationen an unterschiedlichen Stellen, die Referenzen aufeinander beinhalten, benötigt. Diese Referenzen werden über Namen, die frei vergeben werden können, aufgelöst. Dies ist eine große Fehlerquelle. Gleichzeitig ist die Notwendigkeit von Annotationen hoch repetitiv, die Konfiguration des ORM abstrahierbar und die Verwendung von Mustern automatisierbar, so dass dies sehr gut von einem Generator übernommen werden kann. Demgegenüber steht aber im Normalfall die Logik der Enterprise Applikation, die eigentlich implementiert werden soll. Dies rückt häufig durch die komplexe Technologie in den Hintergrund.

Im Rahmen dieser Arbeit wird mit MontiEE ein Framework bereitgestellt, welches durch Modellierungssprachen, aber auch Generatoren von der Technologie abstrahiert und die Entwicklung der Anwendungslogik wieder in den Vordergrund stellt.

3.4 Verwandte Frameworks

Im weiteren Verlauf dieser Arbeit werden zu einem jeweilig bestimmten Thema verwandte Arbeiten und Ansätze in den jeweiligen Kapiteln einzeln präsentiert. In diesem Kapitel werden Frameworks, die der Funktionsweise von MontiEE ähnlich sind, vorgestellt. Dazu werden ihre Gemeinsamkeiten und Unterschiede detailliert erläutert. Die meisten verwandten Frameworks unterstützen die Entwicklung von Webanwendungen mit Fokus auf die GUI. Aus diesem Grund enthalten diese Frameworks Präsentations- und Navigationsmodelle sowie Layoutinformationen, die die GUI und die Abläufe der GUI modellieren. Aus diesen Modellen werden Bestandteile der GUI der Clients generiert. Auf Seite des Servers verwenden diese Frameworks meistens Standards, die nicht konfigurierbar oder durch weitere Modelle veränderbar sind. Dies ist ein grundlegend konzeptioneller Unterschied zu MontiEE. Generell werden die verwandten Frameworks im Hinblick auf die Abstraktion durch Modelle, die benötigten Technologieinformationen, die benötigte Technologiekenntnis, die Möglichkeit zur Modellierung von Rechten und Rollen, die Möglichkeit zur Modellierung von Queries, die Kommunikation mit den Clients sowie die Anpassbarkeit und Erweiterbarkeit vorgestellt. Dabei wird auch die Möglichkeit zur Modellierung einer GUI dargestellt, welche aber aus dem Fokus von MontiEE fällt und somit konzeptionell nicht verglichen werden kann.

Im weiteren Verlauf werden Frameworks zur Modellierung von Webanwendungen vorgestellt und gegen die im Rahmen dieser Arbeit entstandene Umsetzung abgegrenzt.

In [MRV08] werden sieben Referenzmodelle für Webanwendungen vorgestellt, die gleichzeitig eine Klassifikation unterschiedlicher Arten von Webanwendungen bieten: *Document Centric Web Sites*, *Transactional Web Applications*, *Interactive Web Applications*, *Workflow-based Web Applications*, *Collaborative Web Applications*, *Portal-oriented Web Applications* und *Ubiquitous Web Applications*. Dabei fokussieren *Document Centric Web Sites* auf statische HTML Seiten zur Interaktion mit Benutzern. Frameworks, die dies unterstützen, verwenden meist GUI- und Navigationsmodelle. *Transactional Web Applications* zeichnen sich durch eine persistente Datenhaltung, eine effiziente Informationsstruktur und eine strikte Trennung zwischen Daten, Verhalten und GUI aus. *Inter-*

active Web Applications verwenden Prozess- und Nutzerpräferenzmodelle, die einen spezifischen Zuschnitt auf die jeweiligen Präferenzen ermöglichen. *Workflow-based Web Applications* setzen häufig Geschäftsprozessmodelle, Geschäftsregeln, Prozessmodelle, Aktormodelle und Architekturmodelle zur Modellierung der Interaktion ein. Modularität, Rollendefinitionen sowie das Konzept getrennter Arbeitsbereiche werden von *Collaborative Web Applications* eingesetzt. *Portal-oriented Web Applications* stellen eine Integration unterschiedlicher, für einen Nutzer personalisierte, Ressourcen dar. *Ubiquitous Web Applications* zeichnen sich durch eine hohe Adaptierbarkeit an Nutzer, Plattform und Kontext aus. Als Einordung in diese Klassifizierung aus [MRV08] fokussiert MontiEE im Wesentlichen auf *Transactional Web Applications* sowie auf Teile von *Collaborative Web Applications* und *Ubiquitous Web Applications*. MontiEE zeichnet sich ebenfalls durch eine strikte Trennung von Daten, Verhalten und GUI aus, speichert Daten transparent, erlaubt die Definition unterschiedlicher Rollen und bietet eine hohe Adaptierbarkeit.

Zur konkreten Modellierung von Webanwendungen dieser unterschiedlichen Arten existieren eine Reihe von Frameworks mit unterschiedlichen Eigenschaften. Die bekanntesten sind die Frameworks Hera [HBFV03, HSB+08], die Web Semantics Design Method (WSDM) [TCP08], die Object-Oriented Hypermedia Design Method (OOHDM) [SR95, RS08], Netsilon [MSFB05], MontiWIS [RR13, Rei16], WebDSL [Vis08], WebML [BCFM00, CFB00], UML-based Web Engineering (UWE) [KKZB08], Object-Oriented Web Solution (OOWS) [PFPA06] und Eclipse Scout [Zim16]. Eine Übersicht und eine Klassifizierung der Eigenschaften sowie eine Übersicht der Möglichkeiten der einzelnen Frameworks ist in [SRS+08] gegeben. Ein weiterer Ansatz ist der `MERODE` Ansatz [Sno14], der auf die Modellierung von Enterprise Information Systems und nicht auf Enterprise Applikationen oder Webanwendungen, wie in Abschnitt 3.1 unterschieden, fokussiert. Ein Teil dieser Systeme ist das Informationsmodell, welches mit Hilfe dieses Ansatzes modelliert werden kann.

MontiWIS

In [Rei16, RR13] wurde mit MontiWIS ein Werkzeug zur Entwicklung von Web-Informationssystemen entwickelt. Im Rahmen dieser Arbeit wird der Begriff Webanwendungen synonym verwendet. MontiWIS fokussiert dabei auf die schnelle Erstellung von Prototypen. Auf Basis verschiedener Modelle lässt sich mit Hilfe von MontiWIS ein Web-Informationssystem, welches eine auf JavaServer Pages (JSP) basierende Webseite als Client verwendet, generieren.

MontiWIS verwendet UML/P Klassendiagramme zur Definition des Datenmodells des Web-Informationssystems. Darüber hinaus verwendet MontiWIS eine Seitenbeschreibungssprache, deren Modelle der Darstellung clientseitiger Informationen dient, eine Ablaufsteuerungssprache, deren Modelle die Reihenfolge angezeigter Seiten modellieren und eine Applikationssprache, deren Modelle Menüs und Zugriffsrechte auf einzelne Seiten modellieren. Der Fokus liegt dabei immer auf der Darstellung von Informationen im Client. Die verwendete Seitenbeschreibungssprache definiert einzelne Ansichten des Clients auf die vorhandenen Daten. Dabei wird sie in MontiWIS ebenfalls als Sprache zur Definition von Sichten beschrieben, die nicht mit der im Rahmen dieser Arbeit entwickelten

Sichtensprache zu verwechseln ist. MontiWIS verwendet Sichten als visuelle Sichten, also Ansichten, eines Datenbestands. Basierend darauf werden GUI Elemente generiert, die es erlauben, nur den speziellen Ausschnitt der Daten zu betrachten. Aktivitätsdiagramme werden zur Navigation zwischen den modellierten Seiten in MontiWIS verwendet. Hierzu existiert kein direktes Pendant in MontiEE, da die Entwicklung einer GUI nicht im Fokus von MontiEE liegt. Dies kann aber durch handgeschriebene Erweiterungen des Generats, wie in Abschnitt 7.4.2 gezeigt wird, auch in MontiEE umgesetzt werden. Die Applikationssprache von MontiWIS erzeugt Menüs und Zugriffsrechte einzelner Benutzer auf verschiedene Seiten des generierten Clients.

MontiWIS bietet eine ähnliche Abstraktion durch Modelle wie MontiEE. Die Verwendung von Klassendiagrammen ist sehr ähnlich zur Verwendung der Klassendiagramme in MontiEE. MontiWIS verwendet diese als Domänenmodell des Clients, welches sich aber dennoch auf einem Server persistieren lässt. Dabei wird immer eine standardisierte Form der Persistenz verwendet, die nur wenig Konfiguration ermöglicht und sich besonders gut dafür eignet, den generierten Client zu unterstützen. Die serverseitige Verwendung sowie die Möglichkeit zur Einflussnahme auf Kaskadierung, Ladestrategien und Vererbungsmechanismen sind nicht gegeben. Für die schnelle Entwicklung von Prototypen ist dies sehr gut geeignet. Bei der Entwicklung von Enterprise Applikationen im Produktiveinsatz hingegen, ist es unabdingbar, dass feingranulare Entwurfsentscheidungen getroffen werden können. In MontiEE wird dies durch die Definition der Taggingsprachen, die in Abschnitt 4.3 vorgestellt werden, ermöglicht. Darüber hinaus werden in MontiWIS Datentypen, die ebenfalls aus der Verwendung im Client heraus motiviert sind, vordefiniert. Diese sind beispielsweise E-Mail und Password. Basierend auf diesen Datentypen werden von MontiWIS unterschiedliche GUI-Elemente und Validierungen generiert. Dies unterscheidet sich von der Verwendung der Klassendiagramme in MontiEE.

Die von MontiWIS verwendeten Sichten unterscheiden sich semantisch von den Sichten, wie sie in MontiEE verwendet werden. MontiEE verwendet Sichten, wie in Abschnitt 5.2 vorgestellt wird, als Domänenmodelle für Clients und generiert aus Sichten wiederum valide Klassendiagramme. MontiEE verwendet diese Klassendiagramme, wie in Abschnitt 8.2 gezeigt wird, zur Generierung von DTOs, welche zur Kommunikation mit heterogenen Clients dienen. Diese Klassendiagramme könnten von MontiWIS wiederum als normale Klassendiagramme verwendet werden, so dass durch MontiWIS viele verschiedene Clients für einen mit MontiEE entwickelten Server generiert werden könnten, die die DTOs als Domänenmodell verwenden.

Sowohl bei MontiWIS als auch bei MontiEE benötigt der Modellierer keine tiefgreifenden Technologiekenntnisse. Allerdings bietet MontiEE ihm durch die Einführung der Taggingsprachen, wie in Abschnitt 4.3 gezeigt wird, dennoch stärkere Möglichkeiten zur Konfiguration.

Die modellierten Rollen und deren Rechte beziehen sich auf die Anzeige verschiedener Seiten und damit indirekt auf den Datenzugriff. Die von MontiEE zur Verfügung gestellten Sprachen zur Modellierung von Rechten und Rollen, wie sie in Abschnitt 5.3 vorgestellt werden, definieren Zugriffe und CRUD-Funktionalitäten auf Ebene der Datenmodelle und können somit als eine Erweiterung betrachtet werden. Queries sowie die

Kommunikation zwischen Client und Server lassen sich in MontiWIS nicht modellieren. Darüber hinaus kann MontiEE, wie in Abschnitt 7.4.2 gezeigt wird, im Unterschied zu MontiWIS um handgeschriebenen Code erweitert werden.

MontiWIS und MontiEE lassen sich zur Entwicklung von Enterprise Applikationen und ihrer Clients kombinieren. Beide erzeugen Client-Server Applikationen mit jeweils einem unterschiedlichen Fokus. Während MontiWIS dies eher clientseitig beleuchtet, fokussiert MontiEE auf den Applikationsserver und bietet dort eine Vielzahl von Konfigurationsmöglichkeiten zur feingranularen Entwicklung von Enterprise Applikationen und heterogenen Clients wie in Abschnitt 7.2 gezeigt wird.

WebDSL

Ein weiterer Ansatz ist WebDSL [Vis08]. WebDSL ist eine Sprache zur Modellierung von Webanwendungen. Dabei fokussiert WebDSL auf die Entwicklung interaktiver dynamischer Webanwendungen basierend auf einem applikationsspezifischen Datenmodell. Als Sprache steht dem Anwender von WebDSL eine Sprache zur Verfügung, die die zu modellierenden Konzepte, wie Entitäten, enthält. Ebenso enthält diese Sprache aber auch Seitendefinitionen, die von dem generierten Client angezeigt werden [Vis08], Workflow Elemente [HVV08] sowie Elemente zur Steuerung der Zugriffskontrolle [GV08]. Zur Generierung des Quellcodes verwendet WebDSL eine Transformationssprache, die Termersetzung ermöglicht [HKGV10]. Die Zielplattform einer mit WebDSL entwickelten Anwendung ist ähnlich der Zielplattform von MontiEE. Auch hier kommt JEE zum Einsatz. Es werden JPA konforme Entitäten erzeugt und als ORM wird Hibernate verwendet. Für das Frontend werden JavaServer Faces (JSF) verwendet. Die Zielplattform und Teile des Generats von WebDSL Applikationen sind ähnlich zum Ansatz von MontiEE. Es werden Entitäten sowie EJBs generiert. Ebenso sind Konzepte der verwendeten Sprache ähnlich. So existieren auch in WebDSL eingebettete Queries, die in den generierten Quellcode übernommen werden, welche aber keine Kontextbedingungen oder Erweiterbarkeit um andere Sprachen, wie bei MontiEE, bieten. Unterschiedlich hingegen ist, dass WebDSL eine einzige Sprache verwendet, die alle Konzepte enthält. Dies bedeutet, dass ein Modell auch unterschiedliche Eigenschaften modelliert. Innerhalb desselben Modells werden GUI, Persistenz und Zugriffskontrolle modelliert. Gleichzeitig wird zur Modellierung der Persistenz eine Entitätensprache basierend auf dem Entity-Relationship-Modell (ER-Modell) [Che76] verwendet, deren Modelle, [LGOT09] folgend, weniger verständlich als Klassendiagramme sind. Die verwendete Entitätensprache enthält keine Vererbung, keine Interfaces oder Assoziationen [Vis08]. Die GUI-Elemente sowie die Elemente zur Ablaufmodellierung [HVV08] sind nicht im Fokus von MontiEE. Auch die Modellierung der Zugriffskontrolle [GV08] bezieht sich auf die modellierten GUI-Elemente des Clients. Dies liegt wie schon bei MontiWIS erläutert nicht im Fokus von MontiEE. Auch die Art der Codegenerierung durch Termersetzung [HKGV10] ist eine andere als sie in MontiEE verwendet wird. Dies ist keine Entwurfsentscheidung von MontiEE, sondern eine Entscheidung des verwendeten Frameworks MontiCore.

WebML

Ein weiteres Framework zur Modellierung von Webanwendungen ist WebML [BCFM00, CFB00]. WebML umfasst verschiedene Modelle: das Strukturmodell, das Hypertextmodell, das das Kompositionsmodell und das Navigationsmodell beinhaltet, das Präsentationsmodell und das Personalisierungsmodell.

Die Modellierungssprachen von WebML sind graphisch und können als XMI [GDB02] serialisiert werden. Das Strukturmodell definiert das Datenmodell der Webanwendung und bildet die Grundlage dieser. Das Hypertextmodell beinhaltet Elemente, die die spätere GUI und Abläufe innerhalb der GUI im Client definieren. Dazu beinhaltet das Kompositionsmodell Elemente, die beispielsweise der Darstellung einzelner Objekte, mehrerer Objekte, Filter oder Indices dienen. Dabei bietet das graphische `Data Unit` Konstrukt die Möglichkeit eine Sicht auf das Strukturmodell, die nur Teile der enthaltenen Information darstellt, zu definieren. Dies ist mit dem Konzept der Sichten aus MontiWIS, aber nicht mit den Sichten aus MontiEE vergleichbar. Das Navigationsmodell ist mit den Modellen der Aktivitätsdiagrammsprache, wie sie in MontiWIS verwendet werden, vergleichbar. Die Aktivitätsdiagrammsprache dient der Navigation zwischen einzelnen Datensätzen. Das Präsentationsmodell enthält die allgemeingültige Definition der Darstellung der einzelnen Elemente, wohingegen das Personalisierungsmodell dedizierte Elemente, denen nutzersensitive Präferenzen zugeordnet werden können, enthält. Diese Präferenzen definieren nutzersensitive Anpassungen an die Darstellung der Elemente.

Zudem enthält WebML verschiedene vorgefertigte Operationen [BCFM08], wie beispielsweise eine `Login` Operation oder auch Operationen zum Anlegen, Verändern, Löschen oder Lesen eines Elements. Dies entspricht der CRUD-Möglichkeit, die auch von MontiEE angeboten wird. Darüber hinaus bietet WebML die Möglichkeit `Web Service Units` zu modellieren, die die Kommunikation über Webservices ermöglichen [MBC+05]. Auch Persistenzinformationen können zur Modellierung von RIAs modelliert werden [BCFC06]. WebML verwendet einen Applikationsserver, auf dem die Applikation ausgeführt wird. Die Applikation selbst wird mit Hilfe eines Codegenerators erzeugt und auf dem Applikationsserver deployt.

Sowohl MontiEE als auch WebML verwenden Modelle zur Abstraktion. Dabei müssen die Modellierer bei beiden Frameworks keine tiefgreifenden Technologiekenntnisse haben. Allerdings kann MontiEE im Gegensatz zu WebML mit Hilfe der Taggingsprachen um technologiespezifische Informationen angereichert werden. Rechte und Rollen werden von WebML nicht betrachtet. Die Funktionalität von WebML ist stets durch die Anforderungen der Clients motiviert. So existiert keine Möglichkeit, Queries serverseitig zu definieren, sondern mit Hilfe von Filtern können Daten clientseitig ausgeblendet werden. Dennoch werden diese Daten ungefiltert zum Client transferiert, was je nach Art und Menge der Daten kostspielig ist. Auch die `Data Unit` stellt eine clientseitige Ansicht der bereits transferierten Daten dar, wohingegen die DTOs ausschließlich benötigte Daten transferieren. Ein großer Unterschied der beiden Frameworks liegt in der Erweiterbarkeit. WebML verfolgt die Annahme, dass jedes Verhalten und jede Logik innerhalb der Modellierungssprache ausgedrückt werden können und auch müssen. Dies führt dazu, dass es nicht möglich ist, die Geschäftslogik, falls die Konzepte der Model-

lierungssprache nicht ausreichend sind, umzusetzen. Zudem bietet MontiEE eine Reihe von Standardfällen und -verhalten, die bei WebML stets modelliert werden müssen.

Form-Oriented Analysis

Die *Form-Oriented Analysis* wird in [DW05a] vorgestellt. Sie stellt eine Methodik zur Modellierung formularbasierter Websysteme dar. Diese Methodik geht davon aus, dass die modellierten Systeme dem `Submit/Respone` Paradigma folgen. Dieses Paradigma geht davon aus, dass dem Nutzer zu jeder Zeit eine Seite dargestellt wird. Beim Wechsel einer Seite wird vom Client eine Aktion an den Server ausgelöst, welche dort ausgeführt wird und eine Rückmeldung an den Client bewirkt. Auch hier stellt der Client den primären Fokus dar.

Die Methodik basiert auf dem Einsatz unterschiedlicher Modellierungssprachen und unterschiedlicher Modelle zur Modellierung der einzelnen Bestandteile des Websystems. Zur Modellierung des User Interfaces wird eine Seitenmodellierungssprache, die die Struktur sowie die Interaktion des Nutzers mit den Seiten modelliert, eingesetzt. Als Erweiterung dieser Sprache werden sogenannte `Screen Diagrams`, die Skizzen des Seitenlayouts und deren Navigation untereinander beinhalten, verwendet. In sogenannten `Storyboards` [DW05a, DW05b] werden Aktionen des Clients und des Servers basierend auf den beiden zuvor genannten Diagrammen modelliert. Dies ermöglicht die Modellierung unterschiedlicher Nutzungsszenarien aus unterschiedlichen Blickwinkeln und Rollen. Zur Modellierung der Daten werden `User Message Models` verwendet. Sie modellieren die Daten, die mit dem Client ausgetauscht werden. Darüber hinaus werden diese Modelle zur Modellierung einzelner Seiten und deren Aktionen und nicht zur Modellierung des gesamten Datenmodells des Clients verwendet. Zudem werden Informationsmodelle zur Modellierung der Persistenz und *Shared Data Models* zur Modellierung der gemeinsamen Elemente des Informationsmodells und des `User Message Models` benutzt. Zur weiteren Einschränkung von Dialogen wird eine `Dialogue Constraint Language` verwendet. Die Datenmodelle der Applikation werden in der *Parsimonious Data Modeling Language* modelliert. Diese wird zur konzeptionellen Erläuterung des Datenmodells verwendet und lässt sich, [DW05a] folgend, durch Modellierungssprachen wie ER- [Che76] oder UML-Diagramme [OMG15c] ersetzen. Darüber hinaus wurde eine *Data Access Language* zur Modellierung von Queries geschaffen.

Die *Form-Oriented Analysis* besitzt verschiedene Gemeinsamkeiten mit MontiEE, auch wenn der Fokus des Ansatzes wiederum auf dem Client liegt. Die *Form-Oriented Analysis* gibt keine konkrete Modellierungssprache vor, sondern erläutert die Semantik auf Basis verwendeter Konzepte, die bei verschiedenen Modellierungssprachen existieren. Auch hier können UML-Diagramme wie bei MontiEE verwendet werden. Darüber hinaus benötigt der Modellierer keine tiefgehende Kenntnis der verwendeten Technologien. Technologieinformationen lassen sich aber nicht wie bei MontiEE modellieren, sondern werden als Standards vorausgesetzt. Das Verständnis der Modellierung von Rollen ist bei der *Form-Oriented Analysis* eine andere. MontiEE sieht Rechte und Rollen als technische Rollen, die Aktionen auf dem Server ausführen dürfen. Dies wird in Abschnitt 5.3 präsentiert. Die modellierten Rollen der *Form-Oriented Analysis* hingegen beziehen sich

auf domänenspezifische Rollen, die andere Anforderungen an die Informationen und die Darstellung der Daten haben. Queries lassen sich auch bei der *Form-Oriented Analysis* modellieren, werden aber lediglich auf abstrakter Ebene vorgestellt. Die Kommunikation mit den Clients, die *User Message Models* und die *Shared Date Models* zeigen Gemeinsamkeiten zu den von MontiEE bereitgestellten DTOs, die in Abschnitt 8.2 vorgestellt werden. Das Mapping, also die Gemeinsamkeiten zwischen Clientdatenmodell und Persistenzmodell, wird in MontiEE durch die Sichten, die in Abschnitt 5.2 vorgestellt werden, modelliert. Die DTOs selbst stellen das Clientdatenmodell dar und beinhalten Requests, welche die eigentliche Serveraktion auslösen. Dies ist ähnlich zu den Nachrichten, die bei der *Form-Oriented Analysis* verwendet werden. Allerdings liegt der Fokus in MontiEE bei den verschiedenen Arten von Clients, wohingegen die *Form-Oriented Analysis* wieder auf die darzustellenden Seiten eines einzelnen Clients fokussiert. Darüber hinaus lässt sich MontiEE mit handgeschriebenem Code, wie in Abschnitt 7.4.2 gezeigt wird, erweitern.

UWE

Ein weiterer Ansatz, der auf der UML basiert und mit Hilfe von Profilen diese erweitert ist der UWE Ansatz [KKZB08]. Dabei verwendet der UWE Ansatz UML-Modelle zur Modellierung einer Webanwendung und ihrer verschiedenen Bereiche. Die verwendeten UML-Modelle werden von UWE mit Hilfe der UML-Erweiterungsmechanismen um Stereotypen erweitert. UWE verwendet Anforderungs-, Entwurfs- sowie Architektur- und Aspektmodelle, die dazu verwendet werden, einzelne Bereiche, wie Navigation, Inhalt, Ablauf und Darstellung einer Webanwendung zu modellieren.

Für die Anforderungsmodellierung setzt UWE Use-Case-Diagramme, die durch Aktivitätsdiagramme verfeinert werden, ein. Bei den Use-Case-Diagrammen unterscheidet UWE drei verschiedene Arten: Navigations Use-Cases, Prozess Use-Cases und personalisierte Use Cases. Use-Cases zur Navigationsmodellierung und zur Personalisierung werden um verschiedene Stereotypen erweitert. Auch die Aktivitätsdiagramme werden entsprechend erweitert, so dass sie Informationen darüber, ob eine Aktivität mit einer GUI verknüpft ist, eine GUI Navigation darstellt oder ein Query ist, enthalten. Diese Anforderungsmodelle werden in den weiteren Modellen zur Modellierung des Inhalts, der Navigationsstruktur und der Darstellung verfeinert.

Zur Modellierung des Inhalts, der Navigationsstruktur und der Personalisierung verwendet UWE UML-Klassendiagramme. Zur Modellierung der Navigationsstruktur werden die Klassendiagramme um Elemente erweitert, die ein Menü, einen Index oder eine Navigationsklasse darstellen. Zudem können auf Basis dieser Navigationsstruktur Geschäftsprozesse mit Hilfe weiterer Stereotypen abgebildet werden [KKCM04]. Zur Modellierung der Darstellung einer Webanwendung verwendet UWE ein erweitertes Aktivitätsdiagramm, das einzelne Elemente als GUI Elemente auszeichnet. So können `Page`, `Button`, `Image` oder `Form` Elemente modelliert werden. Zur Modellierung verschiedener Aspekte, wie beispielsweise ob ein Nutzer eingeloggt ist oder nicht, werden aspektorientierte Klassendiagramme eingesetzt [Zha05]. Dies ermöglicht die Modellierung von Anpassbarkeit der Webanwendung.

UWE verwendet QVT-Modelltransformationen [Kur08], die, dem PIM-PSM Ansatz der MDA folgend [KWB03], dazu verwendet werden, aus einem unabhängigen ein plattformspezifisches Modell zu erzeugen. Dazu werden vom Anforderungsmodell ausgehend stetig weitere Informationen hinzugefügt. Diese stammen aus Funktions-, Architektur- und Big Picture Modellen und resultieren in einem Integrationsmodell [MKK05]. Plattformspezifische Transformationen, wie eine JEE [KKZB08] oder eine JSF [KKK09] Transformation, transformieren das Integrationsmodell in ein plattformspezifisches Modell. Darüber hinaus bietet UWE mit dem Computer-Aided Software Engineering Tool `ArgoUWE` [KKMZ03] eine Modellierungsumgebung.

Auch der UWE Ansatz fokussiert im Gegensatz zu MontiEE auf die GUI. Darüber hinaus verwenden beide Frameworks Modellierungssprachen zur Abstraktion von Technologiespezifika. Bei beiden Frameworks benötigt der Modellierer keine genaue Technologiekenntnis. Bei MontiEE hingegen kann dies konfiguriert werden und es muss nicht auf Standards zurückgegriffen werden. Der UWE Ansatz ermöglicht die Unterscheidung von Nutzern basierend auf verschiedenen Status. So kann ein eingeloggter von einem nicht eingeloggten Benutzer unterschieden werden. Dies lässt sich in MontiEE mit Hilfe der Fassaden, die die Nutzerprüfung und das Einloggen übernehmen, umsetzen. Dies wird in Abschnitt 8.4 vorgestellt. Darüber hinaus können in MontiEE Rollen, wie in Abschnitt 5.3 gezeigt wird, modelliert werden. Die Modellierung von Queries ist im UWE Ansatz auf verschiedene Modellelemente beschränkt und verwendet eine eigene Querysprache. Hier verwendet MontiEE SQL oder HQL. Auch lässt sich im UWE Ansatz keine Kommunikation modellieren. Die Codegenerierung erfolgt in MontiEE über einen Codegenerator und nicht durch Modelltransformationen. Zudem kann MontiEE mit Hilfe von handgeschriebenem Quellcode, wie in Abschnitt 7.4.2 gezeigt wird, erweitert und angepasst werden.

OOWS

Ein weiterer auf MDA basierender Ansatz ist der der OOWS [PFPA06]. Er stellt ebenfalls ein PIM auf dessen Basis eine Webanwendung generiert werden kann, zur Verfügung. OOWS basiert auf der OO-Method [PIP+97]. Die OO-Method stellt zur Modellierung eines objektorientierten Systems drei Modelle bereit: ein strukturelles, ein dynamisches und ein funktionales Modell. Diese werden von OOWS um ein Benutzermodell, ein Navigationsmodell und ein Präsentationsmodell erweitert. Das Benutzermodell erlaubt die Definition verschiedener Rollen und deren Vererbung untereinander. Dabei verwendet OOWS drei vordefinierte Rollen: anonym, registriert und abstrakt. Von diesen Rollen erben die weiteren modellierten Rollen. Das Navigationsmodell erlaubt die Definition verschiedener Navigationskontexte oder -subsysteme, die von Nutzern verwendet werden können. Sie werden *Abstract Information Units* genannt. Diese grobgranulare Ebene erlaubt die Definition der globalen Navigation auf den Seiten des Systems. Darüber hinaus kann mit Hilfe von Navigationsklassen, die Sichten auf die Klassen des Klassendiagramms darstellen, feingranular die Anzeige von Attributen und Assoziationen modelliert werden. Zudem kann mit Hilfe des Präsentationsmodells die Darstellung der Daten über die Modellierung verschiedener Komponenten einer Seite definiert werden.

Der OOWS Ansatz fokussiert stärker als die anderen Ansätze auf die GUI Modellierung der Clients. Er setzt dazu auf spezifische Modelle, die keine Servertechnologieinformationen enthalten. Die Modellierung der Rollen bezieht sich ebenfalls auf die Verwendung verschiedener Seiten und dargestellter Daten. Allerdings ermöglicht der OOWS Ansatz, wie auch MontiEE die Modellierung von Rollenvererbung, wie in Abschnitt 5.3 gezeigt wird. Der OOWS Ansatz kann auf Ebene der Modelle, aber nicht durch beliebigen handgeschriebenen Quellcode, wie in Abschnitt 7.4.2 gezeigt wird, erweitert werden.

Eclipse Scout

Eclipse Scout [Zim16] stellt ein auf der Eclipseplattform [MLA10] basierendes Framework zur Entwicklung von Geschäftsanwendungen dar. Dabei fokussiert Eclipse Scout auf ähnliche Aspekte wie auch MontiEE. Die grundlegende Architektur ist eine Client-Server Architektur, die viele, heterogene Clients berücksichtigt. Der Argumentation aus [Zim16] folgend bietet Scout einen client- und einen serverseitigen Teil, der sich in eine bestehende Enterprise Applikation integriert. Der serverseitige Teil des Scout Frameworks liegt über der Service- und Datenschicht. Zur Modellierung der GUI können unterschiedliche Technologien, wie Swing [Zuk05], Standard Widget Toolkit (SWT) [HW04a] und Remote Application Platform (RAP) [Lan08] verwendet werden. Ein in die Eclipseplattform integriertes Tool unterstützt beim Entwurf der GUI und ermöglicht die Modellierung einzelner Felder, Dialoge und weiterer GUI-Elemente. Auf technischer Ebene ermöglicht Scout Hotspots [FPR00] zur Anpassung der generierten GUI und zum Hinzufügen von handgeschriebenem Code.

Ebenso lässt sich unterschiedliche Technologie in Scout einbinden. Diese Technologie können Webservices oder unterschiedliche Datenbanken sein. Allerdings bietet Scout an dieser Stelle nur eine API an die jeweilige Technologie aus den Hotspots an. Der Nutzer von Scout muss dennoch Technologiespezifika kennen und beispielsweise SQL-Befehle selbständig schreiben. Hierfür existieren keine Standards. Ebenso muss er in einem vorgefertigten Hotspot die Initialisierung der Datenbank und ihrer Tabellen manuell vornehmen. Auch das Speichern und Laden wird nicht übernommen.

Der Fokus von Scout liegt auf der Erstellung von GUI für Business Applikationen. Dabei ist es sehr gut in die Eclipseplattform integriert und erleichtert mit Hilfe verschiedener Wizards und Plugins [Ble14] die Entwicklung dieser GUI. Auch der serverseitige Teil von Scout kann mit Technologie über Wizards angereichert werden. Dies beschränkt sich auf ein Hinzufügen der entsprechenden Plugins und der Verbindung zu der entsprechenden Technologie. Der Nutzer von Scout muss sich selbst um die Verwendung der Technologie kümmern und auch technologiespezifisches Wissen besitzen. Dies ist ein konzeptioneller Unterschied zu MontiEE, da dort unterschiedliche Modellierungssprachen, die in Kapiteln 4, 5 und 6 vorgestellt werden, von der Technologie abstrahieren und der Nutzer auf die Implementierung der Anwendungslogik fokussieren kann. Auch ist MontiEE nicht an eine konkrete Technologie wie die Eclipseplattform gebunden, sondern lässt sich auf andere Technologien übertragen.

Zusammenfassend existiert eine Vielzahl verschiedener Frameworks zur Modellierung von Webanwendungen. Dabei fokussieren die meisten Frameworks auf die Präsentation

von Daten und Prozessen in Clients. Sie bieten sehr umfassende Modellierungssprachen zur Modellierung der Anzeige der Daten und Prozesse in Clients. Häufig werden Navigations- und Präsentationsmodelle dazu verwendet. Diese Modelle dienen als primäre Informationsquelle und weitere Funktionalitäten basieren auf ihnen. So wird ein Rechte- oder Rollenmodell immer darauf beschränkt, die Anzeige zu regulieren. Auch wird das Datenmodell der Seiten, nicht aber die Kommunikation, berücksichtigt. Für all dies werden immer Standards, die nicht von außen konfigurierbar sind, vorausgesetzt. MontiEE verwendet hierfür Taggingsprachen, die in Abschnitt 4.3 vorgestellt werden.

Auch die Erweiterbarkeit ist bei vielen Frameworks nur mit Hilfe von Framework Elementen gegeben. So müssen bereitgestellte Modellierungskonzepte verwendet werden. Nur wenige Frameworks, wie *Eclipse Scout*, verwenden Hotspots zur Integration von handgeschriebenem Code.

Generell bietet die Generierung solcher Webanwendungen auf Basis verschiedener Standards und getrieben durch die Präsentation von Daten und Prozessen in Clients eine gute Möglichkeit, schnell zu einem lauffähigen und präsentierbaren System zu kommen. Allerdings nimmt die Komplexität des verwendeten Technologiestacks durch eine weitere Verbreitung unterschiedlicher Clients, durch die Verfügbarkeit neuer lösungsorientierter Technologien und durch das Entstehen neuer Plattformen und Services immer weiter zu. Dies bedeutet auch, dass die Anpassbarkeit eines solchen Systems sowie die Integration dieser verschiedenen Technologien immer wichtiger für eine Anwendung eines solchen Frameworks werden. Standardimplementierungen und -konfigurationen reichen zwar für die schnelle Entwicklung eines Prototypen, aber nicht zur Systementwicklung eines erweiterbaren und anpassbaren Systems aus.

3.5 Szenario

In diesem Abschnitt wird ein Szenario vorgestellt, welches im weiteren Verlauf als durchgängiges Anwendungsbeispiel dient und auf dessen Basis Anforderungen definiert werden. Diese Anforderungen umfassen Anforderungen an eine repräsentative Enterprise Applikation, an Entwurfsentscheidungen innerhalb der Umsetzung sowie an SE-Werkzeuge zur Unterstützung des Entwicklungsprozesses. Für die Definition von Anforderungen wird dabei die in der Literatur [Bal10] gängige Definition verwendet:

> *"Anforderungen legen fest, was man von einem Softwaresystem als Eigenschaft erwartet." [Bal10]*

Dabei werden Anforderungen im Allgemeinen in funktionale und nichtfunktionale Anforderungen unterteilt, wobei

> *"[e]ine funktionale Anforderung [. . .] eine vom Softwaresystem oder einer seiner Komponenten bereitzustellende Funktion oder bereitzustellenden Service [festlegt]" [Bal10]*

und

> *"[n]ichtfunktionale Anforderungen (nonfunctional requirements, kurz NFRs), auch Technische Anforderungen oder Quality of Service (QoS) genannt, [...] Aspekte, die typischerweise mehrere oder alle funktionale Anforderungen betreffen bzw. überschneiden (cross-cut)" [Bal10],*

beschreiben. Während die funktionalen Anforderungen (FA) immer festlegen, *was* ein System leisten soll, legen die nichtfunktionalen Anforderungen (NFA) fest, *wie gut* es dies leisten soll. In der Literatur gibt es viele Ansätze einer Unterteilung und Klassifikation dieser Anforderungen [Gli07, Gli08]. Einen Standard gibt die ISO/IEC 9126-1:2001 Norm [ISO01] vor.

Die Definition solcher Anforderungen erfolgt meist umgangssprachlich und ohne formalen Rahmen. Allerdings existieren Ansätze zur modellgetriebenen, formaleren Definition von Anforderungen auf Basis der UML. Der bekannteste Repräsentant ist dabei die UML-Erweiterung SysML [FMS14], die auf der Meta Object Facility (MOF) [OMG15a] basiert. SysML ist mittlerweile als Standard Teil der OMG [OMG15b] geworden. Es kann verwendet werden, um Systementwürfe zu modellieren sowie Anforderungen und Architektur zu verknüpfen [Wei11, Alt12]. Dabei führt SysML eine neue Diagrammart, das Requirement Diagram, sowie verschiedene Stereotypen als UML-Profil ein. Darüber hinaus führt SysML das Konzept von *Views* und *Viewpoints* ein, welche in Abschnitt 5.2 aufgegriffen werden.

Anforderungen können unterschiedliche Rahmenbedingungen besitzen. Diese Rahmenbedingungen können den Anwendungsbereich, die Zielgruppe und die Betriebsbedingungen betreffen oder auch technischer Natur sein. Dies kann nochmals in Rahmenbedingungen der technischen Produktumgebung und in Rahmenbedingungen der Entwicklungsumgebung unterteilt werden.

Im Rahmen dieser Arbeit existieren Anforderungen auf unterschiedlichen Ebenen für verschiedene Zielgruppen. Als generelle Rahmenbedingung ordnet sich diese Arbeit in einen agilen Entwicklungsprozess, der MDD und generative Techniken zur Entwicklung von Systemen einsetzt, ein.

Zur Verdeutlichung der unterschiedlichen Anforderungen werden im Rahmen dieser Arbeit unterschiedliche Rollen, die jeweils in Unterrollen untergliedert sein können, beschrieben. Diese sind Auftraggeber, Produktentwickler und Werkzeugentwickler. Sie werden wie folgt definiert und verwendet:

- Auftraggeber (AG): Auftraggeber lassen das Produkt durch die Produktentwickler entwickeln und stellen das fertiggestellte Produkt für die Nutzer bereit. Sie haben daher Anforderungen an das entwickelte und die Entwicklung des Systems im Hinblick auf Kosten, Effizienz und Änderbarkeit.

- Produktentwickler (PE): Produktentwickler stellen für den Auftraggeber das Produkt fertig. Zur Umsetzung benutzen sie Modellierungssprachen und Codegeneratoren, die ihnen vom Werkzeugentwickler bereitgestellt werden. Produktentwickler modellieren das System, konfigurieren und erweitern den Codegenerator und verwenden Modelle zur Codegenerierung des Systems. Darüber hinaus erweitern sie

zur Anpassung des Systems den generierten Code an geeigneten Stellen. Sie subsumieren dabei sowohl die Rollen Modellierer, Anwendungsanpasser sowie Generatoranpasser. Sie haben daher Anforderungen an die Modellierungswerkzeuge, die Modellierungssprachen, an die Funktionalität des Generators und des generierten Codes sowie an die Erweiterbarkeit des Generators und des generierten Codes.

- Werkzeugentwickler (WE): Werkzeugentwickler entwickeln Modellierungssprachen und Codegeneratoren für die Produktentwickler. Dabei müssen die Codegeneratoren Modelle verschiedener Modellierungssprachen in Code überführen können. Im Rahmen dieser Arbeit subsumieren Werkzeugentwickler die Rollen Sprachfamilienentwickler, Kontextbedingungenentwickler, Syntaxentwickler und Generatorentwickler. Sie haben daher Anforderungen an die Entwicklung der Modellierungssprachen, an deren Integration und an die Generatorentwicklung.

Im Rahmen dieser Arbeit wird das in Abschnitt 2.2 vorgestellte Framework MontiCore verwendet, da es mit seinen Mechanismen zur Spracheinbettung, -vererbung und -aggregation essentielle Funktionen zur Umsetzung der Sprachen von MontiEE bereitstellt. Dieses Framework richtet sich primär an Werkzeugentwickler und erfüllt bereits viele existierende Anforderungen. Es werden daher nur für die Modellierung von Enterprise Applikationen spezifische Anforderungen der Werkzeugentwickler aufgeführt. Der Fokus der Anforderungen liegt auf den Anforderungen der Produktentwickler und des Auftraggebers. Da in dieser Arbeit Werkzeuge und eine Methodik zur Entwicklung von Enterprise Applikationen vorgestellt werden, werden Anforderungen, die sich auf die allgemeine Benutzbarkeit des entwickelten Systems beziehen, nicht aufgeführt, sondern nur solche, die sich auf den technischen Hintergrund einer Klasse von Systemen verallgemeinern lassen.

Im weiteren Verlauf des Kapitels werden zunächst in Abschnitt 3.5.1 Anforderungen an die Benutzung Systems definiert und anschließend in Abschnitt 3.5.2 Anforderungen definiert, die ein Entwicklungsprozess, ein Entwicklungswerkzeug als auch das fertige System in technischer Hinsicht erfüllen sollten.

3.5.1 Benutzung des Systems

Der Auftraggeber möchte zur Erweiterung des Produktportfolios seines Unternehmens ein soziales Netzwerk am Markt einführen. Auf Basis einer solchen Plattform erhofft sich der Auftraggeber eine stärkere Kundenbindung. Für die Umsetzung seiner Wünsche wendet sich der Auftraggeber an den Produktentwickler und stellt ihm die Systemanforderungen aus Sicht eines Nutzers und seiner typischen Anwendungsfällen dar:

Der Nutzer registriert sich zunächst beim System und erstellt sein persönliches Profil. Dabei kann er wählen, ob er ein kommerzielles oder privates Profil erstellen möchte. Das Profil hat einen Namen und ein Passwort, die zur Anmeldung am System genutzt werden.

Hat der Nutzer ein privates Profil erstellt, kann er seinen realen Namen und seine Adresse eintragen. Hat er ein kommerzielles Profil erstellt, kann er wählen, ob er ein `Gold`

oder ein `Platin` Profil anlegen möchte, da davon die Nutzung weiterer Funktionalität im System abhängt.

Daraus lassen sich unterschiedliche Anforderungen an das System ableiten. Dies sind Anforderungen, die den Begriff des Nutzers und des Profils innerhalb des Systems und deren Eigenschaften definieren. Die Anforderungen an das Verständnis eines Nutzers im System lassen sich wie folgt beschreiben:

FA1-AG: Im System besitzt jeder Nutzer ein Profil.

- **FA1.1-AG:** Ein Profil kann privat oder kommerziell sein.
- **FA1.2-AG:** Jedes Profil besitzt einen Namen und ein Passwort.
- **FA1.3-AG:** Bei privaten Profilen kann der reale Name und die Adresse des Nutzers hinzugefügt werden.
- **FA1.4-AG:** Bei kommerziellen Profilen kann gewählt werden, ob es sich um ein `Gold` oder `Platin` Profil handelt.

Darüber hinaus möchte der Auftraggeber, dass Nutzer Posts, die nachträglich verändert und von anderen Nutzern gelesen werden können, erstellen können. Diese Posts können ein Foto beinhalten. Im Rahmen dieses Szenarios wird zunächst nur die Möglichkeit des Erstellen eines Posts betrachtet.

FA2-AG: Nutzer können Posts erstellen.

- **FA2.1-AG:** Posts von Nutzern können von anderen Nutzern gelesen werden.
- **FA2.2-AG:** Posts können nachträglich editiert werden.
- **FA2.3-AG:** Ein Post kann ein Foto beinhalten.

Private Profile sollen zudem eine Freundschaftsbeziehung zu anderen Profilen etablieren und kommerziellen Profilen folgen können, so dass sie deren Posts sofort bemerken. Damit eine solche Freundschafts- oder Folgenbeziehung eingegangen werden kann, sollen die Profile von Nutzern über ihren Namen gefunden werden können. Private Profile generell und befreundete Profile im speziellen sollen dazu über den realen Namen des Nutzers auffindbar sein.

Aus dieser Beschreibung lassen sich Anforderungen an die Verbindung zwischen mehreren Nutzern ableiten. Diese lassen sich wie folgt benennen:

FA3-AG: Nutzer mit privaten Profilen können Beziehungen mit anderen Nutzern eingehen.

- **FA3.1-AG:** Nutzer mit privaten Profilen können mit anderen Nutzern mit privaten Profilen eine Freundschaftsbeziehung eingehen.

FA3.2-AG: Nutzer mit privaten Profilen können Nutzern mit kommerziellen Profilen folgen.

FA4-AG: Es existiert eine Suchfunktion, die Profile von Nutzern findet.

FA4.1-AG: Profile können über ihren Namen gefunden werden.

FA4.2-AG: Private Profile können über den realen Namen des Nutzers gefunden werden.

FA4.3-AG: Befreundete private Profile können über den realen Namen des Nutzers gefunden werden.

Zudem sollen Nutzer Posts, die einen unangebrachten Inhalt haben, einem Moderator melden können. Der Moderator ist ebenfalls ein Nutzer, allerdings mit erweiterten Privilegien. Ihm steht eine erweiterte Funktionalität zur Verfügung. Nach Eingang der Meldung kann sich der Moderator den Post ansehen und löschen. Dann sieht er sich die Profildaten des Erstellers an. Es ist ihm dann möglich, das betroffene Profil wegen wiederholter Verstöße zu löschen.

Im System gibt es also unterschiedliche Arten von Nutzern, die verschiedene Privilegien haben. Dazu lassen sich die Anforderungen wie folgt definieren:

FA5-AG: Im System gibt es Moderatoren und Nutzer.

FA5.1-AG: Posts können einem Moderator gemeldet werden.

FA5.2-AG: Der Moderator kann Posts löschen.

FA5.3-AG: Der Moderator kann Profile löschen.

Neben dem zentralen System, welches über ein nicht näher spezifiziertes Webfrontend zugänglich ist, möchte der Auftraggeber eine mobile Anwendung, welche eine eingeschränkte Funktionalität und weniger Informationen enthält, aber keinesfalls an Effizienz einbüßen darf:

FA6-AG: Das System ist als zentrales System umgesetzt.

FA7-AG: Das System muss mit unterschiedlichen Clients bedient werden können.

FA7.1-AG: Unterschiedliche Clients können eine unterschiedliche Funktionalität bieten.

FA7.2-AG: Die Performanz muss bei allen Clients gleich bleiben.

Um frühzeitig Änderungswünsche einzubringen und Fortschritt zu sehen, ist dem Auftraggeber ein schneller Prototyp sehr wichtig. Dazu möchte er bereits von Beginn an

stark in den Entwicklungsprozess eingebunden werden und ebenfalls, ohne tiefgreifende technische Erfahrung, am Entwurf der Kernelemente und -konzepte des Systems teilhaben. Dennoch ist dem Auftraggeber eine effiziente Entwicklung des Systems im Hinblick auf Kosten, aber auch Zeit sehr wichtig. Um ein zukunftssicheres System zu erhalten, verlangt der Auftraggeber, dass eine Weiterentwicklung des Systems ohne Datenverlust möglich ist. Darüber hinaus muss das gesamte System bei hohen Nutzerzahlen skalieren, angemessene Performanz, Stabilität und Ausfallsicherheit gewährleisten.

Die zeitnahe Entwicklung eines Prototyps sowie die Einbindung in den Entwicklungsprozesses zum Entwurf von Kernelementen und Konzepten ist durch die Rahmenbedingung der Verwendung eines agilen Entwicklungsprozess inhärent gegeben. Darüber hinaus lassen sich daraus unterschiedliche Anforderungen an den Entwicklungsprozess und die eigentliche Entwicklung definieren:

FA8-AG: Das System muss ohne Datenverlust weiterentwickelbar, erweiterbar und wartbar bleiben.

FA9-AG: Das System muss auch bei hohen Nutzerzahlen skalieren. Dies beinhaltet Skalierung in Bezug auf Performanz, Stabilität und Ausfallsicherheit.

Die Anforderungen wurden hier aus Sicht des Auftraggebers beschrieben. Anforderungen an die Benutzbarkeit werden im Rahmen dieser Arbeit ausgelassen. Aus Anforderung FA8-AG ergibt sich eine intensive Nutzung von Modellen und Codegeneratoren im Rahmen einer agilen Entwicklungsmethodik. Dies wird durch die Wahl von MontiCore unterstützt. Im weiteren Verlauf werden diese Anforderungen aufgearbeitet und zu technischen Anforderungen oder Prozessanforderungen verfeinert werden. Diese Anforderungen werden aus Sicht des Produktentwicklers im Folgenden verfeinert und verallgemeinert.

3.5.2 Entwicklung des Systems

Auf Basis des Nutzungsszenarios lassen sich technische Anforderungen, die vom Produktentwickler umgesetzt werden müssen, um den Wünschen des Auftraggebers gerecht zu werden, ableiten. Dieser leitet zunächst die Anforderungen an den Entwicklungsprozess ab. Wie bereits zuvor erläutert, ist eine Rahmenbedingung dieser Arbeit, dass MDD und generative Techniken zur Entwicklung von Systemen in einem agilen Entwicklungsprozess, so dass Prototypen zeitnah entwickelt werden können und der Auftraggeber in den Entwicklungsprozess einbezogen wird [BBB+01], eingesetzt werden. Die Integration des Auftraggebers in den Entwicklungszyklus erfolgt also durch Modelle, die eine geeignete Abstraktion des Systems bieten und als Kommunikationsmedium verwendet werden. Zudem erlaubt der agile Entwicklungsprozess, auf Änderungswünsche reagieren und gleichzeitig effizient einen Prototyp entwickeln zu können. Der Produktentwickler verwendet daher Modellierungssprachen und Codegeneratoren zur Entwicklung des Sys-

tems. Zur Implementierung der Anwendungslogik verwendet er eine GPL, in diesem Fall Java.

Darüber hinaus muss der Entwickler in der Lage sein, verschiedene Sachverhalte mit Hilfe der Modellierungssprachen auszudrücken. Zunächst benötigt er eine Modellierungssprache, welche es erlaubt, die Struktur des Systems zu modellieren. Ferner werden die in Abschnitt 2.3 vorgestellten UML/P Klassendiagramme als Modellierungssprache im Rahmen dieser Arbeit verwendet, da sie sich zur Strukturmodellierung von Systemen etabliert haben. Manche Anforderungen des Auftraggebers lassen sich durch die Modellierung mit Klassendiagrammen erfüllen. Allerdings lassen sich mit Hilfe von Klassendiagrammen nicht alle benötigten, vor allem technische Informationen modellieren. Der Produktentwickler benötigt also weitere Modellierungssprachen, um alle benötigten Sachverhalte auszudrücken. Aus den beiden Anforderungen FA6-AG und FA7-AG ergibt sich, dass eine Client-/Serverarchitektur als generelle Systemarchitektur eingesetzt wird. Dies beinhaltet einen Applikationsserver, eine Datenbank, einen ORM und Schnittstellen zur Kommunikation mit Clients. Zudem müssen die einzelnen Komponenten konfiguriert werden. Darüber hinaus muss der Produktentwickler in der Lage sein, die Kommunikationsschnittstellen zu modellieren.

FA1-PE: Die Architektur des Systems ist eine Client-Server Architektur.

FA2-PE: Anfallende Daten werden in einer Datenbank gespeichert.

FA3-PE: Clients können über verschiedene Schnittstellen mit dem Server kommunizieren.

FA4-PE: Sprachen zur Modellierung von Persistenzinformationen werden benötigt.

FA5-PE: Sprachen zur Modellierung von Kommunikationsschnittstellen werden benötigt.

Aus den Anforderungen FA7-AG, FA7.1-AG und FA7.2-AG ergibt sich die Notwendigkeit zur Modellierung unterschiedlicher Schnittstellen und Funktionalität für verschiedene Clients sowie die Notwendigkeit, unterschiedliche Datenmodelle für Clients zu modellieren, so dass unterschiedliche Funktionalität, aber auch Effizienz erreicht werden kann.

FA6-PE: Für unterschiedliche Clients müssen verschiedene Schnittstellen angeboten werden können.

FA7-PE: Für unterschiedliche Clients müssen unterschiedliche Datenmodelle umgesetzt werden können.

Als weitere Anforderung wurde vom Auftraggeber angeführt, dass es unterschiedliche Rollen (vgl. FA5-AG), die unterschiedliche Rechte haben (vgl. FA5.2-AG und FA5.2-AG), im System gibt.

FA8-PE: Unterschiedliche Rollen müssen definiert werden können.

FA9-PE: Die unterschiedlichen Rollen sollen verschiedene Rechte besitzen.

Zur Unterstützung der Agilität muss das System schnell weiterentwickelt werden können und wartbar bleiben. Darüber hinaus dürfen keine persistenten Daten verloren gehen (vgl. FA8-AG). Damit sich die Weiterentwicklung und Migration in den MDD Entwicklungsprozess integriert, entscheidet der Entwickler, dass die Evolution des Systems und die Datenmigration der in der Datenbank gespeicherten Daten ebenfalls modelliert werden muss.

FA10-PE: Die Evolution des Systems muss effizient durchgeführt werden können.

FA11-PE: Die Migration persistenter Daten muss effizient durchgeführt werden können.

Eine solche effiziente Durchführung einer Evolution und der zugehörigen Datenmigration kann mit Hilfe geeigneter Modellierungs- und Automatisierungstechniken erreicht werden. Neben den Anforderungen an die Modellierung hat der Produktentwickler auch Anforderungen, die den Code betreffen. Aus den zuvor genannten Anforderungen ergibt sich der Einsatz von Modellen und Codegeneratoren. Daraus wiederum ergibt sich, dass der Code generiert wird. Weitere Anforderungen beziehen sich im Wesentlichen auf die Zieltechnologie, gegen die der generierte Code ausgeführt wird. Der Entwickler setzt das System in Java um und möchte daher auch, dass die Zielplattform Java-basiert ist. Die Konfiguration des Systems, wie beispielsweise die URL der Datenbank, zu öffnende Ports oder weitere statische Konfigurationen, möchte der Entwickler spezifizieren. Daraus resultieren die folgenden Anforderungen:

FA12-PE: Der generierte Code muss Java sein.

FA13-PE: Die Zielplattform des Systems ist ein Applikationsserver.

FA14-PE: Benötigte Konfigurationen der Zielplattform müssen angegeben werden können.

FA15-PE: Die Kommunikationsschnittstellen werden generiert.

Gleichzeitig benötigt der Auftraggeber applikationsspezifische Funktionalität, so dass der Entwickler in der Lage sein muss, benutzerdefinierte Funktionalität in den vom Codegenerator generierten Code einzubinden. Dies betrifft vor allem die Umsetzung einer Suchfunktion (vgl. FA4-AG) und die Möglichkeit des Meldens von Posts (vgl. FA5.1-AG). Während die Suchfunktion eher den Charakter einer benutzerdefinierten Datenbankanfrage, die den Clients zur Verfügung gestellt wird, hat, stellt das Melden von Posts Anwendungslogik dar. Bei der Suchfunktion ist es wichtig, dass nach Eigenschaften eines Elements gesucht werden kann (vgl. FA4.3-AG, FA4.2-AG und FA4.1-AG).

FA16-PE: Handgeschriebener Code muss zur Realisierung der Anwendungslogik in den generierten Code integriert werden können.

FA17-PE: Benutzerdefinierte Datenbankanfragen müssen definiert werden können.

Weitere Anforderungen an die Datenbank und die Persistenz resultieren daraus, dass der Entwickler sich zunächst für eine relationale Datenbank entschieden hat, diese aber austauschbar machen möchte und somit Teile der Daten oder den gesamten Datenbestand auf eine andere Datenbankart auslagern können möchte. Gleichzeitig ist dem Entwickler die Kommunikationstechnologie momentan unbekannt, so dass diese nach Bedarf änderbar sein. Zudem möchte er zwar die Persistenz und auch die Kommunikationsschnittstellen mit Hilfe des Modells beeinflussen können, möchte aber auf geeignete Standards zurückgreifen.

FA18-PE: Das gewählte Datenbankparadigma muss austauschbar sein.

FA19-PE: Die Speicherung der Daten muss transparent erfolgen.

FA20-PE: Es müssen geeignete Standards für zu modellierende Informationen existieren.

Im Rahmen dieser Arbeit wurde eine Sprachfamilie, die die oben genannten Anforderungen des Auftraggebers, aber vor allem die des Produktentwicklers umsetzt, entwickelt. Darüber hinaus wurden Generatoren, die die Generierung eines solchen Systems gegen eine spezifische Laufzeitumgebung ermöglichen, entwickelt. Kapitel 4, 5 und 6 stellen dabei die Sprachfamilie vor, während Kapitel 7 und 8 die benötigten Generatoren vorstellen. Im Wesentlichen stellen die Ergebnisse dieser Arbeit die Umsetzung durch den Werkzeugentwickler, der die zuvor genannten Anforderungen erfüllt, dar.

Der Werkzeugentwickler verfeinert dazu die bestehenden technischen Anforderungen in einem kleineren Rahmen, um die Zielplattform deutlicher zu definieren. Da als Zielsprache Java verwendet wird (vgl. FA12-PE), und das gewählte Datenbankparadigma

(vgl. FA18-PE und FA19-PE) austauschbar sein soll, wird die JPA, wie sie bereits in Abschnitt 3.2 kurz vorgestellt wurde, verwendet. Die JPA sieht eine Querysprache, entweder SQL oder HQL, vor (vgl. FA17-PE).

FA1-WE: Die Generatoren sollen JPA konform annotierte Java-Klassen generieren.

FA2-WE: Die unterstützte Querysprache muss die HQL oder die SQL sein.

FA3-WE: Die Daten sollen zunächst in einem RDBMS gespeichert werden.

Um ein möglichst breites Angebot unterschiedlicher Kommunikationsstandards (vgl. FA6-PE) umzusetzen, beschließt der Werkzeugentwickler Webservices oder RPC, die die Daten binär, XML oder JSON serialisiert übertragen, zu nutzen (vgl. FA15-PE).

FA4-WE: Webservices und RPC sollen von den Kommunikationsfassaden unterstützt werden.

FA5-WE: Die Übertragung soll binär, XML oder JSON serialisiert erfolgen.

Neben eigentlichen Systemanforderungen werden an dieser Stelle auch Entwurfsentscheidungen als Anforderungen aufgenommen, die sich bei der Entwicklung solcher Systeme durchgesetzt haben. Dabei handelt es sich um Architekturen, Architekturmuster und Integrationsmuster, die auch in generiertem Code eingehalten werden sollen. Dies gilt vor allem, da im Rahmen von MontiEE zwar ein Großteil der Serveranwendung generiert wird, aber dennoch unterschiedliche Teile durch handgeschriebenen Code ergänzt, oder, wie die Clients, manuell implementiert werden. Damit die Integration und die Entwicklung vereinfacht werden, muss auch das Generat etablierte Muster umsetzen. Als Entwurfsentscheidung sieht der Werkzeugentwickler zur Integration des handgeschriebenen Codes klar definierte Schnittstellen innerhalb der 3-Tier Architecture (vgl. FA1-PE), wie in Abschnitt 3.2 vorgestellt, vor. Dazu definiert er eine klare Schnittstelle von der Kommunikationsfassade zur Anwendungslogik und von der Anwendungslogik zur Persistenzschicht (vgl. FA16-PE).

FA6-WE: Der Generator soll eine Schichtenarchitektur erzeugen.

FA7-WE: Es soll eine klar definierte Schnittstelle zwischen Kommunikationsfassade und Anwendungslogik existieren.

FA8-WE: Es soll eine klar definierte Schnittstelle zwischen Anwendungslogik und Persistenz existieren.

Als weitere Entwurfsentscheidung soll das Generat neben der Systemarchitektur auch Entwurfsmuster umsetzen, die die Kommunikationsfassaden und die Persistenz erweiterbar oder austauschbar machen. Auch zur Steigerung der Effizienz (vgl. FA7.2-AG und FA9-AG) sollen entsprechende Entwurfsmuster eingesetzt werden. Die entsprechenden Entwurfsmuster wurden in Abschnitt 3.3 vorgestellt.

FA9-WE: Zur Kapselung der Persistenzschicht soll das Generat das Data Access Objects Entwurfsmuster umsetzen.

FA10-WE: Zur clientspezifischen Datenübertragung soll das Generat das Data Transfer Object Entwurfsmuster umsetzen.

FA11-WE: Innerhalb der DTOs soll das Command Pattern eingesetzt werden.

3.6 Zusammenfassung

Im Rahmen dieses Kapitels wurden zunächst in Abschnitt 3.1 gängige Definitionen von Enterprise Applikationen, Enterprise Informationsystems und Informationssystemen vorgestellt. Für diese wurden die wichtigsten Merkmale vorgestellt. Auf Basis dieser Merkmale wurde der Begriff der Enterprise Applikation, wie er im Rahmen dieser Arbeit verwendet wird, eingeordnet. Dabei wurde deutlich, dass Enterprise Applikationen im Rahmen dieser Arbeit Daten persistent speichern, Daten mehreren Nutzern parallel zur Verfügung stellen und Kommunikation zwischen Clients und der Enterprise Applikation sowie zwischen mehreren Enterprise Applikationen ermöglichen.

In Abschnitt 3.2 wurde die generelle Architektur von Enterprise Applikationen präsentiert. Dabei wurde eine Schichtenarchitektur vorgestellt, die die Umsetzung von Enterprise Applikationen, wie sie im Rahmen dieser Arbeit verstanden werden, unterstützt. Innerhalb der Architektur wurden die einzelnen Schichten gezeigt. Daran anschließend wurden Techniken und Technologien zur Kommunikation zwischen Server und Clients vorgestellt. Dabei wurden sowohl die Serialisierung und Deserialisierung von Daten wie auch eingesetzte Kommunikationsprotokolle dargelegt. Ferner wurden Techniken und Technologien zur Kommunikation des Servers mit der Datenbank sowie unterschiedliche Datenbankparadigmen präsentiert. Darüber hinaus wurde die Wahl der einzelnen Technologien erläutert und begründet.

In Abschnitt 3.3 wurde auf die Implementierung von Enterprise Applikationen und gängige Entwurfsmuster eingegangen. Abschnitt 3.4 stellte eine Vielzahl verwandter Arbeiten und die Abgrenzung zu MontiEE vor. Die Frameworks `MontiWIS`, `WebDSL`, `WebML`, `Form-Oriented Analysis`, `UWE`, `OOWS` und `Eclipse Scout` wurden detailliert vorgestellt. Dabei wurde jedes Framework einzeln, kurz vorgestellt. Der Fokus lag auf den verwendeten Modellierungssprachen und den resultierenden Funktionalitäten sowie der Erweiterbarkeit des jeweiligen Frameworks.

Abschließend wurde in Abschnitt 3.5 ein Szenario vorgestellt, das im Rahmen dieser Arbeit durchgängig verwendet wird. Auf Basis dieses Szenarios werden Anforderungen erstellt und unterschiedlichen Rollen zugeordnet. Teile der hier vorgestellten Anforderungen werden durch eine geeignete Technologieauswahl unterstützt und vereinfacht. So ermöglicht der Einsatz von MontiCore die Weiterentwicklung, Erweiterung und Wartung eines solchen Systems ohne Datenverlust (vgl. FA8-AG). Darüber hinaus unterstützt MontiCore die Sprachentwicklung und die Umsetzung von Codegeneratoren. Somit wird MontiCore als Werkzeug zur Erfüllung der Anforderungen bezüglich Sprachen und Generierung verwendet. Die Wahl der JEE und des Glassfish Applikationsservers unterstützt eine Client-Server Architektur (vgl. FA1-PE, FA12-PE und FA13-PE). Die Wahl der Datenbank unterstützt die persistente Speicherung anfallender Daten (vgl. FA2-PE). Der Einsatz von Webservices und plattformunabhängiger Serialisierungsformate wie JSON und XML unterstützen unterschiedliche Clients (vgl. FA3-PE, FA5-WE und FA7-AG). Die Wahl der Spezifikation JPA und der Implementierung Hibernate unterstützen die Austauschbarkeit der Datenbank und die transparente Speicherung (vgl. FA18-PE und FA19-PE).

In den nachfolgenden Kapiteln werden zunächst die Sprachen zur Modellierung von Enterprise Applikationen vorgestellt. Dabei fokussiert Kapitel 4 auf Sprachen zur Modellierung der Persistenz, Kapitel 5 auf Sprachen zur Modellierung der Kommunikation und Kapitel 6 auf Sprachen zur Modellierung der Evolution. Daran anschließend werden Generatoren zur Generierung der Persistenz in Kapitel 7, zur Generierung der Kommunikationsinfrastruktur in Kapitel 8 und zur Generierung der Evolutionsinfrastruktur in Kapitel 9 vorgestellt. In Kapitel 10 wird die Verwendung von MontiEE und der Einsatz im Rahmen von Fallstudien gezeigt.

Teil II

Die Sprachfamilie MontiEE

Kapitel 4

Modellierung der Persistenz

Nachdem in den vorangegangenen Kapiteln die Grundlagen sowie die Anforderungen an eine Enterprise Applikation und deren Modellierung und ein Szenario vorgestellt wurden, werden in diesem und im darauffolgenden Kapitel die sich daraus ergebenden Modellierungstechniken, die für die Modellierung von Enterprise Applikationen nötig sind, vorgestellt.

Eine Enterprise Applikation befasst sich, wie in Kapitel 3 vorgestellt, mit der persistenten Speicherung von Daten, die mehreren Nutzern, mit unterschiedlichen Clients zur Verfügung gestellt werden. Diese Daten können strukturierte Daten, die in relationalen Datenbanken abgelegt werden, oder Massendaten, die in hochperformanten Datenbanken abgelegt werden, sein. Gleichzeitig führt die Unterstützung unterschiedlicher Nutzer auch dazu, dass diese unterschiedliche Funktionalität verwenden dürfen, also unterschiedliche Rechte haben. Durch die Multi-User und Multi-Client Eigenschaft müssen Enterprise Applikationen also skalierbar und performant sein. Typischerweise folgen Enterprise Applikationen einer Schichtenarchitektur, die verschiedene Komponenten enthält. Diese beinhalten Kommunikationsschichten zur Kommunikation mit Clients oder der Datenbank und eine Logikschicht, in der die Geschäftslogik repräsentiert ist. Gleichzeitig werden Daten durch die Schichten vom Client zur Datenbank und vice versa gereicht.

Das entwickelte Framework MontiEE stellt Hilfsmittel zur Modellierung der einzelnen Komponenten einer Enterprise Applikation dar. Dabei verwendet es Modellierungssprachen und Generatoren. Mit Hilfe der Sprachen kann der Produktentwickler eine Enterprise Applikation abstrakt und technologieagnostisch beschreiben und durch die Generatoren den Quellcode erzeugen. Die Methodik zur Verwendung von MontiEE wird in Kapitel 10 im Detail vorgestellt. Dennoch wird zum besseren Verständnis die Methodik der Verwendung kurz umrissen. MontiEE ist darauf ausgelegt, dass das Domänenmodell des Applikationsservers mit Hilfe eines Klassendiagramms, wie in Abschnitt 4.2 präsentiert, strukturell beschrieben wird. Auf dem Domänenmodell aufbauend kann der Produktentwickler mit Hilfe einer Tagdefinition, wie sie in Abschnitt 4.3.1 beschrieben wird, das Klassendiagramm mit technologiespezifischen Persistenzinformationen anreichern. Die Tagdefinition folgt dabei einem definierten Schema, das in Abschnitt 7.2 präsentiert wird. Die zugehörige Tagschemasprache wird in Abschnitt 4.3.2 vorgestellt. Auf Basis dieser Modelle werden Entitäten und weitere serverspezifische Strukturen, wie in Kapitel 7 dargelegt, generiert. Hat der Produktentwickler das Klassendiagramm und die Tagdefinition modelliert, so kann er clientspezifische Sichten auf das vollständige Domänenmodell des Servers modellieren. Dies ist notwendig, da die Clients in ihrer Kapazität

und Geschwindigkeit sehr heterogen sein können. Durch eine Sicht kann ein reduziertes und dennoch auf dem eigentlichen Domänenmodell basierendes Modell modelliert werden. Eine Reduktion des Domänenmodells ermöglicht es, nur bestimmte Daten und vor allem eventuelle Massendaten nicht bei jeder Übertragung zu kommunizieren. Die Sichtensprache wird in Kapitel 5 vorgestellt. Auf Basis dieser Sichten werden clientspezifische Strukturen generiert, die in Kapitel 8 vorgestellt werden. Darüber hinaus kann der Produktentwickler mit Hilfe der Rechte-, Rollen- und Mappingsprachen die Rollen des Systems und deren erlaubte Funktionalitäten modellieren. Diese Sprachen werden in Kapitel 5 vorgestellt. Auf Basis dieser Modelle können Zugriffsfassaden generiert werden, die bestimmten Rollen bestimmte Rechte einräumen. Dies wird in Kapitel 8 vorgestellt. Darüber hinaus ist es dem Produktentwickler möglich das deployte und laufende System mit Hilfe der Deltasprache, die in Kapitel 6 vorgestellt wird, weiterzuentwickeln. Auf Basis der modellierten Deltas wird sowohl eine Modellevolution als auch eine Datenmigration durchgeführt, die in Kapitel 9 vorgestellt wird.

Der Fokus dieses Kapitels liegt auf der Anreicherung von Modellen mit technischen Informationen. Dies kann der Produktentwickler mit Hilfe der in diesem Kapitel vorgestellten Tagdefinitionssprache erreichen. Durch die ebenfalls umgesetzte Tagschemasprache kann der Werkzeugentwickler die Modellierungsmöglichkeiten des Produktenwicklers schematisch vorgeben. Dieses und die nachfolgenden Kapitel stellen die Sprachfamilie MontiEE detailliert vor.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Eine allgemeine Methodik zur Ableitung sprachspezifischer Tagschema- und Tagdefinitionssprachen aus beliebigen DSLs, die auf Klassendiagramme zur Modellierung von Persistenzinformationen angewendet wird.
- Vorstellung klassendiagrammspezifischer Tagschema- und Tagdefinitionssprachen zur Modellierung von Persistenzinformationen.
- Vorstellung von Kontextbedingungen zur Konsistenzsicherung zwischen Tagdefinition, die Persistenzinformationen enthält und dem getaggten Domänenmodell.

In Kapitel 5 und 6 stehen Sprachen zur Modellierung der Kommunikation und der Evolution von Enterprise Applikationen im Vordergrund. Zunächst wird mit einem Überblick dieses Kapitels begonnen.

4.1 Überblick

Abbildung 4.1 zeigt die generellen Zusammenhänge zwischen den Modellen und dem jeweiligen Verwendungszweck. Dazu sind die Modelle zur Modellierung der Kommunikation, wie sie in Kapitel 5 vorgestellt werden, und die Modelle zur Modellierung der Evolution, wie sie in Kapitel 6 vorgestellt werden, abgebildet. Die Modelle zur Modellierung der Persistenz sind hervorgehoben. Im Rahmen dieser Arbeit werden Klassendiagramme (CDs) basierend auf einer Grammatik für Klassendiagramme (CD-Grammar) verwendet. Wie zuvor erwähnt dienen diese zur Modellierung des Domänenmodells des Systems

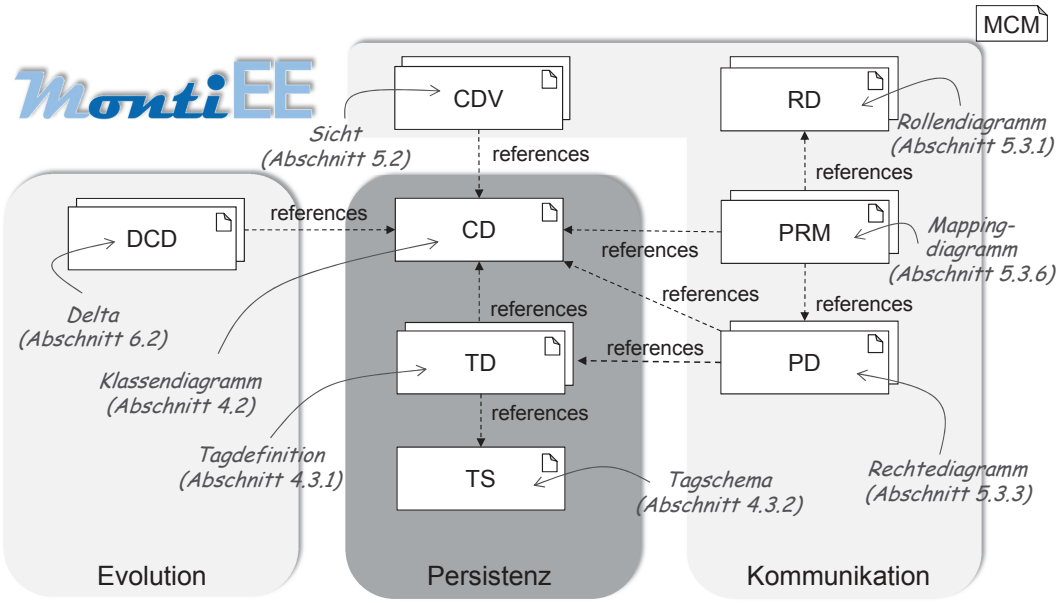


Abbildung 4.1: Überblick über die Modelle der Sprachfamilie MontiEE und deren Zusammenhänge. Der Fokus liegt dabei auf den Modellen zur Modellierung der Persistenz von Enterprise Applikationen.

auf Basis der textuellen UML/P Fassung. Ein exemplarisches Klassendiagramm, das die Applikationsdomäne des im vorangegangenen Szenario vorgestellten sozialen Netzwerks modelliert, wird in Abschnitt 4.2 vorgestellt.

Die zur Anreicherung des Klassendiagramms mit weiteren, technologiespezifischen Informationen geschaffene klassendiagrammspezifische Tagdefinitionssprache (TD-Grammar) ermöglicht die Modellierung von Persistenzinformationen. Im Rahmen dieser Arbeit werden Modelle dieser Sprache als Tagdefinition (TD) bezeichnet. Zudem referenzieren Tagdefinitionen Tagschemas. Im Rahmen dieser Arbeit wurde eine Tagschemasprache geschaffen, deren Modelle als Tagschema (TS) bezeichnet sind. Sie sind durch die Grammatik der Tagschemasprache (TS-Grammar) definiert. Die referenzierten Tagschemas sind Instanzen einer klassendiagrammspezifischen Tagschemasprache, die es erlaubt, mögliche Tags, die in einer Tagdefinition verwendet werden können, zu definieren. Dies ist analog zur Beziehung zwischen XML-Modellen und deren XSD-Schema. Die Tagschemasprache wurde zur Erreichung von Typsicherheit und zur Konformität von Tagdefinitionen geschaffen. Beide Taggingsprachen werden in Abschnitt 4.3 vorgestellt. Darüber hinaus wird eine allgemeine, im Rahmen dieser Arbeit entwickelte Methodik [GLRR15] zum Entwurf beliebiger sprachspezifischer Taggingsprachen, auf deren Basis auch beide klassendiagrammspezifischen Taggingsprachen entwickelt wurden, vorgestellt. Die Verwendung der Modelle als Eingabe für Codegeneratoren wird in Kapitel 7 detailliert erläutert.

Neben den Taggingsprachen wurde im Rahmen dieser Arbeit eine Sprache zur Modellierung von Sichten auf Klassendiagramme, die in Abschnitt 5.2 präsentiert wird, entwi-

ckelt. Wie zuvor erläutert, werden diese Sichten benötigt, um spezifische, auf einen Client zugeschnittene Domänenmodelle modellieren zu können. Die entwickelte Sprache wird im Rahmen dieser Arbeit Sichtensprache genannt. Modelle der Sichtensprache werden klassendiagrammartige Sicht (CDV) genannt. Auch ihnen liegt eine definierende Grammatik (CDV-Grammar) zu Grunde. Dabei referenzieren Elemente der Sicht Elemente des Klassendiagramms, um sie in die modellierte Sicht aufzunehmen. Eine automatisierte Transformation einer Sicht in ein reduziertes Klassendiagramm wird in Abschnitt 5.2.3 beschrieben. Durch die Transformation in ein Klassendiagramm lassen sich die umgesetzten Generatoren, die Klassendiagramme als Eingabe erhalten, wiederverwenden.

Zusätzlich wurden Sprachen zur Modellierung von Rechten und Rollen und der Zuordnung zwischen diesen, die in Abschnitt 5.3 präsentiert werden, im Rahmen dieser Arbeit entwickelt. Der Server bietet den Clients Funktionalität als Webservice oder Remote Bean an. Jedoch nicht jeder Client oder jeder Benutzer hat dabei die gleichen Rechte. Die modellierten Rechte und Rollen werden für die Zugriffskontrolle auf vom Server angebotene Methoden verwendet. Die entwickelten Sprachen werden im Rahmen dieser Arbeit als Rollen-, Rechte- und Mappingsprache bezeichnet. Abbildung 4.1 zeigt die zugehörigen Rollendiagramme (RD), Rechtediagramme (PD) und Mappingdiagramme (PRM). Sie werden ebenfalls durch Grammatiken (RD-Grammar, PD-Grammar und PRM-Grammar) definiert. Rollendiagramme enthalten die Definition der im System vorhandenen Rollen. Rechtediagramme enthalten die im System definierten Rechte. Die Zuordnung von Rechten zu Rollen erfolgt in dem Mappingdiagramm. Rechtediagramme referenzieren das Klassendiagramm, da sie die Rechte in Bezug auf das Klassendiagramm spezifizieren. Mappingdiagramme referenzieren wegen der enthaltenen Zuordnung Rechte und Rollen und referenzieren ebenfalls das Klassendiagramm zur Konsistenzprüfung. An dieser Stelle wird eine automatisierte Ableitung von Rechtediagrammen beschrieben. Die Verwendung zur Codegenerierung wird in Kapitel 8 beschrieben.

Darüber hinaus wird eine Sprache zur Modellierung der Evolution eines Klassendiagramms beschrieben. Diese Sprache wird im Rahmen dieser Arbeit als Deltasprache bezeichnet. Ihre Modelle, Deltas (DCD) genannt, sind in Abbildung 4.1 dargestellt und werden in Kapitel 6 vorgestellt. Mit Hilfe dieser Sprache lassen sich Veränderungen am Domänenmodell modellieren. In Abschnitt 6.2 wird die Definition dieser Sprache detailliert vorgestellt. Diese Modelle werden zur Systemevolution aber auch zur Datenmigration des laufenden Systems verwendet. Die teilautomatisierte Migration vorhandener Daten sowie die Verwendung für Codegeneratoren wird in Kapitel 9 vorgestellt.

Nachfolgend wird, bevor die einzelnen Sprachen vorgestellt werden, zunächst ein Domänenmodell, welches auf dem in Abschnitt 3.5 vorgestellten Szenario und den Anforderungen dort basiert, erläutert.

4.2 Das Domänenmodell

Zur Modellierung der fachlichen Domäne des Systems werden Klassendiagramme verwendet, da sich diese zur Strukturmodellierung etabliert haben. Dazu wird die Klassendiagrammsprache der UML/P [Rum11, Rum12, Sch12], wie in Abschnitt 2.3 dargestellt, als Grundlage verwendet.

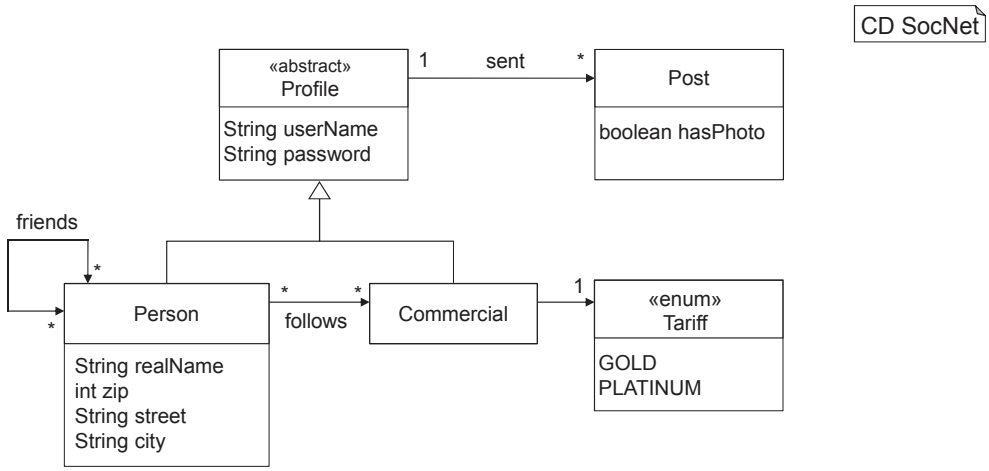


Abbildung 4.2: Klassendiagramm des sozialen Netzwerks auf Basis des Szenarios aus Abschnitt 3.5.

Das gemäß den Anforderungen aus Abschnitt 3.5 modellierte System wird vom Produktentwickler, wie in Abschnitt 3.5 eingeführt, mit einem Klassendiagramm modelliert und ist in Abbildung 4.2 dargestellt. Allerdings erfüllt das Klassendiagramm alleine nicht alle Anforderungen, so dass das Klassendiagramm zur Vorstellung und Erläuterung der verwendeten Sprachen und deren Erweiterungen mit weiteren Modellen ergänzt wird.

Das System besteht aus der abstrakten Klasse `Profile` (vgl. FA1-AG), welche einen `userName` und ein `passwort` besitzt (vgl. FA1.2-AG). Als Subklassen sind die konkreten Klassen `Person` und `Commercial` (vgl. FA1.1-AG) modelliert. Personen stellen die privaten Profile dar und haben einen `realName` sowie die Attribute `zip`, `street` und `city` zur Speicherung der Adresse (vgl. FA1.3-AG). Die Klasse `Commercial` stellt kommerzielle Profile dar. Assoziiert ist die Enumeration `Tariff`, die die Art des kommerziellen Profils angibt (vgl. FA1.4-AG).

Die Klasse `Post` repräsentiert Posts innerhalb des Systems. Die `sent` Assoziation gibt an, dass ein Profil einen Post geschrieben hat (vgl. FA2-AG). Das Attribut `hasPhoto` gibt an, dass der Post ein Foto enthält (vgl. FA2.3-AG). Auf die explizite Modellierung einer Foto Klasse wurde an dieser Stelle aus Gründen der Einfachheit verzichtet.

Die Assoziation `friends` modelliert die Beziehung zwischen zwei privaten Profilen (vgl. FA3.1-AG) und die Assoziation `follows` die Möglichkeit privater Profile, kommerziellen zu folgen (vgl. FA3.2-AG).

Die übrigen Anforderungen werden im weiteren Verlauf durch zusätzliche Modelle und mit Hilfe zusätzlicher Sprachen erfüllt werden. Dazu wird zunächst das Konzept von Taggingsprachen vorgestellt, dass dazu dient, Modelle mit zusätzlichen, meist technologiespezifischen Informationen anzureichern.

4.3 Taggingsprachen

In [GLRR15] wurde eine Methodik, auf der betreuten Vorarbeit [Roi15] basierend, entwickelt, die es erlaubt, zu beliebigen DSLs sprachspezifische Taggingsprachen abzuleiten. Dies dient dazu, Modelle einer DSL, und somit auch die Sprache selbst, nicht mit zusätzlichen Informationen zu belasten, sie lesbar zu halten und gleichzeitig in den unterschiedlichsten Kontexten wiederverwendbar zu machen. Teile von [GLRR15] werden in diesem Kapitel aufgegriffen und verwendet.

Der Begriff Taggingsprachen subsumiert dabei zwei verschiedene Sprachen, eine Tagdefinitionssprache zur Modellierung modellspezifischer Tags und eine Tagschemasprache zur Definition möglicher Tagtypen für eine bestimmte Sprache. Dabei gibt letztere ein spezielles Tagschema, welchem die Modelle der Tagdefinitionssprache folgen müssen, vor. Insgesamt ermöglicht diese Methodik

- die systematische Erstellung einer sprachspezifischen Tagdefinitionssprache
- die systematische Erstellung einer sprachspezifischen Tagschemasprache
- Konformitätsbeziehungen zwischen Tagdefinitionssprache und Tagschemasprache
- eine klare Trennung der Modelle und der Zusatzinformationen, so dass die Wiederverwendbarkeit deutlich erhöht wird.

Im Wesentlichen existieren zwei unterschiedliche Paradigmen, Modelle mit solchen Zusatzinformationen anzureichern: die Anreicherung des Modells mit den benötigten Informationen und die Erstellung eines eigenständigen Artefakts, welches Referenzen zu dem Modell besitzt. Ersteres vermischt Domänenwissen und -informationen mit technischen oder anderweitigen, nicht zur Domäne gehörenden, Informationen. Dadurch werden diese Modelle plattform- und technologiespezifisch und können nicht mehr in anderen Kontexten wiederverwendet werden. Gleichzeitig müssen auch alle im MDD-Prozess beteiligten Rollen, die auf das Modell angewiesen sind, die zusätzlichen Informationen verstehen und können sich nicht mehr auf die wesentlichen Domäneninformationen fokussieren. Dennoch führt die Existenz mehrerer Artefakte sicherlich zu einem größeren Synchronisationsaufwand zwischen den beteiligten Artefakten, falls sich das zu Grunde liegende Domänenmodell ändert.

In [Sel07] werden dazu auf der UML [OMG15c] basierende Ansätze genannt, wovon zwei Ansätze existierende Sprachen erweitern oder verfeinern. Der erste Ansatz beruht darauf, jedes Modellelement mit Stereotypen auszeichnen zu können. Im Wesentlichen ist dies analog zur hier verwendeten Methodik, unterscheidet sich aber dennoch darin, dass der Einsatz von Stereotypen davon abhängt, ob die Ausgangssprache dies auch an den entsprechenden Elementen ermöglicht. In der hier verwendeten Methodik hingegen bleibt die Ausgangssprache unverändert und die zusätzliche Information wird von außen dazu gemischt. Der zweite Ansatz basiert auf der Verwendung von UML-Profilen [OMG15c], die aber UML-spezifisch sind, wohingegen der hier verwendete Ansatz allgemein auf DSLs angewendet werden kann. Dennoch zeigt [Sel07] einen systematischen Ansatz zur

Erstellung dieser. Die in [GLRR15] vorgestellte Methodik ist als systematischer Ansatz damit vergleichbar.

UML-Profile gibt es für unterschiedliche Anwendungsbereiche, wie Webanwendungen [BGP01], Architekturbeschreibungen [FBSG07], Software Produktlinien [ZHJ04] oder eingebettete Echtzeitsysteme [MRD12] und sind vergleichbar mit dem in Abschnitt 7.2 vorgestellten Schema für Tagtypen, die für MontiEE Modelle zur Verfügung stehen und von den MontiEE Generatoren verarbeitet werden können. Neben der Möglichkeit zur manuellen Erstellung solcher Profile gibt es auch Ansätze, diese automatisiert abzuleiten. Der JUMP-Ansatz [BGWK14] leitet aus Java-Annotationen UML-Profile automatisiert ab. Von einer automatisierten Ableitung wurde in MontiEE abgesehen.

Neben den zuvor genannten Erweiterungen und Verfeinerungen für Sprachen existieren auch im Bereich der Transformationssprachen Möglichkeiten, Modelle in eine um entsprechende Informationen angereicherte Version zu transformieren. Ein Überblick dazu findet sich in [MCG05] und [GMPO09]. Auch hier existieren Ansätze zur systematischen Ableitung domänenspezifischer Transformationssprachen [RW11, Wei12, SCGL14, HRW15], die zudem die konkrete Syntax der Ausgangssprache berücksichtigen [BW07, GMP09, RW11, Wei12, HRW15]. Mit Hilfe geeigneter Tools [HJGP99, JK06, JABK08, SVL13] können dann plattformspezifische Modelle aus plattformunabhängigen Modellen transformiert werden. Dennoch unterscheiden sich die Transformationsansätze von der hier angewandten Methodik darin, dass sie stets das Modell anreichern und somit verändern. Eine Aufteilung und damit eine erhöhte Wartbarkeit sowie Wiederverwendbarkeit ist nicht gegeben.

Die Auftrennung in eigenständige Artefakte führt dazu, dass das Abstraktionsniveau der Modelle erhalten bleibt und keine technischen Informationen oder Implementierungsspezifika in dem Modell verankert werden müssen. Somit bleiben die Modelle als Abstraktion und Kommunikationsmedium erhalten. Durch die Externalisierung dieser Informationen wird zudem eine höhere Wiederverwendbarkeit einzelner Modelle erreicht, da sie in unterschiedlichen Kontexten verwendet und technische Informationen nach Belieben ausgetauscht werden können. Neben der erhöhten Wiederverwendbarkeit wird auch eine Verwendung mehrerer Generatoren unterstützt, da für jeden Generator ein eigenes Tagschema erstellt werden kann, welches das Basismodell für jeden verwendeten Generator um exakt die benötigten Informationen anreichert.

Der Methodik aus [GLRR15] folgend, wurden als Teil der MontiEE Sprachfamilie Taggingsprachen, die es ermöglichen Modelle mit Zusatzinformation anzureichern, auf Basis der UML/P CD geschaffen. In MontiEE dienen die Zusatzinformationen dazu, technische Details oder aber auch von Generatoren benötigte Informationen zu spezifizieren. Diese Sprache wurde initial in [Roi15] gemeinsam entwickelt und auf die hier präsentierte Form angepasst.

Abbildung 4.3 zeigt die Abhängigkeiten zwischen den Grammatiken TD-Grammar und TS-Grammar der Taggingsprachen für CDs, der Grammatik CD-Grammar der CD Sprache und Modellen der jeweiligen Sprachen CD, TD, und TS. Die Markierung MCG zeigt an, dass es sich in der oberen Hälfte der Abbildung um MontiCore Grammatiken, wie sie in Abschnitt 2.2 vorgestellt wurden, handelt, wohingegen es sich im unteren

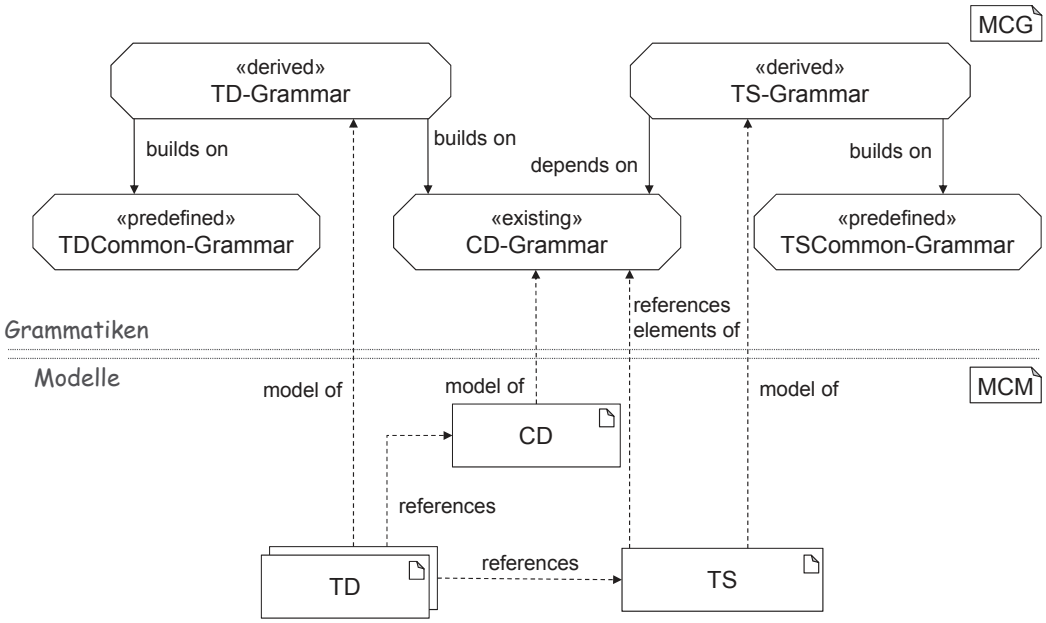


Abbildung 4.3: Abhängigkeiten der Sprachen auf Basis der Grammatik für UML/P CDs in Anlehnung an die allgemeine Form aus [GLRR15].

Teil der Abbildung um MontiCore Modelle handelt, was durch die Markierung MCM angezeigt wird.

Die in [GLRR15] entworfene Methodik erzeugt zu einer gegebenen Sprache zugehörige Taggingsprachen. In diesem Fall handelt es sich um die textuelle Sprache der Klassendiagramme, welche in Abschnitt 2.3 bereits vorgestellt wurde. Abbildung 4.3 stellt die Grammatik der Klassendiagrammsprache als CD-Grammar dar. Die beiden abgeleiteten Grammatiken TD-Grammar und TS-Grammar hängen in unterschiedlicher Art und Weise von CD-Grammar ab. Zwischen TD-Grammar und CD-Grammar ist eine `builds on` Beziehung, welche angibt, dass die Grammatik TD-Grammar eine Subgrammatik von CD-Grammar ist. Sie erbt durch den Sprachvererbungsmechanismus von der CD-Grammar und kann somit Konzepte der abstrakten Syntax der Klassendiagramme wiederverwenden. Die Notwendigkeit dafür wird in Abschnitt 4.3.1 erneut aufgegriffen. TS-Grammar hingegen besitzt eine deutlich losere Kopplung zu CD-Grammar, welche nur während der Erstellung der Grammatik existiert. Der Entwickler der Grammatik der Tagschemasprache muss genaue Kenntnis der Klassendiagrammgrammatik haben, dennoch gibt es keine direkte Abhängigkeit zwischen beiden Sprachen, die über lose Namensreferenzen hinausginge. Dies wird durch die `depends on` Beziehung ausgedrückt und in Abschnitt 4.3.2 kurz aufgegriffen und erläutert. Gleichzeitig existiert keine Beziehung zwischen den Grammatiken der Tagdefinitionssprache und der Tagschemasprache. Somit sind die Sprachen auf Grammatikebene unabhängig voneinander und lediglich über Aggregation verbunden. Diese Beziehung wird auf Modellebene etabliert.

Darüber hinaus erben beide Grammatiken, sowohl TD-Grammar als auch TS-Grammar von vordefinierten Basissprachen, die die sprachunabhängigen Konzepte definieren. Diese Basissprachen werden im folgenden Commonsprachen, ihre Grammatiken TDCommon-Grammar und TSCommon-Grammar, genannt. Beide Sprachen wurden bereits in [GLRR15] vorgestellt und werden der Vollständigkeit halber in Abschnitt 4.3.2 und Abschnitt 4.3.1 erläutert.

Auf Ebene der Modelle existieren die `model of` Beziehungen, welche angeben, dass ein Modell eine Instanz einer Sprache, also ein gültiges Wort über der Grammatik ist. Abbildung 4.3 zeigt, dass ein Klassendiagramme Instanzen von CD-Grammar, Tagdefinitionen Instanzen von TD-Grammar und Tagschemas Instanzen von TS-Grammar sind.

Darüber hinaus bezieht sich eine Tagdefinition immer auf ein konkretes Tagschema, zu dem es konform ist, und auf ein konkretes Klassendiagramm, welches die Elemente enthält, die getaggt werden sollen. Sowohl das Tagschema als auch das Klassendiagramm werden explizit in der Tagdefinition angegeben. Hier wird der Mechanismus der Sprachaggregation von MontiCore, der es ermöglicht, Elemente anderer Modelle anderer Sprachen zu referenzieren, verwendet. Dies ist durch die `references` Beziehung dargestellt.

Die `references elements of` Beziehung gibt an, dass das konkrete Tagschema sich auf Elemente der Klassendiagramme bezieht. Diese Elemente sind die Nichtterminale der Klassendiagrammgrammtik, die im Tagschema, um den Gültigkeitsbereich eines Tagtyps auf bestimmte Nichtterminale einzuschränken, verwendet werden. Hier findet eine Vermischung der Sprach- und Modellebene statt. Diese Vermischung ist an dieser Stelle explizit gewollt, da Aussagen über die abstrakte Syntax der Klassendiagrammsprache getroffen werden sollen.

Im Folgenden werden die resultierenden Taggingsprachen, die es erlauben Klassendiagramme mit technologiespezifischen Informationen anzureichern, vorgestellt.

4.3.1 Die Tagdefinitionssprache

Die Tagdefinitionssprache wird benutzt, um Elemente eines Klassendiagramms, die nicht direkt Teil des Domänenmodells sind, mit zusätzlichen Informationen anzureichern.

```
1 tags SocNetTags for SocNet {                                    TD
2   // weitere Elemente der Tagdefinition                          ...
3 }
```

Listing 4.4: Auszug der Tagdefinition `SocNetTags` für das Klassendiagramm `SocNet`. Die Elemente der Tagdefinition sind ausgelassen.

Listing 4.4 zeigt eine Tagdefinition. Zunächst startet eine Tagdefinition mit einem alle Tags umschließenden Block, beginnend mit dem Schlüsselwort `tags`, gefolgt von einem Namen. Dieser Name gibt den Namen des Modells, welcher im Kontext des Pakets eindeutig ist, an. Der vollqualifizierte Name des Modells ergibt sich aus eben diesem optional angegebenen Paket und dem Namen. Die Möglichkeit zur Paketdeklaration wird hier

nicht dargestellt. Ebenfalls in der Darstellung ausgelassen wird die Möglichkeit zur Angabe einer Konformitätsbeziehung zu einem bestimmten Tagschema, da diese Beziehung im nächsten Abschnitt aufgegriffen und kurz erläutert wird. Sie etabliert die `references` Beziehung zwischen Tagdefinition und Klassendiagramm, wie in Abbildung 4.3 dargestellt.

Nach dem Namen der Tagdefinition folgt, wie in Listing 4.4 dargestellt, eine Referenz auf das zu taggende Modell. Diese Referenz beginnt mit dem Schlüsselwort `for` gefolgt vom vollqualifizierten Namen des Klassendiagramms `SocNet`. Die Namensreferenz nach dem Schlüsselwort `for` spezifiziert das Klassendiagramm, auf welches sich die in der Tagdefinition modellierten Tags beziehen. Diese Referenz wird zur Konsistenzprüfung zwischen Tagdefinition und Klassendiagramm verwendet. Sie etabliert die ebenfalls in Abbildung 4.3 dargestellte `references` Beziehung zwischen Tagdefinition und Klassendiagramm. Innerhalb der Tagdefinition können einzelne Tags modelliert werden.

```
1  tag Profile with Entity, Inheritance = "tablePerClass";        TD
2                                                                  ...
3  tag sent with Cascading {
4    cascade="remove";
5  };
```

Listing 4.5: Darstellung der `Entity`, `Inheritance` und `Cascading` Tags für die Elemente `Profile` und `sent` des Domänenmodells. Gezeigt ist die Möglichkeit zur Modellierung verschiedener Tagtypen sowie die Möglichkeit zur Angabe mehrerer Tags.

Listing 4.5 zeigt dazu drei unterschiedliche Tags, deren Semantik in Abschnitt 7.2 erläutert und hier nicht detailliert aufgegriffen wird, da sie für das Verständnis der Sprache nicht relevant ist. Alle Tags beginnen mit dem Schlüsselwort `tag`, gefolgt von dem Namen des zu markierenden Elements. Dieses Element ist Teil des Klassendiagramms. Generell können Klassen, Enumerationen, Interfaces, Attribute, Assoziationen sowie beide Assoziationsenden getaggt werden. Auf Sprachebene wird dazu typischerweise der Name des Elements oder aber ein anderer Identifikator verwendet. Die Identifikatoren werden im nächsten Abschnitt im Rahmen der Grammatikerläuterung genauer erklärt. In Listing 4.5 werden die Elemente mit Namen `Profile` und `sent` getaggt. Diese Namensreferenz etabliert, wie bereits zuvor die Angabe des Klassendiagramms, die `references` Beziehung zwischen Tagdefinition und Klassendiagramm, wie in Abbildung 4.3 gezeigt. Nach Angabe des Elementidentifikators folgt das Schlüsselwort `with` gefolgt vom eigentlichen Tag. In Listing 4.5 wird die Klasse `Profile` mit zwei verschiedenen Tags getaggt: `Entity` und `Inheritance`. Der `Entity` Tag taggt Klassen als Entitäten, damit diese später persistent in einer Datenbank gespeichert werden können, der `Inheritance` Tag gibt die Strategie zur Abbildung von Vererbung auf ein relationales Datenbankschema an. Ihre genaue Semantik wird in Kapitel 7.2 vorgestellt. In Listing 4.5 zeigt sich, dass Tags sowohl einfache Markierungen, aber auch Werte beinhalten können. Ebenso

ist es möglich, mehrere Elemente mit mehreren Tags zu markieren. Dies erfolgt über eine kommaseparierte Liste von Elementen und Tags, wie in Listing 4.5 gezeigt. Eine weitere mögliche Form eines Tags, die nicht explizit in Listing 4.5 gezeigt ist, stellt der `Cascading` Tag dar. Dieser Tag modelliert die Kaskadierung von Datenbankoperationen auf assoziierte Objekte. Seine genaue Semantik wird in Abschnitt 7.2 detailliert vorgestellt. Er beinhaltet weitere Tags, die einem Element zugeordnet werden. Generell werden Tags über ihren Namen, der in einem Schema definiert wird, adressiert. Eine Ausnahme hiervon bilden die komplexen Tags, deren Subelemente über den verwendeten Variablennamen adressiert werden. Eine Diskussion der unterschiedlichen Tagtypen erfolgt in Abschnitt 4.3.2. Darüber hinaus sind das Aufspannen von Kontexten und das Navigieren in eine Modellelementhierarchie mit Hilfe des `within` Konstrukts möglich. Dieses wird im nächsten Abschnitt genauer erläutert.

Mit Hilfe dieser lose gekoppelten Namensreferenzen ist es möglich, Elemente des Klassendiagramms mit zusätzlichen, technologiespezifischen Informationen anzureichern. Die sprachliche Umsetzung auf Ebene der Grammatik wird im folgenden Abschnitt gezeigt.

MCG
TD-Common
...

```
1  grammar TDCommon extends mc.umlp.common.Common{
2
3    TagDefinition = "conforms" "to"
4    QualifiedName ("," QualifiedName)*";"
5    "tags" Name "for" targetModel:QualifiedName
6    "{"
7      (contexts:Context | tags:TargetElement)*
8    "}";
9
10   Context = "within" ModelElementIdentifierPath
11   "{"
12     (contexts:Context | tags:TargetElement)*
13   "}" ;
14   // weitere Produktionen
15 }
```

Listing 4.6: Aufbereiteter Auszug der Grammatik TDCommon. Dargestellt ist die Startproduktion `TagDefinition` sowie die Produktion `Context`. Weitere Produktionen der Grammatik sind ausgelassen. Die vollständige Version der Grammatik ist in Anhang C.2 dargestellt.

Die Grammatik der Tagdefinitionssprache

Um Modellelemente des Klassendiagramms taggen zu können, wurde eine sprachspezifische Grammatik, die auf einem sprachunabhängigen Teil aufbaut und von diesem erbt, entworfen. Zunächst wird die Grammatik TDCommon-Grammar der Subgrammatik der Tagdefinitionssprache, welche den sprachunabhängigen Teil beinhaltet, vorgestellt. Dar-

auf aufbauend wird die Grammatik TD-Grammar der Tagdefinitionssprache, welche die klassendiagrammspezifischen Konzepte umsetzt, erläutert.

Listing 4.6 zeigt einen Auszug aus der Grammatik TDCommon-Grammar. Diese dient dazu, die Grundstruktur und benötigte Konzepte der sprachspezifischen Subsprachen zu definieren. In MontiCore beinhaltet jede Sprache eine implizite Angabe eines `packages`, welche hier ausgelassen wurde. Diese würde zu Beginn eines Modells angegeben werden können. Ferner erbt die Sprache von der `mc.umlp.common.Common` Grammatik, welche Standardproduktionen, wie `Name`, `QualifiedName`, `String` enthält. Darüber hinaus ist erkennbar, dass jede Instanz der Sprache mit dem Schlüsselwort `conforms to`, gefolgt von einer Namensreferenz, beginnt. Dies legt das Tagschema, zu dem die Tagdefinition konform ist, fest und ermöglicht so statische Analysen und Konsistenzprüfungen. Mit der Produktion `TagDefinition` beginnt die eigentliche Definition der Tags für ein Modell. Innerhalb eines Modells können, wie in Listing 4.6 dargestellt, entweder Tags stehen oder neue Kontexte geöffnet werden. Kontexte können ineinander verschachtelt sein und jeweils wiederum Tags enthalten. Dies ist nützlich, um in zu taggende Elemente hinein zu navigieren und Subelemente dieser zu taggen. Als Beispiel wäre hier ein zu taggendes Attribut innerhalb einer Klasse zu nennen, wo zunächst der Kontext der Klasse aufgespannt und darin der Tag des Attributs definiert würde. Ein Kontext beginnt immer mit dem Schlüsselwort `within`, gefolgt von dem eindeutig identifizierenden Pfad zu einem Element. An dieser Stelle wird sich des Mechanismus zur Definition von Interfaces in MontiCore bedient. Die Grammatik TDCommon-Grammar ist vollständig sprachunabhängig und kennt ihre Subsprachen nicht.

MCG
TD-Common
...

```
1  interface ModelElementIdentifier;
2
3  DefaultIdent implements ModelElementIdentifier =
4    QualifiedName ;
5
6  ModelElementIdentifierPath =
7    parts:ModelElementIdentifier
8    ("." parts:ModelElementIdentifier)* ;
```

Listing 4.7: Darstellung des Interface-Nichtterminal `ModelElementIdentifier` und seiner Standardimplementierung. Die vollständige Version der Grammatik ist in Anhang C.2 dargestellt.

Als Erweiterungspunkte sind das Interface `ModelElementIdentifier` und die Produktion `ModelElementIdentifierPath` in der Grammatik definiert, wie Listing 4.7 zeigt. Der `ModelElementIdentifierPath` dient dabei der durch einen Punkt getrennten Konkatenation mehrerer `ModelElementIdentifer`. Dies wird als Alternative dazu verwendet, dass mehrere Kontexte verschachtelt werden. Der `ModelElement-IdentiferPath` wird von den jeweiligen Produktionen verwendet. Subsprachen, welche den sprachspezifischen Teil bereitstellen, verwenden die Erweiterungspunkte, indem sie

das Interface `ModelElementIdentifer` für alle Elemente, die getaggt werden sollen, implementieren. Auf diese Weise ist eine Vielzahl benötigter Produktionen zur Strukturierung und Navigation bereits in der Grammatik TDCommon-Grammar definiert, so dass die spezifischen Subsprachen, die manuell erzeugt werden müssen, klein und einfach zu erstellen gehalten werden können. Gleichzeitig bietet die Grammatik TDCommon-Grammar eine Standardimplementierung des `ModelElementIdentifiers` an. Diese ist durch einen vollqualifizierten Namen gegeben. Sie kann für alle Elemente verwendet werden, die eindeutig anhand ihres Namens identifiziert werden können. Dies ist meistens der Fall, aber bei manchen Elementen, wie Assoziationen oder Methoden nicht ausreichend.

MCG
TD-Common
...

```
1  interface Tag;
2
3  TargetElement = "tag" ModelElementIdentifierPath
4    ("," ModelElementIdentifierPath)* "with"
5      Tag ("," Tag)* ";" ;
6
7  SimpleTag implements Tag = Name;
8
9  ValuedTag implements Tag = Name "=" String;
10
11 ComplexTag implements Tag = Name
12   "{" (Tag ("," Tag)* ";")? "}" ;
```

Listing 4.8: `TargetElement` Produktion und die verschiedenen Tag Arten, die das Interface `Tag` implementieren. Die vollständige Version der Grammatik ist in Anhang C.2 dargestellt.

Neben der Verwendung in Kontexten, wird der `ModelElementIdentifier` auch in der Produktion `TargetElement`, dargestellt in Listing 4.8, selber verwendet. Sie startet mit dem Schlüsselwort `tag` gefolgt vom identifizierenden Pfad zu dem entsprechenden Element. Darüber hinaus besteht die Möglichkeit, innerhalb der Definition eines Tags gleich mehrere Elemente zu taggen. Dies geschieht, indem mehrere `ModelElement-IdentifierPath` Angaben kommasepariert verkettet werden. Daran anschließend folgt das Schlüsselwort `with` mit einem oder mehreren verschiedenen Tags. Es werden drei verschiedene Tags, von denen jeder das Interface `Tag` implementiert, und die im Wesentlichen Schlüssel-Wert Paare darstellen, unterschieden.

Der `SimpleTag`, wie in Listing 4.8 dargestellt, besteht dabei lediglich aus einem Namen, um das zu taggende Element mit diesem Namen zu markieren. Der `ValuedTag` beinhaltet zusätzlich einen primitiven Wert eines bestimmten Datentyps, der in der Tagdefinition immer als String repräsentiert ist. In einem Tagschema kann auch ein anderer Datentyp, wie beispielsweise ein Element einer Aufzählung oder ein numerischer Typ angegeben werden. Dies wird in Abschnitt 4.3.2 detaillierter erklärt. Der `ComplexTag` besitzt einen komplexeren Wert, welcher eine durch Komma getrennte Menge geschachtelter Subtags beinhalten kann. Diese Subtags haben einen Variablennamen und einen

Tagtyp. Aus einer übergeordneten Tagdefinition werden sie über ihren Variablennamen adressiert, wohingegen die anderen Tagtypen, sofern kein Name angegeben ist, über den Typ adressiert werden. Auf der Grammatik TDCommon-Grammar aufbauend, wurde die Grammatik der Tagdefinitionssprache als Subgrammatik umgesetzt. Diese Grammatik beinhaltet lediglich die sprachspezifischen Produktionen, die benötigt werden, um die Taggingsprachen sprachspezifisch zu entwerfen.

MCG
TD

```
1 grammar TD extends TDCommon, mc.umlp.cd.CD {
2
3   AssociationIdentifier implements ModelElementIdentifier =
        CDAssociation;
4
5   AssociationElementIdentifier implements ModelElementIdentifier
6     = (Name | CDAssociation) (["!lefthand"] | ["!righthand"]);
7
8 }
```

Listing 4.9: Darstellung der benötigten Produktionen zur Erstellung der klassendiagrammspezifischen Subgrammatik TD. Gezeigt sind die Produktionen, die das Interface-Nichtterminal `ModelElementIdentifier` implementieren und zur Identifikation von Assoziationen und Assoziationsseiten dienen. Die vollständige Version der Grammatik ist in Anhang C.3 dargestellt.

Listing 4.9 zeigt eine sprachspezifische Anpassung für Klassendiagramme. Die Aufgabe dieser Subsprache liegt darin, das Interface-Nichtterminal `ModelElementIdentifier` für alle Elemente zu implementieren, die in einem konkreten Modell getaggt werden sollen. Hier wird sich zu Nutze gemacht, dass die Grammatik TDCommon-Grammar, von der die sprachspezifische Anpassung, wie in Listing 4.9 dargestellt, erbt, bereits eine Standardimplementierung für Elemente, die einen eindeutigen Namen besitzen, vorgibt. Für die Klassendiagrammsprache sind dies Klassen, Enumerationen, Interfaces und Attribute innerhalb einer Klasse. Auch für benannte Assoziationen ist eine Identifizierung durch den Namen möglich. In Listing 4.5 wurden jeweils die Namen zur Identifikation verwendet. Da aber auch Assoziationen ohne Namen sowie die Assoziationsenden getaggt werden sollen, muss dies in der Grammatik TD-Grammar umgesetzt werden. Die Produktion `AssociationIdentifier` definiert dabei, dass eine Assoziation auch mit Hilfe ihrer vollständigen konkreten und abstrakten Syntax identifiziert werden kann. Dazu wird sich der Mechanismus der Mehrfachvererbung bei Sprachen, der in Abschnitt 2.2 präsentiert wurde, zu Nutze gemacht, durch den die Grammatik der Tagdefinitionssprache von der Grammatik der Klassendiagrammsprache, die in Abschnitt 2.3 vorgestellt wurde, erbt und somit die Produktion `CDAssociation` verwendet werden kann. Durch die Verwendung der Produktion ist es im Modell möglich, die vollständige Syntax einer Assoziation zu verwenden, so dass diese gegen eine Assoziation im Klassendiagramm gematcht werden kann. Dies bringt sicherlich den Nachteil mit sich, dass die Tagdefinition

von der konkreten Syntax des Zielmodells abhängt.

Assoziationsenden können auf ähnliche Art und Weise identifiziert werden. Bei benannten Assoziationen können die Enden mit Hilfe des Namens und den Suffixen `!left-hand` oder `!righthand` identifiziert werden. Auch hier kann wieder die vollständige konkrete Syntax der Assoziation verwendet werden, da auch hier die zur Verfügung gestellte Produktion `CDAssociation` verwendet wird. Mit Hilfe der geschaffenen Sprache können also Elemente des Klassendiagramms mit spezifischen Tags getaggt werden. Die vollständigen Versionen der Grammatiken sind im Anhang in Listing C.2 und C.3 gegeben. Im nachfolgenden Abschnitt wird die zugehörige Tagschemasprache vorgestellt, die zur Definition einzelner Tagtypen verwendet wird. Eine umfassende Tagdefinition, welche zur Generierung eines Systems verwendet wird, ist in Abschnitt 7.2 gegeben.

4.3.2 Die Tagschemasprache

Zur Definition der in einer Tagdefinition verwendeten Tags wurde die Tagschemasprache geschaffen, welche die Definition bestimmter Tagtypen ermöglicht.

```
1 tagschema MontiEE  {                                                    TS
2   // weitere Elemente der Tagschemas                                     ...
3 }
```

Listing 4.10: Auszug des Tagschemas `MontiEE`. Die Elemente des Tagschemas sind ausgelassen.

Listing 4.10 zeigt ein Modell der Tagschemasprache. Ein Tagschema wird immer mit dem Schlüsselwort `tagschema`, gefolgt von einem Namen, begonnen. Innerhalb dieses Blocks können dann unterschiedliche Tagtypen definiert werden. Ausgelassen wurde die Möglichkeit zur Angabe des Paketnamens.

```
1                                                                         TS
2   tagtype Entity for CDClass;                                           ...
```

Listing 4.11: Definition des `Entity` Tagtyps. Er darf für Elemente des Typs `CDClass` verwendet werden.

Listing 4.11 zeigt die Definition des `Entity` Tagtyps, der in Listing 4.5 verwendet wurde. Die Definition eines Tagtyps beginnt dabei mit dem Schlüsselwort `tagtype`, gefolgt von der inhaltlichen Definition des Tagtyps. Danach folgt das Schlüsselwort `for` sowie eine Namensreferenz auf ein Element der Zielgrammatik, die im Rahmen dieser Arbeit die Grammatik der Klassendiagrammsprache ist. Durch diese Referenz wird die `references elements of` Beziehung, wie in Abbildung 4.3 dargestellt, etabliert. An dieser Stelle werden die Modellierungsebenen vermischt, da der Modellierer des Tagschemas in einem konkreten Modell Elemente des Metamodells der Sprache referenziert. In

Listing 4.11 ist dieses Element die `CDClass` Produktion der Klassendiagrammsprache. Im Tagschema wird diese Referenz als Name dargestellt. Semantisch drückt Listing 4.11 aus, dass der `Entity` Tagtyp nur auf Elemente des Typs `CDClass` angewendet werden darf. Dies wird durch Kontextbedingungen sichergestellt. Darüber hinaus kann auch $\star$ als Platzhalter für beliebige Elemente verwendet werden. Dies ermöglicht es, eine Menge von Tagtypen zu definieren, mit deren Hilfe Modelle erweitert werden können, die die Generatoren verarbeiten können. Ohne ein solches Tagschema und eine solche Definition von Tagtypen wären die technologiespezifischen Informationen, die von Generatoren zwingend benötigt werden, im Wesentlichen strukturlose Freitextinformationen.

```
1  tagtype Inheritance:["singleTable"|"joined"                    TS
2    |"tablePerClass"] for CDClass;                                ...
```

Listing 4.12: Definition des `Inheritance` Tagtyps. Er darf für Elemente des Typs `CDClass` verwendet werden. Seine möglichen Werte sind `singleTable`, `joined` oder `tablePerClass`

Listing 4.12 zeigt eine Definition einer weiteren Art eines Tagtyps. Während der `Entity` Tagtyp eine einfache Markierung eines Elements darstellt, zeigt der `Inheritance` Tagtyp die Möglichkeit zur Angabe möglicher Werte eines Tagtyps. Hier dargestellt ist die Angabe einer Enumeration. Eine Angabe eines Wertetypen, wie String oder Integer, ist aber auch möglich und wird im nächsten Abschnitt gezeigt. Eine Enumeration wird dabei an die von MontiCore bekannte Syntax, die in Abschnitt 2.2 präsentiert wurde, angelehnt. Sie wird mit dem Namen des Tagtyps, einem Doppelpunkt und den möglichen, von eckigen Klammern umschlossenen Optionen definiert. Die einzelnen Optionen werden mit Hilfe des `|` Operators getrennt.

```
1  tagtype Cascading for CDAssociation,                           TS
2    CDAssociation!lefthand, CDAssociation!righthand {            ...
3      cascade:Cascade+;
4  }
5
6  inner tagtype
7    Cascade:["all"|"none"|"merge"|"persist"|"refresh"
8             |"remove"];
```

Listing 4.13: Definition des `Cascading` Tagtyps. Er darf für Elemente des Typs `CDAssociation` sowie linke und rechte Seiten der Assoziation verwendet werden. Er kann aus mehreren inneren `Cascade` Tagtypen bestehen, deren mögliche Werte `all`, `none`, `merge`, `persist`, `refresh` oder `remove` sind.

Listing 4.13 zeigt die Möglichkeit zur Verschachtelung von Tagtypen und zur Modellierung innerer Tagtypen. Dazu zeigt Listing 4.13 den `Cascading` Tagtyp, der für Assoziationen sowie Assoziationsenden verwendet werden kann. Innerhalb des `Cascading` Tagtyps wird der innere `Cascade` Tagtyp verwendet. Dieser Tagtyp kann nur innerhalb verschachtelter Tagtypen verwendet werden. Zudem ist dargestellt, dass die verwendeten verschachtelten Tagtypen mit Kardinalitäten versehen werden können. Diese Kardinalitäten sind analog zu den aus Klassendiagrammen bekannten Kardinalitäten.

Die genaue Semantik des `Cascading` Tagtyps wird in Abschnitt 7.2 detailliert erläutert und hier nicht weiter aufgegriffen, da sie für das Verständnis der Sprache nicht relevant ist. Nachdem die Tagschemasprache mit ihrer konkreten Syntax und ihrer Semantik vorgestellt wurde, wird im Folgenden die Umsetzung auf Basis der Grammatik vorgestellt. Auch hier wird wieder eine Trennung in einen sprachspezifischen und einen sprachunabhängigen Teil verwendet.

Die Grammatik der Tagschemasprache

Analog zur Grammatik der Tagdefinitionssprache und der Grammatik der allgemeinen Supersprache wurde auch die Tagschemasprache in einen allgemeinen, sprachunabhängigen Teil, welcher gemeinsame Produktionen für alle Tagschemas zur Verfügung stellt und einen für CDs spezifischen Teil aufgeteilt. Der sprachunabhängige Teil, die Grammatik TSCommon-Grammar, wird zunächst vorgestellt und daran anschließend der sprachspezifische Teil der Grammatik der Tagschemasprache erläutert.

```
1 grammar TSCommon extends mc.umlp.common.Common {
2
3   TagSchema = "tagschema" Name "{" TagType* "}";
4   // weitere Produktionen
5 }
```

MCG
TS-Common
...

Listing 4.14: Aufbereiteter Auszug der Grammatik der Tagschemasprache. Dargestellt ist die Startproduktion `TagSchema`. Weitere Produktionen der Grammatik sind ausgelassen. Die vollständige Version der Grammatik ist in Anhang C.4 dargestellt.

Listing 4.14 zeigt einen Auszug der Grammatik der Tagschemasprache. Auch hier ist wie schon zuvor bei der Grammatik der Tagdefinitionssprache darauf verzichtet worden, die Möglichkeit zur Paketangabe aufzuführen. Sie ist aber implizit gegeben. Jedes Tagschema beginnt mit dem Schlüsselwort `tagschema`, gefolgt von einem Namen, welcher das Tagschema innerhalb der Paketstruktur eindeutig benennt. Innerhalb des Tagschemas können verschiedene mögliche Tagtypen modelliert werden. Alle Produktionen, die einen Tagtyp definieren, müssen das Interface `TagType`, wie in Listing 4.15, implementieren.

Es werden dabei, wie Listing 4.15 dargestellt, die vier verschiedene Arten von Tagtypen, `SimpleTagType`, `ValuedTagType`, `EnumeratedTagType` und `ComplexTag-`

```
1   interface TagType;                                                    MCG
2                                                                         TS-
3   SimpleTagType implements TagType =                                    Common
4     ["inner"]? "tagtype" Name Scope? ";";                               ...
5
6   EnumeratedTagType implements TagType =
7     ["inner"]? "tagtype" Name ":" "[" String ("|" String)* "]"
8       Scope? ";";
9
10  ValuedTagType implements TagType =
11    ["inner"]? "tagtype" Name ":" ("int"|"String"|"Boolean")
12      Scope? ";";
13
14  ComplexTagType implements TagType =
15    ["inner"]? "tagtype" Scope?
16      "{" Attribute ("," Attribute)* ";" "}" ;
17
18  Attribute = Name ":" ("int"|"String"|"Boolean"|Name) Cardinality?;
19
20  Cardinality = "?" | "+" | "*";
```

Listing 4.15: Darstellung des Interface-Nichtterminals `TagType` und seiner Implementierungen. Die vollständige Version der Grammatik ist in Anhang C.4 dargestellt.

`Type` unterschieden. Diese sollten nicht mit den vorangegangenen Tags der Tagdefinitionssprache verwechselt werden, da die Produktionen, die `TagType` implementieren, die konkrete und abstrakte Syntax zur Definition unterschiedlicher Tagtypen in einem Tagschema beschreiben. Die Produktionen der Tagdefinitionssprache, die `Tag` implementieren, beschreiben hingegen die konkrete und abstrakte Syntax zur Markierung eines Elements des Klassendiagramms mit einem Tag eines entsprechenden Tagtyps in der Tagdefinition. Alle Tagtypen haben dabei gemein, dass sie mit dem Schlüsselwort `tagtype` beginnen, dann einen Namen haben und die Produktion `Scope` verwenden. Diese Produktion wird in Listing 4.16 genauer vorgestellt. Dieser Scope dient dazu, die Elementtypen, die mit diesem Tagtyp getaggt werden dürfen, einzuschränken. Im Folgenden werden die unterschiedlichen Arten von Tagtypen und deren Beziehung zu den Tags detaillierter dargestellt:

- `SimpleTagType`: Tagtypen, die nur einen einfachen Schlüssel zur Markierung eines Elements besitzen, sind von diesem Typ. In der Tagdefinition werden solche Tagtypen als `SimpleTag` ausgedrückt.

- `ValuedTagType`: Tagtypen, die neben dem Schlüssel auch unterschiedliche Werte erlauben, besitzen diesen Typ. Die Werte können, wie in Listing 4.15 gezeigt, vom Typ `int`, `String` oder `Boolean` sein. In der Tagdefinition werden solche

Tagtypen als `ValuedTag`, der jedoch nur einen String als Wert verarbeitet, ausgedrückt. Die Typumwandlung wird intern gehandhabt.

- `EnumeratedTagType:` Tagtypen, die einen Wert aus dem eingeschränkten Wertebereich einer Aufzählung haben können, besitzen diesen Typ. In der Tagdefinition werden diese Tagtypen ebenfalls als `ValuedTag` ausgedrückt. Die Überprüfung, ob der angegebene Wert ein gültiges Element der Aufzählung, ist intern gehandhabt wird.
- `ComplexTagType:` Tagtypen, die komplexe, andere Tagtypen enthaltende Werte haben, besitzen diesen Tagtyp. Sie können beliebige Subtagtypen beinhalten, die wiederum entweder einen primitiven Typ oder ein Attribut als Referenz auf einen modellierten Tagtyp sein können. Innerhalb eines komplexen Tagtyps können die Subtagtypen ebenfalls benannt und eine Kardinalität angegeben werden. Wenn keine Kardinalität angegeben wird, muss der Subtagtyp in dem komplexen Tagtyp angegeben werden. Wird eine Kardinalität angegeben, kann der Subtagtyp optional ("?") sein, mindestens einmal benötigt ("+") werden oder beliebig oft angegeben werden ("∗").

In einem Tagschema können Tagtypen mit dem Schlüsselwort `inner` als innere, d.h. nur in komplexen Tagtypen nutzbar, markiert werden. Dadurch werden nicht alle Tagtypen nach außen sichtbar und können direkt verwendet werden.

MCG
TS-Common
...

```
1  interface ScopeIdentifier;
2
3  Scope = "for" (ScopeIdentifier ("," ScopeIdentifier)*
4                 | "*");
```

Listing 4.16: Darstellung des Interface-Nichtterminals `ScopeIdentifier` und der `Scope` Produktion. Die vollständige Version der Grammatik ist in Anhang C.4 dargestellt.

Die `Scope` Produktion, wie in Listing 4.16 gezeigt, beginnt mit dem Schlüsselwort `for`, gefolgt von einer durch Komma getrennten Aufzählung verschiedener `ScopeIdentifier`, die Elemente der Klassendiagrammsprache referenzieren. Die Grammatik TSCommon-Grammar definiert das Interface `ScopeIdentifier`. Es muss in den Subsprachen, damit ein Tagschema in der Lage ist Elemente der Grammatik der Klassendiagrammsprache zu referenzieren, implementiert werden. Durch die Implementierung dieses Interfaces werden eindeutige Identifikatoren, die die Elemente der Grammatik der Klassendiagrammsprache identifizieren können, bestimmt. Dies sind die Nichtterminale der Grammatik, welche per Konstruktion immer einen eindeutigen Namen haben. Dieser Name, beispielsweise "CDClass" wird in den Subsprachen als Identifikator verwendet. Nichtsdestotrotz sind diese Identifikatoren sprachspezifisch und müssen in den Subsprachen implementiert werden, während der Grammatik der Tagschemasprache nur

das Interface bekannt ist. Die durch den Scope referenzierten Elemente bedingen die `references elements of` Kante zwischen Tagschema und Grammatik der Klassendiagrammsprache, wie in Abbildung 4.3 dargestellt. Auf Ebene der Grammatik der Tagschemasprache und der Klassendiagrammsprache wird dies durch die `depends on` Beziehung, da keine Sprachvererbung verwendet wird, sondern durch die Namensreferenz eine lose Kopplung verwendet wird, dargestellt.

MCG
TS

```
1  grammar TS extends TSCommon {
2
3    CDClassScope implements ScopeIdentifier
4      = "CDClass";
5
6    CDEnumScope implements ScopeIdentifier
7      = "CDEnum";
8
9    CDInterfaceScope implements ScopeIdentifier
10     = "CDInterface";
11
12   CDAttributeScope implements ScopeIdentifier
13     = "CDAttribute";
14
15   CDAssociationScope implements ScopeIdentifier
16     = "CDAssociation";
17
18   CDAssociationLefthandScope implements ScopeIdentifier
19     = "CDAssociation" "!" "lefthand";
20
21   CDAssociationRighthandScope implements ScopeIdentifier
22   = "CDAssociation" "!" "righthand";
23
24 }
```

Listing 4.17: Grammatik der klassendiagrammspezifischen Subsprache `TS`. Die vollständige Version der Grammatik ist in Anhang C.5 dargestellt.

Auf der Grammatik TSCommon-Grammar aufbauend, wurde die Grammatik der Tagschemasprache als Subgrammatik, die in Listing 4.17 gezeigt wird, umgesetzt. Es zeigt sich, dass nur die Produktionen benötigt werden, die das Interface `ScopeIdentifier` implementieren. Somit ist der Aufwand diese Sprache zu erstellen, sehr gering. Ferner zeigt sich, dass nicht alle Konzepte der Klassendiagramme tagbar sind. Es wurde sich hier auf Klassen, Enumerationen, Interfaces, Attribute, Assoziationen sowie deren linke und rechte Seite beschränkt. Eine Erweiterung der Sprache um weitere Konzepte ist möglich. Gleichzeitig ist erkennbar, dass die Identifikatoren den Namen der Nichtterminale der Klassendiagrammgrammatik entsprechen. Lediglich bei der Unterscheidung zwischen linker und rechter Seite einer Assoziation wurde ein Name vergeben, der nicht direkt aus

der Grammatik der Klassendiagramme ableitbar ist. Es sei angemerkt, dass sich die Suffixe `!lefthand` und `!righthand` auch auf Elemente der Grammatik beziehen. Sie bezeichnen also allgemein linke oder rechte Seiten von Assoziationen, wohingegen die in Abschnitt 4.3.1 vorgestellten Suffixe sich auf eine konkrete linke oder rechte Seite einer Assoziation innerhalb eines Klassendiagramms beziehen.

Die sprachspezifischen Erweiterungen, wie die Tagdefinitionssprache und die Tagschemasprache lassen sich nahezu schematisch umsetzen und eignen sich daher dazu, generiert zu werden. In [HHK+13, HHK+15] wurde eine Methodik zur Generierung von Deltasprachen vorgestellt. Diese Methodik und die Konzepte lassen sich auch hier anwenden und können dazu beitragen, die Erstellung sprachspezifischer Taggingsprachen weiter zu vereinfachen. Die vollständigen Versionen der Grammatiken sind im Anhang in Listing C.4 und C.5 gegeben.

Auch im Rahmen dieser Arbeit wird eine Deltasprache, die auf der zuvor genannten Methodik basiert, verwendet. Diese wird in Abschnitt 6.2 vorgestellt. Die hier vorgestellten Taggingsprachen werden im Rahmen dieser Arbeit dazu verwendet, ein spezifisches Tagschema für MontiEE Generatoren zu erschaffen, welches in Abschnitt 7.2 vollständig vorgestellt wird. Dort wird auch die Semantik der einzelnen Tagtypen, wie `Entity`, `Inheritance` und `Cascading`, vorgestellt. Darüber hinaus wird in Abschnitt 7.2 eine exemplarische Tagdefinition zur Modellierung des Szenarios aus Abschnitt 3.5 und zur Anreicherung des Klassendiagramms aus Abbildung 4.2 mit technologiespezifischen Informationen gezeigt werden.

4.3.3 Kontextbedingungen

Zur Konsistenzsicherung der Beziehungen zwischen den einzelnen Modellen wurden im Rahmen dieser Arbeit und gemeinsam in [Roi15] Kontextbedingungen identifiziert, die auch in [GLRR15] als Verallgemeinerung beschrieben werden. Sie dienen im Wesentlichen dazu, die einzelnen Beziehungen zwischen Modellen und Sprachen sicherzustellen. Insbesondere beziehen sie sich auf die Existenz der einzelnen Elemente sowie auf die Konformität der Tagdefinition zum Tagschema. Die Kontextbedingungen der Tagdefinitionssprache betreffen den `ModelElementIdentifierPath`, der die `references` Beziehung zwischen Tagdefinition und Klassendiagramm, wie in Abbildung 4.3 dargestellt, etabliert:

TD-1

Bedingung: Ein in einem `ModelElementIdentifierPath` für ein `TargetElement` referenziertes Element muss innerhalb eines Klassendiagramms existieren. Dies gilt auch für alle Elemente innerhalb einer kommaseparierten Liste.

Schweregrad: Fehler

TD-2

Bedingung: Ein in einem `ModelElementIdentifierPath` für einen `Context` referenziertes Element muss innerhalb eines Klassendiagramms existieren.

Schweregrad: Fehler

Darüber hinaus betreffen die Kontextbedingungen der Tagdefinitionssprache die `references` Beziehung zwischen Tagdefinition und Tagschema zur Sicherung der Konformität:

TD-3
Bedingung: Ein in einer Tagdefinition referenziertes Tagschema muss existieren.
Schweregrad: Fehler

TD-4
Bedingung: Ein in der Tagdefinition verwendeter Tag muss im Tagschema existieren. Dieser kann über seinen eindeutigen Namen identifiziert werden.
Schweregrad: Fehler

TD-5
Bedingung: Ein in der Tagdefinition verwendeter Tag darf im Tagschema nicht als `inner` markiert sein.
Schweregrad: Fehler

TD-6
Bedingung: Der Typ der getaggten Elemente muss zu dem erlaubten `Scope` des Tagtyps kompatibel sein.
Schweregrad: Fehler

TD-7
Bedingung: Der Wert eines `ValuedTagType` muss angegeben sein und dem geforderten Typ entsprechen.
Schweregrad: Fehler

TD-8
Bedingung: Der Wert eines `EnumeratedTagType` muss angegeben und Teil der Enumeration sein.
Schweregrad: Fehler

TD-9
Bedingung: Werte und verschachtelte Tagtypen eines `ComplexTagType` müssen den Kardinalitäten entsprechend angegeben und jeweils einzeln typkorrekt sein.
Schweregrad: Fehler

Für das Tagschema selbst existieren Kontextbedingungen, die die Konsistenz innerhalb des Tagschemas sichern. Die `references elements of` Beziehung zwischen

dem Tagschema und der Grammatik der Klassendiagrammsprache wird an dieser Stelle nicht durch Kontextbedingungen, sondern erst zur Laufzeit der generierten Infrastruktur, gesichert. Die Kontextbedingungen für das Tagschema sind:

TS-1

Bedingung: Jeder Tagtyp innerhalb eines Schemas muss einen eindeutigen Namen besitzen.

Schweregrad: Fehler

TS-2

Bedingung: Jeder von einem komplexen Tagtyp als Attribut referenzierte Tagtyp muss existieren.

Schweregrad: Fehler

Darüber hinaus kann forciert werden, ob Elemente mehrfach getaggt werden können. Dies betrifft sowohl das Taggen mit mehrfach dem gleichen Tag, da dies zu inkonsistenten Zuständen führen kann, falls unterschiedliche Werte bei gleichem Tag verwendet wurden, aber auch das mehrfache Taggen des gleichen Elements mit unterschiedlichen Tags. Dies ist konzeptionell kein Problem, kann aber dazu führen, dass die Tagdefinition, da verschiedene Informationen zu einem Modell an unterschiedlichen Stellen stehen, unübersichtlich und unstrukturiert wird. Darüber hinaus kann es vorkommen, dass auf unterschiedliche Arten in den gleichen Kontext navigiert wird. Auch dies führt zu Unübersichtlichkeit. Generell werden diese Eigenschaften zwar nicht vorgegeben oder abgesichert, sollten aber dennoch bei der Modellierung berücksichtigt werden.

Abschließend lässt sich festhalten, dass die hier vorgestellten Taggingsprachen dazu verwendet werden können, Klassendiagramme mit zusätzlichen Informationen auszuzeichnen. Die Sicherung der Konsistenz der unterschiedlichen Modell- und Sprachbeziehungen erfolgt mit Hilfe der vorgestellten Kontextbedingungen. Die Methodik zur Ableitung der vorgestellten Sprachen wurde in [GLRR15] gezeigt und in [Roi15] sowie im Rahmen dieser Arbeit auf die Sprache der Klassendiagramme angewendet. Ein auf Basis dieser Sprachen erstelltes Tagschema für die MontiEE Generatoren sowie eine detaillierte Erläuterung der definierten Tagtypen werden in Abschnitt 7.2 gezeigt. Zudem wird in Abschnitt 10.3 eine Tagdefinition, welche konform zu dem MontiEE Tagschema ist, vorgestellt. Dieses wird zur Erläuterung der Generatoren und zur Modellierung des in Abschnitt 3.5 vorgestellten Szenarios verwendet.

Nachdem in diesem Abschnitt die Taggingsprachen und ein spezifisches Tagschema zur Modellierung von Enterprise Applikationen dargestellt wurden, wird im nächsten Abschnitt die Sichtensprache für Klassendiagramme vorgestellt, die es erlaubt, Teile eines Basisklassendiagramms als sogenannte Sichten zu modellieren.

4.4 Zusammenfassung

In diesem Kapitel wurde ein Teil der Sprachfamilie MontiEE vorgestellt. Die vorgestellten Sprachen ermöglichen dem Produktentwickler die Modellierung der Persistenz von Enterprise Applikationen. Dazu wurde zunächst das Szenarios aus Abschnitt 3.5 verwendet, um das Domänenmodell der Enterprise Applikationen auf Basis eines Klassendiagramms zu erstellen. Die Modellierung des Domänenmodells alleine reicht aber zur Modellierung einer Enterprise Applikationen nicht aus. Weitere technologiespezifische Informationen werden benötigt.

Dazu wurde zunächst eine verallgemeinerte Methodik in Abschnitt 4.3, die zur Erstellung von Taggingsprachen für beliebige MontiCore Grammatiken geschaffen wurde, vorgestellt. Diese Taggingsprachen bestehen aus einer Tagschema- und einer Tagdefinitionssprache, die konkrete Modellelemente auszeichnet. Die Tagdefinitionssprache ermöglicht es dem Produktentwickler, einem Modell weitere Informationen, die nicht direkt der fachlichen Modellierungsdomäne zugehörig sind, sondern weiterführende Informationen, wie beispielsweise Technologiespezifika, darstellen, hinzuzufügen. Die Tagschemasprache ermöglicht es Werkzeugentwickler die Definition möglicher Tags und der Elemente, an welche die Tags geschrieben werden können. Der Methodik folgend, wurden in Abschnitt 4.3 eine klassendiagrammspezifische Tagdefinitionssprache sowie eine klassendiagrammspezifische Tagschemasprache und ihre Kontextbedingungen vorgestellt. Mit Hilfe dieser Sprachen lässt sich das Domänenmodell mit weiteren technologiespezifischen Informationen erweitern.

Kapitel 5

Modellierung der Kommunikation

Nachdem im vorangegangenen Kapitel 4 die Sprachen zur Modellierung der Persistenz von Enterprise Applikationen vorgestellt wurden, werden in diesem Kapitel die Sprachen zur Modellierung der Kommunikation von Enterprise Applikationen vorgestellt. Diese dienen dem Produktentwickler dazu, sowohl die kommunizierten Daten als auch die Kommunikationstechnologie festzulegen. Bei der Wahl werden die verschiedenen Anforderungen und Möglichkeiten heterogener Clients berücksichtigt. Die Methodik zur Verwendung von MontiEE, die in Kapitel 10 detailliert vorgestellt wird, sieht diesen Schritt parallel zur Modellierung des Domänenmodells und parallel zur Modellierung der Persistenzinformationen vor. Dies bedeutet, dass der Produktentwickler das serverseitige Domänenmodell, wie in Abschnitt 4.2 gezeigt, definieren kann und gleichzeitig Entscheidungen bezüglich der Persistenz getroffen hat. Daneben kann er Sichten, Rechte-, Rollen- und Mappingdiagramme, die in diesem Kapitel vorgestellt werden, modellieren, die das Modell der Enterprise Applikation erweitern. Entgegen der zuvor in Abschnitt 4.3 vorgestellten Taggingsprachen werden in diesem Kapitel eigenständige Sprachen vorgestellt, da die hier modellierten Elemente keine Zusatzinformationen, sondern eigenständige Elemente darstellen.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Vorstellung der Sichtensprache, die die Modellierung des Domänenmodells der Clients ermöglicht.
- Vorstellung einer Transformation von Sichten in Klassendiagramme.
- Kontextbedingungen zur Konsistenzsicherung zwischen einer Sicht und einem Klassendiagramm.
- Vorstellung der Rechte-, Rollen- und Mappingsprachen.
- Vorstellung einer automatisierten Ableitung von Rechten.
- Kontextbedingungen zur Konsistenzsicherung zwischen Rechte-, Rollen- und Mappingsprachen sowie einem Klassendiagramm.

In Kapitel 6 steht die Deltasprache zur Modellierung der Evolution von Enterprise Applikationen im Fokus. Zunächst wird mit einem Überblick des Kapitels begonnen.

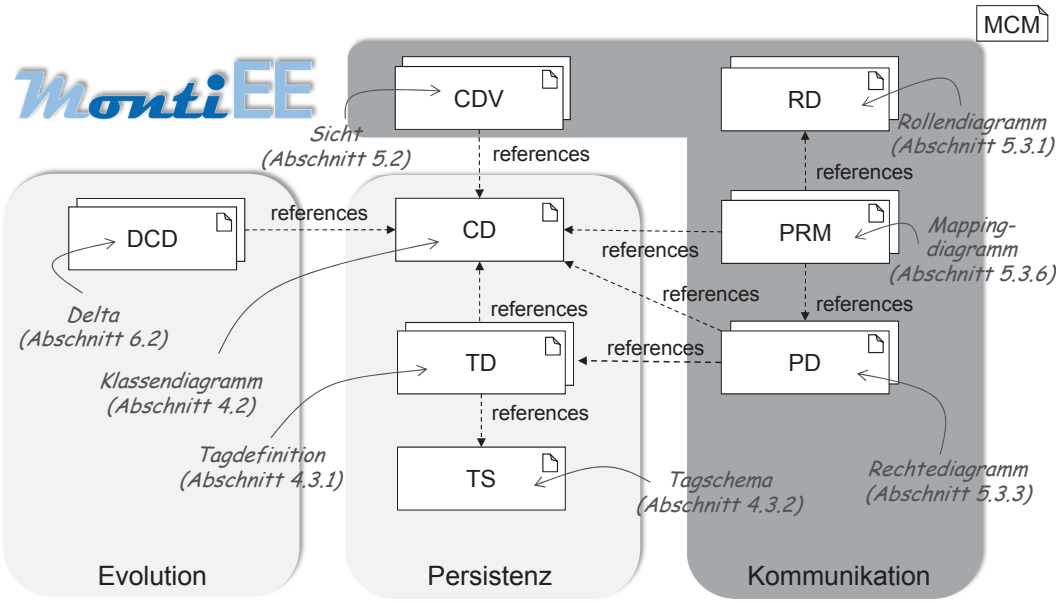


Abbildung 5.1: Überblick über die Modelle der Sprachfamilie MontiEE und deren Zusammenhänge. Der Fokus liegt dabei auf den Modellen zur Modellierung der Kommunikation von Enterprise Applikationen.

5.1 Überblick

Abbildung 5.1 greift zur besseren Darstellung Abbildung 4.1 erneut auf und zeigt den Fokus dieses Kapitels. Die Modelle zur Modellierung der Kommunikation sind hervorgehoben. Diese Modelle sind: Sichten (CDVs), Rollendiagramme (RD), Rechtediagramme (PD) und Mappingdiagramme (PRM). Sie stellen dabei jeweils Modelle der Sichten-, Rollen-, Rechte- oder Mappingdiagrammsprache dar. Diese Sprachen und deren Semantik werden im weiteren Verlauf detailliert vorgestellt.

Sichten, die in Abschnitt 5.2 erläutert werden, werden dazu verwendet, die einzelnen Domänenmodelle der heterogenen Clients zu modellieren. Dies wird benötigt, da die verschiedenen Clients unterschiedliche Anforderungen und Möglichkeiten besitzen. Mit Hilfe der Sicht kann ein reduziertes Domänenmodell, welches nur die für den jeweiligen Client relevanten Datentypen enthält, definiert werden. Modelle referenzieren Klassendiagrammelemente als Teil der Sicht. Dabei werden die Elemente des Klassendiagramms, die Teil der Sicht sind und dem Client zur Verfügung stehen sollen, referenziert. Eine Transformation von Sichten in Klassendiagramme wird in Abschnitt 5.2.3 erläutert.

Rollendiagramme definieren die im System vorhandenen Rollen. Dies wird benötigt, da unterschiedliche Nutzer verschiedene Operationen auf dem Server ausführen können müssen. Dabei existieren typischerweise Nutzer, die die Enterprise Applikation verwenden und eingeschränkten Zugriff auf eine Teilmenge der Daten haben, und solche Nutzer, Administratoren, die administrative Aufgaben übernehmen und auf einen größeren

Teil der Daten zugreifen können. Die Rollendefinitionen werden dazu verwendet, Nutzern Rollen zuzuordnen. Rechtediagramme enthalten unterschiedliche Rechte, die von den unterschiedlichen Rollen auf den zuvor vorgestellten Entitäten ausgeführt werden können. Da sich die Rechte auf Operationen, die auf den Entitäten ausgeführt werden, beziehen, referenzieren die Rechtediagramme das Klassendiagramm. Da sie weitere spezifische Informationen benötigen, werden auch die Tagdefinitionen referenziert. Die Zugriffskontrolle wird mit Hilfe der vom Server angebotenen Funktionalität ausgeführt. Dabei wird im Wesentlichen geprüft, ob ein Nutzer eine Rolle, die das Recht besitzt die gewünschte Operation auszuführen, hat. Für die Zuordnung von Rechten zu Rollen werden Mappingdiagramme verwendet. Mappingdiagramme referenzieren sowohl Rechte- als auch Rollendiagramme. Die Definition und die Semantik der Sprachen wird in Abschnitt 5.3 erläutert. Dort wird ebenfalls eine automatisierte Ableitung von Rechtediagrammen beschrieben. Die Verwendung zur Codegenerierung wird in Kapitel 8 beschrieben.

Im nachfolgenden Kapitel wird der Teil der Sprachfamilie MontiEE vorgestellt, der die Sprachen zur Modellierung der Kommunikation von Enterprise Applikationen beinhaltet.

5.2 Sichten auf Klassendiagramme

Nachdem zuvor eine Möglichkeit zur Anreicherung von Klassendiagrammen mit zusätzlichen Informationen mit Hilfe von Taggingsprachen vorgestellt wurden, wird eine Sprache zur Modellierung von Sichten vorgestellt, die auf der betreuten Vorarbeit [Sch13] beruht. Eine Enterprise Applikation wird meistens von vielen unterschiedlichen Clients verwendet, welche unterschiedliche Voraussetzung bezüglich Performanz oder zur Verfügung stehender Bandbreite mitbringen. Auch kann es je nach verwendeter Technologie zur Kommunikation mit den Clients notwendig sein, bestimmte Teile des serverseitigen Datenmodells, das die Domäne vollständig beschreibt, abzuändern. So kann beispielsweise, wie in Abschnitt 3.2 vorgestellt, Jax-WS keine bidirektionalen Assoziationen serialisieren, die mit Hilfe einer Sicht aufgelöst werden können.

Neben den technologisch bedingten Anforderungen der Clients kann es aber auch notwendig sein, nicht immer das vollständige Datenmodell zu übertragen, sondern eine Teilmenge dessen. Um dieser Anforderung gerecht zu werden, wurde die Sichtensprache, die es ermöglicht, auf Basis des vollständigen Domänenmodells ein verkleinertes Domänenmodell eines Clients zu spezifizieren, entwickelt. Es ist zudem möglich, mehrere Sichten zu einem Klassendiagramm zu spezifizieren, so dass alle erwarteten Clients oder Clienttypen abgedeckt werden können. Eine Sicht bezieht sich immer auf genau ein Bezugsklassendiagramm. Dadurch wird es möglich, dass nur eine Teilmenge der Domäne an den Client übertragen wird. Dabei kann es sich um ein Auslassen von Objekten bestimmter Typen, um ein Auslassen einzelner Attribute oder Assoziationen und um das "Flachklopfen" bestimmter Assoziationen handeln.

Abbildung 5.2 zeigt die Abhängigkeiten zwischen der Sichtensprache, ihrer Modelle und der bereits bekannten Grammatik der Klassendiagrammsprache und ihren Modellen. Dabei wird die Grammatik der Sichtensprache mit Hilfe von Sprachvererbung von der Grammatik der Klassendiagrammsprache abgeleitet. Mit Hilfe von Namensreferenzen können Sichten Elemente eines Klassendiagramms referenzieren.

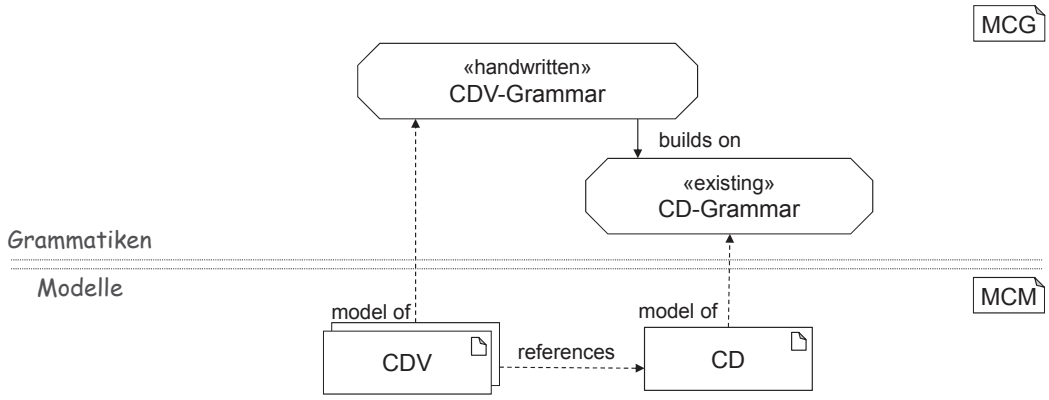


Abbildung 5.2: Abhängigkeiten zwischen der Sichtensprache und der Klassendiagrammsprache sowie ihrer Modelle.

Generell ist festzuhalten, dass die Sichtensprache auf unterschiedliche Arten eingesetzt werden kann. Zum einen kann sie, wie in der Literatur zu finden, zur Synthese eines Gesamtmodells dienen zum anderen zur Definition von Sichten auf ein existierendes Gesamtmodell. Dies ist semantisch ein Unterschied, da somit nicht die Sicht vorgibt, wie ein mögliches Modell aussieht, sondern die Sicht konsistent zu dem vollständigen Modell sein muss und somit im Wesentlichen Elemente ausblendet oder zusammenfasst. Ein Weglassen von Elementen bedeutet im Rahmen dieser Arbeit ein Weglassen eines Elements und nicht etwa eine Unterspezifikation wie in verwandten Arbeiten. Die Sprache selbst besitzt keine Einschränkungen hinsichtlich ihrer Verwendung. Im Rahmen dieser Arbeit werden die modellierten Sichten aber, analog zu Datenbanksichten [Mol06], als Sichten auf ein existierendes Gesamtmodell verwendet.

Verwandte Arbeiten definieren Sichten oder die Modellierung von Sichten meistens unterschiedlich. Ein mögliches Verständnis von Sichten ist die Annahme, dass verschiedene Rollen unterschiedliche Ansichten eines Systems besitzen. In [Rin14] und in [MRR13] werden verschiedene Sichten von Komponenten und Konnektor-Modellen zur Synthese eines Systemmodells verwendet. Dabei wird davon ausgegangen, dass die beteiligten Rollen unterschiedliche Expertise in die Modellierung der Komponenten einbringen und sich somit aus der Gesamtheit der Sichten ein Systemmodell ergibt. Dabei sind die Sichten die primären Artefakte und die Synthese kann an widersprüchlichen Modellen scheitern. Im Rahmen dieser Arbeit existiert ein gemeinsames Domänenmodell, welches die Basis einer Sicht darstellt, so dass die dort vorliegenden Probleme hier nicht auftreten. In [BGH+98] werden Sichten auf ein System als Modelle unterschiedlicher Blickwinkel auf ein System verstanden. Dabei können diese aus struktureller Sicht, aus Verhaltenssicht, aus Datensicht oder aus Schnittstellensicht modelliert sein. Je nach Blickwinkel existieren in der UML unterschiedlich geeignete Modellierungssprachen. Die Konsistenz von Sichten wird hier als Konsistenz der Beziehungen zwischen diesen Modellen verstanden. Im Rahmen dieser Arbeit wird die Sicht als eigenständiges Modell, das ebenfalls konsistent sein muss, verstanden. Allerdings sind die Sichten im Rahmen dieser Arbeit über die

unterschiedlichen Clients und nicht aus den unterschiedlichen Blickwinkeln eines Systems motiviert.

Darüber hinaus wird der Begriff der Sichtenmodellierung in der Modellierung von User-Interfaces verwendet. In [DRRS09] und [RR13] wird eine Sichtensprache zur Modellierung von Webseiten zur Bearbeitung eines Datenmodells verwendet. In beiden Fällen kommt eine eingeschränkte Version der Klassendiagramme zum Einsatz und Konzepte von Vererbung oder Navigationsrichtungen und Rollennamen werden nicht berücksichtigt. Auch können Assoziationen, wie bei der hier entworfenen Sichtensprache, nicht eingebettet werden.

Generell ist im Rahmen dieser Arbeit die Betrachtung modellierter Methoden nicht tiefergehend betrachtet worden. In [Sch13] werden diese sowohl auf Ebene der Grammatik aber auch auf Ebene der Kontextbedingungen definiert und werden hier nicht weiter betrachtet. Im nächsten Abschnitt wird die konkrete und die abstrakte Syntax der Sichtensprache im Detail vorgestellt.

5.2.1 Die Sichtensprache

Nachdem zuvor die Abhängigkeiten zwischen der Sichtensprache und der Klassendiagrammsprache und deren Modellen kurz erläutert wurden, wird in diesem Kapitel die abstrakte und die konkrete Syntax der Sichten vorgestellt.

```
1 classdiagramview ClientView of SocNet {                    CDV
2     // weitere Elemente der Sicht                          ...
3 }
```

Listing 5.3: Auszug der Definition der Sicht `ClientView` für das Klassendiagramm `SocNet`. Die Elemente der Sicht sind hier ausgelassen.

Jede Sicht kann wie auch Klassendiagramme und die weiteren im Rahmen dieser Arbeit vorgestellten Sprachen mit der nicht dargestellten optionalen Angabe eines Pakets beginnen. Listing 5.3 zeigt einen Ausschnitt einer modellierten Sicht auf das Domänenmodell des sozialen Netzwerks. Die Sicht beginnt dann mit dem Schlüsselwort `classdiagramview`, gefolgt von dem Namen der Sicht. Dieser Name muss zusammen mit dem Paketnamen als vollqualifizierter Name eindeutig unter allen existierenden Sichten sein. Nach dem Namen kann mit Hilfe des Schlüsselworts `of` und dem Namen des referenzierten Klassendiagramms eine Namensreferenz zu diesem aufgebaut werden. Der Klassendiagrammname kann ein vollqualifizierter Name sein, welcher das Klassendiagramm auf das sich die Sicht bezieht, referenziert. Ist der Klassendiagrammname nicht vollqualifiziert, so muss das Klassendiagramm im gleichen Paket liegen. Mit Hilfe dieser Angabe können im Rahmen der statischen Analyse verschiedene Eigenschaften, wie Existenz des Klassendiagramms oder auch Existenz der verwendeten Elemente geprüft werden. Dies wird in Abschnitt 5.2.2 detaillierter vorgestellt. Es können mehrere Sichten für ein Klassendiagramm, die jeweils das Domänenmodell eines Clients beschreiben

(vgl. FA7-PE und FA7-AG), modelliert werden. Innerhalb des Sichtenblocks folgt die Spezifikation der in der Sicht enthaltenen Elemente. Generell gilt hier der Grundsatz, dass alles, was in der Sicht enthalten sein soll, auch in der Sicht modelliert sein muss. Ein Weglassen von Elementen entspricht dabei einem gewollten Entfernen der Elemente. Dies dient der Vermeidung von Unterspezifikation, die dazu führen könnte, dass unklar ist, ob ein Element weggelassen werden sollte, oder ob es nicht beachtet wurde.

```
1  public full classview Person;                                    CDV
2                                                                   ...
```

Listing 5.4: Sicht auf die Klasse `Person` mit dem Modifikator `full` als Element der Sicht aus Listing 5.3.

Listing 5.4 zeigt ein Element der Sicht. Dabei wurde die konkrete Syntax der Sichtensprache bewusst sehr nah an der konkreten Syntax der bereits existierenden Klassendiagrammsprache entworfen, damit der Aufwand des Einarbeitens und Neuerlernens möglichst klein gehalten wird. Die meisten Schlüsselwörter der Klassendiagramme wurden übernommen und um das Suffix `-view` erweitert. Dadurch wird eine Verwechslung mit Klassendiagrammen ausgeschlossen, aber ein direkter Wiedererkennungseffekt erreicht.

Damit, falls die Klasse vollständig Teil der Sicht ist, nicht immer alle Attribute und Methoden einer Klasse des Klassendiagramms wiederholt werden müssen, wurde das Schlüsselwort `full` eingeführt. Dieses Schlüsselwort bedeutet, dass ein Element des Klassendiagramms vollständig Teil der Sicht ist. In Listing 5.4 ist modelliert, dass die Klasse `Person` vollständig Teil der Sicht ist und keine Attribute ausgelassen werden.

```
1  public classview Profile {                                       CDV
2    String userName;                                               ...
3  }
```

Listing 5.5: Sicht auf die Klasse `Profile` als Element der Sicht aus Listing 5.3. Eine Teilmenge der Attribute der Klasse `Profile` wurde in die Sicht übernommen.

Listing 5.5 zeigt die Modellierung eines nur zum Teil übernommenen Elements. Dabei ist modelliert, dass die Klasse `Profile` in der Sicht enthalten ist. Die Klasse `Profile` enthält in der Sicht das Attribut `userName`, das Attribut `password` ist nicht Teil der Sicht und wurde demnach weggelassen. Das Weglassen von Elementen und auch Assoziationen kann zum einen dazu verwendet werden, die sichtbaren Daten einzuschränken, aber auch, um die Menge der zum Client transferierten Daten zu reduzieren, so dass auch Thin-Clients effizient bleiben können (vgl. FA7.2-AG). Um dies zu modellieren, wird die Klasse `Profile` sehr ähnlich zu ihrer Modellierung im Klassendiagramm angegeben, allerdings wurde das Schlüsselwort `class` durch das Schlüsselwort `classview`

ersetzt. Diese Veränderung der Schlüsselwörter wurden für die Elemente `class`, `interface`, `enum`, `association` des Klassendiagramms durchgeführt. Die Schlüsselwörter `classview`, `interfaceview`, `enumview` und `associationview` wurden entsprechend eingeführt. Die restliche Syntax entspricht exakt der der Klassendiagramme. Somit werden Sichtbarkeiten, Konstruktoren, Methoden und Attribute wie bekannt angegeben. Dabei müssen vorhandene Sichtbarkeiten, Konstruktoren, Methoden und Attribute redundant angegeben werden, sofern sie Teil der Sicht sind. Eine Reduktion der Sichtbarkeiten ist nicht vorgesehen. Die Erweiterung der Schlüsselwörter wurde zur Explizierung der Sichtenmodellierung, so dass eine Abgrenzung zu den Klassendiagrammen erfolgt, vorgenommen. Die Semantik, dass weggelassene Elemente nicht Teil der Sicht sind, bedingt eine Reihe von Konsistenzbedingungen, da der Modellierer sich für die ausgelassenen Elemente auch um etwaige Vererbungshierarchien oder Assoziationen kümmern muss. Im Rahmen dieser Arbeit wurde bewusst keine automatisierte Vervollständigung oder Erweiterung der Sicht umgesetzt, damit die Semantik des expliziten Weglassens nicht abgeschwächt wird. Stattdessen wird der Modellierer mit Hilfe geeigneter Kontextbedingungen auf die Probleme aufmerksam gemacht. Die Kontextbedingungen werden in Kapitel 5.2.2 vorgestellt.

CDV

```
1  flat association sent [1] Profile -> Post [*];
2                                                    ...
3  flat association [1] Commercial -> Tariff [1];
```

Listing 5.6: Sicht auf zwei Assoziationen als Elemente der Sicht aus Listing 5.3 mit dem Modifikator `flat`.

Listing 5.6 zeigt zwei weitere Elemente der Sicht und eine weitere Möglichkeit der Sichtensprache. Das Schlüsselwort `flat` dient dabei dem "Flachklopfen" einer Assoziation. Eine flachgeklopfte Assoziation bewirkt dabei, dass die Attribute der Zielklasse in die Ausgangsklasse eingebettet werden. Bei Assoziationen mit einer Zielkardinalität von `1` ist dies einfach möglich, da die Attribute der Ausgangsklasse hinzugefügt werden können. Bei anderen Zielkardinalitäten muss dies anderweitig erfolgen. In Listing 5.6 ist zum einen dargestellt, dass die Attribute der Enumeration `Tariff` in die Klasse `Commercial` innerhalb der Sicht eingebettet sind, zum anderen, dass die Attribute der Posts in die Klasse `Profile` eingebettet werden. Generell können auch Teile der Assoziation weggelassen werden. Dies gilt für den Assoziationsnamen, die Rollennamen oder auch ein Teil der Navigationsrichtung bei bidirektionalen Assoziationen.

Ein weiterer Punkt der Semantik des `flat` Modifikators ist dabei das Verhalten bei verketteten Assoziationen. Diese Assoziationen können ebenfalls als `flat` markiert sein, so dass sie transitiv eingebettet werden müssen, oder es muss eine neue Assoziation eingeführt werden, die zu der über Verkettung assoziierten Klasse führt. Die genaue Semantik des `flat` Modifikators wird dabei in Abschnitt 5.2.3 detailliert vorgestellt und diskutiert. Ebenso werden Konsistenzbedingungen im Abschnitt 5.2.2 vorgestellt.

Auf Basis dieser Sicht und dem zugehörigen Klassendiagramm kann ein neues Klas-

sendiagramm, das nur die Elemente der Sicht enthält, erzeugt werden. Dieses Klassendiagramm dient neben dem ursprünglichen Klassendiagramm im weiteren Verlauf dieser Arbeit als Eingabe für die Codegeneratoren. Dies hat den Vorteil, dass die Generatoren nur Klassendiagramme verarbeiten können müssen und somit auch direkt für die Sichten verwendet werden können. Die Transformation der Sicht in ein neues Klassendiagramm wird in Abschnitt 5.2.3 vorgestellt. Zusammenfassend lassen sich mit Hilfe der Sichtensprache Domänenmodelle, die auf die spezifischen Charakteristika heterogener Clients ausgelegt sind und somit neben unterschiedlicher Funktionalität auch unterschiedliche Performanz berücksichtigen (vgl. FA7.1-AG und FA7.2-AG), modellieren.

Die Grammatik der Sichtensprache

Nachdem zuvor die konkrete Syntax der Sichtensprache vorgestellt wurde, wird in diesem Abschnitt die Grammatik der Sprache vorgestellt. Dabei werden nur Ausschnitte der Grammatik zur Erläuterung der Konzepte gezeigt.

```
1  grammar CDV extends mc.umlp.cd.CD {                                MCG
2                                                                      CDV
3    ViewDefinition extends Definition =                               ...
4      "classdiagramview" Name "of" QualifiedName
5      "{"
6        (
7          CDClassView | CDInterfaceView | CDEnumView |
               CDAssociationView
8        )*
9      "}";
10
11   // weitere Produktionen
12 }
```

Listing 5.7: Aufbereiteter Auszug der Grammatik der Sichtensprache. Dargestellt ist die Startproduktion `ViewDefinition`. Weitere Produktionen der Grammatik sind ausgelassen. Die vollständige Version der Grammatik ist in Anhang C.6 dargestellt.

Listing 5.7 zeigt den ersten Ausschnitt der Grammatik `CDV`, die von der Grammatik der Klassendiagrammsprache erbt, um die dort verwendeten Produktionen wiederzuverwenden und geeignet zu erweitern. Die Produktion `ViewDefinition` erweitert dazu die ursprüngliche Produktion `Definition`. Das ursprüngliche Schlüsselwort wird zu `classdiagramview` geändert und die Möglichkeit zur Referenzierung des Klassendiagramms wird mit Hilfe des Schlüsselworts `of` und des Namens hinzugefügt. Zudem werden die im Klassendiagramm enthaltenen Elemente abgeändert und auf Elemente der Sicht geändert. Als modellierbare Elemente der Sicht sind Klassen, Interfaces, Enumerationen und Assoziationen definiert.

```
1  ClassView extends Class =                                          MCG
2    Modifier "classview" Name [...]                                  CDV
3    (                                                                ...
4      ( "{" ( CDAttribute | CDConstructor | CDMethod )* "}" )
5      | ";"
6    );
```

Listing 5.8: Darstellung der Produktion `ViewDefinition` der Grammatik aus Listing 5.7 zur Modellierung einer Sicht auf Klassen. Gezeigt sind die veränderten Schlüsselwörter und die Nutzung der Vererbung bei Produktionsregeln. Die vollständige Version der Grammatik ist in Anhang C.6 dargestellt.

Innerhalb der Grammatik sind weitere Produktionen definiert, die die Elemente der Sicht definieren. Listing 5.8 zeigt die Produktion `ClassView`, welche eine Sicht auf eine Klasse einleitet. Im Wesentlichen wurde in dieser Produktion das Schlüsselwort `classview` eingeführt. Innerhalb der Klasse werden die ursprünglichen Produktionen für Attribute, Methoden und Konstruktoren der Klassendiagrammsprache wiederverwendet, so dass dort auch die Syntax der ursprünglichen Sprache verwendet wird. Alternativ kann die Definition der Sicht auf eine Klasse direkt mit einem Semikolon ohne die Angabe von Elementen beendet werden. Zudem wird das Nichtterminal `Modifier` in der Produktion `classview` verwendet. Dieses Nichtterminal ist in Listing 5.9 gezeigt. Modellierte Methoden werden bei Angabe in der Sicht übernommen. Eine Veränderung der Methodenrümpfe oder ihrer Implementierung ist nicht vorgesehen.

Diese Produktion überschreibt die ursprüngliche Produktion der Klassendiagrammsprache und fügt die beiden Schlüsselwörter `flat` und `full` den möglichen Modifikatoren hinzu. Da das Nichtterminal in der Produktion `ClassView`, aber auch in den anderen Elementen verwendet wird, können beide Modifikatoren vor die jeweiligen Elemente geschrieben werden.

Die vollständige Version der Grammatik, die Definition von Sichten auf Interfaces, Enumerationen und Assoziationen, ist im Anhang in Listing C.6 gegeben. Im nachfolgenden Abschnitt werden die Kontextbedingungen, die die Konsistenz der Sicht zum ursprünglichen Klassendiagramm sichern, vorgestellt.

5.2.2 Kontextbedingungen

In diesem Kapitel werden Kontextbedingungen, welche die Sichtensprache weiter einschränken, vorgestellt. Die Kontextbedingungen wurden in [Sch13] gemeinsam identifiziert und im Rahmen dieser Arbeit abgeändert und angepasst. Diese Kontextbedingungen sind wichtig, da die Sprache eine Vielzahl von Möglichkeiten erlaubt, die semantisch nicht zulässig sind. Dies resultiert aus der erwünschten Semantik des Weglassens. Diese besagt, dass weggelassene Elemente nicht Teil der Sicht und somit nicht Teil des Domänenmodells des Clients sind. Die Möglichkeit zur Unterspezifikation ist nicht vorgesehen.

```
1  Modifier =                                                          MCG
2    (                                                                 CDV
3        "full" | "flat" | "public" | "+" | "private" | "-"            ...
4      | "protected" | "#" | "final" | "abstract" | "local"
5      | "derived" | "/" | "readonly" | "?" | "static"
6    )*;
```

Listing 5.9: `Modifier` Produktion der Grammatik aus Listing 5.7 zur Einführung der neuen Modifikatoren `full` und `flat`. Die vollständige Version der Grammatik ist in Anhang C.6 dargestellt.

Dies bedeutet auch, dass Vererbungshierarchien verletzt oder Assoziationen nicht korrekt enthalten sein können. Es gibt Kontextbedingungen, die die Existenz von Elementen sicherstellen. Auch hier ist nicht nur die Existenz der Elemente, sondern ebenfalls die korrekte Definition ebendieser Elemente, relevant. Darüber hinaus können die neu eingeführten Modifikatoren nur für bestimmte Elemente verwendet werden. Die Grammatik erlaubt hingegen eine beliebige Verwendung.

Daher beschreiben die folgenden Kontextbedingungen, welche Elemente mit welchen Modifikatoren versehen werden können. Die Kontextbedingungen wurden in [Sch13] gemeinsam identifiziert und im Rahmen dieser Arbeit abgeändert und angepasst.

CDV-1

Bedingung: Die Modifikatoren `flat` und `full` können nur für Klassen, Enumerationen, Interfaces und Assoziationen verwendet werden. Da die Produktion `Modifier` erweitert wurde, ist dies syntaktisch, aber nicht semantisch, auch bei Attributen oder anderen Elementen möglich.

Schweregrad: Fehler

CDV-2

Bedingung: Der Modifikator `flat` kann nur bei Assoziationen verwendet werden. Eine Verwendung bei anderen Elementen, wie Klassen, Interfaces und Enumerationen ist nicht zulässig.

Schweregrad: Fehler

CDV-3

Bedingung: Der Modifikator `full` kann nur bei Klassen, Interfaces oder Enumerationen verwendet werden. Eine Verwendung bei Assoziationen ist nicht zulässig.

Schweregrad: Fehler

Zur Sicherung der Konsistenz existieren eine Reihe von Kontextbedingungen, die im Wesentlichen das gültige Weglassen von Elementen, aber auch die Existenz der Elemente und deren Typ sichern.

CDV-4

Bedingung: Das aus der Sicht referenzierte Diagramm muss existieren und ein Klassendiagramm darstellen.

Schweregrad: Fehler

CDV-5

Bedingung: Innerhalb einer Sicht referenzierte Klassen, Interfaces und Enumerationen müssen innerhalb des referenzierten Klassendiagramms mit dem richtigen Typ existieren. Dabei hängt die Typprüfung davon ab, ob eine Sicht auf eine Klasse, Interface oder Enumeration angegeben wurde. Die Existenzprüfung erfolgt über den eindeutigen Namen der Elemente

Schweregrad: Fehler

Wird der Modifikator `full` verwendet, ist die Existenz und die Typsicherung der einzelnen Elemente auf dieser Ebene ausreichend. Werden die Elemente aber genauer modelliert und Subelemente ebenfalls als Teil der Sicht modelliert, muss die folgende Kontextbedingung ebenfalls gelten.

CDV-6

Bedingung: Wird innerhalb einer Klasse, eines Interfaces, oder einer Enumerationen ein Attribut oder eine Enumerationskonstante referenziert, muss dieses innerhalb des Elements oder innerhalb der Vererbungshierarchie im Klassendiagramm existieren und zudem den gleichen Typ sowie, mit Ausnahme von `flat` und `full`, die gleichen Modifikatoren besitzen. Die Existenz des Attributs innerhalb der Vererbungshierarchie ist nur zugelassen, wenn das Element, welches das Attribut enthält, in der Sicht weggelassen wurde. Die Existenzprüfung erfolgt auf Basis des eindeutigen Namens der Elemente

Schweregrad: Fehler

Auch die Konsistenzprüfung für Assoziationen gestaltet sich, da eine Assoziation aus mehreren Teilelementen besteht und zudem nicht zwingend über einen eindeutigen Namen identifizierbar ist, umfangreicher.

CDV-7

Bedingung: Innerhalb einer Sicht referenzierte Assoziationen müssen innerhalb des referenzierten Klassendiagramms existieren. Die Existenzprüfung erfolgt

dabei über den optionalen Assoziationsnamen. Ist dieser nicht angegeben, werden die Typen, die die Assoziation verbindet, zusammen mit den Rollennamen beider Seiten verwendet.

Schweregrad: Fehler

CDV-8

Bedingung: Falls in der Sicht ein Assoziationsname, Rollenname oder Qualifikator angegeben ist, muss er dem Assoziationsnamen, Rollennamen oder Qualifikator der Assoziation im Klassendiagramm entsprechen.

Schweregrad: Fehler

CDV-9

Bedingung: Die in der Sicht angegebene Assoziationsrichtung muss der Assoziationsrichtung der Assoziation des Klassendiagramms entsprechen. Alternativ, ist das Weglassen eines Teils einer bidirektionalen Assoziation möglich.

Schweregrad: Fehler

CDV-10

Bedingung: Die in der Sicht angegebenen Typen und Kardinalitäten der Assoziationen müssen den Typen und Kardinalitäten der Assoziation des Klassendiagramms entsprechen.

Schweregrad: Fehler

Wie bereits zuvor beschrieben, ist es möglich, dass das Weglassen von Elementen innerhalb einer Vererbungshierarchie zu einer inkonsistenten Hierarchie führt. Zur Vermeidung müssen die folgenden Kontextbedingungen geprüft werden:

CDV-11

Bedingung: Eine Sicht auf eine Klasse oder ein Interface muss grundsätzlich von den gleichen Elementen wie die entsprechende Klasse oder das entsprechende Interface des Klassendiagramms erben. Wird eine direkte Superklasse oder ein direktes Superinterface weggelassen, muss die Sicht auf die Klasse oder das Interface von allen direkten Superklassen oder allen direkten Superinterfaces der weggelassenen Klasse oder des weggelassenen Interfaces erben.

Schweregrad: Fehler

CDV-12

Bedingung: Eine Sicht auf eine Klasse oder eine Enumeration muss grundsätzlich die gleichen Elemente implementieren wie die entsprechende Klasse oder Enumeration des Klassendiagramms. Wird ein direktes Superinterface

weggelassen, muss die Sicht auf die Klasse oder die Enumeration alle direkten Superinterfaces des weggelassenen Interfaces implementieren.

Schweregrad: Fehler

Das Erben von mehreren Klassen oder das Implementieren mehrerer Interfaces benötigt die Möglichkeit zur Modellierung von Mehrfachvererbung. Dies ist in der Klassendiagrammsprache möglich und kann dementsprechend aufgelöst werden. Allerdings muss ein Codegenerator je nach Zielsprache eine geeignete Abbildung der Mehrfachvererbung unterstützen. Aber auch bei der Verwendung des Modifikators `flat` muss beachtet werden, dass die folgenden Kontextbedingungen geprüft werden.

CDV-13

Bedingung: Das Einbetten von Assoziationen führt zu neuen Attributen in einer Klasse. Diese Klasse darf kein Attribut beinhalten, welches zu einer Namenskollision mit den neuen Attributen führt.

Schweregrad: Fehler

CDV-14

Bedingung: Eingebettete Assoziationen dürfen keinen Kreis bilden.

Schweregrad: Fehler

CDV-15

Bedingung: Eine Assoziationen kann nur mit dem Modifikator `flat` markiert sein, wenn alle ausgehenden Assoziation der Zielklasse den Modifikator `flat` besitzen und deren Zielkardinalität größer `1` ist.

Schweregrad: Fehler

Nachdem die Kontextbedingungen zur Konsistenzsicherung vorgestellt wurden, wird im folgenden Abschnitt die Transformation der Sicht in ein neues Klassendiagramm gezeigt.

5.2.3 Transformation von Sichten in Klassendiagramme

Nachdem zuvor die Grammatik der Sichtensprache und ihre Kontextbedingungen vorgestellt wurden, wird in diesem Abschnitt die Transformation einer Sicht in ein Klassendiagramm vorgestellt. Die Transformation ist in einem CDV-CD-Transformator umgesetzt worden. Diese Transformation ist sinnvoll, damit die implementierten Generatoren für Klassendiagramme wiederverwendet werden können. Die Möglichkeit zur Wiederverwendung resultiert daraus, dass die Generatoren keine Modelle unterschiedlicher Sprachen

handhaben müssen. Aus diesem Grund kann der Produktentwickler Generatoren in unterschiedlichen Kontexten und der Werkzeugentwickler implementierte Teile wiederverwenden. Die Transformation von Sichten hin zu Klassendiagrammen kann dazu algorithmisch erfolgen. Dabei ist die Prämisse, dass weggelassene Elemente eine Auslassung darstellen, von Vorteil, da dadurch keine Unterspezifikation auftritt. Zudem wurde die Sicht bereits durch die Kontextbedingungen geprüft und wird als valide angesehen.

Zunächst wird ein leeres Klassendiagramm in Form eines Klassendiagramm-AST erstellt, der schrittweise befüllt wird und abschließend mit Hilfe eines PrettyPrinters wieder als Klassendiagramm textuell ausgegeben wird.

In einem ersten Schritt werden die Klassen betrachtet. Eine Klasse die den Modifikator `full`, der in Listing 5.4 vorgestellt wurde, besitzt, wird vollständig in das entstehende Klassendiagramm übernommen. Bei der Übernahme der mit `full` markierten Klasse muss geprüft werden, ob alle Superklassen oder Superinterfaces ebenfalls Teil der Sicht sind. Falls dies nicht so ist, wurden diese ausgelassen und ihre Attribute müssen in die Subklasse übernommen werden. Für alle Klassen, die nicht mit dem Modifikator `full` markiert sind, werden die in der Sicht modellierten Attribute und Beziehungen verwendet. Der AST-Knoten der Sicht, der eine Subklasse der AST-Knoten der Klassendiagramme ist, kann also technisch in den Klassendiagramm AST umgehängt werden, so dass kein neuer Knoten erzeugt werden muss. Hierbei muss die zugehörige Klasse nicht mit Hilfe der Symboltabelle aufgelöst werden. Durch die Kontextbedingungen wird sichergestellt, dass die Attribute einer etwaigen weggelassenen Superklasse oder Superinterface bereits modelliert sind.

Der Modifikator `flat`, der in Listing 5.6 vorgestellt wurde, ist komplexer: Zunächst werden eingebettete Assoziationen als Attribute in einem Klassendiagramm dargestellt. Dies bedeutet, dass alle Attribute der Zielklasse der Assoziation Teil der Ausgangsklasse der Assoziation werden. Diese Attribute werden in einer internen Tagdefinition mit dem Tag `flattened` getaggt. Dies wird benötigt, da die in den Kapitel 7 und 8 beschriebenen Generatoren zwar Klassendiagramme als Eingabemodelle verwenden, aber Informationen über Attribute, die als Ergebnis flacher Assoziationen Teil der Klasse sind, benötigen. Sie benötigen dies aus zwei Gründen: zum einen müssen die Generatoren eine Infrastruktur, die in Abschnitt 8.2 vorgestellt wird, generieren, die aus Entitäten DTOs und umgekehrt erzeugt, so dass sie die Kenntnis über die Herkunft eines Attributs benötigen. Zum anderen müssen die Generatoren Infrastruktur erzeugen, die sich um die Synchronität der Attributwerte kümmert. Es kann sein, dass zwei unterschiedliche Objekte einen Link zum gleichen Objekt besitzen. Wird die zugehörige Assoziation in der Sicht flachgeklopft, so werden die Attribute in die Ausgangsklassen eingebettet. Ändern sich die eingebetteten Attribute innerhalb des einen Objekts, müssen auch die Attribute des anderen synchronisiert werden, da sonst die Semantik des Links auf das gleiche Objekt verändert wurde. Dies kann mit Hilfe von Observern umgesetzt werden.

Wird zunächst eine Assoziation betrachtet, deren Zielklasse keine weiteren Assoziationen besitzt und deren Zielkardinalität `1` ist, können die Attribute einfach übernommen werden. Dies ist in Abbildung 5.10 ganz links dargestellt. Dazu werden die Symbole mit Hilfe der Symboltabelle aufgelöst und dem AST-Knoten, wie in Abschnitt 2.2 gezeigt,

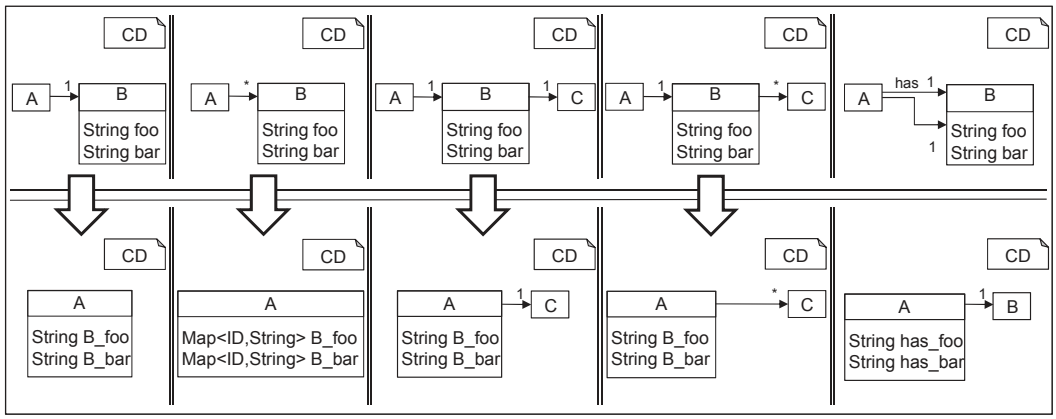


Abbildung 5.10: Darstellung der unterschiedlichen Auswirkungen eingebetteter Assoziationen.

hinzugefügt. Der Name der hinzugefügten Attribute wird dazu mit Hilfe eines Präfixes abgeändert. Der neue Name setzt sich aus dem Namen der Zielklasse, einem Unterstrich und dem Namen des Attributs zusammen. Betrachtet man die gleiche Assoziation wieder, diesmal mit einer Zielkardinalität, die größer `1` ist, müssen die Attribute mehrerer Instanzen gespeichert werden können. Zudem bietet die Zielklasse eine Gruppierung von Attributmengen. Diese Zuordnung darf nicht verloren gehen. Daher werden die Attribute als `Maps` über den Datentyp des Attributs in der neuen Klasse dargestellt. Für jedes Attribut einer Klasse existiert eine separate `Map`. Diese Map beinhaltet den Attributwert als Wert und eine eindeutige ID, beispielsweise die Objekt-ID, als Schlüssel. Dadurch können alle Attributwerte eines Objekts in den verschiedenen Maps wiedergefunden werden. Dies ist in Abbildung 5.10 im zweiten Abschnitt dargestellt.

Darüber hinaus kann die Zielklasse auch ausgehende Assoziationen haben. Diese Assoziationen können ebenfalls mit dem Modifikator `flat`, der in Listing 5.6 dargestellt wurde, markiert, oder nicht markiert sein. Sind sie nicht markiert, werden die Attribute der Zielklasse in die Ausgangsklasse übernommen und die Assoziation wird umgehängt. Sie geht von der Ausgangsklasse direkt zur Zielklasse der Assoziation. Dabei werden die Kardinalitäten übernommen. Dies ist nur möglich, wenn die Zielkardinalität `1` ist, da sonst die Zuordnung zwischen Attributen und assoziierten Objekten verloren ginge. Dies wird bereits zuvor durch die Kontextbedingungen geprüft. Ist die ausgehende Assoziation der Zielklasse ebenfalls als `flat` markiert, so werden die Attribute transitiv in die Ausgangsklasse eingebettet. Die Namen werden dabei ebenfalls, wie zuvor beschrieben, abgeändert und über die Klassennamen konkateniert. Dies ist in Abbildung 5.10 im dritten Abschnitt gezeigt. Eine weitere Möglichkeit ist das Einbetten einer Zielklasse, die zwar die Zielkardinalität `1` besitzt, aber selbst wiederum Assoziationen besitzt, deren Zielkardinalität $\star$ entspricht. Diese werden wie im vierten Abschnitt in Abbildung 5.10 gezeigt eingebettet. Die letzte Möglichkeit, die in Abbildung 5.10 ganz rechts gezeigt ist, ist die Möglichkeit, dass die Ausgangsklasse mehrere Assoziationen zur Zielklasse besitzt.

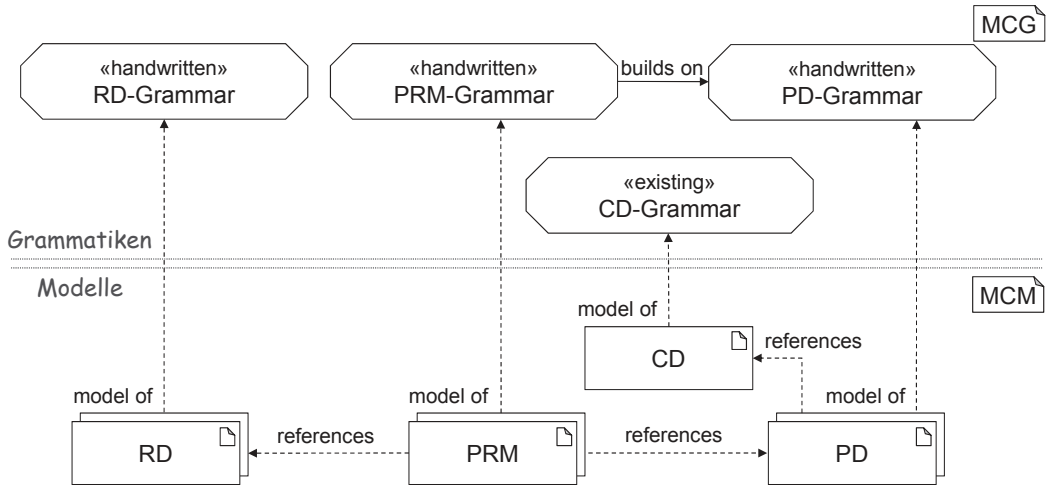


Abbildung 5.11: Abhängigkeiten zwischen den Grammatiken der Rollensprache zur Modellierung von Rollen, der Rechtesprache zur Modellierung von Rechten und der Mappingsprache zur Modellierung der Zuordnung zwischen Rechten und Rollen sowie die zugehörigen Modelle.

Auch diese können individuell eingebettet werden. Dazu wird die Eigenschaft, dass auch Assoziationen eindeutig identifizierbar sind, verwendet. Dabei gilt, dass eine Assoziation zwischen zwei Klassen sich eindeutig identifizieren lässt, wenn sie die einzige Assoziation zwischen diesen Klassen ist, oder aber einen Assoziationsnamen oder Rollennamen besitzt. Beim Einbetten werden die Assoziations- oder Rollennamen dazu verwendet, die Attribute in der Ausgangsklasse zu erzeugen.

5.3 Rollenbasierte Zugriffskontrolle

Nachdem im vorangegangenen Kapitel 4 Taggingsprachen zur Anreicherung von Modellen um technologiespezifische Informationen und eine Sprache zur Modellierung von Sichten auf Klassendiagramme zur Unterstützung verschiedener Clients vorgestellt wurden, werden Sprachen zur Modellierung von Rechten und Rollen und der Zuordnung zwischen diesen im folgenden Abschnitt vorgestellt. Dabei werden drei domänenspezifische Sprachen zur Modellierung einer rollenbasierten Zugriffskontrolle vorgestellt, die auf der betreuten Vorarbeit [Cöm13] beruhen. Diese Zugriffskontrolle ist in einem Mehrbenutzersystem unerlässlich, um ein Rechtemanagement zu ermöglichen und den Zugriff auf Funktionen des Systems zu regulieren. Ein Mehrbenutzersystem wird dabei dadurch definiert, dass es viele Benutzer hat, die unterschiedliche Daten und unterschiedlichen Zugriff auf Daten besitzen. Die Daten liegen auf einer zentralen Instanz und die Benutzer greifen über verschiedene Clients darauf zu.

Zur Beschreibung der statischen Rechte (vgl. FA9-PE), Rollen (vgl. FA8-PE) und deren Zuordnung wurden im Rahmen dieser Arbeit drei unterschiedliche DSLs, die eine

Modellierung dieser Sachverhalte ermöglichen, entwickelt. Abbildung 5.11 zeigt die Abhängigkeiten zwischen den beteiligten Sprachen. Es wurde eine Sprache zur Definition von Rollen *(engl. roles)*, die in Abschnitt 5.3.1 vorgestellt wird, und deren Hierarchie geschaffen. Die Rollendiagramme modellieren alle im System existierenden Rollen, welche von den anderen Modellen referenziert werden. Somit besitzen Rollendiagramme selbst keine Referenzen auf andere Elemente. Darüber hinaus wurde eine Sprache zur Modellierung von Rechten *(engl. permissions)*, die in Abschnitt 5.3.3 vorgestellt wird, geschaffen. Rechtediagramme modellieren alle im System verfügbaren Rechte und referenzieren Elemente aus dem Klassendiagramm und zugehörigen Tags. Zudem wird in Abschnitt 5.3.4 ein Algorithmus zur automatisierten Erstellung von Rechtediagrammen, die Referenzen auf Klassendiagramme beinhalten, gezeigt. Zudem wurde eine Sprache, die unterschiedlichen Rollen Rechte zuordnet, geschaffen. Dies ermöglicht es, Rechte und Rollen getrennt voneinander zu definieren, unterschiedlichen Rollen mehrere Rechte und mehreren Rollen das gleiche Recht zuzuordnen. Auf Ebene der Sprache werden Produktionen der Grammatik der Rechtesprache in die Grammatik der Mappingsprache mit Hilfe der in Abschnitt 2.2 vorgestellten Mechanismen eingebettet. Daher baut die Grammatik der Mappingsprache auf der Grammatik der Rechtesprache auf. Auf Ebene der Modelle referenzieren Mappingdiagramme Elemente aus Rollen- und Rechtediagrammen. Diese Trennung ermöglicht die Wiederverwendung modellierter Teile des Gesamtsystems und erhöht gleichzeitig die Modularität einzelner Aspekte.

Die grundlegende Idee basiert dabei auf dem Modell der Rollenbasierten Zugriffskontrolle (RBAC), die formal in [FKC92] vorgestellt und 2001 dem National Institute of Standards and Technology (NIST) als Standard vorgestellt wurde [FSG+01]. Dieses grundlegende Konzept beruht darauf, dass es im System die Konzepte von Nutzern, Rollen und Rechten gibt. Dabei haben Nutzer mehrere Rollen, welche wiederum verschiedene Rechte haben. Im System authentisierte Nutzer haben dabei eine aktive Rolle, in der sie Operationen im System ausführen. Sie können ihre Rolle auch wechseln. Im Rahmen dieser Arbeit werden die vom Applikationsserver bereitgestellten Konzepte einer Benutzerverwaltung und deren Authentisierung verwendet. Diese Funktionalität wird zur Laufzeit des Systems benötigt. Zum Zeitpunkt des Deployments des Systems, müssen die zur Verfügung stehenden Rollen und Rechte bereits bekannt sein. Daher werden diese im Rahmen der Arbeit mit einer DSL modelliert. Auch wenn die Modelle nur für den statischen Teil verwendet werden, sind sie doch so entworfen, dass sie auch eine dynamische Autorisierung zur Laufzeit des Systems ermöglichen. Die Modelle enthalten alle benötigten Informationen, lediglich der Generator, der in Kapitel 8 vorgestellt wird, muss zur Unterstützung dynamischer Autorisierung erweitert werden.

Bei der Autorisierung von Nutzern wird geprüft, ob die aktive Rolle eines Nutzers das benötigte Recht hat oder über dieses Recht nicht verfügen. In [FCK95] wird eine gute Übersicht über die unterschiedlichen Mechanismen gegeben und hier kurz dargestellt.

Historisch existieren vier verschiedene Formen von Role Based Access Control (RBAC) [FCK95]: RBAC0, welches die Basisvariante mit Nutzern, Rechten und Rollen darstellt. RBAC1, welches hierarchische Rollen hinzufügt; RBAC2, das RBAC1 nicht erweitert, sondern RBAC0 um Bedingungen erweitert und RBAC4, welches die Vereinigung von

RBAC1 und RBAC2 darstellt. Im Rahmen dieser Arbeit wird RBAC1 durch einen Generator umgesetzt, da dieser die vom Applikationsserver zur Verfügung gestellte Funktionalität ausnutzt. Die Modellierungssprachen hingegen ermöglichen ferner die Umsetzung von Attribute-Based Access Control (ABAC). ABAC wird im weiteren Verlauf dieses Abschnitts vorgestellt. Neben RBAC existieren Discretionary Access Control (DAC) und Mandatory Access Control (MAC) als verbreitete Sicherheitsmechanismen und Standards [Eck13]. Eine detaillierte Analyse und Darstellung des Zusammenhangs zwischen RBAC und DAC ist in [SM98] dargestellt, wohingegen der Zusammenhang zwischen RBAC und MAC in [Osb97] dargestellt ist.

DAC basiert dabei darauf, dass der Besitzer eines Objekts dieses für andere Nutzer freigeben kann. Besitzer eines Objekts ist immer der Ersteller des Objekts, was in Organisationen und Unternehmen nicht immer der fachlichen Realität entspricht. Die Autorisierung wird bei DAC meistens mit Access Control Lists (ACLs), die im Wesentlichen die Erlaubnis oder das Verbot speichern, welcher Nutzer ein Objekt modifizieren darf, umgesetzt. Generell gibt es dabei zwei grundlegende Verhaltensweisen: nicht spezifizierte Operationen werden erlaubt oder verboten. Mögliche Modifikationen sind dabei meistens die typischen CRUD-Operationen. Im Rahmen dieser Arbeit wurden für die RBAC Umsetzung die Verhaltensweise des Verbietens und die CRUD-Operationen als mögliche Modifikationen gewählt. Vor allem basiert DAC auf den eigentlichen Nutzern und ihrer Repräsentanz im System, so dass jeder neue Nutzer zunächst überall hinzugefügt werden muss. Das Konzept von Gruppen von Nutzern schafft hier Abhilfe. Durch die Verwendung von Gruppen ist dieses Konzept sehr nah an RBAC.

MAC stellt einen hohen Sicherheitsstandard dar [Eck13]. Dabei hat jeder Nutzer ein Sicherheitslevel. Die im System bekannten Sicherheitslevel besitzen eine totale Ordnung. Dadurch wird eine Autorisierung für jedes Sicherheitslevel ermöglicht. Dabei gilt das *read down* und das *write up* Prinzip. Dies bedeutet, dass ein Sicherheitslevel seine eigenen Objekte und alle niedrigeren Objekte lesen kann. Demgegenüber können Objekte höherer Sicherheitslevel höchstens geschrieben, also erstellt, aber dann nicht mehr gelesen werden. Basierend auf MAC existieren Implementierungen, wie das Bell LaPadula-System [BL73], das Biba-System [Bib77] oder auch das LoMAC-System [Fra00]. Darüber hinaus existieren Systeme, die neben den Sicherheitsleveln auch orthogonale Zugriffsrechte beinhalten. Diese Systeme werden als multilateral bezeichnet. Hier wären das Lattice-Modell [Den76] oder das Brewer-Nash-Modell [BN89] zu nennen.

Darüber hinaus wird mit ABAC eine Möglichkeit geschaffen, die Autorisierung auf Basis bestimmter Attribute umzusetzen [PDM+05]. Dabei werden nicht mehr die Rechte betrachtet, sondern Attribute und Attributwerte. Die Definitionen, welche Werte welcher Attribute autorisiert werden, werden als *claims* bezeichnet. Dabei wird in der Literatur häufig das Beispiel verwendet, dass ein Nutzer zur Anmietung eines Autos nachweisen muss, dass er älter als 18 Jahre ist. Sein Attribut *alter* muss also größer als 18 sein. Verglichen mit RBAC beziehen sich die *claims* auf Eigenschaften der Instanzen und nicht auf statische Rechte. Somit ist ABAC feingranularer und flexibler, erhöht aber die Komplexität.

Auch im Bereich der DSLs und der Modellierung dieser Mechanismen existieren ver-

schiedene Ansätze. Die beiden bekanntesten sind SecureUML [LBD02] und die eXtensible Access Control Markup Language (XACML) [GAP+02].

SecureUML ist dabei ein Sprachprofil der UML zur Modellierung von RBAC. Die Hauptelemente sind Rollen, Rechte und Nutzer, analog zum RBAC-Modell. Die Autorisierung wird dabei mit Hilfe von OCL Bedingungen definiert. Bekannte Realisierungen im Umfeld von Java-Applikationen sind das Apache Shiro Framework [Shi16] und der IBM Tivoli Security Policy Manager [Tiv16]. Sie erlauben beide eine externe Spezifikation von Sicherheitsrichtlinien.

Im Rahmen dieser Arbeit wurde sich für das RBAC-Modell entschieden, da es eine grundlegende Absicherung von Applikationen bietet und die Basis für tiefergehende Konzepte darstellt. Im Rahmen der JPA ist das RBAC-Modell bereits vorgesehen und wird von den gängigen JPA Implementierungen und den Applikationsservern unterstützt. Darüber hinaus werden Erweiterungspunkte für tiefergehende Konzepte vorgestellt und angeboten, so dass RBAC die Ausgangsbasis darstellt.

5.3.1 Die Rollensprache

Zur Definition der dem System bekannten Rollen wird die Rollensprache verwendet. Diese ermöglicht es, einzelne Rollen, die dem System bekannt sein sollen (vgl. FA5-AG), zu definieren. Es sei an dieser Stelle erneut angemerkt, dass im Rahmen dieser Arbeit die Rollen als statische Elemente des Systems betrachtet werden. Neuen Nutzern können natürlich Rollen zur Laufzeit des Systems zugeordnet werden.

```
1 rolediagram SocNetRoles {                                    RD
2   // weitere Elemente des Rollendiagramms                     ...
3 }
```

Listing 5.12: Auszug des Rollendiagramms `SocNetRoles` für das Klassendiagramm `SocNet`. Die Elemente des Rollendiagramms sind hier ausgelassen.

Wie bereits zuvor können auch Rollendiagramme in Paketen organisiert sein. Listing 5.12 zeigt einen Ausschnitt des Rollendiagramms für das zu entwerfende System. Es beginnt mit dem Schlüsselwort `rolediagram`, gefolgt von einem eindeutigen Namen. Das Rollendiagramm kann unabhängig von anderen Modellen modelliert werden und besitzt daher auch keine Referenzen auf weitere Systemartefakte. Innerhalb des Rollendiagramms werden die Rollen des Systems modelliert.

Listing 5.13 zeigt die Rollendefinitionen für das soziale Netzwerk innerhalb des Rollendiagramms. Jede Rollendefinition wird durch das Schlüsselwort `role`, ebenfalls gefolgt von einem innerhalb des Diagramms eindeutigen Rollennamen, eingeleitet. Es ist zudem möglich, eine Vererbungsbeziehung zwischen Rollen herzustellen. Dies wird durch das Schlüsselwort `extends` und einer Referenz auf einen existierenden Rollennamen ausgedrückt. Dabei ist eine Mehrfachvererbungsbeziehung zwischen Rollen möglich. In Listing 5.13 werden die Rollen `User` und `Moderator` definiert (vgl. FA5-AG). Die Rolle des

Moderators erweitert die Rolle des Users. Dies bedeutet, dass alle Rechte, die der Rolle `User` zugeordnet werden, auch der Rolle `Moderator` zur Verfügung stehen.

```
1  role User;                                                    RD
2                                                                ...
3  role Moderator extends User;
```

Listing 5.13: Definition der Rollen `User` und `Moderator`. Die Rolle `Moderator` erbt von der Rolle `User`

Die Grammatik der Rollensprache

Listing 5.14 zeigt eine für die bessere Lesbarkeit aufbereitete Version der Grammatik der Rollensprache. Bei der Rollensprache handelt es sich um eine sehr kleine Grammatik, die im Wesentlichen aus zwei Produktionen besteht. Die Grammatik besteht aus der Startproduktion `Roles`, die das generelle Aussehen eines Rollendiagramms, wie das Schlüsselwort `rolediagram` und dem Namen definiert, und beinhaltet ferner die Produktion `RoleDefinition` zur Definition einer einzelnen Rolle. Hier lässt sich die Möglichkeit zur Modellierung von Einfach- und Mehrfachvererbung erkennen. Die vollständige Grammatik der Sprache ist im Anhang in Abschnitt C.7 dargestellt.

```
1  grammar RD {                                                  MCG
2                                                                RD
3    Roles = "rolediagram" Name "{" (RoleDefinition)* "}" ;      ...
4
5    RoleDefinition = "role" roleName:Name ( ["extends"]
6      parentRoleNames:Name ("," parentRoleNames:Name)* )? ";"
7    ;
8  }
```

Listing 5.14: Aufbereiteter Auszug der Grammatik der Rollensprache. Dargestellt ist die Startproduktion `Roles` und die Produktion `RoleDefinition`. Weitere Produktionen der Grammatik sind ausgelassen. Die vollständige Version der Grammatik ist in Anhang C.7 dargestellt.

5.3.2 Kontextbedingungen der Rollensprache

Neben der Grammatik der Sprache zur Syntaxdefinition sind im Rahmen dieser Arbeit auch Kontextbedingungen, die die möglichen Rollendiagramme weiter einschränken, definiert worden. Die Kontextbedingungen wurden in [Cöm13] gemeinsam identifiziert und im Rahmen dieser Arbeit abgeändert und angepasst. Dabei wird hier zunächst

auf die Kontextbedingungen, die innerhalb eines Rollendiagramms gelten, eingegangen. Kontextbedingungen zwischen mehreren Modellen und vor allem Kontextbedingungen zwischen Modellen der Rollen-, der Rechte- und der Mappingdiagramme werden in Abschnitt 5.3.6 definiert.

Zunächst muss jedes Rollendiagramm einen eindeutigen Namen innerhalb des Paketes und jede Rolle einen eindeutigen Namen innerhalb des Diagramms besitzen. Diesen Sachverhalt beschreiben die Kontextbedingungen RD-1 und RD-2.

RD-1

Bedingung: Der vollqualifizierte Name eines Rollendiagramms, bestehend aus Paketname und Diagrammname, muss eindeutig sein.

Schweregrad: Fehler

RD-2

Bedingung: Die Rollennamen müssen innerhalb eines Diagramms eindeutig sein.

Schweregrad: Fehler

Darüber hinaus müssen Kontextbedingungen sicherstellen, dass die Vererbung von Rollen nicht zu inkonsistenten Modellen führt.

RD-3

Bedingung: Referenzierte Rollen müssen existieren.

Schweregrad: Fehler

RD-4

Bedingung: Die Vererbungshierarchie einer Rolle darf keine Zyklen enthalten.

Schweregrad: Fehler

RD-5

Bedingung: Eine modellierte Vererbungsbeziehung innerhalb einer Vererbungshierarchie einer Rolle sollte nicht als Duplikat modelliert werden.

Schweregrad: Warnung

Nachdem die Sprache zur Modellierung von Rollen mit ihrer konkreten Syntax, ihrer Grammatik und ihren Kontextbedingungen in diesem Abschnitt vorgestellt wurde, wird im nächsten Abschnitt die Sprache zur Modellierung von Rechten vorgestellt. Daran anschließend werden die Sprache zur Zuordnung von Rechten und Rollen und zudem die Kontextbedingungen zwischen allen Sprachen vorgestellt.

5.3.3 Die Rechtesprache

Nachdem zuvor die Sprache zur Definition der im System existierenden Rollen vorgestellt wurde, wird die Sprache zur Definition der möglichen Rechte vorgestellt. Dabei werden in der Rechtesprache alle möglichen Rechte einer Applikation modelliert und nicht nur die gewünschten. Im weiteren Verlauf dieses Abschnitts wird eine automatisierte Ableitung dieser Rechte basierend auf einem Klassendiagramm vorgestellt. Zudem wird eine Möglichkeit zur Erweiterung um benutzerdefinierte Rechte gezeigt.

```
1 permissiondiagram SocNetRights for montiee.SocNet {                PD
2   // weitere Elemente des Rechtediagramms                           ...
3 }
```

Listing 5.15: Auszug des Rechtediagramms `SocNetRights` für das Klassendiagramm `SocNet`. Die Elemente des Rechtediagramms sind hier ausgelassen.

Ein Rechtediagramm kann mit der optionalen Angabe eines Pakets beginnen. Listing 5.15 zeigt einen Ausschnitt eines Rechtediagramms. Es beginnt mit dem Schlüsselwort `permissiondiagram` und einem eindeutigen Namen. Nach dem eindeutigen Namen wird eine Referenz zu dem Bezugsklassendiagramm aufgespannt. Nach dem Schlüsselwort `for` muss der vollqualifizierte Name des Klassendiagramms, um die Konsistenz des Diagramms zu sichern, angegeben werden. Hier können auch, durch Komma getrennt, mehrere Klassendiagramme angegeben werden. Bei einer automatisierten Ableitung des Diagramms ist eine solche Konsistenzsicherung, da sich ein Rechtediagramm nur auf ein Klassendiagramm bezieht, implizit gegeben. Da aber die Möglichkeit zur manuellen Erweiterung existiert, werden Informationen über die Klassendiagramme benötigt. Innerhalb des Rechtediagramm Blocks können einzelne Rechte definiert werden.

```
1   context p : Post {                                               PD
2     read, create, delete, update p;                                ...
3   }
```

Listing 5.16: Definition der Rechte für Elemente des Typs `Post`. Dargestellt sind die Rechte `read`, `create`, `delete` und `update`.

Listing 5.16 zeigt einen Auszug der verfügbaren Rechte für die Klasse `Post`. Die Rechtedefinition ist dabei an die konkrete Syntax der OCL/P [Rum11, Rum12, Sch12] angelehnt. Es beginnt dabei mit dem Aufspannen des für das zu definierende Recht notwendigen Kontextes, welcher in Listing 5.16 durch `Post` definiert wird. Dabei beginnt der Kontext mit dem Schlüsselwort `context`, gefolgt von einem Variablennamen, einem Doppelpunkt und der Angabe des Typs. Geschweifte Klammern beginnen einen Block zur Definition von Rechten innerhalb des Kontexts. Dort können, wie in Listing 5.16 präsentiert, die Rechte `read`, `create`, `delete`, `update`, die als Schlüsselwörter

Teil der Sprache sind, definiert werden. Dabei können die Rechte einzeln, in einer Zeile oder mit Komma getrennt hintereinander modelliert werden. Jedes Recht beginnt dabei mit der Angabe eines Schlüsselwortes, welches entweder `read`, `create`, `delete` oder `update` ist, gefolgt von dem Variablennamen des Kontexts. Diese Rechte modellieren, dass eine Instanz eines Typs gelesen, angelegt, gelöscht oder verändert werden darf. Diese Rechte beziehen sich letztendlich auf Datenbankoperationen und nicht auf die bloße Objekterstellung im System. Das `read` Recht bedeutet dabei, dass es erlaubt ist, alle Elemente eines Typs zu lesen. Dies entspricht einem Query an die Datenbank, welches alle Elemente des Typs zurückgibt.

PD

```
1  context p : Profile {
2    read, create, delete, update p;
3    query getProfileByUserName;
4
5    association p.sent;
6  }
```

Listing 5.17: Definition der Rechte für Elemente des Typs `Profile`. Dargestellt sind die CRUD-Rechte sowie die Rechte zum Ausführen von Queries und zur Navigation von Assoziationen.

Listing 5.17 zeigt eine weitere Definition verschiedener Rechte im Kontext der Klasse `Profile`. Neben den CRUD Operationen sind das Recht zur Benutzung des Queries `getProfileByUserName` (vgl. FA4.1-AG) und ein Recht zur Navigation einer Assoziationen modelliert. Queries werden ebenfalls als Rechte modelliert. Rechte zur Benutzung von Queries beginnen dabei mit dem Schlüsselwort `query`, gefolgt vom Namen des Queries. Diese Queries sind diejenigen, die in einer Tagdefinition definiert wurden, oder vorgefertigte Standardqueries. Aus Sicht des Rechts bedeutet dies, dass das Query mit dem Namen auf der Datenbank ausgeführt und das Ergebnis des Queries gelesen werden kann. Die vorgefertigten Queries enthalten immer ein Query für jedes Attribut einer Klasse, welches die Elemente des entsprechenden Typs basierend auf einem Attributwert zurückgibt. Bei dem Query `getProfileByUserName` handelt es sich also um ein vorgefertigtes Query, da die Klasse `Profile` bereits das Attribut `userName` enthält. Die inhaltliche Definition eines zusätzlichen Queries findet in der Tagdefinition statt, lediglich der Name des Queries wird, wie in Listing 5.18 gezeigt, als Referenz verwendet. Dies wird in Abschnitt 5.3.4 genauer erläutert.

Rechte zur Navigation von Assoziationen beginnen mit dem Schlüsselwort `association` gefolgt von dem Variablennamen des Kontexts, einem Punkt und dem Assoziationsnamen. Assoziationen benötigen keine CRUD Rechte, da dies über die CRUD Rechte für den assoziierten Typ abgebildet wird. Kann ein Objekt geschrieben werden, so können immer auch seine assoziierten Objekte kaskadiert geschrieben werden, wenn die Kaskadierung konfiguriert wurde. Werden die assoziierten Objekte nicht kaskadiert, so wird auch das Recht benötigt, diese Objekte zu schreiben. Lediglich das Navigati-

onsrecht wird explizit benötigt, damit unterschieden werden kann, ob eine Assoziation in einem bestimmten Kontext gelesen werden darf. Dies ist semantisch unterschiedlich zu dem Recht, den assoziierten Typ zu lesen. Am Beispiel des Klassendiagramms, wie in Listing 4.2 dargestellt, des sozialen Netzwerks, bedeutet dies, dass, wenn ein Profil gelesen werden darf, auch alle gesendeten Posts des Profils gelesen werden dürfen. Das in Listing 5.16 modellierte Recht, Posts zu lesen, hingegen bedeutet, dass es erlaubt ist, alle Nachrichten des Systems zu lesen. Somit kann das Recht existieren, Objekte eines Typs in einem speziellen Kontext zu lesen, auch wenn nicht alle Objekte des Typs gelesen werden können.

PD

```
1  context p : Person {
2    query getPersonByRealName;
3    query getFriendByRealName;
4  }
```

```
...
```

Listing 5.18: Definition der Rechte für Elemente des Typs `Person`. Dargestellt sind Rechte zur Ausführung benutzerdefinierter Queries.

Listing 5.18 zeigt zwei Rechte zur Ausführung von Queries. Das Query `getPersonByRealName` (vgl. FA4.2-AG) ist dabei wieder ein vorgefertigtes Query, wohingegen das Query `getFriendByRealName` (vgl. FA4.3-AG) ein benutzerdefiniertes Query darstellt, welches in der Tagdefinition in Abschnitt 10.3 definiert wird. Durch die unterschiedlichen Rechte wird es möglich, unterschiedlichen Rollen, wie z.B. einem Moderator das Recht, alle Nachrichten zu lesen, zuzuordnen, wohingegen ein Nutzer nur das Recht, seine eigenen Nachrichten seines Profils zu lesen, erhält.

MCG

PD

```
1 grammar PD {
2
3   Permissions= "permissiondiagram" Name "for" QualifiedName
4     "{"
5       (PermissionDefinition)*
6     "}";
7   // weitere Produktionen
8 }
```

```
...
```

Listing 5.19: Aufbereiteter Auszug der Grammatik der Rechtesprache. Dargestellt ist die Startproduktion `Permissions`. Weitere Produktionen der Grammatik sind ausgelassen. Die vollständige Version der Grammatik ist in Anhang C.8 dargestellt.

Die Grammatik der Rechtesprache

Nachdem zuvor die konkrete Syntax der Rechtesprache vorgestellt wurde, wird in diesem Abschnitt die Grammatik dieser Sprache kurz erläutert. Die hier vorgestellten Auszüge der Grammatik sind verkürzt und für die bessere Lesbarkeit angepasst worden.

Listing 5.19 zeigt dazu zunächst den Beginn der Grammatik und die Startproduktion. Es ist erkennbar, dass die Referenz auf das zugehörige Klassendiagramm durch den vollqualifizierten Namen nach dem Schlüsselwort `for` angegeben wird. Innerhalb des mit geschweiften Klammern beginnenden und endenden Blocks können dann mehrere `PermissionDefinition` angegeben werden. Listing 5.20 zeigt diese `PermissionDefinition`. Jede dieser Definition definiert einen Kontext, der eine Variable eines Typs enthält. Dieser Typ, in der Grammatik als vollqualifizierter Name dargestellt, stellt eine Referenz auf ein Element des Klassendiagramms dar. Innerhalb einer Definition gibt es drei verschiedene Arten von Rechten: `ModifyPermission`, `QueryPermission` und `AssociationPermission`.

```
1  PermissionDefinition = "context" Name ":" QualifiedName
2    "{"
3      (ModifyPermission | QueryPermission
4       | AssociationPermission)*
5    "}";
```

MCG
PD
...

Listing 5.20: Darstellung der Produktion `PermissionDefinition`, die den Kontext für die eigentlichen Rechte aufspannt. Die vollständige Version der Grammatik ist in Anhang C.8 dargestellt.

Listing 5.21 zeigt die drei unterschiedlichen Permissions. Eine `ModifyPermission` hat vier verschiedene Typen: `create`, `read`, `update` und `delete`. Der `instanceName` stellt den innerhalb des Kontexts deklarierten Variablennamen dar.

Die `QueryPermission` und `AssociationPermission` beinhalten wiederum Namensreferenzen auf Queries und Assoziationen innerhalb des referenzierten Klassendiagramms. Die vollständige Grammatik der Sprache ist im Anhang in C.8 dargestellt.

5.3.4 Automatisierte Ableitung eines Rechtediagramms

Wie bereits zuvor angemerkt, dient diese Sprache zur Modellierung möglicher Rechte des Systems. Diese Rechte beziehen sich dabei auf die typischen CRUD-Operationen für Entitäten sowie die Ausführung von Queries und die Navigation von Assoziationen innerhalb eines Kontexts. Aus diesem Grund gibt es für ein Klassendiagramm immer die gleichen Rechte, welche dann um benutzerdefinierte Rechte erweitert werden können.

Daher wurde im Rahmen dieser Arbeit, wie auch in [Cöm13] betreut, ein Algorithmus zur initialen Erstellung der Rechtedefinition entwickelt. Der Algorithmus erhält ein

```
1  ModifyPermission =                                                    MCG
2    (modifyTypes:ModifyType) ("," modifyTypes:ModifyType)*              PD
3      instanceName:Name ";";                                            ...
4
5  ModifyType =
6    ["create"] | ["read"] | ["update"] | ["delete"];
7
8  QueryPermission =
9    "query" queryName:Name ";";
10
11 AssociationPermission =
12   "association" Name "." QualifiedName ";";
```

Listing 5.21: Darstellung der Produktionen der drei unterschiedlichen Rechte: `ModifyPermission`, `QueryPermission` und `AssociationPermission`. Die vollständige Version der Grammatik ist in Anhang C.8 dargestellt.

Klassendiagramm und eine Tagdefinition als Eingabe und generiert daraus ein Rechtediagramm mit allen möglichen Rechten für das Klassendiagramm. Dabei werden die einzelnen Klassen, ihre Assoziationen und die Tagdefinitionen zu Fetch- und Cascade Optionen berücksichtigt. Dabei sei daran erinnert, dass beim Nachladen von Objekten aus Datenbank die `Fetch` Ausprägungen `lazy` und `eager` und die `Cascade` Ausprägungen `all`, `none`, `merge`, `persist`, `refresh` und `remove` verwendet werden können, wie in Abschnitt 3.3 beschrieben.

Der Algorithmus arbeitet dabei wie folgt:

1. Für jede Klasse K des Klassendiagramms wird ein Kontext der Form `context k : K { ...}` innerhalb des Rechtediagramms erzeugt.
2. Für jede Klasse wird das Recht, Instanzen zu lesen, erstellt. Dazu wird `read k;` innerhalb des Kontexts `k` generiert.
3. Für jede Klasse wird das Recht, Instanzen zu aktualisieren, genau dann erstellt, wenn es eine eingehende Assoziation, die die Kaskadierung `none`, `remove` oder `refresh` besitzt, gibt. Dazu wird `update k;` innerhalb des Kontexts `k` generiert.
4. Für jede Klasse wird das Recht, Instanzen zu löschen, genau dann, wenn alle eingehenden Assoziationen, die Kaskadierung `delete` oder `all` besitzen, erstellt. Dazu wird `delete k;` innerhalb des Kontexts `k` generiert.
5. Für jede Klasse wird das Recht, Instanzen zu erstellen, genau dann, wenn es eine eingehende Assoziation gibt, die nicht die Kaskadierung `all`, `merge` oder `persist` besitzt, erstellt. Dazu wird `create k;` innerhalb des Kontexts `k` generiert.
6. Für jede Klasse wird für jedes Attribut ein Recht zur Ausführung von Queries der Form `query get(`k_{name}`)By(`$attribute_{name}$`);` innerhalb des Kontexts `k` generiert.

7. Für jede Klasse, die mit einem Query q getaggt ist, wird ein Recht zur Ausführung von Queries erstellt. Dazu wird `query` q_{name}`;` innerhalb des Kontexts `k` generiert.

8. Für jede Klasse wird ein Navigationsrecht für eine Assoziation genau dann, wenn die Assoziation einen Fetch Tag mit der Option lazy besitzt, erstellt. Dazu wird `association k.association(`$_{name}$`)` innerhalb des Kontexts `k` generiert.

Generell wird eine Aggregation wie eine Assoziation behandelt. Bei einer Komposition werden weder `create`, `update` oder `delete` Rechte erstellt. Die Funktionsweise des Algorithmus begründet sich in der Bedeutung der Tags. Das Verhalten für die CRUD-Operationen ist dabei dadurch bedingt, dass es immer das Recht geben muss, die Entitäten des Systems zu lesen. Die Entscheidung, ob eine Entität aktualisiert werden kann, liegt in den umgebenden Kontexten. Da Kaskadierung bedeutet, dass die assoziierte Entität ebenfalls aktualisiert wird, wenn die auslösende Entität aktualisiert wird, sollte keine Möglichkeit zur eigenständigen Aktualisierung, da es sonst zu Inkonsistenzen in den Daten führt, existieren. Daher muss es bei einer Aktualisierung das Recht dazu nur geben, wenn es eine Assoziation gibt, die nicht entsprechend kaskadiert. Aus dem gleichen Grund wird die Entscheidung für Löschen oder Erstellen getroffen. Die Assoziationsnavigation berücksichtigt den Fetch Tag. Hat dieser die `eager` Option, bedeutet das, dass immer der vollständige Objektgraph aus der Datenbank geladen wird. Dies bedeutet auch, dass beim Lesen der Entität auch die assoziierte Entität geladen ist. Aus diesem Grund kann durch Lesen der Entität immer auch die Assoziation navigiert werden. Anders ist dies bei der Option `lazy`. Dort müssen die Entitäten aktiv nachgeladen werden, so dass ein Navigationsrecht erforderlich ist.

Natürlich bietet dieses Verfahren nur einen Anfang der möglichen Rechte. Eigene, benutzerdefinierte Rechte müssen in einem eigenständigen Rechtediagramm definiert werden. Hier können auch Rechte, die gewünscht, aber vom Algorithmus nicht erstellt werden, hinzugefügt werden, da dieser nur ein Standardverfahren anbietet, welches sich je nach System unterscheiden kann.

Nachdem in diesem Abschnitt die Grammatik, die kontextfreie Syntax der Rechtesprache und ein Algorithmus zur Generierung der möglichen Rechte eines Klassendiagramms vorgestellt wurden, werden im folgenden Abschnitt Kontextbedingungen der Rechtesprache vorgestellt, die die Konsistenz des Diagramms sichern.

5.3.5 Kontextbedingungen der Rechtesprache

Die Kontextbedingungen wurden ebenfalls [Cöm13] gemeinsam identifiziert und im Rahmen dieser Arbeit abgeändert und angepasst. Neben der syntaktischen, kontextfreien Definition der Rechtesprache existieren Kontextbedingungen, die die Konsistenz innerhalb des Diagramms, aber auch zwischen dem Diagramm und dem referenzierten Klassendiagramm sichern. Dazu muss zunächst die Existenz der referenzierten Elemente sichergestellt werden.

PD-1

Bedingung: In einem Kontext referenzierte Typen müssen in dem referenzierten Klassendiagramm modelliert sein.

Schweregrad: Fehler

PD-2

Bedingung: Der Assoziationsname, der in einem Recht zur Modellierung einer Assoziationsnavigation verwendet wird, muss in dem referenzierten Klassendiagramm modelliert sein.

Schweregrad: Fehler

PD-3

Bedingung: In einer Tagdefinition müssen die Queries, die als Namensreferenz innerhalb eines Rechts zur Ausführung von Queries verwendet werden, existieren.

Schweregrad: Fehler

Die Kontextbedingungen erfordern auch, dass die referenzierten Klassendiagramme frei von Namenskonflikten sind, und alle Elemente eindeutig durch ihren vollqualifizierten Namen identifiziert werden können. Diese Kontextbedingung ist allerdings eine Bedingung an die Aggregation mehrerer Klassendiagramme und nicht spezifisch für die Rechtesprache, so dass sie hier nicht zusätzlich aufgeführt wird. Darüber hinaus sollen keine Rechte doppelt definiert werden.

PD-4

Bedingung: Innerhalb eines Kontextes soll kein Recht doppelt definiert sein. Dies betrifft Assoziationen, Modifikationen oder Queries.

Schweregrad: Warnung

Nachdem bisher die beiden Sprachen zur Modellierung von Rechten und zur Modellierung von Rollen vorgestellt wurden, wird im nächsten Abschnitt die Sprache zur Zuordnung von Rechten und Rollen vorgestellt.

5.3.6 Die Mappingsprache

Nachdem beide Sprachen zur Definition von Rollen und zur Definition von Rechten vorgestellt wurden, können Rollen spezifische Rechte mittels der Mappingsprache zugeordnet werden. Dabei wird im Kontext einer Rolle eine Teilmenge der Rechte hinzugefügt.

Listing 5.22 zeigt einen Ausschnitt aus der konkreten Syntax eines Mappingdiagramms (vgl. FA9-PE). Es kann mit der Angabe eines Paketnamens beginnen. Zudem können

```
1 import montiee.roles.SocNetRoles.*;                                PRM
2 import montiee.rights.SocNetRights.*;                              ...
3
4 rolemappingdiagram SocNetMapping {
5   // weitere Elemente des Mappingdiagramms
6 }
```

Listing 5.22: Auszug des Mappingdiagramms `SocNetMapping` für das Klassendiagramm `SocNet`. Die Elemente des Mappingdiagramms sind hier ausgelassen.

Importe spezifiziert werden. Diese Importe werden benötigt, um die zur Verfügung stehenden Rechte und Rollen in das Diagramm zu importieren. Die dargestellten Importe importieren alle Rechte und alle Rollen. Das Diagramm beginnt mit dem Schlüsselwort `rolemappingdiagram` gefolgt von einem Namen. Innerhalb des Containers werden die Elemente des Diagramms modelliert.

```
1   User {                                                           PRM
2     context p : Profile {                                          ...
3       association p.sent;
4     }
5     context p : Person {
6       query getFriendByRealName;
7     }
8     context p : Post {
9       create, update p;
10    }
11  }
```

Listing 5.23: Zuordnung modellierter Rechte zur modellierten Rolle `User`

Listing 5.23 zeigt die Zuordnung von Rechten zu der Rolle `User`. Dabei beginnt die Zuordnung mit dem Rollennamen, dem Rechte zugeordnet werden sollen. Geschweifte Klammern umfassen den Block von Rechten, die den Rollen zugeordnet werden sollen. Die zugeordneten Rechte sind eine Teilmenge der zuvor in der Rechtesprache modellierten Rechte. In Listing 5.23 sind die Rechte für das Erstellen und Editieren von Nachrichten angegeben. Zudem kann ein User im Kontext seines Profils auf seine eigenen Posts über das Recht zur Navigation der Assoziation `p.sent` zugreifen (vgl. FA2.2-AG). Zudem darf er das Query `getFriendByRealName` ausführen (vgl. FA4.3-AG). Dem User wird an dieser Stelle nicht das `read` Recht zugeordnet, da er nur Posts in seinem eigenen Kontext, nicht aber alle im System vorhandenen Posts lesen soll.

Darüberhinausgehend zeigt Listing 5.24 die modellierten Rechte eines Moderators. In Listing 5.13 wurde modelliert, dass die Rolle des Moderators die Rolle des Users

```
1   Moderator {                                                    PRM
2     context p: Profile {                                         ...
3       read, create, delete, update p;
4       query getProfileByProfileName;
5     }
6
7     context p : Post {
8       read, delete p;
9     }
10  }
```

Listing 5.24: Zuordnung modellierter Rechte zur modellierten Rolle `Moderator`. Durch die in Listing 5.13 modellierte Vererbung beinhaltet die Rolle `Moderator` auch die Rechte der Rolle `User` aus Listing 5.23.

erweitert. Somit hat der Moderator die bereits in Listing 5.23 gezeigten Rechte des Users und die hinzukommenden Rechte für `Profile` und `Posts`. Im Kontext von Profilen darf der Moderator das Query ausführen und Profile lesen, erstellen, löschen und editieren. Zudem darf er alle Posts im System lesen und löschen (vgl. FA4.1-AG, FA5.2-AG und FA5.3-AG).

Zusammenfassend lassen sich somit Rechte und Rollen für Enterprise Applikationen modellieren. Diese Rechte lassen sich Rollen zuordnen und können somit die angebotene Funktionalität eines Systems auf unterschiedliche Rollen zuschneiden.

Die Grammatik der Mappingsprache

Nachdem zuvor die konkrete Syntax der Mappingsprache vorgestellt wurde, wird in diesem Abschnitt die Grammatik dieser Sprache kurz erläutert. Die hier vorgestellten Auszüge der Grammatik sind verkürzt und für die bessere Lesbarkeit angepasst worden.

```
1  grammar PRM {                                                   MCG
2    RoleMapping =                                                 PRM
3      "rolemappingdiagram" Name "{"                               ...
4        Mapping*
5      "}";
6    // weitere Produktionen
7  }
```

Listing 5.25: Aufbereiteter Auszug der Grammatik der Mappingsprache. Dargestellt ist die Startproduktion `RoleMapping`. Weitere Produktionen der Grammatik sind ausgelassen. Die vollständige Version der Grammatik ist in Anhang C.9 dargestellt.

Listing 5.25 zeigt dazu den Beginn der Grammatik und die Startproduktion `RoleMapping`. Die Startproduktion definiert den generellen Aufbau des Diagramms. Innerhalb des Diagramms sind mehrere Mappings angegeben. Die Möglichkeit zur Angabe von Importen ist eine implizite Funktionalität von MontiCore und hier nicht explizit dargestellt. Sie wird aber dennoch zum Import des Rechte- und des Rollendiagramms verwendet.

```
1  Mapping =                                                          MCG
2    QualifiedName "{" MappingPermissionDefinition* "}";              PRM
3                                                                     ...
```

Listing 5.26: Darstellung der `Mapping` Produktion, die die Zuordnung zwischen Rollen und Rechten ermöglicht. Die vollständige Version der Grammatik ist in Anhang C.9 dargestellt.

Listing 5.26 zeigt die Definition der `Mapping` Produktion. Sie beginnt mit einer Referenz auf einen importierten Rollennamen als vollqualifizierter Name, gefolgt von mehreren Definitionen von Rechten. Zur Angabe dieser Definitionen wird die Möglichkeit zur Spracheinbettung verwendet. Dazu wird eine externe Produktion definiert, wie in Listing 5.27 dargestellt. Diese Produktion wird an die in Listing 5.20 dargestellte Produktion gebunden, so dass sie hier verwendet werden kann. Dadurch existiert eine Abhängigkeit auf Sprachebene.

```
1  external MappingPermissionDefinition;                              MCG
2                                                                     PRM
3                                                                     ...
```

Listing 5.27: Darstellung der externen Produktion `MappingPermissionDefinition`, die die Einbettung der Rechtesprache mit Hilfe der Mechanismen zur Spracheinbettung von MontiCore realisiert. Die vollständige Version der Grammatik ist in Anhang C.9 dargestellt.

Die vollständige Grammatik der Sprache ist im Anhang in C.9 dargestellt. Nachdem in diesem Abschnitt die konkrete Syntax sowie die Grammatik der Sprache vorgestellt wurde, werden im nächsten Abschnitt Kontextbedingungen für die Mappingsprache, aber auch für die Beziehungen zwischen den Sprachen vorgestellt.

5.3.7 Kontextbedingungen der Mappingsprache

Nachdem zuvor bereits die Kontextbedingungen der Rollen- und der Rechtesprache vorgestellt wurden, sollen die Kontextbedingungen der Mappingsprache vorgestellt werden, die ebenfalls in [Cöm13] gemeinsam identifiziert und im Rahmen dieser Arbeit abgeändert und angepasst wurden. Da in der Mappingsprache durch Sprachaggregation beide

Sprachen adressiert werden, werden an dieser Stelle auch Kontextbedingungen zwischen den jeweiligen Sprachen definiert.

Zunächst werden Kontextbedingungen formuliert, die die Existenz der unterschiedlichen Importe sicherstellen.

PRM-1
Bedingung: Die importierten Rollendiagramme müssen existieren.
Schweregrad: Warnung

PRM-2
Bedingung: Die importierten Rechtediagramme müssen existieren.
Schweregrad: Warnung

Darüber hinaus müssen die importierten Typen überschneidungsfrei und konfliktfrei sein. Ansonsten kann es zu einem inkonsistenten Modell kommen. Als Inkonsistenz können gleiche Namen, aber auch Widersprüche auftreten.

PRM-3
Bedingung: Die importierten Rollen müssen überschneidungsfrei sein.
Schweregrad: Warnung

PRM-4
Bedingung: Die importierten Rechte müssen überschneidungsfrei sein.
Schweregrad: Warnung

Werden die Rollen, aber auch die Rechte im Modell verwendet, müssen diese Teil des Imports sein oder vollqualifiziert angegeben werden.

PRM-5
Bedingung: Verwendete Rollen müssen aus importieren Rollendiagrammen aufgelöst oder über den vollqualifizierten Namen identifiziert werden können.
Schweregrad: Warnung

PRM-6
Bedingung: Verwendete Rechte müssen aus importieren Rechtediagrammen aufgelöst oder über den vollqualifizierten Namen identifiziert werden können.
Schweregrad: Warnung

Gleichzeitig muss die Zuordnung der Rechte zu den Rollen konfliktfrei sein. Dies kann insbesondere durch die Rollenvererbung verschattet werden. Aus diesem Grund werden keine negativen Rechte zugelassen. Somit kann immer die Obermenge aller Rechte aller Rollen der Vererbungshierarchie gebildet werden.

5.4 Zusammenfassung

In diesem Kapitel wurde ein weiterer Teil der Sprachfamilie MontiEE vorgestellt. Dabei lag der Fokus der vorgestellten Sprachen auf den Sprachen zur Modellierung der Kommunikation von Enterprise Applikationen. Die Sprachen zur Modellierung der Persistenz von Enterprise Applikationen wurden in Kapitel 4 vorgestellt.

Die vorgestellten Sprachen zur Modellierung der Kommunikation von Enterprise Applikationen unterstützen den Produktentwickler bei der Modellierung unterschiedlicher Clients. Dazu wurde in Abschnitt 5.2 eine Sichtensprache mit ihren Kontextbedingungen vorgestellt, die auf Basis einer Sicht auf das zu Grunde liegende Domänenmodell das Domänenmodell für spezifische Clients modellieren kann. Die Sichtensprache wird dazu verwendet, auf Technologie- oder Implementierungsspezifika der Clients einzugehen. Darüber hinaus wurde eine Transformation einer Sicht in ein syntaktisch korrektes Klassendiagramm vorgestellt. Diese Transformation ermöglicht die Wiederverwendung bestehender Generatoren, die aus Klassendiagrammen Quellcode generieren.

Zur Modellierung unterschiedlicher Rechte und Rollen im System wurden in Abschnitt 5.3 drei Sprachen und ihre Kontextbedingungen vorgestellt, die es dem Produktentwickler ermöglichen, unterschiedlichen Rollen verschiedene Rechte zuzuweisen. Darüber hinaus wurde ein Algorithmus zur Ableitung spezifischer auf dem Klassendiagramm und der Tagdefinition basierender Rechte gezeigt.

Im nachfolgenden Kapitel 6 wird eine Sprache zur Modellierung der Evolution einer Enterprise Applikation vorgestellt.

Kapitel 6

Modellierung der Evolution

Nachdem in den vorangegangenen Kapiteln 4 und 5 die Sprachen zur Modellierung der Persistenz wie auch zur Modellierung der Kommunikation vorgestellt wurden, wird in diesem Kapitel eine Sprache zur Unterstützung von Evolution vorgestellt. Die zuvor vorgestellten Sprachen fokussieren auf die Modellierung der Implementierung des Systems, wohingegen die Deltasprache die Weiterentwicklung und die Evolution des Systems in den Blick nimmt. Als Evolution wird dabei die Weiterentwicklung der Modelle, auf deren Basis das System generiert wird, verstanden. Dabei wird davon ausgegangen, dass die Enterprise Applikation innerhalb eines agilen Prozesses entwickelt wird und häufige Releases stattfinden. Da bereits frühzeitig Nutzer das System verwenden können, muss die Weiterentwicklung und Wartung des Systems ohne Ausfallzeiten erfolgen. Bei der Weiterentwicklung steht das zugrunde liegende Domänenmodell im Vordergrund, da es die Struktur und die persistenten Daten des Systems definiert, die die Enterprise Applikation dem Nutzer zur Verfügung stellt und die diese bearbeiten können. Aus diesem Grund muss neben der Evolution des Domänenmodells auch die Migration der persistenten Daten berücksichtigt werden, die, sofern die Weiterentwicklung kein Löschen von Daten intendiert, verlustfrei erfolgen muss. Dies ist bei Enterprise Applikationen wichtig, da die Weiterentwicklung auch immer eine Datenmigration erfordert. Die Daten des Systems müssen zur Evolution konform migriert werden. Zur Automatisierung dieser Migration wird auch die Systemevolution mit Hilfe von Modellen beschrieben.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Vorstellung einer automatisiert abgeleiteten klassendiagrammspezifischen Deltasprache zur Modellierung der Evolution einer Enterprise Applikation.
- Vorstellung einer handgeschriebenen Erweiterung der automatisiert abgeleiteten Deltasprache.
- Vorstellung von Basisoperationen zur Modellierung von Deltas.
- Vorstellung von weiterführenden Operationen zur Modellierung von Refactorings in Deltas.

In den nachfolgenden Kapiteln 7, 8 und 9 stehen die Generatoren zur Umsetzung einer Enterprise Applikation im Fokus. Zunächst wird mit einem Überblick des Kapitels begonnen.

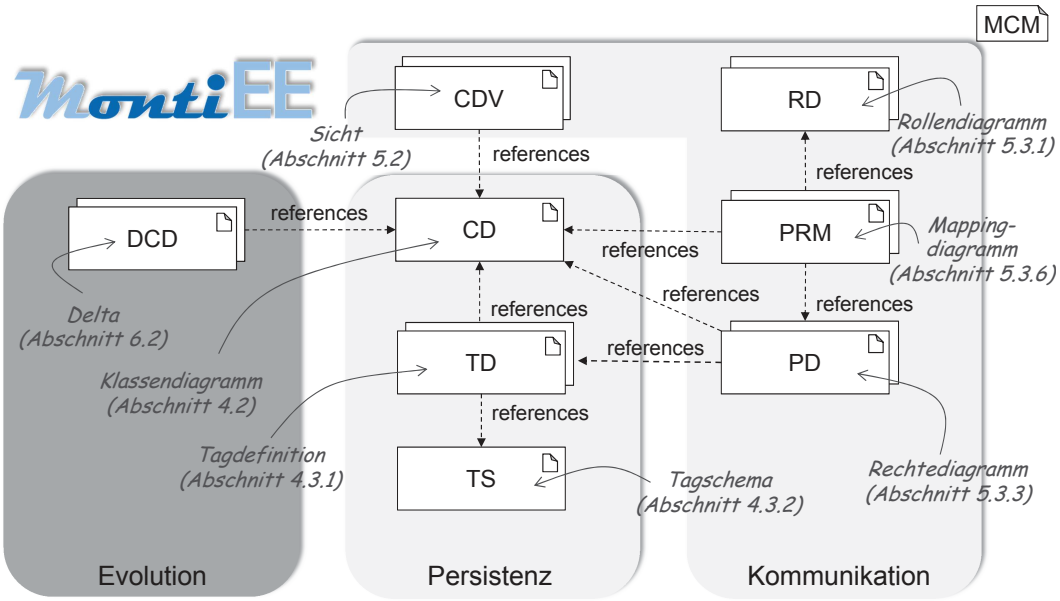


Abbildung 6.1: Überblick über die Modelle der Sprachfamilie MontiEE und deren Zusammenhänge. Der Fokus liegt dabei auf den Modellen zur Modellierung der Evolution von Enterprise Applikationen.

6.1 Überblick

Abbildung 6.1 greift Abbildung 4.1 auf und zeigt den Fokus dieses Kapitels. Im Rahmen dieser Arbeit wurde für den Produktentwickler eine Deltasprache entwickelt, deren Modelle die Systemevolution beschreibt und auf deren Basis eine Datenmigration erfolgen kann. Die Modelle (DCD) referenzieren dabei Klassendiagramme, da sich die Evolution immer auf die Elemente des Klassendiagramms bezieht. Die Veränderung der Datentypen wird in Deltas modelliert. In Abschnitt 6.2 werden die Definition und die Semantik dieser Sprache detailliert vorgestellt. Die teilautomatisierte Migration vorhandener Daten sowie die Verwendung für Codegeneratoren werden in Kapitel 9 vorgestellt.

6.2 Modellevolution

Neben der Modellierung des Systems, geeigneter Sichten und verschiedener Rechte und Rollen, auf deren Basis das System generiert werden kann, ist es für den Produktentwickler wichtig, dass das System strukturiert weiterentwickelt werden kann. Zur Modellierung von Evolution wurde daher im Rahmen dieser Arbeit eine Deltasprache für Klassendiagramme geschaffen, die es erlaubt, strukturelle Veränderungen an Klassendiagrammen zu modellieren. Diese Deltasprache, ihre Operationen und Refactoringoperationen werden in diesem Abschnitt vorgestellt. Neben der Evolution des Datenmodells fallen in einem Produktivsystem Daten, welche trotz der Evolution erhalten bleiben müssen, an. Dies

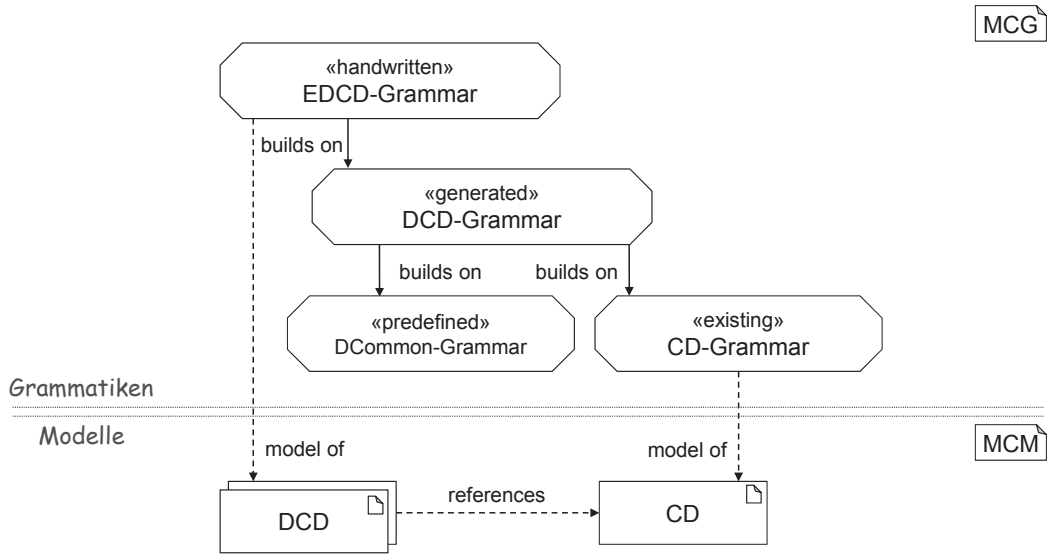


Abbildung 6.2: Abhängigkeiten zwischen den Grammatiken der Deltasprache, ihrer Supergrammatik, der handgeschriebenen Erweiterung und der Grammatik der Klassendiagrammsprache sowie ihrer Modelle.

ist vor allem bei strukturellen Änderungen des Domänenmodells, da sich dadurch auch das zugrunde liegende Datenbankschema sowie das objektrelationale Mapping ändert, entscheidend. Dazu wird im Rahmen dieser Arbeit eine Koevolution von Objektdiagrammen verwendet. Diese Objektdiagramme, deren Serialisierung und Deserialisierung in Abschnitt 9.4 vorgestellt wird, beschreiben die persistent gespeicherten Daten und sind konform zu dem modellierten Klassendiagramm. Die Deltas selbst beschreiben nur die Veränderung des Klassendiagramms. Darüber hinaus benötigen sie zusätzliche Information, die die Migration der Objektdiagramme anleitet. Auf Basis dieser Deltas wird dazu eine Infrastruktur generiert, die die Evolution und die Migration ermöglicht. Diese Infrastruktur wird in Abschnitt 9.3 vorgestellt.

Zum Verständnis der geschaffenen Sprachen wird zunächst ein Überblick über die Zusammenhänge der einzelnen Sprachen gegeben und diese daran anschließend vorgestellt. Abbildung 6.2 zeigt die Beziehungen zwischen den beteiligten Sprachen und Modellen. Die Deltasprache ist spezifisch für Klassendiagramme generiert. Ihre Grammatik erbt sowohl von der Grammatik der Klassendiagrammsprache, die in Abschnitt 2.3 vorgestellt wurde als auch von der Grammatik DCommon-Grammar, einer gemeinsamen Basissprache, die für alle Deltasprachen verwendet werden kann. Dies wird durch die `builds on` Beziehung ausgedrückt und durch die Möglichkeit der Mehrfachvererbung bei Grammatiken in MontiCore umgesetzt. Die Beziehung zwischen der Grammatik der Deltasprache und DCommon-Grammar resultiert aus den bereitgestellten Interfaces der DCommon-Grammar. Darüber hinaus kann die handgeschriebene Grammatik EDCD-Grammar die Produktionen aus DCD-Grammar der Deltasprache verfeinern und anpassen, so dass

auch hier eine Beziehung zwischen diesen beiden Grammatiken existiert. Modelle der Sprache, Deltas, referenzieren Elemente eines Klassendiagramms und modellieren die Evolution dieser referenzierten Elemente.

Die Methodik und die gemeinsame Basissprache wurden nicht im Rahmen dieser Arbeit entwickelt, sondern werden nur zum besseren Verständnis und der Vollständigkeit halber kurz vorgestellt. Die Methodik wird in [HHK+13, HHK+15] detailliert erläutert.

Generell ist die Idee zur toolgestützten Evolution und Datenmigration nicht neuartig. Es gibt in vielen Bereichen, wie Metamodelevolution, Datenbankschemaevolution, XML-Schemaevolution oder Grammatikevolution bereits Ansätze zur Lösung. Vor allem der Bereich der Datenbankschemaevolution ist weit erforscht [BBC01]. Neben der reinen Evolution beschäftigen sich die meisten dieser Ansätze auch mit der Fragestellung, wie konforme Informationen, wie Modelle eines Metamodells, Daten eines Datenbankschemas, XML-Dateien, die konform zu einem XML-Schema sind, oder Sprachen, die durch eine Grammatik definiert werden, ebenfalls migriert werden können. Im Rahmen dieser Arbeit wird die Evolution auf Klassendiagrammen, wie in Abschnitt 9.2 präsentiert wird, beschrieben und eine Migration der Objektdiagramme, die in Abschnitt 9.3 dargelegt wird, abgeleitet. Dabei stellt das Klassendiagramm das Metamodell des Objektdiagramms oder das Schema der Datenbank dar. Auch wenn diese Beziehung, wie in Abschnitt 2.3.2 erläutert, aus modellierungssprachtheoretischer Sicht nicht korrekt ist, werden Objektdiagramme als konforme Instanzen eines Klassendiagramms angenommen, so dass die Fragestellungen hinsichtlich der Evolution übertragen werden können. Auch stellt das Klassendiagramm nicht direkt das Datenbankschema dar, sondern wird zur Generierung des Schemas verwendet, so dass auch hier die Fragestellungen übertragbar sind. Aus diesem Grund werden zunächst Ansätze der Schemaevolution sowie der Metamodellevolution vorgestellt und Gemeinsamkeiten sowie Unterschiede aufgezeigt.

Wedemeijer [Wed99] identifiziert dazu eine Reihe möglicher Evolutionsschritte, die konzeptionell für ein Datenbankschema auftreten können. Zudem beschreibt er mögliche Umsetzungen dieser Schritte. Dabei beschreibt er Veränderungen für Entitäten und Relationen auf Schemaebene, beschreibt aber keine Veränderungen einzelner Attribute. Die im Rahmen dieser Arbeit entwickelte Sprache sowie die generative Umsetzung der Migration, wie in Kapitel 9 beschrieben, kann dabei als Umsetzung der von Wedemeijer beschriebenen Schritte auf Klassendiagrammen angesehen werden.

Ein weiterer Ansatz, der mögliche Veränderungen der Metamodelle beschreibt, wird in [BG10] gegeben. Dort werden Veränderung an Meta Object Facility (MOF) [WS08, OMG15a] konformen Metamodellen auf ihre Auswirkungen auf M1-Modelle untersucht. Dabei zeigt sich, dass manche Änderungen keine Auswirkungen haben, manche sich automatisiert beheben lassen und einige nicht automatisiert zu beheben sind. Eine ähnliche Klassifikation und eine semi-automatisierte Lösungsstrategie auf Basis von Petrinetz Metamodellen wird in [CREP08] gegeben. Auch hier zeigt sich, dass einzelne Änderungen automatisiert erfolgen können, aber auch Nutzerinteraktion benötigt wird. Viele Ansätze beschäftigen sich mit der Frage der Sprachevolution und der Frage, welche Rahmenbedingungen erfüllt sein müssen, um eine solche Evolution erfolgreich durchführen zu können [RIP11]. Dazu gehören periphere Elemente wie Editoren oder auch existierende Trans-

formationen [RIP12b]. Im Rahmen dieser Arbeit findet keine klassische Veränderung des Metamodells, sondern eine Klassendiagrammevolution und eine analoge Evolution eines konformen Objektdiagramms, statt. Auch hier existieren strukturelle Veränderungen, die nur das Klassendiagramm betreffen, Veränderungen, die im Objektdiagramm automatisch aufgelöst und angewendet werden können und solche, wo mit Hilfe einer zusätzlichen Infrastruktur auf handgeschriebenen Quellcode zurückgegriffen werden muss, der die Veränderung umsetzt.

Auch das bereits zuvor erwähnte Framework WebDSL [GHKV08, Vis08] bietet Mechanismen zur Datenmigration [VWV11] an. Dazu klassifiziert es die möglichen Migrationen in drei unterschiedliche Arten: Schemamodifikationen, verlustbehaftete Modifikationen und erhaltende Modifikationen. Schemamodifikationen beziehen sich auf Operationen, die zwar das Schema, also das Datenmodell, aber nicht die gespeicherten Daten, betreffen. Beispiele dafür sind das Hinzufügen von Feldern oder neuen Datentypen. Konservative Modifikationen beschreiben eine Veränderung der gespeicherten Daten, die den Informationsgehalt beibehalten. Beispiele dafür sind im Wesentlichen das Verschieben oder Umbenennen einzelner Felder. Verlustbehaftete Operationen hingegen beziehen sich ebenfalls auf die gespeicherten Daten und beschreiben Veränderungen, die den Informationsgehalt der Daten, wie beispielsweise das Löschen einzelner Felder, verringern. Innerhalb von WebDSL wird eine eigenständige DSL, die zur Modellierung der Evolution und der verschiedenen Operationen verwendet wird, eingesetzt. Auch im Rahmen dieser Arbeit lässt sich diese Dreiteilung an den verschiedenen Deltaoperationen ablesen. Die entwickelte Sprache bietet ebenfalls Operationen zum Hinzufügen, zum Refactoring oder zum Löschen an. Allerdings ist der konzeptionelle Ansatz in [VWV11] ein anderer. Während dort eine direkte Datenbank Modifikation auf Basis der Datenbankzielsprache generiert wird, verfolgt der Ansatz dieser Arbeit das Paradigma, die Daten einzulesen, zu transformieren und nach erneuter Datenbankinitialisierung die Daten zurückzuschreiben. Zudem verwendet der Ansatz aus [VWV11] Adaptertabellen, die unterschiedliche Datenversionen emulieren.

In [COV06, BCPV07] wird eine typsichere Evolution auf zwei verschiedenen Ebenen vorgestellt. Dabei wird auf XML-Schemas und SQL-Schemas fokussiert. Hierbei wird ein zugrunde liegendes Framework vorgestellt, auf dessen Basis ein SQL- sowie ein XML-Frontend geschaffen wurden. Zur Angabe der Evolution wird Haskell verwendet. Die XML-Evolution basiert dabei auf den Erkenntnissen aus [LL01]. Die Autoren verwenden zwei verschiedene Ebenen: `type-level` und `value-level`. Diese Ebenen beinhalten die Schemaevolution, also eine Evolution auf Typebene, und die Migration, also eine Evolution auf Wertebene. Auch dieser Ansatz zeigt Parallelen zu dem hier vorgestellten Vorgehen, unterscheidet sich aber in dem Fokus auf SQL und der zur Laufzeit stattfindenden Transformation der Daten der Datenbank.

Eine Umsetzung innerhalb von EMF stellt das COPE Framework [Her11] dar. Hierbei werden Metamodelländerungen in einem History-Modell gespeichert und können nachvollzogen werden. Dabei muss die Veränderung nicht explizit modelliert werden, sondern wird rein auf den Änderungen am Metamodell inferiert. Sollte dies nicht möglich sein, wird es gemeldet und der Benutzer muss die Veränderungen manuell implementieren.

Im Rahmen dieser Arbeit wird keine automatische Inferierung der Änderungen angestrebt, sondern die Veränderungen müssen explizit modelliert werden. Eine Übersicht über weitere existierende Tools wird in [RIP12a] gegeben.

Auch die Form der verwendeten Sprache zur Modellierung der Evolution unterscheidet sich von Ansatz zu Ansatz. Häufig werden keine Delta-, sondern Transformationssprachen verwendet [Wac07, CREP08]

Das Konzept der Deltasprachen entspringt der Modellierung von Softwareproduktlinien. Sie ermöglichen die Modellierung der gewünschten Variabilität der Produktlinie als Transformationsansatz [HKR+11]. Darüber hinaus existieren annotative und kompositionale [VG07, HRRS11] Ansätze zur Modellierung von Variabilität. Typischerweise liegt Transformationsansätzen immer ein Kernmodell, welches durch geeignete Transformationen in ein konkretes Modell, das die entsprechenden Varianten enthält, überführt wird, zu Grunde. Dabei fügen Deltas typischerweise Elemente hinzu, entfernen Elemente oder verändern bzw. ersetzen einzelne Elemente. Annotative Ansätze verwenden ein 150% Modell und können daher im Rahmen dieser Arbeit nicht verwendet werden. Kompositionale Ansätze komponieren Modelle auf Basis benötigter Features und sind im Rahmen dieser Arbeit ebenfalls nicht weiter berücksichtigt. Eine detaillierte Abgrenzung zur Verwendung von Deltasprachen wird in [HHK+13, HHK+15] gegeben.

Im Rahmen dieser Arbeit werden die Evolution des Domänenmodells sowie die Migration durch die Deltasprache auf die Modellierungsebene gehoben, so dass eine Modellierung der Systemevolution und der Datenmigration ermöglicht wird.

6.2.1 Die Deltasprache

Zur Modellierung der Evolution von Klassendiagrammen wurde eine Deltasprache entwickelt. Sie basiert auf der in [HHK+13, HHK+15] vorgestellten Methodik zur systematischen Entwicklung von Deltasprachen.

Dieser Methodik folgend, lässt sich eine klassendiagrammspezifische Deltasprache zu einer beliebigen Ausgangssprache ableiten. Wie schon bereits bei den Taggingsprachen, basiert auch diese Methodik darauf, durch Konzepte der Sprachvererbung einen sprachunabhängigen Teil einer Grammatik in einer gemeinsamen Grammatik bereitzustellen und die zur Ausgangssprache spezifischen Teile der Grammatik der Deltasprache hinzufügen zu können. Die Methodik aus [HHK+13, HHK+15] zeigt aber darüber hinaus, dass auch weite Teile des sprachspezifischen Teils der Grammatik der Deltasprache automatisiert generiert werden können und lediglich minimale Anpassungen in einer handgeschriebenen Erweiterung erforderlich sind. Mit Hilfe dieser Sprache lässt sich die Evolution des Systems modellieren und mit Hilfe dieser Modelle und geeigneter Codegeneratoren Infrastruktur, die in Kapitel 9 vorgestellt wird, generieren, die eine verlustfreie Datenmigration ermöglicht (vgl. FA8-AG, FA10-PE und FA11-PE).

Zunächst wird die konkrete Syntax der Deltas vorgestellt. Darauf aufbauend werden die handgeschriebene Erweiterung der Grammatik sowie die in der Vererbungshierarchie beteiligten Grammatiken erläutert. Als Teil der konkreten Syntax werden auch die möglichen Operationen der Deltas vorgestellt.

```
1  package montiee.deltas;                                          DCD
2
3  delta AddLabel {
4
5    modify classdiagram SocNet{
6      add class PhotoPost extends Post{
7      String metadata;
8      }
9
10     add association labeledIn [*] Person -> PhotoPost [*];
11   }
12
13 }
```

Listing 6.3: Auszug der Definition des Deltas `AddLabel`. Modifiziert wird das Klassendiagramm `SocNet`. Es werden eine Klasse und ein Attribut hinzugefügt.

Zur Erweiterung der Anforderungen aus Abschnitt 3.5 möchte der Auftraggeber die Funktionalität des sozialen Netzwerks um die Möglichkeit von `PhotoPosts` und dem Labeln von Personen in Fotos erweitern. Listing 6.3 zeigt dazu ein Delta, welches dem Klassendiagramm des sozialen Netzwerks eine `PhotoPost` Klasse, die von der ursprünglichen Klasse `Post` erbt, hinzufügt. Zudem wird eine Assoziation, die die Klasse `Person` und die Klasse `PhotoPost` verbindet, hinzugefügt. Ein Delta beginnt, wie bereits von den zuvor vorgestellten Sprachen bekannt, mit der Angabe eines Pakets. Danach wird das Delta mit dem Schlüsselwort `delta`, gefolgt von einem Namen, eingeleitet. Innerhalb dieses Blockes können dann mehrere `modify` Ausdrücke oder unterschiedliche Operationen stehen. Ein `modify` Ausdruck beginnt mit dem gleichnamigen Schlüsselwort, gefolgt von einem Identifikator des zu verändernden Elements, gefolgt von einer Namensreferenz auf das entsprechende Element. Es sei dabei angemerkt, dass wie in Listing 6.3 gezeigt, das Klassendiagramm selbst auch ein veränderbares Element ist. Innerhalb eines `modify` Ausdrucks können weitere `modify` Ausdrücke oder aber unterschiedliche Operationen geschachtelt sein. Mit Hilfe der `add` Operation lassen sich Elemente hinzufügen. Eine solche Operation beginnt mit dem Schlüsselwort `add`, gefolgt von dem hinzuzufügenden Element. In Listing 6.3 wird eine Klasse mit dem Namen `PhotoPost` und dem enthaltenen Attribut `metadata` hinzugefügt. Dabei entspricht die konkrete Syntax des hinzuzufügenden Elements der konkreten Syntax der Klassendiagrammelemente. Dies wird im weiteren Verlauf bei der Vorstellung der entsprechenden Grammatik und der allgemeinen Methodik weitergehend erläutert.

Zudem wird in Listing 6.3 eine Assoziation hinzugefügt. Es können aber mit Hilfe handgeschriebener Erweiterungen der Grammatik der Deltasprache nicht nur vollständige Elemente adressiert oder verändert werden, sondern auch Teile dieser Elemente, wie beispielsweise Rollennamen der linken oder der rechten Assoziationsseite oder die Navigationsrichtung.

Die `add` Operation dient dem Hinzufügen von Elementen zu einer Menge von Ele-

menten eines Modells, beispielsweise dem Hinzufügen einer Klasse zu einem Klassendiagramm. Neben der `add` Operation stehen ferner die Operationen `remove` und `set` zur Verfügung. Die set Operation dient dem Setzen eines einzelnen Elements, wie beispielsweise dem Setzen des Typs eines Attributs. Die remove Operation dient dem Entfernen von Elementen.

```
1  modify class SocNet.Post {                                    DCD
2    remove attribute hasPhoto;                                  ...
3  }
```

Listing 6.4: Modifikation der Klasse `Post`. Das Attribut `hasPhoto` wird entfernt.

Listing 6.4 zeigt zwei unterschiedliche Dinge. Zum einen zeigt es eine `remove` Operation, die das Attribut mit dem Namen `hasPhoto` entfernt. Die `remove` Operation für Klassen wurde in der handgeschriebenen Grammatik dahingehend erweitert, dass zu Beginn der Operation ein `force remove` angegeben werden kann. Dies wird vor allem dann benötigt, wenn Klassen oder Interfaces eines Klassendiagramms entfernt werden sollen. Haben diese Klassen eingehende Assoziationen, müssen diese ebenfalls behandelt werden. Generell existiert hier die Möglichkeit, dass alle eingehenden Assoziationen ebenfalls entfernt werden, oder ein Entfernen verboten wird. Dies ist das Standardverhalten der `remove` Operation. Ein Verbot führt in der Praxis allerdings dazu, dass Klassen de facto nie entfernt werden können. Allerdings kann der Modellierer sich zuvor darum kümmern, die eingehenden Assoziationen ebenfalls umzuändern. Durch die Angabe von `force` kann das Verhalten geändert werden, so dass das Element und eingehende Assoziationen entfernt werden. Da dies die Ausnahme in der Modellierung sein sollte, wurde das Standardverhalten wie beschrieben derart gewählt. Ein Aufheben des Verbots muss daher explizit modelliert werden.

Neben der `remove` Operation zeigt Listing 6.4 zum anderen, dass die Adressierung eines Elements in einem `modify` Block entweder, wie gezeigt, über einen durch Punkte getrennten Pfad zum Element oder über verschachtelte Blöcke modelliert werden kann. Es sei dabei angemerkt, dass die Adressierung der Elemente sowohl innerhalb eines `modify` Ausdrucks als auch in einer `remove` Operation über eindeutige Namen der Elemente erfolgt. Besitzt ein Element keinen Namen, wie beispielsweise einige Assoziationen, wird die vollständige konkrete Syntax des Elements in eckigen Klammern verwendet. Dies wird im weiteren Verlauf bei der Vorstellung der entsprechenden Grammatik und der allgemeinen Methodik weitergehend erläutert.

Weitergehende Operationen

Neben den Basisoperationen `add`, `set` und `remove` wurden im Rahmen dieser Arbeit weitergehende Operationen, die sich aus den Basisoperationen zusammensetzen, umgesetzt.

```
1  extract class Person {                                    DCD
2    int zip;                                                ...
3    String street;
4    String city;
5  } as class Address;
```

Listing 6.5: Darstellung der `extract` Operation. Die Klasse `Address` wird aus der Klasse `Person` mit den angegebenen Attributen extrahiert.

Listing 6.5 zeigt die Verwendung der `extract` Operation. Diese Operation extrahiert eine Menge von Attributen einer Klasse, in Listing 6.5 aus der Klasse `Person`, in eine neue Klasse, wie beispielsweise die Klasse `Address`. Dabei lässt sich diese weiterführende Operation ebenfalls durch eine Konkatenation der Basisoperationen erreichen. Im Wesentlichen bedeutet ein Extrahieren einer Klasse ein Hinzufügen einer neuen Klasse mit den entsprechenden Attributen, ein Entfernen der Attribute aus der ursprünglichen Klasse und ein Aufspannen einer Assoziation zwischen beiden Klassen, mit der Kardinalität `1` an beiden Enden der Assoziation und einer Navigationsrichtung von der ursprünglichen Klasse zur extrahierten Klasse.

Insgesamt beinhaltet die handgeschriebene Erweiterung der Deltasprache sechs weitere Operationen, welche auf einer Auswahl der in [HHK+13, HHK+15] vorgestellten Refactoring Operationen basieren. Nachfolgende Aufzählung erläutert die Operationen kurz.

1. `rename`: Die `rename` Operation dient der Umbenennung eines Elements. Sie kann auf Klassen, Interfaces, Attribute, Methoden, Parameter, Enumerationen, Enumerationskonstanten, Assoziationen und Methoden angewandt werden.

2. `move`: Die `move` Operation dient dem Verschieben eines Attributs oder einer Methode zu einer anderen Klasse.

3. `merge`: Die `merge` Operation vereinigt zwei oder mehr Klassen oder Interfaces zu einer neuen Klasse.

4. `extract`: Die `extract` Operation erhält als Parameter eine Menge von Attributen oder Methoden und extrahiert diese in eine neue Klasse. Dabei ist es möglich, diese neue Klasse als Super- oder Subklasse der ursprünglichen Klasse zu extrahieren. Auch die Extraktion als Interface ist möglich.

5. `pullup`: Die `pullup` Operation erlaubt das Hochziehen gemeinsamer Attribute von Unterklassen in eine gemeinsame Superklasse.

6. `pushdown`: Die `pushdown` Operation ist die inverse Operation zu der pullup Operation und dient dem Herunterziehen von Attributen einer Superklasse in alle Subklassen.

Bei der Migration persistenter Daten in Form von Objektdiagrammen, die in Abschnitt 9.3 präsentiert wird, werden nicht nur das Klassendiagramm verändert, sondern auch die persistenten Daten entsprechend migriert. Eine `move` Operation für ein Attribut muss beispielsweise den Attributwert im Objektdiagramm erhalten oder würde mit initialen Werten befüllt.

```
1  modify SocNet.Commercial {                                    DCD
2    add attribute int maxFollowers                               ...
3      init with FollowerInitializer;
4  }
```

Listing 6.6: Darstellung der Modellierung einer Attributinitialisierung. Das hinzugefügte Attribut `maxFollowers` wird mit dem Initializer `FollowerInitializer` initialisiert.

Listing 6.6 zeigt das Hinzufügen des Attributs `maxFollowers` zur Klasse `Commercial`. Dabei wird die Möglichkeit der Angabe eines Initialisierers, der das Attribut auf Basis des vorhandenen ASTs initialisieren kann, genutzt. Der Initialisierer erhält den vollständigen AST und kann das Attribut beliebig initialisieren. Diese Angabe hat keine Auswirkung auf das Klassendiagramm, sondern lediglich auf die Migration der Objektdiagramme. In Abschnitt 9 wird die Migration und auch die Möglichkeit zur Attributinitialisierung detailliert erläutert.

Abbildung 6.7 zeigt das Klassendiagramm aus Abbildung 4.2. Das Klassendiagramm wurde um die durch die zuvor vorgestellten Deltaoperationen hinzugefügten Elemente erweitert. Basierend auf Listing 6.3 wurde die Klasse `PhotoPost` hinzugefügt und die Assoziation entsprechend eingefügt. Darüber hinaus wurde das Attribut `hasPhoto` aus der Klasse `Post`, wie in Listing 6.4 modelliert, entfernt. Auch die Anwendung der in Listing 6.5 modellierten weitergehenden Operation ist in Abbildung 6.7 gezeigt. Außerdem beinhaltet die Klasse `Commercial` das neue Attribut `maxFollowers`

Neben den hier vorgestellten weiterführenden Operationen sind weitere Operationen denkbar, aber im Rahmen dieser Arbeit nicht umgesetzt. Im Folgenden wird der Entwurf der Grammatiken der Deltasprache und ihrer handgeschriebenen Erweiterung näher erläutert.

Die Grammatiken der Deltasprache

Zum besseren Verständnis der Grammatiken wird kurz die in [HHK+15] vorgestellte Methodik wiedergegeben und auf Klassendiagramme im Speziellen angewendet.

Die gemeinsame Supergrammatik gibt die allgemeine syntaktische Struktur konkreter Deltas vor und definiert gleichzeitig Interface-Produktionen, die von Subsprachen implementiert werden müssen. Darüber hinaus werden auch Basisoperationen sowie Erweiterungspunkte für komplexere Operationen definiert.

Listing 6.8 zeigt einen Ausschnitt der gemeinsamen Supergrammatik der Deltasprache.

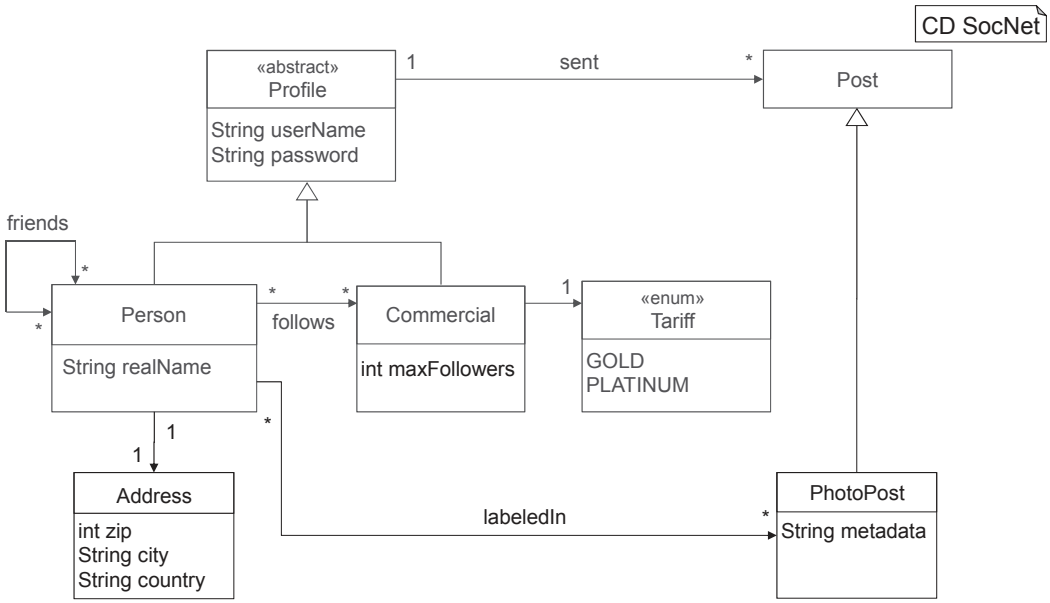


Abbildung 6.7: Darstellung des Klassendiagramms aus Abbildung 4.2 nach Anwendung der Deltaoperationen der Listings 6.3, 6.4, 6.5 und 6.6.

Dieser Ausschnitt zeigt die angebotenen Interfaces, die von spezifischen Deltasprachen implementiert werden müssen. Diese Interfaces sind `ScopeIdentifier` und `ModelElementIdentifier`. Das Interface `ScopeIdentifier` dient dazu, den Typ eines Modelelements identifizieren zu können, wohingegen das `ModelElementIdentifier` Interface dazu dient, ein konkretes Element identifizieren zu können. Die beiden außerdem vorhandenen Interfaces `DeltaOperation` und `DeltaOperand` werden von anwendbaren Operationen implementiert. Der Operand ist dabei jeweils das Schlüsselwort der jeweiligen Operation. Alle Operationen können innerhalb eines `modify`-Blocks verwendet werden.

```
1  interface DeltaOperand;
2
3  interface DeltaOperation;
4
5  interface ModelElementIdentifier;
6
7  interface ScopeIdentifier;
```

MCG
D-Common
...

Listing 6.8: Darstellung der Interface-Produktionen als Erweiterungspunkte zur Erstellung einer sprachspezifischen Deltasprache. Eine detaillierte Erklärung der gemeinsamen Supergrammatik der Deltasprache und der Konstrukte ist in [HHK+13, HHK+15] zu finden.

Wie in Listing 6.9 gezeigt, wird auch die generelle Struktur vorgegeben. Ein Delta beginnt mit dem Schlüsselwort `delta` gefolgt von einem Namen und beinhaltet Elemente, die das Interface `DeltaElement` implementieren. Dieses Interface und auch das Interface `DeltaOperation` wird von der Produktion `DeltaModify` implementiert. Dadurch kann ein `modify` Ausdruck als Element eines Deltas benutzt und gleichzeitig, da ein `modify` Ausdruck beliebig viele `DeltaOperation` Elemente enthalten kann, verschachtelt werden. Gleichzeitig wird das generelle Aussehen eines `modify` Ausdrucks vorgegeben. Der `modify` Ausdruck beginnt, wie zuvor beschrieben, mit dem entsprechenden Schlüsselwort, gefolgt von einem Identifikator des Elementtyps und einem Namen oder einem Pfad, bestehend aus mehreren Namen, getrennt durch einen Punkt zur verschachtelten Navigation. Ebenso wird in der gemeinsamen Supergrammatik die `remove` Operation in ihrer Standardausprägung vorgegeben. Diese sieht vor, dass die Operation mit dem Schlüsselwort `remove` beginnt und ebenfalls einen Namen oder einen Pfad des zu löschenden Elements beschreibt. Eine detaillierte Erklärung der gemeinsamen Supergrammatik der Deltasprache und der Konstrukte ist in [HHK+13, HHK+15] zu finden.

```
1   interface DeltaElement;                                          MCG
2                                                                    D-
3   Delta =                                                          Common
4     "delta" Name "{"                                               ...
5       elements:DeltaElement*
6     "}";
7
8   DeltaModify implements DeltaOperation, DeltaElement =
9     "modify" ScopeIdentifier ModelElementIdentifierPath
10    "{"
11      DeltaOperation*
12    "}";
```

Listing 6.9: Dargestellt ist die Startproduktion `Delta` und die Produktion `DeltaModify`. Weitere Produktionen der Grammatik sind ausgelassen. Eine detaillierte Erklärung der gemeinsamen Supergrammatik der Deltasprache und der Konstrukte ist in [HHK+13, HHK+15] zu finden.

Spezifische Subsprachen müssen die Interfaces der gemeinsamen Supergrammatik implementieren und können dadurch in die vorgegebene Struktur eingebettet werden. Auf Implementierungsebene wird dazu der Sprachvererbungsmechanismus von MontiCore verwendet. Die Deltasprache kann automatisiert abgeleitet werden.

Listing 6.10 zeigt einen Auszug der automatisch generierten Grammatik für die Klassendiagrammsprache, basierend auf der in [HHK+13, HHK+15] vorgestellten Methodik. Dabei erbt diese Grammatik von der zuvor vorgestellten gemeinsamen Basissprache und implementiert das bereitgestellte Interface `ScopeIdentifier`. Im dargestellten Listing 6.10 werden die Interfaces spezifisch für die Modifikation einer Klasse implementiert. Die Implementierung des Interfaces `ScopeIdentifier` ermöglicht dabei, Klassen in ei-

```
1  grammar DCD extends DCommon {                                    MCG
2                                                                   DCD
3    DeltaCDClassScopeIdentifier implements ScopeIdentifier =       ...
4      "CDClass";
5
6    DeltaCDClassOperation implements DeltaOperation =
7      DeltaOperand CDClass;
8
9    DeltasuperclassesReferenceTypeDeltaOperation
10     implements DeltaOperation =
11       DeltaOperand "superclasses" Name ";";
12
13 }
```

Listing 6.10: Aufbereiteter Auszug der Grammatik der Deltasprache. Dargestellt sind die benötigten sprachspezifischen Implementierungen der Interface Produktionen. Die vollständige Version der Grammatik ist in Anhang C.10 dargestellt.

nem `modify` Ausdruck zu adressieren. Die Implementierung des Interfaces `ModelElementIdentifier` wurde ausgelassen, da die gemeinsame Supergrammatik den Namen eines Elements als Standardidentifikator definiert. In der generierten Sprache müssen, der Methodik folgend, nur Elemente, die keinen oder keinen eindeutigen Namen besitzen, das Interface `ModelElementIdentifier` implementieren. Zudem wird, wie Listing 6.10 darstellt, die Produktion für klassenspezifische Operationen generiert. Diese Produktion verwendet die zur Verfügung stehenden Operanden und das Nichtterminal der Grammatik der Klassendiagramme. Dadurch kann ein Hinzufügen und ein Entfernen einer Klasse ausgedrückt werden. Durch die Verwendung des Nichtterminals der Grammatik der Klassendiagramme wird an dieser Stelle eine vollständige Definition einer Klasse in der bekannten Syntax der Klassendiagramme ermöglicht. Gleichzeitig existiert durch diese Verwendung auch eine Beziehung zwischen Delta und Klassendiagramm. Die letzte Produktion, die in Listing 6.10 dargestellt ist, zeigt die ebenfalls automatisiert generierte Produktion, die für das Hinzufügen und Entfernen von Klassen in einer Vererbungshierarchie verantwortlich ist. Dabei lässt sich erkennen, dass das automatisiert gewählte Schlüsselwort `superclasses` nicht gut geeignet ist. Dies kann in der handgeschriebenen Erweiterung der Deltasprache überschrieben werden. Die genauen Regeln zur automatisierten Ableitung der Deltasprache lassen sich [HHK+13, HHK+15] entnehmen. Die vollständige Version der generierten Grammatik ist im Anhang in Abschnitt C.10 dargestellt. Auf Basis der Grammatik der Deltasprache können mit Hilfe der handgeschriebenen Erweiterung der Grammatik Änderungen, Erweiterungen und Individualisierungen integriert werden. Im Rahmen der Deltasprache für Klassendiagramme werden die vorgestellten weiterführenden Operationen gezeigt. Zudem werden syntaktische Anpassungen, die Erweiterung der `remove` Operation und die Möglichkeit der Initialisierer gezeigt.

```
grammar EDCD extends DCD {                                          MCG
                                                                    EDCD
  DeltaCDClassScopeIdentifier implements ScopeIdentifier =          ...
    "class";

  DeltasuperclassesReferenceTypeDeltaOperation implements
      DeltaOperation =
   DeltaOperand "superclass" Name ";";

  // weitere Produktionen
}
```

Listing 6.11: Aufbereiteter Auszug der handgeschriebenen Erweiterung der Grammatik der Deltasprache. Dargestellt ist eine benutzerdefinierte Produktion. Die vollständige Version der Grammatik ist in Anhang C.11 dargestellt.

Listing 6.11 zeigt die Grammatik, die die zuvor vorgestellte automatisch abgeleitete Deltasprache DCD erweitert. Zudem wird das Überschreiben der Produktionen gezeigt, die in der generierten Sprache eine ungewollte konkrete Syntax erzeugt. Diese wurde in Listing 6.11 in `superclass` abgeändert. Das Überschreiben von Produktionen wird durch den Sprachvererbungsmechanismus von MontiCore ermöglicht und stellt die erste Möglichkeit zur Individualisierung der Sprache dar.

```
  Initializer = "init" "with" fullQualifiedName:Name ";";           MCG
                                                                    EDCD
  DeltaCDAttributeOperation implements DeltaOperation =             ...
    operand:DeltaOperand CDAttribute Initializer?;

  Flag = "force";

  DeltaRemoveOperation implements DeltaOperation =
    Flag? DeltaRemove target:ModelElementIdentifierPath ";";
```

Listing 6.12: Dargestellt sind die `Initializer` Produktion sowie das hinzugefügte `Flag` zum Entfernen von Elementen. Die vollständige Version der Grammatik ist in Anhang C.11 dargestellt.

Eine weitere Möglichkeit ist in Listing 6.12 dargestellt. Dazu werden die Operationen für Attribute überschrieben und durch die Möglichkeit zur Angabe eines Initialisierers erweitert. Zur Verwendung eines Initialisierers muss demnach der vollqualifizierte Name des aufzurufenden Initialisierers angegeben werden. Dies ermöglicht die Modellierung der in Listing 6.6 dargestellten Attributinitialisierung

Zudem wird die Standardproduktion für das Entfernen von Elementen überschrieben und um ein `Flag` erweitert. Auch hier wird der Mechanismus zum Überschreiben von Produktionen, der es ermöglicht, bestehende Funktionalität zu verändern und somit die

Sprache zu erweitern, genutzt. Dies ermöglicht die Modellierung des beschriebenen forcierten Entfernens eines Elements. Die vollständige Version der erweiterten Grammatik ist im Anhang in Abschnitt C.11 dargestellt.

```
1  DeltaExtractClassOperation implements DeltaOperation =
2    DeltaExtract "class" className:Name
3    "{"
4      (
5        ("method" methodName:Name ";")
6        |
7        ("attribute" attributeName:Name ";")
8      )*
9    "}" "as" CDClass;
```

MCG
EDCD
...

Listing 6.13: Dargestellt ist die `DeltaExtractClassOperation` Produktion, die eine weitergehende Operation definiert. Die vollständige Version der Grammatik ist in Anhang C.11 dargestellt.

Neben den zuvor genannten Möglichkeiten, bestehende Syntax oder Funktionalität mit Hilfe des Überschreibens von Produktionen zu verändern, besteht zudem über die bereitgestellten Interfaces die Möglichkeit des Hinzufügens neuer Funktionalität. Dazu zeigt Listing 6.13 die Definition der `extract` Operation. Die Produktion `DeltaExtract-ClassOperation` implementiert das bereitgestellte Interface `DeltaOperation` und kann dadurch innerhalb eines `modify` Ausdrucks verwendet werden. Der Operand wird ebenfalls definiert und in der Operation verwendet. Der Rest der Produktion zeigt die Definition der abstrakten und konkreten Syntax, so dass die `extract` Operation, wie in Listing 6.5 dargestellt, ausgedrückt werden kann. Auch hier wird das Nichtterminal `CDClass` der Klassendiagrammsprache verwendet. Es ist auffällig, dass an dieser Stelle nicht die Nichtterminale der Methoden und Attribute verwendet werden, sondern Namen als Referenzen auf Attribute und Methoden. Dies ist dadurch begründet, dass sich die Operation auf bestehende Elemente bezieht und somit eine Namensreferenz ausreicht.

Alles in allem kann durch die vorgestellten Sprachen und ihren Abhängigkeiten untereinander eine Deltasprache definiert werden, die es ermöglicht, die Datenmodellmigration, aber auch die Datenmigration und Initialisierung zu modellieren. Die Umsetzung der Migration und der Migration wird in Kapitel 9 detaillierter erläutert. Nachdem bisher lediglich die abstrakte und konkrete Syntax der Sprache vorgestellt wurden, werden im folgenden Abschnitt die Kontextbedingungen zur Konsistenzsicherung aus [HHK+15] kurz wiederholt.

6.2.2 Kontextbedingungen

Nachdem zuvor die Sprache und ihre konkrete sowie abstrakte Syntax vorgestellt wurden, werden in diesem Kapitel Kontextbedingungen vorgestellt, die die Semantik der Sprache definieren. Dazu wurden in [HHK+15] bereits acht Kontextbedingungen vorge-

stellt, welche auch für die handgeschriebene Erweiterung der Deltasprache gültig sind. Diese Kontextbedingungen stellen die Konsistenz sicher und fordern, dass referenzierte Elemente existieren und eine Operation im gegebenen Kontext anwendbar sein muss. So kann beispielsweise keine Assoziation innerhalb einer Klasse hinzugefügt werden. Dies ist durch die methodische Ableitung der Sprache nicht gesichert und wäre syntaktisch korrekt. Zudem erfordern die Kontextbedingungen, dass Elemente nicht entfernt werden können, wenn sie das letzte Element an einer Stelle sind, wo mindestens ein Element der Grammatik zufolge erforderlich ist. Darüber hinaus darf ein hinzuzufügendes Element nicht bereits existieren, wohingegen ein zu entfernendes Element existieren muss. Gleichzeitig sind auch die Kontextbedingungen der weiterführenden Operationen implizit, da sie alle durch Basisoperationen ausgedrückt werden können und somit bereits Kontextbedingungen zur Konsistenzsicherung existieren, gegeben. Diese werden hier nicht mehr zusätzlich aufgeführt.

6.3 Zusammenfassung

Abschließend wurde in diesem Kapitel eine klassendiagrammspezifische Deltasprache zur Modellierung der Domänenmodellevolution vorgestellt. Dazu wurden die Basisoperationen und weiterführende Operationen sowie die Kontextbedingungen der Sprache gezeigt. Während die vorangegangenen Sprachen die Entwicklung von Enterprise Applikationen unterstützen, unterstützt diese Sprache die Wartung und Weiterentwicklung des Systems. Nachdem in den Kapiteln 4 und 5 die Sprachen zur Modellierung der Persistenz und der Kommunikation von Enterprise Applikationen vorgestellt wurden, werden in den folgenden Kapiteln 7 und 8 die entwickelten Generatoren vorgestellt, die das System letztendlich generieren. Dabei wird gezeigt werden, wie Entitäten, Datenbankzugriffsfassaden, Datenbankschema, Clientzugriffsfassaden und Datentransferobjekte generiert werden. Zudem wird die generierte Schnittstelle zur manuell zu implementierenden Anwendungslogik vorgestellt. Auch die Ausführung der modellierten Domänenmodellevolution und der Datenmigration wird in Kapitel 9 vorgestellt.

Teil III

MontiEE-Generatoren

Kapitel 7

Generierung der Persistenz

Nachdem in den Kapiteln 4, 5 und 6 die Sprachen der MontiEE-Sprachfamilie präsentiert wurden, werden in den nachfolgenden Kapiteln die umgesetzten Generatoren und die benötigte Infrastruktur, die vom Produktentwickler genutzt werden, vorgestellt. Die Generatoren verwenden Modelle der zuvor vorgestellten Modellierungssprachen und erzeugen daraus Teile der Enterprise Applikation. Mit Hilfe der Generatoren wird technologiespezifischer Quellcode erzeugt. Die Anwendungslogik kann durch handgeschriebenen Java-Code ergänzt werden. Dazu wird der Quellcode der Enterprise Applikation, der die typische Architektur von Enterprise Applikationen, wie sie in Abbildung 3.1 dargestellt wurde, umsetzt, generiert (vgl. FA6-AG und FA1-PE). Es werden Kommunikationsfassaden, die es unterschiedlichen Clients des Client Tiers erlauben, mit dem JavaEE-Server zu kommunizieren, erzeugt (vgl. FA7-AG). Innerhalb dieser Kommunikationsfassaden des Web Tiers findet eine Autorisierung der Clients und eine rollenbasierte Zugriffskontrolle statt. Die Clients verwenden eigene, ebenfalls generierte Domänenmodelle und kommunizieren unterschiedliche Operationen an den Server. Ist der Client autorisiert eine Operation auszuführen, wird eine ebenfalls generierte Schnittstelle zu einer handgeschriebenen Geschäftslogik des Business Tiers aufgerufen, so dass die implementierte Geschäftslogik ausgeführt wird. Dieser Geschäftslogik steht eine generierte Schnittstelle zur Persistenz zur Verfügung. Über diese Schnittstelle wird mit einer generierten Kommunikationsfassade zum Datenbankserver innerhalb des EIS Tiers kommuniziert. Dabei werden generierte Entitäten kommuniziert. Zusätzlich wird die Datenbank des Datenbankservers mit Hilfe eines ebenfalls generierten SQL-Skripts initialisiert. Die Verwendung des Java-basierten Technologiestacks ist exemplarisch zur Anwendbarkeit von MontiEE gewählt worden. Für andere Technologien müssen einzelne Generatoren ausgetauscht werden. Aus diesem Grund wurden die einzelnen Generatoren so geschnitten, dass sie logische Komponenten, die einzeln wiederverwendbar oder erweiterbar sind, bilden. MontiEE verwendet die Java Enterprise Edition mit einem Glassfish Applikationsserver (vgl. FA13-PE), Hibernate als ORM (vgl. FA1-WE und FA2-WE) und PostgreSQL als Datenbankserver (vgl. FA3-WE). Zur Kommunikation mit Clients werden RPC und Webservices (vgl. FA4-WE) verwendet.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Generierung einer Infrastruktur für Generatoren zur Berücksichtigung technologiespezifischer Informationen auf Basis eines Tagschemas.
- Mechanismen zur Befüllung dieser Infrastruktur auf Basis einer Tagdefinition

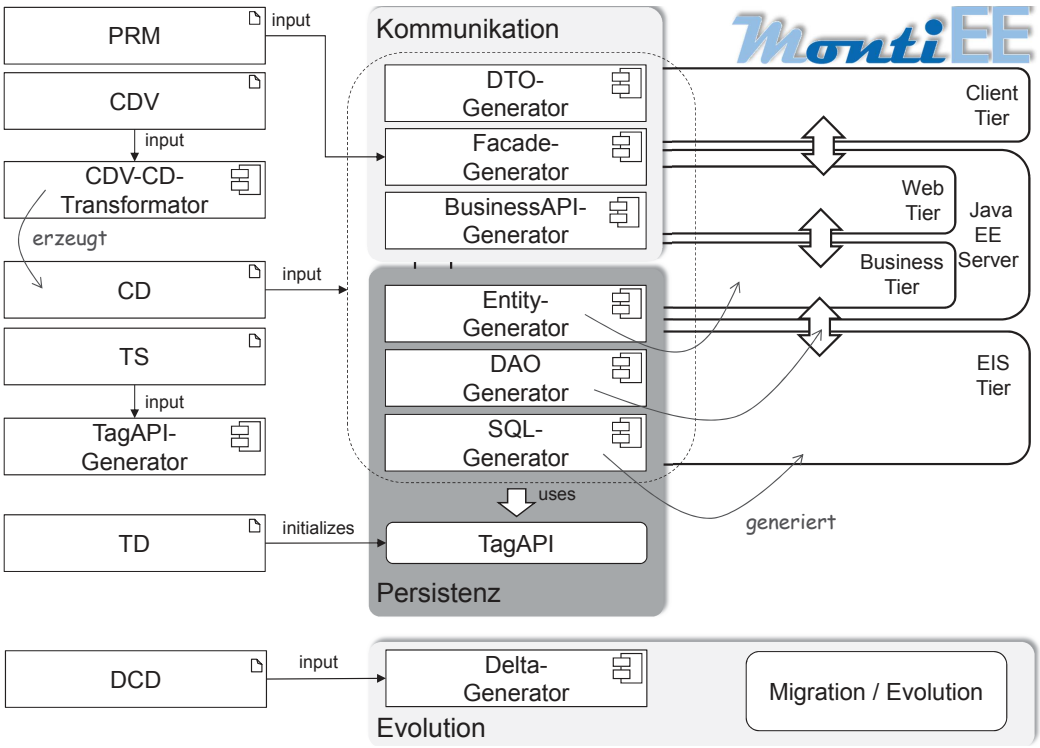


Abbildung 7.1: Übersicht der in MontiEE umgesetzten Generatoren. Gezeigt sind die Eingabemodelle sowie die Teile der Enterprise Applikation die mit Hilfe der Generatoren generiert werden. Der Fokus liegt dabei auf den Generatoren zur Generierung der Persistenz von Enterprise Applikationen.

- Generatoren für benötigte Teile der persistenten Datenhaltung, wie Entitäten, DAOs und Datenbankinitialisierungen.
- Kontextbedingungen zur Sicherstellung der Generierbarkeit.

Der Rest des Kapitels ist wie folgt strukturiert: Zunächst werden in Abschnitt 7.2 ein MontiEE spezifisches Tagschema gefolgt von der Infrastruktur für Tags in Abschnitt 7.3 vorgestellt. Daran anschließend werden die einzelnen Generatoren erläutert

Anschließend wird in Kapitel 8 die Generierung der Kommunikationsinfrastruktur von Enterprise Applikationen gefolgt von der Generierung der Infrastruktur zur Evolution des Systems in Kapitel 9 vorgestellt. In Kapitel 10 wird die Methodik der Verwendung der Generatoren, deren Konfiguration und die Umsetzung des Szenarios vorgestellt.

7.1 Überblick

Abbildung 7.1 zeigt eine Übersicht über die beteiligten Generatoren, die Modelle, die von den Generatoren verwendet werden, und die Teile der Enterprise Applikationen, die

von MontiEE generiert werden.

Im Wesentlichen existieren sechs Generatoren, die das Zielsystem generieren. Hinzu kommt der CDV-CD-Transformator, der die Umsetzung der in Abschnitt 5.2.3 vorgestellten Transformation einer Sicht in ein Klassendiagramm darstellt. Zudem generiert der TagAPI-Generator, der in Abschnitt 7.3 vorgestellt wird, einen Teil der benötigten Generierungsinfrastruktur. Außerdem generiert der Delta-Generator Infrastruktur zur Systemevolution und zur Datenmigration.

Im Rahmen dieses Kapitels liegt der Fokus auf der Generierung der Persistenz von Enterprise Applikationen. Dazu werden der Entity-, der DAO- und der SQL-Generator in den Abschnitten 7.4, 7.5 und 7.6 vorgestellt. Zudem wird der TagAPI-Generator vorgestellt, der die benötigte Generatorinfrastruktur erzeugt. Daneben erzeugt der CDV-CD-Transformator aus Sichten valide Klassendiagramme. Er folgt dem in Abschnitt 5.2.3 beschriebenen Prozess und wird hier nicht weiter erläutert. Generell verwenden Generatoren stets valide Klassendiagramme. Somit werden zur Generierung nicht direkt Sichten verwendet. Dies erhöht die Wiederverwendbarkeit der Generatoren.

Von den sechs Generatoren zur Erzeugung des Zielsystems verwenden alle, bis auf den SQL-Generator, Java als Zielsprache. Der SQL-Generator generiert valides SQL, indem er SQL-Skripte erzeugt. Alle Generatoren verwenden im Hintergrund eine Konfigurationsdatei, die Standardverhalten und -namen vorgibt. Dabei handelt es sich um Konfigurationen, die global für alle Generatoren gleich sind und einmalig für ein neues Projekt festgelegt werden. Die genaue Funktionsweise der Konfigurationen und die Umkonfiguration werden in Kapitel 10 dargestellt.

Technologiespezifische Anreicherungen des Domänenmodells werden in einer Tagdefinition modelliert. Diese Tagdefinition wird in einem vorgelagerten Schritt dazu verwendet, eine Infrastruktur zu befüllen, die auf Basis des Tagschemas generiert wurde. Dadurch stehen den Generatoren die Informationen der Tagdefinition zur Verfügung. Die TagAPI wird mit Hilfe des TagAPI-Generators auf Basis des Tagschemas generiert. Ein beispielhaftes, für MontiEE-spezifisches Tagschema wird in Abschnitt 7.2 vorgestellt. Die Befüllung der TagAPI wird in Abschnitt 7.3 gezeigt. Der Generierungsprozess, die generierte Infrastruktur und deren API wird dort ebenfalls detailliert präsentiert.

Der Entity-Generator, als zentraler Generator, verwendet Klassendiagramme sowie die aus dem Tagschema generierte und durch die Tagdefinition befüllte TagAPI zur Generierung der in Java umgesetzten Entitäten. Er generiert somit einen Teil des Business Tiers und wird in Abschnitt 7.4 detailliert vorgestellt.

Der DAO-Generator hat die Aufgabe, die DAOs der Applikation zu generieren. Dazu benötigt er Klassendiagramme sowie die TagAPI. Die generierten DAOs beinhalten CRUD-Operation für das Laden und Speichern von Entitäten sowie Möglichkeiten zur Ausführung von Queries. Dadurch stellen sie die Schnittstelle zwischen dem Applikationsserver und dem Datenbankserver dar. Diese Schnittstelle wird vom DAO-Generator erzeugt und in Abschnitt 7.5 detailliert vorgestellt.

Der SQL-Generator erhält die gleiche Eingabe wie der Entity-Generator, erzeugt aber ein SQL-Skript zur Initialisierung der Datenbank, des Datenbankschemas und der Tabellen des Schemas sowie ein Skript zur Entfernung dieser. Er generiert somit einen Teil

des EIS Tiers und wird in Abschnitt 7.6 detailliert vorgestellt.

Die weiteren Generatoren dienen der Generierung der Kommunikations- und der Evolutionsinfrastruktur von Enterprise Applikationen. Für die Generierung der Kommunikationsinfrastruktur werden in Kapitel 8 der DTO-, der Facade- und der BusinessAPI-Generator vorgestellt. In Kapitel 9 wird der Delta-Generator vorgestellt, der Infrastruktur für die Systemevolution und die Datenmigration generiert.

Innerhalb des Entwicklungsprozesses werden die Generatoren, die von MontiEE bereitgestellt werden, zu unterschiedlichen Zeitpunkten verwendet. Werden Generatoren verwendet, wird die Erstellung der Modelle, also die Modellierung, zunächst ausgeführt. Dann muss das Generat vor der Kompilierung erzeugt werden. Der CDV-CD-Transformator erzeugt aber aus Modellen neue Modelle, so dass dieser vor der Kompilierung ausgeführt werden muss. Ebenso erzeugt der TagAPI-Generator Infrastruktur, die von den weiteren Generatoren verwendet wird. Daher muss zunächst der TagAPI-Generator verwendet werden, um die Infrastruktur, die dann kompiliert werden muss, zu erzeugen. Anschließend kann der CDV-CD-Transformator, um die Sichten in Klassendiagramme zu transformieren, verwendet werden. Daran anschließend kann die TagAPI durch eine Tagdefinition befüllt werden. Dadurch können die einzelnen Generatoren die TagAPI verwenden und das Generat erzeugen. Dies kann kompiliert und die Enterprise Applikation kann als Datei gebündelt werden. Eine detaillierte Erläuterung der Konfiguration der Generatoren findet in Kapitel 10 statt.

Abbildung 7.2 zeigt einen exemplarischen Ablauf der Verwendung des Generats auf Basis des Szenarios des sozialen Netzwerks. Der dargestellte Akteur ist die Geschäftslogik, die eine neue Person anlegt und speichern möchte. Im unteren Bereich der Abbildung ist der Zustand der Datenbank, bevor die Person eingefügt wird, gezeigt. Die `Persons` Tabelle ist leer, die `IDValues` Tabelle enthält einen Eintrag. Die Generierung und Semantik dieser Tabellen wird in Abschnitt 7.6 vorgestellt. Die Geschäftslogik setzt den Namen der Person und ruft die `storePerson(p)` Methode des `PersonDAO` auf. Dieses speichert das Objekt `p` in der Datenbank und gibt ein neues Objekt `p'`, welches nun auch eine ID enthält, an die Geschäftslogik zurück. Der Grund, dass ein neues Objekt durch den ORM angelegt wird, liegt darin, dass etwaige vorhandene Objektreferenzen auf das ursprüngliche Objekt nach dem Speichern zur Umsetzung des Transaktionsmanagements und von Cachingmechanismen invalidiert werden. Dies bedeutet aber gleichzeitig auch, dass Referenzen, die andere Objekte, wie Manager oder weitere beteiligte Objekte halten, nicht erlaubt sind oder aktualisiert werden müssen. Dies erhöht die Komplexität der Handhabung. Der neue Zustand der Datenbank ist im unteren Abschnitt von Abbildung 7.2 gezeigt. Die Tabelle `Persons` enthält die gespeicherte Person und der Wert der `pk-ColumnValue` Spalte der `IDValues` Tabelle wurde erhöht. Abbildung 7.2 zeigt zudem, welche Klassen der dargestellten Objekte generiert werden. Die einzelnen Generatoren werden nachfolgend vorgestellt.

Zunächst wird ein MontiEE spezifisches Tagschema in Abschnitt 7.2 vorgestellt. Dieses Tagschema definiert mögliche Tagtypen, die dazu verwendet werden können, Modelle mit weiteren Informationen anzureichern. Die definierten Tagtypen repräsentieren die von den MontiEE Generatoren benötigten Informationen zur Generierung der Enterprise

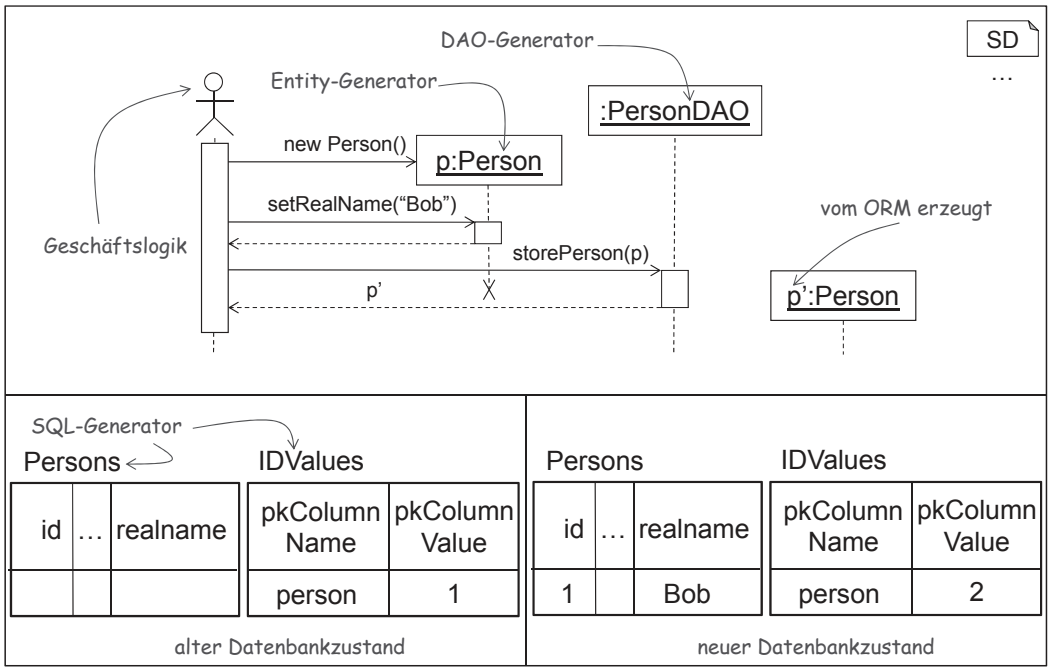


Abbildung 7.2: Darstellung eines exemplarischen Ablaufs der Speicherung eines Objekts mit Hilfe des MontiEE-Generats. Darüber hinaus sind die verantwortlichen Codegeneratoren abgebildet.

Applikation. Zudem wird die Generierung der TagAPI und die Befüllung dieser vorgestellt. Diese API wird von den Generatoren zur Abfrage der modellierten Informationen verwendet.

7.2 Die MontiEE Tagtypen

Nachdem in den vorangegangenen Abschnitten die Sprachfamilie MontiEE vorgestellt wurde, wird in diesem Abschnitt das vollständige MontiEE-Tagschema mit allen zur Verfügung stehenden Tagtypen für Klassendiagramme im Kontext von MontiEE vorgestellt. Die Tagtypen werden vom Werkzeugentwickler definiert und vom Produktentwickler in einer Tagdefinition genutzt. Dies erlaubt es dem Werkzeugentwickler mögliche Informationen, die die Generatoren verarbeiten können, zu definieren. Der Produktnutzer erhält gleichzeitig ein Schema, welches ihm die unterschiedlichen Möglichkeiten vorgibt. In [Mar11] wurden zunächst Stereotypen zur Anreicherung des Klassendiagramms verwendet und im Rahmen dieser Arbeit durch geeignete Tags ersetzt. Die vorgestellten Tagtypen basieren auf den Stereotypen aus [Mar11]. Sie resultieren aus der Analyse gängiger Operationen und benötigter Informationen gängiger ORMs, wie Hibernate. Sie werden dazu verwendet, um Modelle mit ORM spezifischen Informationen anzureichern. Aus diesem Grund sind die geschaffenen Tagtypen sehr nah an den von JPA verwendeten

Java-Annotationen. Die MontiEE Generatoren verwenden Tagdefinitionen, die konform zu dem hier vorgestellten Tagschema sind, zur Generierung von Java-Annotationen, die wiederum vom ORM verwendet werden.

Nachfolgend sind die einzelnen Tagtypen des Tagschemas der MontiEE Generatoren gezeigt. Auf Tagdefinitionsebene markieren Tags modellierte Elemente und ordnen ihnen Eigenschaften zu. Die Auswirkungen der einzelnen Tagtypen auf das Generat werden in den nachfolgenden Abschnitten beschrieben. Wie zuvor bereits beschrieben, wurde sich an dieser Stelle gegen die Verwendung von Stereotypen entschieden, da sie das eigentliche, logische und abstrahierte Modell mit technologiespezifischen Informationen verunreinigen. Der JPA Spezifikation [JPA16] folgend (vgl. FA1-WE), werden Elemente, die persistent gespeichert werden, als Entitäten bezeichnet. Dazu werden sie innerhalb der Tagdefinition getaggt (vgl. FA4-PE). Die Semantik der hier vorgestellten Tagtypen folgt dabei der Semantik des ORMs. Dies bedeutet, dass die Annotationen, die auf Basis der vorgestellten Tagtypen generiert werden bei dem ORM die hier vorgestellte Semantik bewirken.

```
1
2  tagtype Entity for CDClass;                                    TS
                                                                  ...
```

Listing 7.3: Definition des `Entity` Tagtyps. Er darf für Elemente des Typs `CDClass` verwendet werden.

Listing 7.3 zeigt den `Entity` Tagtyp. Dieser Tagtyp bezieht sich auf Elemente vom Typ `CDClass`. Wie bereits zuvor in Abschnitt 4.3 erläutert, bezieht sich die Angabe der Elemente, die mit einem Tag getaggt werden können, auf die Elemente der Grammatik der Klassendiagrammsprache. Der Tag selbst kann dazu verwendet werden, modellierte Klassen als Entitäten, um sie im laufenden System in einer Datenbank mit Hilfe des ORMs persistieren zu können (vgl. FA2-PE), auszuzeichnen.

```
1  tagtype Fetch:["eager"|"lazy"] for CDAssociation,              TS
2      CDAssociation!lefthand, CDAssociation!righthand;           ...
```

Listing 7.4: Definition des `Fetch` Tagtyps. Er darf für Elemente des Typs `CDAssociation` sowie linke oder rechte Seite der Assoziation verwendet werden. Seine möglichen Werte sind `eager` oder `lazy`

Listing 7.4 zeigt den `Fetch` Tagtyp. Er besitzt zwei verschiedene Optionen: `eager` und `lazy`. Er markiert eine ganze Assoziation, linke oder rechte Seite der Assoziation und beschreibt die Ladestrategie der assoziierten Elemente aus der Datenbank. Bei `eager` werden stets alle assoziierten Objekte, sobald ein Element der markierten Quellklasse der Assoziation geladen wird, ebenfalls geladen. Bei `lazy` hingegen werden diese erst nachgeladen, wenn ein Zugriff auf sie erfolgt. Dies ist besonders hilfreich, falls eine

große Anzahl oder besonders datenintensive, assoziierte Elemente erwartet werden. Dieser Tagtyp hat, wie in Kapitel 7 beschrieben, Auswirkungen auf den Entity- und den DTO-Generator. Zusätzlich beeinflusst er, wie in Abschnitt 5.3.4 genauer erläutert wird, die generierten Rechte.

TS

```
1  tagtype Cascading for CDAssociation,                                    ...
2    CDAssociation!lefthand, CDAssociation!righthand {
3      cascade:Cascade+;
4  }
5
6  inner tagtype
7    Cascade:["all"|"none"|"merge"|"persist"|"refresh"
8             |"remove"];
```

Listing 7.5: Definition des `Cascading` Tagtyps. Er darf für Elemente des Typs `CDAssociation` sowie linke oder rechte Seite der Assoziation verwendet werden. Er kann aus mehreren inneren `Cascade` Tagtypen, dessen mögliche Werte `all`, `none`, `merge`, `persist`, `refresh` oder `remove` sind, bestehen.

Listing 7.5 zeigt den `Cascading` und den inneren `Cascade` Tagtypen. Der `Cascading` Tagtyp beinhaltet dabei beliebig viele `Cascade` Tagtypen und bedient sich der Möglichkeit der Verschachtelung von Tagtypen. Dies ist notwendig, da die Kaskadierung aus mehreren Optionen bestehen kann. Ferner markiert der `Cascading` Tagtyp, analog zum `Fetch` Tagtyp, entweder eine ganze Assoziation, linke oder rechte Seite der Assoziation. Der innere `Cascade` Tagtyp modelliert die eigentliche Kaskadierungsoption. Dazu besitzt er sechs verschiedene Optionen: `all`, `none`, `merge`, `persist`, `refresh` und `remove`, welche das Verhalten der Kaskadierung des markierten Typs beschreiben. Kaskadierung bedeutet, dass eine Datenbankoperation wie `persist`, die auf einer Instanz ausgeführt wird, auch auf assoziierte Instanzen angewendet wird. Gleiches gilt für die Operationen `merge`, `refresh` und `remove`. Die `persist` Operation steht für ein Speichern eines neuen Objekts, die `merge` Operation steht für eine Aktualisierung oder bei einem Nichtvorhandensein des Objekts eine Speicherung des Objekts. Die Operation `refresh` steht für eine Synchronisation des Objekts mit dem Datenbankzustand und die Operation `remove` steht für ein Löschen des Objekts. Zusätzlich kann mit der Operation `all` modelliert werden, dass alle Operationen kaskadieren. Die Operation `none` bewirkt, dass keine Operationen kaskadiert werden. Die verschiedenen Optionen des Tagtyps können syntaktisch beliebig kombiniert werden. Die aggregierenden Optionen `all` und `none` stellen, da sie nicht kombiniert werden können, Ausnahmen dar. Die Verwendung des `Cascading` Tagtyps beeinflusst den Entity- sowie den DTO-Generator.

Listing 7.6 zeigt den `Inheritance` Tagtyp. Dieser Tagtyp erlaubt es, die Abbildung von Vererbung in einer Datenbank zu modellieren. Dabei erlaubt er die Optionen `singleTable`, `joined` und `tablePerClass` und kann modellierte Klassen auszeich-

```
1  tagtype Inheritance:["singleTable"|"joined"                    TS
2    |"tablePerClass"] for CDClass;                                ...
```

Listing 7.6: Definition des `Inheritance` Tagtyps. Er darf für Elemente des Typs `CDClass` verwendet werden. Seine möglichen Werte sind `singleTable`, `joined` oder `tablePerClass`

nen. Generell wird bei der Persistierung, wie in Kapitel 3 beschrieben, ein Objekt in seine primitiven Attribute zerlegt und auf Datenbanktabellen abgebildet. Objektreferenzen werden dabei mit Hilfe von Jointabellen gespeichert. Der Tagtyp dient dazu, festzulegen, in welcher Art Objekte innerhalb einer Vererbungshierarchie auf Datenbanktabellen abgebildet werden können, um die doppelte Persistierung von Information zu minimieren. Die erste in Listing 7.6 dargestellte Option, `singleTable`, modelliert, dass alle Objekte einer Vererbungshierarchie in einer einzelnen Datenbanktabelle abgelegt werden. Dabei wird jedes Attribut der Klassen der Vererbungshierarchie in einer eigenen Spalte der Datenbanktabelle persistiert. Zusätzlich wird eine Diskriminatorspalte, die den konkreten Typ des gespeicherten Elements als Wert des Diskriminators enthält, benötigt. Dieser wird dazu benutzt, festzuhalten, zu welcher Klasse der persistierte Eintrag gehört. Bei der Persistierung eines Objekts wird der Typ des Objekts in die Diskriminatorspalte persistiert. Die Attributwerte werden in den jeweiligen zugehörigen Spalten gespeichert. Da die Tabelle die Vereinigung aller Attribute aller Klassen der Vererbungshierarchie enthält, führt dies dazu, dass nicht alle Spalten mit einem Attributwert belegt werden können. Diese werden mit `null` befüllt. Die zweite in Listing 7.6 dargestellte Option, `joined` bewirkt dass für jede Klasse der Vererbungshierarchie, auch für abstrakte Superklassen, eine eigene Tabelle erzeugt wird. Die Attribute der jeweiligen Klassen werden in den entsprechenden Tabellen gespeichert und über die für alle Tabellen gleichbleibenden IDs gejoined. Die dritte in Listing 7.6 dargestellte Option, `tablePerClass`, führt dazu, dass für jeden konkreten Typ innerhalb einer Vererbungshierarchie eine eigene Datenbanktabelle angelegt wird. Dies verringert die Anzahl der mit `null` befüllten Spalten, wie bei der `singleTable` Option, und minimiert die Anzahl der notwendigen Joins, wie bei der `joined` Option, verstößt dafür aber unter Umständen gegen die Eigenschaft der dritten Datenbanknormalform [KE11], da Informationen in den Tabellen der Subtypen doppelt abgelegt sein können und nicht mittels eines Joins aufgelöst werden. Dieser Tagtyp beeinflusst den Entity- und den SQL-Generator.

```
1  tagtype IDGen:["sequence"|"auto"|"identity"|"none"|"table"]    TS
2      for CDClass;                                               ...
```

Listing 7.7: Definition des `IDGen` Tagtyps. Er darf für Elemente des Typs `CDClass` verwendet werden. Seine möglichen Werte sind `sequence`, `auto`, `identity`, `none` oder `table`.

Listing 7.7 zeigt den `IDGen` Tagtyp. Er besitzt die Optionen `sequence`, `auto`, `identity`, `none` und `table`. Er ermöglicht die Modellierung der Art und Weise, auf die der Primärschlüssel für die zu persistierende Instanz gebildet wird. Die Option `identity` spezifiziert dabei, dass der Primärschlüssel, der beim Speichern und Laden eindeutig bleibt, auf Basis der Objektidentität berechnet wird. Die in Listing 7.7 gezeigte Option `table` geht von der Existenz einer Datenbanktabelle aus, in der mit Hilfe eines Schlüssel-Wert-Paares die jeweiligen aktuellen IDs gespeichert sind und auf deren Basis eine nachfolgende ID berechnet werden kann. Die in Listing 7.7 gezeigte `auto` Option gibt an, dass ein interner Sequenzgenerator verwendet wird, der datenbankweite eindeutige IDs für Objekte erzeugt. Die in Listing 7.7 gezeigte Option `sequence` gibt an, dass der Primärschlüssel als fortlaufende Nummer der persistierten Objekte gebildet wird. Dabei wird keine Tabelle, sondern ein Sequenzgenerator verwendet. Die `auto` und die `identity` Strategien sind im Wesentlichen ähnlich, da auch der interne Sequenzgenerator die Objektidentität verwendet. Der Unterschied liegt darin, dass die `auto` Strategie konfiguriert werden kann, wohingegen die `identity` Strategie die Verwendung der Objektidentität festlegt. Ebenso sind die `table` und die `sequence` Strategie ähnlich. Allerdings trifft JPA keine Annahme über die verwendete Datenbank und das verwendete Datenbankparadigma, so dass nicht immer Tabellen zur Verfügung stehen. Die `identity` Strategie erstellt IDs, die innerhalb eines Typs eindeutig sind, wobei unterschiedliche Typen gleiche IDs haben können. Die `sequence` und die `table` Strategien garantieren keine fortlaufenden IDs. Dieser Tagtyp beeinflusst den Entity- sowie den SQL-Generator.

```
1  tagtype Owner for CDAssociation!lefthand,                    TS
2                    CDAssociation!righthand;                    ...
```

Listing 7.8: Definition des `Owner` Tagtyps. Er darf für die linke oder die rechte Seite von Elementen des Typs `CDAssociation` verwendet werden.

Listing 7.8 zeigt den `Owner` Tagtyp. Er kennzeichnet ein Assoziationsende. Dabei kann er die linke oder aber die rechte Seite als "Eigentümer" der Assoziation auszeichnen. Dies ermöglicht die Handhabung von Bidirektionalität und bedeutet, dass eine Instanz der Eigentümerklasse dafür sorgt, dass die bidirektionale Beziehung aufrecht und konsistent gehalten wird. Aus diesem Grund muss keine manuelle Konsistenzsicherung erfolgen. Insbesondere müssen die zur Verfügung gestellten Getter- und Setter-Methoden nicht manuell aufgerufen werden. Dies geschieht automatisch. Dieser Tagtyp beeinflusst den Entity-Generator.

Listing 7.9 zeigt den `Unique` Tagtyp. Er zeichnet ein Attribut als einzigartig aus und legt dabei fest, dass der Wert des Attributs bezogen auf alle Instanzen der modellierten Klasse einzigartig innerhalb des Systems ist. Dabei bezieht sich diese Auszeichnung auf den Zeitpunkt der Persistierung der Instanz und gilt nicht zwingend für Objekte im Speicher. Ebenso kann mit diesem Tagtyp eine Seite einer Assoziation ausgezeichnet werden. Dies ist aber nur bei unidirektionalen Assoziationen mit Zielkardinalität `1`, mög-

```
1  tagtype Unique for CDAttribute,                                    TS
2      CDAssociation!lefthand, CDAssociation!righthand;                ...
```

Listing 7.9: Definition des `Unique` Tagtyps. Er darf für Elemente des Typs `CDAttribute` oder für linke oder rechte Seiten von Elementen des Typs `CDAssociation` verwendet werden.

lich da diese im entstehenden Quellcode als Attribut abgebildet werden und somit auch dies einzigartig sein kann. Dies wird durch eine Kontextbedingung gesichert. Somit kann dieser Tagtyp nur an der Zielseite der unidirektionalen Assoziation verwendet werden. Dieser Tagtyp beeinflusst den SQL-Generator.

```
1  tagtype NotNull for CDAttribute,                                   TS
2      CDAssociation!lefthand, CDAssociation!righthand;                ...
```

Listing 7.10: Definition des `NotNull` Tagtyps. Er darf für Elemente des Typs `CDAttribute` oder für linke oder rechte Seiten von Elementen des Typs `CDAssociation` verwendet werden.

Listing 7.10 zeigt den `NotNull` Tagtyp. Dieser kann bei Elementen des Typs `CDAssociation` verwendet werden. Er zeichnet ein Attribut dahingehend aus, dass es immer belegt sein muss und nicht `null` sein darf. Dabei bezieht sich diese Auszeichnung ebenfalls auf den Zeitpunkt der Persistierung der Instanz. Bei Assoziationen ist dies implizit durch die Kardinalität der Assoziation vorgegeben und muss nicht zusätzlich modelliert werden. Der `NotNull` Tagtyp beeinflusst den SQL-Generator.

```
1  tagtype Transient for CDAttribute,                                 TS
2      CDAssociation!lefthand, CDAssociation!righthand;                ...
```

Listing 7.11: Definition des `Transient` Tagtyps. Er darf für Elemente des Typs `CDAttribute` oder für linke oder rechte Seiten von Elementen des Typs `CDAssociation` verwendet werden.

Listing 7.11 zeigt den `Transient` Tagtyp. Dieser kann, wie auch der `Unique` Tagtyp zur Auszeichnung von Attributen oder der linken oder rechten Seite von Assoziation verwendet werden. Auch hier gilt, dass er nur das Ziel einer unidirektionalen Assoziation mit Zielkardinalität `1` auszeichnen kann, was durch eine Kontextbedingung gesichert ist. Er definiert dabei, dass ein entsprechendes Attribut nicht in der Datenbank gespeichert wird. Dies ist ein Unterschied zum Java-Schlüsselwort `transient`, welches angibt, dass ein Attribut nicht serialisiert und bei einer Datenübertragung nicht kommuniziert wird. Der `Transient` Tagtyp beeinflusst dabei den Entity- und den DTO-Generator.

```
1
2  tagtype Ordered for CDAssociation;                                TS ...
```

Listing 7.12: Definition des `Ordered` Tagtyps. Er darf für Elemente des Typs `CDAssociation` verwendet werden.

Listing 7.12 zeigt den `Ordered` Tagtyp. Mit diesem Tagtyp lässt sich eine Assoziation, wie aus der UML/P [Rum11, Rum12, Sch12] bekannt, als geordnet auszeichnen. Dabei wurde bewusst nicht der ebenfalls in der UML/P vorhandene Marker `{ordered}` verwendet, da sich die Tagtypen auf die technische Realisierung der Enterprise Applikation beziehen und das Domänenmodell mit technischen Zusatzinformationen anreichern. Der in der UML/P vorhandene Marker `{ordered}` hingegen bezieht sich auf die fachliche Semantik des Domänenmodells. Auch wenn der Übergang beider fließend ist, so wurde sich hier für eine Trennung beider entschieden. Dieser Tagtyp beeinflusst dabei den Entity-Generator.

```
1  tagtype Queries for CDClass {                                    TS ...
2    query:Query+;
3  }
4
5  inner tagtype Query {
6    name:String,
7    value:String;
8  }
```

Listing 7.13: Definition des `Queries` Tagtyps. Er darf für Elemente des Typs `CDClass` verwendet werden. Er kann aus mehreren inneren `Query` Tagtypen, die wiederum aus einem Namen und einem Wert bestehen.

Abschließend zeigt Listing 7.13 den zusammengesetzten `Queries` Tagtyp. Dieser Tagtyp kann Elemente vom Typ `CDClass` auszeichnen. Er besteht dabei aus mindestens einem innerem `Query` Tagtyp. Jeder `Query` Tagtyp besteht aus einem Namen und einem Wert. Er wird dazu verwendet, benutzerdefinierte Queries zu modellieren (vgl. FA17-PE). Ein Beispiel eines solchen benutzerdefinierten Queries ist in Listing 10.10 dargestellt. Der Wert eines Queries kann dabei auf unterschiedliche Art und Weise modelliert sein. Hierbei sind verschiedene Sprachen zur Modellierung denkbar. Es können spezifische, für das gewählte Datenbankparadigma passende Querysprachen, wie SQL [ISO03a, ISO03b] oder HQL [ML05] verwendet werden. Darüber hinaus ist auch eine Modellierung mit Hilfe der OCL [Rum11, Rum12] oder aber der Epsilon Language [KPP06], die Modell Queries erlaubt, denkbar. Im Rahmen dieser Arbeit werden die SQL sowie die HQL Variante verwendet. Die Verwendung von Querysprachen, die stärker das Modell in den Vordergrund stellen, ist in weiteren Ausbaustufen denkbar. Zudem verwendet die der Arbeit zugrunde liegende Implementierung nicht die MontiCore Mechanismen zur

Spracheinbettung, sondern verwendet als Datentyp zur Querymodellierung einen String, der anschließend mit Hilfe eines SQL bzw. HQL Parser validiert und verarbeitet wird. Auf Basis des Beginns des Strings wird entschieden, ob der SQL oder der HQL Parser zur Weiterverarbeitung des Queries verwendet wird.

Für den Parser wurde in [Mül12] eine MontiCore SQL-Grammatik, die auf dem SQL 2003 Standard [ISO03a, ISO03b] basiert, umgesetzt. Die Herausforderung bestand dabei darin, dass der SQL-Standard nicht LL(k) parsebar ist, so dass einzelne Teile der SQL Grammatik, wie die eingebettete Sprache Fortran [SC12], nicht umgesetzt wurden. Es wurden aber dennoch 81 SQL-Funktionen in 676 Produktionsregeln und 224 Interfaceproduktionen umgesetzt. Zudem wurden 21 Kontextbedingungen umgesetzt. Auf Basis dieser Grammatik konnte die NIST Test Suite [NIS16], eingeschränkt auf den Teil der keine eingebetteten Sprachen wie Fortran verwendet, erfolgreich umgesetzt werden. Bei diesen Testfällen handelt es sich um standardkonforme Testfälle, die ein Tool verarbeiten können muss, um sich standardkonform nennen zu dürfen.

Bei der entwickelten Sprache handelt es sich um eine Repräsentation der bekannten SQL [ISO03a, ISO03b] Sprache. Die entwickelte Sprache wurde dahingehend angepasst, dass sie sowohl SQL als auch HQL parsen kann (vgl. FA2-WE). Der größte Unterschied der beiden Sprachen liegt in der Semantik und nicht in der Syntax. Ein wesentlicher syntaktischer Unterschied ist, dass nach einer `SELECT` Anweisung kein $\star$ oder eine Menge von Spaltennamen folgt, sondern der Platzhalter `Object(x)` für ein spezifisches Objekt. Nach dem `FROM` Befehl folgt dann ein semantischer Unterschied. Hier werden keine Tabellennnamen angegeben, sondern Namen von Objekttypen. In der Implementierung setzt Hibernate als ORM diese Typen dann auf die entsprechenden Tabellen, in denen die Objekte persistiert sind, um. Auch innerhalb des `WHERE` Blocks werden keine Spaltennamen für join Anweisungen verwendet, sondern die Felder des zuvor benannten Objekts ”x” können mit Hilfe der gängigen Punktnotation adressiert werden. Parameter werden als Variablen mit einem vorangestellten Doppelpunkt, `:parameter`, dem Query übergeben. Die JPA unterscheidet zwischen zwei verschiedenen Arten von Queries: `NamedQueries` und `NamedNativeQueries`. Die durch den Tag definierten Queries haben alle einen Namen. Die Unterscheidung, ob es ein natives Query ist oder nicht, hängt davon ab, ob ein Query in der Zielsprache der Datenbank oder in der Querysprache des ORM modelliert wurde. Im Wesentlichen werden SQL-Queries als native Queries und HQL-Queries als nicht native Queries umgesetzt. Der `Queries` Tagtyp beeinflusst den Entity- und den DAO-Generator.

Tagtypen ermöglichen es, das Klassendiagramm mit technologiespezifischen Tags, die nicht direkt zur fachlichen Modellierung der Applikation benötigt werden, anzureichern. Sie ermöglichen die Modellierung von Persistenzinformationen und benutzerdefinierten Datenbankanfragen (vgl. FA4-PE und FA17-PE). Dazu stellen sie die technischen Informationen für die entwickelten Generatoren, die in Kapitel 7 vorgestellt werden, dar. Auf Basis der gerade vorgestellten Tagtypen lassen sich Kontextbedingungen, die nachfolgend vorgestellt werden, ableiten.

7.3 Generierung und Befüllung der Infrastruktur für Tags

Nachdem zuvor eine exemplarische Tagdefinition in Listing 10.10 und die MontiEE Tagtypen in Abschnitt 7.2 vorgestellt wurden, wird die Generierung und die Befüllung der TagAPI vorgestellt. Die TagAPI wird dazu verwendet, die in einer Tagdefinition modellierten technologiespezifischen Informationen den Generatoren zur Verfügung zu stellen. Für die Generatoren ist von Vorteil, dass sie, im Gegensatz zu unstrukturierten Freitextinformationen gegen eine festgelegte Infrastruktur umgesetzt werden können. Darüber hinaus bietet diese Infrastruktur eine starkgetypte API, die die Abfrage der Tags eines Modells ermöglicht, an. Dies erleichtert die Implementierung eines Codegenerators.

In einem ersten Schritt werden die schemaspezifischen Bestandteile der TagAPI auf Basis eines Tagschemas generiert. Der TagAPI-Generator erzeugt die schemaspezifischen Teile der TagAPI, die zur Laufzeit der MontiEE-Generatoren kompiliert zur Verfügung stehen muss, da die Generatoren, die das System erzeugen, auf die TagAPI angewiesen sind und diese verwenden. Im Buildprozess muss die Laufzeit des TagAPI-Generators vor der Laufzeit der MontiEE-Generatoren und vor dem Kompilieren des Generats liegen. Alternativ kann die Generierung der TagAPI auch losgelöst von der Entwicklung des Systems betrachtet werden. Die generierten Klassen können eigenständig released und von den MontiEE-Generatoren, solange sich das Tagschema nicht verändert, verwendet werden. Die MontiEE-Generatoren reagieren auf ein vorgefertigtes Tagschema, können aber konzeptionell auf mehrere Tagschemas reagieren. Es werden dann für jedes beteiligte Tagschema durch den TagAPI-Generator schemaspezifische Teile der Infrastruktur mit unterschiedlichen Namenspräfixen generiert.

In einem zweiten Schritt wird die TagAPI auf Basis einer konkreten Tagdefinition, wie in Abbildung 7.1 dargestellt, befüllt. Die Befüllung der TagAPI erfolgt direkt zu Beginn der Laufzeit der MontiEE Generatoren. Im weiteren Verlauf des Kapitels wird daher zwischen der Generierung und der Befüllung unterschieden. Zudem wird zwischen Klassen, die vom TagAPI-Generator oder von den MontiEE Generatoren generiert werden, unterschieden. Die hier vorgestellte Infrastruktur beruht auf der betreuten Vorarbeit [Roi15]. Alternativ wäre eine Interpretation der Tagdefinition zur Laufzeit des Generators denkbar gewesen. Gegen diese Lösung wurde sich aber aus Komfortgründen entschieden, da die Infrastruktur eine stark getypte API sowie Komfortfunktionen, die der AST nicht anbietet, ermöglicht.

7.3.1 Generierung der Infrastruktur

Die konzeptionelle Idee des Ansatzes ist, dass der Generatorentwickler, oder allgemein Werkzeugentwickler, ein schemaspezifisches Repository als Teil der Infrastruktur zur Verfügung gestellt bekommt. Dieses Repository bietet eine API an und ermöglicht es dem Generatorentwickler, unterschiedliche Abfragen an das Modell zu tätigen. So kann er abfragen, ob ein Element einen Tag besitzt, welche Elemente einen Tag haben oder aber welche Tags ein Element besitzt. Da mehrere Tagschemas verwendet werden können, werden diese in jeweils eigenen schemaspezifischen Repositories repräsentiert. Der Generatorentwickler kann dann die Repositories in seinen Templates eigenständig verwenden.

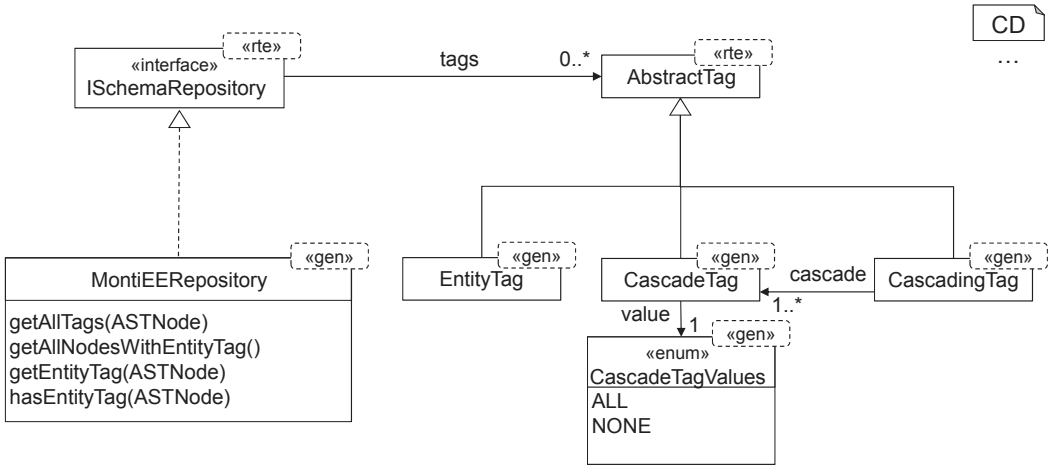


Abbildung 7.14: Darstellung der Klassen der generierten Infrastruktur und der Klassen der Laufzeitumgebung.

Abbildung 7.14 zeigt die vom TagAPI-Generator aus dem MontiEE-Tagschema, das in Abschnitt 7.2 vorgestellt wurde, generierte Infrastruktur sowie die von der Laufzeitumgebung bereitgestellte Klasse `AbstractTag`. Es wird generell für jeden Tagtyp des Tagschemas eine eigene Klasse vom TagAPI-Generator generiert, die von der Klasse `AbstractTag` erbt. Diese Klassen sind nicht die AST-Klassen, die von MontiCore generiert werden, sondern eine zum AST parallele Struktur. Der Grund des Aufbaus einer parallelen Struktur liegt in der generischen Eigenschaft der Sprache und der dadurch schlechteren Benutzbarkeit der AST-Klassen. Die AST-Klassen werden von MontiCore auf Basis der Sprachkonzepte generiert. Dies bedeutet, dass die vier verschiedenen Tagtypen: `SimpleTagType`, `ValuedTagType`, `EnumeratedTagType` und `ComplexTagType` als AST-Klassen von MontiCore generiert werden. Da jeder Tagtyp einen Namen, wie beispielsweise `Entity` besitzt, wird dieser im AST als Attribut dargestellt. Der Generatorentwickler müsste somit mit einer Instanz eines `ASTSimpleTagType` mit einem Attribut `Name`, welches den Wert `Entity` besitzt, arbeiten. Zur stärkeren Typisierung wurde sich im Rahmen dieser Arbeit dazu entschieden, dass die einzelnen im Tagschema definierten Tagtypen als jeweils eigenständige Repräsentationen in Form individuell generierter Klassen dem Generatorentwickler zur Verfügung gestellt werden. Dies bedeutet nicht, dass diese Klassen die AST-Klassen ersetzen, sondern sie werden auf Basis des geparsten Modells, also auf Basis der instanziierten AST-Klassen, generiert und später durch eine Tagdefinition befüllt.

Bei der Generierung dieser spezifischen Klassen der Tagtypen wird je nach Art des Tagtyps, `SimpleTagType`, `ValuedTagType`, `EnumeratedTagType` oder `ComplexTagType` eine leicht unterschiedliche Klasse generiert. Bei einem `SimpleTagType`, wie dem `EntityTag`, wird nur ein Klassengerüst ohne weitere Attribute erstellt, da der Tagtyp auch keine weiteren Informationen besitzt. Bei einem `EnumeratedTagTyp`, wie

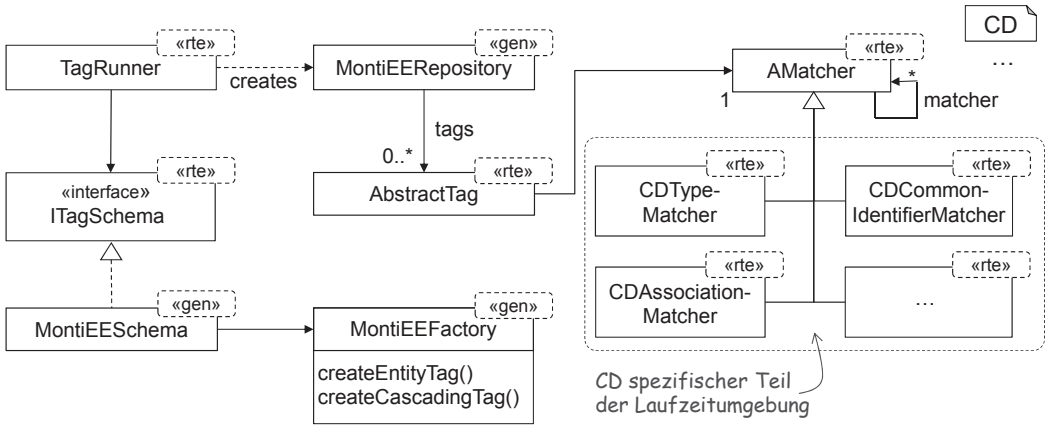


Abbildung 7.15: Exemplarische Darstellung der Zuordnung von AST-Knoten und Tag.

dem `CascadeTag`, wird zusätzlich zu der generierten Klasse ein `value` Attribut als Enumeration generiert. Die ebenfalls generierte Enumeration beinhaltet alle Elemente des Tagtyps. In Abbildung 7.14 ist dies durch die Enumeration `CascadeTagValues` exemplarisch für die Klasse `CascadeTag` gezeigt.

Für einen `ValuedTagType` wird analog ebenfalls eine Klasse mit einem `value` Attribut generiert, wobei diesmal das Attribut keine Enumeration ist, sondern den Typ besitzt, der im Tagschema modelliert wurde. Die Attribute der Klassen können über generierte Getter-Methoden zur Laufzeit der MontiEE-Generatoren verwendet werden. Methoden zum Setzen von Attributen werden nicht generiert, da das Repository eine Repräsentation der Tagdefinition ist und nach der Befüllung nicht verändert wird. Daher werden auch die Attribute stets als private Attribute generiert. Als letzte mögliche Art eines Tagtyps wird für einen `ComplexTagType` eine entsprechende Klasse, die eine Assoziation zu den enthaltenen Tagtypen besitzt, generiert. Die Zielkardinalität der Assoziation entspricht der im Tagschema modellierten Kardinalität. Sollte der komplexe Tagtyp zusätzlich zu verschachtelten Tagtypen Attribute besitzen, werden sie als Attribute der generierten Klasse wie bei einem `ValuedTagtype` mit generiert. Diese generierten Tagklassen stehen dem Generatorentwickler zur Verfügung und können in den Templates verwendet werden.

Neben der Klasse `AbstractTag` wird von der Laufzeitumgebung auch das Interface `ISchemaRepository` bereitgestellt. Dieses Interface dient dazu, die schemaspezifischen Repositories auszuzeichnen. Die schemaspezifischen Repositories werden für jedes modellierte Schema vom TagAPI-Generator generiert und müssen alle das `ISchemaRepository` Interface implementieren. In Abbildung 7.14 ist das schemaspezifische Repository `MontiEERepository` exemplarisch dargestellt. Es beinhaltet tagspezifische Methoden und muss nicht auf generische Methoden zurückgreifen. Es enthält die `getAllTags(ASTNode)` Methode, die alle Tags eines AST-Knotens des Modells zurückgibt. Zudem besitzen die sprachspezifischen Repositories für jeden Tagtyp drei spezifische

Methoden, die am Beispiel des `EntityTag` gezeigt werden. Die erste Methode, `getAllNodesWithEntityTag()`, gibt alle AST-Knoten, die mit dem `Entity` Tag getaggt wurden, zurück. Die zweite Methode, `getEntityTag(ASTNode)`, gibt den `Entity` Tag eines AST-Knotens zurück, sofern er einen solchen besitzt. Darüber hinaus steht die `hasEntityTag(ASTNode)` Methode zur Verfügung, die angibt, ob ein AST-Knoten einen Tag besitzt. Für den Generatorentwickler ist Typsicherheit von großer Bedeutung, so dass sich bei der Entwicklung eines Generators schemaspezifische Repositories mit starkgetypten Methoden gut verwenden lassen.

7.3.2 Befüllung der Infrastruktur

Nachdem die vom TagAPI-Generator generierten Klassen der Taginfrastruktur vorgestellt wurden, muss diese zur Laufzeit der MontiEE-Generatoren befüllt werden, so dass sie aus den Templates heraus verwendet werden kann. Dazu werden vor dem Start der MontiEE-Generatoren die Repositories auf Basis des Eingabemodells und der Tagdefinition befüllt. Es erfolgt dazu eine Instanziierung der vom TagAPI-Generator generierten Tagklassen und eine Zuordnung von Tag zu AST-Knoten des Klassendiagramms. Es werden also Eingabemodell und Tagdefinition geparst und die Referenzen auf das Eingabemodell aufgelöst, so dass der referenzierte AST-Knoten zur Laufzeit der MontiEE Generatoren zur Verfügung steht. Diese Zuordnung wird dann in der Datenstruktur `Map` geeignet gespeichert.

Abbildung 7.15 zeigt die Klassen, die zur Befüllung der Infrastruktur verwendet werden. Erneut dargestellt sind die Klassen `MontiEERepository` und `AbstractTag`, die in Abbildung 7.14 bereits detailliert erklärt wurden. Der von der Laufzeitumgebung zur Verfügung gestellte `TagRunner` wird von MontiCore gestartet. Der `TagRunner` dient dazu, dass `MontiEERepository` zu erstellen. Er ist in der Lage, mehrere Tagdefinitionen zu verarbeiten. Dabei können diese Tagdefinitionen konzeptionell zu unterschiedlichen Tagschemas konform sein. Zur Realisierung wird eine für jedes bekannte Schema vom TagAPI-Generator spezifisch generierte Klasse, die das Interface `ITagSchema` implementiert, verwendet. In Abbildung 7.15 ist dies die `MontiEESchema` Klasse. Die generierten Schemaklassen beinhalten benötigte Konfigurationsinformationen, wie beispielsweise die Zielsprache des Schemas oder Informationen zur Prüfung von Kontextbedingungen. Auf Basis dieser Informationen kann der `TagRunner` zur Laufzeit der MontiEE-Generatoren alle Tagdefinitionen, die zu dem Tagschema, das dem `TagRunner` über das Interface bekannt ist, konform sind, verarbeiten. Zusätzlich kann über die generierten Schemaklassen ebenfalls eine vom TagAPI-Generator generierte, schemaspezifische Factory verwendet werden. In Abbildung 7.15 ist dies durch die Klasse `MontiEEFactory` dargestellt.

Die Factory enthält Möglichkeiten zur Instanziierung der generierten Tagklassen. Der `TagRunner` parst zur Laufzeit der MontiEE Generatoren die Tagdefinitionen und verwendet die entsprechende Schemaklasse und die `MontiEEFactory` dazu, Instanzen der generierten Tagklassen auf Basis der modellierten Tags der Tagdefinition zu erstellen. Dies erzeugt die eigene interne Repräsentation der Tags auf Basis der nach dem Parsen entstehenden AST-Knoten der Tagdefinition. Jeder Tag besitzt verschiedene Matcher, die dazu dienen, die Referenzen zum Zielmodell aufzulösen. Dazu traversieren die Mat-

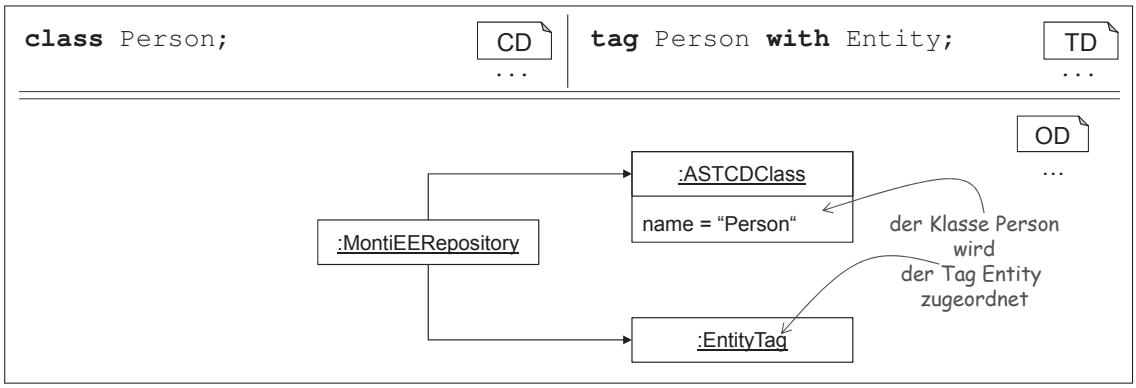


Abbildung 7.16: Objektdiagrammdarstellung des Zustands des Repositories nach der Zuordnung eines AST-Knotens zu einem Tag.

cher auf Basis der in der Tagdefinition modellierten Referenz den AST des Klassendiagramms und ordnen den aufgelösten AST-Knoten der Instanz der Tagklasse zu. Dafür stellt die Laufzeitumgebung die Klasse `AMatcher`, die hierarchische Matcher ermöglicht, bereit. Diese Klasse definiert die allgemeine Superklasse für benötigte sprachspezifische Matcher. Abbildung 7.16 zeigt exemplarisch die Funktionalität der Matcher. Für ein gegebenes Klassendiagramm, welches eine Klasse `Person` enthält und einer Tagdefinition, die die Klasse `Person` mit dem `Entity` Tag taggt, enthält das Repository nach der Befüllung eine Zuordnung zwischen dem AST-Knoten und einer instanziierten Klasse des Tagschemas. Ähnlich wie die Implementierung des `ModelElementIdentifier` Interfaces zur Umsetzung der sprachspezifischen Variante der Tagdefinitionssprache auf Ebene der Sprache, wie in Abschnitt 4.3 gezeigt, muss auch auf Implementierungsebene die Laufzeitumgebung um sprachspezifische Konzepte, wie die Matcher, erweitert werden. Manuell implementiert werden müssen daher sprachspezifische Matcher, wie der `CDTypeMatcher` oder der `CDAssociationMatcher`. Diese Matcher ermöglichen es, auf Basis der im Modell verwendeten Referenz den AST-Knoten eines geparsten Klassendiagramms aufzulösen. Dazu ordnen sie einem `ModelElementIdentifier` den zugehörigen AST-Knoten zu. Bei Klassen ist der `ModelElementIdentifier` beispielsweise durch einen Namen gegeben. Die Aufgabe des Matchers ist es, den entsprechenden AST-Knoten vom Typ `CDClass` zu finden, der ein Attribut `name` hat, dessen Wert dem des `ModelElementIdentifier` entspricht. Nach dem Auflösen der Referenzen können die AST-Knoten der internen Repräsentation der Tags zugeordnet werden. Ist diese Zuordnung für alle Elemente der Tagdefinition erfolgt und das `MontiEERepository` damit befüllt und initialisiert, kann es zur Laufzeit der MontiEE Generatoren verwendet werden. Der entsprechende Matcher wird durch Benutzung der Erweiterungsmechanismen des ASTs von MontiCore gefunden. Auf Ebene des ASTs muss die `createMatcher()` Methode mit Hilfe der Möglichkeit abgeleiteter Produktionen implementiert werden, die für einen Tagtyp des Schemas die möglichen Matcher zurückgibt. Der erste Matcher, der einen passenden AST-Knoten finden kann, wird zum Matching verwendet.

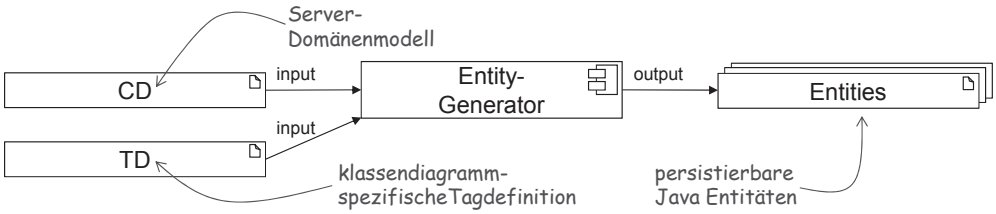


Abbildung 7.17: Darstellung der Eingabe- und Ausgabeartefakte des Entity-Generators. Er verwendet das Domänenmodell des Servers und erzeugt daraus viele Entitäten.

7.4 Der Entity-Generator

In diesem Abschnitt wird der Entity-Generator detailliert vorgestellt. Er verwendet Klassendiagramme und eine zu dem Klassendiagramm gehörige Tagdefinition als Eingabe. Die zum Klassendiagramm gehörige Tagdefinition muss zu dem in Abschnitt 7.2 vorgestellten MontiEE-Tagschema konform sein. Dadurch kann zu speichernde Information vom Produktentwickler abstrakt modelliert werden und in eine technische Implementierung transformiert werden.

Abbildung 7.17 gibt einen Überblick über den Entity-Generator. Er verwendet ein Klassendiagramm als Eingabe, um daraus JPA-konforme Entitäten (vgl. FA1-WE) zu generieren. Er verwendet das Server-Domänenmodell als Klassendiagramm und erzeugt daraus die Implementierung dessen als Java-Entitäten. Diese Entitäten, wie in Abschnitt 3.3 beschrieben, sind im Wesentlichen normale Java-Klassen, angereichert um Attribute zur Abbildung von Assoziationen. Entitäten können mit Hilfe des ORM in einer Datenbank persistent gespeichert werden. Daher enthalten sie Annotationen, die die Abbildung des Objekts auf die Relationen einer Datenbank definieren (vgl. FA3-WE).

Zur detaillierten Erläuterung des Generats werden die Konzepte der Eingabemodelle auf die Konzepte der Ausgabemodelle abgebildet. Dazu wird die Auswirkung der relevanten Konzepte von Klassendiagrammen sowie die Auswirkung jedes einzelnen Tagtyps des MontiEE-Tagschemas auf den Entity-Generator erläutert.

7.4.1 Abbildung der Modellierungskonzepte auf das Generat

In diesem Abschnitt werden die einzelnen Konzepte der Eingabemodelle, die den Entity-Generator beeinflussen, vorgestellt. Dazu werden zunächst die Elemente des Klassendiagramms gefolgt von einer detaillierten Erläuterung der Auswirkungen der einzelnen Tagtypen des MontiEE-Tagschemas, welches in Abschnitt 7.2 präsentiert wurde, betrachtet. Generell werden Annotationen zur Konfiguration und Steuerung der Persistenz im Java-Code generiert. Dabei können diese entweder an zu persistierende Attribute oder an zugehörige Setter-Methoden generiert werden. Dies wird in der JPA als `Field-` bzw. `Propertyaccess` bezeichnet. Der Unterschied liegt im technischen Zugriff des ORM auf die Attribute der Klasse beim Laden der Daten aus der Datenbank und der Instanziierung der Klasse. Im weiteren Verlauf wird immer der `Fieldaccess` zur Darstellung

verwendet, so dass Annotationen immer an den zugehörigen Attributen vorgestellt werden. MontiEE kann, wie in Kapitel 10 gezeigt wird, so konfiguriert werden, dass beide Arten verwendet werden können.

Auswirkungen der Klassendiagrammkonzepte

Zur Generierung der Java-Klassen wird im Wesentlichen den Generierungsvorschriften aus [Sch12] gefolgt, so dass diese hier nur kurz umrissen werden und für weitergehende Informationen auf [Sch12] verwiesen wird. Abweichungen davon werden vorgestellt.

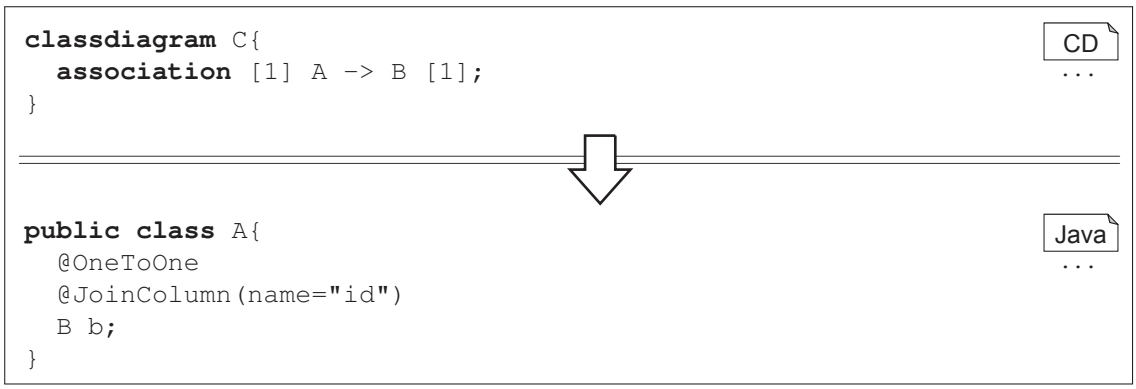


Abbildung 7.18: Abbildung einer unidirektionalen `1-zu-1` Assoziation im Generat des Entity-Generators.

Zunächst wird für jede modellierte Klasse eine Java-Klasse generiert. Diese Klasse beinhaltet alle modellierten Attribute. Die Attribute werden als private Attribute realisiert und ebenfalls generierte Getter- und Setter-Methoden erhalten die modellierte Sichtbarkeit des Attributs. Attribute einer Klasse benötigen im Standardfall keine Annotation, da der verwendete ORM eine vordefinierte Menge an Datentypen automatisch verarbeiten kann. Dazu gehören alle primitiven Datentypen oder `String`. Ebenso werden Enumerationen, auch wenn sie im Klassendiagramm über eine Assoziation modelliert werden, wie in Abbildung 4.2 die Enumeration `Tariff`, mit Hilfe einer Annotation gespeichert. Dazu erhält die beinhaltende Klasse ein Attribut, welches mit der Annotation `@Enumerated(EnumType.STRING)` versehen wird. Dadurch wird der ORM angewiesen, die einzelnen Enumerationswerte als Strings in der Datenbank zu speichern. Dies wird vom Entity-Generator ohne zusätzlichen Tag gemacht. Assoziationen werden als komplexe Attribute generiert. Dabei werden sie als einzelnes Attribut, als `Set`, `List` oder `Map` abhängig von der Kardinalität und der Art der Assoziation repräsentiert. Normalerweise wird der Datentyp `Set` verwendet. Bei einer geordneten Assoziation wird der Datentyp `List` verwendet. Die Verwendung des Tags `Ordered` bestimmt, dass eine Assoziation als eine geordnete Assoziation umgesetzt wird. Dies wird bei der Erklärung der Auswirkungen des Tags erläutert.

Darüber hinaus können Assoziation aber unterschiedliche Navigationsrichtungen und

Kardinalitäten, wie in Abschnitt 2.3 gezeigt wurde, besitzen. Für die unterschiedlichen Richtungen und Kardinalitäten werden jeweils andere Annotationen verwendet. Zudem werden Selbstassoziationen, da diese sowohl Quelle als auch Ziel innerhalb der gleichen Klasse besitzen, gesondert behandelt. Die Namensgebung wird so verändert wird, dass es zu keinen Konflikten kommt.

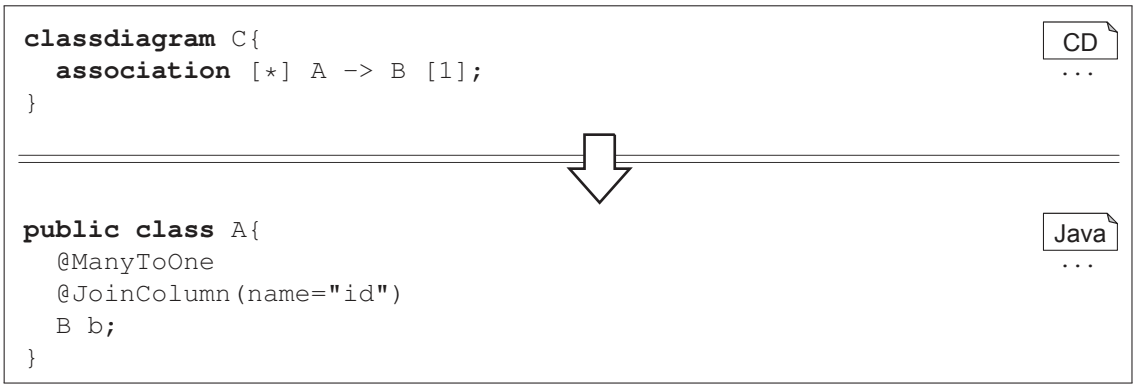


Abbildung 7.19: Abbildung einer unidirektionalen `*-zu-1` Assoziation im Generat des Entity-Generators.

Abbildung 7.18 zeigt die Abbildung einer unidirektionalen `1-zu-1` Assoziation. Die gezeigten Assoziationen werden durch den Entity-Generator an das Attribut der Klasse `A` geschrieben. Die Annotation `@OneToOne` kann verschiedene Parameter, wie Kaskadierungs- oder Ladeoptionen, die später wieder aufgegriffen werden, besitzen. Sie bestimmen, wie in Abschnitt 7.2 erklärt, die Kaskadierung und die Art des Ladens aus der Datenbank. Der Entity-Generator verwendet für diese Parameter entweder Standardwerte oder die `Cascading` oder `Fetch` Tags einer Tagdefinition. Er reagiert dazu entsprechend auf die möglichen Optionen der Tags `Cascade` und `Fetch`.

Die Annotation `@JoinColumn` gibt die Spalte, über die die Assoziation in der Datenbank aufgelöst wird, an. Sie bezieht sich auf die Spalte, in der das ID-Attribut der Klasse `B` gespeichert wird. Das generierte Attribut erhält den Datentyp `B`. Der SQL-Generator, der in Abschnitt 7.6 vorgestellt wird, muss diese Modellinformationen ebenfalls berücksichtigen. Der Name des Attributs der Klasse wird aus den Informationen der Assoziation abgeleitet. Es verwendet, falls vorhanden, den Rollennamen des Elements. Ist dieser nicht vorhanden, wird der Assoziationsname verwendet. Ist auch dieser nicht vorhanden, wird der Name der Klasse, wie in Abbildung 7.18, verwendet. Auch hier sei angemerkt, dass die ORMs vorschreiben, dass die Angabe der Spalten und Namen stets auf String Basis erfolgt und ohne einen geeigneten Codegenerator, da es keine statische Analyse dieser Zusammenhänge gibt, sehr fehleranfällig ist.

Abbildung 7.19 zeigt die generierten Annotationen einer unidirektionalen `*-zu-1` Assoziation. Die generierten Annotationen entsprechen den bereits in Abbildung 7.18 vorgestellten Annotationen, da die Zielkardinalität die benötigten Annotationen bestimmt und diese in beiden Fällen `1` entspricht. Auf Ebene der Datenbank werden Einschrän-

```
classdiagram C{                                                    CD
  association [1] A -> B [*];                                      ...
}
```

```
public class A{                                                    Java
  @OneToMany                                                       ...
  @JoinTable(name="abs", schema="c",
      joinColumns=
          @JoinColumn(name="a", referencedColumnName="id"),
      inverseJoinColumns=
          @JoinColumn(name="bs", referencedColumnName="id"))
  Set<B> bs;
}
```

Abbildung 7.20: Abbildung einer unidirektionalen `1-zu-*` Assoziation im Generat des Entity-Generators.

kungen bezüglich der Startkardinalität nicht umgesetzt. Gegenüber Abbildung 7.18 wird nicht mehr die Annotation `@OneToOne`, sondern die Annotation `@ManyToOne` verwendet. Abbildung 7.20 zeigt die Abbildung einer unidirektionalen `1-zu-*` Assoziation. Der generierte Code ist dabei dem in Abbildung 7.18 sehr ähnlich. Es wird die Annotation `@OneToMany`, die die veränderte Kardinalität abbildet, verwendet. Zudem wird keine Joinspalte, sondern eine Jointabelle verwendet. Konzeptionell beinhaltet eine Jointabelle dabei die Primärschlüssel der beiden beteiligten Elemente. Die Jointabelle selbst wird über den `name` Parameter der Annotation `@JoinTable` referenziert. Dabei setzt sich dieser Name aus der linken und der rechten Seite der Assoziation, wobei dem Namen, dessen zugehörige Assoziationsseite die Kardinalität `*` besitzt, ein "s" angehängt wird, zusammen. Der Parameter `joinColumns` beschreibt den Primärschlüssel der Klasse `A` über eine Menge von `@JoinColumn`. Eine solche Spalte wird über ihren Namen in der Jointabelle und ihren zugehörigen Namen in der Tabelle des eigentlichen Elements identifiziert. Das Gegenstück dazu stellt der Parameter `inverseJoinColums` dar, der den Primärschlüssel der Klasse `B` in der Jointabelle beschreibt und auf die eigentliche Tabelle für `B` verweist. Der SQL-Generator muss diese Modellinformationen ebenfalls berücksichtigen. Als Datentyp wird, da es sich nicht um eine geordnete Assoziation handelt, `Set` verwendet. Der Name des Attributs folgt dem gleichen Namensschema wie bei der Annotation aus Abbildung 7.18. Zusätzlich wird auch hier ein "s" angehängt.

Abbildung 7.21 zeigt die Abbildung einer unidirektionalen `*-zu-*` Assoziation. Der generierte Code ist, bis auf die Annotation und die Namensgebung, identisch mit dem generierten Code aus Abbildung 7.20. Der Name der Jointabelle und der Spaltenname erhalten ein angehängtes "s". Diese Annotationen werden für alle unidirektionalen Assoziationen generiert. Ist keine Navigationsrichtung ("–", vgl. Abschnitt 2.3.1) angegeben, wird als Standardnavigationsrichtung "–>" angenommen.

```
classdiagram C{
  association [*] A -> B [*];
}
```

CD
...

```
public class A{
  @ManyToMany
  @JoinTable(name="asbs", schema="c",
      joinColumns=
          @JoinColumn(name="as", referencedColumnName="id"),
      inverseJoinColumns=
          @JoinColumn(name="bs", referencedColumnName="id"))
  Set<B> bs;
}
```

Java
...

Abbildung 7.21: Abbildung einer unidirektionalen `*-zu-*` Assoziation im Generat des Entity-Generators.

```
classdiagram C{
  association [1] A <-> B [1];
}
```

CD
...

```
public class B{
  @OneToOne(mappedBy="b")
  A a;
}
```

Java
...

Abbildung 7.22: Abbildung der Rückrichtung einer bidirektionalen `1-zu-1` Assoziation im Generat des Entity-Generators. Die Richtung aus Sicht des `Owner` ist in Listing 7.18 dargestellt.

Die Annotationen für Assoziationen mit der Navigationsrichtung "<-" werden analog zur Navigationsrichtung "->" in der jeweiligen anderen Klasse generiert.

Eine Sonderbehandlung benötigen bidirektionale Assoziationen. Dort werden Annotationen in beide Klassen generiert. Generell kümmern sich der ORM und das Datenbanksystem um die Synchronisation einer bidirektionalen Assoziation. Damit dem ORM dies möglich ist, wird von der JPA gefordert, dass eine Seite der Assoziation die sogenannte `Owner` Seite der Assoziation darstellt. Dies bedeutet, dass dort die bidirektionale Assoziation initialisiert und immer wenn Setter-Methoden der `Owner` Assoziation verwendet werden, auch das Attribut der assoziierten Klasse gesetzt wird. In MontiEE modelliert der `Owner` Tag die analoge Information. Ist er nicht angegeben, wird immer die linke

```
classdiagram C{
  association [1] A <-> B [*];
}
```

CD

...

```
public class B{
  @ManyToOne(mappedBy="bs")
  A a;
}
```

Java

...

Abbildung 7.23: Abbildung der Rückrichtung einer bidirektionalen `1-zu-*` Assoziation im Generat des Entity-Generators. Die Richtung aus Sicht des `Owner` ist in Listing 7.20 dargestellt.

Seite einer bidirektionalen Assoziation als Standardverhalten verwendet. Die Assoziation aus Sicht der `Owner` Seite der Assoziation wird, wie in Listing 7.18, auf die gleiche Art und Weise annotiert wie eine entsprechende unidirektionale Assoziation. Die Annotationen der Rückrichtung sind dabei leicht unterschiedlich. Würden sie gleich sein, wären sie auf technischer Ebene nicht von zwei unidirektionalen Assoziationen zu unterscheiden.

```
classdiagram C{
  association [*] A <-> B [1];
}
```

CD

...

```
public class B{
  @OneToMany(mappedBy="b")
  Set<A> a;
}
```

Java

...

Abbildung 7.24: Abbildung der Rückrichtung einer bidirektionalen `*-zu-1` Assoziation im Generat des Entity-Generators. Die Richtung aus Sicht des `Owner` ist in Listing 7.19 dargestellt.

Abbildung 7.22 zeigt die generierte Annotation und das generierte Attribut der Rückrichtung. Sie zeigt einen Auszug des Generats der Klasse `B`. Es wird wiederum die Annotation `@OneToOne` verwendet, aber diesmal mit dem Parameter `mappedBy`. Dieser nimmt Bezug auf das Attribut der `Owner`-Klasse. Hier sei angemerkt, dass dies der tatsächliche Attributname und nicht der Name der Datenbanktabelle oder der Spalte ist, in der das entsprechende Attribut gespeichert ist.

Abbildung 7.23 zeigt das Generat der Rückrichtung einer `1-zu-*` Assoziation. Der generierte Java-Code sieht aus wie der Code aus Abbildung 7.22. Die verwendete An-

notation und der Attributname sind verändert. Es wird die Annotation `@ManyToOne` verwendet. Die Richtung aus Sicht des `Owner`, wie in Abbildung 7.20 dargestellt, verwendet hingegen die Annotation `@OneToMany`. Dies entspricht der jeweiligen Richtung, aus der die Assoziation betrachtet wird. Ebenso folgt der Name des Attributs des `mappedBy` Parameters dem Namensschema mit dem angehängten "s".

Abbildung 7.24 zeigt das Generat der Rückrichtung einer `*-zu-1` Assoziation. Hierbei wird die Annotation `@OneToMany` verwendet und als Datentyp, da die Assoziation nicht geordnet ist, eine `Set` Implementierung verwendet.

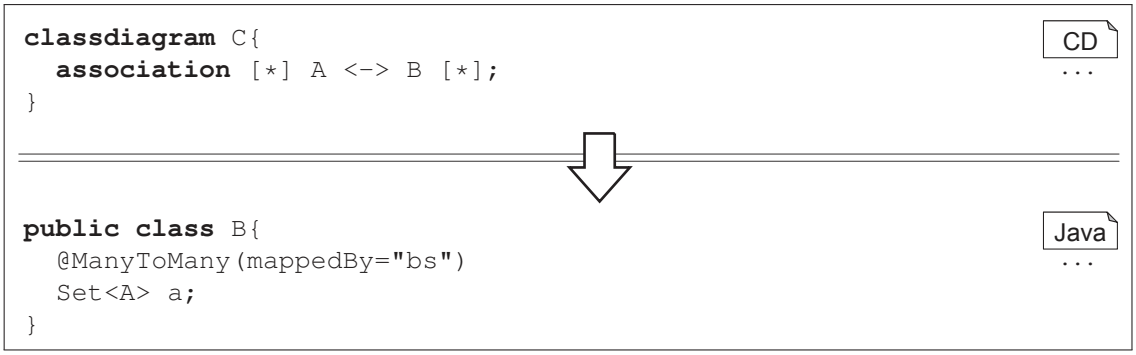


Abbildung 7.25: Abbildung der Rückrichtung einer bidirektionalen `*-zu-*` Assoziation im Generat des Entity-Generators. Die Richtung aus Sicht des `Owner` ist in Listing 7.21 dargestellt.

Darüber hinaus zeigt Abbildung 7.25 den generierten Code der Rückrichtung einer `*-zu-*` Assoziation. Hierbei wird in beiden Richtungen die Annotation `@ManyToMany` verwendet. Auch hier referenziert der `mappedBy` Parameter das entsprechend generierte Attribut der Klasse `A`. Neben unterschiedlichen Assoziationsrichtungen können Assoziationen qualifiziert oder geordnet sein. Qualifizierte Assoziation werden momentan vom Entity-Generator nur teilweise unterstützt. Ein Qualifikator, der Attribut einer Klasse ist, wird unterstützt. Ein beliebiger Typ als Qualifikator hingegen nicht. Geordnete Assoziationen werden durch den `Ordered` Tag modelliert und werden dort detailliert beschrieben. Generell erhalten die verwendeten Annotationen `@OneToOne`, `@OneToMany`, `@ManyToOne` oder `@ManyToMany` Kaskadierungs- oder Ladestrategieinformationen. Diese werden durch die `Cascading` und `Fetch` Tags modelliert und im entsprechenden Abschnitt detailliert erläutert.

Weitere Konzepte des Klassendiagramms sind Vererbung, deren Abbildung auf die Persistenz mit Hilfe des `Inheritance` Tags modelliert wird. Die übrigen Konzepte werden wie in [Sch12] beschrieben abgebildet.

Auswirkungen des `Entity` Tags

Der Entity-Generator behandelt alle Klassen, die mit dem `Entity` Tag getaggt sind. Dazu generiert er bei jeder Klasse, welche innerhalb der Tagdefinition getaggt ist, die

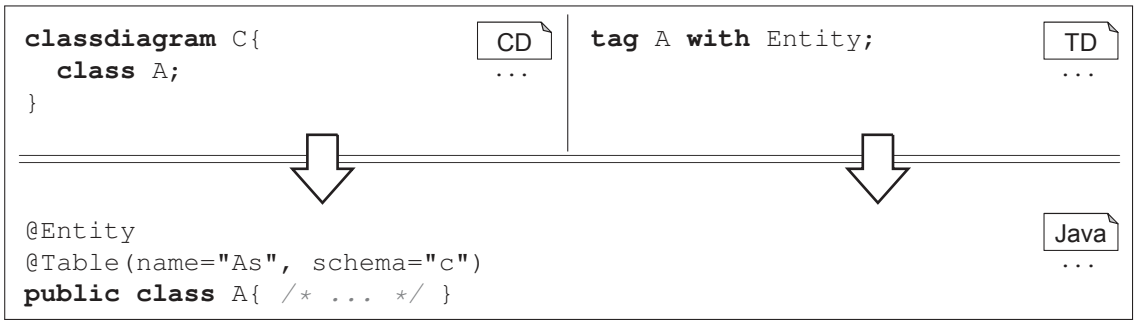


Abbildung 7.26: Auswirkungen des `Entity` Tags auf das Generat des Entity-Generators.

Annotation `@Entity`. Das Taggen der Klassen als Entitäten ist dabei nicht optional, da sonst normale Java-Klassen ohne Annotationen generiert werden. Andere Tags können, da hier Standardwerte verwendet werden, optional sein. Abhängig vom `@Inheritance` Tag und dem Modifikator der Klasse generiert der Entity-Generator die Annotation `@MappedSuperclass` anstelle der Annotation `@Entity` für jede mit dem `Entity` Tag getaggte Klasse. Dies bewirkt, dass der ORM bei der Abbildung der Attribute der Klasse in ein relationales Datenbanksystem diese nicht wie eine normale Entität behandelt, sondern annimmt, dass eine Subklasse existiert, die auf eine Tabelle abgebildet wird und deren Zieltabelle auch die Attribute der `@MappedSuperclass` enthält. Dies geschieht immer dann, wenn die Option `tablePerClass` des `Inheritance` Tags gewählt wurde und die Klasse eine abstrakte Klasse ist. Wird eine andere Option gewählt oder ist die Klasse nicht abstrakt, aber dennoch mit dem `Entity` Tag getaggt, wird auch die Annotation `@Entity` generiert. Wird die Annotation `@Entity` generiert, wird auch die Annotation `@Table` generiert. Diese definiert, in welche Tabelle welchen Schemas Instanzen der Klasse abgelegt werden. Das Attribut `name` definiert dabei den Namen der Tabelle, wohingegen das Attribut `schema` den entsprechenden Namen des Schemas definiert. Dazu verwendet der Entity-Generator konfigurierbare Standardwerte für den Tabellen- und den Schemanamen. Der Tabellenname basiert auf dem Namen der Klasse mit einem angehängten ”s” als Suffix. Der Schemaname basiert auf dem Namen des Klassendiagramms. Abbildung 7.26 zeigt die Generierung der Annotationen `@Entity` und `@Table`. Dieses Standardverhalten kann, wie in Kapitel 10 beschrieben, verändert werden. Gleichzeitig generiert der Entity-Generator in Abhängigkeit vom `Inheritance` Tag ein ID-Attribut. Die eigentliche Generierung des ID-Attributs wird bei der Erläuterung des `IDGen` Tags erklärt. Die Stelle, an der das Attribut generiert wird, wird im entsprechenden Abschnitt des `Inheritance` Tags detailliert erläutert.

Auswirkungen des `Fetch` Tags

Abbildung 7.27 zeigt die Auswirkungen des `Fetch` Tags. Dargestellt ist die Auswirkung des Tags auf eine `1-zu-1` Assoziation. Der `Fetch` Tag bewirkt, dass der Parameter `fetch` der verwendeten Annotation `@OneToOne` gesetzt wird. Der Parameterwert ergibt

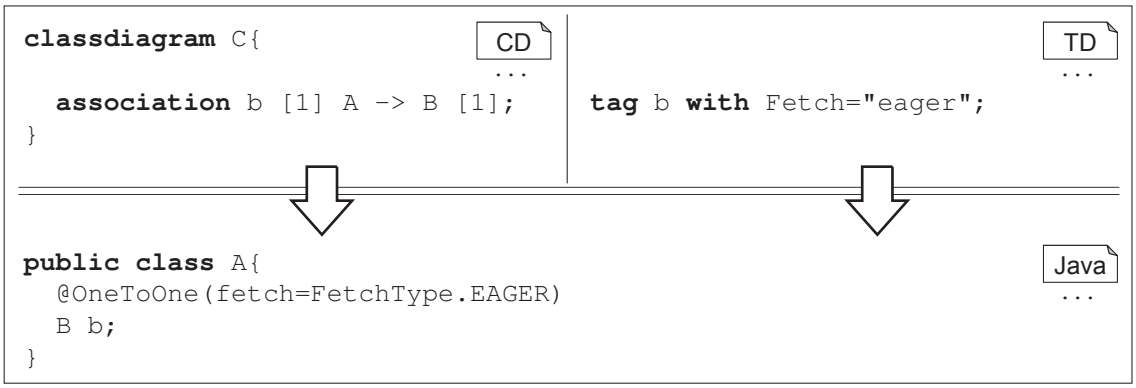


Abbildung 7.27: Auswirkungen des `Fetch` Tags auf das Generat des Entity-Generators.

sich aus der im Tag verwendeten Option `eager`. Bei Verwendung der `lazy` Option verändert sich der Parameterwert analog. Der Tag wirkt sich ebenso auf die Annotationen `@OneToMany`, `@ManyToOne` und `@ManyToMany` für Assoziationen aus.

Auswirkungen des `Cascading` Tags

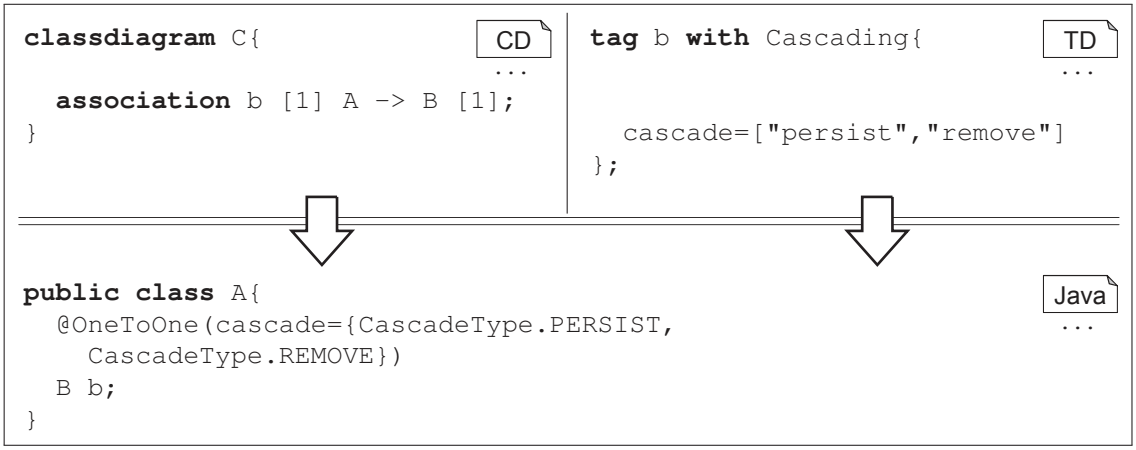


Abbildung 7.28: Auswirkungen des `Cascading` Tags auf das Generat des Entity-Generators.

Abbildung 7.28 zeigt die Auswirkungen des `Cascading` Tags. Dargestellt ist die Auswirkung des Tags auf eine `1-zu-1` Assoziation. Dabei wurden die Optionen `persist` und `remove` in der Tagdefinition gewählt. Durch die Verwendung der Optionen werden die Parameterwerte des `cascade` Parameters gesetzt. Wie in Abbildung 7.28 gezeigt, ist der Parameterwert mengenwertig. Für jede Option des Tags wird ein Element der Menge erzeugt. Wird für den Tag die Option `none` gewählt, wird der `cascade` Parameter nicht generiert, da die Nichtangabe innerhalb der Annotation der Option `none` des

Tags gleich ist. Auf Ebene des Java-Codes wird durch den ORM keine statische Analyse, die die Verwendung widersprüchlicher Optionen angibt, durchgeführt. Dies wird erst zur Laufzeit während der Initialisierung des Systems geprüft. Hier bietet MontiEE einen Vorteil, indem mit Hilfe der in Abschnitt 7.4.3 vorgestellten Kontextbedingungen des Generators dies bereits auf Ebene der Tagdefinition geprüft wird. Analog wirkt sich der Tag auf die anderen Annotationen `@OneToMany`, `@ManyToOne` und `@ManyToMany` für Assoziationen aus.

Auswirkungen des `Inheritance` Tags

Abbildung 7.29: Auswirkungen des `Inheritance` Tags auf das Generat des Entity-Generators.

Abbildung 7.29 zeigt die Auswirkung der Verwendung des `Inheritance` Tags auf den generierten Java-Code. Zunächst wird die `@Inheritance` Annotation an den Klassenkopf der Klasse `A` generiert. Diese Annotation beinhaltet, abhängig von der gewählten Option, `singleTable`, `joined` oder `tablePerClass`, des `Inheritance` Tags, den `strategy` Parameter. In Abbildung 7.29 ist die `singleTable` Option reflektiert. Die Semantik der Annotationen entspricht der Semantik der unterschiedlichen Optionen des Tags, die in Abschnitt 7.2 detailliert erläutert wurden. Für die `singleTable` Option wird ebenfalls die Annotationen `@DiscriminatorColumn` und `@DiscriminatorValue` generiert. Erstere referenziert die Spalte der Datenbanktabelle. Sie enthält den die unterschiedlichen Typen der Vererbungshierarchie unterscheidenden Wert. Letztere definiert den Wert, der für die Klasse `A` als Diskriminator verwendet wird. Dazu wird vom Entity-Generator immer der Klassenname verwendet. Bei den beiden anderen Optionen werden diese Informationen nicht benötigt und auch nicht generiert. Abhängig von der gewählten Vererbungsstrategie bestimmt sich aber zusätzlich die Generierung des ID-Attributes. Die genaue Generierung wird bei der Auswirkung des `IDGen` Tags beschrieben. Die Lokalisierung des Attributs wird, da sie auf dem `Inheritance` Tag beruht, hier beschrieben. Wird die `singleTable` Option gewählt, wird das ID-Attribut in die oberste Klasse der Vererbungshierarchie, da alle Instanzen der Vererbungshierarchie in der gleichen Datenbanktabelle gespeichert werden und somit auch die gleiche

Strategie zur Erzeugung und Berechnung der ID verwenden, generiert. Die Strategie wird bei den Auswirkungen des `IDGen` Tags detailliert erläutert. Bei Verwendung der `joined` Option wird das ID-Attribut aus dem gleichen Grund ebenfalls in die oberste Klasse der Vererbungshierarchie generiert. Lediglich bei der `tablePerClass` Option wird für jede konkrete Klasse ein eigenes ID-Attribut generiert. Ist die Klasse abstrakt, werden abstrakte Getter- und Setter-Methoden generiert. Getter- und Setter-Methoden für die ID werden in allen Klassen in jedem Fall generiert.

Auswirkungen des `IDGen` Tags

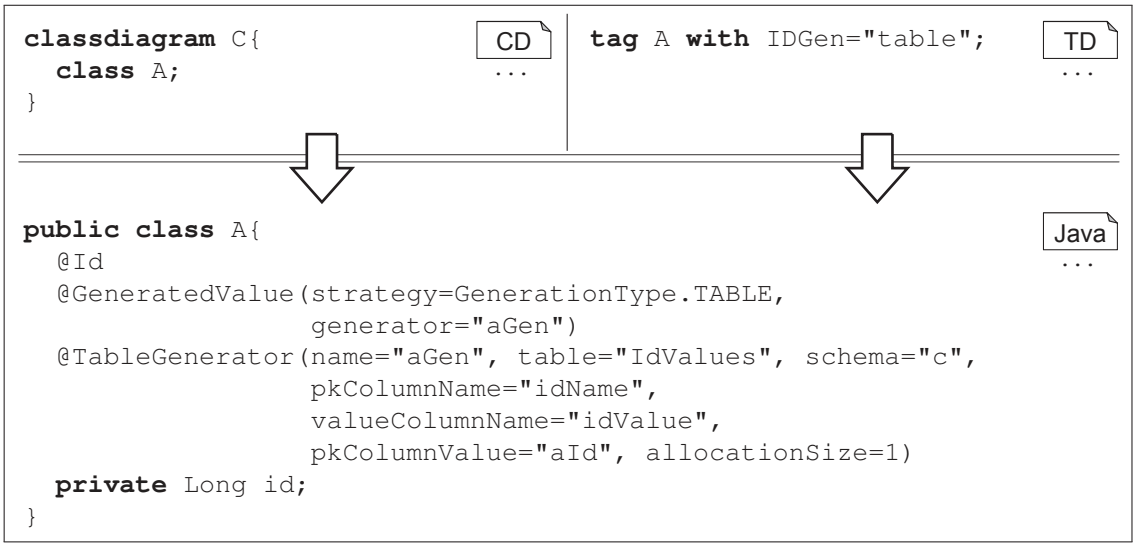


Abbildung 7.30: Auswirkungen des `IDGen` Tags auf das Generat des Entity-Generators.

Die Auswirkungen des `IDGen` Tags auf den generierten Java-Code sind in Abbildung 7.30 dargestellt. Generell wird für jede Entität, die mit der Annotation `@Entity` getaggt wurde, das ID-Attribut `protected Long id` abhängig vom `Inheritance` Tag generiert. Dieses ID-Attribut ist vom Typ `Long` und hat den Namen `id`. Zusätzlich wird die Annotation `@Id` verwendet, die das Attribut als Primärschlüssel auszeichnet. Mit Hilfe des `IDGen` Tags kann zusätzlich eine Strategie zur Bildung des Primärschlüssels angegeben werden. Die unterschiedlichen Strategien sind durch die möglichen Optionen des Tags dargestellt. Generell führt der `IDGen` Tag zur Verwendung der Annotation `@GeneratedValue` im generierten Java-Code. Diese besitzt den `strategy` Parameter, der verschiedene Optionen annehmen kann. In Abbildung 7.30 ist die entsprechende Annotation bei Verwendung der `table` Option des Tags dargestellt. Die Optionen `auto` und `identity` führen zur Generierung einer Annotation `@GeneratedValue`, die nur den Parameter `strategy` verwendet. Die Verwendung der `none` Option führt dazu, dass die Annotation nicht generiert wird. Die Verwendung der Optionen `sequence` und `table` führen dazu, dass, wie dargestellt, der zusätzliche Parameter `generator`

verwendet wird. Bei der Option `sequence` referenziert dieser Parameter einen vom zu Grunde liegenden Datenbanksystem dargestellten Sequenzgenerator, wie in Abschnitt 7.2 beschrieben. Bei Verwendung der `table` Option referenziert der Parameterwert eine durch die Annotation `@TableGenerator` definierte Datenbanktabelle, die ID-Werte für alle Typen enthält. Der Name entspricht dabei der zuvor verwendeten Referenz. Ebenso sind die Werte der `table`, `pkColumnName` und `valueColumnName` Parameter in MontiEE als Standardwerte fest vorgegeben. Der SQL-Generator, der in Abschnitt 7.6 präsentiert wird, generiert stets eine Datenbanktabelle namens `IDValues`, welche vom `table` Parameter referenziert wird. Diese Tabelle besitzt zwei Spalten: `idName` und `idValue`. Die Spalte `idName` wird durch den `pkColumnName` Parameter referenziert und enthält einen eindeutigen Namen für einen Typ. Die Spalte `idValue` wird durch den `valueColumnName` Parameter referenziert und enthält den fortlaufenden Wert zur Bildung der nächsten konsekutiven ID für einen Typ. Der `schema` Parameter bezeichnet wiederum das Datenbankschema und wird aus dem Klassendiagrammnamen gebildet. Der Parameter `pkColumnValue` benennt den eindeutigen Namen, der in der `idName` Spalte verwendet wird. Der Parameter `allocationSize` bestimmt die Schrittweite der ID-Werte.

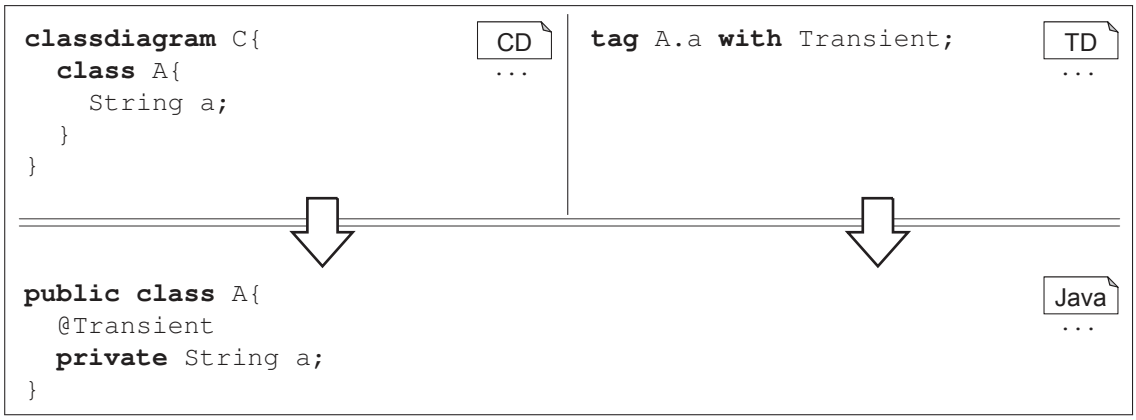


Abbildung 7.31: Auswirkungen des `Transient` Tags auf das Generat des Entity-Generators.

Auswirkungen des `Owner` Tags

Die Auswirkungen des `Owner` Tags wurden bereits zur Abbildung bidirektionaler Assoziationen verwendet und dort vorgestellt. Die Seite der Assoziation, die als `Owner` getaggt wird, erhält dabei die Annotationen für unidirektionale Assoziationen. Die andere Seite erhält die entsprechende Annotation der Rückrichtung unter Verwendung des `mappedBy` Attributs der jeweiligen Annotation.

Auswirkungen des `Transient` Tags

Attribute können mit dem Tag `Transient` getaggt werden. Dies führt zur Verwendung der Annotation `@Transient` im generierten Quellcode. Es sei angemerkt, dass sich die Semantik der Annotation von der des Java-Schlüsselworts `transient` dahingehend unterscheidet, dass das Schlüsselwort angibt, ob ein entsprechendes Attribut serialisiert wird, wohingegen die Annotation bestimmt, ob ein Attribut persistent gespeichert wird. Es kann daher Attribute geben, die serialisiert werden, also z.B. zum Client versendet, aber nicht in der Datenbank gespeichert werden. Demgegenüber können Java-Attribute in der Datenbank gespeichert, aber nicht serialisiert und nicht zum Client übertragen werden.

Auswirkungen des `Ordered` Tags

Abbildung 7.32: Auswirkungen des `Ordered` Tags auf das Generat des Entity-Generators.

Abbildung 7.32 zeigt die Auswirkungen des `Ordered` Tags. Mit Hilfe des Tags lässt sich spezifizieren, dass eine Assoziation geordnet ist. Er kann nur bei Assoziationen mit Zielkardinalität $*$ eingesetzt werden. Im generierten Javacode wird die Annotation `@OrderColumn` generiert und der Datentyp des generierten Attributs von `Set` auf `List` geändert. Der Name des Attributs entspricht dem Namen der Assoziation. Die Semantik einer geordneten Assoziation bezieht sich dabei auf die Reihenfolge der Elemente in der Liste, nicht aber auf eine Sortierung der Elemente innerhalb der Liste nach einer bestimmten Eigenschaft. Dies könnte als Erweiterung des Tags umgesetzt werden. Die Annotation `@OrderColumn` besitzt darüber hinaus den Parameter `name`. Dieser definiert den Namen der Datenbankspalte, die dazu verwendet wird, die Sortierung der Liste nach dem Laden aus der Datenbank wieder herzustellen. Sie muss vom SQL-Generator, der in Abschnitt 7.6 vorgestellt wird, in der Jointabelle zur Abbildung der Assoziation zusätzlich generiert werden.

Auswirkungen des `Queries` Tags

Abbildung 7.33 zeigt die Auswirkungen des `Queries` Tag auf das Generat. Der `Queries` Tag besteht aus mehreren `Query` Tags, die wiederum einen Namen und einen Wert besitzen. Dieser Wert kann, wie bereits beschrieben, entweder ein HQL- oder ein SQL-String sein. Innerhalb des Generats werden Queries durch die Annotationen `@NamedQueries` oder `@NamedNativeQueries` unterschieden. Erstere beinhalten eine Menge von Queries, die in HQL geschrieben sind. Jedes Query beginnt dabei mit der Annotation `@NamedQuery`. Letztere beinhalten eine Menge von Zielplattform spezifischen Queries, in diesem Fall SQL-Queries. Die Annotationen haben, analog zum Tag, zwei Parameter, die den Namen und den Inhalt des Queries repräsentieren. Der Name des Queries wird später noch einmal bei der Erläuterung der Generierung der DAOs benötigt.

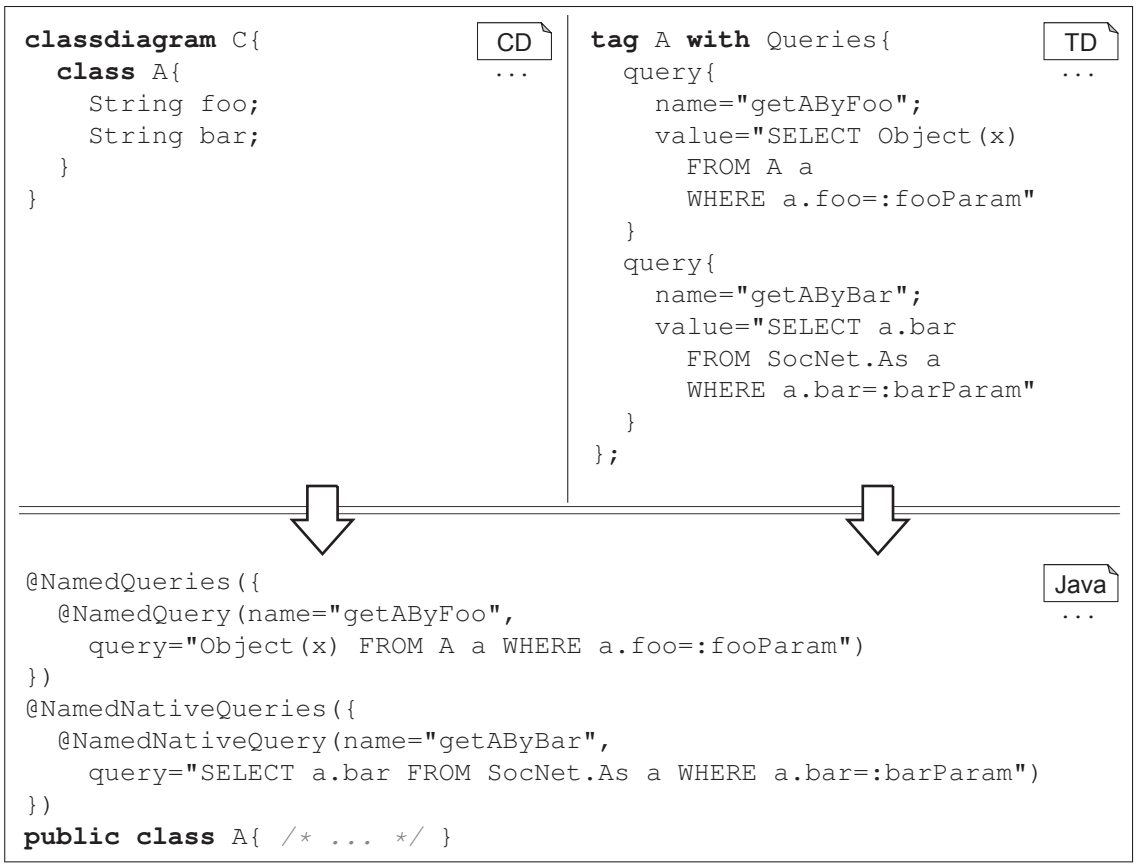


Abbildung 7.33: Auswirkungen des `Queries` Tags auf das Generat des Entity-Generators.

7.4.2 Generaterweiterungen

Nachdem zuvor die Auswirkungen der einzelnen Tags und die Abbildung der Klassendiagrammkonzepte auf das Generat beschrieben wurden, werden in diesem Abschnitt Erweiterungen sowie Erweiterungsmöglichkeiten des Generats vorgestellt (vgl. FA16-PE und FA17-PE). In [Sch12] wird eine Methodik zur Erweiterung bestehender Templates über vordefinierte Hotspots erläutert. Diese Methodik wird auch von MontiEE unterstützt, so dass das Generat erweitert werden kann. Diese Erweiterung bezieht sich auf Methoden, die den Klassen hinzugefügt werden können. Innerhalb von MontiEE wird dieser Mechanismus dazu genutzt, den generierten Java-Klassen drei Methoden hinzuzufügen. Jeder generierten Klasse wird eine `equals(Object o)` Methode hinzugefügt, die die bereits existierende überschreibt. Entgegen der Empfehlung aus [Blo08], nach denen eine solche `equals` Methode die Eigenschaften Reflexivität, Symmetrie, Transitivität, Konsistenz und "Nicht-Nullbarkeit" erfüllen muss und nur spärlich überschrieben werden darf, wird dies durch die Verwendung des ORM gefordert. In [Blo08] ist dargestellt, dass die `equals` Methode immer dann überschrieben werden muss, wenn Objekte im Datentyp `Set` gespeichert werden, da die JPA Objektidentität nur innerhalb einer Session zusichern kann. Dies ist auf dem Applikationsserver nicht zwingend gegeben und der Vergleich auf Objektidentität ist die Standardimplementierung der `equals` Methode. Der Applikationsserver erzeugt bei jedem Sessionwechsel und bei jedem Laden und Speichern in der Datenbank neue Objekte, so dass die Objektidentitäten sich stets ändern. Aus diesem Grund muss die überschriebene `equals` Methode auf Basis der Datenbank-ID implementiert werden. Dazu wird sich das generierte ID-Attribut zu Nutze gemacht. Zwei Objekte sind die gleichen Objekte, wenn sie den gleichen Typ und die gleiche Datenbank-ID haben. Analog wird die `hashCode` Methode, [Blo08] folgend, generiert. Um dennoch Objekte auf inhaltliche Gleichheit prüfen zu können, wird für jedes Objekt eine `deepEquals` Methode, die die Werte der Attribute, aber auch transitiv alle Assoziationen auf deren Gleichheit überprüft, generiert. Die Methode nimmt an, dass alle assoziierten Typen ebenfalls eine `deepEquals` Methode bereitstellen. Ihre Implementierung folgt der Implementierung aus [Blo08], die diese Implementierung für eine `equals` Methode präsentieren. Da die assoziierten Elemente generiert sind, kann dies ohne Einschränkung angenommen werden. Die generierte Methode kann auch mit zyklischen Objektgraphen umgehen, da sie sich bereits besuchte Objekte merkt und diese nicht erneut besucht. Weitere Methoden können über den von den Templates wie in [Sch12] beschriebenen Hotspot eingehängt werden. Dadurch lassen sich die standardmäßig generierten Methoden erweitern.

Darüber hinaus müssen alle generierten Klassen serialisierbar sein. Dies wird sowohl für den Transfer bei der Kommunikation mit der Datenbank als auch mit Clients benötigt. Dazu implementiert jede generierte Klasse das von der Zielsprache Java bereitgestellte Interface `Serializable`. Darüber hinaus erhält jede Klasse auch das vom Interface geforderte Feld `private static final long serialVersionUID` mit einem generierten Wert. Das Feld sowie sein Wert werden im Rahmen dieser Arbeit nicht weiter benötigt und betrachtet. Zudem werden zu jeder Klasse weitere, nicht in der Tagdefinition modellierte Queries hinzugefügt. Es werden Queries generiert, die ein Objekt auf

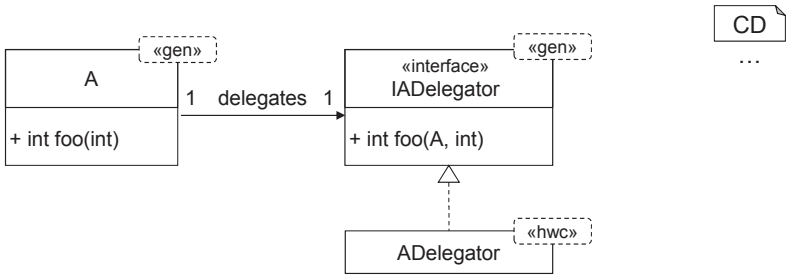


Abbildung 7.34: Darstellung des Konzepts der Delegation zum Hinzufügen spezifischer Funktionalität.

Basis eines Attributwerts aus der Datenbank laden. Dazu wird für jedes modellierte Attribut im Klassendiagramm ein Query der Form `getObjectsByAttribute` generiert. Dem Query wird ein möglicher Attributwert als Parameter übergeben. Zusätzlich zu den modellierten Attributen wird auch für jede Assoziation einer Klasse ein Query der Form `getObjectsByObject` generiert. Dabei wird dem Query ein Objekt als Parameter übergeben. Hierbei wird ausgenutzt, dass die HQL mit Objekten und nicht mit Relationen arbeitet.

Ein zusätzlicher Erweiterungspunkt ist die Möglichkeit, im Klassendiagramm Methoden ohne Implementierung zu modellieren. Die Implementierung kann bereits im Klassendiagramm mit Hilfe von Java als eingebetteter Sprache [Sch12] modelliert werden. Dies wird aber nicht erzwungen. Um dennoch mit modellierten Methoden umgehen zu können und gleichzeitig die Methoden dazu zu nutzen, spezifische Funktionalität hinzuzufügen, werden Delegatoren, wie in [Blo08] vorgeschlagen, generiert.

Abbildung 7.34 zeigt den generierten Code für eine modellierte Klasse `A` mit einer Methode `+ int foo(int bar)`. Aus dieser Klasse wird eine entsprechend annotierte Java-Klasse `A` wie zuvor beschrieben generiert. Diese Klasse besitzt einen Delegator, dessen Implementierung handgeschrieben werden muss. Ein Interface, welches die modellierte Methode enthält, wird als Unterstützung für die Implementierung angegeben. Dabei werden die Methodenparameter dahingehend verändert, als dem Delegator die Instanz der delegierenden Klasse zu weiteren Berechnungen mitgegeben wird. Ebenso wird in der delegierenden Klasse davon ausgegangen, dass die manuell implementierte Klasse dem vorgegebenen Namen folgt. Weitere Alternativen zur Integration von handgeschriebenem Code werden in [HLMSN$^+$15a, HLMSN$^+$15b] gegeben. Eine Alternative, die im Rahmen von MontiEE nur bedingt anwendbar ist, stellt die Möglichkeit zur Subklassenbildung dar. Generell ist dies möglich, allerdings erfordert dies, dass die Subklasse ebenfalls die korrekten Annotationen verwendet, so dass sie vom ORM verarbeitet werden können. Ferner muss sie der Strategie zur Abbildung von Vererbung in der Datenbank folgen. Auch der SQL-Generator, der in Abschnitt 7.6 präsentiert wird, muss erweitert werden. Dies bedeutet, dass der Entwickler sehr genaue technologiespezifische Kenntnisse besitzen und in der Lage sein muss, seinen handgeschriebenen Quellcode korrekt zu annotieren. Da ein Ziel von MontiEE gerade ist, von der Technologie zu abstrahieren,

wird hier die Möglichkeit zur Delegation verwendet. Dennoch lässt sich auch auf diese Weise zusätzliche Funktionalität in die generierten Entitäten mischen.

7.4.3 Kontextbedingungen

Das MontiEE Tagschema aus Abschnitt 7.2 bedingt einige generatorspezifische Kontextbedingungen, die aus der Verwendung der Tags durch die Generatoren und nicht direkt aus der Tagdefinitionssprache oder der Tagschemasprache entstehen. Sie entstehen durch die Komposition von Klassendiagrammsprache und MontiEE Tagdefinitionen sowie der Semantik der Tags, die von dem jeweiligen Generator umgesetzt werden muss. Die allgemeinen Kontextbedingungen der Sprachen, die prüfen, dass die Tags nur an die erlaubten Elemente modelliert wurden und dass keine gleichen Tags doppelt verwendet wurden, sind Kontextbedingungen der Tagdefinitionssprache bzw. der Tagschemasprache und werden hier nicht erneut aufgeführt, sondern sind in Abschnitt 4.3.3 erläutert.

Die Kontextbedingungen werden für die einzelnen Tags vorgestellt. Begonnen wird dabei mit den Kontextbedingungen, die aus der Verwendung des `Entity` Tags entstehen.

MontiEE-1

Bedingung: Assoziationen müssen immer zwischen zwei als `Entity` getaggten Elementen modelliert werden. Es sei denn, es handelt sich um eine unidirektionale Assoziation mit Zielkardinalität `1`, dann greift die Kontextbedingung MontiEE-2.

Schweregrad: Fehler

MontiEE-2

Bedingung: Ist das Zielelement einer unidirektionalen Assoziation nicht als `Entity` getaggt, muss es als `Transient` getaggt sein.

Schweregrad: Fehler

Darüber hinaus gibt es für den `Fetch` Tag lediglich eine Warnung als Kontextbedingung, da die anderen Fälle bereits durch die allgemeinen Kontextbedingungen der Taggingsprachen abgedeckt sind.

MontiEE-3

Bedingung: Wird die linke Seite einer Assoziation mit dem `Fetch` Tag getaggt, muss die Navigationsrichtung beim Taggen der linken Seite von links nach rechts oder bidirektional, beim Taggen der rechten Seite andersherum, sein.

Schweregrad: Warnung

Diese Kontextbedingung ist kein Fehler, da ein Taggen der jeweiligen anderen Seite keine semantische Auswirkung besitzt, sondern lediglich nicht benötigte Informationen

modelliert. Für den `Cascading` Tag existiert diese Warnung analog. Zudem werden weitere Kontextbedingungen als Warnungen definiert, die die multiple Verwendung des `Cascade` Tags innerhalb des `Cascading` Tags prüfen.

MontiEE-4

Bedingung: Wird die linke Seite einer Assoziation mit dem `Cascading` Tag getaggt, muss die Navigationsrichtung beim Taggen der linken Seite von links nach rechts oder bidirektional sein, beim Taggen der rechten Seite umgekehrt.

Schweregrad: Warnung

MontiEE-5

Bedingung: Wird die Option `all` des `Cascade` Tag verwendet, darf kein weiterer `Cascade` Tag mit der Option `none` verwendet werden.

Schweregrad: Fehler

MontiEE-6

Bedingung: Wird die Option `none` des `Cascade` Tags verwendet, darf kein weiterer `Cascade` Tags mit einer der anderen Optionen verwendet werden.

Schweregrad: Fehler

MontiEE-7

Bedingung: Wird die Option `all` des `Cascade` Tags verwendet, ist ein weiterer `Cascade` Tag mit einer der Optionen `merge`, `persist`, `refresh` oder `remove` redundant.

Schweregrad: Warnung

Die nächsten Kontextbedingungen betreffen den `Inheritance` Tag. Sie resultieren zum Teil aus dem JPA Standard, da dieser verschiedene Bedingungen, die für den Modellierer durch MontiEE bereits auf Modellebene abgefangen werden können, vorgibt. Zudem erlauben einige Implementierungen, wie Hibernate, auch einen Wechsel der Abbildung von Vererbung innerhalb einer Vererbungshierarchie. Da dies aber nicht allgemein umgesetzt ist und die Vermischung der Strategien eher Spezialfälle darstellen, wird dies auch auf Generatorebene unterbunden.

MontiEE-8

Bedingung: Innerhalb einer Vererbungshierarchie ist die Verwendung des Tags `Inheritance` nur einmalig zulässig.

Schweregrad: Fehler

MontiEE-9

Bedingung: Die Vererbungsstrategie muss auf der höchstmöglichen Hierarchieebene mit Hilfe des `Inheritance` Tags spezifiziert sein.

Schweregrad: Fehler

MontiEE-10

Bedingung: Wird mit Hilfe des `Inheritance` Tags die Strategie zur Abbildung von Vererbung in der Datenbank auf `singleTable` festgelegt, so dürfen in keiner Subklasse Attribute existieren, die paarweise den gleichen Namen, aber einen unterschiedlichen Typ haben.

Schweregrad: Fehler

Eine weitere Kontextbedingung betrifft die Verwendung des `IDGen` Tags. Basierend auf der gewählten Strategie zur Vererbung sind nicht alle Optionen verfügbar. Dies hat Framework interne Gründe, die auf die Speicherung des faktischen ID-Wertes zurückzuführen sind und hier der Vollständigkeit halber erwähnt, aber nicht weiter betrachtet werden.

MontiEE-11

Bedingung: Wird mit Hilfe des `IDGen` Tags die Generierungsstrategie eines Primärschlüssels der Datenbanktabelle spezifiziert, muss die getaggte Klasse ebenfalls mit `Entity` getaggt sein.

Schweregrad: Fehler

MontiEE-12

Bedingung: Wird mit Hilfe des `Inheritance` Tags die Strategie zur Abbildung von Vererbung in der Datenbank auf `tablePerClass` festgelegt, so darf der `IDGen` Tag nicht auf `auto` oder `identity` festgelegt sein.

Schweregrad: Fehler

Darüber hinaus benötigt auch der `Owner` Tag weitere Kontextbedingungen, die über die in Abschnitt 4.3.3 vorgestellten hinausgehen.

MontiEE-13

Bedingung: Der `Owner` Tag sollte nur bei bidirektionalen Assoziationen verwendet werden, da er keine Auswirkung auf unidirektionale Assoziationen hat.

Schweregrad: Warnung

MontiEE-14

Bedingung: Der `Owner` Tag kann genau an einer Seite einer bidirektionalen Assoziation verwendet werden. Die Verwendung an beiden Seiten ist nicht zulässig.

Schweregrad: Fehler

MontiEE-15

Bedingung: Ist eine Assoziation bidirektional und sind beide Elemente mit dem `Entity` Tag getaggt, sollte eine Seite der Assoziation mit dem `Owner` Tag getaggt sein. Sonst wird die linke Seite der Assoziation als Standardwert verwendet.

Schweregrad: Warnung

Die letzten Kontextbedingungen beziehen sich auf den `Queries` Tag und den `Query` Tag. Dabei stellen diese Kontextbedingungen Bedingungen zwischen der Klassendiagrammsprache und der SQL bzw. HQL dar. Zur Prüfung dieser Kontextbedingungen werden die Informationen der anderen Tags teilweise benötigt. So hat die verwendete Strategie zur Abbildung von Vererbung, aber auch die Verwendung eingebetteter Typen Einfluss auf die entstehenden Tabellen und Spalten der Datenbank. Wird als Wert eines Queries SQL verwendet, müssen die verwendeten Referenzen den Tabellen und Spalten entsprechen. Wird hingegen die HQL verwendet, müssen die Klassen- und Attributnamen der annotierten Java-Klassen verwendet werden. Dies beeinflusst die Prüfung der Kontextbedingungen. Die folgenden Kontextbedingungen lassen sich definieren.

MontiEE-16

Bedingung: Der Name eines Queries muss eindeutig sein.

Schweregrad: Fehler

MontiEE-17

Bedingung: Die in einem `FROM` Block eines Queries referenzierten Tabellen müssen als Typen im Klassendiagramm existieren.

Schweregrad: Fehler

MontiEE-18

Bedingung: Die in einem `SELECT` Block eines Queries referenzierten Spalten unterschiedlicher Tabellen müssen als Attribute der zugehörigen Typen im Klassendiagramm existieren.

Schweregrad: Fehler

MontiEE-19

Bedingung: Die in einem `WHERE` Block eines Queries referenzierten Spalten unterschiedlicher Tabellen sowie die verwendeten Navigationsreferenzen müssen als Attribute oder Assoziationen der zugehörigen Typen im Klassendiagramm existieren.

Schweregrad: Fehler

Zusammenfassend generiert der Entity-Generator Entitäten des vom Server verwendeten Domänenmodells, welches innerhalb einer Datenbank gespeichert werden kann. Dies erlaubt es dem Produktentwickler zu speichernde Information abstrakt zu modellieren und die technische Implementierung durch den Generator zu erhalten. Dazu wurde gezeigt, wie sich einzelne Tags der Tagdefinition auf die generierten Entitäten des Generats auswirken. Dabei wurden die Tags zur Generierung von Java-Annotationen, die wiederum vom ORM zur Persistierung verwendet werden, verwendet. Darüber hinaus wurden Abhängigkeiten zwischen einzelnen Tagdefinitionen aufgezeigt. Die Konsistenzsicherung der Tagdefinition bezogen auf die Semantik der Tagtypen wird mit Hilfe von Kontextbedingungen sichergestellt. Gleichzeitig kann der Entity-Generator mit Hilfe von Delegatoren, so dass handgeschriebener Quellcode ohne Kenntnis spezifischer Annotationen integriert werden kann, erweitert werden. Dies erlaubt es dem Produktentwickler zusätzliche Funktionalität zu integrieren. Auch die Möglichkeit zur Definition und zur automatischen Ableitung von Queries, basierend auf SQL und HQL, sowie die Konsistenzsicherung der Queries mit dem Klassendiagramm wurden gezeigt.

7.5 Der DAO-Generator

Nachdem im vorangegangenen Abschnitt der Entity-Generator vorgestellt wurde, wird in diesem Abschnitt der DAO-Generator, der die DAOs für eine Applikation generiert, vorgestellt. DAOs stellen typische CRUD Funktionalität bereit und ermöglichen zudem die Ausführung von Queries. Sie stellen die Schnittstelle zwischen Applikationsserver und Datenbank dar. Das DAO-Pattern [KS06] dient zur Abstraktion der plattformspezifischen Kommunikation mit der Datenbank (vgl. FA9-WE, FA19-PE und FA18-PE). Wie in Abschnitt 3.2 beschrieben, kommt dazu meistens JDBC zum Einsatz, aber auch andere Kommunikationsmöglichkeiten sind möglich. Ebenso können sich die DAOs je nach verwendetem Datenbankparadigma unterscheiden. Zur Entkopplung der Applikation von der spezifischen Persistenzplattform und Kommunikation werden DAOs verwendet. Der Produktentwickler erhält dadurch eine transparente Schnittstelle zur Datenbank, die er aus der handgeschriebenen Geschäftslogik nutzen kann.

Abbildung 7.35 gibt einen Überblick über den DAO-Generator. Er verwendet Klassendiagramme als Eingabe und produziert DAOs. Die DAOs stellen eine transparente Schnittstelle zwischen Applikations- und Datenbankserver dar. Dabei ist vorgesehen, dass der DAO-Generator DAOs für das Server-Domänenmodell generiert. Generell sieht

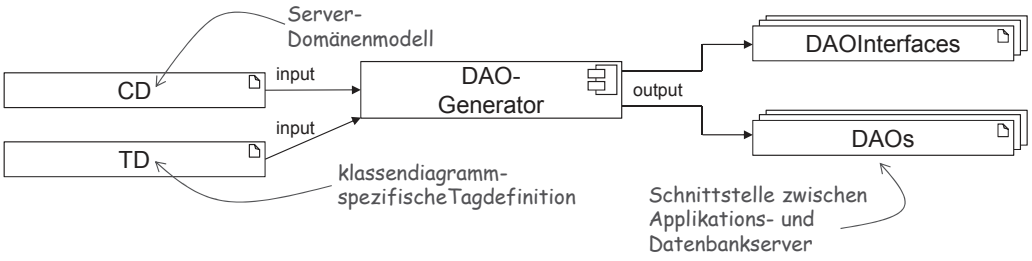


Abbildung 7.35: Darstellung der Eingabe- und Ausgabeartefakte des DAO-Generators. Er verwendet das Domänenmodell des Servers und erzeugt daraus viele DAOs und zugehörige Interfaces.

das Konzept des DAO-Generators vor, dass zu jeder Entität ein Interface mit den benötigten Methodensignaturen sowie eine spezifische auf den im Rahmen dieser Arbeit verwendeten Technologiestack zugeschnittene Implementierung generiert werden. Es sei an dieser Stelle angemerkt, dass die Funktionalität des Speicherns und des Ladens durch das DAO-Pattern externalisiert wird und nicht Teil des Verhaltens der Entitäten ist. Die zu speichernde Entität enthält keinerlei Wissen darüber, wo sie zu speichern ist, sondern überlässt dies einem Dritten, den DAOs. Diese werden typischerweise von der Businesslogik über eine API, die in Abschnitt 8.3 vorgestellt wird, aufgerufen und bekommen die zu speichernden Objekte übergeben.

Zunächst wird die Auswirkung einzelner Konzepte auf das Generat vorgestellt.

7.5.1 Abbildung der Modellierungskonzepte auf das Generat

In diesem Abschnitt wird die Abbildung der Eingabemodelle auf das Generat des DAO-Generators detailliert beschrieben. Dazu werden zunächst, wie bereits beim Entity-Generator, die Konzepte der Klassendiagramme und deren Auswirkungen aufgegriffen. Daran anschließend werden die in Abschnitt 7.2 vorgestellten MontiEE-Tagtypen aufgegriffen und deren Auswirkungen auf das Generat erläutert. Dabei werden nur die Tagtypen erläutert, die eine Auswirkung besitzen. Alle weiteren werden ausgelassen.

Auswirkungen der Klassendiagrammkonzepte

Für den DAO-Generator sind nur einige Konzepte des Klassendiagramms relevant. Relevant sind Attribute, Assoziationen und Klassen. Generell wird für jede Klasse des Klassendiagramms ein Interface und eine Implementierung des Interfaces vom DAO-Generator generiert. In Abbildung 7.36 sind exemplarisch die generierten Interfaces und DAOs für die drei modellierten Klassen dargestellt. Gezeigt sind die beiden Interfaces `IBDAO` und `ICDAO` sowie die implementierenden Klassen `BDAO` und `CDAO`. Die beiden Interfaces geben eine Menge von Methodensignaturen zur Erfüllung unterschiedlicher Aufgaben vor. Die Methoden `storeB(B)`, `getB(Long)`, `getBs()`, `updateB(B)` und `deleteB(B)`

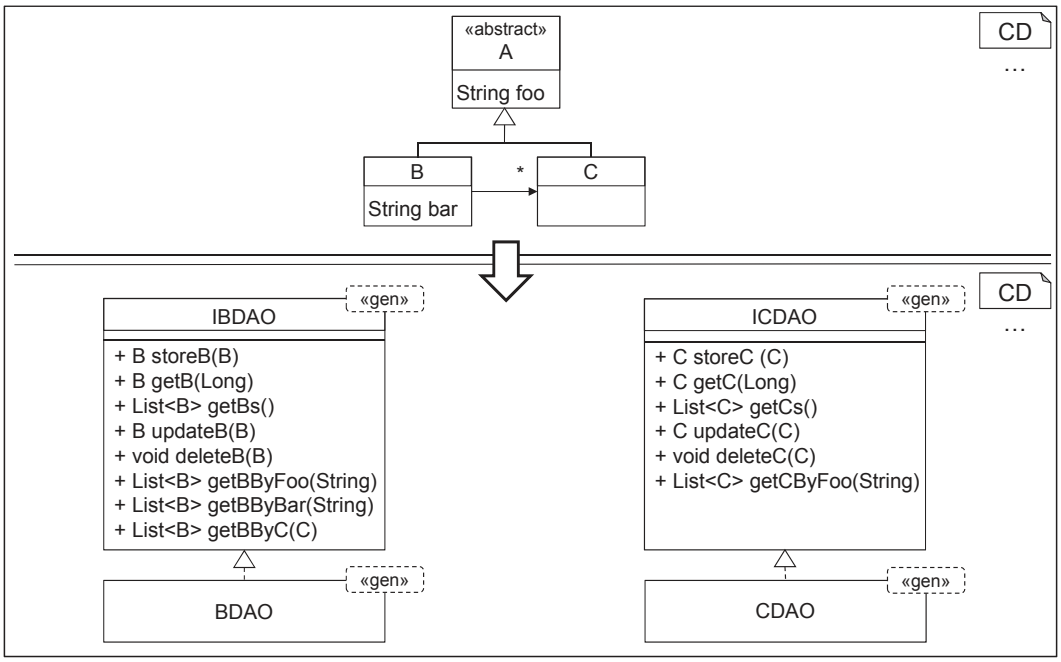


Abbildung 7.36: Exemplarische Darstellung des Generats des DAO-Generators basierend auf einem Klassendiagramm.

stellen die CRUD-Funktionalität dar. Es können zudem alle, oder ein einzelnes `B`, abhängig von seiner ID, aus der Datenbank gelesen werden. Darüber hinaus werden, wie zuvor in Abschnitt 7.4 beschrieben, für Attribute und Assoziationen entsprechende Queries generiert, die mit Hilfe des DAOs ausgeführt werden können. Dazu werden auch Methoden, die die Attribute von Superklassen berücksichtigen, generiert. Die Methoden `getBByBar(String)` und `getBByC(C)` verwenden Attribute der Klasse `B`. Die Methode `getBByFoo(String)` entstammt der Superklasse `A`. Ebenso sind Methoden für benutzerdefinierte Queries im Interface enthalten. Die schreibenden Methoden `storeB(B)` und `updateB(B)` liefern ein Objekt vom Typ `B` zurück. Dieses Objekt stellt das geschriebene Objekt dar. Dies ist unabhängig davon, ob das Objekt gerade erstellt oder aktualisiert wurde. Dies wird benötigt, da bei Speicherung in der Datenbank ein neues Objekt erstellt wird, so dass die Objektreferenz, die der Methode übergeben wurde, nicht mehr gültig ist. Dies ist durch das interne Management von Entitäten durch den ORM bedingt. Die Rückgabewerte der Methoden zur Ausführung von Queries sind immer als Listen, da ein Query generell mehrere Ergebnisse liefern kann, gekapselt. Eine Ausnahme stellt ein mit dem `Unique` Tag getaggtes Attribut dar. Dies wird im weiteren Verlauf detailliert erläutert. Das Interface `ICDAO` und dessen implementierende Klasse `CDAO` sind analog generiert und in Abbildung 7.36 dargestellt.

Für abstrakte Klassen und solche, die nicht mit dem `Entity` Tag getaggt sind, werden weder Interface noch Implementierung vom DAO-Generator generiert. Die Auswirkung

des `Entity` Tags und die Implementierung des DAOs wird nachfolgend genauer erläutert.

Für den Verwender der DAOs bedeutet dies, dass er für unterschiedliche zu speichernde Objekte verschiedene DAOs verwenden muss. Allerdings wird die Kapselung dadurch erhöht. Zudem werden durch die Kaskadierung, sofern sie nicht mit Hilfe des `Cascading` Tags einer Assoziation ausgeschaltet wurde, immer Objektgraphen in der Datenbank gespeichert. Somit muss der Benutzer der DAOs lediglich ein Objekt eines semantisch zusammenhängenden Teilgraphen speichern oder aktualisieren und die gleiche Operation wird auf den verbundenen Objekten ebenfalls ausgeführt. Gleiches gilt beim Laden aus der Datenbank. Auch hier werden immer verbundene Objekte geladen, so dass auch hier immer Teilgraphen des vollständigen Datenmodells geladen werden.

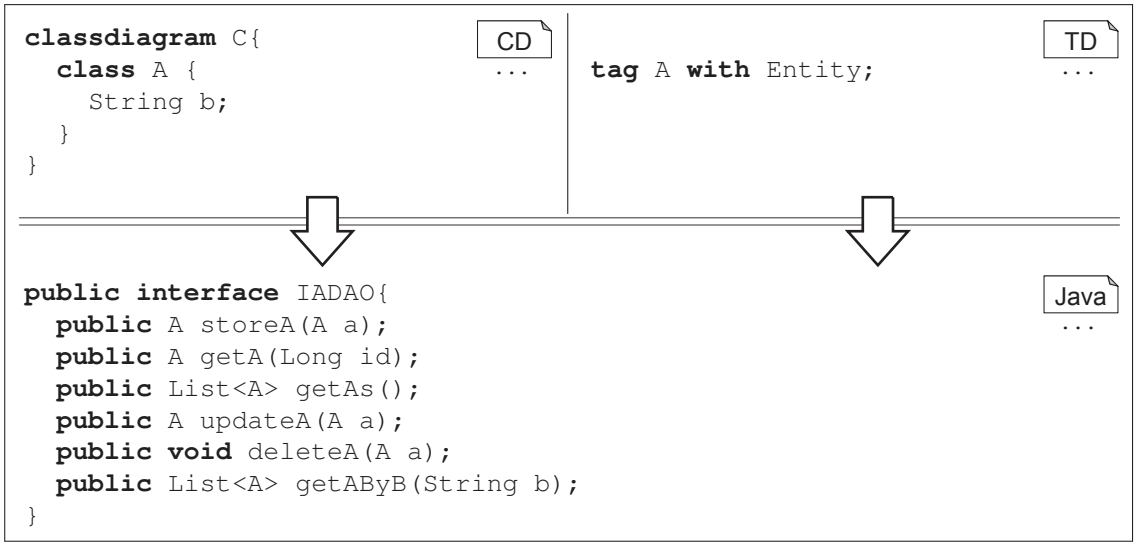


Abbildung 7.37: Auswirkungen des `Entity` Tags auf das durch den DAO-Generator generierte Interface.

Das Laden der Objekte erfolgt dabei entweder direkt oder als transparentes Nachladen bei Wahl der `lazy` Option des `Fetch` Tags. Diese Auftrennung der DAOs wird unter anderem benötigt, damit an verschiedenen Punkten im Datenmodell die entsprechenden Teilgraphen geladen werden können. Somit müssen nicht immer alle vorhandenen Daten geladen und gespeichert werden. Dies ist bei einer großen Datenmenge vorteilhaft. Zudem wird in Abschnitt 8.4 eine Fassade, die eine gemeinsame API über alle generierten DAOs legt, so dass der Benutzer nicht alle DAOs einzeln, sonden die delegierende Fassade verwenden kann, vorgestellt.

Auswirkungen des `Entity` Tags

Wie zuvor beschrieben, wird für jede Klasse, die als Entität getaggt wurde, ein Interface und eine Implementierung generiert. Der `Entity` Tag bewirkt, dass die Generierung stattfindet.

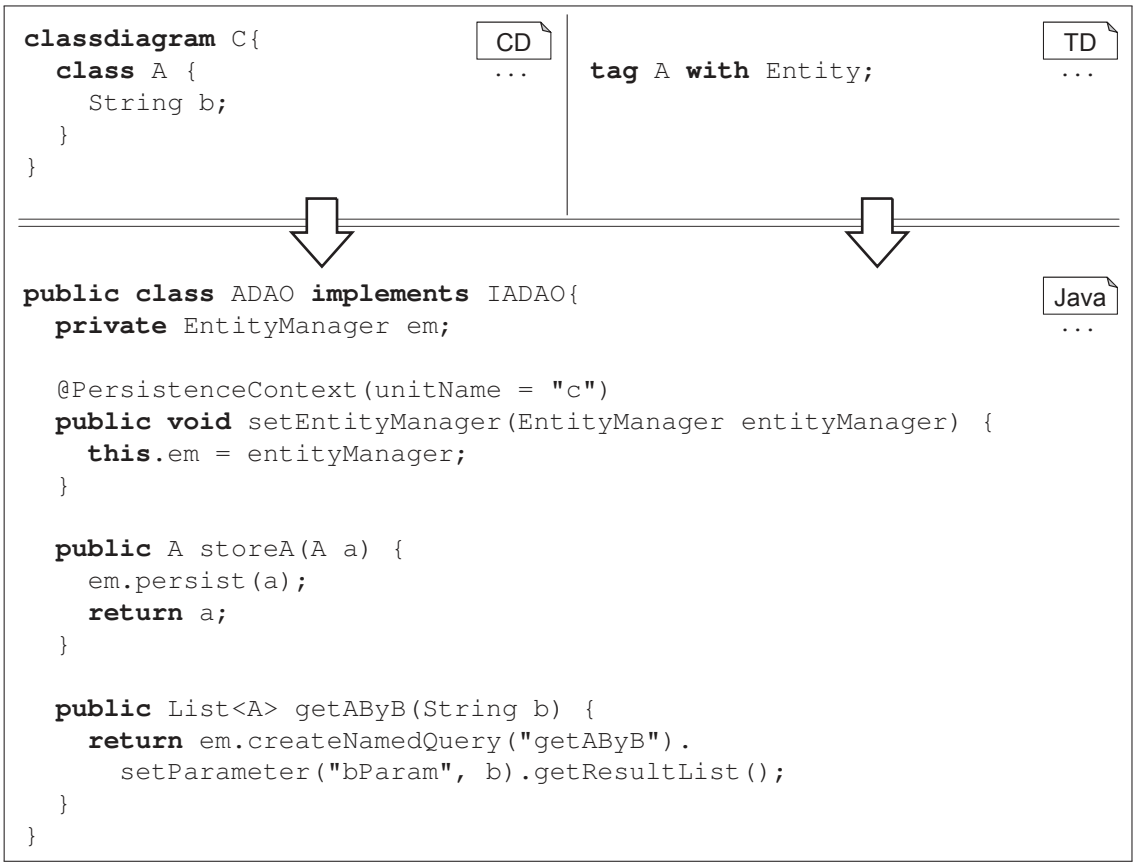


Abbildung 7.38: Implementierung der durch den DAO-Generator generierten Klasse.

Abbildung 7.37 zeigt das Interface, welches für eine Klasse mit einem Attribut vom Typ `String` generiert wird. Im Interface sind die zuvor erläuterten CRUD-Methoden gezeigt. Diese folgen dem dargestellten Namensschema. Darüber hinaus sind die Methoden zur Ausführung von Queries gezeigt. Der Name des Interfaces setzt sich aus einem vorangestellten "I", dem Klassennamen und dem angehängten Suffix "DAO" zusammen. Die Methoden selbst folgen der Namenskonvention eines führenden "get", gefolgt vom Klassennamen, gefolgt vom Infix "By" und dem Namen eines Attributs. Der Parameter erhält Namen und Typ vom entsprechend modellierten Typ.

Abbildung 7.38 zeigt die generierte Implementierungsklasse des DAOs. Sie folgt dem gleichen Namensschema ohne Präfix, wie das Interface, welches sie implementiert. Sie verwendet einen vom ORM bereitgestellten `EntityManager` zur Kommunikation mittels JDBC mit der Datenbank. Dieser `EntityManager` wird vom ORM über die dargestellte Setter-Methode gesetzt und mit Hilfe der `@PersistenceContext` Annotation vom Framework initialisiert. Dazu wird der Name der `Persistence Unit` angegeben, die in Abschnitt 10.4 erläutert wird.

Zudem ist die Implementierung der Methode zum Speichern eines Objekts dargestellt.

Sie verwendet den `EntityManager`. Die weiteren CRUD-Methoden sind analog implementiert. Zusätzlich dargestellt ist die Methode zur Ausführung des standardmäßig verfügbaren Queries. Auch diese Implementierung verwendet den `EntityManager`. Dieser kann Queries anhand ihres in der Annotation der Klasse definierten Namens ausführen. Auch hier ist, wie durch den ORM vorgegeben, die Referenz nur mit Hilfe eines Strings gegeben und gestaltet sich bei manueller Umsetzung ohne Verwendung eines Generators als fehleranfällig. Zudem wird der Parameter der Namenskonvention entsprechend gesetzt und die Liste der in der Datenbank gefundenen Objekte zurückgegeben.

Auswirkungen des `Unique` Tags

Attribute, die mit `Unique` getaggt sind, sind für alle in der Datenbank gespeicherten Instanzen der Klasse eindeutig. Dies bedeutet, dass sie, wie der Primärschlüssel dazu verwendet werden können, Objekte eindeutig über den Attributwert zu identifizieren.

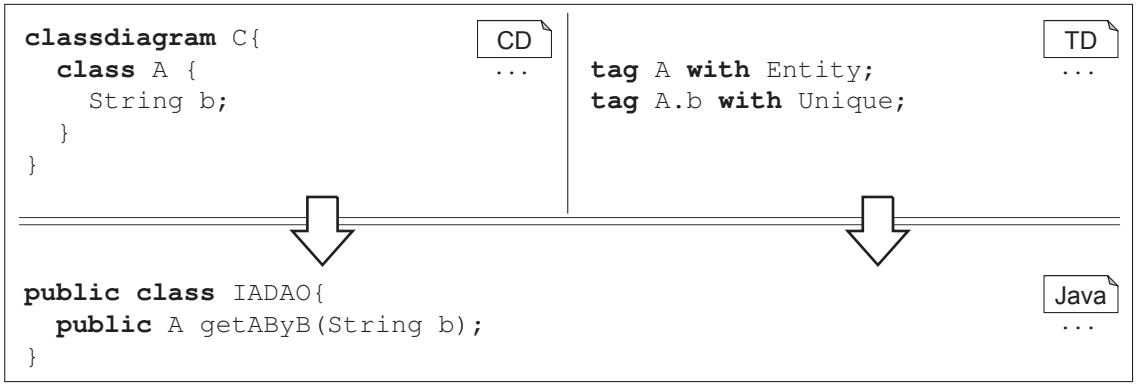


Abbildung 7.39: Auswirkungen des `Unique` Tags auf das Generat des DAO-Generators.

Abbildung 7.39 zeigt die zugehörige Methode zur Ausführung des entsprechenden Queries des generierten Interfaces. Diese unterscheidet sich durch die Verwendung des `Unique` Tags dadurch, dass keine Liste, sondern lediglich ein Objekt zurückgegeben wird. Dies ist nur durch die Zusatzinformation möglich.

Auswirkungen des `Transient` Tags

Die Verwendung des `Transient` Tags führt dazu, dass kein Query, welches ein Objekt der Klasse mit Hilfe des getaggten Attributs sucht, erzeugt wird. Da das Attribut, welches mit `Transient` getaggt ist, nicht in der Datenbank gespeichert ist, können auch keine Queries darauf zugreifen. Aus diesem Grund werden diese Attribute nicht weiter berücksichtigt.

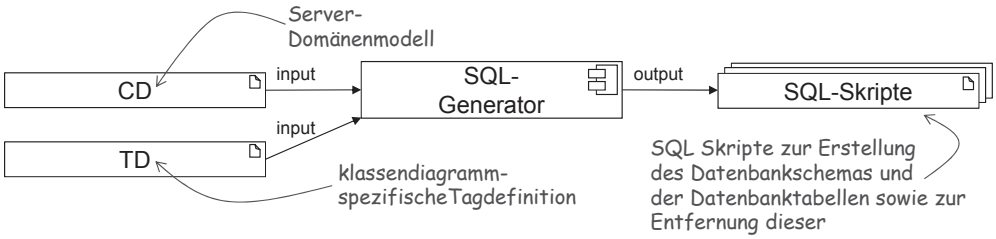


Abbildung 7.40: Darstellung der Eingabe- und Ausgabeartefakte des SQL-Generators. Er verwendet das Domänenmodell des Servers und erzeugt daraus ein SQL-Skript.

Auswirkungen des `Queries` Tags

Weitere Queries, die mit Hilfe des `Queries` Tags modelliert wurden, werden genauso, wie die bereits vorgestellten standardmäßig verfügbaren Queries, umgesetzt. Für diese wird der modellierte Name des Queries zum Namen der Methode im Generat. Darüber hinaus werden die adressierten Parameter des Queries als Methodenparameter verwendet. Der Rückgabetyp der Methode ist stets eine Liste von Objekten der zugehörigen Klasse. Bei der Implementierung wird, wie bereits bei den Annotationen in Abschnitt 7.4.1, zwischen nativen und nicht nativen Queries unterschieden. Die Unterscheidung der Queries erfolgt analog. Lediglich die Implementierung des DAOs verwendet eine andere vom `EntityManager` bereitgestellte Methode zur Ausführung des Queries.

Zusammenfassend generiert der DAO-Generator die DAOs der Servers. Diese dienen zur transparenten Kommunikation mit dem Datenbankserver. Der DAO-Generator besitzt keine eigenständigen Kontextbedingungen. Diese sind bereits durch die Kontextbedingungen des Entity-Generators, der in Abschnitt 7.4 vorgestellt wurde, abgedeckt. Zudem wurde die Berücksichtigung der modellierten und der automatisch abgeleiteten Queries gezeigt.

7.6 Der SQL-Generator

Nachdem im vorangegangenen Abschnitt der DAO-Generator detailliert vorgestellt wurde, wird in diesem Abschnitt die Generierung des Datenbankschemas vorgestellt. Zu Beginn dieser Arbeit war eine automatisierte Generierung eines Datenbankschemas auf Basis eines Klassendiagramms oder annotierter Klassen durch gängige ORMs nur prototypisch möglich und führte zu nicht menschenlesbaren Schema- und Tabellennamen. Wollte der Anwender native Queries schreiben, musste er die teilweise kryptischen Namen der Tabellen, die die Daten enthalten, kennen. Hinzu kommt, dass diese bei einer Weiterentwicklung des Datenmodells nicht änderungsresistent waren. Dem Produktentwickler gibt dies die Möglichkeit auf Basis seiner abstrakten Modellierung ein Datenbankschema zu erhalten.

Abbildung 7.40 gibt einen Überblick über den SQL-Generator. Während der Entity- und der DAO-Generator, die in den Abschnitten 7.4 und 7.5 präsentiert wurden, je-

weils mehrere Klassen erzeugen, so erzeugt der SQL-Generator ein Skript zur Erstellung des Datenbankschemas und der Erzeugung der Tabellen und ein Skript zur Entfernung dieser. Der Entity- und der DAO-Generator erzeugen technologieunabhängige Abstraktionen, der SQL-Generator hingegen ist technologiespezifisch und muss bei Verwendung eines anderen Datenbankparadigmas erweitert oder ersetzt werden.

7.6.1 Abbildung der Modellierungskonzepte auf das Generat

Der SQL-Generator verwendet Klassendiagramme als Eingabe und erzeugt valide SQL-Skripte. Generell wird ein Skript zur Erstellung eines Datenbankschemas erzeugt, das Tabellen enthält. Zusätzlich wird ein Skript zum Löschen der Tabellen und des Schemas erzeugt.

```sql
CREATE SCHEMA montiee
  AUTHORIZATION montiee;
```

Listing 7.41: Darstellung der `CREATE SCHEMA` Elemente des generierten SQL-Skripts.

Listing 7.41 zeigt die Erstellung des Schemas. Dabei wird ein standardmäßig vorkonfigurierter Schemaname sowie eine Autorisierung verwendet. In Kapitel 10 werden die Änderungsmöglichkeiten dieser Standardkonfiguration vorgestellt. Die Autorisierung gibt den Benutzernamen, mit dem der Applikationsserver auf die Datenbank über die JDBC Verbindung zugreift, an. Er ist vollständig von Benutzern der eigentlichen Applikation losgelöst und sollte nicht verwechselt werden. Allerdings wird der Name des Schemas in den vom Entity-Generator erzeugten Annotationen, wie beispielsweise der Annotation `@OneToOne`, genutzt. Bei Enterprise Applikationen müssen diese Namen übereinstimmen. Eine statische Prüfung dieser Informationen bei manueller Implementierung erfolgt nicht. Durch MontiEE wird diese Konsistenz gesichert.

Darüber hinaus wird die Erzeugung einer Tabelle generiert, die für die ID-Werte zuständig ist. Die Erzeugung der Tabelle wird generiert, falls in der Tagdefinition die `table` Option des `IDGen` Tags verwendet wird. Wie bereits in Abschnitt 7.2 erklärt, werden dazu zwei Spalten benötigt, wobei die eine einen Wert für den jeweiligen Typ speichert und die andere Spalte den fortlaufenden ID-Wert.

```sql
CREATE TABLE IdValues (
  idName TEXT,
  idValue BIGINT NOT NULL,
  PRIMARY KEY (idName)
);
```

Listing 7.42: Darstellung der SQL-Befehle zur Erzeugung der `IdValues` Tabelle des generierten SQL Skripts.

Der SQL-Generator generiert immer erst die Teile des Skripts, die unabhängig angelegt werden können. Dadurch besteht der erste Teil des Skripts aus den Anweisungen zum Anlegen des Schemas und der Tabellen und der zweite Teil des Skripts aus einem Hinzufügen von Primär- und Fremdschlüsseln sowie Bedingungen, die auf Datenbankebene definiert werden. In bestimmten Fällen kann es ansonsten bei Ausführung des Skripts zu Reihenfolgeverletzungen kommen, da sich die Konsistenzprüfung des Datenbankservers nicht deaktivieren lässt, sondern nach jedem SQL-Befehl des Skripts ausgeführt wird. Daher werden bei der Erzeugung der Tabelle im SQL-Skript keine Primär- oder Fremdschlüssel definiert, da diese sich auf Tabellen beziehen können, die bisher nicht erzeugt wurden.

Der SQL-Generator verwendet datenbankserverspezifische Datentypen, die bei Verwendung mit einem anderen Datenbankserver unter Umständen ausgetauscht werden müssen. Im Rahmen dieser Arbeit wird ein PostgreSQL-Server verwendet.

```sql
DROP TABLE /* fuer alle erzeugten Tabellen */
DROP TABLE IdValues CASCADE;
DROP SCHEMA montiee CASCADE;
```

SQL

...

Listing 7.43: Darstellung der `DROP TABLE` Elemente des generierten SQL Skripts.

Listing 7.43 zeigt das Skript zum Löschen der Datenbanktabellen und des Schemas, das ebenfalls erzeugt wird. Dabei wird für jede Tabelle, die angelegt wurde, eine `DROP` Anweisung erzeugt. Falls die `IDValues` Tabelle erzeugt wurde, wird diese ebenfalls gelöscht. Abschließend wird das Schema entfernt. Neben den allgemeinen Strukturen, die vom SQL-Generator basierend auf der Tagdefinition, erzeugt werden, werden auch die Klassendiagrammkonzepte berücksichtigt.

Auswirkungen der Klassendiagrammkonzepte

Modellierte Klassen des Klassendiagramms und die Vererbung zwischen diesen werden auf verschiedene Arten in der Datenbank repräsentiert. Dies wird im Wesentlichen durch den `Entity` und den `Inheritance` Tag gesteuert. Die Erzeugung von Tabellen für Klassen und zur Abbildung von Vererbung wird bei der Erklärung der einzelnen Tags genauer erläutert. Generell wird der Name der Tabelle, sofern für die Klasse eine Tabelle generiert wird, aus dem Namen der Klasse mit einem angehängten ”s” gebildet. Der Tabellenname wird in den vom Entity-Generator, der in Abschnitt 7.4 präsentiert wurde, erzeugten Annotationen wie der Annotation `@Table` verwendet. Durch die Generierung beider Teile kann die Konsistenz gesichert werden.

Abbildung 7.44 zeigt, dass primitive Attribute von Klassen immer als Spalten der entsprechenden Tabellen abgebildet werden. Zusätzlich wird auch das `id` Attribut, welches vom Entity-Generator generiert wird, als Spalte der Datenbanktabelle erzeugt. Die ID wird als `BIGINT` und Strings als `TEXT` gespeichert. Für komplexe Attribute werden SQL-Befehle zur Erzeugung separater Tabellen generiert. Im Rahmen dieser Arbeit wer-

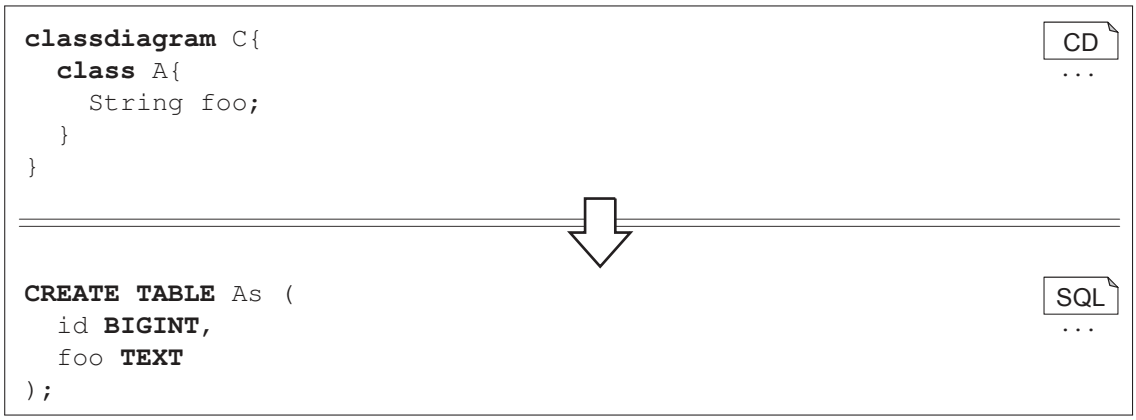


Abbildung 7.44: Abbildung einer Klasse im Generat des SQL-Generators.

den komplexe Attribute immer mit Hilfe von Assoziationen modelliert. Daher werden hier nur Attribute, die sich auf eine einfache Datenbankspalte abbilden lassen, berücksichtigt. Hierbei ist es wichtig, dass der Name der Spalte, in der der Attributwert gespeichert wird, dem Namen des Klassenattributs, da der ORM dies als Konvention erwartet, entspricht.

Assoziationen hingegen werden abhängig von ihrer Zielkardinalität entweder als Joinspalte oder als -tabelle abgebildet. Dabei werden auf Ebene der Datenbank keine Navigationsrichtungen unterschieden, da Joins immer in beide Richtungen möglich sind.

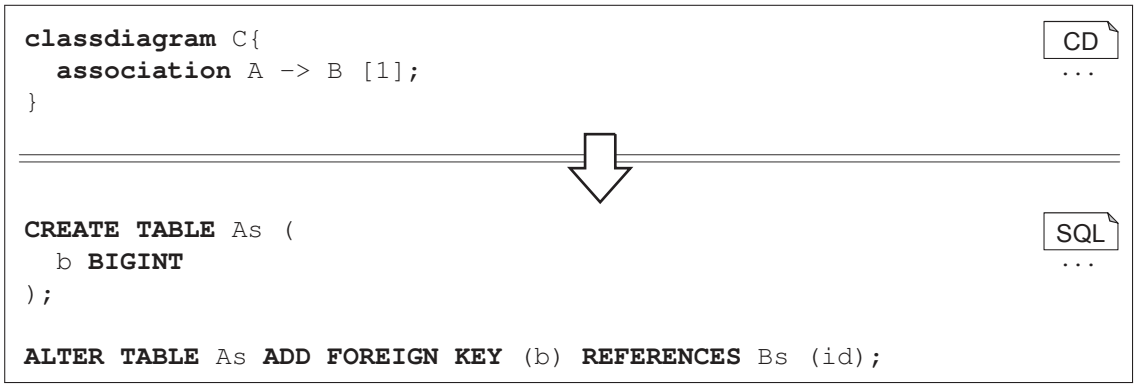


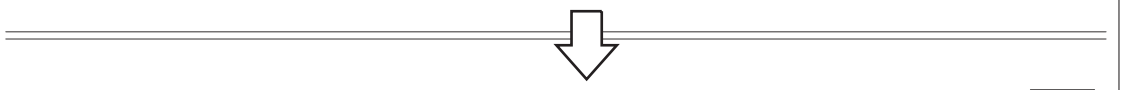
Abbildung 7.45: Abbildung einer `1-zu-1` oder `*-zu-1` Assoziation im Generat des SQL-Generators.

Listing 7.45 zeigt den vom SQL-Generator generierten Teil des Skripts für eine Joinspalte. Eine Joinspalte wird immer dann verwendet, wenn die Zielkardinalität der Assoziation `1` entspricht. Die Startkardinalität ist dabei nicht relevant. Allerdings kann bei einer `1-zu-1` Assoziation die Joinspalte in eine der beiden Tabellen generiert werden, wohingegen die Joinspalte bei einer `*-zu-1` Assoziation zu der Klasse der `*` Kardinalität gehört. Bei einer unidirektionalen `1-zu-1` Assoziation wird die Tabelle mit der

Joinspalte durch die Navigationsrichtung, bei einer bidirektionalen 1-zu-1 Assoziation durch den Owner Tag, da dies der natürlichen Semantik der modellierten Assoziation entspricht, bestimmt. Die Joinspalte enthält die ID des zugehörigen Objekts. Der Name der Joinspalte entspricht dem Namen der Assoziationen oder, falls dieser nicht vorhanden ist, dem Klassennamen des Assoziationsziels. Neben dem Hinzufügen der Joinspalte wird ein Fremdschlüssel, der besagt, dass der Inhalt der Joinspalte sich auf die ID-Spalte der Tabelle der Zielklasse der Assoziation bezieht, hinzugefügt. Die Joinspalte wird von der Annotation @JoinColumn, die vom Entity-Generator erzeugt wird, konsistent referenziert.

CD

```
classdiagram C{
  association A -> B [*];
}
```



SQL

```
CREATE TABLE AsBs (
  as BIGINT,
  bs BIGINT
);

ALTER TABLE AsBs ADD PRIMARY KEY (as,bs);
ALTER TABLE AsBs ADD FOREIGN KEY (as) REFERENCES As (id);
ALTER TABLE AsBs ADD FOREIGN KEY (bs) REFERENCES Bs (id);
```

Abbildung 7.46: Abbildung einer 1-zu-* oder *-zu-* Assoziation im Generat des SQL-Generators.

Abbildung 7.46 zeigt das Verhalten des SQL-Generators bei 1-zu-* oder *-zu-* Assoziationen. Hier wird anstelle einer Joinspalte eine eigenständige Tabelle, die zwei Spalten enthält, erzeugt. Diese referenzieren die ID-Attribute der Tabellen für beide Seiten der Assoziation. Der Primärschlüssel dieser Tabelle setzt sich als Tupel aus beiden Spalten zusammen. Zudem werden zwei Fremdschlüssel, die die Referenz auf die jeweiligen Tabellen der Assoziationselemente sichern, erzeugt. Der Name der Tabelle setzt sich aus dem Assoziationsnamen oder, falls dieser nicht vorhanden ist, den beiden assoziierten Klassen mit jeweils angehängtem "s" zusammen. Dieser Name wird in der Annotation @JoinTable des Entity-Generators referenziert. Die Erstellung der Jointabellen ist unabhängig von der gewählten Option des Inheritance Tags. Nicht unabhängig hingegen ist die Erzeugung des Fremdschlüssels, da dieser den Namen der entsprechenden Tabellen verwendet. Die Verwendung geordneter Assoziationen wird im weiteren Verlauf mit Hilfe des Ordered Tags erläutert. Enumerationen werden als Spalten in der Tabelle der Klassen, die die Enumeration verwendet, abgebildet. Die weiteren Elemente des Klassendiagramms werden nicht in der Datenbank repräsentiert.

Auswirkungen des `Entity` Tags

Zunächst werden nur für Elemente, die mit dem `Entity` Tag getaggt wurden, SQL-Befehle zur Erzeugung einer Tabelle angelegt. Darüber hinaus werden für Elemente innerhalb einer Vererbungshierarchie nur dann SQL-Befehle zur Erzeugung eigener Tabellen angelegt, wenn die `tablePerClass` Option gewählt wurde. Wurde eine andere Strategie gewählt, werden SQL-Befehle zur Erzeugung einer Tabelle erstellt, die die Vereinigung aller Attribute der Subklassen sowie die Diskriminatorspalten enthält. Ebenso werden die Attribute von abstrakten Klassen in den Tabellen der Subklassen abgebildet.

Auswirkungen des `Inheritance` Tags

Der `Inheritance` Tag bestimmt, für welche modellierten Klassen Datenbanktabellen erzeugt werden. Dazu wird für die Option `tablePerClass` für jede nicht abstrakte Klasse einer Vererbungshierarchie eine eigene Tabelle generiert. Dabei enthält die Tabelle für eine modellierte Klasse immer alle Attribute aller Superklassen, unabhängig davon, ob diese Klassen abstrakt sind. Der Name der generierten Tabelle wird von der Annotation `@Table` des Entity-Generators referenziert.

Abbildung 7.47: Abbildung des `Inheritance` Tags mit der `singleTable` Option im Generat des SQL-Generators.

Abbildung 7.47 zeigt die Auswirkungen der `singleTable` Option auf das generierte SQL-Skript. Es wird eine `discriminator` Spalte für die Tabelle, in der disjunkte Werte für jede Subklasse in der Vererbungshierarchie gespeichert werden, generiert. Der identifizierende Wert einer Subklasse wird über die Annotation `@DiscriminatorValue` des Entity-Generators angegeben. Ohne einen generativen Ansatz muss manuell gesichert werden, dass keine doppelten Werte bei unterschiedlichen Subklassen verwendet werden. Gleichzeitig wird der Name der Spalte von der `@DiscriminatorColumn` Annotation referenziert. Auch dies muss in einem nicht generativen Kontext manuell konsistent gehalten werden. Der Name der generierten Tabelle folgt dem gleichen Namensschema, dem alle Tabellennamen folgen. Ihm wird die oberste Klasse einer Vererbungshierarchie als Ausgangspunkt für die Namensbildung zu Grunde gelegt. Die Tabelle erhält zusätzlich Spalten für alle Attribute aller Subklassen der Vererbungshierarchie. Attribute einer zu speichernden Klasse werden in den entsprechenden Spalten gespeichert. Spalten für

Attribute anderer Klassen der Vererbungshierarchie werden mit `null` belegt. Dies führt zu einer Vielzal von `null` Werten in der Datenbank. Die `joined` Option wird analog abgebildet. Ebenso enthält die Tabelle für eine Klasse immer auch eine Spalte `id`, in der der Primärschlüssel abgelegt wird. Die Generierung der entsprechenden Spalte folgt der gleichen Logik des Entity-Generators zur Bestimmung der Klasse, die das `id` Attribut enthält.

Auswirkungen des `IDGen` Tags

Der `IDGen` Tag bewirkt, dass die `IdValues` Tabelle generiert wird. Hierbei wird die Tabelle nicht für ein einzelnes Attribut oder eine Assoziation erstellt, sondern global für das gesamte Datenbankschema. Alle Entitäten, die als Strategie zur Generierung der ID die Option `table` benutzen, verwenden die `IdValues` Tabelle zur Speicherung ihrer IDs. Für den SQL-Generator reicht es aus, dass ein Element existiert, welches diese Option verwendet. Nur für den Fall, dass kein Element existiert, wird die Tabelle nicht generiert.

Auswirkungen des `Unique` Tags

Der `Unique` Tag zeichnet ein Attribut oder ein Assoziationsende als eindeutig innerhalb des Systems aus.

Abbildung 7.48: Abbildung des `Unique` Tags im Generat des SQL-Generators.

Abbildung 7.48 zeigt die Auswirkungen auf das Generat. Für ein Attribut wird in der generierten Tabelle das Schlüsselwort `UNIQUE` verwendet. Dies bewirkt, dass die Datenbank beim Einfügen prüft, ob der Wert bereits innerhalb der Attributspalte vorhanden ist und dementsprechend einen Fehler meldet. Dies gilt auch für die generierte Joinspalte falls der `Unique` Tag bei einem Assoziationsende verwendet wird.

Auswirkungen des `NotNull` Tags

Der `NotNull` Tag bewirkt, dass ein Attribut immer belegt sein muss.

Abbildung 7.49: Abbildung des `NotNull` Tags im Generat des SQL-Generators.

Abbildung 7.49 zeigt den zugehörigen Teil des SQL-Skripts. Dort wird ein Attribut mit dem Schlüsselwort `NOT NULL` markiert. Dies bedeutet, dass beim Einfügen immer ein Wert für das Attribut vorhanden sein muss. Ebenso gilt dies für Assoziationsenden, so dass die zugehörige Joinspalte entsprechend markiert wird.

Auswirkungen des `Transient` Tags

Der `Transient` Tag bedeutet semantisch, dass das Attribut, welches mit diesem getaggt ist, nicht in der Datenbank gespeichert wird. Aus diesem Grund bewirkt die Verwendung dieses Tags, dass für dieses Attribut keine zugehörige Spalte in der Tabelle der modellierten Klasse generiert wird. Ebenso wird bei Assoziationen, deren Assoziationsenden mit dem `Transient` Tag getaggt sind, verfahren. Bei `1-zu-1` oder `*-zu-1` Assoziationen wird keine Joinspalte und bei `1-zu-*` oder `*-zu-*` Assoziationen keine Jointabelle generiert.

Auswirkungen des `Ordered` Tags

Der `Ordered` Tag bewirkt, dass eine zusätzliche Spalte zur Sortierung generiert wird. Der Name der Spalte ergibt sich aus dem Namen der Assoziation oder, falls dieser nicht vorhanden ist, aus dem Zieltyp der Assoziation und dem angehängten Suffix "_ordered". Der Name dieser Spalte wird von der Annotation `@OrderColumn` des Entity-Generators referenziert. Mit Hilfe dieser Spalte wird die Sortierung der Liste in Form von Indices beim Speichern mengenwertiger Assoziationen ebenfalls gespeichert und kann somit beim Laden wiederhergestellt werden.

Die weiteren Tags haben keine Auswirkungen auf das Generat des SQL-Generators. Insbesondere der `Queries` Tag wirkt sich nur auf das Generat des Entity-Generators aus. Nachdem in diesem Abschnitt das Generat des SQL-Generators im Detail vorgestellt wurde, werden nachfolgend die Kontextbedingungen des SQL-Generators vorgestellt.

7.6.2 Kontextbedingungen

Der SQL-Generator besitzt wie der Entity-Generator Kontextbedingungen die aus der Verwendung der Tagdefinition resultieren.

Die folgenden drei Kontextbedingungen sind sich im Grunde sehr ähnlich und beziehen sich auf die Verwendung der `Unique`, `NotNull` und `Transient` Tagtypen. Sie fordern bestimmte Eigenschaften einer Assoziation.

MontiEE-1

Bedingung: Wird der `Unique` Tag bei einer Seite einer Assoziation verwendet, muss diese Assoziation unidirektional, die Zielkardinalität `1` und der Tag an der Zielseite der Assoziation verwendet worden sein.

Schweregrad: Fehler

MontiEE-2

Bedingung: Wird der `NotNull` Tag bei einer Seite einer Assoziation verwendet, muss diese Assoziation unidirektional, die Zielkardinalität `1` und der Tag an der Zielseite der Assoziation verwendet worden sein.

Schweregrad: Fehler

MontiEE-3

Bedingung: Wird der `Transient` Tag bei einer Seite einer Assoziation verwendet, muss diese Assoziation unidirektional, die Zielkardinalität `1` und der Tag an der Zielseite der Assoziation verwendet worden sein.

Schweregrad: Fehler

Zusammenfassend generiert der SQL-Generator SQL-Skripte zur Initialisierung und zum Löschen der Datenbank. Sowohl das Schema als auch die Tabellen und benötigte Infrastrukturen, wie eine ID-Tabelle, werden erzeugt. Dies hilft dem Produktentwickler bei der Umsetzung des Systems. Darüber hinaus wurden die Kontextbedingungen zur Konsistenzsicherung des Generators vorgestellt.

7.7 Zusammenfassung

In diesem Kapitel wurden die MontiEE-Generatoren zur Generierung der Persistenz von Enterprise Applikationen detailliert vorgestellt.

Zunächst wurde in Abschnitt 7.2 ein MontiEE-spezifisches Tagschema, welches die Grundlage der von MontiEE umgesetzten Generatoren bildet, vorgestellt. Dazu wurden alle umgesetzten Tagtypen einzeln detailliert vorgestellt und ihre Semantik erläutert.

In Abschnitt 7.3 wurde die Generierung und Befüllung der TagAPI vorgestellt. Diese verläuft in zwei Schritten zu unterschiedlichen Zeitpunkten im Buildprozess. Zunächst wird auf Basis des Tagschemas die Infrastruktur generiert und auf Basis der Tagdefinition befüllt. Dabei wurden die sprachspezifischen Komponenten und die Komponenten der Laufzeitumgebung detailliert vorgestellt.

In Abschnitt 7.4 wurde der Entity-Generator vorgestellt, der das Domänenmodell des Servers erzeugt. Er verwendet dazu Klassendiagramme und erzeugt JPA konforme Entitäten. Die Auswirkung der unterschiedlichen Tags auf das Generat wurden detailliert erläutert. Darüber hinaus wurde die Möglichkeit zur Erweiterung des Generats durch Delegatoren und Kontextbedingungen des Generators vorgestellt.

In Abschnitt 7.5 wurde der DAO-Generator, der die Data Access Objects des Servers generiert, vorgestellt. Diese dienen als Kommunikationsschnittstelle zwischen Applikations- und Datenbankserver. Auch hier wurden die Auswirkungen der einzelnen Tags detailliert erläutert.

In Abschnitt 7.6 wurde der SQL-Generator vorgestellt, der ein SQL-Skript zur Initialisierung des Datenbankschemas, der Tabellen und der benötigten Infrastruktur generiert. Dabei wurden die Auswirkungen detailliert für alle Tags erläutert. Zudem wurden Kontextbedingungen vorgestellt.

Mit Hilfe dieser MontiEE-Generatoren kann der Produktentwickler einen Großteil der Persistenz von Enterprise Applikationen generieren. Erweiterungen können manuell hinzugefügt werden. Im nächsten Kapitel werden die Generatoren zur Generierung der Kommunikationsinfrastruktur von Enterprise Applikationen vorgestellt.

Kapitel 8

Generierung der Kommunikationsinfrastruktur

Nachdem im vorangegangenen Kapitel die Generierung der Persistenz von Enterprise Applikationen vorgestellt wurde, fokussiert dieses Kapitel auf die Möglichkeiten des Produktentwicklers zur Generierung der Kommunikationsinfrastruktur von Enterprise Applikationen.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Generierung von DTOs, Requests und DTOs-Assemblern auf Basis der Sichten.
- Der BusinessAPI-Generator zur Generierung einer definierten API von der Kommunikationsfassade zur Geschäftslogik und von der Geschäftslogik zur Persistenz.
- Der Facade-Generator zur Generierung der Kommunikationsfassaden unter Einbezug der Sichten, Rechte-, Rollen- und Mappingdiagramme.

Zunächst wird der DTO-Generator vorgestellt. Daran anschließend werden der BusinessAPI- und der Facade-Generator vorgestellt. Anschließend wird dann in Kapitel 9 die Generierung der Infrastruktur zur Evolution des Systems vorgestellt. In Kapitel 10 wird die Methodik der Verwendung der Generatoren, deren Konfiguration und die Umsetzung des Szenarios vorgestellt.

8.1 Überblick

Abbildung 8.1 zeigt erneut die im Rahmen von MontiEE umgesetzten Generatoren. Im vorangegangenen Kapitel wurde bereits der Entity-, der DAO- und der SQL-Generator in den Abschnitten 7.4, 7.5 und 7.6 vorgestellt. Darüber hinaus wurden die Generierung sowie die Befüllung der TagAPI vorgestellt.

In diesem Kapitel werden der DTO-, der BusinessAPI- und der Facade-Generator in den Abschnitten 8.2, 8.3 und 8.4 vorgestellt. Der Delta-Generator wird in Kapitel 9 vorgestellt. Der DTO-Generator generiert auf Basis von Klassendiagrammen und der TagAPI DTOs für die Kommunikation mit Clients. Seine Eingabemodelle sind normalerweise transformierte Sichten. Aber auch das Domänenmodell kann zur Generierung der DTOs verwendet werden. Dadurch können vollwertige Clients die generierte Kommunikationsinfrastruktur verwenden. Zusätzlich zu den DTOs werden Requests generiert. Neben den

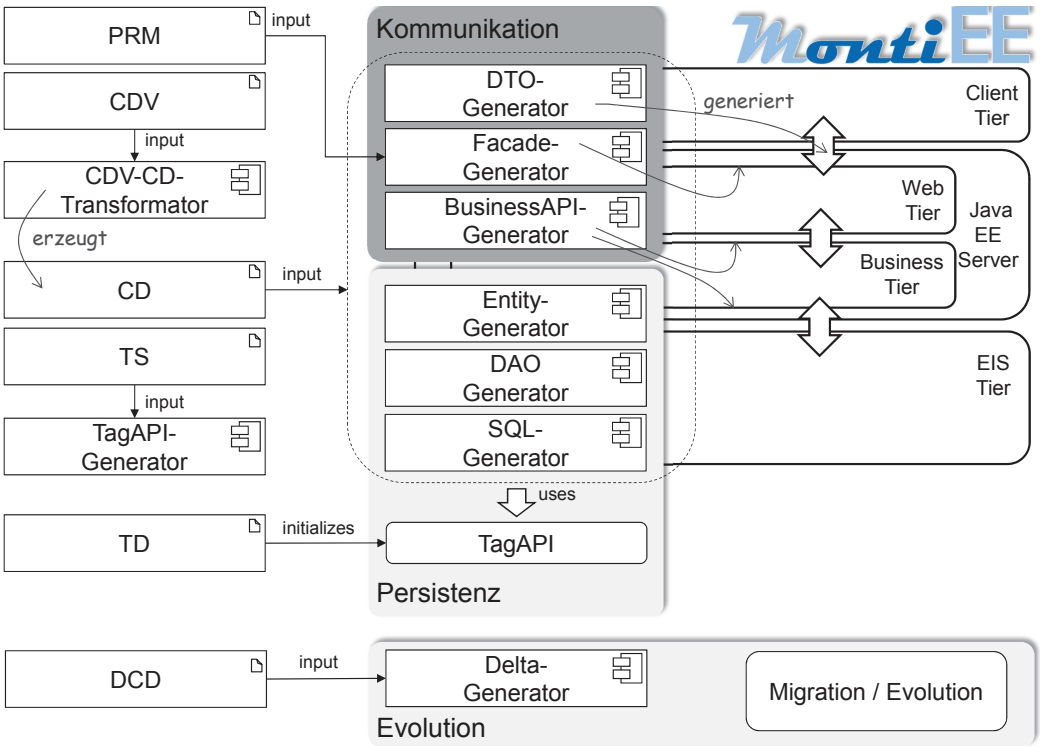


Abbildung 8.1: Übersicht der in MontiEE umgesetzten Generatoren. Gezeigt sind die Eingabemodelle sowie die Teile der Enterprise Applikation, die mit Hilfe der Generatoren generiert werden. Der Fokus liegt dabei auf den Generatoren zur Generierung der Kommunikationsinfrastruktur von Enterprise Applikationen.

DTOs und den Requests werden auch Assembler zur Instanziierung der DTOs aus den eigentlichen Entitäten generiert. Der DTO-Generator generiert die Kommunikationsinfrastruktur sowie die zwischen Client Tier und Web Tier kommunizierten Elemente. Er wird in Abschnitt 8.2 genauer vorgestellt.

Der BusinessAPI-Generator erhält ebenfalls die gleichen Eingaben wie die zuvor genannten Generatoren. Er generiert eine API zur Kapselung der Applikationslogik, so dass der Entwickler des Systems von dem Technologiestack der Enterprise Applikation abstrahierend die Anwendungslogik umsetzen kann. So lange er sich an die API hält oder diese erweitert, muss er die dahinterliegende Technologie nicht kennen. Der BusinessAPI-Generator generiert die Schnittstelle zwischen Web Tier und Business Tier. Er wird in Abschnitt 8.3 detailliert vorgestellt. Der letzte Generator ist der Facade-Generator, der die Webservicefassade generiert. Er erhält Klassendiagramme, sowohl das Domänenmodell als auch die aus den Sichten, die in Kapitel 5 präsentiert wurden, generierten Klassendiagramme, verwendet die TagAPI und das Mappingdiagramm als Eingabe und generiert daraus Java-Code, der die Webservice-, bzw. RPC-Fassaden (vgl. FA4-WE und FA5-WE) darstellt. Dabei kann für jedes Klassendiagramm, unabhängig ob es sich um

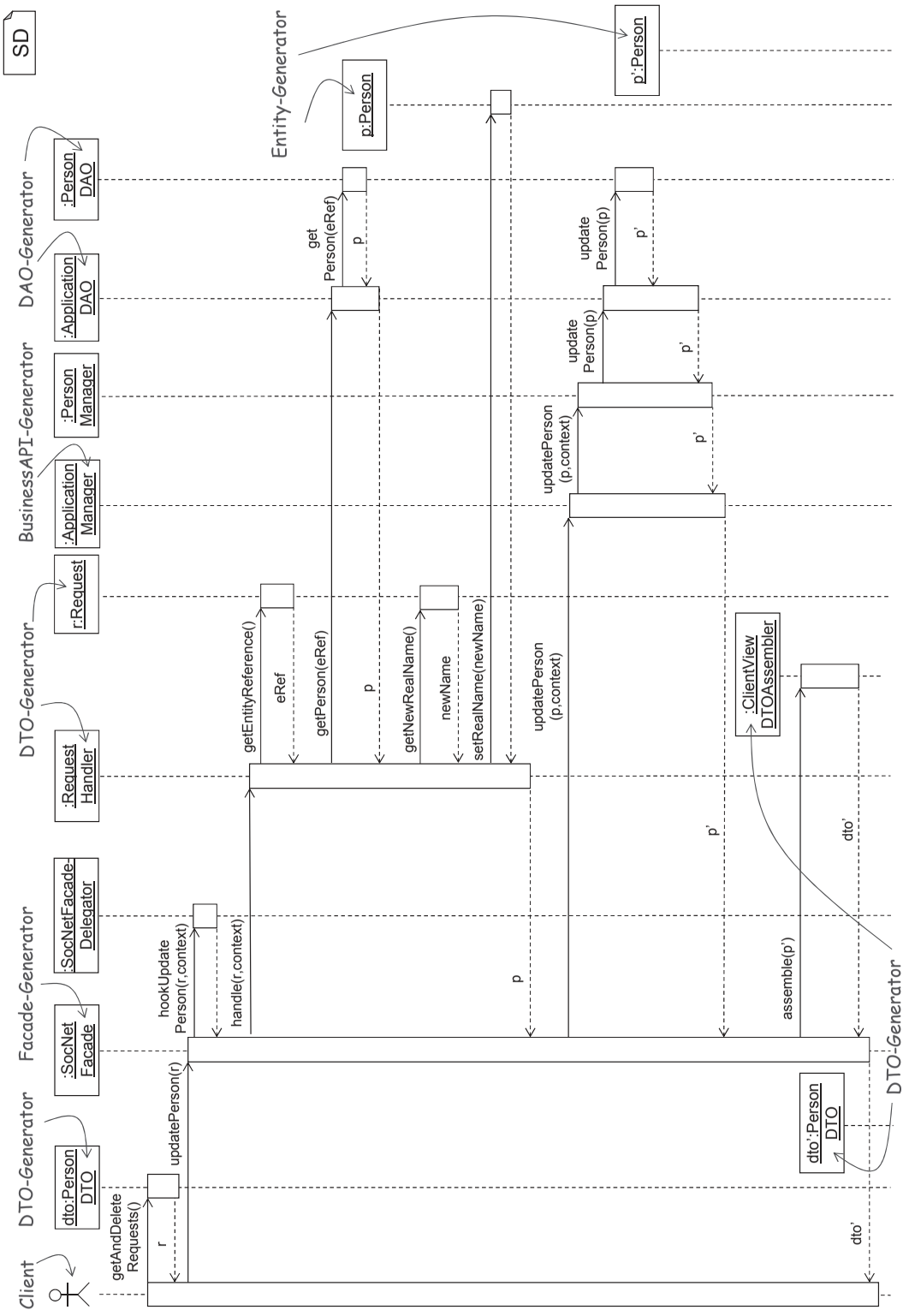


Abbildung 8.2: Darstellung eines exemplarischen Ablaufs der Aktualisierung eines Objekts vom Client bis zur Datenbank mit Hilfe des MontiEE-Generats.

das Domänenmodell oder ein aus einer Sicht transformiertes Klassendiagramm handelt, je ein Mapping angegeben werden, so dass für jeden Client unterschiedliche Fassaden entstehen können. Er generiert die Schnittstelle des Web Tiers, die von den Clients zur Kommunikation verwendet wird. Er wird in Abschnitt 8.4 im Detail vorgestellt. Abbildung 8.2 zeigt einen exemplarischen Ablauf einer Aktualisierung eines Objekts in der Datenbank auf Basis des Szenarios des sozialen Netzwerks. Das dargestellte Sequenzdiagramm zeigt den vollständigen Aufruf, der vom Client ausgelöst bis zur Datenbank durch die gesamte Serverarchitektur geleitet wird. Dabei werden die möglichen Erweiterungspunkte berücksichtigt, die Verwendung von Requests gezeigt und die bereits zuvor vorgestellte Verwendung der DAOs dargestellt. Der dargestellte Akteur ist der Client. Dieser verwendet die `SocNetFacade` zur Kommunikation mit dem Server. Der Client ruft die `updatePerson(r)` Methode der Fassade auf und übergibt ihr eine Menge von `Requests`. Die Fassade ruft die `hookUpdatePerson(p, context)` Methode des Delegators angereichert um den Sessioncontext `context` auf. Dieser Delegator kann handgeschrieben werden und dient dem Einbezug von handgeschriebener Funktionalität. Nach dem Delegator löst die Fassade mit Hilfe des `RequestHandler` die zu den Requests gehörende Entität auf, wie Abbildung 8.2 zeigt. Dies erfolgt auf Basis der Entitätenreferenz `eRef`, die über die `getEntityReference()` Methode der Requests bekannt ist. Der `RequestHandler` verwendet direkt das `ApplicationDAO`, in dem er mit Hilfe der `eRef`, die `getPerson(eRef)` Methode aufruft. Das `ApplicationDAO` verwendet das `PersonDAO` und lädt mit Hilfe der `getPerson(eRef)` Methode die Entität `p`, die dem `RequestHandler` zurückgegeben wird. Dieser setzt auf Basis des `Requests` den neuen Namen und gibt die Entität an die `SocNetFacade` zurück. Die Fassade ruft dann die `updatePerson(p,context)` des `ApplicationManager`, die die `updatePerson(p,context)` Methode des `PersonManager` aufruft, auf. Der `PersonManager` muss handgeschrieben werden und dient der Integration der Geschäftslogik. Der Manager verwendet die `updatePerson(p)` Methode des `ApplicationDAO` zur Aktualisierung der Person in der Datenbank. Das `ApplicationDAO` ruft die `updatePerson(p)` des `PersonDAO` auf, das die Person in der Datenbank speichert und ein neues Objekt zurückgibt. Dieses Objekt wird, wie in Abbildung 8.2 dargestellt, bis zur `SocNetFacade` zurückgegeben. Die `SocNetFacade` verwendet den `ClientView-DTOAssembler`, um aus dem Personenobjekt wieder ein DTO zu erstellen. Das DTO wird an den Client zurückgegeben, so dass dieser mit diesem Objekt weiterarbeitet. Abbildung 8.2 zeigt zudem, welche Klassen der dargestellten Objekte generiert werden. Die einzelnen Generatoren werden nachfolgend vorgestellt. Das Kapitel ist wie folgt strukturiert: Zunächst wird der DTO-Generator vorgestellt. Dabei werden die DTOs, Requests und Assembler präsentiert. Daran anschließend wird der BusinessAPI-Generator, gefolgt vom Facade-Generator, vorgestellt. Anschließend wird in Kapitel 9 die Generierung der Infrastruktur zur Evolution des Systems vorgestellt. In Kapitel 10 werden die Methodik der Verwendung der Generatoren und deren Konfiguration vorgestellt.

8.2 Der DTO-Generator

In diesem Abschnitt wird der DTO-Generator vorgestellt. Der DTO-Generator erzeugt Data Transfer Objects nach dem DTO-Pattern [Fow03], die zur Kommunikation mit unterschiedlichen Clients verwendet werden und dessen Domänenmodell darstellen (vgl. FA7-AG, FA7.1-AG und FA10-WE). Teile des Generators wurden in der betreuten Vorarbeit [Wol12] implementiert. Zusätzlich beinhalten die DTOs Requests, die vom Server verarbeitet werden können. Diese Request sind entfernt an das Request-Reply Pattern [HW04b] angelehnt, in dem sie an den Server versendet und dort verarbeitet werden (vgl. FA11-WE). Ein expliziter Reply wurde im Rahmen dieser Arbeit allerdings nicht umgesetzt, sondern die Rückgabewerte der Methoden, oder ihre geworfenen Exceptions, dienen als Antworten an den Client. Zur Laufzeit des Systems werden die DTOs aus, den aus der Datenbank geladenen Entitäten instanziiert und dem Client zur Verfügung gestellt. Der Client verwendet die DTOs und führt Operationen auf ihnen aus. Dadurch werden intern im DTO eine Menge von Requests erzeugt, die wiederum zum Server kommuniziert werden. Die Requests enthalten Modifikationen, die auf den Entitäten ausgeführt werden müssen. Dazu generiert der DTO-Generator die DTO-Klassen, DTO-Assembler Klassen, die die Umkapselung vornehmen, und Request Klassen, die die Ausführung einer Operation auf dem Server bewirken. Dies ermöglicht es dem Produktentwickler für verschiedene Arten von Clients Domänenmodelle, die an die Clients angepasst sind, zu generieren (vgl. FA7.1-AG und FA7.2-AG). Darüber hinaus erhält er Infrastruktur zur Verwendung und Verarbeitung von DTOs, die er intern bei der Entwicklung des Systems aus handgeschriebenem Quellcode heraus verwenden kann.

Abbildung 8.3 zeigt eine Übersicht der Ein- und Ausgabe des DTO-Generators. Er verwendet Modelle der Klassendiagrammsprache als Eingabe. Diese können entweder das Domänenmodell des Servers oder aber die aus den Sichten erzeugten Klassendiagramme sein. Die Erzeugung eines Klassendiagramms aus einer Sicht wurde in Abschnitt 5.2.3 erläutert. Für jede mögliche Art eines Clients modelliert der Produktentwickler eine Sicht auf das Server-Domänenmodell und definiert dadurch ein Domänenmodell für den Client. Das Server-Domänenmodell, wie auch die aus den einzelnen Sichten transformierten Klassendiagramme, können Klassen gleichen Namens enthalten. Dies führt dazu, dass die generierten DTO-Klassen den gleichen Namen haben, auch wenn sie mit einem Suffix angereichert werden. Die Sichten und auch das Domänenmodell sind daher in Paketen organisiert. Der DTO-Generator erzeugt zu jeder Sicht und zu dem Domänenmodell DTOs. Dies bedeutet, dass für jedes Modell eine Menge an Klassen generiert wird, die innerhalb des Systems zur Verfügung stehen. Somit existieren DTOs für das Domänenmodell und DTOs für jede Art von Client. Im weiteren Verlauf werden diese DTOs zur Kommunikation verwendet. Dabei werden die DTOs, die auf dem Domänenmodell basieren, über eine eigenständige vom Facade-Generator generierte Fassade an einen vollwertigen Client versendet. Andere Arten von Clients nutzen speziell für sie vom Facade-Generator, der in Abschnitt 8.4 gezeigt wird, generierte eigenständige Fassaden, die die DTOs verwenden, die aus ihrer spezifischen Sicht generiert wurden. Dies bedeutet, dass im generierten System mehrere Fassaden existieren, die mit den unterschiedlichen Arten von

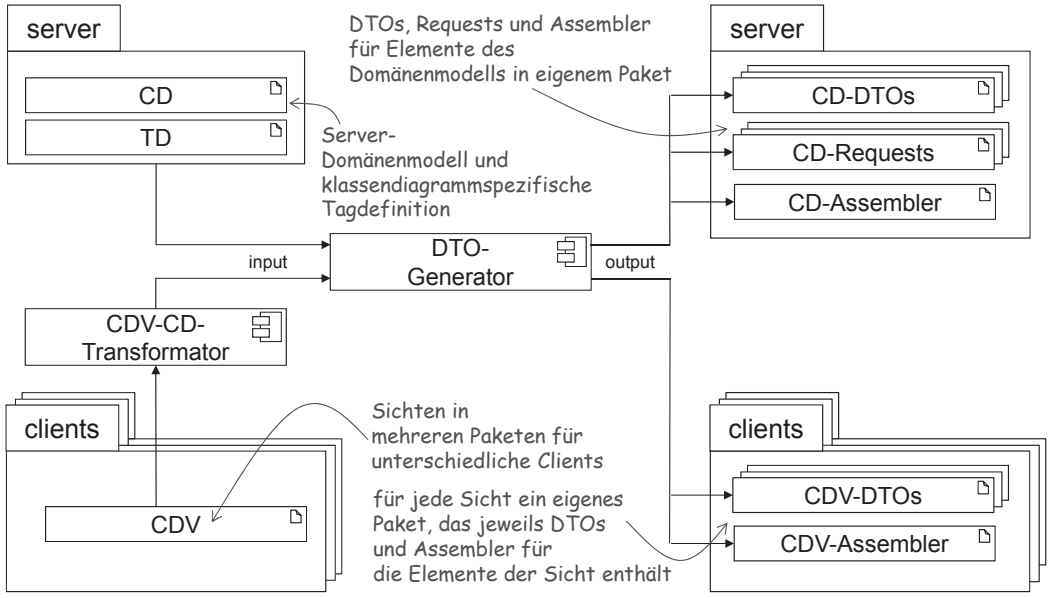


Abbildung 8.3: Darstellung der Eingabe- und Ausgabeartefakte des DTO-Generators. Er verwendet das Domänenmodell des Servers und in Klassendiagramme transformierte Sichten und erzeugt daraus mehrere DTOs, die in unterschiedlichen Paketen organisiert sind. Zudem werden Assembler für alle Eingaben und Requests für das Domänenmodell des Servers generiert.

Clients kommunizieren und die jeweils unterschiedlichen generierten DTOs verwenden. Die DTOs selbst werden aus den Entitäten, die vom Entity-Generator, der in Abschnitt 7.4 präsentiert wurde, erzeugt werden, instanziiert. Dazu werden DTO-Assembler generiert. Es wird für jede Sicht, die in Kapitel 5 vorgestellt wurde, und für das Domänenmodell je ein DTO-Assembler generiert. Für jedes Element des Server-Domänenmodells existieren Requests. Diese werden nur auf Basis des Server-Domänenmodells generiert, da die Sichten immer einen Teil des Klassendiagramms beinhalten. Die DTOs der Sichten verwenden intern ebenfalls die Requests, die aus dem Server-Domänenmodell generiert werden. Nachfolgend werden die Details der Generierung der DTOs erläutert.

8.2.1 Abbildung der Modellierungskonzepte auf das Generat

Der DTO-Generator erstellt für jede Klasse eines Klassendiagramms ein zugehöriges DTO. Dazu zeigt Abbildung 8.4 die erzeugten DTOs basierend auf einer in ein Klassendiagramm transformierten Sicht. In der Sicht wurde das Attribut `foo` der Klasse `A` weggelassen. Zudem wurde die Assoziation zwischen der Klasse `C` und der Enumeration `D` als flache Assoziation modelliert.

Der DTO-Generator erzeugt drei DTOs: die abstrakte Klasse `ADTO` und die konkreten Klassen `BDTO` und `CDTO`. Basierend auf der Sicht wird in den generierten Klassen `ADTO`

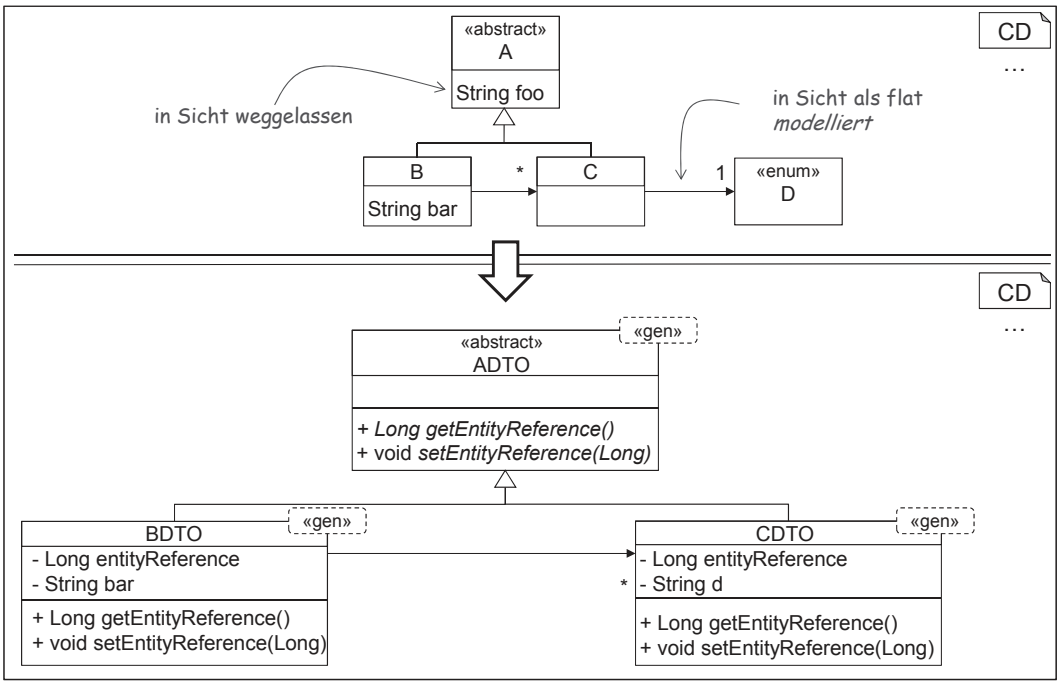


Abbildung 8.4: Exemplarische Darstellung des Generats des DTO-Generators basierend auf einem Klassendiagramm.

kein Attribut generiert. Die Klasse `CDTO` erhält wegen der flachen Assoziation ein zusätzliches Attribut. Darüber hinaus haben die konkreten DTOs Getter- und Setter-Methoden für das zusätzliche Attribut `Long entityReference` während die abstrakten DTOs nur die abstrakten Methoden erhalten. Dieses Attribut referenziert die zugehörige in der Datenbank persistierte Entität, welche zu dem entsprechenden DAO gehört. Getter- und Setter-Methoden der Attribute sind in Abbildung 8.4 nicht dargestellt.

Die generierte DTO-Struktur erlaubt es, Daten dieser Form zu Clients zu transportieren, so dass diese sie als Domänenmodell verwenden und verarbeiten können. Im Folgenden wird der erzeugte Quellcode des Generators für ein beliebiges Klassendiagramm vorgestellt, auch wenn er typischerweise mehrere Klassendiagramme verarbeitet und das Generat in Paketen, für jede Sicht und das Domänenmodell des Servers, unterschiedlich organisiert. Dabei folgt die Generierung der Logik des Entity-Generators. Dieser generiert für jedes modellierte Attribut eine Getter- und Setter-Methode. Zusätzlich werden Assoziationen als Attribute oder als Mengen von Attributen dargestellt. Die vom Entity-Generator verwendeten Persistenzannotationen werden vom DTO-Generator nicht verwendet.

Der DTO-Generator generiert eigene Annotationen zur Serialisierung und zum Transfer der DTOs über Webservices, wie in Abbildung 8.5 dargestellt. Diese Annotation muss als Parameter alle Subklassen der Klasse enthalten. Dies verschlechtert die Erweiterbarkeit eines Systems im Allgemeinen, wirkt sich aber an dieser Stelle nicht negativ aus,

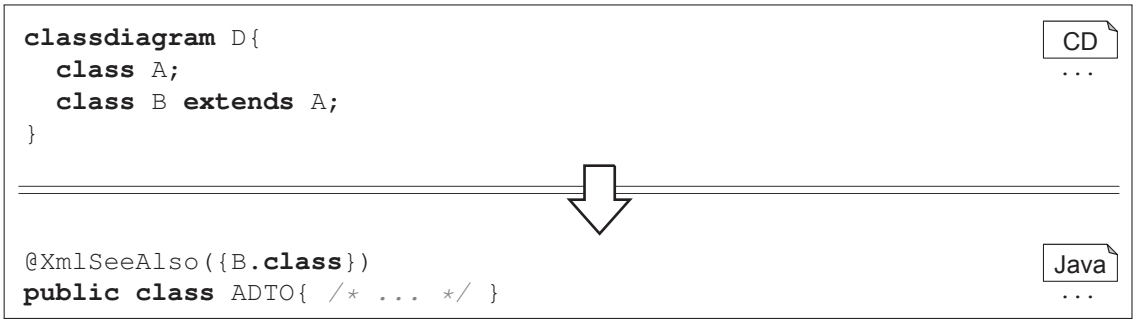


Abbildung 8.5: Abbildung von Vererbung in der `@XMLSeeAlso` Annotation im Generat des DTO-Generators.

da die Klasse generiert wird und aufgrund des ganzheitlichen Verfahrens aus einem monolithischen Klassendiagramm heraus die Subklassen bekannt sind. Ansonsten handelt es sich bei DTOs um normale Java-Klassen. Zudem wird jeder Klasse, die mit dem Tag `Entity` getaggt ist, ein zusätzliches Attribut `Long entityReference` hinzugefügt. Diese beinhaltet eine Referenz auf die in der Datenbank gespeicherte Entität und wird, um die Verbindung zwischen Entität und DTO aufrecht zu erhalten, benötigt. Das Attribut wird von dem ebenfalls generierten DTO-Assembler initialisiert. Die weiteren Konzepte der Klassendiagramme werden analog zum Entity-Generator behandelt. Die verschiedenen Tags, haben bis auf den `Entity` Tag, keine Auswirkungen auf das Generat des DTO-Generators. Darüber hinaus beinhalten DTOs immer verschiedene Requests, die serverseitig verarbeitet werden.

Der `Fetch` Tag wirkt sich auf den DTO-Generator aus. Ist eine Assoziation mit dem `Fetch` Tag und der Option `eager` getaggt, werden normale Getter- und Setter-Methoden generiert. Wird die Option `lazy` verwendet, bewirkt die Getter-Methode einen transparenten Serveraufruf, der die Objekte nachlädt. Dies ist konsistent zur Semantik des Tags und geschieht bei den Entitäten serverseitig automatisch durch den ORM. Auf Seite des Clients ist dies aber, da dort kein ORM existiert, nicht möglich. Somit wird dies durch ein transparentes Nachladen umgesetzt. Dies ist möglich, da der DTO-Generator durch den Facade-Generator Kenntnis von der Kommunikation und der Kommunikationsart mit dem Server besitzt.

8.2.2 Generierung der Requests

In diesem Abschnitt wird die Generierung der Requests und ihre Verankerung in den DTOs beschrieben. Dazu werden zunächst Klassen, die von einer Laufzeitumgebung bereitgestellt werden gezeigt. Daran anschließend werden die aus dem Klassendiagramm generierten Klassen gezeigt. Abbildung 8.6 gibt einen Überblick über die Klassen der Laufzeitumgebung und einer Klasse `DTO`, die für eine beliebige generierte DTO-Klasse steht. Jedes DTO besitzt eine geordnete Menge von Requests sowie eine Methode, die die Requests in einem `CompositeRequest` geschachtelt liest und zurücksetzt.

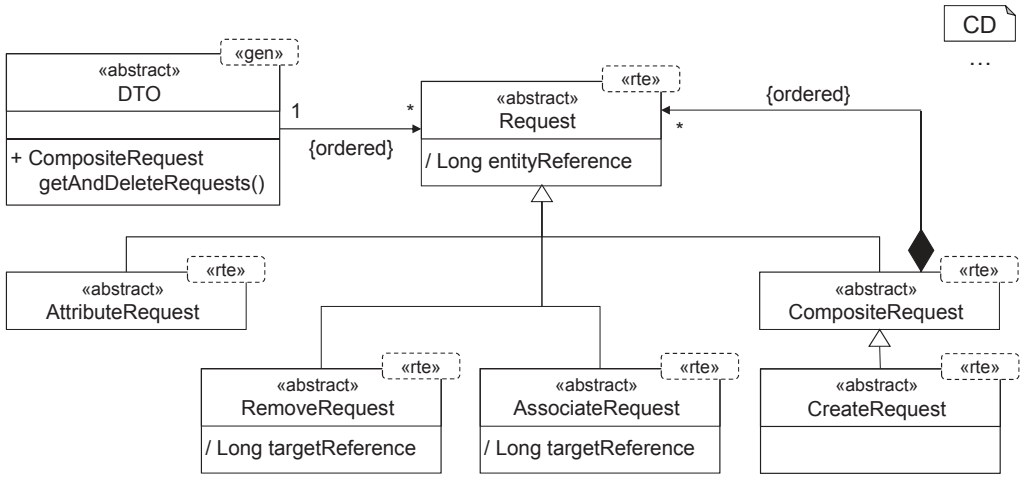


Abbildung 8.6: Auszug der Request Klassen der Laufzeitumgebung der generierten Klassen des DTO-Generators.

Die Klasse `Request` dient als allgemeine Oberklasse für alle existierenden Requests. Die Klasse selbst enthält das abgeleitete Attribut `Long entityReference`. Dieses referenziert die Datenbank-ID der Entität, auf die sich der Request und die damit verbundene Operation bezieht. Das Attribut ist, da es sich aus dem im vorangegangenen Abschnitt beschriebenen Attribut der DTOs ableiten lässt und bei Erstellung eines Requestobjekts auf Basis des Attributwerts initialisiert wird, abgeleitet. Die `Request` Klasse besitzt vier Subklassen. Generell dienen die Requests dazu, Veränderungen an der Objektstruktur auszudrücken. Diese kann ein Verändern von Attributwerten, ein Hinzufügen oder ein Löschen eines Links zwischen zwei Objekten oder aber eine Veränderung eines Enumerationsattributs sein. Die Klasse `AttributeRequest` wird als Superklasse für alle Requests, die den Wert eines Attributs ändern, verwendet. Die Subklassen sind nicht Teil der Laufzeitumgebung, sondern werden attributspezifisch auf Basis des Klassendiagramms generiert. Die Klassen `RemoveRequest` und `AssociateRequest` werden dazu verwendet, einen Link zwischen zwei Objekten hinzuzufügen oder zu entfernen. Aus diesem Grund haben diese abstrakten Klassen auch ein Attribut `Long targetReference`. Dieses referenziert die Datenbank-ID des Objekts zu dem der Link hinzugefügt oder entfernt wird. Die Klasse `CompositeRequest` verwendet das Composite Pattern [GHJV95] zur Verschachtelung mehrerer Requests. Sie dient zur Kapselung mehrerer Requests. Dies ist notwendig damit die Requests als Gruppe zum Server geschickt werden können und auch als solche dort behandelt werden. Als Subklasse des `CompositeRequest` existiert der `CreateRequest`. Dieser wird verwendet um das Anlegen einer neuen Entität auf dem Server auszulösen. Aus diesem Grund kann der `CreateRequest` als Subklasse des `CompositeRequest` weitere Requests kapseln, die die initialen Attributbelegungen der neu zu erstellenden Entität beschreiben. Diese Klassen sind in einer Laufzeitumgebung realisiert.

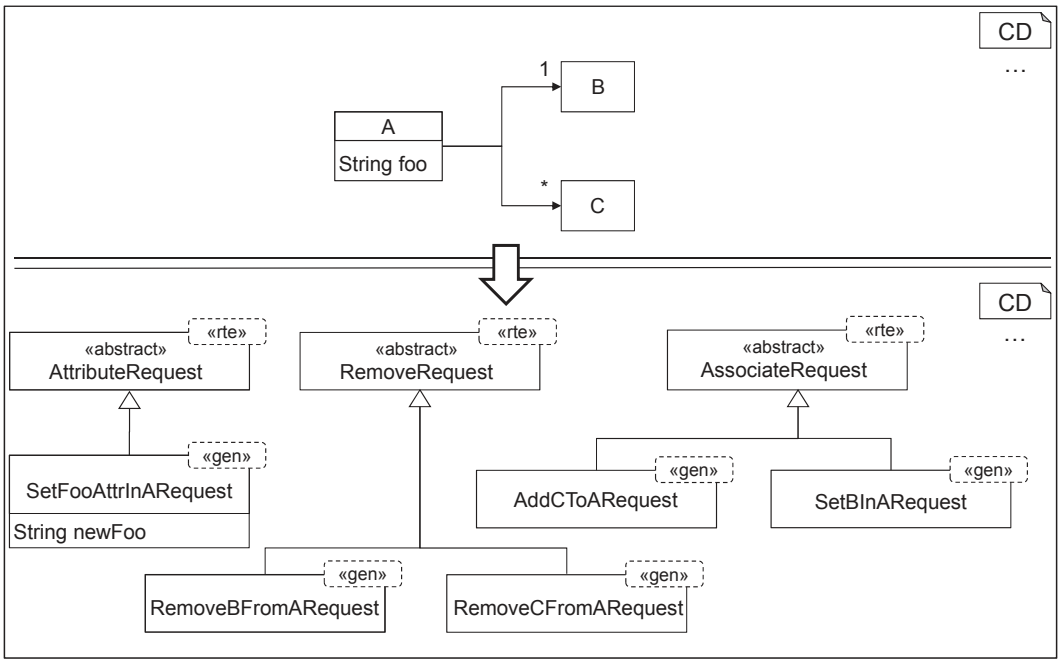


Abbildung 8.7: Auszug der attribut- und assoziationsspezifischen Request Klassen als Subklassen der Request Klassen der Laufzeitumgebung, die vom DTO-Generator basierend auf einem Klassendiagramm generiert werden.

Die geordnete Assoziation der DTOs zu den Requests ist notwendig, da verschiedene Requests eine Reihenfolge, die semantisch wichtig ist, haben können. Dadurch kann eine mehrfache Veränderung eines Attributwerts optimiert werden, so dass nur der letzte Wert in Form eines Requests behandelt werden muss. Ebenso können überflüssige Requests, wie beispielsweise das Hinzufügen und Entfernen des gleichen Links, gefiltert werden. Gleichzeitig erhält das DTO durch die Requests eine Historie, welche durch eine Aktualisierung des persistenten Zustands auf dem Server zurückgesetzt wird. Dies bedeutet, dass die Requests zum Server übertragen, dort ausgeführt werden und die Historie des DTOs wieder zurückgesetzt wird. Aus diesem Grund löscht die Methode zum Abrufen der Requests diese.

Die Erzeugung der Requests im Kontext eines DTOs geschieht immer in den Settern der Attribute oder Assoziationen. Daher wird nachfolgend auf die Generierung der einzelnen attribut- und assoziationsspezifischen Requests sowie deren Erzeugung in den Getter- und Setter-Methoden eingegangen.

Abbildung 8.7 zeigt die attribut- und assoziationsspezifisch generierten Requests. Für jedes modellierte Attribut wird eine Subklasse der `AttributeRequest` Klasse generiert. Dabei folgt die generierte Klasse dem Namensschema eines Präfixes ”Set” gefolgt vom Attributnamen mit dem Suffix ”Attr”, dem Infix ”In”, dem Klassennamen und dem Suffix ”Request”. Das Präfix ”Set” wird verwendet, wenn es sich um einwertige Attribute handelt. Bei mengenwertigen Attributen wird das Präfix ”Add” verwendet. Das ange-

hängte Suffix "Attr", das an den Attributnamen angehängt wird, wird zur Unterscheidung von Attributen und Assoziationen benötigt, da dies sonst zu Überschneidungen führen könnte. Der Einbezug des Klassennamens in den Namen des generierten Request wird benötigt, da es sonst bei zwei Attributen gleichen Namens unterschiedlicher Klassen zu einer Namensüberschneidung kommt. In Abbildung 8.7 stellt die generierte Klasse `SetFooAttrInARequest` eine Subklasse der `AttributeRequest` Klasse dar, die Veränderungen des modellierten Attributs `foo` der Klasse `A` kapselt. Aus diesem Grund enthält diese Klasse auch ebenfalls ein Attribut vom gleichen Typ und Namen mit dem Präfix "new" wie das Attribut `foo`. Dieses Attribut beinhaltet den neugesetzten Wert, falls das Attribut `foo` geändert wird.

Für Assoziationen werden ähnliche Requests generiert. Für jede Assoziation wird eine Subklasse der `RemoveRequest` Klasse der Laufzeitumgebung generiert. Instanzen dieser Klasse werden verwendet, um zu signalisieren, dass ein Link zwischen zwei Objekten entfernt werden soll. In Abbildung 8.7 ist dies für die beiden modellierten Assoziationen durch die Klassen `RemoveBFromARequest` und `RemoveCFromARequest` dargestellt. Das Namensschema der generierten Klassen ist ähnlich zum verwendeten Namensschema der Subklassen des `AttributeRequest`. Das Infix "In" wird durch das Infix "From" ersetzt und es wird kein Namenssuffix wie "Attr" verwendet. Ebenso wird für das Hinzufügen eines Links je eine Subklasse des `AssociateRequest` generiert. Das dort verwendete Namensschema verwendet je nach Zielkardinalität einer Assoziation das Präfix "Set" für die Kardinalität `1` und "Add" für `*`. Bei einer einwertigen Assoziation folgt das Namensschema im Wesentlichen dem Namensschema der Attribute, wobei das Namenssuffix "Attr" weggelassen wird. Bei mehrwertigen Assoziationen wird anstelle des Infixes "In" das Infix "From" verwendet.

CD

```
classdiagram D{
  class A {
    String foo;
  }
}
```

Java

```
public class ADTO {
  String foo;
  List<Request> requests;
  public void setFoo(String foo){
    this.foo = foo;
    this.requests.add(new SetFooAttrInARequest(foo));
  }
}
```

Abbildung 8.8: Darstellung der Objekterzeugung von Requests in den Setter-Methoden der DTOs.

Die Vorgehensweise, für jedes Attribut und jede Assoziation jeder Klasse eine eigene Requestklasse zu erstellen, ist sicherlich aufwendig, wenn dies manuell geschehen muss. Da dies im Rahmen dieser Arbeit aber durch einen Generator basierend auf dem Klassendiagram erzeugt wird, existiert kein Mehraufwand für die Erstellung der Request Klassen. Gleichzeitig erhält der Server die Möglichkeit, sehr feingranular auf die Historie von Objektveränderungen zu reagieren. Dadurch lassen sich Operationen zusammenfassen, überflüssige Operationen auslassen oder Operationen delegieren. Für den Nutzer der Objekte ist dies völlig transparent, da er wie gewohnt Getter- und Setter-Methoden verwendet, die intern Requestobjekte erstellen, die dem Benutzer verborgen bleiben. Zudem ist es üblich, dass Änderungen am Objekt innerhalb des Clients sofort dem Server mitgeteilt wird. Dies führt dazu, dass der Server auf jede Änderung sofort reagieren kann, aber auch andere Strategien zur Bearbeitung umsetzen kann. Der Server kann Requests cachen und diese zu gegebener Zeit, ohne dass dies den Client oder die Konsistenz beeinflusst, behandeln.

Abbildung 8.8 zeigt das generierte DTO, die generierten Attribute und die generierte Setter-Methode des DTOs zu dem ebenfalls dargestellten Klassendiagramm. Es wird sowohl gezeigt, dass das Attribut des DTOs durch den Setter gesetzt wird, damit der Client weiterhin auf einem konsistenten Zustand arbeitet, als auch, dass der Request erzeugt und der internen Liste hinzugefügt wird. Durch den Inhalt der Liste lässt sich die Historie der Veränderungen nachvollziehen und bleibt gleichzeitig transparent für den Nutzer. Wie Abbildung 8.3 zeigt, werden die Requests nur für das Domänenmodell des Servers und nicht für jede Sicht generiert. Dies stellt keine Einschränkung, da die Sicht das Domänenmodell nie erweitert und somit auch keine neuen Requests benötigt werden können, dar. Lediglich durch flache Assoziationen können neue Attribute innerhalb der Sicht entstehen. Aber auch hier lässt sich dies durch die kompositionale Struktur der Requests intern abbilden. Bei Verwendung eines Setters, wie in Abbildung 8.8 dargestellt, wird in einem solchen Fall kein Request des Attributs erzeugt, sondern der zum Attribut der flachen Assoziation gehörende Request wird erzeugt. Die Logik, welche Requests in welchen Situationen erzeugt werden, ist im DTO-Generator lokalisiert.

Nachdem bisher die Generierung der DTO-Klassen selbst und die Generierung der Requests sowie deren Laufzeitumgebung vorgestellt wurden, wurde die Erstellung von Instanzen der DTO-Klassen bisher nicht betrachtet. Dabei müssen die persistent in der Datenbank gespeicherten Entitäten auf die unter Umständen weniger Informationen umfassenden DTOs abgebildet werden. Dies wird von dem für eine Sicht generierten DTO-Assembler übernommen.

8.2.3 Generierung des DTO-Assemblers

Der für die Sicht spezifische DTO-Assembler wird ebenfalls auf Basis des Klassendiagramms und einer Sicht generiert. Er wird benötigt, um die DTOs zu instanziieren, die dem Client zur Verfügung gestellt werden. Immer wenn Daten zum Client übertragen werden, werden diesem DTOs zur Verfügung gestellt. Dazu wird die entsprechende Entität aus der Datenbank geladen und das entsprechende DTO mit Hilfe des DTO-Assemblers instanziiert.

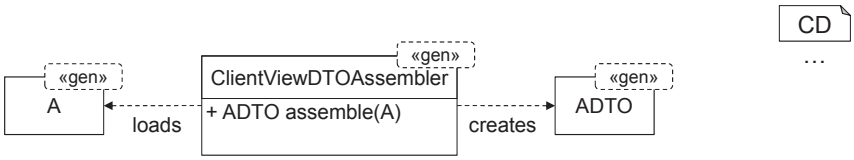


Abbildung 8.9: Schematische Darstellung des generierten DTO-Assemblers basierend auf der `ClientView` Sicht.

Abbildung 8.9 zeigt den Assembler für eine Sicht. Der Assembler `ClientView-DTOAssembler` enthält zu jeder Entität eine `assemble` Methode, die das zugehörige DTO erstellt. Dabei werden nicht nur für die Instanz der Entität, sondern für den vollständigen Objektgraphen, beginnend bei der Instanz der Entität, DTOs aller Objekte und somit ein Objektgraph der DTOs erstellt. Dies bedeutet, dass die Assoziationen der Entität traversiert und für alle aufgefundenen Objekte entsprechend DTOs instanziiert werden müssen. Für die Implementierung der `assemble` Methode wird die Information des Domänenmodells, also des vollständigen Klassendiagramms und des Klassendiagramms, welches aus einer modellierten Sicht entstanden ist, verwendet. Dabei müssen die weggelassenen Konzepte sowie flache Assoziationen betrachtet werden. Als erstes wird das Attribut `entityReference` des DTOs mit der Datenbank-ID der Entität initialisiert. Danach werden alle Attribute der Entität, die in dem aus der Sicht resultierenden Klassendiagramm noch vorhanden sind, entsprechend über Getter- und Setter-Methoden im DTO initialisiert. Attribute, die kein Teil des aus der Sicht resultierenden Klassendiagramms sind, werden nicht initialisiert. Gleiches gilt für Attribute der Entität, die Teil einer Superklasse sind. Dazu wird die in dem aus der Sicht resultierenden Klassendiagramm modellierte Vererbungshierarchie mitbetrachtet. Assoziationen werden entsprechend traversiert. Für jede assoziierte Klasse wird ebenfalls die `assemble` Methode aufgerufen und dem DTO entsprechend hinzugefügt. Dabei müssen Assoziationen, die in der Sicht als flach modelliert waren, besonders berücksichtigt werden. Wie bereits in Abschnitt 5.2.3 vorgestellt, werden Attribute, die das Resultat einer flachen Assoziation sind, bei dem aus der Sicht resultierenden Klassendiagramm intern mit dem `flattened` Tag getaggt. Durch diesen Tag und das Namensschema der Attribute kann identifiziert werden, welche Getter- und Setter der verbundenen Entität benötigt werden, um die Attribute des DTO zu initialisieren. Basierend auf diesen Informationen können die DTOs entsprechend der Sicht initialisiert und den Clients zur Verfügung gestellt werden. Die Rückrichtung gestaltet sich weniger aufwendig, da die DTOs eine Referenz auf die Datenbank-ID der Entität enthalten und diese somit geladen werden kann. Die im DTO vorhandenen Requests müssen dann mit Hilfe eines Requesthandlers auf der Entität ausgeführt werden und diese kann dann in der Datenbank gespeichert werden. Auch wenn der Requesthandler an dieser Stelle die Brücke zwischen DTO und Entität darstellt, wird er erst in Abschnitt 8.4 bei der Erläuterung der Fassade genauer beschrieben, da er dort für die Erklärung des Kommunikationsflusses benötigt wird.

Zusammenfassend generiert der Produktentwickler mit dem DTO-Generator die DTOs, die das Domänenmodell der Clients darstellen. Sie dienen zur Kapselung der zum Client kommunizierten Daten und ermöglichen den spezifischen, auf die Anforderungen des Clients zugeschnittenen Datenaustausch. Der DTO-Generator verwendet das Domänenmodell des Servers und die aus den Sichten generierten Klassendiagramme und erzeugt daraus die DTOs, Requests und Assembler. Für das Domänenmodell und jede Sicht werden sowohl DTOs als auch Assembler in Paketen organisiert generiert. Für das Domänenmodell des Servers werden zudem Requests generiert. Der DTO-Generator besitzt keine eigenständigen, zusätzlichen Kontextbedingungen. Diese sind bereits durch die Kontextbedingungen des Entity-Generators, die in Abschnitt 7.4.3 gezeigt wurden, abgedeckt. Auch die Erzeugung der Requests und die Auswirkungen des `Fetch` Tags wurden gezeigt.

8.3 Der BusinessAPI-Generator

In diesem Abschnitt wird der BusinessAPI-Generator vorgestellt. Dieser dient dazu, die Schnittstelle von der Clientkommunikationsfassade zur Geschäftslogik der Enterprise Applikation (vgl. FA7-WE) sowie von der Geschäftslogik zur Persistenzschicht (vgl. FA8-WE) zu generieren. Dies wird zur Einhaltung und Vorgabe der typischen Schichtenarchitektur benötigt. Für den Produktentwickler wird dadurch die Möglichkeit geschaffen, die handgeschriebene Geschäftslogik in vorgegebene Schnittstellen zu integrieren (vgl. FA16-PE), die gleichzeitig eine Schnittstelle zur Persistenz nutzen kann.

Dazu verwendet der BusinessAPI-Generator das Domänenmodell des Servers und erzeugt daraus ein einzelnes `ApplicationDAO` und mehrere `ApplicationManager` als Schnittstellen. Die vom Facade-Generator generierte Kommunikationsfassade, die in Abschnitt 8.4 vorgestellt wird, kommuniziert nur über die hier präsentierte generierte Schnittstelle mit der Geschäftslogik. Die Geschäftslogik kommuniziert ebenfalls nur über eine klar definierte Schnittstelle mit den in Abschnitt 7.5 vorgestellten generierten DAOs. Die Rückrichtung der Kommunikation erfolgt über die Rückgabewerte der Methoden. Eine aktive Kommunikation der DAOs mit der Geschäftslogik oder der Geschäftslogik mit den Clients wird nicht benötigt.

8.3.1 Abbildung der Modellierungskonzepte auf das Generat

In diesem Abschnitt wird das Generat konzeptionell beschrieben. Dazu werden die Konzepte, die die Generierung beeinflussen, erläutert.

Für die Schnittstelle zwischen der Kommunikationsfassade und der Geschäftslogik erstellt der BusinessAPI-Generator für modellierte Klassen unterschiedliche Interfaces. Dabei werden nicht alle Klassen, sondern nur die, die als `Entity`, analog der Logik der DAO-Generators, der in Abschnitt 7.5 gezeigt wurde, getaggt wurden, berücksichtigt. Dabei folgt der Name des Interfaces dem Namensschema eines vorangestellten "I", gefolgt vom Namen der Klasse und dem angehängten Suffix "Manager". Diese Klassen sind als Enterprise Beans umgesetzt und sind für eine Serverinstanz ein Singleton. Bei mehreren

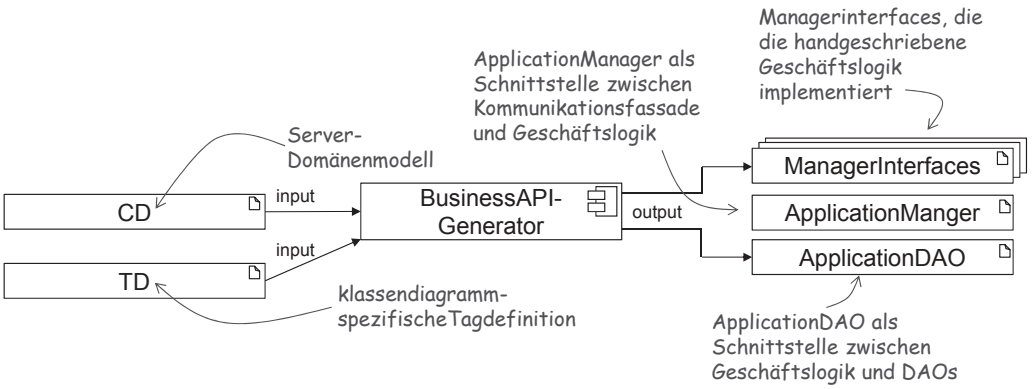


Abbildung 8.10: Darstellung der Eingabe- und Ausgabeartefakte des BusinessAPI-Generators. Er verwendet das Domänenmodell des Servers und erzeugt daraus ein `ApplicationDAO` und einen `ApplicationManager` sowie mehrere `ManagerInterfaces`.

Serverinstanzen können sie hingegen pro Instanz einmal existieren. In diesem Managerinterface, welches mit der `@Local` Annotation, die in Abschnitt 3.3 beschrieben wurde, im Java-Code annotiert wird, werden alle Methoden, die auch vom zugehörigen DAO angeboten werden, definiert. Dies sind die CRUD- und die Query-Methoden für Instanzen der Klasse. In Abbildung 8.11 ist dies durch die zwei Interfaces `IBManager` und `ICManager` dargestellt. Weitere Klassen, die diese Interfaces implementieren, sind ebenfalls dargestellt. Die beiden Klassen `BManager` und `CManager` sind handgeschrieben. Für diese Interfaces werden keine Implementierungen generiert, da diese manuell umgesetzt werden müssen. Dies dient dazu, dass die Requests, die über die Kommunikationsschnittstelle vom Client an den Server gesendet werden, von einem entsprechenden Manager bearbeitet werden können. Diese Bearbeitung kann nicht aus den verfügbaren Modellen abgeleitet werden, da beispielsweise, immer wenn ein bestimmtes Objekt erzeugt wird, komplexe Geschäftsprozesse angestoßen werden müssen. Dies muss von einem handgeschriebenen Manager umgesetzt werden. Eine Modellierung dieser Geschäftslogik über Verhaltensmodelle, wie Statecharts oder Sequenzdiagramme [Rum11, Rum12, Sch12], sowie die Modellierung mit Hilfe von Business Process Model and Notation (BPMN) Mechanismen [GDW09] stellt eine mögliche Erweiterung für MontiEE dar.

Neben den Managerinterfaces für einzelne modellierte Klassen wird immer auch ein übergreifendes Interface mit dem Namen `IApplicationManager`, welches eine Superimposition aller Methoden der einzelnen Managerinterfaces darstellt, erzeugt. Dazu wird ebenfalls eine implementierende Klasse `ApplicationManager` generiert. Diese

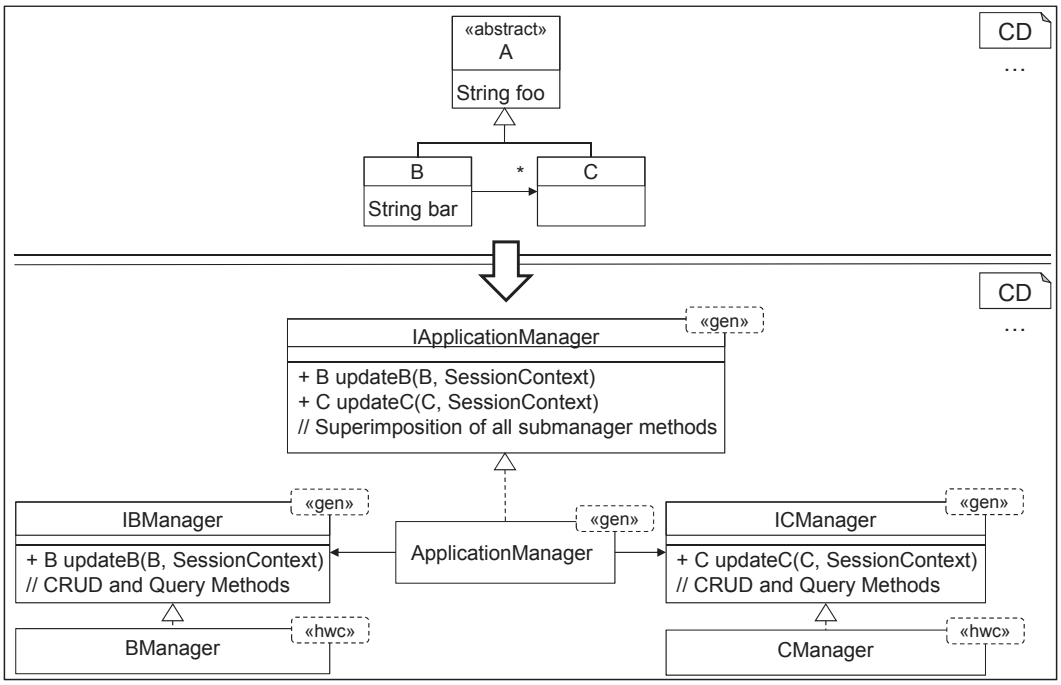


Abbildung 8.11: Exemplarische Darstellung des Generats des BusinessAPI-Generators basierend auf einem Klassendiagramm. Gezeigt ist die Schnittstelle der Kommunikationsfassade zur Geschäftslogik.

Klasse implementiert alle Methoden und delegiert entsprechend an die handgeschriebenen Implementierungen der einzelnen Managerinterfaces. Auf Objektebene wird der Link zwischen `ApplicationManager` und manuell implementierter Klasse des Managerinterfaces der `ApplicationManager` Klasse per Dependency Injection injiziert. Dadurch besitzt die `ApplicationManager` Klasse keine statische sondern eine dynamische Abhängigkeit zu den manuellen Implementierungen.

Das Hinzufügen neuer Funktionalität in Form neuer Methoden ist durch die Möglichkeit zur Subklassen und -interfacebildung, wie in [GHK+15a, GHK+15b] beschrieben, gegeben. Der Produktentwickler kann die generierten Interfaces und entsprechend auch die manuell implementierte Klasse mit handgeschriebenem Code erweitern. Allerdings muss der Entwickler stets das Interface, aber auch die manuell implementierte Klasse, da dem übrigen Generat mit Hilfe des zuvor beschriebenen Dependency Patterns nur Interfaces bekannt sind und diese die zusätzlichen Methoden enthalten müssen, erweitern. Auch hier wird wieder das Dependency Injection Pattern [Pra09] eingesetzt. Eine weitere Möglichkeit ist, dass der Generator Kenntnis über die Existenz von handgeschriebenem Code besitzt. Je nach Vorhandensein des handgeschriebenen Codes erzeugt der Generator, wie in [GHK+15a, GHK+15b] beschrieben, ein entsprechend angepasstes Generat. Diese Kenntnis stellt eine Alternative mit leicht unterschiedlichen Eigenschaften zur Dependency Injection dar. Die weiteren Klassendiagrammelemente oder Tags haben

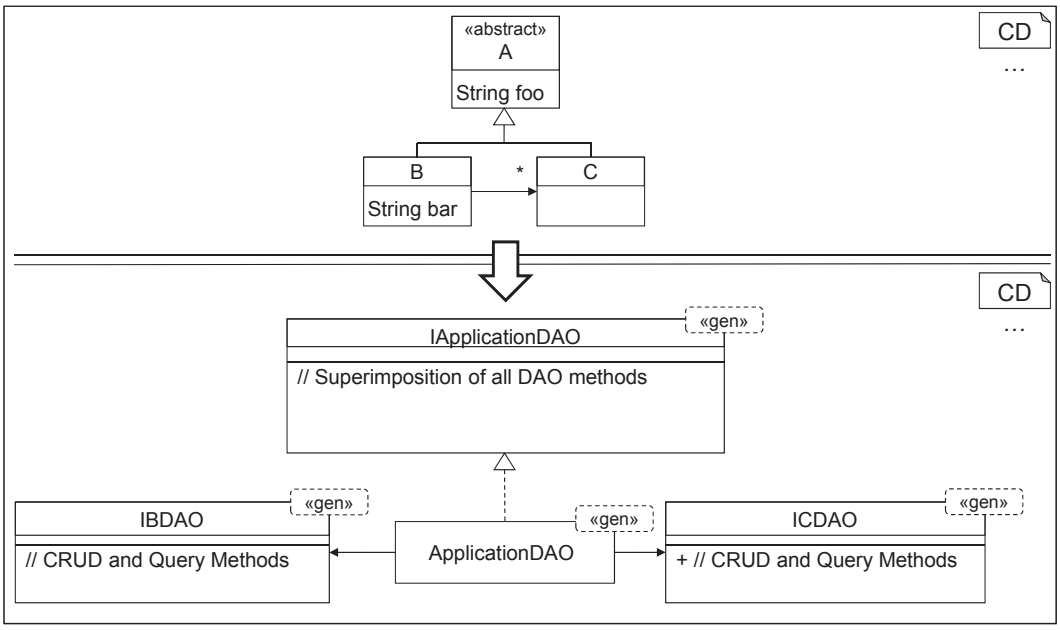


Abbildung 8.12: Exemplarische Darstellung des Generats des BusinessAPI-Generators basierend auf einem Klassendiagramm. Gezeigt ist die Schnittstelle der Geschäftslogik zur Persistenz.

keine Auswirkungen auf das Generat. Die Logik zur Generierung des Java-Codes folgt der Logik zur Erzeugung der DAOs.

Das Interface `IApplicationManager` oder dessen manuelle Erweiterung stellt die API der Kommunikationsfassade zur Geschäftslogik dar. Der Kommunikationsfassade wird zur Laufzeit ein Objekt der `ApplicationManager` Klasse oder ihrer manuell implementierten Subklasse, so dass die Kommunikationsfassade mit der Geschäftslogik kommunizieren kann, bereitgestellt. Auch dies erfolgt per Dependency Injection, so dass lediglich eine statische Beziehung zu einem Interface existiert.

Neben dieser Schnittstelle der Kommunikationsfassade zur Geschäftslogik wird vom BusinessAPI-Generator ebenfalls eine Schnittstelle zu den DAOs generiert.

Diese Schnittstelle der Geschäftslogik zur DAO Schicht folgt ebenfalls einem Muster, welches Abbildung 8.12 zeigt. Sie basiert auf der in Abschnitt 7.5 beschriebenen Logik des DAO-Generators und setzt die Existenz der Interfaces und Implementierung der einzelnen DAOs voraus. Der BusinessAPI-Generator erstellt analog zum Konzept der Manager ein übergreifendes Interface `IApplicationDAO` und eine Implementierung dessen mit Namen `ApplicationDAO`. Auch diese Implementierung stellt, wie der `Application-Manager`, eine Superimposition aller von den DAOs zur Verfügung gestellten Methoden dar. Diese Klasse dient der Geschäftlogik als singulärer Kommunikationspunkt mit der DAO Schicht. Die Erweiterung des übergreifenden DAOs ist analog zur Erweiterung des

`ApplicationManager` möglich. Allerdings wird davon ausgegangen, dass dies selten notwendig ist. Somit stellt das `ApplicationDAO` die Schnittstelle der Geschäftslogik zu den DAOs dar.

Zusammenfassend generiert der Produktentwickler mit dem BusinessAPI-Generator Schnittstellen zwischen Kommunikationsfassaden, die im nächsten Abschnitt vorgestellt werden, und DAOs, die in Abschnitt 7.5 vorgestellt wurden. Dazu wird eine einzelne Schnittstelle zwischen Geschäftslogik und DAOs sowie viele Schnittstellen zwischen Kommunikationssfassade und Geschäftslogik generiert. Diese Schnittstellen sind unabhängig von DTOs oder Sichten, weil diese bereits vorher aufgelöst werden.

8.4 Der Facade-Generator

Der Facade-Generator generiert Kommunikationsfassaden, die von den unterschiedlichen Clients genutzt werden können (vgl. FA15-PE). Die Kommunikationsfassade stellt den Startpunkt der Kommunikation der Clients mit dem Server dar. Dabei verwendet jeder Client seine zugehörige Fassade. Der Applikationsserver autorisiert den Client für den Zugriff auf die ihm erlaubten Methoden. Der Facade-Generator generiert gegen die vom Applikationsserver bereitgestellten Mittel zur Prüfung unterschiedlicher Rollen, die eine Zugriffskontrolle auf Funktionalitätsebene erlauben. Dann leitet er den Aufruf an einen Delegator zur Inklusion handgeschriebener Funktionalität weiter. Der weitere Zugriff läuft dann über die API zur Businesslogik, wird dort von entsprechenden Managern verarbeitet, weiter über die API zu den DAOs und von dort zum Datenbankserver. Bei Verwendung der DTOs wird zudem ein Requesthandler verwendet, der die eingehenden Requests verarbeitet, so dass die Manager und die DAOs keine Kenntnis über die Requests benötigen. Der Produktentwickler erhält durch diesen Generator Zugriffsfassaden, die für jede Art eines Clients spezifisch sind.

Abbildung 8.13 gibt einen Überblick über den Facade-Generator. Er verwendet Klassen-, Rollen-, Rechte- und Mappingdiagramme, die in Kapitel 5 vorgestellt wurden. Er generiert für jedes Klassendiagramm, also für das Domänenmodell und die aus den unterschiedlichen Sichten erzeugten Klassendiagramme, je eine Kommunikationsfassade. Er generiert auf Basis des Rollendiagramms eine Annotation, die die Rollen dem Server bekannt macht, erzeugt die Fassade mit Hilfe der Rechtediagramme und verwendet die zugehörigen Mappingdiagramme dazu, die Autorisierung der Clients zu ermöglichen und unterschiedliche Rollen und Rechte zu prüfen. Zudem werden Interfaces generiert, die von Delegatoren implementiert werden können, so dass diese weitere Funktionalität hinzufügen können. Eine Integration weiterer Frameworks, wie beispielsweise Apache Shiro [Shi16], stellt eine Erweiterung dar, die eine Zugriffskontrolle nicht nur auf Funktionalitätsebene, sondern auch auf Instanzebene ermöglicht. Somit lassen sich feingranularere Aussagen treffen. Bei der Einbindung eines solchen Frameworks kann sich zu Nutze gemacht werden, dass die Rechtesprache bereits OCL-ähnliche Ausdrücke erlaubt, die feingranularere Aussagen erlauben. Die Rechtesprache erlaubt es, Aussagen über die Objekte, deren Attribute und Assoziationen des jeweiligen modellierten Kontexts, analog zur OCL, zu treffen. Alternativ ist eine manuelle Erweiterung über Delegatoren möglich.

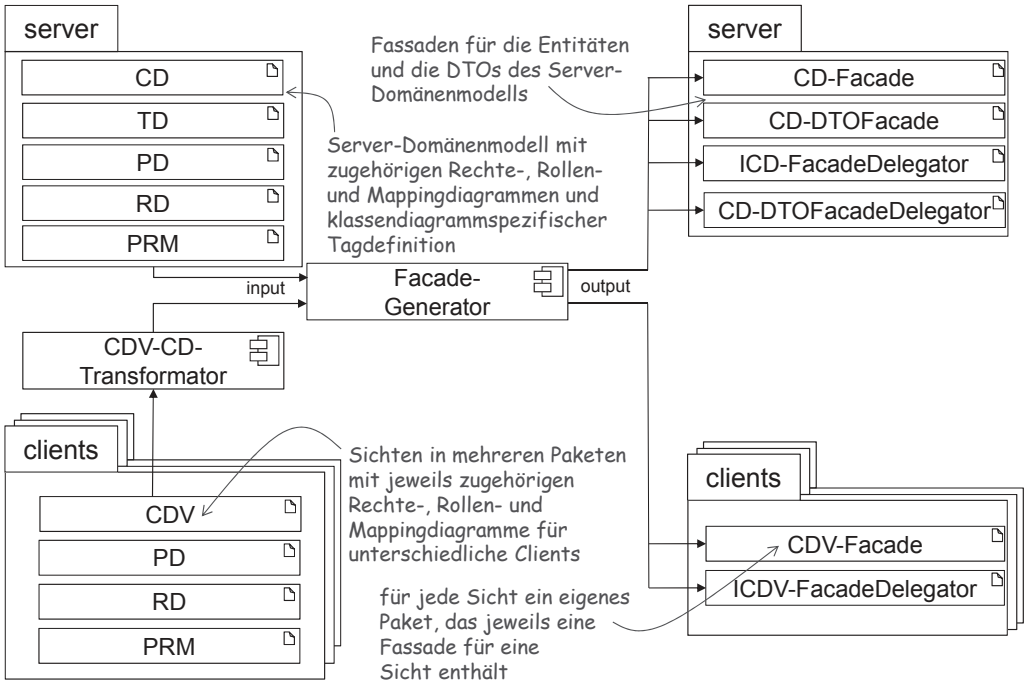


Abbildung 8.13: Darstellung der Eingabe- und Ausgabeartefakte des Facade-Generators. Er verwendet das Domänenmodell des Servers, Rechte- Rollen- und Mappingdiagramme sowie in Klassendiagramme transformierte Sichten und erzeugt daraus mehrere Fassaden, die in unterschiedlichen Paketen organisiert sind. Für das Domänenmodell erzeugt der Facade-Generator eine Fassade, die die Entitäten unterstützt, und eine Fassade, die DTOs unterstützt. Für die Sichten werden stets Fassaden, die die DTOs unterstützen, generiert.

8.4.1 Abbildung der Modellierungskonzepte auf das Generat

Der Facade-Generator verwendet mehrere Eingabemodelle. Er generiert für jedes Klassendiagramm, welches als Eingabe verwendet wird, ein Interface und eine implementierende Klasse. Dabei verwendet er das Domänenmodell und die Klassendiagramme, die für jede Sicht modelliert werden. Zudem verwendet der Facade-Generator ein Rollendiagramm, Rechte- und Mappingdiagramme. Nachfolgend wird die Abbildung der Eingabemodelle auf das Generat exemplarisch an der Fassade und dem Delegationsinterface für das Domänenmodell des Servers gezeigt. Die Auswirkungen der DTOs, die in Abschnitt 8.2 vorgestellt wurden, und der Sichten, die in Kapitel 5 präsentiert wurden, werden im nächsten Abschnitt kurz erläutert. Das Rollendiagramm wird für die gesamte zu modellierende Enterprise Applikation systemweit erstellt. Aus diesem Grund existiert genau ein Rollendiagramm für ein System. Auch wenn die Rollendiagramme Vererbung auf Ebene der modellierten Rollen unterstützen, wird als Rollendiagramm immer jenes verwendet, von dem keine weiteren Rollendiagramme erben.

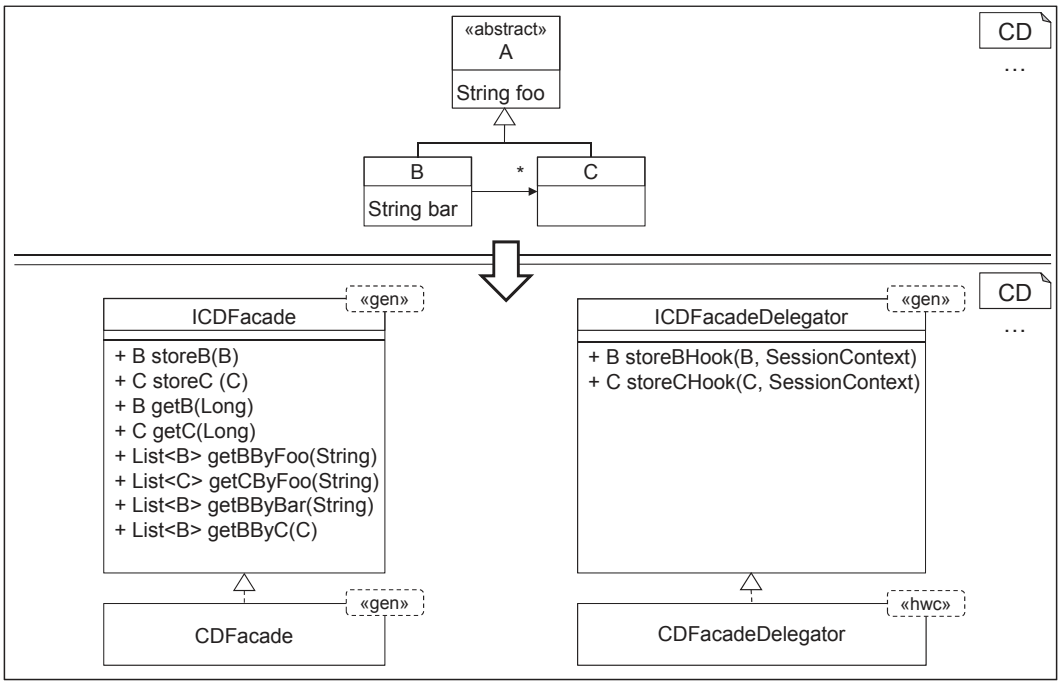


Abbildung 8.14: Exemplarische Darstellung des Generats des Facade-Generators basierend auf einem Klassendiagramm.

Abbildung 8.14 zeigt das generierte Interface und die Implementierung der Fassade. Der Name der Fassade basiert auf dem Namen des Klassendiagramms. Die Methoden des Interfaces basieren auf den generierten Rechten des Rechtediagramms. Dazu ist ein Teil der möglichen Methoden der Klassen `B` und `C` dargestellt. Das Interface der Fassade enthält alle möglichen Methoden für alle Klassen. Dies sind die benötigten CRUD-Methoden, die generierten Querymethoden und auch die benutzerspezifischen Querymethoden. Dabei werden nur Methoden generiert, die benötigt werden, um Objekte aufzulösen. Dies ist beispielsweise beim Query von Objekten entgegen der Assoziationsrichtung der Fall. Aus diesem Grund existiert die Methode `getBByC(C)`, aber nicht die Methode `getCByB(B)`, da dies über die Getter- und Setter-Methoden des Parameters bereits aufgelöst werden könnte und somit nicht als Client-Server-Kommunikation umgesetzt werden muss. Für jedes Klassendiagramm kann, wie in Abschnitt 5.3.4 beschrieben, ein Rechtediagramm automatisch generiert werden. Dieses Rechtediagramm hat einen Namen und bezieht sich auf genau ein Klassendiagramm. Der Facade-Generator benutzt diese Rechtediagramme zur Generierung der Fassaden. Zudem wird für das Rollendiagramm und jedes Rechtediagramm ein Mappingdiagramm, welches die Zuordnung von Rollen und Rechten vornimmt, erstellt. Diese Mappingdiagramme werden vom Facade-Generator verwendet, um mit Hilfe der Sicherheitsmechanismen des Applikationsservers den Zugriff auf bestimmte Methoden für bestimmte Rollen zu ermöglichen. Verschiedene Rechte sind der Logik zur Erstellung des Rechtediagramms folgend nicht

enthalten. Dies ist ein Unterschied zum DAO-Generator, der alle möglichen Operationen anbietet. Nicht dargestellt sind die Auswirkungen des Rollendiagramms und des Mappingdiagramms, da dies in der Implementierung umgesetzt ist. Gleichzeitig wird das Interface `ICDFacadeDelegator`, dass weitere Prüfungen oder handgeschriebene Funktionalität erlaubt, generiert. Das Interface beinhaltet zu jeder Methode der Fassade eine Hookmethode an die direkt zu Beginn der aufgerufenen Methode delegiert wird. Zudem werden die Parameter um den aktuellen Kontext der Session angereichert. Die Implementierung des Delegators muss handgeschrieben erfolgen.

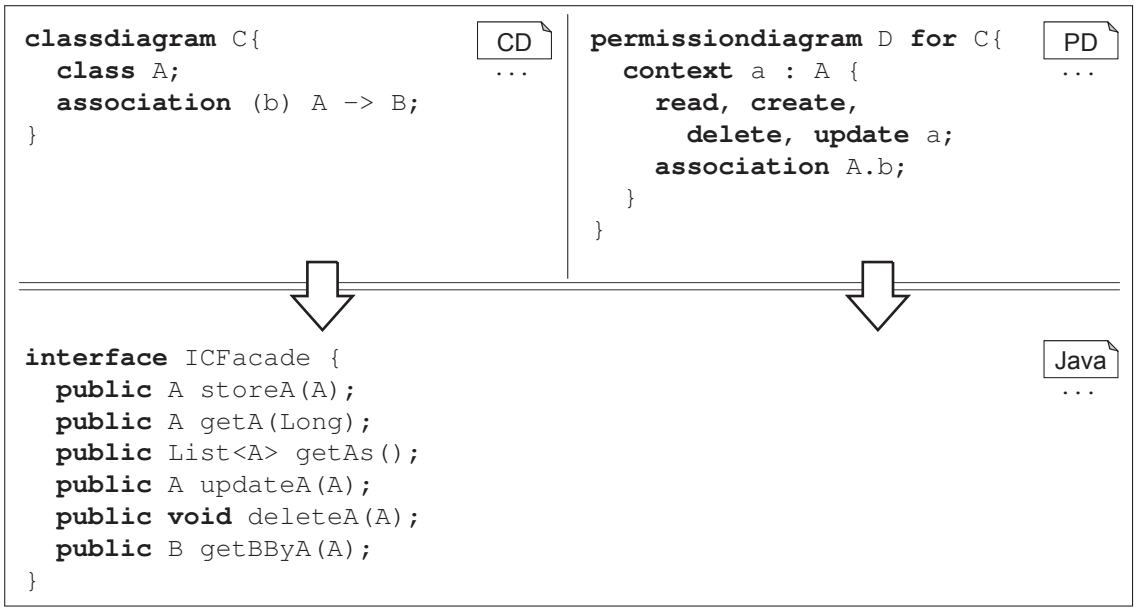


Abbildung 8.15: Abbildung eines Klassen- und eines Rechtediagramms im generierten Interface des Facade-Generators ohne Verwendung der DTOs.

Abbildung 8.15 zeigt das generierte Interface für ein Klassen- und ein Rechtediagramm. Für jedes `read` Recht werden die beiden `get` Methoden generiert. Für jedes `create` Recht wird die `store`, für jedes `delete` Recht die `delete` Methode generiert. Zudem wird für jedes `update` Recht eine `update` Methode generiert.

Für jedes Recht zur Ausführung von Queries wird ebenfalls eine `get` Methode analog zu den Methoden für Queries im DAO generiert. Für ein Recht zur Navigation einer Assoziation wird ebenfalls eine Methode generiert. Diese Methode liefert, wie in Abbildung 8.15 dargestellt, ein Objekt des Assoziationsendes, oder eine Liste von Objekten bei mehrwertigen Assoziationen.

Dadurch können alle Rechte, die in einem Rechtediagramm modelliert werden können, als Methoden repräsentiert werden. Es sei angemerkt, dass das Recht zur Assoziationsnavigation nicht die Möglichkeit einschränkt, den entsprechenden Getter einer Klasse zu verwenden. Dieser muss aber in einer Client-Server Architektur nicht zwingend belegt

sein, so dass das modellierte Recht nicht zwangsweise umgangen werden kann. Allerdings kann es durchaus Situationen geben, in denen dies möglich ist. Dies auszuschließen obliegt dem Modellierer.

Die Implementierung der Klasse verwendet das ebenfalls generierte Interface eines Delegators und fügt eine Hookmethode hinzu. Diese kann zur Erweiterung verwendet werden. Durch die Verwendung des Dependency Injection Patterns [Pra09] muss die handgeschriebene Implementierung im Generat nicht bekannt sein. Zudem verwendet die Implementierung der Klasse das vom BusinessAPI-Generator bereitgestellte Interface `IApplicationManager`, um den Methodenaufruf an die Geschäftslogik weiterzuleiten, wo diese dann verarbeitet werden kann. Dazu werden die Parameter der Methode um den Kontext der aktuellen Serversession angereichert. Dieser enthält weitere Informationen, wie den Benutzernamen oder das Passwort.

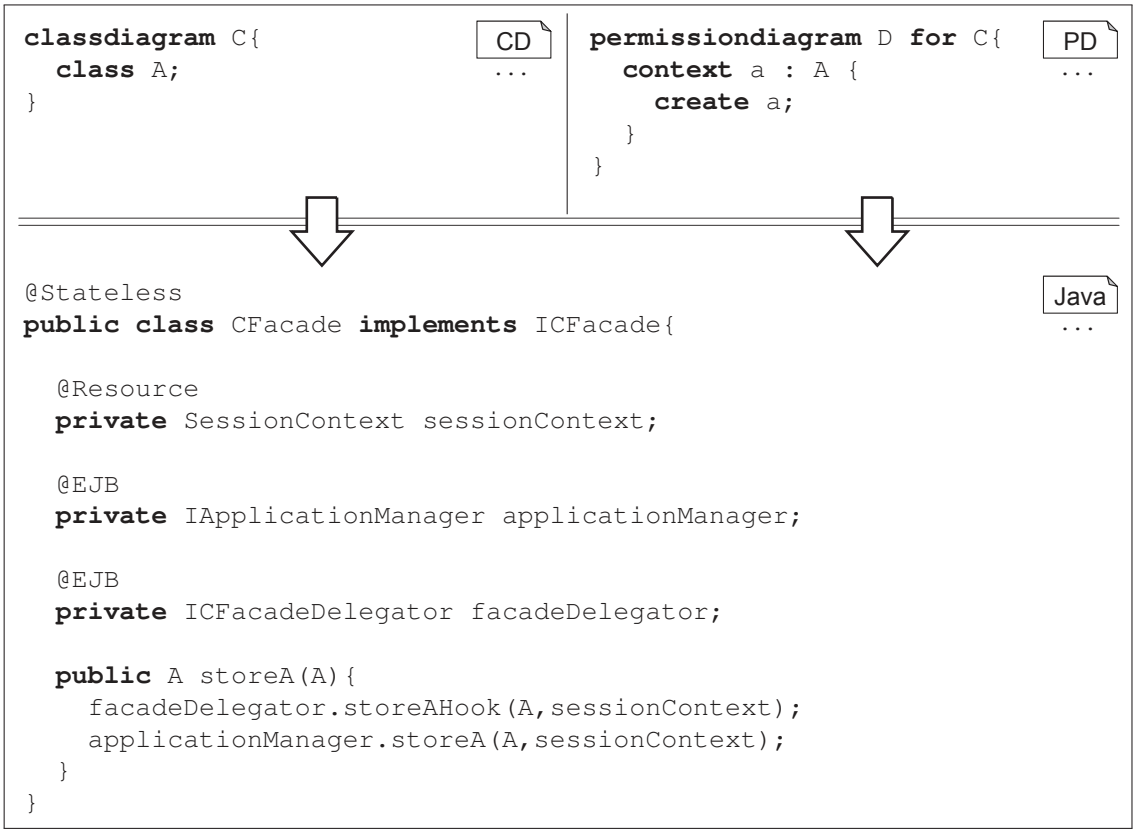


Abbildung 8.16: Abbildung eines Klassen- und eines Rechtediagramms in der generierten Klasse des Facade-Generators ohne Verwendung der DTOs.

Abbildung 8.16 zeigt die Implementierung der Fassade. Sie wird als zustandslose Bean mit Hilfe der Annotation `@Stateless`, wie in Abschnitt 3.3 beschrieben, umgesetzt. Die Fassade besitzt zwei Felder, die zur Laufzeit mittels Dependency Injection gesetzt werden. Der `sessionContext` wird für jede Verbindung zu einem Client neu initiali-

siert. Dies wird durch die generierte Annotation `@Resource` umgesetzt. Das `applicationManager` Feld wird mit der Implementierung, wie in Abschnitt 3.3 beschrieben, belegt. Dies ist durch die generierte Annotation `@EJB` realisiert. Die Implementierung der `store` Methode delegiert an die Geschäftslogik, die dies weiter verarbeitet.

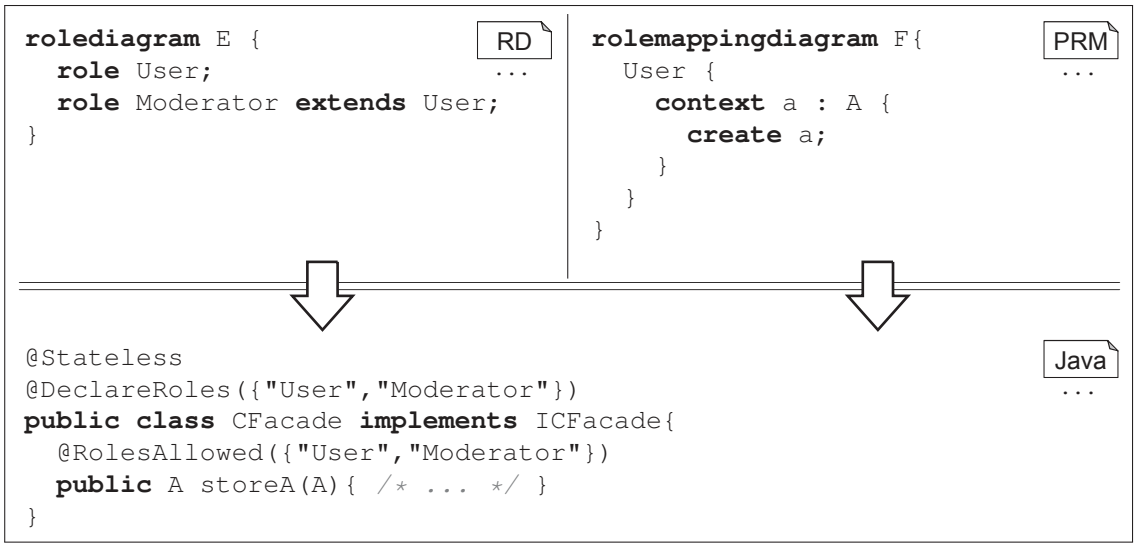


Abbildung 8.17: Auswirkungen des Rechte- und des Mappingdiagramms auf das Generat des Facade-Generators.

Abbildung 8.17 zeigt die Auswirkungen der modellierten Rollen und der Zuordnung zwischen Rechten und Rollen. Zunächst werden am Klassenkopf mit der Annotation `@DeclareRoles` alle verfügbaren Rollen angegeben. Die Informationen über die möglichen Rollen sind durch das Rollendiagramm, welches vom Facade-Generator verarbeitet wird, gegeben. Darüber hinaus werden die Informationen des Mappingdiagramms dazu verwendet, die einzelnen Methoden, die die Methode aufrufen dürfen, mit den Rollen zu annotieren. Dazu wird die Annotation `@RolesAllowed` verwendet. Auch die Vererbungsbeziehung zwischen Rollen wird berücksichtigt, so dass auf die Rollen, die von einer anderen Rolle erben, die Rechte zum Ausführen einer Methode erhalten. Die Angabe der Rollen und der Rechte sind bei Verwendung der Sicherheitsmechanismen des Applikationsservers statisch. Dies bedeutet, dass sie zur Laufzeit nicht erweitert werden können. Eine Erweiterung der Zugriffskontrolle zur Laufzeit und zur Prüfung von Rechten auf Instanzebene kann mit Hilfe der Delegatoren, wie sie in Abbildung 8.16 dargestellt sind, handgeschrieben hinzugefügt werden.

Die Generierung des Interfaces und der Klasse wird für alle verfügbaren Klassen-, Rechte- und Mappingdiagramme, so dass für unterschiedliche Clients unterschiedliche Zugriffsfassaden entstehen, die diesen angeboten werden und von diesen genutzt werden können, durchgeführt. Auch die zuvor beschriebenen DTOs, welche spezifisch für verschiedene Clients generiert werden, können berücksichtigt werden. Dazu verwenden die Fassaden immer die Klassen, die aus dem entsprechend verwendeten Klassendiagramm

generiert wurden. Dies bedeutet, dass jede Fassade, die aus einem Klassendiagramm generiert wurde, die DTOs verwendet, die ebenfalls aus diesem Klassendiagramm generiert wurden. Der einzige Sonderfall ist der, dass das Domänenmodell als Eingabe verwendet wird. Hierbei wird unterschieden, ob DTOs für das Domänenmodell generiert wurden oder nicht. Inwiefern die DTOs sich auf das Generat des Facade-Generators auswirken, wird nachfolgend beschrieben. Im Anschluss daran werden die unterschiedlichen Arten von Fassaden, Webservice oder RMI, kurz erläutert.

Auswirkungen der DTOs

Wie bereits erläutert, wirkt sich die Verwendung von DTOs auf das Generat aus. Werden DTOs verwendet, ändert sich die API der generierten Fassade.

```
1 interface ICFacade {                                          Java
2   public ADTO storeA(Request);                                 ...
3   public ADTO updateA(Request);
4 }
```

Listing 8.18: Darstellung des vom Facade-Generator generierten Interfaces bei Verwendung der DTOs.

Listing 8.18 zeigt die veränderte Signatur des generierten Interfaces. Anstelle der Entitäten werden die in den DTOs gekapselten Requests verwendet. Als Rückgabewerte werden die DTOs verwendet. Generell ist auch eine einzelne Methode, die alle Requests verarbeitet, denkbar, da die Requests bereits die Informationen über die gewünschte Operation enthalten. Dies ist aber aus technologischen Gründen nur eingeschränkt möglich, da die Webservice Frameworks überladen von Methoden mit unterschiedlichen Parametern, die die gleiche Superklasse haben, nicht gut unterstützen. Aus diesem Grund wurde sich an dieser Stelle entschieden, die Methoden analog zur zuvor vorgestellten Fassade zu definieren. Somit ändert sich nur der Parameter und der Rückgabewert der Methode.

Darüber hinaus wird ein sogenannter Requesthandler, der die Requests auflöst und auf die Entitäten abbildet, verwendet. Die Geschäftslogik arbeitet auf Basis der Entitäten, so dass der Requesthandler die Entitäten gemäß der Requests aus der Datenbank lädt, die einzelnen Requests ausführt und dazu auch die API zur Geschäftslogik verwendet. Somit ist der Requesthandler dafür verantwortlich, dass die Verbindung von DTO zur Entität hergestellt wird und die Geschäftslogik nur auf der Entität arbeiten kann. Für den Rückgabewert wird der in Abschnitt 8.2 vorgestellte Assembler, so dass aus einer Entität wieder ein DTO erzeugt wird, verwendet. Der Aufruf kommt an der Fassade an, wird an den zuvor vorgestellten Delegator zur Integration von handgeschriebenem Code weitergeleitet, wird danach dem Requesthandler, der die betroffenen Entitäten auflöst und die Requests mit Hilfe der Geschäftslogik ausführt, übergeben. Die Entität wird über die Geschäftslogik und über die Schnittstelle zur Datenbank persistiert. Für den Rückgabewert wird die von der Datenbank zurückgegebene Entität durch die Geschäftslogik

gereicht. Dann wird mit Hilfe des Assemblers in der Fassade aus der veränderten Entität wieder ein DTO erzeugt und an den Client zurückgeliefert. Wie bereits zuvor ist die API der Geschäftslogik unverändert und wird um den Kontext der Session angereichert. Im weiteren Verlauf werden die Generierung der Webservice- und der RPC-Fassade (vgl. FA4-WE) genauer erläutert.

Generierung als Webservice-Fassade

Der Facade-Generator erlaubt es, entweder Webservice-Fassaden oder aber RPC-Fassaden zu erstellen. Damit eine Klasse als Webservice-Fassade verwendet werden kann, muss sie mit zusätzlichen Annotationen versehen werden. Die meisten Annotationen werden dabei beim Interface der Klasse verwendet.

```
1 @WebService(serviceName="ICFacadeService",                    Java
2             portName="ICFacadePort")                           ...
3 interface ICFacade {
4   @WebMethod
5   @WebResult(name="storeAResult")
6   public A storeA(A);
7 }
```

Listing 8.19: Darstellung des generierten Interfaces und der Annotationen `@Webservice`, `@WebMethod` und `@WebResult` zur Realisierung einer Webservice-Fassade.

Listing 8.19 zeigt die Annotationen, die für das Interface generiert werden. Die Klasse selbst wird mit der Annotation `@WebService`, die zwei Parameter besitzt, annotiert. Die Parameter werden vom verwendeten Webservice Framework, um die benötigten Webservice Infrastrukturen, wie die WSDL, zu generieren, verwendet. Ebenso ergeben sich daraus Namen generierter Klassen. Zusätzlich wird jede Methode, die von dem Webservice zur Verfügung gestellt wird, mit der Annotation `@WebMethod` annotiert. Gibt die Methode einen Wert zurück, muss diesem ein Name mit Hilfe der Annotation `@WebResult` und dem `name` Parameter gegeben werden.

Darüber hinaus benötigt auch die generierte Klasse eine zusätzliche Annotation.

```
1 @WebService(endpointInterface="ICFacade")                     Java
2 public class CFacade implements ICFacade{                      ...
3   public A storeA(A){ /* ... */};
4 }
```

Listing 8.20: Darstellung der generierten Klasse und der `@Webservice` Annotation zur Realisierung einer Webservice-Fassade.

Listing 8.20 zeigt dazu die Annotation `@WebService`, die auch an den Klassenkopf generiert wird. Allerdings wird hier der Parameter `endpointInterface`, der eine Referenz als String auf das zuvor vorgestellte Interface darstellt, verwendet.

Dadurch können Clients, die über Webservices kommunizieren, den Endpunkt ansprechen und dessen Methoden verwenden. Auf Seiten des Clients können existierende Webservice-Client Frameworks verwendet werden.

Generierung als RPC-Fassade

Neben der zuvor vorgestellten Möglichkeit des Facade-Generators, die Kommunikationsfassade als Webservice-Fassade zu generieren, kann diese auch als RPC-Fassade generiert werden. Listing 8.21 zeigt dazu das generierte Interface. Zur Deklaration als RPC-Fassade wird lediglich die Annotation `@Remote`, die in Abschnitt 3.3 vorgestellt wurde, benötigt. Diese führt dazu, dass der Applikationsserver das Interface und seine Implementierung für Clients zugänglich macht.

Java

```
1 @Remote
2 interface ICFacade {
3   public A storeA(A);
4 }
```

...

Listing 8.21: Darstellung des generierten Interfaces und der Annotation `@Remote` zur Realisierung einer RPC-Fassade.

Zusammenfassend generiert der Produktnutzer mit dem Facade-Generator die Kommunikationsfassaden des Servers. Dabei generiert er für das Domänemodell des Servers eine Fassade für die Entitäten und die DTOs. Zudem generiert er für jede Sicht eine weitere Fassade, die von den unterschiedlichen Clients verwendet werden kann. Innerhalb der Fassade werden die vom Applikationsserver bereitgestellten Mechanismen zur Autorisierung verwendet. Zusätzlich wird ein Interface eines Delegators erzeugt, welches zur Integration handgeschriebener Funktionalität, wie beispielsweise einer erweiterten Autorisierung, verwendet wird. Darüber hinaus werden zwei verschiedene Arten von Fassaden unterstützt: Webservice und RPC. Je nach Art des Clients können diese die unterschiedlichen Fassaden verwenden.

8.5 Zusammenfassung

In diesem Kapitel wurden die MontiEE-Generatoren zur Generierung der Kommunikationsinfrastruktur von Enterprise Applikationen detailliert vorgestellt.

Zunächst wurde in Abschnitt 8.2 der DTO-Generator zur Generierung der Transferobjekte für spezifische Clients präsentiert. Hierbei wurden die DTOs, das Request Pattern sowie die Assembler in einer detaillierten Erklärung der Auswirkungen aller beteiligten Tags vorgestellt.

In Abschnitt 8.3 wurde der BusinessAPI-Generator, der Schnittstellen zur Geschäftslogik generiert, gezeigt. Dabei generiert er zum einen Schnittstellen von den Kommunikationsfassaden zur Geschäftslogik und zum anderen eine Schnittstelle von der Geschäftslogik zu den DAOs.

In Abschnitt 8.4 wurde der Facade-Generator, der die Kommunikationsfassaden zur Kommunikation mit den Clients generiert, vorgestellt. Dazu werden zwei verschiedene Fassaden, Webservice und RPC, angeboten.

Mit Hilfe dieser MontiEE-Generatoren kann der Produktnutzer einen Großteil der Enterprise Applikation generieren. Die generierten Teile lassen sich an verschiedenen Erweiterungspunkten um handgeschriebene Methoden erweitern. Durch die geschaffenen Schnittstellen zur Geschäftslogik benötigt der Entwickler kaum technologiespezifische Kenntnisse und er kann die Geschäftslogik in Java, ohne Annotationen, umsetzen. Somit muss er sich nicht um die Handhabung des komplexen Technologiestacks kümmern.

Kapitel 9

Generierung der Evolutionsinfrastruktur

In den vorangegangenen Kapiteln wurden Generatoren, die Teile des Systems, wie die Persistenz oder die Kommunikationsinfrastruktur, generieren, vorgestellt. Dazu wurden der Entity-, der DAO-, der SQL-, der DTO-, der Facade- und der BusinessAPI-Generator, die in Kapitel 7 und 8 präsentiert wurden, vorgestellt. Jeder Generator generiert einzelne Teile des Systems, die in Summe die Enterprise Applikation ergeben. In diesem Kapitel wird ein Generator, der Systemevolution und Datenmigration ermöglicht, vorgestellt.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Die Generierung von Operationen zur Evolution von Klassendiagrammen.
- Die Generierung von Operationen zur Migration von Objektdiagrammen.
- Eine Laufzeitumgebung zur Unterstützung der Operationen.
- Serialisierung und Deserialisierung persistenter Daten in Objektdiagramme.

Zunächst wird der Delta-Generator vorgestellt. Daran anschließend wird die Evolution von Klassendiagrammen, gefolgt von der Migration von Objektdiagrammen erläutert. In Kapitel 10 wird die Methodik der Verwendung der Generatoren, deren Konfiguration und die Umsetzung des Szenarios vorgestellt.

9.1 Überblick

Abbildung 9.1 zeigt die im Rahmen von MontiEE umgesetzten Generatoren mit Fokus auf den Delta-Generator zur Generierung der Infrastruktur zur Evolution und Migration des Systems. Dazu wurde in Kapitel 6 eine Sprache zur Modellierung der Systemevolution und zur gleichzeitigen Ableitung einer benötigten Datenmigration vorgestellt. Die Modelle der in Kapitel 6 beschriebenen Sprache werden zur Generierung einer Infrastruktur, die in diesem Kapitel vorgestellt wird, eingesetzt. Die beschriebenen Deltas besitzen zwei unterschiedliche Funktionen: Zum einen beschreiben sie eine Systemevolution in Form einer Veränderung des zu Grunde liegenden Domänenmodells, zum anderen beschreiben sie, in welcher Art und Weise die in der Datenbank persistent gespeicherten Daten verändert werden müssen, um wieder konform zu dem Domänenmodell des Servers zu sein. Die Evolution des Klassendiagramms beeinflusst alle zuvor beschriebenen Generatoren, die die Entitäten, die DAOs, die DTOs sowie die Fassaden und das Datenbankschema

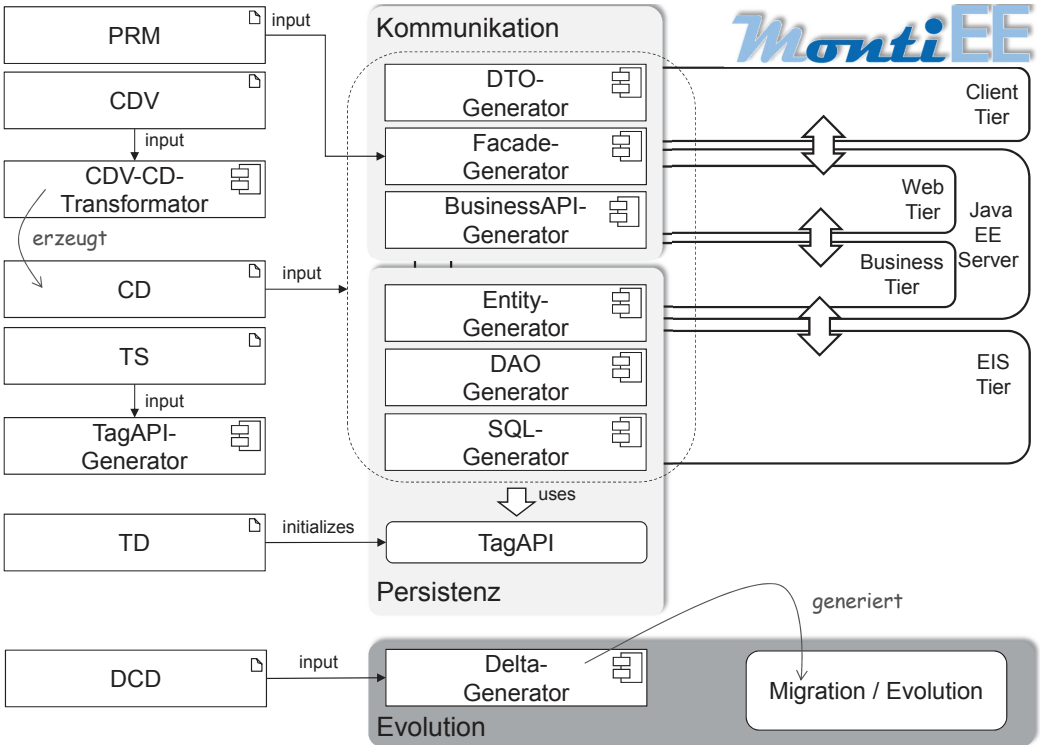


Abbildung 9.1: Übersicht der in MontiEE umgesetzten Generatoren. Gezeigt sind die Eingabemodelle sowie die Teile der Enterprise Applikation, die mit Hilfe der Generatoren generiert werden. Der Fokus liegt dabei auf den Generatoren zur Generierung der Evolution von Enterprise Applikationen.

generieren. Vor allem aber kommt es bei der Evolution eines laufenden Systems zu einer Inkonsistenz zwischen dem in Verwendung befindlichen Datenbankschema und den aktuellen Entitäten. Dem ORM ist es nicht mehr möglich, die weiterentwickelten Entitäten zu mappen. Daher reicht es nicht aus, das Klassendiagramm weiterzuentwickeln. Eine gleichzeitige Datenmigration der Daten in der Datenbank ist unabdingbar (vgl. FA11-PE). Die generierte Infrastruktur erlaubt die Evolution von Klassendiagrammen und der zugehörigen Datenmigration der persistenten Daten.

Der Delta-Generator unterscheidet sich von den bisher vorgestellten Generatoren darin, dass er keinen Teil der eigentlichen Enterprise Applikation, sondern Werkzeuginfrastruktur zur Weiterentwicklung und Wartung der Applikation, generiert. Er stellt auch keine Infrastruktur, wie die TagAPI, die von den übrigen Generatoren benutzt wird, bereit, sondern bietet ein eigenständiges Werkzeug. Dieses Werkzeug basiert darauf, dass im Rahmen dieser Arbeit ein generativer, modellgetriebener Ansatz zur Weiterentwicklung des Klassendiagramms und zur Migration der Daten in Form von Objektdiagrammen gewählt wird.

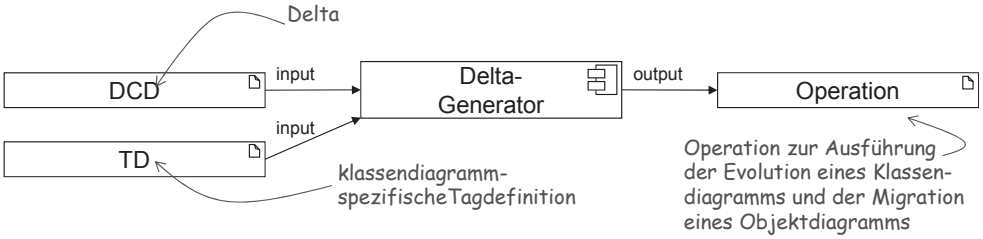


Abbildung 9.2: Darstellung der Eingabe- und Ausgabeartefakte des Delta-Generators. Er verwendet ein Delta und erzeugt daraus eine Operation, die das Delta ausführt.

Abbildung 9.2 zeigt die grundlegende Funktionsweise des Delta-Generators. Der Delta-Generator erzeugt eine Operation, die die Evolution durchführt. Diese Operation ist in Java implementiert und unterstützt die Evolution des Klassendiagramms und die Migration von Objektdiagrammen.

Der Generator benötigt weder Klassen- noch Objektdiagramme zur Generierung des Quellcodes, sondern lediglich Deltas. Das Generat verwendet Klassen- und Objektdiagramme während der Ausführung. Das Klassendiagramm ist das Domänenmodell des Servers und die Objektdiagramme sind Serialisierungen der in der Datenbank gespeicherten Daten. Aus dem Delta wird eine Java-Klasse, die eine API zur Ausführung des Deltas anbietet, generiert. Als Eingabe erhält die API den AST des CDs oder des ODs, welche durch den von MontiCore zur Verfügung gestellten Parser erstellt wurden. Die generierte Klasse transformiert den AST des CDs und den AST des ODs und verwendet den von MontiCore bereitgestellten Prettyprinter zur textuellen Ausgabe des ASTs. Dadurch können sowohl Klassendiagramme als auch Objektdiagramme migriert werden. Da sich das Delta aber immer, wie in Abschnitt 6.2 beschrieben, auf ein Klassendiagramm bezieht und die konkrete Evolution immer auf Basis eines solchen Deltas generiert wird, werden die Objektdiagramme, welche durch die konkrete Evolution transformiert werden, immer analog zur Klassendiagrammveränderung migriert. Sie können somit nicht unabhängig verändert werden. Die Serialisierung und Deserialisierung der Daten der Datenbank in ein Objektdiagramm werden in Abschnitt 9.4 detailliert beschrieben.

Zur detaillierten Erläuterung der Bestandteile des Evolutionsmechanismus wird zunächst beschrieben, wie die Evolution des Klassendiagramms erfolgt und wie sie auf Basis des Deltas generiert wird. Daran anschließend wird die Datenmigration in Form von Objektdiagrammen näher erläutert. Dies unterteilt sich zum einen in die Serialisierung der Objektdiagramme aus dem aktuellen Datenbestand der Datenbank und der Deserialisierung zurück in die Datenbank und zum anderen in die eigentliche Migration der Objektdiagramme sowie die Verwendung der Initialisierer.

Die Datenmigration kann auch auf die Objektdiagramme, die in Testfällen verwendet werden, angewendet werden, so dass diese gleichzeitig migriert werden. Dazu wurde innerhalb der Arbeit ein angepasstes Testframework in Zusammenarbeit mit [Plo12], basierend auf der Testfallsprache der UML/P [Sch12] zum Testen von Enterprise Ap-

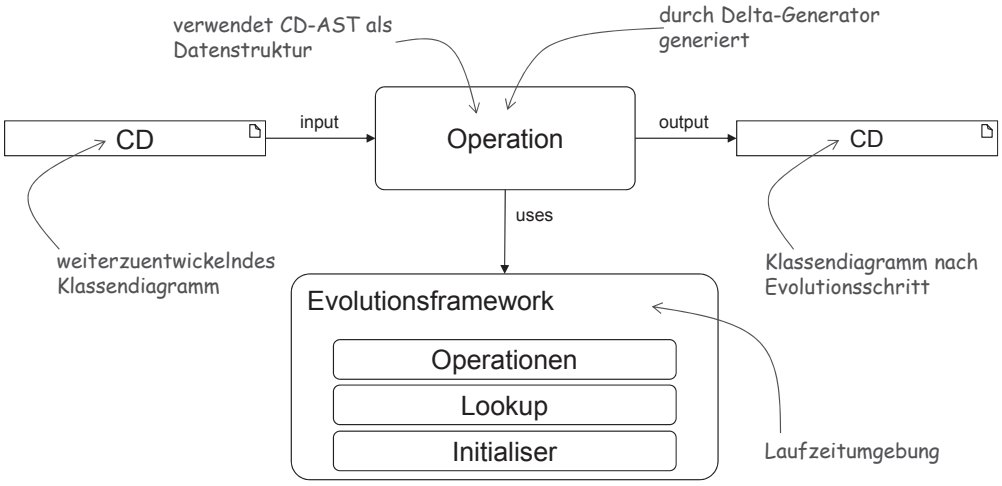


Abbildung 9.3: Darstellung des Konzepts zur Evolution von Klassendiagrammen.

plikationen entwickelt und verwendet. Diese Sprache verwendet Objektdiagramme als Eingabe und als Erwartungswert. Ein Testtreiber führt den Test aus und vergleicht den Sollzustand mit dem Istzustand. Der in [Plo12] erweiterte Testtreiber befüllt eine Testdatenbank mit dem Objektdiagramm, führt den Test aus und berücksichtigt dabei Datenbankänderungen. Anschließend wird der Zustand der Datenbank mit dem Erwartungswert verglichen. Basierend auf dieser Anpassung lassen sich, über die in [Rum12] definierten Testsemantiken wie `match:complete` hinausgehende, erweiterte Testsemantiken unterscheiden, die in [Plo12] genauer erläutert werden. Darüber hinaus wurde in [KLM+16] ein Framework zum Testen templatebasierter Codegeneratoren entwickelt. Im Rahmen dieser Arbeit wird auf eine tiefgreifende Beschreibung dieser Testmethodik verzichtet.

9.2 Evolution des Klassendiagramms

In diesem Abschnitt wird die Evolution des Klassendiagramms genauer beschrieben. Dazu wird zunächst die generelle Infrastruktur und anschließend ein Generator, der einen spezifischen Evolutionsschritt erstellt, vorgestellt.

Abbildung 9.3 zeigt das Konzept der Evolution des Klassendiagramms. Dieses wird als Eingabe für die generierte Operation verwendet. Die generierte Operation verwendet ein Framework als Laufzeitumgebung, welches vordefinierte Operationen, wie die Basis- und die weiterführenden Operationen aus Kapitel 6, Funktionalität zur Auflösung einer Namensreferenz sowie Initialisiererlogik, enthält. Zur Ausführung des Deltas verkettet die generierte Java-Klasse die zur Verfügung gestellte Funktionalität des Frameworks. Die Ausgabe der generierten Operation ist ein Klassendiagramm im neuen Zustand. Zum besseren Verständnis des verwendeten Frameworks wird dieses nachfolgend zusammen mit der generierten Operation detailliert vorgestellt.

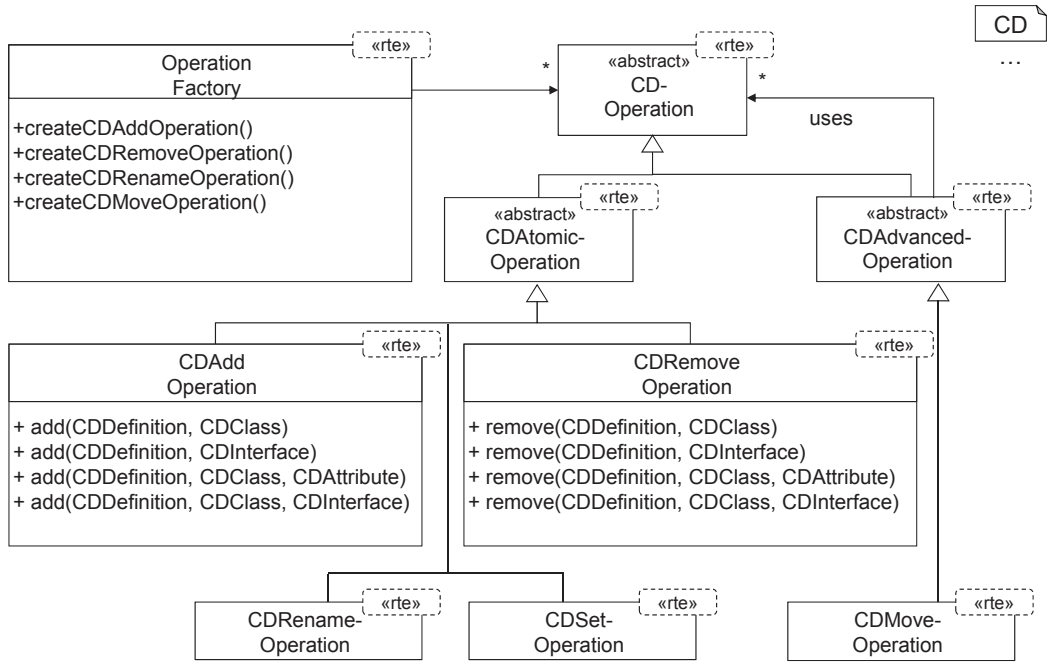


Abbildung 9.4: Darstellung der Klassen der Laufzeitumgebung des Frameworks zur Evolution von Klassendiagrammen.

Abbildung 9.3 zeigt die verwendeten Klassen des handgeschriebenen Evolutionsframeworks. Die Hauptklasse, die vom Framework bereitgestellt wird, ist die Klasse `OperationFactory`, die später von der generierten Klasse, welche die konkrete Evolution umsetzt und ausführt, verwendet wird. Dazu dient sie als Fabrik [GHJV95] und stellt statische Methoden, die einzelne Operationen erzeugen, bereit. Diese Klasse `CDOperation` stellt die gemeinsame Superklasse aller Klassendiagrammoperationen dar. Sie besitzt zwei Subklassen: `CDBasicOperation` und `CDAdvancedOperation`. Dabei stellt die erstgenannte Klasse die gemeinsame Superklasse der Basisoperationen `add`, `set`, `remove` und `rename`, die in Kapitel 6 vorgestellt wurden, dar. Jede Basisoperation ist in einer eigenen Klasse gekapselt und bietet überladene Methoden zur Ausführung der Operation in unterschiedlichen Kontexten an. In Abbildung 9.4 sind exemplarisch Auszüge der API der Klassen `CDAddOperation` und `CDRemoveOperation` dargestellt. Das erste Argument der jeweiligen Methoden ist stets der Kontext des hinzugefügten Elements, wohingegen das zweite Argument das hinzuzufügende Element darstellt. Der Auszug der Klasse `CDAddOperation` zeigt vier Methoden, wobei

- `add(CDDefinition, CDClass)` einem Klassendiagramm eine neue Klasse hinzufügt,
- `add(CDDefinition, CDInterface)` einem Klassendiagramm ein neues Interface hinzufügt,

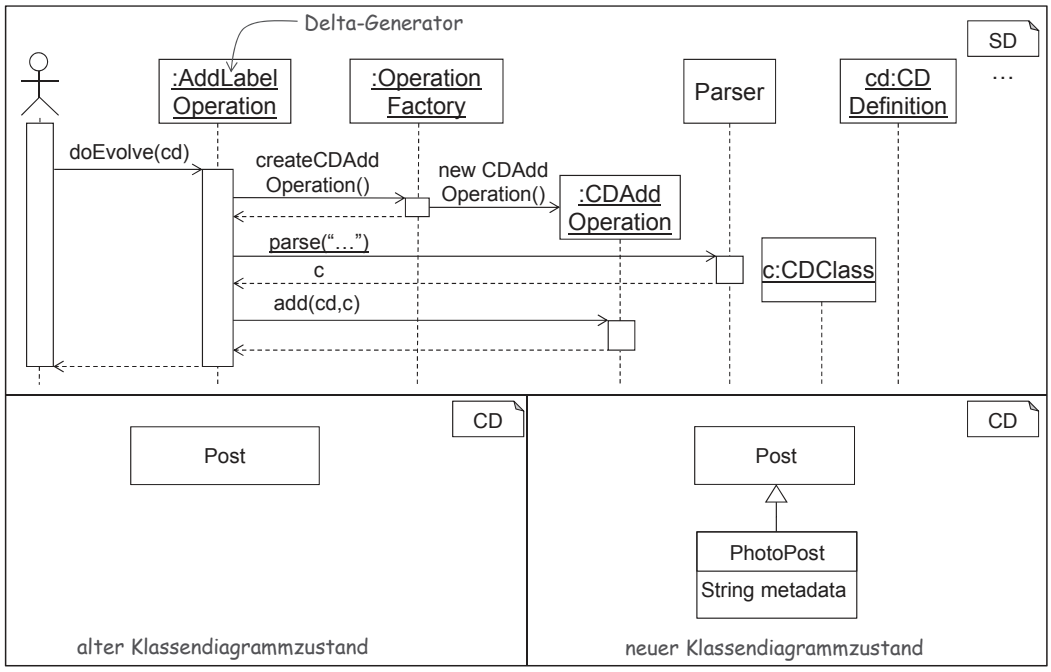


Abbildung 9.5: Darstellung eines exemplarischen Ablaufs der Transformation eines Klassendiagramms.

- `add(CDDefinition, CDClass, CDAttribute)` einer existierenden Klasse eines Klassendiagramms ein neues Attribut hinzufügt,
- `add(CDDefinition, CDClass, CDInterface)` einer Klasse eines Klassendiagramms eine Implementierungsbeziehung zu einem Interface hinzufügt.

Es wird deutlich, dass diese Operationen unterschiedliche Semantiken eines Hinzufügens abdecken. Dadurch wird das Hinzufügen neuer Elemente, aber auch das Hinzufügen neuer Beziehungen zwischen Elementen abgedeckt. Die ausgelassenen Methoden umfassen alle Möglichkeiten des Hinzufügens von Elementen und Beziehungen in einem Klassendiagramm. Die Methoden selbst und das Hinzufügen der Elemente sind manuell implementiert. Die dargestellten Methoden der `CDRemoveOperation` Klasse bewirken analoge Effekte.

Die abstrakte Superklasse `CDAdvancedOperation` stellt die gemeinsame Superklasse für weiterführende Operationen dar. Eine weiterführende Operation ist, wie schon in Abschnitt 6.2 beschrieben, aus mehreren Basisoperationen oder aber weiterführenden Operationen komponiert. Dies wird durch die `uses` Assoziation in Abbildung 9.4 ermöglicht. Exemplarisch als Subklasse dargestellt ist die Klasse `CDMoveOperation`. Sie besitzt eine zu den Basisoperationen analoge API und kann ebenso verwendet werden. Die abstrakten Klassen `CDBasicOperation` und `CDAdvancedOperation` dienen als Erweiterungspunkte zum Hinzufügen neuer Operationen.

Abbildung 9.5 zeigt einen exemplarischen Ablauf der Transformation eines Klassendiagramms. Dazu wird erneut das Deltamodell aus Listing 6.3 betrachtet. Es modifiziert das Klassendiagramm und fügt ihm eine neue Klasse `PhotoPost` sowie eine Assoziation hinzu. Das Hinzufügen der Assoziation ist in Abbildung 9.5 nicht gezeigt, verläuft aber analog zum Hinzufügen der Klassen. Zur Transformation des Klassendiagramms generiert der Delta-Generator, wie in Abbildung 9.5 dargestellt, die Klasse `AddLabelOperation`. Für die beiden anderen Veränderungen, die in Kapitel 6 vorgestellt wurden, wird der Quellcode analog generiert.

```java
public class AddLabelOperation {

  public CDDefinition doEvolve(CDDefinition cdDefinition) {

    CDAddOperation addOperation =
      OperationFactory.createCDAddOperation();

    cdDefinition = addOperation.add(cdDefinition, Parser.
        parseCDClass(
      "class PhotoPost extends Post{ String metadata }"
    ));

    cdDefinition = addOperation.add(cdDefinition, Parser.
        parseCDAssociation(
      "association labeledIn [*] Person -> PhotoPost [*];"
    ));

    return cdDefinition;
  }
}
```

Listing 9.6: Darstellung des generierten Quellcodes zur Klassendiagrammevolution. Der Quellcode verwendet die vom Framework bereitgestellten Methoden und verkettet diese adäquat.

Für jedes Delta wird eine eigene Klasse, deren Name sich aus dem Namen des Deltas und einem Suffix zusammensetzt, generiert. Daher heißt die Klasse in Listing 9.6 `AddLabelOperation`. Die eigentliche aus dem Modell generierte Operation wird in der Methode `doEvolve(ASTCDDefinition cdDefinition)`, die in Abbildung 9.5 von außen aufgerufen wird, umgesetzt. An dieser Stelle werden die einzelnen im Delta modellierten Operationen verkettet. Als Parameter erhält die Methode das geparste Klassendiagramm als Wurzelknoten des zu verändernden ASTs. Zusätzlich werden die benötigten Operationen mit Hilfe der `OperationFactory`, wie in Abbildung 9.5 gezeigt, instanziiert. Die `LookUp` Funktionalität zur Suche des entsprechenden Kontexts, beispielsweise beim Einfügen eines Attributs in eine Klasse, ist hier nicht gezeigt, sondern wird in Listing 9.11 dargestellt. In Abbildung 9.5 wird zunächst mit Hilfe der `Ope-`

rationFactory eine CDAddOperation erzeugt. Daran anschließend wird ein Teil des Deltas, wodurch ein AST-Knoten vom Typ CDClass erzeugt wird, geparst. Dieser AST-Knoten wird mit Hilfe der zuvor vorgestellten add-Methoden der CDAddOperation dem Klassendiagramm hinzugefügt. Dann wird das Klassendiagramm nach außen zurückgegeben. Durch Ausführen der add-Methode wird das Klassendiagramm verändert und als Rückgabetyp der Methode zurückgegeben.

Listing 9.6 zeigt den generierten Quellcode der Klasse AddLabelOperation mit der Methode doEvolve(CDDefinition cdDefinition) exemplarisch. Der Quellcode entspricht dem des Sequenzdiagramms aus Abbildung 9.5 und zeigt zusätzlich noch das Hinzufügen einer Assoziation. Der Vorgang des Parsens zur Erzeugung eines AST-Knotens ist deutlicher dargestellt. Die konkrete Syntax der hinzuzufügenden Klasse wird mit Hilfe eines Parsers geparst und der Methode als Parameter übergeben. Dieses Vorgehen wirkt auf den ersten Blick nicht intuitiv, ist aber an dieser Stelle notwendig. Zur Generierungszeit der Klasse wird durch die Spracheinbettungsmechanismen von MontiCore, wie in Abschnitt 2.2 beschrieben, der AST-Knoten der hinzuzufügenden Klasse eingebettet und ist verwendbar. Allerdings ist die Laufzeit der generierten Klasse davon losgelöst und die AST-Knoten nicht mehr bekannt. Daher muss der im Delta verwendete AST-Knoten im generierten Quellcode serialisiert und zur Laufzeit wieder geparst werden. Ein alternativer Ansatz zu diesem Vorgehen wäre das Nachladen des Deltas zur Laufzeit der generierten Klasse, aber auch dies würde ein Parsen bewirken. Darüber hinaus wäre der Einsatz eines Interpreters möglich. Dazu würde kein Quellcode aus dem Delta generiert, sondern das Delta würde für die Evolution des Klassendiagramms zur Laufzeit interpretiert werden. Dies bringt aber Geschwindigkeitseinbußen mit sich.

Darüber hinaus wird in Listing 9.6 das Hinzufügen der Assoziation dargestellt. Im nachfolgenden Abschnitt wird die Migration der Daten beschrieben. Dazu wird sowohl die Migration der Objektdiagramme als auch die Serialisierung und die Deserialisierung der persistent gespeicherten Daten detailliert erläutert.

9.3 Migration der Objektdiagramme

Nachdem zuvor die Evolution der Klassendiagramme beschrieben wurde, wird in diesem Abschnitt die Migration der Objektdiagramme sowie deren Erstellung erläutert. Dabei wird mit der Migration der Objektdiagramme begonnen. Analog zur zuvor in Abschnitt 9.2 beschriebenen Evolution von Klassendiagrammen, verwendet auch der Migrationsmechanismus der Objektdiagramme das handgeschriebene Evolutionsframework.

Das Konzept der Migration der Objektdiagramme ist in Abbildung 9.7 dargestellt. Analog zur Klassendiagrammevolution verwendet die generierte Operation ein Objektdiagramm als Eingabe, migriert dieses mit Hilfe des Frameworks und erzeugt ein migriertes Objektdiagramm. Darüber hinaus muss das Objektdiagramm aus den persistenten Daten serialisiert und nach der Migration wieder in die Datenbank deserialisiert werden. Zunächst wird die Migration eines Objektdiagramms beschrieben. Daran anschließend wird die Serialisierung und die Deserialisierung der Objektdiagramme beschrieben.

Abbildung 9.8 zeigt einen kleinen Ausschnitt der bereits zuvor vorgestellten Opera-

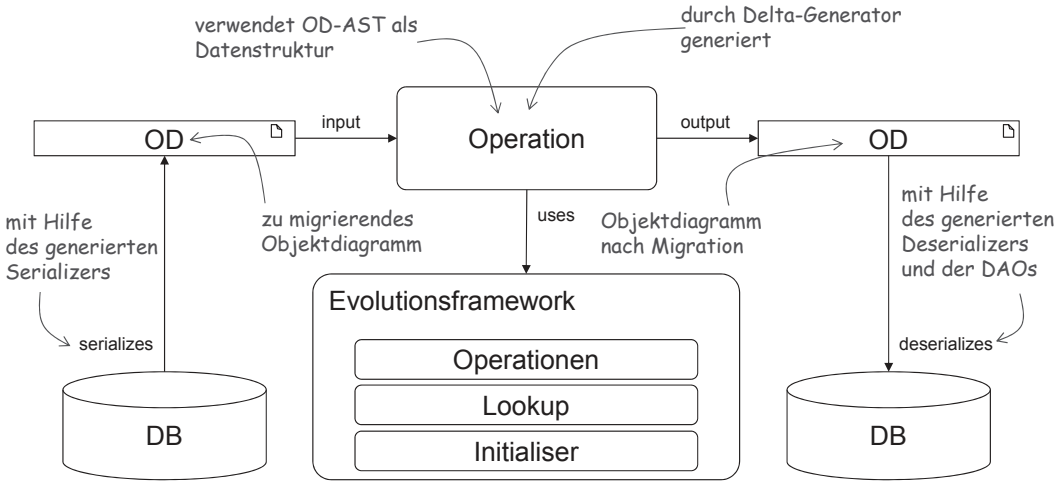


Abbildung 9.7: Darstellung des Konzepts zur Migration von Objektdiagrammen.

`tionFactory`, die neben den Operationen für Klassendiagramme auch die Migrationsoperationen für Objektdiagramme erzeugt. Die Klassen `ODOperation`, `ODAtomicOperation` und `ODAdvancedOperation` dienen analog zu den bereits in Abschnitt 9.2 vorgestellten Klassen als Erweiterungspunkte für weitere Operationen. Zum jetzigen Zeitpunkt sind die gleichen Operationen, die bereits für Klassendiagramme implementiert sind, auch für Objektdiagramme implementiert, so dass eine Datenmigration auf Basis des Deltas erfolgen kann. Zudem ist in Abbildung 9.8 exemplarisch die Signatur einer Methode der `ODAddOperation` dargestellt. Sie fügt allen Objekten eines Typs innerhalb eines Objektdiagramms ein Attribut hinzu. Die Signatur unterscheidet sich in ihren Details von den in Abbildung 9.4 vorgestellten Signaturen. Die abgebildete Methode benötigt das Objektdiagramm, welches migriert wird. Dazu wird der erste Parameter, der Wurzelknoten des ASTs des ODs, benötigt. Zusätzlich werden sowohl das Klassendiagramm als auch die Klasse, der innerhalb des Deltas ein Attribut hinzugefügt wurde wie auch das hinzugefügte Attribut selbst von der Methode benötigt. Listing 6.6 zeigt ein Beispiel eines hinzugefügten Attributs innerhalb der Klasse `Commercial`. Das Klassendiagramm wird innerhalb der Methode benötigt, um etwaige Subklassen der veränderten Klasse zu identifizieren, da auch in deren Instanziierungen das Attribut entsprechend hinzugefügt werden muss. Die Klasse sowie das Attribut dienen als Typdefinition des neu hinzuzufügenden Attributs. Es sei daran erinnert, dass sich das Delta stets auf ein Klassendiagramm bezieht und somit Typen, also Klassen, wie auch Attribute, in Klassen hinzufügt und die Auswirkungen auf ein Objektdiagramm nur abgeleitet werden. Der letzte Parameter ist der Initialwert, welcher von einem Initialisierer erstellt wurde und mit dem das Attribut initialisiert wird. Dabei kann es sich um unterschiedliche Literale handeln.

Abbildung 9.9 zeigt die Architektur dieser Initialisierer. Das Interface `Intializer` definiert die Methode `getValue( ODDefinition, CDDefinition, CDClass,`

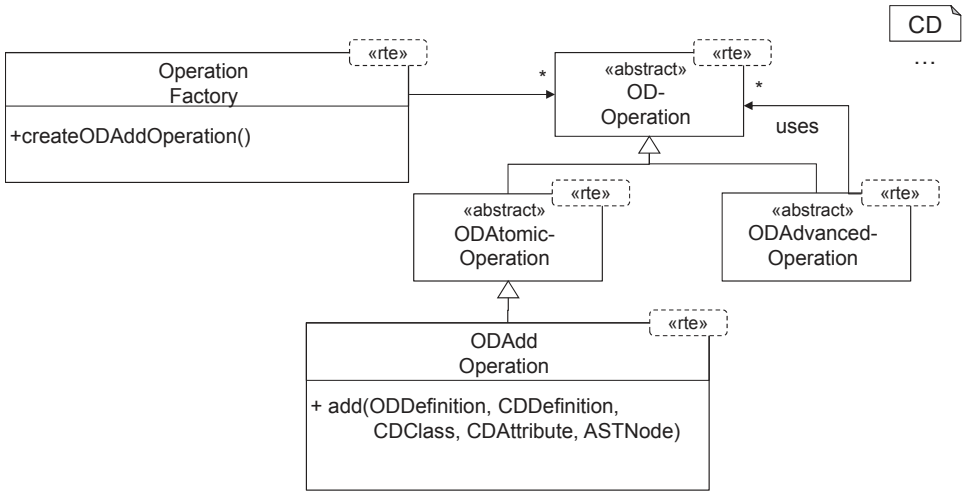


Abbildung 9.8: Darstellung der Klassen der Laufzeitumgebung des Frameworks zur Migration von Objektdiagrammen.

`CDAttribute)`. Diese Methode erhält, analog zu der zuvor vorgestellten `add` Operation das OD, das CD, die Klasse und das Attribut als Typdefinition. Drei abstrakte Klassen implementieren dieses Interface: `DefaultInitializer`, `RefactoringInitializer` und `CustomInitializer`. Die Klasse `DefaultInitializer` dient als Basisklasse für eine Standardinitialisierung existierender Datentypen. Die Deltasprache sieht vor, dass, wie in Kapitel 6 vorgestellt, beim Hinzufügen von Attributen angegeben werden kann, ob dieses initialisiert werden soll. Falls dies der Fall ist, kann der Name des Initialisierers angegeben werden. Im momentanen Stand sieht die Sprache lediglich einen Namen des Initialisierers vor, kann aber leicht dahingehend erweitert werden, dass gerade die Standardinitialisierung automatisiert auf Basis des Datentyps des hinzugefügten Attributs ausgewählt wird. Somit können die vordefinierten Initialisierer: `StringInitializer`, `BooleanInitializer`, `DoubleInitializer` und `IntegerInitializer` automatisiert verwendet werden. Auch die Möglichkeit der Klassendiagrammsprache, einem Attribut direkt einen Wert zuzuweisen, kann verwendet werden.

Ist keine Standardinitialisierung gewünscht, können `CustomInitializer` manuell implementiert werden. Ein Beispiel dafür ist der `FollowerInitializer` aus Listing 6.6. Dieser Initialisierer muss die Signatur des Interfaces erfüllen. Zur Berechnung des initialen Wertes stehen ihm die Parameter zur Verfügung. Dadurch erhält er die Möglichkeit, Berechnungen sowohl auf dem AST des Objektdiagramms als auch auf dem AST des Klassendiagramms durchzuführen und kann dadurch Attribute individuell initialisieren. Darüber hinaus existieren `AdvancedInitializer`, die innerhalb der weiterführenden Operationen programmatisch verwendet werden können und dazu dienen, die dort bearbeiteten Attribute zu initialisieren.

Abbildung 9.10 zeigt einen schematischen Ablauf der Objektdiagrammmigration sowie der Initialisierung der Attribute. Sowohl die `OperationFactory` als auch die Initialisie-

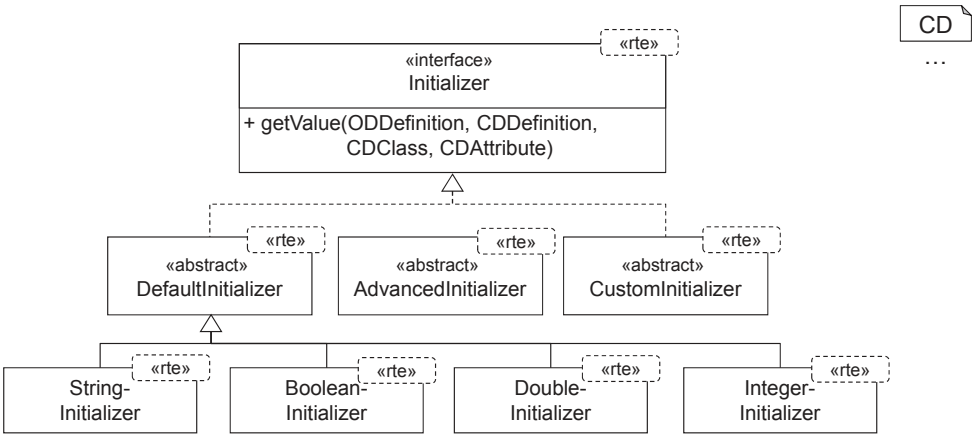


Abbildung 9.9: Darstellung der `Initializer` Klassen der Laufzeitumgebung des Frameworks zur Datenmigration.

rer werden vom Codegenerator, der die konkrete Evolution aus dem Delta generiert, verwendet. Listing 9.11 zeigt dazu einen Ausschnitt des generierten Codes. Wie bereits zuvor wird aus dem Delta die Klasse `ODAddMaxFollowersOperation` generiert. Dieser Name leitet sich aus dem Namen des Deltas, `AddMaxFollowers`, der in Listing 6.6 ausgelassen wurde, ab. Dabei sind die Interna der generierten `doEvolve(CDDefinition cdDefinition, ODDefinition odDefinition)` Methode dargestellt. Im Gegensatz zur generierten Methode der Evolution von Klassendiagrammen in Listing 9.6 benötigt diese Methode das Objektdiagramm `odDefinition` als weiteren Parameter. Zunächst wird, analog zu den Klassendiagrammen, die konkrete Operation berechnet. Darüber hinaus wird der umgebende AST-Knoten berechnet. Dazu wird eine Variable zur Speicherung des Kontexts definiert. Danach wird der Pfad des entsprechenden Kontexts berechnet. Auf Basis des berechneten Pfads wird mit Hilfe der `LookUp` Klasse der entsprechende AST-Knoten aufgelöst. Auf diesem Knoten wird die entsprechende Operation ausgeführt.

Im nächsten Schritt wird der Initialwert des Attributs mit Hilfe der zuvor vorgestellten Initialisierer berechnet. Dazu wird zunächst das zu initialisierende Attribut des Klassendiagramms berechnet. Dann wird ein Objekt des im Delta angegebenen Initialisierers `FollowerInitializer` erzeugt und die entsprechende Methode aufgerufen. Wie bereits zuvor vorgestellt, erhält dazu jeder Initialisierer das Klassendiagramm, die entsprechende Klasse und das zu initialisierende Attribut. Abschließend wird dann die eigentliche Operation, die das Objektdiagramm migriert, ausgeführt. Dazu erhält die Operation das Objektdiagramm, die Elemente des Klassendiagramms und den Initialwert des Attributs. Die Operation selbst ist dafür verantwortlich, alle Objekte des Objektdiagramms zu migrieren. Listing 9.11 zeigt den generierten Quellcode der Klasse `ODAddMaxFollowersOperation` mit der Methode `doEvolve(CDDefinition cdDefinition, ODDefinition odDefinition)` exemplarisch. Ausgelassen ist in Listing 9.6 die Berechnung des Pfades. Sie kann auf mehrere Arten geschehen. Dies hängt

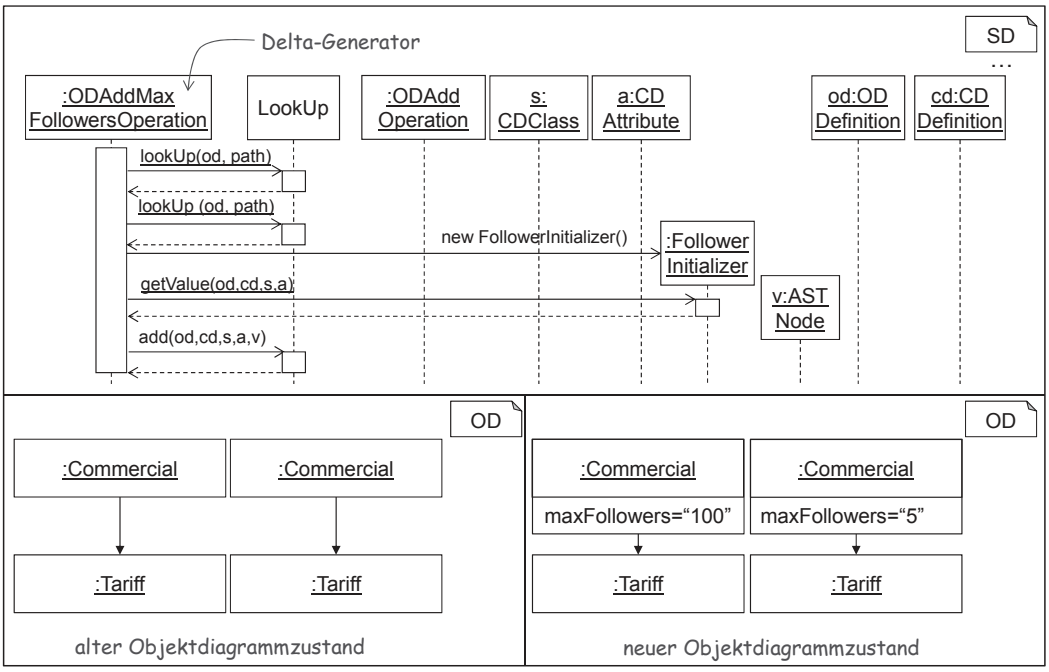


Abbildung 9.10: Darstellung eines exemplarischen Ablaufs der Datenmigration und der Attributinitialisierung.

von den jeweiligen `modify` Statements ab. Diese können, wie in Abschnitt 6.2 beschrieben, beliebig ineinander verschachtelt sein, vollqualifizierte Namensreferenzen oder sogar konkrete Elemente in deren konkreter Syntax enthalten. Deshalb ist die Berechnung des Pfades daran angepasst.

9.4 Serialisierung und Deserialisierung von Objektdiagrammen

Nachdem zuvor die Evolution und Migration von Klassendiagrammen und Objektdiagrammen beschrieben wurde, wird im Folgenden die Serialisierung und die Deserialisierung des Datenbestands der Datenbank erläutert. Zunächst wird die Laufzeitumgebung der Infrastruktur, wie in Abbildung 9.12 dargestellt, gezeigt.

Die Laufzeitumgebung der Infrastruktur bietet drei Elemente, die von unterschiedlichen Teilen verwendet werden müssen, an. Das erste Element ist das Interface `IODSerializable`. Dieses Interface muss von allen handgeschriebenen Klassen, die nicht in einem Klassendiagramm modelliert wurden, aber dennoch serialisiert und deserialisiert werden sollen, implementiert werden. Dadurch wird es den weiteren Elementen möglich, diese Elemente ebenfalls zu verarbeiten. Zusätzlich bietet die Laufzeitumgebung das Interface `ISerializationManager` an, dass von handgeschrieben Klassen, die selbständig die Serialisierung bzw. Deserialisierung durchführen sollen, implementiert werden. Das Interface bietet die beiden Einstiegsmethoden `serialize(Set<Object>)` und `serialize(ODDefinition)` an, die aufgerufen werden, um eine Menge von Ob-

```java
public class ODAddMaxFollowersOperation {
...
  public ODDefinition doEvolve(CDDefinition cdDefinition,
      ODDefinition odDefinition) {

    ODAddOperation addOperation =
          OperationFactory.createODAddOperation();
        ASTNode scopeNode;

    // [...] calculate path, based on modify statement
    scopeNode = LookUp.lookUpCDClass(cdDefinition, path);

    // handle Initialization and recalculate path
    CDAttribute initializedAttribute =
      LookUp.lookUpCDAttribute(cdDefinition, path);
    ASTNode initialValue;
    initialValue = new FollowerInitializer()
      .getValue(odDefinition, classDiagram,
        (CDClass) scopenode, initializedAttribute);

    odDefinition = addoperation.add(odDefinition,
      cdDefinition, (CDClass) scopeNode,
      initializedAttribute, initialValue));

    return odDefinition;
  }
}
```

Listing 9.11: Darstellung des generierten Quellcodes zur Objektdiagrammmigration. Der Quellcode verwendet die vom Framework bereitgestellten Methoden und verkettet diese adäquat. Zudem ist die Verwendung der Initialisierer dargestellt.

jektgraphen zu serialisieren bzw. in ein einzelnes Objektdiagramm zu deserialisieren. Das dritte Element der Laufzeitumgebung ist die abstrakte Klasse `AExternalSerializer`, die ebenfalls abstrakte Methoden zur Serialisierung und zur Deserialisierung bereitstellt. Dabei dient die Methode `serialize(Set<Object>)` und die Methode `serialize(ODDefinition)` wieder dazu, eine Menge von Objektgraphen bzw. ein Objektdiagramm zu bearbeiten. Die Methode `handlesDeserialization(ODObject)` dient dazu, dass ein handgeschriebener Deserialisierer signalisieren kann, dass er den Typ eines Objekts eines Objektdiagramms deserialisieren kann. Dies wird benötigt, da auch mehrere handgeschriebene Deserialisierer verwendet werden können. Die Methode `handleDeserializationLink(Object, Object)` hingegen dient dem Aufspannen von Links zwischen Objekten handgeschriebener Klassen, bzw. zwischen einer handgeschriebenen und einer modellierten Klasse. Im Wesentlichen muss in dieser Methode

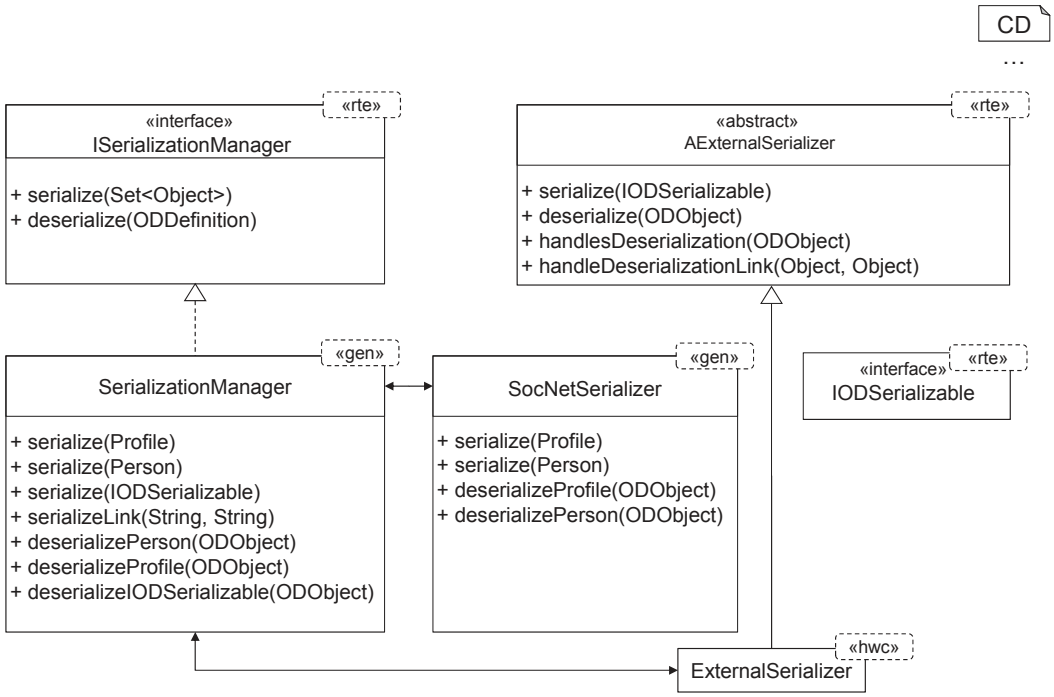


Abbildung 9.12: Darstellung der Serialisierungs- und Deserialisierungsinfrastruktur. Die Klassen der Laufzeitumgebung sowie generierte Klassen sind abgebildet.

die Setter-Methode des Objekts der handgeschriebenen Klasse mit dem Objekt der modellierten Klasse aufgerufen werden.

Die restlichen Klassen der Infrastruktur, die spezifisch für ein modelliertes Klassendiagramm sind, werden generiert. Dabei wird ein `SerializationManager`, der spezifische Serialisierungs- und Deserialisierungsmethoden für jeden im Klassendiagramm modellierten Typ enthält, generiert. Zudem enthält er Methoden zur Serialisierung und Deserialisierung von handgeschriebenen Klassen, die das Interface `IODSerializable` implementieren. Die Hauptaufgabe des Managers ist die Delegation an die jeweiligen konkreten Manager. Diese sind in diesem Fall der spezifisch für das Klassendiagramm `SocNet` generierte `SocNetSerializer` und der handgeschriebene `ExternalSerializer`. Die generierte Klasse `SocNetSerializer` enthält für jeden modellierten Datentyp eine generierte Methode zur Serialisierung und Deserialisierung dieses Typs. Dabei werden die public Getter- und Setter-Methoden der Objekte verwendet, von denen angenommen wird, dass sie vom Entity-Generator generiert werden. Dadurch können keine Attribute serialisiert oder deserialisiert, die keine public Zugriffsmethoden besitzen, werden. Als Namen der Objekte wird der Hashcode der Objekte verwendet, da er eindeutig ist und die entstehenden Objektdiagramme nicht für menschliche Lesbarkeit ausgelegt sind. Darüber hinaus übernehmen diese Methoden nicht die Serialisierung von Links zwischen Objekten. Dies wird von den nachfolgenden Methoden übernommen.

Abbildung 9.13 zeigt dazu einen exemplarischen Ablauf der Serialisierung und der De-

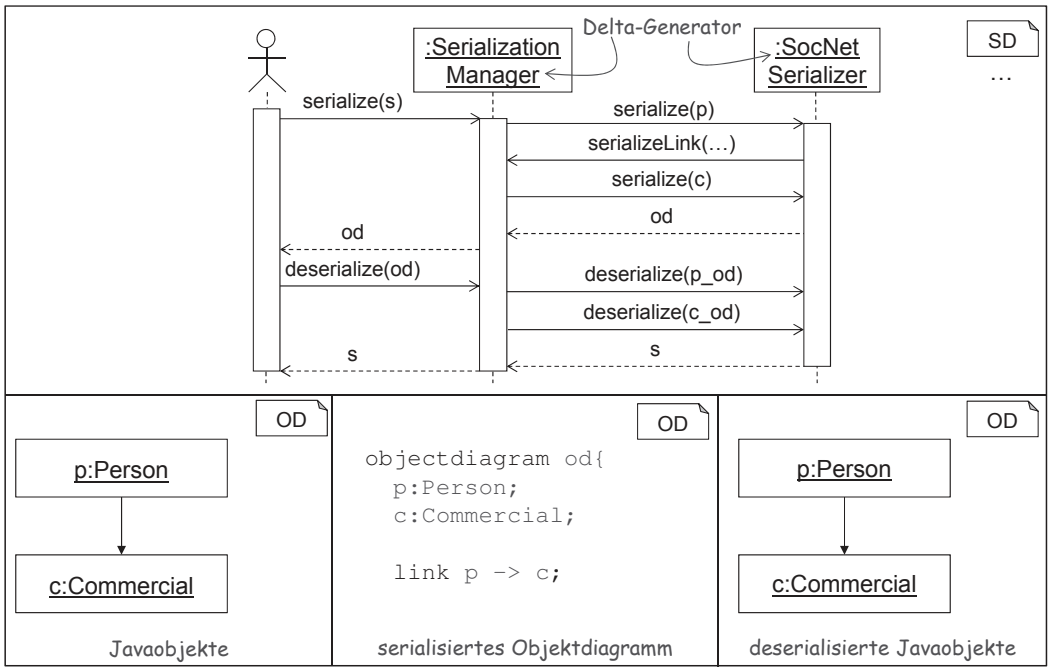


Abbildung 9.13: Darstellung eines exemplarischen Ablaufs der Serialisierung und der Deserialisierung von Daten.

serialisierung. Im unteren Bereich sind Java-Objekte in Form eines graphischen Objektdiagramms, die durch den Aufruf der `serialize(s)` Methode in textuelle Objektdiagramme serialisiert werden, gezeigt. Durch den Aufruf der `deserialize(od)` Methode werden die Java-Objekte wieder instanziiert. Dies geschieht mit Hilfe der generierten Serialisierungsmanger. Der `SerializationManager` delegiert die Serialisierung eines einzelnen Objekts an den `SocNetSerializer`. Dieser ruft zur Serialisierung von Links die generierte Methode `serializeLink(String, String)`, die einen Link zwischen zwei Hashcodes speichert, auf. Der `SerializationManager` delegiert anschließend das nächste Objekt an den `SocNetSerializer`. Am Ende der Serialisierung werden dann alle Links in das entstehende Objektdiagramm geschrieben. Hier ist es von Vorteil, dass die Objektdiagrammsprache die Links einzeln innerhalb des Modells modelliert und nicht in einzelne Objekte einbettet. Die Deserialisierung der Links erfolgt ebenfalls im `SerializationManager` durch Aufruf der `deserialize(od)` Methode. Dieser merkt sich bei der Deserialisierung die deserialisierten Objekte und deren im Objektdiagramm modellierten Hashcodes. Auf dieser Basis ruft der Manager die entsprechenden Setter, die einem Namensschema folgend generiert wurden, in den jeweiligen Objekten auf. Für externe Objekte sucht er mit Hilfe der `handlesDeserialization` Methode den richtigen handgeschriebenen Serialisierer und verwendet die `handleDeserializationLink` Methode, damit dieser den Link deserialisieren kann.

Durch diese Infrastruktur lassen sich auf Basis eines modellierten Klassendiagramms Manager, die Java-Objektgraphen serialisieren und wieder deserialisieren können, generieren. Zum Laden und Speichern der Objekte aus der Datenbank wird die DAO-Infrastruktur verwendet. Somit werden die Objekte geladen, serialisiert und anschließend, nach der Evolution, wieder deserialisiert und erneut in der Datenbank gespeichert.

9.5 Zusammenfassung

In diesem Kapitel wurde der Delta-Generator vorgestellt. Der Delta-Generator verwendet Deltas zur Generierung einer Infrastruktur, die den Produktentwickler bei der Systemevolution und der Datenmigration, unterstützt. Diese Infrastruktur verarbeitet Klassendiagramme und Objektdiagramme. Da Klassendiagramme im Rahmen von MontiEE das System modellieren, stellt die Evolution selbiger eine Systemevolution dar. Zur Evolution eines Klassendiagramms wird eine Operation, die intern ein manuell implementiertes Framework als Laufzeitumgebung verwendet, generiert. Dieses Framework beinhaltet unterschiedliche Basis- und weiterführende Operationen, die von einer generierten Operation entsprechend verkettet werden. Das Resultat der generierten Operation ist ein modifiziertes Klassendiagramm.

Darüber hinaus wurde die Datenmigration auf Basis von Objektdiagrammen vorgestellt. Eine solche Migration erfordert neben der strukturellen Veränderung auch die Initialisierung von Attributen. Dazu wurde das Konzept der Initialisierer vorgestellt. Des Weiteren wurde die Serialisierung und die Deserialisierung persistenter Daten der Datenbank in ein Objektdiagramm vorgestellt. Der Delta-Generator kann unabhängig von den anderen MontiEE-Generatoren, die in den Kapiteln 7 und 8 verwendet werden, da er ein Entwicklungswerkzeug generiert. Das Generat lässt sich so verketten, dass es auch mehrere Evolutionsschritte, also mehrere Versionssprünge gleichzeitig behandeln kann. Dazu wird eine übergeordnete Infrastruktur, die Webservices verkettet, generiert. Jeder dieser Webservices ist, nach dem Chain-of-Responsibility Muster [GHJV95], für exakt einen Schritt verantwortlich und delegiert den nächsten Schritt an den jeweiligen Verantwortlichen. Im Rahmen dieser Arbeit werden diese ersten Ideen nicht weiter vertieft. Darüber hinaus kann die Deltasprache als Eingabesprache für den Entwickler verstanden werden. Andere Arbeiten [Wei12, HRW15], die sich mit der Transformation von Klassen- und Objektdiagrammen beschäftigen, können intern dazu verwendet werden, die Modelltransformation umzusetzen. Dies würde das momentan manuell entwickelte Framework mit vorgefertigten Operationen ablösen und durch ein modellgetriebenes Framework ersetzen.

Kapitel 10

Eine MontiEE-basierte Methodik

Nachdem in den vorherigen Kapiteln und Abschnitten die Sprachfamilie MontiEE sowie die umgesetzten Generatoren vorgestellt wurden, wird in diesem Kapitel die Methodik der Entwicklung von Enterprise Applikationen mit MontiEE vorgestellt. Dabei wird

(1) auf die Verwendung,

(2) die Konfiguration,

(3) und die Ausführung

von MontiEE eingegangen. Darüber hinaus wird auf die Umsetzung des Szenarios und das Deployment der Applikation eingegangen. Abschließend werden Fallstudien, die die Integration von MontiEE in bestehende Projekte und in Neuentwicklungen zeigen, präsentiert.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Methodik der Generierung der Persistenz mit MontiEE (Abschnitt 10.1).
- Methodik der Generierung der Kommunikationsinfrastruktur mit MontiEE (Abschnitt 10.1).
- Methodik der Generierung der Evolutionsinfrastruktur mit MontiEE (Abschnitt 10.1).
- Darstellung der Konfiguration und Ausführung von MontiEE (Abschnitt 10.2).
- Vorstellung des Einsatzes von MontiEE in Fallstudien (Abschnitt 10.6).

Der Rest des Kapitels ist wie folgt strukturiert: Zunächst wird die Verwendung von MontiEE gefolgt von der Umsetzung des Szenarios, dem Deployment des Generats, der Werkzeugunterstützung und dem Einsatz in Fallstudien vorgestellt. Daran anschließend wird der Einsatz von MontiEE in Fallstudien im Rahmen von Forschungsprojekten vorgestellt.

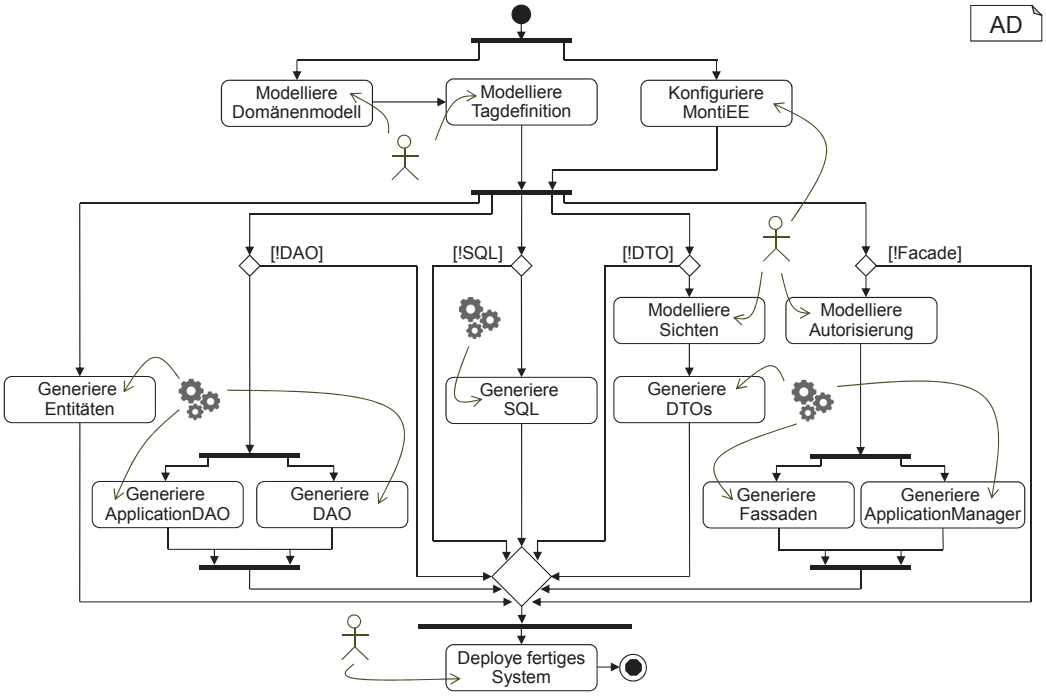


Abbildung 10.1: Darstellung der zu treffenden Entscheidungen und der auszuführenden Aktivitäten bei der Verwendung von MontiEE. Manuelle und automatisch durchführbare Aktivitäten sind entsprechend markiert.

10.1 Methodik

Abbildung 10.1 zeigt das Aktivitätsdiagramm, das die Verwendung von MontiEE darstellt. Generell ist MontiEE darauf ausgelegt, dass ein Domänenmodell, das als Klassendiagramm modelliert wurde, die Basis des Systems darstellt. Die Methodik zur Modellierung des Domänenmodells wird in Abschnitt 10.3 gezeigt.

Neben dem Domänenmodell, das als Klassendiagramm modelliert wurde, muss die zugehörige Tagdefinition, welche zum MontiEE-Tagschema aus Abschnitt 7.2 konform ist, zur Verfügung stehen. Die Tagdefinition reichert das Domänenmodell mit technologiespezifischen Informationen an. Die Tagdefinition des Szenarios wird in Abschnitt 10.3 in Listing 10.10 gezeigt. Beide Modelle werden von allen Generatoren zur Generierung einzelner Bestandteile des Applikationsservers und der Datenbank verwendet. Darüber hinaus muss MontiEE noch konfiguriert werden. Im Rahmen dieser Konfiguration werden verschiedene Parameter und die Verwendung der einzelnen Generatoren definiert. Die allgemeinen Konfigurations- und Ausführungsmöglichkeiten von MontiEE sowie die Konfiguration zur Umsetzung des Szenarios werden in Abschnitt 10.2.1 beschrieben. Diese verschiedenen Aktivitäten können parallel ausgeführt werden. So können das Klassendiagramm, die Tagdefinition und die Konfiguration entsprechend iterativ entwickelt werden. Diese drei Elemente stellen die Basiskonfiguration von MontiEE dar.

Die weiteren Modelle und Generatoren können optional verwendet werden.

Nachdem diese drei Elemente modelliert und erstellt wurden, können die Aktivitäten der einzelnen Generatoren ebenfalls parallel durchgeführt werden. Die Generierung der Entitäten basierend auf den zuvor erstellten Modellen stellt eine Möglichkeit dar, MontiEE zu verwenden. Wie bereits zuvor beschrieben, kann MontiEE zur Generierung unterschiedlicher Aspekte des Servers verwendet werden:

- Generierung der Persistenz
- Generierung der Kommunikationsinfrastruktur
- Generierung der Evolutionsinfrastruktur

Abbildung 10.1 stellt dar, welche unterschiedlichen Aktivitäten zur Verwendung der einzelnen Generatoren durchgeführt werden müssen. Nachfolgend werden die Schritte, die zur Generierung unterschiedlicher Aspekte des Servers benötigt werden, detailliert beschrieben. Dabei ist die Modellierung des Domänenmodells, der Tagdefinition und die Konfiguration von MontiEE, wie sie in Kapitel 4 gezeigt wurde, obligatorisch.

Methodik der Generierung der Persistenz mit MontiEE

Zur Generierung der Persistenz werden Entitäten, DAOs und SQL-Skripte von MontiEE zur Verfügung gestellt. Die Generierung der Entitäten, die in Abschnitt 7.4 gezeigt wird, wird als Basiskonfiguration angenommen, während DAOs oder SQL-Skripte, deren Generierung in den Abschnitten 7.5 und 7.6 präsentiert wurde, optional sind. Die Entscheidung, ob die einzelnen Komponenten benötigt werden, hängt von der gewählten Datenbank und von den eingesetzten Entwurfsmustern ab. Andere Datenbankparadigmen können schemafrei sein oder kein SQL unterstützen. Beide Komponenten können auch durch handgeschriebene Elemente ersetzt oder ergänzt werden. Sollen DAOs generiert werden, wird, Abbildung 10.1 folgend, das Domänenmodell, wie in Abschnitt 7.5 beschrieben, verwendet. Darüber hinaus wird der BusinessAPI-Generator dazu verwendet, das `ApplicationDAO`, wie in Abschnitt 8.3 beschrieben, als Schnittstelle der Geschäftslogik zur persistenten Datenspeicherung zu generieren. Beide Generatoren können parallel ausgeführt werden. Der DAO-Generator benötigt keine zusätzlichen Eingabemodelle. Die Verwendung von DAOs stellt ein Entwurfsmuster für die Architektur des Zielsystems dar. Sie ist nicht zwingend benötigt, aber empfohlen. Daher kann sich für oder gegen die Generierung der DAOs entschieden werden. Die Verwendung von DAOs zur Umsetzung des Szenarios wird in Abschnitt 10.3 gezeigt.

Darüber hinaus können die SQL-Skripte zur Erstellung und Entfernung des Schemas generiert werden. Der SQL-Generator, wie in Abschnitt 7.6 beschrieben, erzeugt auf Basis des Domänenmodells zwei SQL-Skripte. Die Verwendung des SQL-Generators zur Umsetzung des Szenarios wird in Abschnitt 10.3 beschrieben.

Methodik der Generierung der Kommunikationsinfrastruktur mit MontiEE

Soll die Infrastruktur zur Kommunikation mit verschiedenen Clients generiert werden, so können dazu DTOs und Zugriffsfassaden generiert werden. Auch die Verwendung dieser Generatoren ist optional, da die Verwendung von DTOs eine Entwurfsentscheidung darstellt, die technologisch nicht zwingend notwendig ist. Diese Entwurfsentscheidung sollte aber für eine skalierbare, nachhaltige Enterprise Applikation getroffen werden. Generell werden Zugriffsfassaden zur Kommunikation mit Clients für jeden Applikationsserver benötigt, dennoch können auch die Fassaden manuell umgesetzt werden. Aus diesem Grund bietet sich die Verwendung beider Generatoren zwar an, ist aber optional. Sollen DTOs verwendet werden und müssen mehrere verschiedene Clients unterstützt werden, lassen sich mit Hilfe der Sichtensprache die entsprechenden Sichten auf das Klassendiagramm modellieren und die DTOs daraus generieren. In Abschnitt 5.2 wurden modellierte Sichten für das Szenario aus Abschnitt 3.5 vorgestellt. Die Sichten werden, wie in Abschnitt 5.2.3 beschrieben, in Klassendiagramme transformiert und von verschiedenen Generatoren als Eingabe verwendet. Durch die Entscheidung, DTOs zu generieren, werden die modellierten Sichten vom DTO-Generator zur Generierung der DTOs, der Requests und der Assembler verwendet. Die Umsetzung des Szenarios unter Verwendung von DTOs wird in Abschnitt 10.3 gezeigt.

Wird sich für eine Generierung der Zugriffsfassaden, die in Abschnitt 8.4 präsentiert wurde, entschieden, müssen die im System vorhandenen Rechte und Rollen, wie in Kapitel 5 gezeigt, modelliert werden. Zudem muss die Zuordnung zwischen diesen modelliert werden. Für die im System vorhandenen Rechte kann die automatisierte Ableitung von Rechten auf Basis des Domänenmodells und deren manuelle Erweiterung, wie in Abschnitt 5.3.4 beschrieben, verwendet werden. In einem Mappingdiagramm werden diese Rechte, wie in Abschnitt 5.3.6 für das Szenario beschrieben, den entsprechenden Rollen zugeordnet. Der Facade-Generator verwendet diese Modelle, wie in Abschnitt 8.4 beschrieben, zur Generierung der Zugriffsfassaden. Darüber hinaus verwendet er das Domänenmodell. Wurden Sichten modelliert, werden das Domänenmodell sowie die aus Sichten erzeugten Klassendiagramme verwendet. Dies hängt von der Entscheidung, ob DTOs benötigt werden und demzufolge Sichten modelliert wurden, ab. Wurde sich gegen die Generierung von Zugriffsfassaden entschieden, muss dennoch eine Möglichkeit geschaffen werden, dass Clients mit dem Server kommunizieren können. Somit müssen in diesem Fall handgeschriebene Zugriffsfassaden umgesetzt werden. Gleichzeitig werden, wie in Abschnitt 8.3 erläutert, die Manager als API zur Geschäftslogik generiert. Diese API wird nur benötigt, wenn auch die Zugriffsfassaden erzeugt wurden, da dies sonst der manuellen Umsetzung überlassen ist. Die Umsetzung des Szenarios unter Verwendung der Zugriffsfassaden wird in Abschnitt 10.3 gezeigt.

Der letzte Schritt des Aktivitätsdiagramms ist das Deployment des fertigen Systems. Dabei wird der generierte Quellcode kompiliert und in verschiedene Java Archives (JARs) gekapselt. Diese werden wiederum in ein Enterprise Archive (EAR) gekapselt. Hinzu kommen verschiedene XML-Dateien, die zur Applikations- und Serverkonfiguration generiert werden. Das Kapseln der Applikation, die Erzeugung der XML-Dateien und das Deployment werden in Abschnitt 10.4 genauer beschrieben.

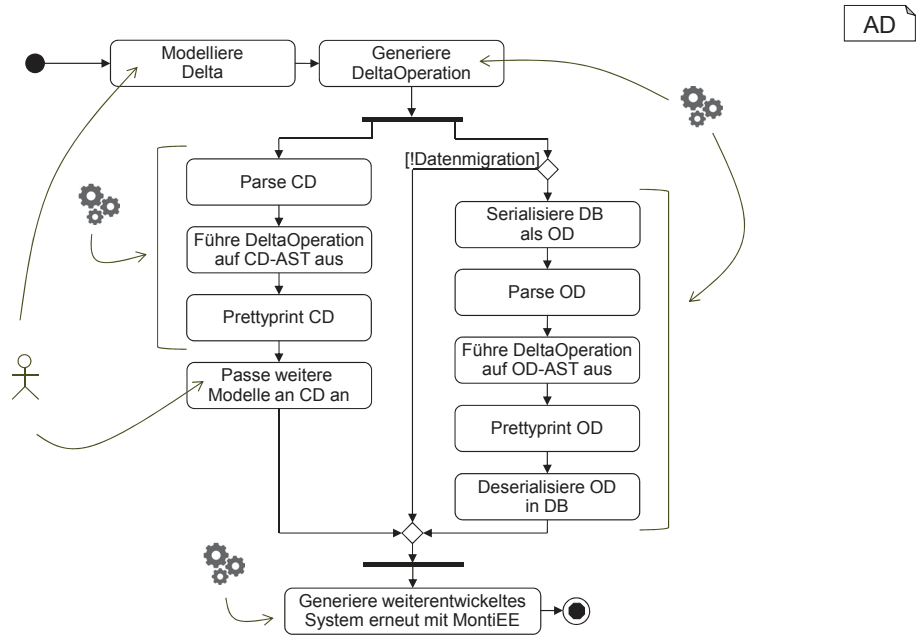


Abbildung 10.2: Darstellung der zu treffenden Entscheidungen und der auszuführenden Aktivitäten bei der Verwendung der Evolutions- und Migrationsfunktionalitäten von MontiEE. Manuelle und automatisch durchführbare Aktivitäten sind entsprechend markiert.

Methodik der Generierung der Evolutionsinfrastruktur mit MontiEE

Neben der Möglichkeit zur Generierung einer Enterprise Applikation und der zugehörigen Methodik der Verwendung, die in Abbildung 10.1 gezeigt wurde, bietet MontiEE ebenfalls die Möglichkeit zur Weiterentwicklung generierter Systeme, die sich bereits im Betrieb befinden und bei denen Produktionsdaten in der Datenbank persistent gespeichert sind. Dazu werden Deltas, die in Abschnitt 6 vorgestellt wurden, eingesetzt. Deltas als Modelle der Deltasprachen modellieren die Weiterentwicklung des Systems, indem sie die Weiterentwicklung des zugrunde liegenden Domänenmodells abbilden. Abbildung 10.2 zeigt die Methodik der Weiterentwicklung von Systemen, die mit MontiEE generiert wurden. Zunächst muss die Weiterentwicklung, also die Veränderung des Klassendiagramms, in Form eines Deltas modelliert werden. Auf Basis dieses Deltas wird, wie in Kapitel 9 dargestellt, die Evolutionsinfrastruktur generiert. Diese generiert für jedes Delta konkrete Operationen, die sich Bibliotheksfunktionen des Evolutionsframeworks zu Nutze machen. Diese generierten Operationen erlauben die Evolution eines Klassendiagramms und die optionale Migration persistenter Daten. Für die Evolution des Klassendiagramms wird zunächst das Klassendiagramm geparst, mit Hilfe der generierten Operation, wie in Abschnitt 9.2 beschrieben, transformiert und anschließend wieder

geprettyprinted. Daran anschließend müssen die auf dem Domänenmodell basierenden Modelle und unter Umständen vorhandener handgeschriebener Quellcode angepasst werden.

Der Produktentwickler kann dann die Entscheidung treffen, ob die persistent gespeicherten Daten ebenfalls migriert werden sollen. Trifft er diese Entscheidung, werden die Daten der Datenbank, wie in Abschnitt 9.4, in ein Objektdiagramm serialisiert. Dieses Objektdiagramm wird als Eingabe für die generierte Deltaoperation verwendet und die enthaltenen Daten werden, wie in Abschnitt 9.3 dargestellt, migriert. Anschließend wird das migrierte Objektdiagramm, wie in Abschnitt 9.4 präsentiert, wiederum in die Datenbank deserialisiert. Die gleiche Vorgehensweise lässt sich auch für Objektdiagramme, die im Rahmen des modellbasierten Testens verwendet werden, anwenden. Dabei wird auf die Serialisierung und Deserialisierung verzichtet. Sind die Evolutions- und Migrationsschritte abgeschlossen, so kann das vollständige System neu generiert und deployt werden. Zur Vermeidung von Ausfallzeiten wird die Evolution und Migration typischerweise in einem Parallelbetrieb durchgeführt. Dazu wird das laufende System so lange unverändert weiterbetrieben, bis das weiterentwickelte System wieder deployt ist. Daran anschließend werden die Zugriffsadressen der Clients so umgeleitet, dass sie nicht mehr auf das ursprüngliche System, sondern auf das weiterentwickelte verweisen.

Im weiteren Verlauf dieses Kapitels wird in Abschnitt 10.2 auf die Konfiguration und Ausführung von MontiEE eingegangen. Dazu wird zunächst dargelegt, wie die einzelnen Generatoren konfiguriert werden können und wie das vordefinierte Standardverhalten von MontiEE abgeändert werden kann. Anschließend wird die Verwendung von MontiEE näher erläutert. Dazu wird gezeigt, wie die einzelnen Entscheidungen, die anhand des Aktivitätsdiagramms aus Abbildung 10.1 vorgestellt wurden, im Entwicklungsprozess getroffen werden können. Daran anschließend wird in Abschnitt 10.3 die Umsetzung des Szenarios beschrieben. Das Deployment einer Applikation wird in Abschnitt 10.4 präsentiert. Abschließend werden in Abschnitt 10.6 Fallstudien im Rahmen von Forschungsprojekten gezeigt. Innerhalb dieser Projekte wurde MontiEE in bestehende Systeme integriert oder bei Neuentwicklungen verwendet.

10.2 Konfiguration und Ausführung von MontiEE

Die einzelnen Generatoren von MontiEE verwenden jeweils Standardwerte bei der Generierung des Quellcodes. Diese werden vom Zielsystem benötigt, haben aber keine Entsprechung im Modell, so dass die Generatoren an dieser Stelle vorgegebene Werte oder Verhalten anwenden. Da diese Werte bei verschiedenen Enterprise Applikationen unterschiedlich sein können oder sogar, wenn sie auf dem gleichen Zielsystem ausgeführt werden, müssen, wurde eine Möglichkeit geschaffen, dies zu konfigurieren. Darüber hinaus lässt sich MontiEE, wie zuvor in Abbildung 10.1 gezeigt, modular verwenden. Deshalb ist die Möglichkeit geschaffen worden, einzelne Generatoren zu aktivieren oder zu deaktivieren.

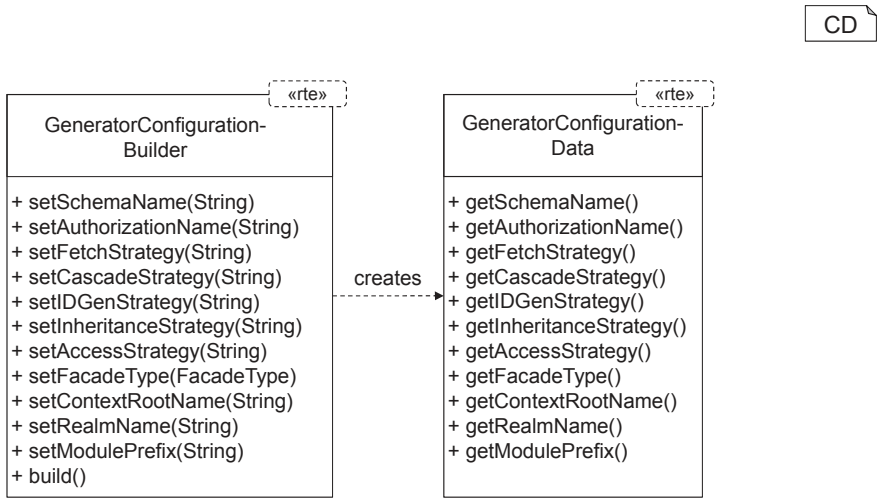


Abbildung 10.3: Darstellung der Klassen der Laufzeitumgebung zur Konfiguration der MontiEE-Generatoren.

10.2.1 Konfiguration

Zunächst wird die Konfiguration der Standardwerte, gefolgt von der Erläuterung der Verwendung einzelner Generatoren, erläutert. Zur internen Verwendung der Konfiguration der Standardwerte wurde eine allgemeine Datenstruktur, die allen Generatoren zur Verfügung steht, geschaffen (vgl. FA14-PE).

Abbildung 10.3 zeigt diese allgemeine Datenstruktur. Sie setzt das Builder-Pattern [GHJV95] ein. Die Aufgabe des Builders ist die Erstellung von Konfigurationsobjekten. Aus diesem Grund erlaubt seine API das Setzen unterschiedlicher Standardwerte, ohne diese zu erzwingen. Nach Ausführen der `build()` Methode wird ein Objekt der Klasse `GeneratorConfigurationData` erzeugt. Die Klasse `GeneratorConfiguration-Data` kapselt verschiedene Standardwerte, die vom Nutzer von MontiEE verändert werden können. Dabei lassen sich zwei Arten von Standardwerten unterscheiden. Zum einen existieren Werte, die von den Generatoren verwendet werden und nicht Teil der Modellierungssprachen von MontiEE sind. Zum anderen existieren Werte, die als Standardverhalten der Generatoren angenommen werden. Erstere sind im Wesentlichen Namen. Innerhalb von MontiEE wird technische Information mit Hilfe der Tagdefinitionen modelliert. Häufig wird aber technische Information zwingend benötigt, so dass sie für alle Modellelemente angegeben werden müsste. Damit innerhalb der Tagdefinition nicht alle Elemente getaggt werden müssen, kann mit Hilfe dieser Konfiguration ein Standardverhalten der Generatoren definiert werden. Dies bedeutet, dass die Generatoren immer die in der Konfiguration eingestellten Standardwerte verwenden, wenn in der Tagdefinition nichts anderes angegeben ist (vgl. FA20-PE). Während des Generierungsprozesses wird den Generatoren ein Konfigurationsobjekt mit Hilfe der MontiCore Mechanismen zur Verfügung gestellt. Die Integration in das Tooling des Entwicklungsprozesses wird in

Abschnitt 10.2.2 erläutert. Zunächst werden die zur Verfügung stehenden Standardwerte und -strategien näher erläutert.

Diese Werte sind im Einzelnen:

- `SchemaName`: Der SQL-Generator erzeugt ein Datenbankschema. Jedes Datenbankschema benötigt einen festgelegten Namen. Mit Hilfe dieser Konfigurationsmöglichkeit kann der Name des Schemas konfiguriert werden. Das Setzen des `SchemaName` erfolgt durch die `setSchemaName(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getSchemaName()` Methode auf den Wert der Konfiguration zugreifen. Wird der Schemaname nicht konfiguriert, wird "`montiee`" als Standardwert verwendet.

- `AuthorizationName`: Der SQL-Generator benötigt neben dem Namen des Schemas auch den Namen des auf die Datenbank zugreifenden Nutzers. Dieser Nutzer ist im JEE Technologiestack immer der Applikationsserver, so dass es stets genau einen Nutzer gibt. Das Setzen des `AuthorizationName` erfolgt durch die `setAuthorizationName(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getAuthorizationName()` Methode auf den Wert der Konfiguration zugreifen. Wird der Name nicht konfiguriert, wird "`montiee`" als Standardwert verwendet.

- `FetchStrategy`: Der Entity-Generator generiert Annotationen, die das Laden der Objekte und ihrer Links aus der Datenbank definieren. Dies lässt sich mit Hilfe der Tagdefinition für jede Assoziation spezifisch modellieren. Dennoch muss die Information für jede Assoziation einer Klasse, da die Annotation im Generat vorhanden sein muss, zur Verfügung stehen. Damit nicht für jede Assoziation eine entsprechende Information in der Tagdefinition modelliert werden muss, greifen die Generatoren auf einen Standardwert, der verwendet wird, wenn keine Information in der Tagdefinition vorhanden ist, zurück. Das Setzen dieses Standardverhaltens erfolgt durch die `setFetchStrategy(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getFetchStrategy()` Methode auf den Wert der Konfiguration zugreifen. Wird kein Standardverhalten definiert, wird "`eager`" als Standardwert verwendet. Alternativ kann "`lazy`" verwendet werden.

- `CascadeStrategy`: Wie bei der `FetchStrategy` benötigt der Entity-Generator ein Standardverhalten für den Fall, dass die Tagdefinition keine Kaskadierungsinformationen für eine Assoziation enthält. Das Setzen der Strategie erfolgt durch die `setCascadeStrategy(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getCascadeStrategy()` Methode auf den Wert der Konfiguration zugreifen. Wird die Strategie nicht konfiguriert, wird "`none`" als Standardwert verwendet. Als weitere Werte stehen "`all`", "`merge`", "`persist`", "`refresh`" und "`remove`" zur Verfügung.

- `IDGenStrategy`: Auch diese Möglichkeit zur Konfiguration dient dazu, das Verhalten der Generatoren, falls entsprechende Informationen in der Tagdefinition

nicht modelliert wurden, zu konfigurieren. Die `IDGenStrategy` konfiguriert die entsprechende Standardstrategie zur Erzeugung von IDs. Das Setzen der Strategie erfolgt durch die `setIDGenStrategy(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getIDGenStrategy()` Methode auf den Wert der Konfiguration zugreifen. Wird das Standardverhalten nicht konfiguriert, wird "`table`" als Standardwert verwendet. Als weitere Werte stehen "`sequence`", "`auto`", "`identity`" und "`none`" zur Verfügung.

- `InheritanceStrategy`: Zur Konfiguration des Standardverhaltens der Abbildung von Vererbung kann dieser Wert angepasst werden. Er wird immer dann verwendet, wenn in der Tagdefinition keine Angaben zur Abbildung von Vererbung gemacht wurden. Das Setzen der Strategie erfolgt durch die `setInheritancesStrategy(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getInheritancesStrategy()` Methode auf den Wert der Konfiguration zugreifen. Wird kein Standardverhalten konfiguriert, wird "`singleTable`" als Standardwert verwendet. Als weitere Werte stehen "`joined`" und "`tablePerClass`" zur Verfügung.

- `AccessStrategy`: Die `AccessStrategy` wird zur Konfiguration des ORM benötigt. Sie gibt an, ob der ORM direkt auf die Felder einer Klasse zugreift oder Getter- und Setter-Methoden verwendet. Der ORM verwendet die Felder oder die Methoden beim Schreiben oder Lesen von Objekten. Werden die zu speichernden Felder annotiert, verwendet der ORM diese direkt. Wird stattdessen eine Methode annotiert, muss diese einem vorgegebenem Namensschema für Getter- und Setter folgen. Die MontiEE-Generatoren reagieren auf diesen Standardwert, indem sie den Ort der Generierung der Annotationen entsprechend anpassen. Das Setzen des Werts erfolgt durch die `setAccessStrategy(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getAccessStrategy()` Methode auf den Wert der Konfiguration zugreifen. Wird die Zugriffsstrategie nicht konfiguriert, wird "`field`" als Standardwert verwendet. Alternativ kann "`method`" verwendet werden.

- `FacadeType`: Diese Möglichkeit zur Konfiguration wirkt sich auf den Facade-Generator aus. Abhängig von dieser Konfiguration wird eine Webservice-Fassade oder eine RPC-Fassade erzeugt. Das Setzen des Typs erfolgt durch die `setFacadeType(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getFacadeType()` Methode auf den Wert der Konfiguration zugreifen. Wird der Typ nicht konfiguriert, wird "`webservice`" als Standardwert verwendet. Alternativ kann "`rpc`" verwendet werden.

- `ContextRootName`: Für das Deployment der Applikation auf dem Server wird ein eindeutiger Name eines Kontextes benötigt. Dieser Name muss unter allen auf dem Server deployten Applikationen eindeutig sein. Das Setzen des `ContextRootName` erfolgt durch die `setContextRoot(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `get-`

`ContextRootName()` Methode auf den Wert der Konfiguration zugreifen. Wird der Name nicht konfiguriert, wird "`montiee`" als Standardwert verwendet.

- `RealmName:` Für das Deployment der Applikation auf dem Server wird der Name des zu verwendenden Realms benötigt. Ein Realm ist serverseitig konfiguriert und kapselt Sicherheitsmechanismen, Datenbankverbindungen und weitere Serverfunktionalität. Die Konfiguration des Realms muss manuell vom Administrator des Servers konfiguriert werden. Die erzeugte Applikation referenziert diesen Realm mit Hilfe dieses Standardwertes. Das Setzen des Wertes erfolgt durch die `setRealmName(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getRealmName()` Methode auf den Wert der Konfiguration zugreifen. Wird der Name nicht konfiguriert, wird "`montiee`" als Standardwert verwendet.

- `ModulePrefix:` Für das Deployment wird ferner ein Prefix des deployten Moduls benötigt. Das Setzen des Wertes erfolgt durch die `setModulePrefix(String)` Methode des `GeneratorConfigurationBuilder`. Die Generatoren können mit Hilfe der `getModulePrefix()` Methode auf den Wert der Konfiguration zugreifen. Wird das Präfix nicht konfiguriert, wird "`montiee`" als Standardwert verwendet.

Die bisher beschriebenen Standardwerte werden zur Laufzeit der Generatoren verwendet, Entscheidungen, wie Standardverhalten oder die Vergabe von Namen zu konfigurieren. Mit Hilfe dieser Strukturen kann MontiEE konfiguriert werden.

10.2.2 Ausführung

Im Weiteren werden die Möglichkeiten zur Ausführung von MontiEE beschrieben. MontiEE lässt sich mit Hilfe eines Groovy-Skripts [KKL+15] über die Kommandozeile oder mit Hilfe eines Maven-Plugins [ABC+14] in den Entwicklungsprozess integrieren. Dazu agieren sowohl das Groovy-Skript als auch das Maven-Plugin nur als Frontend zur Ausführung von MontiEE. Beide verwenden intern die gleiche Datenstruktur.

Diese Datenstruktur ist in Abbildung 10.4 dargestellt. Auch hier wird, wie bereits bei der Konfiguration von MontiEE, das Builder Pattern eingesetzt. Die Klasse `MontiEERunnerBuilder` wird als Builder für einen `MontiEERunner` verwendet. Der `MontiEERunner` wird von dem Builder konfiguriert und anschließend erzeugt. Die API des Builders bietet einen Konstruktor, der drei String Parameter enthält, an. Diese Parameter sind der Pfad der Eingabemodelle, der Zielpfad des Generats und der Pfad zu den Tagdefinitionen. Darüber hinaus bietet der Builder einzelne Methoden zur Aktivierung oder Deaktivierung einzelner Generatoren an. Die einzelnen Generatoren, DAO-, SQL-, DTO- und Facade-Generator, lassen sich jeweils einzeln, wie am Aktivitätsdiagramm aus Abbildung 10.1 gezeigt, verwenden. Der BusinessAPI-Generator wird, wie ebenfalls im Aktivitätsdiagramm gezeigt, immer dann verwendet, wenn der Facade-Generator verwendet wird. Die Methode `customizeData(GeneratorConfigurationData)` erwartet als Parameter eine Instanz der zuvor in Abbildung 10.3 vorgestellten Konfigurati-

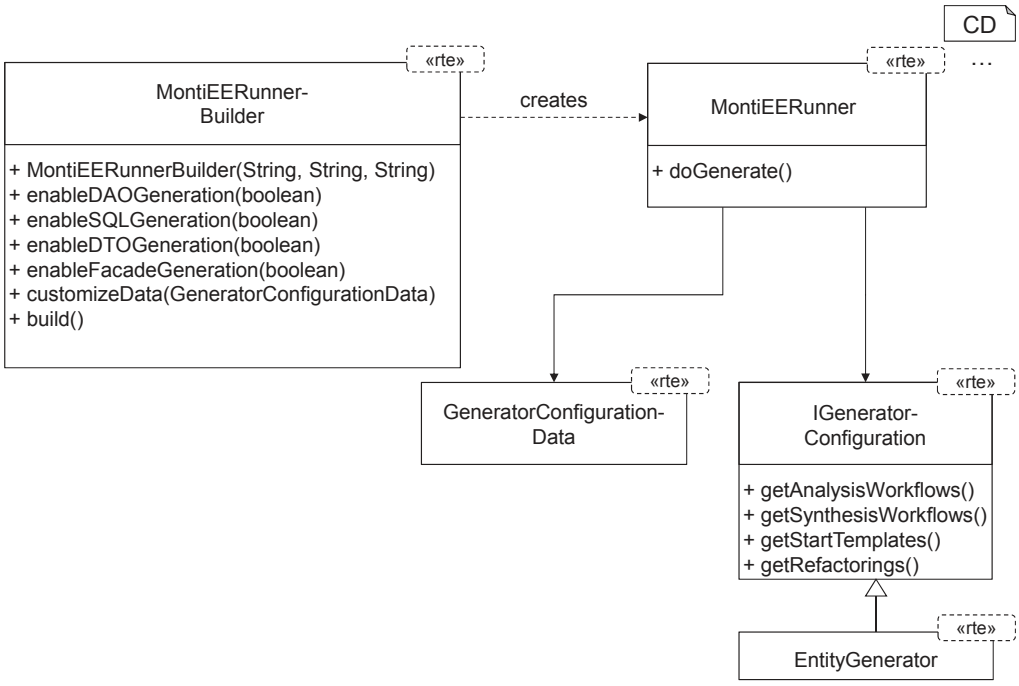


Abbildung 10.4: Darstellung der Klassen der Laufzeitumgebung zur Ausführung von MontiEE.

onsobjekte für MontiEE. Die `build()` Methode erzeugt den eigentlichen `MontiEERunner`. Der `MontiEERunner` enthält als einzige API Methode die `doGenerate()` Methode. Darüber hinaus besitzt er eine Assoziation zu der entsprechenden MontiEE-Konfiguration und eine Assoziation zu einer Menge von `IGeneratorConfiguration`. Für jeden MontiEE-Generator existiert eine Subklasse dieser Konfiguration als interne Konfigurationsrepräsentanz des Generators. Diese interne Konfiguration wird benötigt, um die von einem Generator ausgeführten Workflows in der Analyse- und Synthesephase, die ausgeführten Refactorings und das ausgeführte Starttemplate zu bestimmen. Der `MontiEERunner` aggregiert alle Workflows, Refactorings und Starttemplates der MontiEE-Generatoren und erzeugt daraus eine gültige MontiCore Parametrierung. Mit Hilfe der `doGenerate()` Methode wird daran anschließend MontiCore mit Hilfe der Parametrierung aufgerufen. Auf eine Erläuterung der Details dieser Struktur wird im Rahmen dieser Arbeit verzichtet. Sie wurde geschaffen, um auf einfache Art und Weise Generatoren in die bestehende MontiCore Infrastruktur zu integrieren, ohne MontiCore mehrfach oder ineffizient zu verwenden. Für den Nutzer von MontiEE ist der `MontiEERunner-Builder` und der in Abbildung 10.3 gezeigte `GeneratorConfigurationBuilder` der Einstiegspunkt. Sollen weitere Generatoren in die MontiEE Familie integriert werden, müssen diese ebenfalls eine Subklasse der `IGeneratorConfiguration` bilden und die Methoden entsprechend implementieren.

Das Groovy-Skript, das Maven-Plugin oder die Kommandozeile verwenden alle diese Datenstruktur und die jeweiligen Builder zur Erzeugung des `MontiEERunner`. Für die Kommandozeile wurde eine Methode geschaffen, die ein Groovy-Skript als Parameter erhält und auf Basis des Skripts die `doGenerate()` Methode des Runners ausführt. Diese Methode kann auch direkt aus dem Groovy-Skript ausgeführt werden. Im Weiteren wird das Groovy-Skript und das Maven-Plugin vorgestellt.

Das Groovy-Skript

Zur Ausführung von MontiEE kann ein Groovy-Skript verwendet werden. Aufgabe des Groovy-Skripts ist die Erstellung des `MontiEERunner`. Dazu muss der `MontiEERunnerBuilder` instanziiert werden. Der Konstruktor der Klasse benötigt den Pfad zu den Eingabemodellen, den Ausgabepfad und den Pfad zu den Tagdefinitionen.

```
1 inputPath = "de/montiee/socnet"                                    Groovy
2 outputPath = "target/de/montiee/socnet"                             ...
3 tagDefinition = "de/montiee/socnet/SocNetTags.tags"
```

Listing 10.5: Definition der benötigten Parameter zur Ausführung von MontiEE im Groovy-Skript.

Listing 10.5 zeigt die Definition der Parameter als Teil des Groovy-Skripts. Der Parameter `inputPath` beinhaltet den Pfad zu den Modellen des sozialen Netzwerks. Diese Modelle beinhalten das Domänenmodell, die Sichten, die Rollen- und Rechtediagramme sowie das Mappingdiagramm. Der Parameter `outputPath` definiert das Ausgabeverzeichnis für das Generat und der Parameter `tagDefinition` den Pfad zu einer Tagdefinition.

```
1                                                                    Groovy
2 builder = new MontiEERunnerBuilder(inputPath, outputPath,           ...
3                                    tagDefinition)
4
5 configurationData = new GeneratorConfigurationBuilder()
6                         .setSchemaName("SocNet")
7                         .build()
8
9 runner = builder.enableDAOGeneration(true)
10                .customizeData(configurationData)
11                .build()
```

Listing 10.6: Konfiguration und Auswahl der Generatoren zur Ausführung von MontiEE im Groovy-Skript.

Listing 10.6 zeigt die Verwendung der zuvor in Listing 10.5 definierten Parameter zur Instanziierung des `MontiEERunnerBuilder`. Darüber hinaus wird mit Hilfe des `GeneratorConfigurationBuilder` die Konfiguration von MontiEE dahingehend an-

gepasst, dass der Name des Datenbankschemas auf ”SocNet” abgeändert wird. Mit Hilfe des `enableDAOGeneration(true)` Aufrufs wird der DAO-Generator aktiviert. Die `customizeData(configurationData)` Methode übergibt das zuvor erstellte Konfigurationsobjekt dem `MontiEERunnerBuilder`. Mit Hilfe der `build()` Methode wird der `MontiEERunner` erzeugt, mit dessen `doGenerate()` Methode die Generierung gestartet werden kann.

Mit Hilfe dieses Skripts lässt sich MontiEE von der Kommandozeile oder direkt aus Groovy heraus verwenden. Als Alternative kann MontiEE mit Hilfe eines Maven-Plugins, welches nachfolgend vorgestellt wird, konfiguriert werden.

Das Maven-Plugin

Neben der Konfiguration mit Hilfe eines Groovy-Skripts kann MontiEE auch mit Hilfe eines Maven-Plugins in den Entwicklungsprozess integriert werden.

```
1 <plugin>                                                          XML
2   <groupId>de.montiee.utilities.maven</groupId>                    ...
3   <artifactId>maven.plugin</artifactId>
4   <!--  weitere Elemente -->
5 </plugin>
```

Listing 10.7: Koordinaten des Maven-Plugins.

Listing 10.7 zeigt die Verwendung des Plugins innerhalb des `<plugin>...</plugin>` Blocks eines Project Object Model (POM). Mit Hilfe dieser Koordinaten kann das MontiEE-Plugin in jedes Maven-Projekt integriert werden. Zur Konfiguration des Plugins und zur Auswahl der Generatoren wird der `<configuration>...</configuration>` Block innerhalb des `<plugin>...</plugin>` Blocks verwendet.

Listing 10.8 zeigt die Angabe der benötigten Parameter. Die Pfade zu den Eingabemodellen, der Ausgabepfad und der Pfad zu den Tagdefinitionen sind angegeben. Darüber hinaus wird der DAO-Generator aktiviert und analog zum Groovy-Skript der Schemaname konfiguriert. Intern verwendet das Maven-Plugin diese Parameter, um die jeweiligen Builder analog zum Groovy-Skript zu instanziieren und den MontiEE-Runner zu starten. Dies wird über das Goal `generate` in der `generate-sources` Phase ausgeführt. Eine Einführung in die Entwicklung von Maven-Plugins, Goals und die jeweiligen Phasen ist in [ABC+14] gegeben.

Nachdem die Konfiguration und die Ausführung von MontiEE vorgestellt wurde, wird im nachfolgenden Abschnitt auf die Erzeugung eines EARs, welches auf dem Applikationsserver deployt wird, eingegangen. Dabei werden die einzelnen Teile des Generats in unterschiedlichen Artefakten gebündelt und XML-Dateien zur Konfiguration des Servers und der Applikation erläutert.

```xml
<configuration>
  <inputFiles>
    <inputFile>de/montiee/socnet</inputFile>
  </inputFiles>
  <outputDirectory>target/de/montiee/socnet</outputDirectory>
  <tagDefinition>
    de/montiee/socnet/SocNetTags.tags
  </tagDefinition>
  <generateDAO>true</generateDAO>
  <schemaName>SocNet</schemaName>
  <!--  weitere Elemente -->
</configuration>
```

Listing 10.8: Definition der benötigten Parameter sowie Konfiguration und Auswahl der Generatoren zur Ausführung von MontiEE bei Verwendung des Maven-Plugins.

10.3 Anwendung der MontiEE-basierten Methodik auf das Szenario

Nachdem die Verwendung, die Konfiguration und die Ausführung von MontiEE zuvor beschrieben wurden, wird in diesem Abschnitt die Umsetzung des Szenarios unter Beteiligung aller Generatoren vorgestellt. Dazu wird zunächst die Modellierung des Domänenmodells und die Erstellung einer Tagdefinition vorgestellt. Daran anschließend wird die Generierung der Entitäten, des SQL-Skripts und der DAOs gezeigt. Im Anschluss daran wird die Generierung der DTOs und der Fassaden erläutert.

Modellierung des Domänenmodells

Zur Modellierung des Domänenmodells wird sich auf das in Abbildung 10.9 wiederholte Klassendiagramm des sozialen Netzwerks aus Abbildung 4.2 beschränkt. Abbildung 10.9 zeigt das Domänenmodell des sozialen Netzwerks. Es beinhaltet die zur Umsetzung des Szenarios geforderten Klassen, Attribute und Assoziationen. Die Semantik des Domänenmodells wurde bereits in Abschnitt 4.2 detailliert erläutert. Eine vollständige textuelle Fassung, wie sie von MontiEE verwendet wird, ist in Anhang C.1 gegeben.

Modellierung der Tagdefinition

Die Modellierung der Tagdefinition `SocNetTags` ist in Listing 10.10 gezeigt. Teile dieser Tagdefinition wurden zur Erklärung der Tagdefinitionssprache bereits in den Listings 4.4 und 4.5 gezeigt. Es beginnt mit dem Namen der Tagdefinition und der Referenz auf das Domänenmodell `SocNet`. Danach werden die einzelnen Elemente des Domänenmodells getaggt. Alle beteiligten Klassen werden mit dem `Entity` Tag getaggt. Zudem wird die Klasse `Profile` mit dem `Inheritance` Tag und dem `IDGen` Tag getaggt. Dies

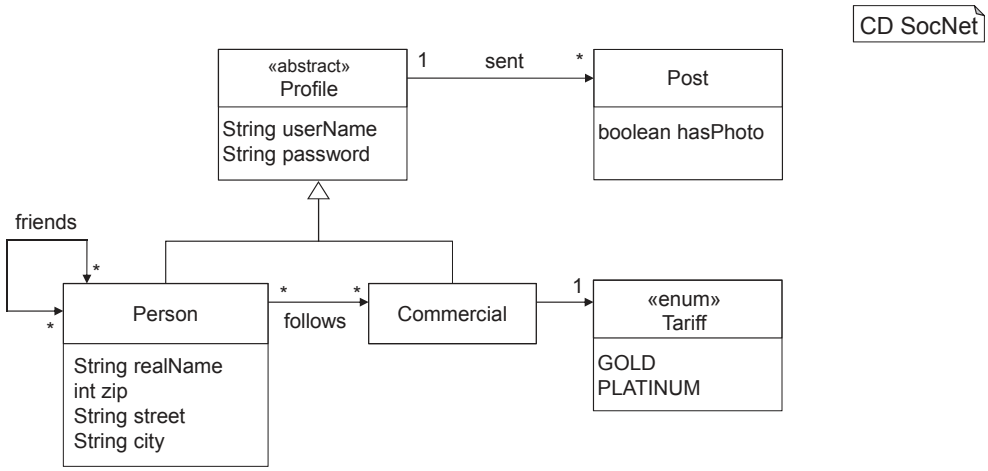


Abbildung 10.9: Wiederholung des Domänenmodells des sozialen Netzwerks aus Abbildung 4.2.

bewirkt, wie in Abschnitt 7.2 erläutert, dass es innerhalb des Datenbankschemas eine Tabelle pro Subklasse der Klasse `Profil` gibt und die benötigten Datenbank-IDs mit Hilfe einer speziellen Tabelle, die den nächsten fortlaufenden Wert der Primärschlüssel aller Typen enthält, gebildet werden. Die Assoziation `follows` wird mit dem `Cascading` Tag und der Option `none` getaggt. Dies bedeutet, dass bei Ausführung einer CRUD-Operation auf der Klasse `Person` die Operation nicht kaskadiert und nicht auf den über die `follows` Assoziation verbundenen Elementen ausgeführt wird. Demgegenüber wird die `sent` Assoziation mit der Option `remove` getaggt. Dies bedeutet, dass, immer wenn ein Element des Typs `Profil` gelöscht wird, auch alle assoziierten Elemente des Typs `Post` gelöscht werden. Zudem wird diese Assoziation mit dem `Fetch` Tag und der Option `lazy` getaggt. Dadurch wird die Ladestrategie, wie in Abschnitt 7.2 beschrieben, bestimmt. Da ein Profil eine große Menge Posts erstellt haben kann und diese nicht alle geladen werden müssen, kann so die Performanz erhöht werden. Abschließend wird die Klasse `Person` mit dem `Queries` Tag getaggt, welcher bewirkt, dass den Clients Queries innerhalb der Fassaden zur Verfügung gestellt werden. Das hier gezeigte Query lädt alle Elemente, deren `realName` Attribut den Wert `friendsRealName` besitzt und die über die `friends` Assoziation mit einem Objekt verbunden sind, dessen `userName` Attribut den Wert `myUserName` besitzt, aus der Datenbank.

10.3.1 Konfiguration von MontiEE

Die in Listing 10.10 dargestellte Tagdefinition modelliert stets Abweichungen von einem konfigurierbaren Standardverhalten. Die Konfiguration des Standardverhaltens wurde in Abschnitt 10.2 erläutert. Typischerweise wird zum Beispiel angenommen, dass alle Operationen kaskadieren und Objektgraphen stets vollständig aus der Datenbank geladen werden. Soll dies, zum Beispiel mit der `lazy` Option, eingeschränkt werden, muss

```
tags SocNetTags for SocNet {                                    TD
                                                                ...
  tag Profile, Person, Commercial, Post, Tariff
    with Entity;

  tag Profile
    with Inheritance="tablePerClass", IDGen="table";

  tag follows
    with Cascading {
      cascade = "none";
    };

  tag sent
    with Fetch="lazy", Cascading {
      cascade = "remove";
    };

  tag Person
    with Queries {
      query {
        name = "getFriendByRealName",
        value = "SELECT Object(x)
                 FROM Person p, me
                 WHERE p.realName = :friendsRealName
                 AND me.userName = :myUserName
                 AND p IN ELEMENTS(me.friends)"
      }
    };

}
```

Listing 10.10: Tagdefinition zur Anreicherung des Domänenmodells des sozialen Netzwerks mit zusätzlichen Informationen.

dies modelliert werden. Die weiteren Optionen aller Tags und eine detaillierte Erklärung der Semantik der Optionen wird in Abschnitt 7.2 gegeben. Die Konfiguration und die Ausführung von MontiEE kann, wie in Abschnitt 10.2 beschrieben, mit Hilfe eines Groovy-Skripts oder eines Maven-Plugins erfolgen.

10.3.2 Generierung der Entitäten

Das Beispiel des sozialen Netzwerks in Form des Domänenmodells aus Abbildung 10.9 und der Tagdefinition aus Listing 10.10 aufgreifend wird in Abbildung 10.11 das Generat des Entity-Generators gezeigt.

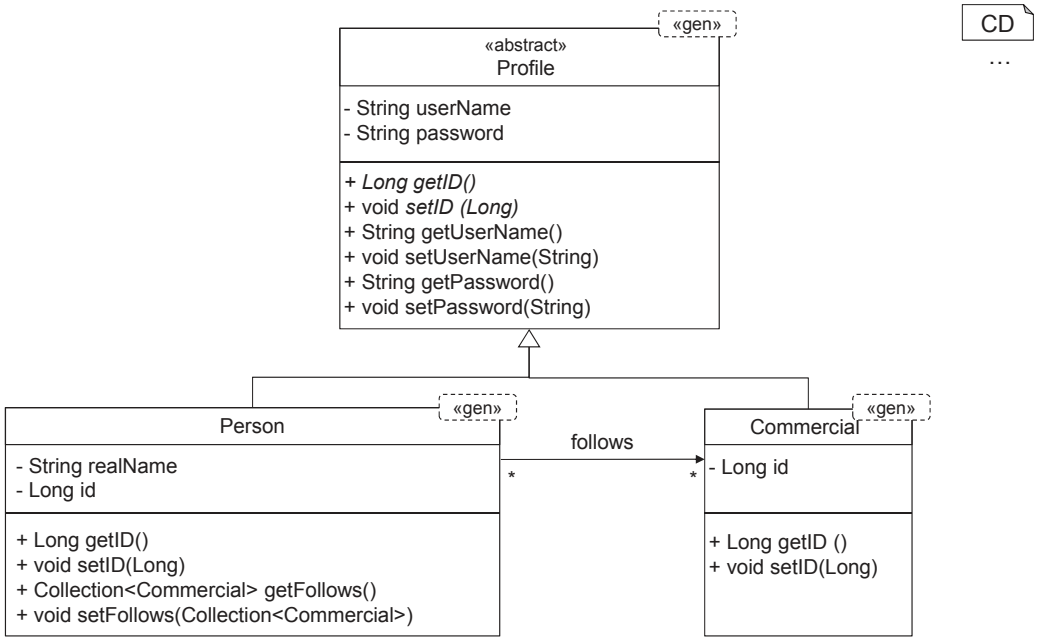


Abbildung 10.11: Auszug der durch den Entity-Generator generierten Java-Klassen. Diese basieren auf dem Domänenmodell aus Abbildung 10.9 und sind um verschiedene Methoden und Attribute erweitert.

```
1 @MappedSuperclass                                                    Java
2 class Profile { /* ... */ }                                          ...
3
4 @Entity
5 @Table(name="Persons", schema="socnet")
6 @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
7 class Person extends Profile{ /* ... */ }
```

Listing 10.12: Darstellung der generierten Annotation der Entitäten des Szenarios.

Abbildung 10.11 zeigt die generierten Klassen `Profile`, `Person` und `Commercial`. Zusätzlich werden die im Klassendiagramm modellierten Attribute und Methoden generiert. In Abbildung 10.11 sind die Attribute `userName`, `password` und `realName` sowie die Methoden `getUserName()` und `setUserName(String userName)` dargestellt. Darüber hinaus sind die abstrakten Methoden `getID()` und `setID(Long ID)` der Klasse `Profile` und ihre Implementierung in den Subklassen dargestellt. Dies ist das Resultat der Option `tablePerClass` des `Inheritance` Tags.

Listing 10.10 taggt die Klassen `Profile`, `Person`, `Commercial`, `Post` und `Tariff` mit dem `Entity` Tag. In Listing 10.12 ist die daraus resultierende Annotation der Klasse `Profile` sowie der generierte Code der Klasse `Person` dargestellt. Die Annotationen,

```java
@Id
@GeneratedValue(strategy=GenerationType.TABLE,
                generator="personGen")
@TableGenerator(name="personGen", table="IdValues",
                pkColumnName="idName",
                valueColumnName="idValue",
                pkColumnValue="personId",
                allocationSize=1)
@Column(name="id")
protected Long id;
```

Listing 10.13: Darstellung der Annotationen zur Konfiguration und Erstellung der Datenbank-ID der Klasse `Person`.

wie die Annotation `@MappedSuperClass`, resultieren aus dem `Entity` Tag. Darüber hinaus ist die Klasse `Profile` in Listing 10.10 mit dem `Inheritance` Tag getaggt. Dieser resultiert in der Annotation `@Table` der Klasse `Person`, die die Vererbung, wie in in Abschnitt 7.4 beschrieben, abbildet.

Weiterhin benötigt jede Entität eine eindeutige ID. Listing 10.13 zeigt den für die ID-Vergabe notwendigen Auszug des generierten Quellcodes. Die Klasse `Profile` wurde in Listing 10.10 mit dem `IDGen` Tag getaggt. Dieser Tag wird dann für alle Subklassen, also auch für die hier dargestellte Klasse `Person`, verwendet. Dazu wird die Annotation `@GeneratedValue` verwendet. Sie beschreibt, dass der Wert des Attributs mit Hilfe der in Abschnitt 7.4.1 beschriebenen Strategie von einem Generator mit Namen `personGen` generiert ist. In Listing 10.10 wurde zudem die `table` Option gewählt. Diese bewirkt, dass die Annotation `@TableGenerator` verwendet wird. Zusätzlich wird eine Tabelle angegeben, in der der entsprechende ID-Wert gespeichert wird.

In Listing 10.10 werden die Assoziationen `follows` und `sent` mit dem `Cascading` und dem `Fetch` Tag getaggt. Die `follows` Assoziation ist dabei eine unidirektionale `*-zu-*` Assoziation, die `sent` Assoziation eine unidirektionale `1-zu-*` Assoziation. Listing 10.14 zeigt die Umsetzung beider Assoziationen mit den entsprechenden Annotationen. Zunächst wird die `follows` Assoziation zwischen der Klasse `Person` und der Klasse `Commercial` erklärt. Die Assoziation selbst wird als `Set` modelliert, da eine Person vielen kommerziellen Profilen folgen kann. Daher wird das Attribut `follows` mit der Annotation `@ManyToMany` annotiert. Der Name des Attributs leitet sich aus dem Namen der Assoziation ab. Innerhalb der Annotation `@ManyToMany` wird das Attribut `cascade` verwendet, welches in diesem konkreten Fall durch den gewählten Kaskadierungstyp angibt, dass keine Kaskadierung erfolgt. Dies resultiert, wie in Listing 10.10 dargestellt, aus der Verwendung des `Cascading` Tags und seinen Optionen.

Darüber hinaus ist in Listing 10.14 die `sent` Assoziation dargestellt. Sie unterscheidet sich von der `follows` Assoziation in verschiedenen Details. Sie besitzt andere Kardinalitäten, wurde mit dem `Fetch` Tag getaggt und modelliert eine Assoziation zwischen

```java
@ManyToMany(cascade=CascadeType.NONE)
@JoinTable(name="PersonsCommercials", schema="socnet",
    joinColumns=
        @JoinColumn(name="person", referencedColumnName="id"),
    inverseJoinColumns=
        @JoinColumn(name="commercial", referencedColumnName="id"))
protected Set<Commercial> follows = new HashSet<Commercial>();

@OneToMany(cascade=CascadeType.REMOVE, fetch=FetchType.LAZY)
@JoinTable(name="PersonPosts", schema="socnet",
    joinColumns=
        @JoinColumn(name="person", referencedColumnName="id"),
    inverseJoinColumns=
        @JoinColumn(name="post", referencedColumnName="id"))
protected Set<Post> sent = new HashSet<Post>();
```

Listing 10.14: Darstellung der Abbildung der beiden modellierten Assoziationen `follows` und `sent` unter Berücksichtigung der Tagdefinition aus Listing 10.10.

```java
@NamedQueries({
  @NamedQuery(name=getFriendByRealName,
    query = "SELECT Object(x) FROM Person p, me
             WHERE p.realName = :friendsRealName
             AND me.userNaem = :myUserName
             AND p IN ELEMENTS(me.friends)")
})
class Person extends Profile{ /* ... */ }
```

Listing 10.15: Darstellung der `@NamedQueries` Annotation am Beispiel der Klasse `Person`.

einer `MappedSuperclass` und einer Entität. Die `1-zu-*` Kardinalität schlägt sich in der Verwendung der Annotation `@OneToMany` nieder. Die Ausprägung der Kaskadierung und des Fetchtyps innerhalb der Annotation wird durch die gewählten Optionen in Listing 10.10 bestimmt. Darüber hinaus wurde eingangs angemerkt, dass es sich bei der Klasse `Profile` um eine Superklasse, deren Attribute in die Subklassen abgebildet werden, so dass die Klasse `Profile` keine eigenständige Entität ist und keine eigene Datenbanktabelle besitzt, handelt. Dies ist an dieser Stelle auch kein Hindernis, da die Jointabelle davon unberührt ist. Lediglich die Spaltenreferenz ist hier problematisch, da sie sich sowohl auf Spalten der `Person` Tabelle oder der `Commercial` Tabelle beziehen könnte. Dadurch ist aber der Primärschlüssel der Jointabelle nicht mehr eindeutig. Aus diesem Grund wird das Attribut, welches die Assoziation darstellt, nicht in die `Profile` Klasse, sondern in die jeweiligen Subtypen einzeln generiert. Dies hat zur Folge, dass im Generat sowohl eine `sent` Assoziation zwischen `Person` und `Post` als auch zwischen

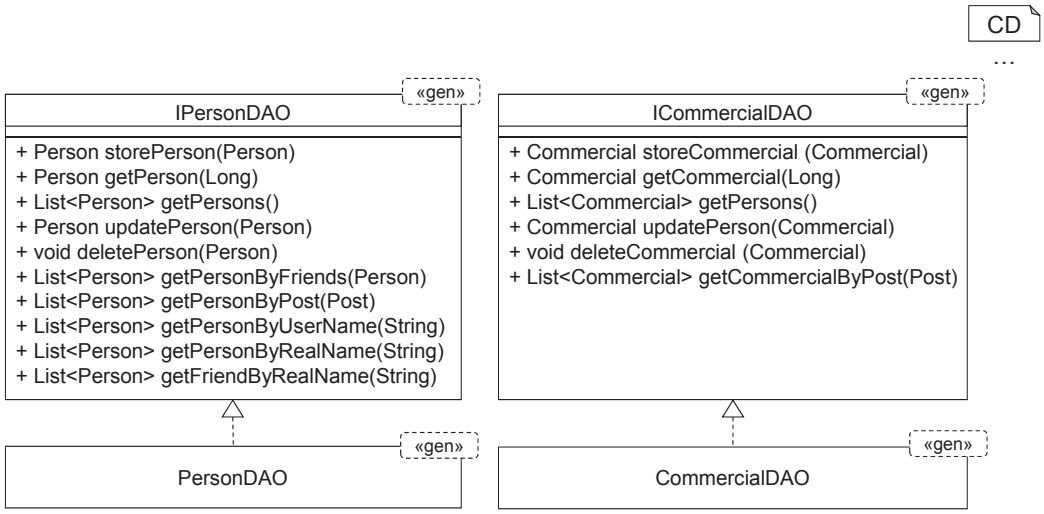


Abbildung 10.16: Auszug der durch den DAO-Generator generierten Java-Klassen. Diese basieren auf dem Domänenmodell aus Abbildung 10.9 und sind um verschiedene Methoden und Attribute erweitert.

`Commercial` und `Post` existiert. Die Getter- und Setter-Methoden sind in der `Profile` Klasse wiederum als abstrakte Methoden, so dass der Zugriff auf das Attribut auch auf dem abstrakten Typ sichergestellt werden kann, dargestellt. Listing 10.15 zeigt die Auswirkungen des `Queries` Tags aus Listing 10.10, welcher die Klasse `Person` taggt. Er erweitert die Klasse `Person` um ein später zur Laufzeit ausführbares Query.

10.3.3 Generierung der DAOs

Abbildung 10.16 zeigt das Generat des DAO-Generators für das soziale Netzwerk aus Abbildung 10.9. Gezeigt sind die beiden Interfaces `IPersonDAO` und `ICommercialDAO` sowie die implementierenden Klassen `PersonDAO` und `CommercialDAO`. Dargestellt sind die CRUD-Funktionalität sowie die Methoden zur Ausführung von Queries. Die allgemeine Abbildung der Modelle in das Generat des DAO-Generators wurde in Abschnitt 7.5 gezeigt. Durch die angebotene CRUD-Funktionalität können Objekte gelesen oder geschrieben werden. Zudem können automatisiert generierte oder benutzerdefinierte Queries verwendet werden. Das benutzerdefinierte Query, das durch die Methode `getFriendByRealName(Person,String)` verwendbar ist, ist in der Tagdefinition in Listing 10.10 definiert.

Das Interface `ICommercialDAO` und dessen implementierende Klasse `CommercialDAO` sind analog generiert. Darüber hinaus wird mit Hilfe des BusinessAPI-Generators eine Schnittstelle von der Geschäftslogik zur Persistenz generiert, wenn DAOs verwendet werden. Abbildung 10.17 zeigt eben diese generierte Schnittstelle von der Geschäftslogik zur Persistenz. Das Interface `IApplicationDAO` und die implementierende Klasse

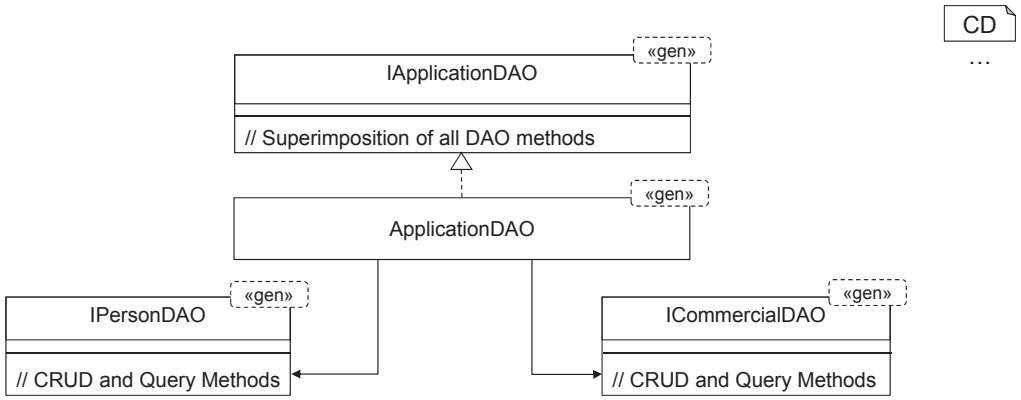


Abbildung 10.17: Darstellung der durch den BusinessAPI-Generator generierten API als Schnittstelle zwischen Geschäftslogik und DAOs. Diese basiert auf dem Domänenmodell aus Abbildung 10.9.

`ApplicationDAO` wurden generiert. Das Interface beinhaltet alle Methoden der beiden vom DAO-Generator erzeugten Interfaces `IPersonDAO` und `ICommercialDAO`. Ebenfalls vom DAO-Generator erzeugt werden die implementierenden Klassen dieser beiden Interfaces. Die vom BusinessAPI-Generator generierte Klasse `ApplicationDAO` implementiert das generierte Interface und delegiert an die entsprechenden DAOs.

10.3.4 Generierung der SQL-Skripte

Das Generat des SQL-Generators basiert auf dem Domänenmodell aus Abbildung 10.9 und der Tagdefinition aus Listing 10.10 als Eingabe. Die allgemeine Abbildung der Modelle in das Generat des SQL-Generators wurde in Abschnitt 7.5 gezeigt.

Listing 10.18 zeigt den Teil des SQL-Skripts, der die Erstellung der Tabellen für die Klassen `Person` und `Commercial` bewirkt. Die Tabellen entsprechen der `tablePerClass` Option des `Inheritance` Tags und enthalten ein `id` Attribut. Listing 10.18 zeigt zudem die Erstellung der für die `follows` Assoziation benötigten Jointabelle. Sie besteht aus zwei Spalten: `persons` und `commercials`. Diese Spalten speichern die ID Werte der paarweise assoziierten Objekte. Dadurch können private Profile kommerziellen Profilen zugeordnet werden. Listing 10.18 zeigt den Teil des generierten Skripts, der für das Hinzufügen der Primär- und Fremdschlüssel zuständig ist. Es werden, wie in Abschnitt 7.6 beschrieben, die Primärschlüssel der Tabellen `Persons` und `Commercials` und entsprechende Fremdschlüssel festgelegt.

10.3.5 Modellierung der Sichten

Die Modellierung von Sichten für Clients des sozialen Netzwerks wurde in Abschnitt 5.2 bereits vorgestellt. Dabei wurde in Listing 5.5 eine Sicht auf die Klasse `Profile` model-

```
SQL
...
```

```
CREATE TABLE Persons (
  id BIGINT,
  userName TEXT,
  passwordName TEXT,
  realName TEXT
);

CREATE TABLE Commercials (
  id BIGINT,
  userName TEXT,
  passwordName TEXT
);

CREATE TABLE PersonsCommercials (
  persons BIGINT,
  commercials BIGINT
);

ALTER TABLE Persons
  ADD PRIMARY KEY (id);
ALTER TABLE Commercials
  ADD PRIMARY KEY (id);
ALTER TABLE PersonsCommercials
  ADD PRIMARY KEY (persons,commercials);
ALTER TABLE PersonsCommercials
  ADD FOREIGN KEY (persons) REFERENCES Persons (id);
ALTER TABLE PersonsCommercials
  ADD FOREIGN KEY (commercials) REFERENCES Commercials (id);
```

Listing 10.18: Darstellung der generierten `CREATE TABLE` und der `ALTER` Elemente des generierten SQL-Skripts am Beispiel der Klassen `Person` und `Commercial` sowie ihrer Jointabelle.

liert, die lediglich das Attribut `userName` enthält. In Listing 5.4 wurde modelliert, dass die Klasse `Person` vollständig enthalten ist. Zusätzlich wurden die `sent` Assoziation sowie die Assoziation zwischen der Klasse `Commercial` und der Enumeration `Tariff` als flache Assoziationen in Listing 5.6 modelliert. An dieser Stelle wird die Modellierung nicht nochmals erläutert.

10.3.6 Generierung der DTOs

Das aus der Modellierung der Sichten resultierende Generat wird exemplarisch am Beispiel aus Abbildung 10.9 erläutert. Abbildung 10.19 zeigt die erzeugten DTOs basierend auf der in Abschnitt 5.2 modellierten Sicht. Die allgemeine Abbildung der Modelle in das Generat des DTO-Generators wurde in Abschnitt 8.2 gezeigt.

Der DTO-Generator erzeugt für das Beispiel des sozialen Netzwerks drei DTOs: die

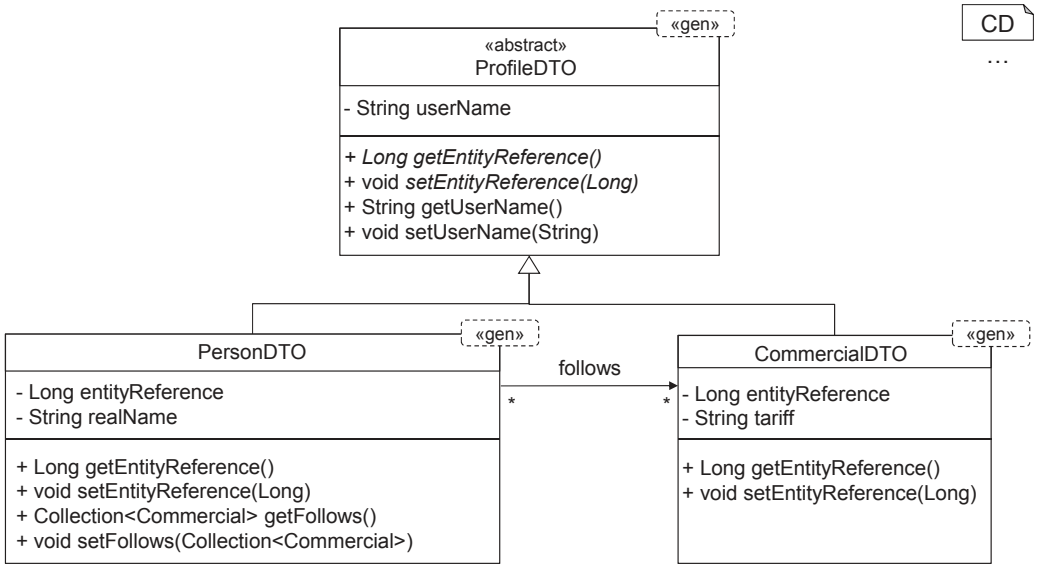


Abbildung 10.19: Auszug der durch den DTO-Generator generierten Java-Klassen. Diese basieren auf dem Domänenmodell aus 10.9 und sind um verschiedene Methoden und Attribute erweitert.

abstrakte Klasse `ProfileDTO` und die konkreten Klassen `PersonDTO` und `CommercialDTO`. Basierend auf der Sicht wird in der generierten Klasse `ProfileDTO` das Attribut `String password` entfernt. Die Attribute der Klasse `PersonDTO` sind vollständig dargestellt, die Klasse `CommercialDTO` hat ein zusätzliches Attribut `String tariff` erhalten. Dieses zusätzliche Attribut resultiert aus der flachen Assoziation zur `Tariff` Enumeration. Die zweite flache Assoziation ist in Abbildung 10.19, da sie sich auf ein Interface bezieht, nicht dargestellt. Da die Klasse `Post` im Beispiel des sozialen Netzwerks keine Attribute besitzt, können keine in die Klasse `Profile` eingebettet werden. Zusätzlich sind die Getter- und Setter-Methoden sowie die zusätzlichen Attribute gezeigt.

10.3.7 Modellierung der Autorisierung

Die Modellierung von Rechte-, Rollen- und Mappingdiagrammen für Clients des sozialen Netzwerks wurde in Abschnitt 5.3 bereits vorgestellt. Dabei wurden in Listing 5.12 und in Listing 5.13 die Rollen `User` und `Moderator` definiert. In Listing 5.15, Listing 5.16, Listing 5.17 und in Listing 5.18 wurden die Rechte für die Klassen `Post`, `Profile`, und `Person` vorgestellt. In Listing 5.22, Listing 5.23 und in Listing 5.24 wurden die zuvor modellierten Rechte der Klassen den Rollen zugeordnet. An dieser Stelle wird die Modellierung nicht nochmals erläutert, sondern auf die entsprechenden Abschnitte verwiesen.

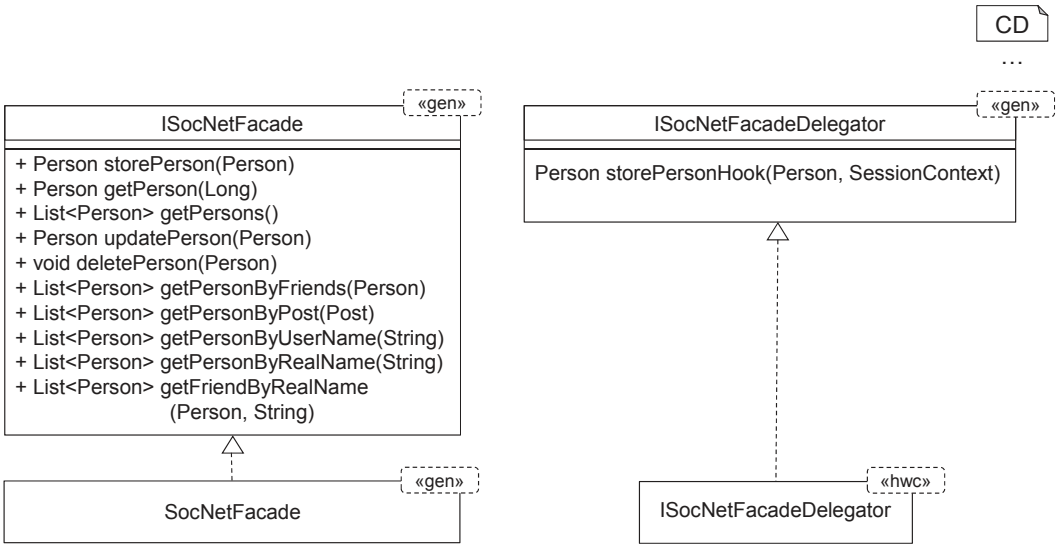


Abbildung 10.20: Darstellung des durch den Facade-Generator generierten Interfaces und der implementierenden Klasse. Diese basieren auf dem Domänenmodell aus Abbildung 10.9 sowie auf Rollen-, Rechte- und Mappingdiagramm.

10.3.8 Generierung der Fassaden

Auf Basis der Modellierung der Autorisierung wird eine generierte Fassade des sozialen Netzwerks basierend auf Abbildung 10.9 und der Tagdefinition aus Listing 10.10 gezeigt. Die allgemeine Abbildung der Modelle in das Generat des Facade-Generators wurde in Abschnitt 8.4 gezeigt.

Abbildung 10.20 zeigt das generierte Interface und die Implementierung der Fassade. Dazu sind die möglichen Methoden der Klasse Person dargestellt. Nicht dargestellt sind die Auswirkungen des Rollendiagramms und des Mappingdiagramms. Dies wird mit Hilfe von Annotationen im Quellcode realisiert und wurde in Abschnitt 8.4 vorgestellt. Gleichzeitig wird das Interface `ISocNetFacadeDelegator`, das weitere Prüfungen oder Erweiterungen erlaubt, generiert. Die Implementierung des Delegators muss handgeschrieben erfolgen. Darüber hinaus wird mit Hilfe des BusinessAPI-Generators eine Schnittstelle von der Fassade zur Geschäftslogik, wie in Abschnitt 8.3 detailliert erläutert, generiert.

Abbildung 10.21 zeigt diese generierte Schnittstelle von den Kommunikationsfassaden zur Geschäftslogik. Dies sind die drei Interfaces `IApplicationManager`, `IPerson-Manager` und `ICommercialManager`. Weitere drei Klassen, die die Interfaces implementieren, sind ebenfalls dargestellt. Die Klasse `ApplicationManager` ist generiert, die beiden Klassen `PersonManager` und `CommercialManager` sind handgeschrieben. Das Interface `IApplicationManager` beinhaltet die Vereinigung der Methoden der beiden Interfaces `IPersonManager` und `ICommercialManager`. Die generierte, im-

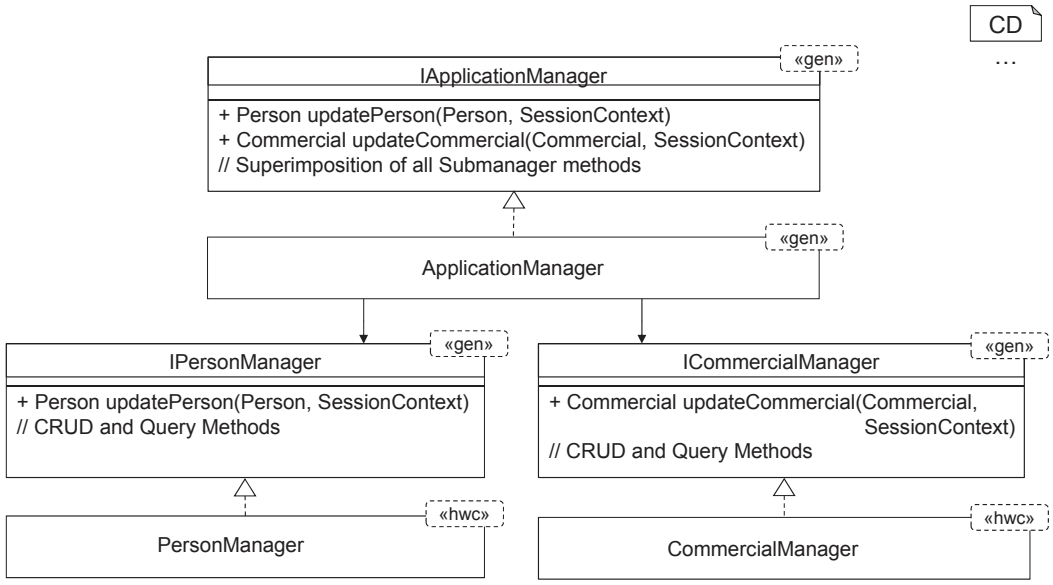


Abbildung 10.21: Darstellung der durch den BusinessAPI-Generator generierten API als Schnittstelle zwischen Kommunikationsfassade und Geschäftslogik. Diese basiert auf dem Domänenmodell aus Abbildung 10.9.

plementierende Klasse `ApplicationManager` implementiert die Interface Methoden und delegiert an die Submanager. Nachdem in diesem Abschnitt die Umsetzung des Szenarios unter Verwendung aller Generatoren gezeigt wurde, wird im nächsten Abschnitt auf das Deployment einer mit MontiEE-generierten Enterprise Applikation eingegangen.

10.4 Deployment des Generats

Das Deployment der Enterprise Applikation auf dem Applikationsserver (vgl. FA13-PE) erfolgt in Form eines EARs. Ein solches EAR besteht aus mehreren JARs und XML-Konfigurationsdateien (vgl. FA14-PE). Das Generat der einzelnen Generatoren muss für das Deployment in unterschiedliche JARs gepackt werden.

Abbildung 10.22 zeigt den generellen Aufbau des EARs. Es enthält ein Verzeichnis, das alle von der Enterprise Applikation benötigten Bibliotheken enthält. Dieses Verzeichnis trägt den Namen `lib`. In diesem Verzeichnis sind mehrere JARs enthalten. Diese sind zum einen Laufzeitabhängigkeiten der Applikation zu externen Bibliotheken, zum anderen ein JAR mit dem Namen `domain`, welches die vom Entity-Generator generierten Entitäten enthält und mehrere JARs, die die vom DTO-Generator für jeden Client separat generierten DTOs enthalten. Die JARs der DTOs enthalten lediglich die kompilierten Klassen und keine weiteren Zusatzinformationen. Das die Entitäten enthaltene JAR hingegen beinhaltet neben den kompilierten Klassen ein weiteres Verzeichnis, welches den Namen `META-INF` trägt und eine XML-Datei mit dem Namen `persistence.xml`

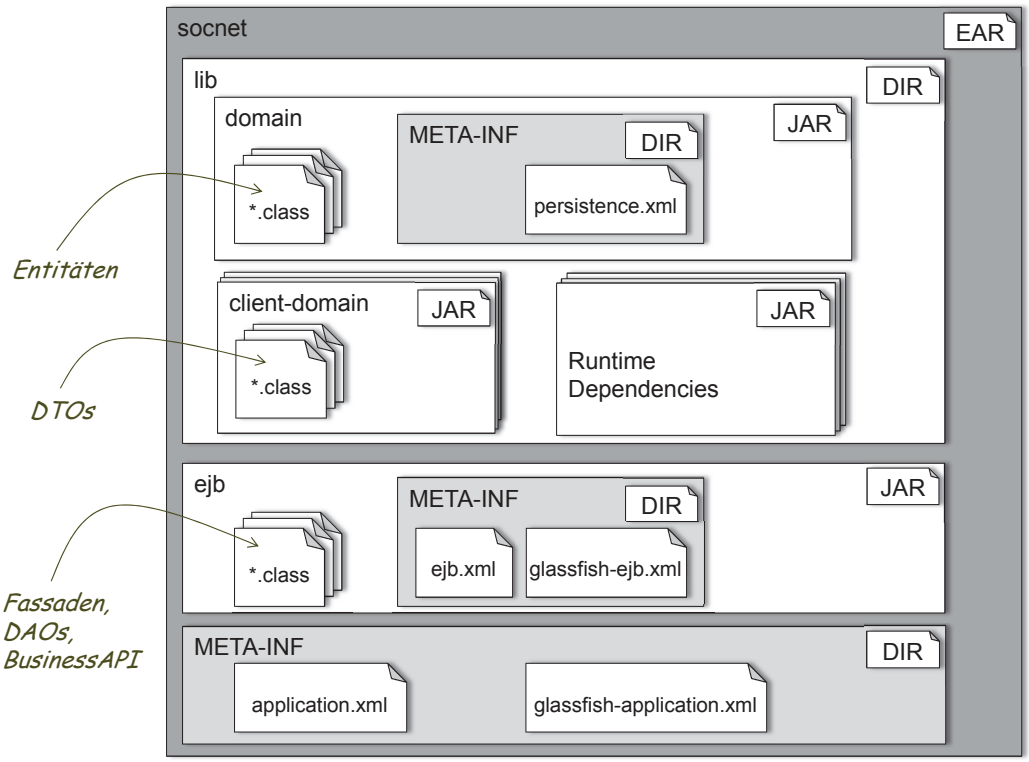


Abbildung 10.22: Darstellung des generelleren Aufbaus eines EARs, das aus mehreren JARs, Verzeichnissen und XML-Konfigurationsdateien besteht.

enthält. Diese Namen werden vom verwendeten Applikationsserver vorgeschrieben. Der Inhalt der `persistence.xml` wird im weiteren Verlauf dieses Abschnitts vorgestellt. Außerhalb des Bibliotheksverzeichnisses befindet sich ein weiteres JAR, das die Enterprise Beans enthält. In Abbildung 10.22 trägt dieses JAR den Namen `ejb`. Enthalten sind die vom DAO-Generator generierten DAOs, die vom BusinessAPI-Generator generierten Schnittstellen und die vom Facade-Generator generierten Kommunikationsfassaden. Neben den kompilierten Klassen beinhaltet dieses JAR wiederum ein Verzeichnis mit dem Namen `META-INF`, welches zwei XML-Dateien enthält. Diese Dateien sind die `ejb.xml` und die `glassfish-ejb.xml`. Erstere stellt dabei eine vom spezifischen Applikationsserver unabhängige Datei, die von der JEE gefordert wird, dar. Letztere hingegen enthält Informationen, die die unabhängigen Informationen um für den Applikationsserver spezifische Informationen ergänzen.

Auch direkt innerhalb des EARs existiert ein Verzeichnis mit Namen `META-INF`. In diesem Verzeichnis existieren erneut zwei XML-Dateien. Zum einen die `application.xml` und zum anderen die `glassfish-application.xml`. Wie zuvor, sind auch diese Dateien in einen vom Applikationsserver unabhängigen und einen spezifischen Teil getrennt. Das fertig gekapselte EAR kann auf dem Applikationsserver in eine Domäne mit einem `Realm` deployt werden. Der Realm sowie die Domäne müssen bereits ser-

verseitig konfiguriert sein. Sie kapseln Sicherheitsmechanismen, Datenbankverbindungen in Form von `ConnectionPools` und weitere Serverfunktionalität. Innerhalb der zuvor beschriebenen XML-Dateien ist spezifiziert, welcher Realm genutzt und welcher ORM zu verwenden ist. Zudem sind weitere spezifische Informationen enthalten. Ein Teil der XML-Dateien ist dabei generierbar.

Im nächsten Abschnitt werden die XML-Dateien und eine mögliche Generierung vorgestellt.

10.4.1 Erstellung der Konfigurationsdateien

Das Deployment einer Enterprise Applikation erfordert ein auf eine bestimmte Weise gepacktes EAR. Innerhalb dieses EARs werden XML-Dateien zur Konfiguration benötigt. Innerhalb des JARs, welches die Entitäten enthält, wird eine Datei, die den Namen `persistence.xml` trägt, benötigt. Ihr Inhalt wird nachfolgend vorgestellt.

persistence.xml

Die `persistence.xml` beinhaltet alle Informationen, die zur Persistierung der anfallenden Daten benötigt werden. Innerhalb der `persistence.xml` werden so genannte `persistence-units` definiert. Eine solche `persistence-unit` ist eine Verbindung zum Datenbankserver. Sie besitzt einen Namen und verwendet einen konfigurierten Transaktionstyp. Im Kapitel 7.5 wurde der Name als Attribut der Annotation `@PersistenceContext` als Referenz auf die hier verwendete `persistence-unit` angegeben. Darüber hinaus definiert sie den zu benutzenden ORM, die JDBC Verbindung zur Kommunikation mit dem Datenbankserver, den Dialekt der Datenbank und weitere ORM spezifische Konfigurationen.

XML

```
1  <persistence-unit name="socnet" transaction-type="JTA">
2    <provider>org.hibernate.ejb.HibernatePersistence</provider>    ...
3    <jta-data-source>jdbc/socnet</jta-data-source>
4    <properties>
5      <property name="hibernate.dialect"
6                value="org.hibernate.dialect.PostgreSQLDialect" />
7      <property name="hibernate.max_fetch_depth" value="5" />
8      <property name="hibernate.show_sql" value="true" />
9    </properties>
10 </persistence-unit>
```

Listing 10.23: Auszug aus der von MontiEE standardisiert generierten XML-Konfigurationsdatei `persistence.xml`.

Listing 10.23 zeigt einen Ausschnitt der `persistence.xml`. Der Name der `persistence-unit` ist `socnet` und der Transaktionstyp lautet `JTA`. Auf eine Unterscheidung

der möglichen Transaktionstypen wird im Rahmen dieser Arbeit nicht weiter eingegangen, sondern auf [Hef10] verwiesen. Innerhalb der `persistence-unit` wird mit Hilfe des `<provider>...</provider>` Blocks der verwendete ORM definiert. In Listing 10.23 wird Hibernate als ORM verwendet. Der verwendete Name des Providers ist der vollqualifizierte Name einer Klasse der Providerimplementierung. Dies ist sehr fehleranfällig und ein Entwickler benötigt sehr spezifisches Wissen. Zudem wird die serverseitige Abstraktion der JDBC Verbindung in dem `<jta-data-source>...</jta-data-source>` Block angegeben. Der dort verwendete Name ist serverseitig festgelegt, so dass die Konfiguration des Servers dem Entwickler bekannt sein muss. Der nachfolgende `<properties>...</properties>` Block beinhaltet Schlüssel-Wert-Paare. Die Schlüssel und die möglichen Werte sind von der Implementierung des jeweiligen Providers abhängig. In Listing 10.23 wird festgelegt, dass es sich um eine PostgreSQL-Datenbank handelt, das Nachladen tiefer Assoziationen beschränkt ist und das die vom ORM erzeugten SQL-Befehls im Log ausgegeben werden.

Die Erstellung dieser Konfigurationsdatei erfordert ein sehr hohes Maß an Technologiekenntnis. Die verwendete Technologie, aber auch ihre Konfigurationsdetails, müssen bekannt sein. MontiEE generiert eine standardisierte XML-Datei, die den Klassendiagrammnamen als `persistence-unit` Namen verwendet und die Technologien, wie PostgreSQL und Hibernate, sowie den Transaktionstyp fixiert. Allerdings muss diese standardisiert generierte Datei meistens auf die spezifischen Bedürfnisse angepasst werden. Als Erweiterung wäre die Nutzung eines Modells für das Deployment möglich, mit dessen Hilfe sowohl die Technologie ausgewählt als auch konfiguriert werden könnte.

ejb.xml und glassfish-ejb.xml

Neben der `persistence.xml`, die von dem JAR benötigt wird, welches die vom Entity-Generator generierten Entitäten enthält, benötigt das JAR, welches die Enterprise Beans, wie Fassaden, DAOs, BusinessAPI enthält, eine Konfigurationsdatei, die den Namen `glassfish-ejb.xml` hat. Da die allgemeinere `ejb.xml` im Rahmen von MontiEE nicht benötigt wird, wird sich an dieser Stelle auf die Erläuterung der `glassfish-ejb.xml` beschränkt.

Listing 10.24 zeigt einen Auszug der `glassfish-ejb.xml`. Innerhalb des `<glassfish-ejb-jar>...</glassfish-ejb-jar>` Blocks werden zwei unterschiedliche Elemente konfiguriert. Das erste Element definiert eine Zuordnung von Rollen zu Gruppen im `<security-role-mapping>...</security-role-mapping>` Block. Die dort angegebene Rolle wird von der Annotation `@DeclareRoles`, die, wie in Abschnitt 8.4 beschrieben, vom Generat des Facade-Generators verwendet wird, benötigt. Sie beschreibt die Rollen, die auf Methoden der Fassade zugreifen können. Der Applikationsserver verwendet intern Gruppen und keine Rollen. Mit Hilfe dieses Elements werden diese beiden Elemente vereint. Im Rahmen von MontiEE wird stets angenommen, dass eine gleichnamige Gruppe zu einer Rolle existiert. Die Informationen über die dem System bekannten Rollen sind in einem Rollendiagramm modelliert. Das zweite Element ist der `<enterprise-beans>...</enterprise-beans>` Block. Innerhalb dieses Blocks werden die Kommunikationsfassaden dem Applikationsserver bekannt gemacht,

```xml
<glassfish-ejb-jar>
  <security-role-mapping>
    <role-name>User</role-name>
    <group-name>User</group-name>
  </security-role-mapping>
  <enterprise-beans>
    <ejb>
      <ejb-name>ClientViewFacade</ejb-name>
        <webservice-endpoint>
          <port-component-name>ClientViewFacade</port-component-name
              >
          <login-config>
            <auth-method>BASIC</auth-method>
            <realm>socnetRealm</realm>
          </login-config>
          <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </webservice-endpoint>
    </ejb>
  </enterprise-beans>
</glassfish-ejb-jar>
```

Listing 10.24: Auszug aus der von MontiEE standardisiert generierten XML-Konfigurationsdatei `glassfish-ejb.xml`.

damit diese von den unterschiedlichen Clients verwendet werden können. Dazu wird der Name der Bean als entsprechender Endpunkt definiert. In Listing 10.24 ist die Variante einer Webservice-Fassade dargestellt. Zusätzlich zum Namen wird die Loginmethode konfiguriert. Dazu wird die standardmäßig vorhandene HTTP Autorisierung verwendet. Der Name des Realms kann mit Hilfe der zuvor vorgestellten Konfigurationsmöglichkeiten angepasst werden. Im Falle einer RPC-Fassade muss eine andere Konfiguration verwendet werden.

Im Rahmen von MontiEE wird diese Datei generiert. Dazu wird für jede Rolle des Rollendiagramms eine Zuordnung zu einer gleichnamigen Gruppe erstellt. Für jede Fassade wird, abhängig davon ob diese eine Webservice-oder RPC-Fassade ist, ein entsprechendes Element generiert. Zur Autorisierung wird immer die HTTP Autorisierung verwendet. Der Name des Realms kann konfiguriert werden.

Auch wenn diese Datei generiert wird, kommt es in der Praxis doch vor, dass Details verändert werden müssen. Momentan bietet MontiEE dafür keine Mechanismen an.

application.xml und glassfish-application.xml

Die letzten beiden Konfigurationsdateien sind die `application.xml` und die `glassfish-application.xml`. Beide liegen im `META-INF` Verzeichnis des EAR. Sie enthalten unterschiedliche Metainformationen der zu deployenden Enterprise Applikation. Dabei enthält die `application.xml` Informationen, die nicht Applikationsserver spe-

zifisch sind, sondern die Applikation generell beschreiben. Die für den Applikationsserver spezifischen Informationen befinden sich in der `glassfish-application.xml`.

```
1  <application>                                                    XML
2    <display-name>socnet-ear</display-name>                        ...
3    <module>
4      <ejb>ejb.jar</ejb>
5    </module>
6    <security-role>
7      <role-name>User</role-name>
8    </security-role>
9    <library-directory>lib</library-directory>
10 </application>
```

Listing 10.25: Auszug aus der von MontiEE standardisiert generierten XML-Konfigurationsdatei `application.xml`.

Listing 10.25 zeigt einen Ausschnitt der `application.xml`. In dieser Datei sind innerhalb des `<application>...</application>` Blocks der Name, die enthaltenen Module, die Rollen und das Bibliotheksverzeichnis definiert. Der Name der Applikation wird serverseitig angezeigt und hat keine Auswirkungen auf die Clients. Er dient lediglich der Übersicht bei der Serveradministration. Der `<module>...</module>` Block benennt den Namen der JARs, die die Enterprise Beans enthalten. Im Rahmen von MontiEE ist dies immer ein einziges JAR, welches die Fassaden-, die DAOs- und die BusinessAPI-Klassen sowie die `ejb.xml` und die `glassfish-ejb.xml` enthält. In Abbildung 10.22 heißt dieses JAR `ejb` und ist hier in der `application.xml` referenziert. Der Applikationsserver durchsucht nach dem Deployment der Applikation alle Module und macht die enthaltenen Webservices und Beans für Clients zugänglich. Daher müssen die zu durchsuchenden Module angegeben werden.

Zudem sind die Rollen, die dem System bekannt sind, hier definiert. Diese müssen mit den verwendeten Rollen in der `glassfish-ejb.xml`, wie in Listing 10.24 gezeigt, übereinstimmen. Darüber hinaus muss das Bibliotheksverzeichnis innerhalb des `<library-directory>...</library-directory>` Blocks angegeben werden. Der Applikationsserver spannt für jede Applikation einen Classpath auf. Dieser Classpath beinhaltet alle Bibliotheken, die im Bibliotheksverzeichnis vorhanden sind. Dies erlaubt es, unterschiedliche Applikationen auf demselben Applikationsserver zu deployen, ohne dass dabei Konflikte zwischen verschiedenen Bibliotheken entstehen.

Listing 10.26 zeigt einen Auszug aus der für den Applikationsserver spezifischen Konfigurationsdatei `glassfish-application.xml`. Diese enthält lediglich den Namen des Realms, der beim Deployment verwendet wird.

Im Rahmen von MontiEE sind alle Informationen zur Generierung der `application.xml` und der `glassfish-application.xml` vorhanden. Als Name wird der Name des Klassendiagramms mit dem Suffix `-ear` verwendet. Das Modul, welches die

```xml
<glassfish-application>
  <realm>socnetRealm</realm>
</glassfish-application>
```

Listing 10.26: Auszug aus der von MontiEE standardisiert generierten XML-Konfigurationsdatei `glassfish-application.xml`.

Enterprise Beans enthält, wird immer `ejb` genannt. Die Rollen sind durch die Rollendiagramme bekannt und das Bibliotheksverzeichnis wird stets `lib` genannt. Darüber hinaus ist der Name des Realms in der Konfiguration einstellbar.

10.5 Werkzeugunterstützung

Nachdem zuvor die Modellierung, die Konfiguration und das Deployment von Enterprise Applikationen mit MontiEE sowie dessen Ausführung vorgestellt wurden, wird in diesem Abschnitt die geschaffene Werkzeugunterstützung, die dem MontiEE-Entwickler zur Verfügung steht, kurz vorgestellt. Die Werkzeugunterstützung bezieht sich dabei auf Eclipse-Editoren, Maven-Archetyps und einen Eclipse-Wizard. Die Werkzeugunterstützung wurde im Rahmen studentischer Arbeiten umgesetzt. Sie liegt nicht im Fokus dieser Arbeit und wird daher nur kurz einführend erklärt. MontiEE bietet Werkzeugunterstützung im Umfeld von Eclipse unter Verwendung von Maven als Buildwerkzeug an. MontiEE selbst ist unabhängig von diesen Werkzeugen und kann auch ohne deren Verwendung von der Kommandozeile ausgeführt werden.

Zunächst werden die Editoren vorgestellt. Bereits von der UML/P werden Editoren für Klassen- und Objektdiagramme bereitgestellt [Sch12]. Zur Modellierung der Sichten wurde in [Sch13] ein Editor umgesetzt. Dieser Editor verwendet die von MontiCore bereitgestellten Möglichkeiten zur Editorgenerierung und erlaubt Syntaxhighlighting von Sichten. Darüber hinaus werden die Kontextbedingungen direkt im Editor geprüft und die Hyperlink Funktionalität von Eclipse kann dazu verwendet werden, zwischen Elementen der Sicht und des Klassendiagramms zu navigieren. Zur Modellierung der Rechte- und Rollendiagramme sowie der Mappingdiagramme wurde in [Cöm13] ebenfalls ein Editor umgesetzt, der Syntaxhighlighting und Hyperlinks unterstützt. Zur Modellierung der Tagdefinitionen existiert momentan keine Editorunterstützung.

Neben den Editoren wurden in [Pot14] Archetyps geschaffen, die Templates für verschiedene Arten von Projekten im Umfeld von Maven und Eclipse darstellen. Wie zuvor in Abschnitt 10.4 beschrieben, wird zum Deployment einer Enterprise Applikation eine vorgegebene Struktur in Form eines EARs benötigt. MontiEE schreibt dem Verwender nicht vor, wie er die Enterprise Applikation zusammenbaut oder deployt. Allerdings vereinfacht die Verwendung der Maven-Archetyps dies enorm, da sie ein Projektlayout vorgeben. Es existieren fünf verschiedene Archetyps für unterschiedliche Projekte. Der erste Archetyp erzeugt ein Modell-Projekt, welches alle von MontiEE verwendeten Modelle enthält. Dies bedeutet, dass vorgesehen ist, dass in ihm das Domänenmodell, eine

Tagdefinition, Sichten, Rechte-, Rollen- und Mappingdiagramme liegen. Der Archetyp legt für jede Art von Modell ein beispielhaftes Modell in das erzeugte Projekt.

Darüber hinaus steht ein zweiter Archetyp, der ein Projekt angelegt, welches das JAR erzeugt, das die Entitäten enthält, zur Verfügung. Innerhalb dieses Projekts wird das Klassendiagramm und die Tagdefinition von den MontiEE-Generatoren dazu verwendet, die Entitäten und das SQL-Schema zu generieren. Das erzeugte JAR wird `domain.jar` genannt. Zusätzlich wird die `persistence.xml` dort generiert und ebenfalls in dem JAR gebündelt.

Der dritte Archetyp erzeugt ein Projekt, das ein JAR, welches DTOs für einen exemplarischen Client erzeugt, erzeugt. Die dazu benötigte Sicht liegt in dem Projekt, welches die Modelle enthält. Das JAR heißt dabei wie die exemplarische Sicht. Sollen mehrere Clients unterstützt werden, muss der Archetyp mehrfach angewendet und mehrere solcher Projekte für unterschiedliche Clients angelegt werden. Zudem müssen im Modell-Projekt mehrere Sichten angelegt werden.

Ein vierter Archetyp erstellt ein Projekt, das das JAR erzeugt, welches die Enterprise Beans enthält. Hier werden die MontiEE-Generatoren dazu verwendet, die DAOs, Fassaden und die BusinessAPI zu erzeugen. Auch hier werden die im Modell-Projekt abgelegten Modelle verwendet. Gleichzeitig werden die `ejb.xml` und die `glassfish-ejb.xml` generiert. Die generierten Klassen und die generierten XML-Dateien werden in einem JAR gebündelt, das den Namen `ejb.jar` trägt.

Der letzte Archetyp erzeugt ein Projekt, dessen Aufgabe das Bündeln der JARs in ein EAR ist. Auch die `application.xml` und die `glassfish-application.xml` werden hier generiert. Die JARs, die Entitäten und DTOs enthalten, werden in das Bibliotheksverzeichnis gebündelt. Das JAR, das die Enterprise Beans enthält, wird zusammen mit der `application.xml` und der `glassfish-application.xml` in das EAR gebündelt.

Das Projektlayout, welches durch die Archetypen vorgegeben wird, ist technologisch bedingt und wird zum Erstellen des EARs benötigt. Die Modelle befinden sich allesamt in einem eigenständigen Projekt, das von den anderen Projekten nur verwendet wird. Somit muss der Entwickler sich nur mit diesem einen Projekt und den dort enthaltenen Modellen befassen. Eine weitergehende Einführung über den Zweck von Archetyps ist in [Sir15] gegeben. In [Pot14] wurde ein Projektwizard entwickelt, der auf Basis der zuvor beschriebenen Archetyps Eclipseprojekte erzeugt und konfiguriert. Darüber hinaus bettet dieser Wizard die Projekte direkt in verwendete Entwicklungsinfrastruktur ein. Als Infrastruktur wird das SSELab [Her14] verwendet, welches ein Subversion (SVN) Repository und das Ticketsystem Trac [Tra16] bereitstellt. Der Wizard konfiguriert das Eclipseplugin Mylyn [Myl16], so dass es mit dem vom SSELab bereitgestellten Ticketsystem verbunden wird. Auch die Verbindung zu einem Continuous Integration System, wie Jenkins [Jen16], wird hergestellt und mit Hilfe eines Eclipseplugins in die Entwicklungsumgebung integriert. Darüber hinaus erstellt der Wizard zwei Konfigurationsskripte, die den Applikationsserver und den Datenbankserver initial konfigurieren. Dazu werden Benutzer, `ConnectionPools`, Domäne und weitere benötigte Elemente erzeugt.

10.6 Fallstudien

In den vorangegangenen Kapiteln und Abschnitten wurde MontiEE vorgestellt. Dabei wurden die enthaltenen Sprachen zur Modellierung der einzelnen Teile des Systems und die Generatoren, die aus den abstrahierten, technologieunabhängigen Modellen JEE konformen Java-Code erzeugen, vorgestellt. Dieser Code ist technologiespezifisch und kann auf einem Glassfish Applikationsserver ausgeführt werden. Er verwendet Hibernate als ORM und PostgreSQL als Datenbankserver. Zudem wurden die Verwendung von MontiEE, seine Konfiguration und seine Erweiterungsmöglichkeiten vorgestellt. Dabei wurde gezeigt, dass die einzelnen Generatoren auch unabhängig voneinander verwendet werden können und nicht immer alle Bestandteile einer Enterprise Applikation generiert werden müssen. Dies bedeutet, dass MontiEE in unterschiedlichen Ausprägungen eingesetzt werden kann. Dadurch kann es flexibel bei Neuentwicklungen, aber auch bei bestehenden Systemen eingesetzt werden.

In diesem Abschnitt werden Fallstudien und Erfahrungen des Einsatzes von MontiEE vorgestellt. Dazu werden die Integration von MontiEE in ein bestehendes System sowie Hindernisse und Nutzen der nachträglichen Integration beschrieben. Darüber hinaus wird der Einsatz bei neuentwickelten Systemen im Rahmen internationaler Forschungsprojekte beschrieben. Sowohl die nachträgliche Integration als auch die Verwendung bei der Neuentwicklung beinhalten unterschiedliche Herausforderungen. Es sei dabei angemerkt, dass die im Rahmen dieser Arbeit erhobenen Zahlen nicht statistisch belastbar sind, sondern lediglich die Erfahrungen, die im Umgang mit MontiEE gemacht wurden widerspiegeln. Die erhobenen Zahlen beruhen auf einer Implementierung unterschiedlicher Systeme. Dennoch können diese Zahlen nur Hinweise auf den Einsatz und den Nutzen geben, lassen aber keine statistisch relevanten Folgerungen zu.

Zunächst wird die Integration in bestehende Systeme, wie den Energie Navigator [Pin14], vorgestellt. Daran anschließend wird die Verwendung bei der Neuentwicklung im Rahmen der Forschungsprojekte COOPERATE und WATTALYST vorgestellt.

10.6.1 Integration in bestehende Systeme

Bei der Integration in bestehende Systeme liegt die Herausforderung in der Ablösung des zuvor handgeschriebenen Codes durch einen generativen Ansatz.

MontiEE wurde im Rahmen der Entwicklung des Energie Navigators [FLP+11a, Pin14] eingesetzt. Der Energie Navigator stellt eine Methodik und ein Expertenwerkzeug zur Planung und Analyse sowie zur Optimierung des Gebäudebetriebs dar. Übergeordnet setzt sich der Energie Navigator als Ziel, logische Anlagen- und Gebäudefunktionen zu modellieren und die korrekte Funktionsweise automatisiert zu analysieren. Dazu wurde im Rahmen von [Pin14] eine DSL zur Modellierung von Performanzzielen und Regelungsverhalten als ”aktive Funktionsbeschreibung”, wie in [Ple13], aus Sicht der fachlichen Anwendungsdomäne beschrieben, vorgestellt und umgesetzt.

Mit Hilfe der aktiven Funktionsbeschreibung wird stets das Sollverhalten eines Gebäudes oder einer Anlage bereits zur Planungszeit definiert. Auf Basis dieses Sollverhaltens wird das Gebäude errichtet und die Anlage entsprechend umgesetzt und konfiguriert.

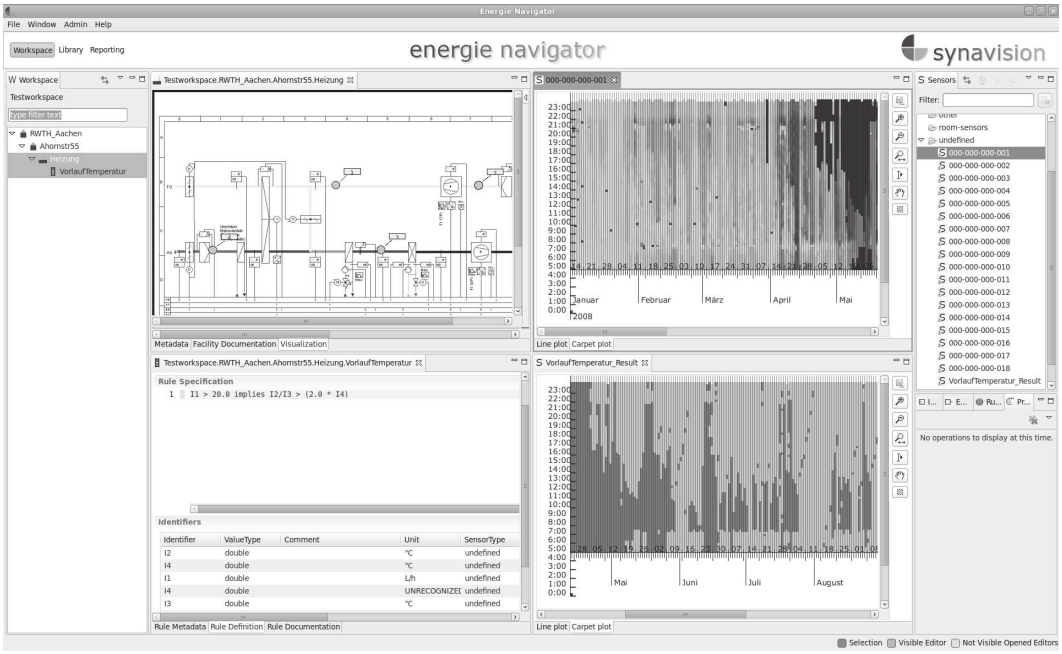


Abbildung 10.27: Abbildung des Expertentools des Energie Navigators. Das Domänenmodell der Visualisierung wurde mit MontiEE generiert. Quelle: [Pin14].

Während des Gebäudebetriebs wird das spezifizierte Sollverhalten mit Sensordaten angereichert. Dies ermöglicht einen Vergleich zwischen dem spezifizierten Sollverhalten und dem gemessenen Ist-Zustand des Gebäudes auf Basis der Sensordaten. Zur Spezifikation stehen dem Experten unterschiedliche Modellierungselemente [FLP+11b, KLPR12] zur Verfügung: Regeln, Funktionen, Konstanten, Metriken, Zeitprogramme, Kennlinien und Zustandsmodellierung. Regeln und Funktionen erlauben die Modellierung mathematischer Ausdrücke, wobei Regeln immer einen Wahrheitswert und Funktionen einen Zahlenwert als Ergebnis liefern. Metriken aggregieren Sensordaten über einen gewissen Zeitraum. Zeitprogramme erlauben die Selektion ausgewählter Daten eines Zeitraums. Kennlinien modellieren eine vorgegebene Eigenschaft, der die gemessenen Sensorwerte unter Berücksichtigung einer Toleranz entsprechen müssen. Die Zustandsmodellierung erlaubt es, unterschiedliche Betriebsmodi eines Gebäudes oder einer Anlage zu beschreiben. Die genaue Semantik der Elemente ist in [KLPR12, Pin14] detailliert aufgeführt.

Neben der Methodik stellt der Energie Navigator auch ein Softwarewerkzeug zur Verfügung. Abbildung 10.27 zeigt einen Ausschnitt des Expertenwerkzeugs, welches einen vollwertigen, selbständigen Client darstellt. In der Abbildung sind die schematische Darstellung einer Anlage und der verbauten Sensoren in der linken oberen Hälfte zu erkennen. Darunter befindet sich die Definition einer Regel, die die Sensoren der Anlage verwendet. In der rechten Hälfte sind zwei Auswertungen erkennbar. Oben ist eine graphische Darstellung von Sensorwerten mit Hilfe eines `Carpet Plots` [Pin14] dargestellt.

Unten ist die graphische Darstellung der Auswertung der spezifizierten Regel gezeigt. Da eine Regel immer Wahrheitswerte als Ergebnis liefert, stellt der Carpet Plot diese als rot oder grün dar. Beim obigen Carpet Plot wird die Farbe durch den Messwert des Sensors bestimmt. Je kleiner der gemessene Sensorwert ist, desto blauer ist die graphische Darstellung. Je größer der gemessene Sensorwert, desto roter ist die Darstellung.

Der Energie Navigator ist als Client-Server Applikation mit unterschiedlichen Clients realisiert. Neben dem zuvor vorgestellten Expertenwerkzeug existiert eine Administrationsoberfläche, eine Benutzerverwaltung, ein Ticketsystem und ein Webclient zur einfachen Erstellung von Grafiken, der in Abbildung 10.28 dargestellt ist.

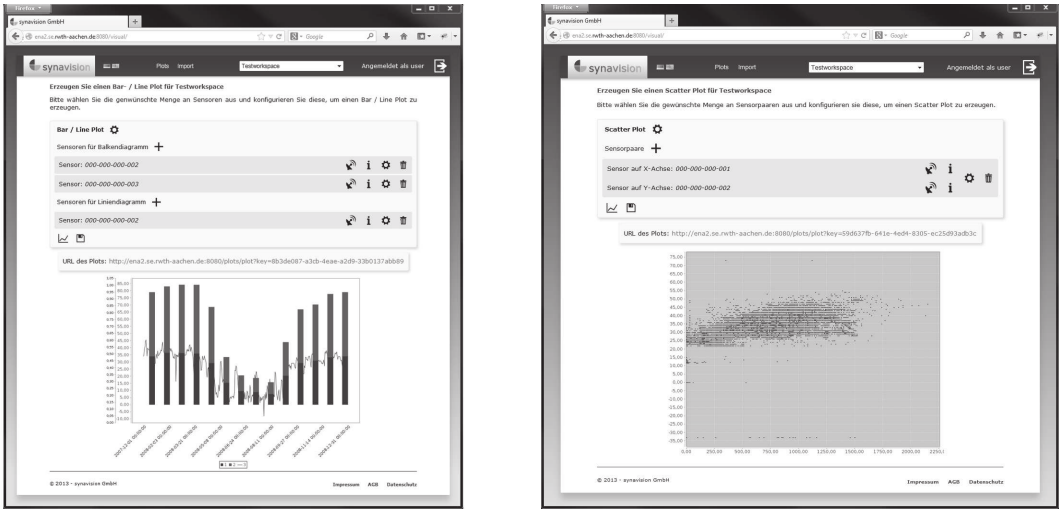


Abbildung 10.28: Abbildung der Visualisierungsmöglichkeiten des Webclients des Energie Navigators. Das Domänenmodell der Visualisierung wurde mit MontiEE generiert. Quelle: [Pin14].

Der dargestellte Webclient zeigt zwei weitere mögliche Plots, die mit dem Energie Navigator erzeugt werden können: den `Line Plot` und den `Scatter Plot`. Auf Seite des Servers ist der Energie Navigator eine JEE Anwendung, die auf einem Glassfish Application Server läuft. Die Applikation stellt RPC-Fassaden zur Kommunikation bereit, die CORBA zur Kommunikation verwenden. Der Webclient verwendet bereitgestellte Webservice-Fassaden und auch die entsprechende Kommunikation. Darüber hinaus verwendet der Energienavigator eine PostgreSQL sowie eine HDF5 Datenbank [RW12] zur Speicherung persistenter Daten. Als ORM wird Hibernate verwendet. Zudem kommt das DAO-Muster zur transparenten Speicherung zum Einsatz.

MontiEE wurde im Rahmen des Energie Navigators zur Generierung der Entitäten der Visualisierung eingesetzt. Zum Zeitpunkt des Einsatzes von MontiEE waren bereits ein großer Teil des Energie Navigators und seiner Architektur manuell umgesetzt. Das Generat von MontiEE musste sich also in eine bestehende Architektur und gewachsene Strukturen eingliedern und die getroffenen Entwurfsentscheidungen ebenfalls umsetzen.

Das Generat erweitert den handgeschriebenen Java-Code, ersetzt diesen zum Teil aber auch. Dadurch muss das Generat ebenfalls die Vorgaben des bestehenden Codes erfüllen. Diese Vorgaben beinhalten die Implementierung bestimmter Interfaces, die in der Gesamtarchitektur benötigt werden, die Einhaltung von Namens- und Programmierkonventionen und Funktionalität. Allerdings gibt generierter Code auch immer eine in gewisser Weise schematische API und Verwendung des Codes vor. Dies führt dazu, dass im Wesentlichen zwei große Probleme bei der Integration bestehen. Zum einen muss das Generat hochspezifisch um manuelle Vorgaben erweitert werden, zum anderen muss der manuell geschriebene Quellcode sich an die schematischen API Vorgaben des Quellcodes halten. Dies erzeugt einen hohen Aufwand bei der Generatorentwicklung, aber auch bei der Integration durch Anpassung des manuellen Codes. So muss der manuelle Quellcode dahingehend angepasst werden, dass er nicht mehr der bisher geltenden manuellen Konvention, sondern dem vom Generat vorgegebenen Schema, folgt. Gleichzeitig muss das Generat Funktionalität, die von der Architektur und den Entwurfsentscheidungen vorausgesetzt wird, bereitstellen. Diese Funktionalität beruht unter Umständen auf der Verwendung handgeschriebener Klassen aus dem Generat. Dies ist problematisch, da der Generator von der Existenz bestimmter Methoden ausgeht, die einen vorgegebenen Namen und Parameter besitzen. Typischerweise wird dies durch eine Laufzeitumgebung des Generats, die Interfaces bereitstellt, gelöst. Handgeschriebene Klassen müssen diese Interfaces implementieren, so dass sichergestellt werden kann, dass die benötigten Methoden existieren. Dies ist aber in einer bestehenden Architektur und in bestehendem Quellcode nicht der Fall. Somit müssen entweder die von der Laufzeitumgebung bereitgestellten Interfaces nachträglich implementiert werden oder das Generat muss darauf reagieren können. Letzteres ist nur schwerlich möglich, ersteres hingegen erfordert einen sehr hohen Aufwand der Anpassung des existierenden Quellcodes.

Falls der handgeschriebene Quellcode häufigen Änderungen unterliegt und stetig weiterentwickelt wird, kann der benötigte Änderungsaufwand, indem die benötigten Änderungen dort mit einfließen, reduziert werden. Konvergiert der handgeschriebene Quellcode bereits, hat eine gewisse Stabilität und wird nicht mehr stark weiterentwickelt, stellt sich der Aufwand als sehr hoch dar. Aus diesem Grund wurden im Rahmen des Energie Navigators nur begrenzt bereits existierende Klassen durch einen generativen Ansatz abgelöst. Die Klassen der Visualisierungskomponente und weitere Entitäten wurden modellgetrieben entwickelt. Dazu wurden die benötigten Klassen und Entitäten der Visualisierung, also der unterschiedlichen Plots, mit MontiEE generiert.

Im Rahmen des Energie Navigators wurde MontiEE für zwei unterschiedliche Dinge eingesetzt. Zum einen wurden mit Hilfe von MontiEE Entitäten und zugehörige SQL-Skripte, wie in den Abschnitten 7.4 und 7.6 dargestellt, generiert. Dabei wurden durch den modularen Aufbau des Energie Navigators zwei verschiedene Datenbankschemas, so dass je ein Modell zur Erstellung des Schemas und ein Skript zum Entfernen des Schemas existiert, benötigt. Daher werden vier verschiedene SQL-Skripte generiert. Zum anderen wurden normale Java-Klassen als Domänenmodell der Visualisierung modelliert. Diese Java-Klassen enthalten keinerlei technologiespezifische Annotationen. In Tabelle 10.29 werden diese Modelle als unterschiedliche Zeilen dargestellt. Generell wurde zur Auswer-

tung die Anzahl der Java-Dateien, der Modelldateien, der SQL-Skriptdateien und der Lines of Code (LOC) gezählt. Für die Java-Dateien wurden die Quelldateien des Generats verwendet. Für das Errechnen der LOC wurden keine Leerzeilen und Kommentare verwendet. Darüber hinaus wurden Zeilen, die nur eine einzelne geschweifte Klammer, zum Öffnen oder Schließen eines Blocks aufweisen, nicht gezählt. Darüber hinaus stellt das Betrachten der LOC eine Metrik dar, die bei einem generativen Ansatz nicht unbedingt aussagefähig ist. Im Rahmen von MontiEE wird eine generierte im Vergleich zu einer äquivalenten handgeschriebenen Klasse ebenfalls einen größeren LOC Wert erhalten. Dies liegt aber nicht daran, dass der Generator beliebigen Quellcode generiert, der nicht nutzbar ist, sondern in der naturgemäßen schematischen Abbildung von Eingabemodellen in das Generat. Handgeschriebene Klassen verkürzen verschiedene Dinge oder sind durch Auslassungen unvollständig. Dies ist bei einem Generator nicht gegeben. Somit sind die absoluten LOC weniger aussagekräftig als der prozentuale Vergleich, der hier angestrebt wird.

Tabelle 10.29: Überblick über die Anzahl der generierten Artefakte und der enthaltenen Quellcodezeilen (LOC) im Rahmen des Energie Navigators.

	Dateien		LOC		Anteil	
	«gen»	«hwc»	«gen»	«hwc»	«gen»	«hwc»
Domänenmodell (Visualisierung)	0	11	0	462	0%	100%
Domänenmodell (Entitäten)	0	3	0	194	0%	100%
Klassen	273	109	3007	4818	38%	61%
Entitäten	191	17	5658	1309	81%	19%
SQL	4	0	382	0	100%	0%
Summe	468	126	9047	6127	60%	40%

Tabelle 10.29 zeigt, dass aus den drei Eingabemodellen, die die Entitäten beinhalten, 191 Klassen mit 5658 LOC erzeugt wurden. Zur Integration von handgeschriebenem Quellcode mussten 17 Klassen, die 1309 LOC enthalten, manuell angelegt werden. Hier werden 81% des Quellcodes generiert. Das SQL-Skript wird vollständig generiert. Dem gegenüber konnten 273 Java-Klassen aus den elf Modellen des Domänenmodells der Visualisierung, die aus 3007 LOC bestehen, generiert werden. Dazu wurden 109 Klassen, die aus 4818 LOC bestehen, manuell implementiert. Hier ist das Verhältnis von generiertem zu handgeschriebenem Quellcode umgekehrt. Es konnten lediglich 38% generiert werden. Eine mögliche Erklärung dafür ist, dass Entitäten meistens als reine Datenbehälter fungieren und somit keine eigene Funktionalität besitzen. Normale Java-Klassen hingegen enthalten eigene Funktionalität, die manuell implementiert wurde. Zudem enthalten Entitäten eine Vielzahl technologiespezifischer Annotationen, die sich gut dazu

eignen, generiert zu werden. MontiEE selbst zielt auf die Generierung von Enterprise Applikationen mit den beteiligten Komponenten. Diese lassen sich, wie Tabelle 10.29 zeigt, gut generieren. In Summe konnten im betrachteten Teil des Energie Navigators 60% des Quellcodes generiert werden. Wird die Anzahl der Zeilen des Modells mit der Anzahl Zeilen des Quellcodes verglichen, ergibt sich ein Verhältnis von 1:14.

Generell zeigte sich, dass die Modellierung der Entitäten vorteilhaft war. Auch die Generierung des Domänenmodells der Visualisierung war für die Entwickler, da diese durch die Modellierung ihre Effizienz steigern konnten, vorteilhaft. Ein weitaus größer Vorteil ist aber zu erwarten, wenn MontiEE in Neuentwicklungen eingesetzt wird, da beim Energie Navigator bereits ein Großteil der Domäne manuell implementiert war.

Nachdem MontiEE im Rahmen der Energie Navigator Plattform in ein bestehendes System integriert wurde, wurde es auch im Rahmen weiterer Forschungsprojekte bei Neuentwicklungen verwendet. Diese werden im nachfolgenden Abschnitt vorgestellt.

10.6.2 Verwendung in Neuentwicklungen

MontiEE wurde nicht nur in bestehende Systeme integriert, sondern auch bei Neuentwicklungen verwendet. Im Rahmen der Verwendung in Neuentwicklungen wurde MontiEE in zwei verschiedenen Projekten eingesetzt: COOPERATE und WATTALYST. Dabei wurde der Entity-, der DAO-, der SQL-, der DTO- sowie der Facade-Generator verwendet. Der weitere Quellcode wurde manuell umgesetzt.

COOPERATE

Das Forschungsprojekt COOPERATE[1] (Contral and Optimization for Energy Positive Neighbourhoods) befasst sich mit der Erforschung energieeffizienter Nachbarschaften. Dabei wird die Möglichkeit zur Kooperation existierender Gebäude und Infrastrukturen untersucht. Einzelne Gebäude, die bereits mit einem Automatisierungssystem ausgestattet sind, werden häufig als singuläre Einheiten betrachtet und ebenso optimiert. Dies führt aber zu einem lokalen Optimum. Werden diese Gebäude integriert, lässt sich ein nachbarschaftsweites Optimum finden. Die Optimierung beruht auf vielen Faktoren, wie verbesserte Materialien, effizientere Verbraucher aber vor allem auf der Möglichkeit zur Nutzung erneuerbarer Energien. Die so erzeugte Energie kann direkt vom Erzeuger verbraucht, zwischengespeichert oder abgegeben werden. Wird ein einzelnes Gebäude betrachtet, hängt das Optimum häufig am aktuellen Verbrauch oder der zur Verfügung stehenden Batteriekapazität. Wird dies auf eine ganze Nachbarschaft erweitert, können Verbraucher in anderen Gebäuden oder Verbraucher der Infrastruktur, wie Straßenbeleuchtung, versorgt werden. Zudem können Energiespeicher effizienter genutzt werden. Dies erfordert auch eine Verbesserung bestehender Demand-Response Systeme. Durch eine Ausweitung der Betrachtung lässt sich nicht nur ein nachbarschaftsweites Optimum finden, es lassen sich auch neue Dienstleistungen und Interaktionen, die die einzelnen Elemente einer Stadt berücksichtigen, schaffen [GLPR15].

[1] http://cooperate-fp7.eu/

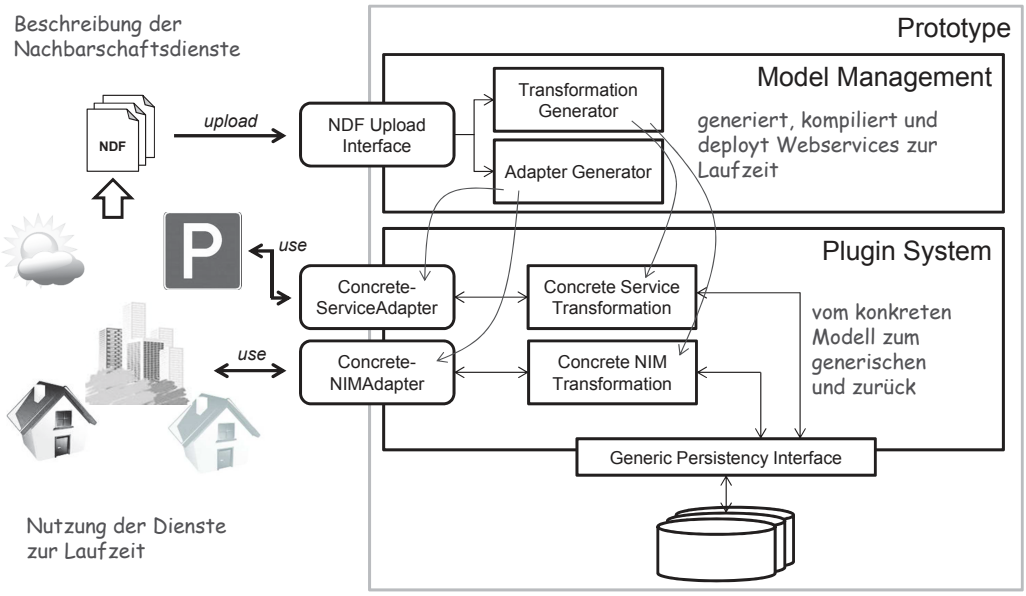


Abbildung 10.30: Schematische Abbildung des im Rahmen des COOPERATE Projekts mit MontiEE entwickelten Prototypen. Quelle: [GLRK14].

Auf technischer Ebene erfordert die Umsetzung des COOPERATE Projekts eine Integration der bereits existierenden Automatisierungssysteme der einzelnen Gebäude. Zudem muss die Entwicklung von Services dahingehend erweitert werden können, dass diese mit den verschiedenen proprietären Systemen interagieren können. Dazu wurde im Rahmen des Projekts ein System-of-Systems Ansatz, der eine agile Integration der heterogenen Systeme ermöglicht, gewählt. Ein Bestandteil dieses Ansatzes ist ein Nachbarschaftsinformationsmodell (NIM). Dieses Datenmodell stellt ein Metamodell, auf das weitere Datenmodelle abgebildet werden können, dar. Darüber hinaus wurde eine prototypische Plattform, die Webservice-Schnittstellen für unterschiedliche Services anbietet, geschaffen. Die Idee ist, dass jeder Service sein eigenes Datenmodell verwendet und auch Daten, die zu seinem eigenen Datenmodell konform sind, über eine Webservice-Schnittstelle anfragt. Innerhalb der Plattform sind die Daten in dem NIM Metamodell in einer Datenbank gespeichert. Die Plattform lädt diese Daten und transformiert sie in Daten des Datenmodells des Service.

Abbildung 10.30 gibt eine Übersicht des Prototyps. Services verwenden die konkreten Adapter, um Daten, die konform zu ihrem Datenmodell sind, abzufragen. Dazu muss zu jedem Service eine Datenmodellbeschreibung, die eine Abbildungsvorschrift zwischen dem NIM und dem Datenmodell des Services enthält, erstellt werden. Aus dieser Beschreibung wird die Transformation generiert [Han14]. Die Transformationen transformieren die Instanzen des Datenmodells des Service in NIM Instanzen und vice versa. Dazu verwenden sie ein Persistenzinterface, welches NIM Instanzen in einer Datenbank speichert.

Im Rahmen des Projekts wurde MontiEE dazu verwendet, das Datenbankschema, die DAOs und die Entitäten, wie in den Abschnitten 7.6, 7.5 und 7.4 präsentiert, des NIMs zu erstellen. Die weiteren MontiEE-Generatoren wurden an dieser Stelle, da der Fokus des Projekts auf dem Datenmodell und der persistenten Speicherung der anfallenden Daten lag, nicht benötigt. Aus diesem Grund wurden die weiteren Generatoren nicht verwendet. Die Daten wurden, wie zuvor beim Energie Navigator beschrieben, erhoben.

Tabelle 10.31: Überblick über die Anzahl der generierten Artefakte und der enthaltenen Quellcodezeilen (LOC) im Rahmen des COOPERATE Projekts.

	Dateien		LOC		Anteil	
	«gen»	«hwc»	«gen»	«hwc»	«gen»	«hwc»
Modell	0	2	0	88	0%	100%
Entitäten	50	2	911	43	95%	5%
DAO	26	15	702	753	48%	52%
SQL	2	0	136	0	100%	0%
Summe	78	17	1749	796	69%	31%

Tabelle 10.31 zeigt eine Statistik des generierten Codes und die handgeschriebenen Elemente. Es wurden zwei Domänenmodelle handgeschrieben. Die Modelle bestehen aus 88 Zeilen. Aus diesen Modellen wurden 911 LOC erzeugt. Zusätzlich wurden zwei handgeschriebene Delegatoren, die die Entitäten um Funktionalität anreichern, benötigt. Bezogen auf die Entitäten konnten 94% des benötigten Quellcodes generiert werden. Bei den DAOs wurden 15 Klassen manuell implementiert. Dies sind im Wesentlichen Exception Klassen. Die Klassen bestehen aus 753 LOC. Demgegenüber wurden 26 DAOs mit 702 LOC erzeugt. Dadurch beträgt der Anteil des Generats bezogen auf die DAOs 48%. Die generierten SQL-Skripte bestehen aus 136 Zeilen generierten SQL-Befehls. Insgesamt konnten im Rahmen des COOPERATE Projekts 69% der benötigten Infrastruktur generiert werden. Dies entspricht in etwa einem Verhältnis von 1:20 von Modell zu Quellcode. Es hat sich während des Projekts gezeigt, dass die Modellierung des Datenmodells es ermöglicht, agil auf Änderungen zu reagieren und sehr schnell eine erste lauffähige Version der Implementierung zu erschaffen. Dazu konnten die Entwickler sich auf die Modellierung des Domänenmodells beschränken und mussten sich nicht mit Technologiespezifika beschäftigen. Zusammenfassend war die Verwendung von MontiEE vorteilhaft für die Umsetzung der Implementierung des COOPERATE Projekts.

WATTALYST

Das Forschungsprojekt WATTALYST[2] (WATT anALYST - Modeling and Analyzing Demand Response Systems) untersucht Anreize und Methoden für Energieverbraucher, den eigenen Verbrauch des Haushalts zu reduzieren. Dazu wird untersucht, welche Anreize für Mitglieder des Haushalts interessant sein können, wo diese Energie sparen können und welche Herausforderungen dies für Demand-Response Systeme birgt. Dabei wird eine Klassifikation, welche Verbraucher innerhalb eines Haushalts zeitlich verschoben werden können und welche zeitlich gebunden sind und somit zu einem gegebenen Zeitpunkt entsprechend Energie verbrauchen müssen, angestrebt. Dem Mitglied eines Haushalts wird über einen Client ein Anreiz geboten, einen Verbraucher erst ab einem bestimmten Zeitpunkt oder innerhalb einer bestimmten Zeitspanne zu verwenden. Dieser Anreiz basiert auf Prognosen des Energiemarktes und des Energiepreises sowie Alter, Profil und Geschlecht des Haushaltsmitglieds zur Minimierung von Lastspitzen. Wird auf das Angebot eingegangen, erhält das Haushaltsmitglied eine entsprechende Belohnung, beispielsweise in monetärer Form. Aus technischer Sicht wurde im Rahmen von WATTALYST untersucht, welche Demand-Response Nachrichten zwischen dem zentralen Demand-Response System und den verschiedenen Clients ausgetauscht werden müssen. Dabei wurde insbesondere ein Datenmodell dieser Nachrichten entworfen. Dieses Datenmodell beinhaltet die Informationen über die entsprechenden Anreize, Belohnungen, zeitlichen Bedingungen und verschiedene Zustände, ob ein bestimmter Anreiz angenommen, abgelehnt oder erfüllt wurde. Darüber hinaus wurde in WATTALYST eine hochperformante Datenbank und deren Anbindung zur effizienten Persistierung von Sensordaten entwickelt.

Im Rahmen des Projekts wurde MontiEE dazu verwendet, das Demand-Response Datenmodell zu entwerfen und die einzelnen Demand-Response Signale in einer Datenbank zu speichern. Das Datenmodell wurde in Form eines Klassendiagramms als Domänenmodell des Servers modelliert. Tabelle 10.32 zeigt die Anzahl der handgeschriebenen und der generierten Dateien. Darüber hinaus ist die Anzahl der LOC sowie der prozentuale Anteil des generierten und des handgeschriebenen Quellcodes dargestellt. Die Daten wurden, wie zuvor beim Energie Navigator beschrieben, erhoben.

Zur Umsetzung des Domänenmodells wurden zwei Klassendiagramme mit insgesamt 128 Zeilen umgesetzt. Aus diesen Klassendiagrammen wurden die Entitäten, die DAOs und das SQL-Skript, wie in den Abschnitten 7.4, 7.5 und 7.6 dargestellt, mit MontiEE direkt generiert. Die DTOs und die Fassaden wurden zunächst handgeschrieben und dienten als Generatvorlage für die Entwicklung des DTO- und des Facade-Generators, die in den Abschnitten 8.2 und 8.4 gezeigt wurden. Da das Generat der Vorlage entspricht, wurde der nunmehr generierbare Code hier zugerechnet. Es zeigt sich, dass aus dem handgeschriebenen Modell 14478 Zeilen Quellcode und SQL-Skript erzeugt wurden. Dieser Quellcode wurde um handgeschriebene Funktionalität, die die Infrastruktur ergänzt, erweitert.

Der Anteil der handgeschriebenen Erweiterungen der Infrastruktur beträgt 6%. Es ist also ersichtlich, dass geringe manuelle Erweiterungen bereits ausreichen, um das Gene-

[2] http://cordis.europa.eu/project/rcn/100984_de.html

Tabelle 10.32: Überblick über die Anzahl der generierten Artefakte und der enthaltenen Quellcodezeilen (LOC) im Rahmen des WATTALYST Projekts.

	Dateien		LOC		Anteil	
	«gen»	«hwc»	«gen»	«hwc»	«gen»	«hwc»
Modell	0	2	0	128	0%	100%
Entitäten	163	17	7577	415	95%	5%
DAO	54	0	3235	0	100%	0%
SQL	2	0	292	0	100%	0%
DTO[a]	41	3	1354	184	88%	12%
Fassaden[a]	11	22	2020	570	78%	22%
Infrastruktur	271	42	14478	1169	93%	7%
Geschäftslogik	0	121	0	7142	0%	100%
Summe	271	163	14478	8311	64%	36%

[a]Der DTO- und der Fassaden-Generator sind aus dem Projekt WATTALYST heraus entstanden. Der aufgeführte Quellcode wurde handgeschrieben und als Generatvorlage für die Entwicklung des DTO- und des Fassaden-Generators verwendet. Aus diesem Grund wurde der nunmehr generierbare Code dem Generat zugezählt

rat der Enterprise Applikation anzupassen. Die Geschäftslogik, der eigentliche Kern der Enterprise Applikation, muss manuell implementiert werden. Dazu wurden im WATTALYST Projekt 121 Dateien mit 7142 LOC benötigt. MontiEE zielt darauf ab, die Infrastruktur zu einem Großteil zu generieren, damit der Entwickler sich auf die Umsetzung der Geschäftslogik konzentrieren kann. Tabelle 10.32 zeigt, dass die Geschäftslogik mit 7142 LOC einen Anteil von 31% an den gesamten 22789 LOC des WATTALYST Projekts einnimmt. Dies bedeutet, dass 69% des Quellcodes dazu verwendet werden, die Infrastruktur bereitzustellen. Durch MontiEE konnten 64% des gesamten Quellcodes generiert werden. Der benötigte Prozentsatz von Quellcode der Infrastruktur konnte somit reduziert werden. Zudem wurden aus 128 Zeilen des Modells 14478 Zeilen Quellcode generiert. Dies entspricht ungefähr dem Verhältnis 1:100. Der generierte Code und seine Technologieinformationen sind demnach also um den Faktor 100 größer als die Abstraktion durch das Modell.

Im Rahmen des WATTALYST Projekts half dies vor allem, neuen Entwicklern den Einstieg in den komplexen Technologiestack zu erleichtern. Durch die Generatoren konnten sie direkt beginnen, die Anwendung weiterzuentwickeln und mussten nicht zunächst die Technologie vollständig verstehen.

Die Fallstudien konnten zeigen, dass der Einsatz von MontiEE sowohl in bestehenden

Projekten als auch in Neuentwicklungen einen Mehrwert bietet. Dabei bringt die Integration in bestehende Systeme verschiedene Herausforderungen, die zur Nutzung des vollen Umfangs von MontiEE gemeistert werden müssen, mit. Daher ist der Nutzen von MontiEE bei Neuentwicklungen deutlich größer.

10.7 Zusammenfassung

In diesem Kapitel wurde die Methodik der modellgetriebenen Entwicklung von Enterprise Applikationen mit MontiEE vorgestellt und anhand des Szenarios demonstriert. Dazu wurde zunächst ein Aktivitätsdiagramm in Abbildung 10.1, das die zu treffenden Entscheidungen bei der Verwendung von MontiEE zeigt, präsentiert. Dabei wurde erläutert, wie die einzelnen Generatoren aktiviert oder deaktiviert werden können.

Daran anschließend wurde die Konfigurationsinfrastruktur von MontiEE gezeigt. Dabei wurde zwischen der Konfiguration verschiedener Parameter, der Konfiguration des Standardverhaltens der Generatoren und der Auswahl der einzelnen Generatoren unterschieden. Diese Konfiguration kann mit Hilfe eines Groovy-Skriptes oder per Kommandozeile sowie mit Hilfe eines Maven-Plugins erfolgen.

Danach wurde die Umsetzung des durchgängigen Szenarios mit MontiEE in Abschnitt 10.3 vorgestellt. Die zu modellierenden Elemente sowie die entstehenden Generate wurde gezeigt.

In Abschnitt 10.4 wurde das Deployment einer mit MontiEE generierten Applikation beschrieben. Dazu wurde zunächst der benötigte Aufbau eines EARs detailliert erläutert. Darauf aufbauend wurde gezeigt, in welchen JARs das Generat gebündelt wird und auf Basis welcher Informationen zusätzlich benötigte XML-Dateien ebenfalls generiert werden können.

In Abschnitt 10.5 wurde die vorhandene Werkzeuglandschaft, wie Editoren, Archetyps und Wizards, erläutert. Abschließend wurden in Abschnitt 10.6 Fallstudien in Form von Forschungsprojekten beschrieben, bei denen MontiEE zum Einsatz kam. Dabei wurde die Integration in Neuentwicklungen und in bestehende Systeme berücksichtigt.

Teil IV

Epilog

Kapitel 11

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde das Framework MontiEE und eine Methodik zur Anwendung vorgestellt. MontiEE beinhaltet Sprachen und Generatoren zur Unterstützung der modellgetriebenen, agilen Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen. Dazu wurde eine Sprachfamilie, die sieben Sprachen enthält, präsentiert.

In Kapitel 4 wurden Modellierungssprachen zur Modellierung der Persistenz eingeführt. Zunächst wurde die Verwendung von Klassendiagrammen als Domänenmodell der Enterprise Applikation vorgestellt. Danach wurde die Tagdefinitions- und die Tagschemasprache dargelegt. Dazu wurde zunächst eine allgemeine Methodik zur Ableitung von Taggingsprachen aus beliebigen DSLs präsentiert und auf Klassendiagramme angewendet. Die daraus resultierenden klassendiagrammspezifischen Taggingsprachen werden in MontiEE dazu verwendet, Klassendiagramme um technologiespezifische Informationen anzureichern. Das Tagschema definiert eine Menge möglicher Tagtypen. Die Tagdefinition taggt Elemente des Klassendiagramms mit diesen Tagtypen. Darüber hinaus wurden Kontextbedingungen zur statischen Analyse von Tagdefinition und Klassendiagrammen vorgestellt. Kapitel 7 stellt Generatoren, die Modelle dieser Sprachen verwenden, um die Persistenz und das Domänenmodell einer Enterprise Applikation zu generieren, vor. Dabei werden der Entity-, der DAO- und der SQL-Generator vorgestellt. Der Entity-Generator erzeugt die Entitäten der Enterprise Applikation, die vom ORM in der Datenbank persistiert werden. Die verwendeten Annotationen, die Abbildung von Vererbung und der Einbezug benutzerdefinierter Queries wurde präsentiert. Der DAO-Generator erzeugt DAOs zur Umsetzung des DAO-Patterns [KS06]. Dabei wurden typische Create, Read, Update, Delete (CRUD)-Operationen sowie Methoden zur Ausführung der benutzerdefinierten Queries dargelegt. Der SQL-Generator erzeugt ein SQL-Skript, dass die Datenbank initialisiert. Die Tabellen der Datenbank werden so generiert, dass sie der konfigurierten Abbildung der Vererbung entsprechen. Dies führt dazu, dass das Datenbankschema, die Entitäten und die DAOs, da sie aus den gleichen Eingabemodellen generiert werden, stets konsistent sind. Kapitel 4 und 7 zeigen, wie sich die Persistenz der Daten einer Enterprise Applikation modellieren lässt und wie daraus Quellcode generiert werden kann. Sie beantworten damit Forschungsfrage RQ2.

In Kapitel 5 wurden die Modellierungssprachen zur Modellierung der Kommunikation von Enterprise Applikationen präsentiert. Dabei wurden eine Sichten-, eine Rollen-, eine Rechte- und eine Mappingsprache gezeigt. Die Sichtensprache definiert Sichten auf ein Klassendiagramm und erlaubt so die Modellierung spezifischer Domänenmodelle für heterogene Clients. Dies ermöglicht es, eine reduzierte oder zusammengefasste Form des

Domänenmodells zu modellieren. Es wurde darauf geachtet, dass die Sicht immer konsistent zu dem Bezugsklassendiagramm ist. Zu diesem Zweck wurden Kontextbedingungen, die die Konsistenz sichern und eine statische Analyse ermöglichen, vorgestellt. Darüber hinaus wurde eine Transformation einer Sicht in ein syntaktisch valides, verkleinertes Klassendiagramm gezeigt, dass von Generatoren verarbeitet wird. Diese Generatoren, der DTO-, der BusinessAPI- und der Facade-Generator, wurden in Kapitel 8 vorgestellt. Sie dienen der Generierung der Kommunikationsinfrastruktur von Enterprise Applikationen. Neben den Generatoren wurde die Erzeugung einer allgemeinen Infrastruktur, damit diese technologiespezifische Informationen berücksichtigen können, erschaffen. Diese Infrastruktur kann aus beliebigen Tagschemas generiert und durch Tagdefinitionen befüllt werden. Im Rahmen dieser Arbeit wurde ein für MontiEE-spezifisches Tagschema präsentiert, das es ermöglicht, das Domänenmodell von Enterprise Applikationen um Informationen bezüglich des komplexen Technologiestacks anzureichern. Von den in Kapitel 8 vorgestellten Generatoren erzeugt der DTO-Generator auf Basis der Sichten DTOs, die den Clients als Domänenmodell dienen. Hierbei wurde neben der Generierung der DTOs auch die Generierung von Requests zur Umsetzung des Request Patterns [HW04b] sowie die Generierung der Assembler, die aus Entitäten DTOs erzeugen, vorgestellt. Der BusinessAPI-Generator erzeugt Schnittstellen von der Kommunikationsfassade zur Geschäftslogik und von der Geschäftslogik zur Persistenz. Dabei wurde die Integration der handgeschriebenen Geschäftslogik vorgestellt. Der Facade-Generator erstellt für spezifische Clients Kommunikationsfassaden, über die diese auf den Server zugreifen können. Der Entwickler kann durch das Generat von der Technologie abstrahieren und sich auf die Entwicklung der Geschäftslogik fokussieren. Er benötigt kein Detailwissen der Technologie. Kapitel 5 und 8 beantworten die Forschungsfragen RQ3 und RQ4. Zur Modellierung verschiedener Benutzer mit unterschiedlichen Befugnissen werden Rollen-, Rechte- und Mappingdiagramme verwendet. Mit Hilfe des Facade- und des BusinessAPI-Generators können Kommunikationsfassaden und Schnittstellen, die spezifisch für die jeweiligen Nutzer sind und ihre Berechtigungen prüfen, generiert werden. Zur Modellierung unterschiedlicher Arten von Clients wird die Sichtensprache verwendet. Aus modellierten Sichten können mit Hilfe des DTO- und des Facade-Generators clientspezifische Domänenmodelle und Zugriffsfassaden generiert werden.

In Kapitel 6 wurde eine klassendiagrammspezifische Deltasprache vorgestellt, die die Evolution des Systems auf Basis von Klassendiagrammen modelliert. Die Sprache enthält Basis- und weiterführende Operationen zur Modellierung. Auf Basis dieser Modelle wurde in Kapitel 9 ein Generator vorgestellt, der die Infrastruktur zur Evolution des Systems und zur Datenmigration generiert. Dabei wurde die Evolution des Domänenmodells sowie die Migration persistenter Daten in Form von Objektdiagrammen detailliert erläutert. Zur Migration wurden Initialisierer vorgestellt, die die Initialisierung migrierter Attribute durchführen. Kapitel 6 und 9 beantworten die Forschungsfrage RQ5. Die Evolution der Applikation wird mit Deltas modelliert und die Daten durch die generierte Infrastruktur migriert.

In Kapitel 10 wurde eine Methodik zur modellgetriebenen Entwicklung mit MontiEE vorgestellt und anhand einer Fallstudie demonstriert. Dazu wurde die Funktionsweise der

Generatoren auf Basis eines Aktivitätsdiagramms dargelegt. Daran anschließend wurden Konfigurations- und Verwendungsmöglichkeiten von MontiEE präsentiert. Dabei wurde gezeigt wie MontiEE im Entwicklungsprozess verwendet wird. Danach wurden das Deployment des Generats und vorhandene Werkzeuge zur Verwendung von MontiEE erläutert. Darüber hinaus wurde MontiEE auf das in Kapitel 3 vorgestellte Szenario angewendet. Die Modellierung der Enterprise Applikation sowie das Generat wurde gezeigt. Abschließend wurde MontiEE auf Basis unterschiedlicher Fallstudien im Rahmen von Forschungsprojekten evaluiert.

Nachdem die Forschungsfragen RQ2 - RQ5 im Rahmen der zuvor zusammengefassten Kapitel beantwortet wurden, kann daraus RQ1 beantwortet werden. Die Modelle der Sprachen der Sprachfamilie von MontiEE können zur Abstraktion des komplexen Technologiestacks verwendet werden. Zur Erhaltung des Abstraktionsgrads werden Informationen mit Hilfe von Tagdefinitionen, so dass die Modelle nicht belastet werden, externalisiert. Zusammenfassend lässt sich festhalten, dass die modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit Hilfe von MontiEE durch Sprachen und Generatoren unterstützt wird.

MontiEE stellt einen guten Ausgangspunkt für die Modellierung von Enterprise Applikationen dar. Dennoch existieren auch im Rahmen von MontiEE eine Reihe offener Punkte, die in nachfolgenden Arbeiten zur Erweiterung von MontiEE erforscht werden können.

Erweiterung der Umsetzung der Geschäftslogik

Momentan bietet MontiEE die Möglichkeit Geschäftslogik handgeschrieben zu integrieren. Dazu muss sich die Geschäftslogik an einer generierten Schnittstelle orientieren. Als Erweiterung ist ein Ausbau auf eine Modellierung der Geschäftslogik denkbar. Dazu müssen geeignete Modelle, wie Sequenz- oder Aktivitätsdiagramme aber auch BPMN Modelle auf Ihre Eignung zur Integration und zur Codegenerierung im MontiEE Umfeld untersucht werden. Daran anschließend müssen Codegeneratoren umgesetzt werden, die diese Modelle in technologiespezifischen Quellcode transformieren.

Erweiterung der Persistenz

In seiner momentanen Form verwendet MontiEE die JPA zur Generierung der persistenzspezifischen Annotationen, Hibernate als ORM sowie PostgreSQL als Datenbank. Die Verwendung der konkreten relationalen Datenbank wird dabei durch den ORM abstrahiert. So unterstützt Hibernate verschiedene Datenbanken, wie PostgreSQL oder Oracle10g oder MySQL [RW12]. Allerdings muss dies als Dialekt in der `persistence.xml` konfiguriert werden. Hier bietet MontiEE momentan keine Variabilität an.

Darüber hinaus bietet MontiEE zur Zeit keine unterschiedlichen Datenbankparadigmen neben einer relationalen Datenbank an. Zur effizienten Speicherung von unstrukturierten Daten sollte eine Unterstützung für NoSQL Datenbanken geschaffen werden. Dies würde die Erstellung eines neuen Generators, der analog zum SQL-Generator ein Skript, dass die Datenbank initialisiert, erzeugt, bedingen. Gleichzeitig müsste auch die

Abbildung der Entitäten auf die Konzepte der Datenbank, wie sie momentan vom ORM übernommen wird, verändert werden. Dies würde in einer Erweiterung des Entity-Generators resultieren. Darüber hinaus werden neue Tagtypen, die angeben, welche Teilgraphen des Klassendiagramms in welcher Datenbank gespeichert werden müssen, benötigt. Für beide Erweiterungen wird ein zusätzliches Modell, das die zur Verfügung stehende Infrastruktur sowie das Deployment modelliert benötigt. In [HNPR13] wird solch ein Modell vorgestellt. Dies kann zur Bereitstellung der benötigten Informationen verwendet werden.

Erweiterung der Mehrbenutzerfähigkeit

Neben den zuvor genannten Erweiterungen der Persistenz kann auch die Modellierung der Rollen und der Rechte erweitert werden. Momentan ist es nicht möglich, instanzbasierte Rechte zu modellieren. Die Rechtesprache sieht dazu bereits einen Erweiterungspunkt, in dem ihre Syntax an die OCL angelehnt ist, vor. Allerdings muss nicht nur die Modellierung der Rechte, sondern auch der Facade-Generator, der die Zugriffskontrollmechanismen des Applikationsservers verwendet, erweitert werden. Dies erfordert, dass innerhalb der Kommunikationsfassade nicht nur der Methodenzugriff, sondern auch der Instanzzugriff innerhalb der Methode, geprüft wird. Darüber hinaus kann MontiEE von einer Mehrbenutzerfähigkeit zu einer Multimandantenfähigkeit erweitert werden. Dazu muss die Rollenmodellierung erweitert werden, so dass die Modelle Aussagen über konkrete Nutzer des Systems treffen können. Da sich diese typischerweise zur Laufzeit des Systems ändern und nicht zur Entwicklungszeit bekannt sind, muss dieses Modell zur Laufzeit der Enterprise Applikation verwendbar sein.

Erweiterung der Kommunikation

Neben der Erweiterung der Persistenz und der Mehrbenutzerfähigkeit kann auch die Kommunikation erweitert werden. MontiEE verwendet Webservices als offene Kommunikationsschnittstelle, die sich etabliert hat. Zum Informationstransfer verwendet MontiEE JSON oder XML. Eine Erweiterung zur Integration von Protocol Buffers ist denkbar. Dazu müssen die generierten Kommunikationsfassaden so erweitert werden, dass sie die von Protocol Buffers generierten Serialisierer und Deserialisierer verwenden. Diese werden aus einem Protocol Buffer Schema, welches eine Beschreibung der zu kommunizierenden Daten ist, generiert. Es lässt sich aus dem mit MontiEE modellierten Domänenmodell ableiten. Dadurch kann das Protocol Buffers Schema automatisch erstellt und zur Generierung der Serialisierer und Deserialisierer verwendet werden. Gleichzeitig kann ein Protocol Buffer Schema Informationen über eine Rückwärtskompatibilität eines Datenmodells enthalten. Auch diese Informationen können aus den Deltas abgeleitet werden. MontiEE lässt sich mit geringen Anpassungen auf die Verwendung von Protocol Buffers erweitern.

Replikation einzelner Teilkomponenten

In [HNPR13, NPR13] wird eine Modellierungssprache zur Modellierung von Cloud-Architekturen und ihrem Deployment beschrieben. Diese ermöglicht die Modellierung ver-

schiedener Komponenten eines Applikationsservers und der Kommunikation dieser. Zur Erweiterung können mit Hilfe der Modellierungssprache aus [HNPR13, NPR13] Komponenten, wie Kommunikationsfassade, Manager, Geschäftslogik oder DAOs, deren Implementierung von MontiEE generiert wird, modelliert werden. Innerhalb der Modellierungssprache kann Replikation ausgedrückt werden, so dass diese modellierten Komponenten skalieren können. Dazu müssen die Generatoren, die die Kommunikationsinfrastruktur der Enterprise Applikation generieren, erweitert werden. Der generierte Quellcode muss Komponenten und deren Kommunikation nach dem in [HNPR13, NPR13] beschriebenen Modell unterstützen.

Erweiterung zur Unterstützung von Microservices

Eine weitere Erweiterung ist die Veränderung der Client-Server Architektur, die zur Zeit von MontiEE angenommen wird, hin zu einer Microservice [New15] oder DevOps [Has15] Architektur. Auch wenn beide Architekturen unterschiedliche Dinge darstellen, so ergänzen sie sich im Wesentlichen [Has16]. Der DevOps Ansatz stellt Teamstrukturen in den Vordergrund und erfordert ein hohes Maß an Automatisierung bis hin zu Continuous Delivery [HF10]. Der Microservice Architekturansatz hingegen bezieht sich auf den Schnitt einzelner Funktionalitäten. Microservices folgen dem Paradigma, dass es für unterschiedliche Teilbereiche, so genannte *Bounded Contexts* [Eva04, New15], einen einzelnen Service gibt, der Kommunikationsfassaden, Logik und Persistenz für diesen Teilbereich enthält. Darüber hinaus sind Microservice Architekturen elastisch skalierbar und erlauben automatisierten starten und stoppen weiterer Instanzen eines Services. Dadurch können Lastspitzen gehandhabt werden. Diese Vollautomatisierung stellt hohe Anforderungen an das Monitoring solcher Infrastrukturen. Dabei muss das Monitoring die gleiche Elastizität besitzen [FWWH13, FRH15, FKH16]. Da die MontiEE-Generatoren bereits modular geschnitten sind, kann dazu ein Großteil dieser zur Erstellung einzelner Microservices wiederverwendet werden. Allerdings unterliegt MontiEE der Prämisse, dass es ein gemeinsames Domänenmodell der Enterprise Applikation gibt. Dies ist ein Unterschied zu Microservices, wo jeder Service ein eigenes Domänenmodell besitzt, welche in Teilen überlappen. In MontiEE wurde eine Sichtensprache, die Teilmodelle eines gemeinsamen Domänenmodells modelliert, verwendet. Diese Sichten können somit das Domänenmodell eines einzelnen Microservice modellieren. Da eine Sicht in ein Klassendiagramm transformiert werden kann, können die MontiEE-Generatoren diese als Eingaben verwenden. Dies ermöglicht die Generierung einzelner persistierbarer Entitäten, die jeweils die Persistenz eines Microservices darstellen. Offen ist, dass verschiedene Microservices unterschiedliche Domänenmodelle, die aber gleiche Konzepte unter anderem Namen beinhalten, besitzen können. Dazu werden Abbildungen zwischen zwei Microservices definiert. Dies ist momentan nicht in MontiEE enthalten und müsste erweitert werden. Eine Microservice Architektur wird im Zeitalter der Digitalisierung, da die anfallenden Daten und die benötigte Funktionalität stetig steigen, häufig angestrebt. Microservices bieten eine Form der Modularisierung an, die eine unabhängige Entwicklung und Wartung verspricht.

Literaturverzeichnis

[ABC+14] Lorenzo Anardu, Roberto Baldi, Umberto Antonio Cicero, Riccardo Giomi und Giacomo Veneri. *Maven Build Customization*. Packt Publishing, 2014.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno und Vijay Machiraju. *Web Services*. Springer, 2004.

[AKS03] Aditya Agrawal, Gabor Karsai und Feng Shi. Graph Transformations on Domain-Specific Models. *Software and Systems Modeling*, 37:1–43, 2003.

[ALPS11] Anthony Anjorin, Marius Lauder, Sven Patzina und Andy Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *Informatik 2011 - Informatik schafft Communities*, Seite 281, 2011.

[Alt12] Oliver Alt. *Modellbasierte Systementwicklung mit SysML*. Carl Hanser Verlag, Deutschland, 2012.

[Avr16] Apache Avro. `https://avro.apache.org/`, 2016. [Online; abgerufen am: 27.03.2016].

[Bal10] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Springer, Deutschland, 2010.

[Bas15] Lindsay Bassett. *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*. O'Reilly, August 2015.

[BBB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries et al. Agile Manifesto. `http://agilemanifesto.org/`, 2001. [Online; abgerufen am: 13.11.2015].

[BBC01] Herman Balsters, Bert. O. de Brock und Stefan Conrad, Editoren. *Database Schema Evolution and Meta-Modeling, 9th International Workshop on Foundations of Models and Languages for Data and Objects, Selected Papers*, LNCS 2065. Springer, 2001.

[BCFC06] Alessandro Bozzon, Sara Comai, Piero Fraternali und Giovanni Toffetti Carughi. Conceptual Modeling and Code Generation for Rich Internet Applications. In *Proceedings of the 6th International Conference on Web Engineering*, Seiten 353–360. ACM, 2006.

[BCFM00] Aldo Bongio, Stefano Ceri, Piero Fraternali und Andrea Maurino. Modeling Data Entry and Operations in WebML. In *The World Wide Web and Databases*, Seiten 201–214. Springer, 2000.

[BCFM08] Marco Brambilla, Sara Comai, Piero Fraternali und Maristella Matera. Designing Web Applications with WebML and WebRatio. In *Web Engineering: Modelling and Implementing Web Applications*, Seiten 221–261. Springer, 2008.

[BCPV07] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco und Joost Visser. Coupled Schema Transformation and Data Conversion for XML and SQL. In *Practical Aspects of Declarative Languages*, LNCS 4354, Seiten 290–304. Springer, 2007.

[BCW12] Marco Brambilla, Jordi Cabot und Manuel Wimmer. Model-driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[BDH+15] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi und Dániel Varró. VIATRA 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, Seiten 101–110. Springer, 2015.

[BG10] Erik Burger und Boris Gruschko. A Change Metamodel for the Evolution of MOF-Based Metamodels. In Gregor Engels, Dimitris Karagiannis und Heinrich C. Mayr, Editoren, *Modellierung*, LNI 161, Seiten 285–300. GI, 2010.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe und Wolfgang Schwerin. Systems, Views and Models of UML. In M. Schader und A. Korthaus, Editoren, *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, Seiten 93–109. Physica Verlag, 1998.

[BGP01] Luciano Baresi, Franca Garzotto und Paolo Paolini. Extending UML for Modeling Web Applications. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, Jan 2001.

[BGWK14] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer und Gerti Kappel. JUMP—From Java Annotations to UML Profiles. In *Model-Driven Engineering Languages and Systems*, LNCS 8767, Seiten 552–568. Springer, 2014.

[Bib77] Kenneth Biba. Integrity Considerations for Secure Computer Systems. Technical report, Defense Technical Information Center Document, 1977.

[BL73] D. Elliott Bell und Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, Defense Technical Information Center Document, 1973.

[Ble14] Alex Blewitt. *Mastering Eclipse Plug-in Development*. Packt Publishing, 2014.

[Blo08] Joshua Bloch. *Effective Java (the Java Series)*. Prentice Hall PTR, 2nd Edition, 2008.

[BN89] David Brewer und Michael Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, Seiten 206–214. IEEE, 1989.

[Bre12] Eric Brewer. Pushing the CAP: Strategies for Consistency and Availability. *Computer*, 45(2):23–29, 2012.

[Bur09] Bill Burke. *RESTful Java with JaX-RS*. O'Reilly, 2009.

[BW07] Thomas Baar und Jon Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI)*, 2007.

[CE00] Krzysztof Czarnecki und Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CFB00] Stefano Ceri, Piero Fraternali und Aldo Bongio. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks*, 33(1):137–157, 2000.

[CH03] Krzysztof Czarnecki und Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the Workshop on Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003.

[CH06] Krzysztof Czarnecki und Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[CHKT05] Stefan Conrad, Wilhelm Hasselbring, Arne Koschel und Roland Tritsch. *Enterprise Application Integration*. Spektrum Akademischer Verlag, 2005.

[Cöm13] Mehmet Can Cömert. Generating Access Control for the Business Tier in Enterprise Information Systems. Masterarbeit, RWTH University, Software Engineering, Germany, 2013.

[COV06] Alcino Cunha, José Nuno Oliveira und Joost Visser. Type-safe Two-level Data Transformation. In *Formal Methods (FM'06)*, Seiten 284–299. Springer, 2006.

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo und Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, Seiten 222–231, Washington, DC, 2008. IEEE Computer Society.

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DRRS09] Michael Dukaczewski, Dirk Reiss, Bernhard Rumpe und Mark Stein. MontiWeb – Modular Development of Web Information Systems. In Matti Rossi, Jonathan Sprinkle, Jeff Gray und Juha-Pekka Tolvanen, Editoren, *Workshop on Domain-Specific Modeling (DSM '09)*. HSE Print, Orlando, Florida, 2009.

[DSM15] DSM Forum. `http://www.dsmforum.org/cases.html`, 2015. [Online; abgerufen am: 11.10.2015].

[DW05a] Dirk Draheim und Gerald Weber. *Form-oriented Analysis: A New Methodology to Model Form-based Applications*. Springer Science and Business Media, 2005.

[DW05b] Dirk Draheim und Gerald Weber. Modelling Form-based Interfaces with Bipartite State Machines. *Interacting with Computers*, 17(2):207–228, 2005.

[EB10] Moritz Eysholdt und Heiko Behrens. XText: Implement your Language Faster than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, Seiten 307–309. ACM, 2010.

[Eck13] Claudia Eckert. *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. Oldenbourg Verlag, 2013.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North und Gordon Woodhull. Graphviz — Open Source Graph Drawing Tools. In *Graph Drawing*, Seiten 483–484. Springer, 2002.

[ESV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh et al. The State of the Art in Language Workbenches. In *Software Language Engineering*, Seiten 197–217. Springer, 2013.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[FBSG07] Madeleine Faugere, Thimothée Bourbeau, Robert De Simone und Sebastien Gerard. MARTE: Also an UML Profile for Modeling AADL Applications. In *12th IEEE International Conference on Engineering Complex Computer Systems*, Seiten 359–364, July 2007.

[FCK95] David Ferraiolo, Janet Cugini und D. Richard Kuhn. Role-Based Access Control (RBAC): Features and Motivations. In *Proceedings of 11th Computer Security Application Conference*, Seiten 241–48, 1995.

[FG98] Mark Stephen Fox und Michael Gruninger. Enterprise Modeling. *AI magazine*, 19(3):109, 1998.

[FKC92] David Ferraiolo, Richard Kuhn und Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, 1992.

[FKH16] Florian Fittkau, Alexander Krause und Wilhelm Hasselbring. Software Landscape and Application Visualization for System Comprehension with ExplorViz. *Information and Software Technology*, 2016.

[FLP+11a] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser und Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36–41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser und Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Proceedings of the 11th International Conference for Enhanced Building Operations (ICEBO'11)*, New York, NY, USA, October 2011.

[FMS14] Sanford Friedenthal, Alan Moore und Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2014.

[For13] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing, 2013.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[FPR00] Marcus Fontoura, Wolfgang Pree und Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Fra00] Timothy Fraser. LOMAC: Low Water-mark Integrity Protection for COTS Environments. In *IEEE Symposium on Security and Privacy*, Seiten 230–245, 2000.

[Fre13] Freemarker. `http://freemarker.org/`, 2013. [Online; abgerufen am: 16.01.2016].

[FRH15] Florian Fittkau, Sascha Roth und Wilhelm Hasselbring. ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes. AIS, 2015.

[FSG+01] David Ferraiolo, Ravi Sandhu, Serban Gavrila, Richard Kuhn und Ramaswamy Chandramouli. Proposed NIST Standard for Role-based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[FWWH13] Florian Fittkau, Jan Waller, Christian Wulf und Wilhelm Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Seiten 1–4. IEEE, 2013.

[GAP+02] Simon Godik, Anne Anderson, Bill Parducci, Polar Humenn und Sekhar Vajjhala. eXtensible Access Control 2 Markup Language (XACML) 3. Technical report, OASIS, 2002.

[GC03] Joseph Gradecki und Jim Cole. *Mastering Apache Velocity*. Wiley, 2003.

[GDB02] Timothy Grose, Gary Doney und Stephen Brodsky. *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley and Sons, 2002.

[GDW09] Alexander Grosskopf, Gero Decker und Mathias Weske. *The Process: Business Process Modeling using BPMN*. Meghan Kiffer Press, 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler und Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray und Joaquim Filipe Filipe, Editoren, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Seiten 74–85, Angers, Loire Valley, France, February 2015. SciTePress.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin

Schindler und Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, CCIS 580, Seiten 112–132. Springer, 2015.

[GHKV08] Danny M. Groenewegen, Zef Hemel, Lennart Kats und Eelco Visser. WebDSL: A Domain-specific Language for Dynamic Web Applications. In *Companion to the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, Seiten 779–780. ACM, 2008.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler und Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, TU Braunschweig, Deutschland, August 2006.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler und Steven Völkel. MontiCore: a Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, Seiten 925–926, 2008.

[Gli07] Martin Glinz. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference*, Seiten 21–26, 2007.

[Gli08] Martin Glinz. A Risk-based, Value-oriented Approach to Quality Requirements. *IEEE Software*, 25(2):34–41, 2008.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell und Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, chapter 56, Seiten 511–520. Springer Berlin Heidelberg, April 2015.

[GLRK14] Timo Greifenberg, Markus Look, Bernhard Rumpe und Ellis A. Keith. Integrating Heterogeneous Building and Periphery Data Models at the District Level: The NIM Approach. In *Proceedings of the 10th European Conference on Product and Process Modelling (ECPPM 2014)*, ECPPM 2014 - eWork and eBusiness in Architecture, Engineering and Construction, Seiten 821–828, Vienna, Austria, September 2014. CRC Press Balkema, Leiden, Netherlands.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl und Bernhard Rumpe. Engineering Tagging Languages for DSLs. In Timothy Lethbridge, Jordi Cabot und Alexander Egyed, Editoren, *Conference on Model Driven*

Engineering Languages and Systems (MODELS), Seiten 34–43, Ottawa, Canada, 2015. ACM New York/IEEE Computer Society.

[GMP09] Roy Grønmo und Birger Møller-Pedersen. Concrete Syntax-Based Graph Transformation, 2009. Research Report 389.

[GMPO09] Roy Grønmo, Birger Møller-Pedersen und GøranK. Olsen. Comparison of Three Model Transformation Languages. In *Model Driven Architecture - Foundations and Applications*, LNCS 5562, Seiten 2–17. Springer, 2009.

[Gra72] Carl Graumann. *Interaktion und Kommunikation*. Verlag für Psychologie, 1972.

[GSO16] GSON. `https://github.com/google/gson`, 2016. [Online; abgerufen am: 27.03.2016].

[GT02] David Gourley und Brian Totty. *HTTP: The Definitive Guide*. O'Reilly, 2002.

[GV08] Danny Groenewegen und Eelco Visser. Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns. In Daniel Schwabe und Francisco Curbera, Editoren, *International Conference on Web Engineering (ICWE'08)*, LNCS. IEEE, 2008.

[Han14] Matthias Hannen. Integration heterogener Datenmodelle unter Verwendung von Plugin-Generierung. Masterarbeit, RWTH University, Software Engineering, Germany, 2014.

[Has00] Wilhelm Hasselbring. Information System Integration. *Communications of the ACM*, 43(6):32–38, 2000.

[Has15] Wilhelm Hasselbring. DevOps: Softwarearchitektur an der Schnittstelle zwischen Entwicklung und Betrieb. 2015.

[Has16] Wilhelm Hasselbring. Microservices for Scalability. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, Seiten 133–134. ACM, 2016.

[HBFV03] Geert-Jan Houben, Peter Barna, Flavius Frasincar und Richard Vdovjak. Hera: Development of Semantic Web Information Systems. In *Web Engineering*, Seiten 529–538. Springer, 2003.

[Hef10] David Heffelfinger. *Java EE 6 with GlassFish 3 Application Server*. Packt Publishing, 2010.

[Her11] Markus Herrmannsdoerfer. COPE - A Workbench for the Coupled Evolution of Metamodels and Models. *Software Language Engineering*, Seiten 286–295, 2011.

[Her14] Christoph Herrmann. *Integrierte Software Engineering Services zur effizienten Unterstützung von Entwicklungsprojekten.* Aachener Informatik-Berichte, Software Engineering Band 16. Shaker Verlag, Aachen, Deutschland, 2014.

[HF10] Jez Humble und David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe und Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, Seiten 22–31, Tokyo, Japan, September 2013. ACM.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer und Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe und Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp)*, CEUR Workshop Proceedings 1463, Seiten 18–23, Ottawa, Canada, September 2015.

[Hib16] Hibernate. `http://hibernate.org/`, 2016. [Online; abgerufen am: 27.03.2016].

[HJGP99] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec und François Pennaneac'h. UMLAUT: An Extendible UML Transformation Framework. In *14th IEEE International Conference on Automated Software Engineering*, Seiten 275–278, 1999.

[HKGV10] Zef Hemel, Lennart Kats, Danny Groenewegen und Eelco Visser. Code Generation by Model Transformation. A Case Study in Transformation Modularity. *Software and Systems Modeling*, 9(3):375–402, 2010.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *ECSA '11 5th European Conference on Software Architecture: Companion Volume*, New York, NY, USA, September 2011. ACM New York.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel und Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, CCOS 580, Seiten 45–66. Springer, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel und Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray und Joaquim Filipe Filipe, Editoren, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Seiten 19–31, Angers, France, February 2015. SciTePress.

[HM15] Steffen Heinzl und Markus Mathes. *Middleware in Java: Leitfaden zum Entwurf verteilter Anwendungen—Implementierung von verteilten Systemen über JMS—Verteilte Objekte über RMI und CORBA*. Springer, 2015.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe und Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type-Safe Visitors. In *European Conference on Modelling Foundations and Applications (ECMFA'16) (to appear)*, 2016.

[HNPR13] Lars Hermerschmidt, Antonio Navarro Perez und Bernhard Rumpe. A Model-based Software Development Kit for the SensorCloud Platform. In *Workshop Wissenschaftliche Ergebnisse der Trusted Cloud Initiative*, Seiten 125–140. Springer, Schweiz, 2013.

[Hol08] Anthony T. Holdener. *AJAX: The Definitive Guide*. O'Reilly, 2008.

[HR83] Theo Haerder und Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[HR04] David Harel und Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, Seiten 1 – 10, Munich, Germany, February 2011. fortiss GmbH.

[HRW11] John Hutchinson, Mark Rouncefield und Jon Whittle. Model-driven Engineering Practices in Industry. In *33rd International Conference on Software Engineering (ICSE)*, Seiten 633–642, May 2011.

[HRW15] Katrin Hölldobler, Bernhard Rumpe und Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In Timothy Lethbridge, Jordi Cabot und Alexander Egyed, Editoren, *Conference on Model Driven Engineering Languages and Systems (MODELS)*, Seiten 136–145, Ottawa, Canada, 2015. ACM New York/IEEE Computer Society.

[HSB+08] Gert-Jan Houben, Kees van der Sluijs, Peter Barna, Jeen Broekstr, Sven Casteleyn, Zoltán Fiala und Flavius Frascinar. Hera. In *Web Engineering: Modelling and Implementing Web Applications*, Seiten 263–302. Springer, 2008.

[HVV08] Zef Hemel, Ruben Verhaaf und Eelco Visser. WebWorkFlow: An Object-Oriented Workflow Modeling. In Krzysztof Czarnecki, Editor, *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, LNCS. Springer, 2008.

[HW04a] Robert Harris und Robert Warner. *The Definitive Guide to SWT and JFace*. Apress, 2004.

[HW04b] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield und Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Seiten 471–480. ACM, 2011.

[ISO01] International Standard Organization. Software Engineering - Product Quality, ISO/IEC 9126-1, 2001.

[ISO03a] International Standard Organization. Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework) ISO/IEC 9075-1:2003, 2003.

[ISO03b] International Standard Organization. Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation) ISO/IEC 9075-2:2003, 2003.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev und Patrick Valduriez. ATL: A QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, Seiten 719–720, New York, NY, 2006. ACM.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin und Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1):31–39, 2008.

[Jac16] Jackson JSON Processor. `http://wiki.fasterxml.com/JacksonHome`, 2016. [Online; abgerufen am: 27.03.2016].

[Jen16] Jenkins. `https://jenkins.io/`, 2016. [Online; abgerufen am: 21.04.2016].

[JK06] Frédéric Jouault und Ivan Kurtev. Transforming Models With ATL. In *Satellite Events at the MoDELS 2005 Conference*, Seiten 128–138. Springer, 2006.

[JPA16] JPA Spezifikation. `http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`, 2016. [Online; abgerufen am: 27.03.2016].

[Kü05] Thomas Kühne. What is a Model? In *Dagstuhl Seminar Proceedings*, 2005.

[Kaj12] Ejub Kajan. *Information Technology Encyclopedia and Acronyms*. Springer Science and Business Media, 2012.

[Kay00] Michael Kay. *XSLT Programmer's Reference*. Wrox Press, Birmingham, UK, 2000.

[KE11] Alfons Kemper und André Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 2011.

[KKCM04] Nora Koch, Andreas Kraus, Cristina Cachero und Santiago Meliá. Integration of Business Process in Web Application Models. *Journal of Web Engineering*, 3(1):22–49, 2004.

[KKK09] Christian Kroiss, Nora Koch und Alexander Knapp. UWE4JSF: A Model-Driven Generation Approach for Web Applications. In *ICWE*, Seiten 493–496. Springer, 2009.

[KKL+15] Dierk Koenig, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cedric Champeu, Erik Pragt und Jon Skeet. *Groovy in Action*. Manning, 2015.

[KKMZ03] Alexander Knapp, Nora Koch, Flavia Moser und Gefei Zhang. ArgoUWE: A CASE Tool for Web Applications. In *International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE)*, 2003.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler und Steven Völkel. Design Guidelines for Domain Specific Languages. In Matteo Rossi, Jonathan Sprinkle, Jeff Gray und Juha-Pekka Tolvanen, Editoren, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, Techreport B-108, Seiten 7–13, Orlando, Florida, October 2009. Helsinki School of Economics.

[KKZB08] Nora Koch, Alexander Knapp, Gefei Zhang und Hubert Baumeister. UML-based Web Engineering. In *Web Engineering: Modelling and Implementing Web Applications*, Seiten 157–191. Springer, 2008.

[KLM+16] Carsten Kolassa, Markus Look, Klaus Müller, Alexander Roth, Dirk Reiß und Bernhard Rumpe. TUnit - Unit Testing For Template-based Code Generators. In *Modellierung 2016 Conference*, LNI 254, Seiten 221–236, 2016.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell und Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Proceedings of the Modelling of the Physical World Workshop (MOTPW'12)*, Seiten 2:1–2:6, Innsbruck, Austria, October 2012. ACM Digital Library.

[KLR96] Steven Kelly, Kalle Lyytinen und Matti Rossi. MetaEdit+ a Fully Configurable Multi-user and Multi-tool CASE and CAME Environment. In *Advanced Information Systems Engineering*, Seiten 1–21. Springer, 1996.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige und Fiona A.C. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations and Applications*, Seiten 128–142. Springer, 2006.

[KR05] Vinay Kulkarni und Sreedhar Reddy. Model-driven Development of Enterprise Applications. In *UML Modeling Languages and Applications*, Seiten 118–128. Springer, 2005.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Deutschland, März 2010.

[KS06] Mike Keith und Merrick Schincariol. *Pro EJB 3: Java Persistence API*. Apress Series. Apress, 2006.

[Kur08] Ivan Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *Applications of Graph Transformations with Industrial Relevance*, Seiten 377–393. Springer, 2008.

[KV10] Lennart Kats und Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *ACM Sigplan Notices*, Seiten 444–463. ACM, 2010.

[KWB03] Anneke Kleppe, Jos Warmer und Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[Lan08] Fabian Lange. *Eclipse Rich Ajax Platform: Bringing Rich Client to the Web*. Apress, 2008.

[LBD02] Torsten Lodderstedt, David Basin und Jürgen Doser. SecureUML: A UML-based modeling language for model-driven security. In *UML 2002—The Unified Modeling Language*, Seiten 426–441. Springer, 2002.

[LGOT09] Andrea De Lucia, Carmine Gravino, Rocco Oliveto und Genoveffa Tortora. An Experimental Comparison of ER and UML Class Diagrams for Data Modelling. *Empirical Software Engineering*, 15(5):455–492, 2009.

[LL01] Ralf Lämmel und Wolfgang Lohmann. Format Evolution. *Proceedings of the 7th International Conference on Reverse Engineering for Information Systems (RETIS)*, 155:113–134, 2001.

[LNPR+13] Markus Look, Antonio Navarro Pérez, Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In B. Combemale, J. De Antoni und R. B. France, Editoren, *Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC)*, CEUR Workshop Proceedings 1102, Miami, Florida, USA, 2013.

[Mac06] Lori MacVittie. *XAML in a Nutshell*. O'Reilly, 2006.

[Mae12] Kazuaki Maeda. Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats. In *Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, Seiten 177–182. IEEE, 2012.

[Mar11] Jan Marten. Development of a Framework for Generating Persistent Data Models. Masterarbeit, RWTH University, Software Engineering, Germany, 2011.

[MBC+05] Ioana Manolescu, Marco Brambilla, Stefano Ceri, Sara Comai und Piero Fraternali. Model-driven Design and Deployment of Service-enabled Web Applications. *ACM Transactions on Internet Technology (TOIT)*, 5(3):439–479, 2005.

[MCG05] Tom Mens, Krzysztof Czarnecki und Pieter Van Gorp. A Taxonomy of Model Transformations. In *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[MD05] Tom Marrs und Scott Davis. *JBoss at Work: A Practical Guide: A Practical Guide*. O'Reilly, 2005.

[MHS05] Marjan Mernik, Jan Heering und Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[MKK05] Santiago Meliá, Andreas Kraus und Nora Koch. MDA Transformations Applied to Web Application Development. In *Web Engineering*, Seiten 465–471. Springer, 2005.

[ML05] Dave Minter und Jeff Linwood. *Pro Hibernate 3*. Apress, 2005.

[MLA10] Jeff McAffer, Jean-Michel Lemieux und Chris Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2010.

[MLG+10] Jürgen Muller, Martin Lorenz, Felix Geller, Alexander Zeier und Hasso Plattner. Assessment of Communication Protocols in the EPC Network-Replacing Textual SOAP and XML with Binary Google Protocol Buffers Encoding. In *17Th International Conference on Industrial Engineering and Engineering Management (IE&EM)*, Seiten 404–409, 2010.

[Mol06] Anthony Molinaro. *SQL Cookbook*. O'Reilly, 2006.

[MRD12] Waqar Aziz Muhammad, Mohamad Radziah und N.A. Jawawi Dayang. SOA4DERTS: A Service-Oriented UML profile for Distributed Embedded Real-Time Systems. In *IEEE Symposium on Computers Informatics (ISCI)*, Seiten 64–69, March 2012.

[MRR13] Shahar Maoz, Jan Oliver Ringert und Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Seiten 444–454. ACM, 2013.

[MRV08] Nathalie Moreno, José Raúl Romero und Antonio Vallecillo. An Overview of Model-driven Web Engineering and the MDA. In *Web Engineering: Modelling and Implementing Web Applications*, Seiten 353–382. Springer, 2008.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédéric Fondement und Jean Bézivin. Platform Independent Web Application Modeling and Development with Netsilon. *Software and Systems Modeling*, 4(4):424–442, 2005.

[Mül12] Kevin Müller. Design and Implementation of an SQL Grammar for the MontiCore Framework. Masterarbeit, RWTH University, Software Engineering, Germany, 2012.

[Myl16] Mylyn. http://www.eclipse.org/mylyn/, 2016. [Online; abgerufen am: 21.04.2016].

[New15] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[NIS16] U.S. National Institute of Standards and Technology NIST. NIST SQL Test Suite Version 6.0. http://www.itl.nist.gov/div897/ctg/sql_form.htm, 2016. [Online; abgerufen am: 29.03.2016].

[NPR13] Antonio Navarro Pérez und Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J.-M. Bruel, M. Felderer, D. Lugato und A. Dabholka, Editoren, *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, CEUR 1118, Seiten 15–24, Miami, Florida, USA, 2013. CEUR-WS.org.

[NPRI09] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds und Clemente Izurieta. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Caine*, 9:157–162, 2009.

[OMG15a] Object Management Group. MOF Specification Version 2.5. http://www.omg.org/spec/MOF/2.5/PDF/, 2015. [Online; abgerufen am: 05.04.2016].

[OMG15b] Object Management Group. SysML Specification Version 1.4. http://www.omg.org/spec/SysML/1.4/PDF, 2015. [Online; abgerufen am: 05.04.2016].

[OMG15c] Object Management Group. Unified Modeling Language (UML), Version 2.5. http://www.omg.org/spec/UML/2.5/PDF/, 2015. [Online; abgerufen am: 05.04.2016].

[Ora10] Oracle. The Java EE 5 Tutorial. http://docs.oracle.com/javaee/5/tutorial/doc/bnaay.html, 2010. [Online; abgerufen am: 16.04.2016].

[Osb97] Sylvia Osborn. Mandatory Access Control and Role-based Access Control Revisited. In *Proceedings of the Second ACM Workshop on Role-based Access Control*, RBAC '97, Seiten 31–40, New York, NY, 1997. ACM.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first Edition, May 2007.

[PDM+05] Torsten Priebe, Wolfgang Dobmeier, Björn Muschall, Günther Pernul et al. ABAC - Ein Referenzmodell für attributbasierte Zugriffskontrolle. In *Sicherheit*, Seiten 285–296, 2005.

[PFPA06] Oscar Pastor, Joan Fons, Vicente Pelechano und Silvia Abrahão. Conceptual Modelling of Web Applications: The OOWS approach. In *Web Engineering*, Seiten 277–302. Springer, 2006.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen.* Aachener Informatik-Berichte, Software Engineering Band 17. Shaker Verlag, Aachen, Deutschland, 2014.

[PIP+97] Oscar Pastor, Emilio Insfrán, Vicente Pelechano, José Romero und José Merseguer. OO-Method: an OO software production environment combining conventional and formal methods. In *Advanced Information Systems Engineering*, Seiten 145–158. Springer, 1997.

[Ple13] Stefan Plesser. *Aktive Funktionsbeschreibungen zur Planung und Überwachung des Betriebs von Gebäuden und Anlagen.* Dissertation, Technischen Universität Braunschweig, 2013.

[Plo12] Dmytro Plotnikov. Generation and Evolution of Model Based DBUnit Test Cases. Masterarbeit, RWTH University, Software Engineering, Germany, 2012.

[Pot14] Fabian Pottgießer. Ein Eclipse-Wizard für das initiale Setup der generatioven Entwicklung von Enterprise Information Systems. Bachelorarbeit, RWTH University, Software Engineering, Germany, 2014.

[Pra09] Dhanji R. Prasanna. *Dependency Injection.* Manning Publications Co., Greenwich, Connecticut, 1st Edition, 2009.

[Pro15] Protocol Buffers Version 3 Language Specification. `https://developers.google.com/protocol-buffers/docs/reference/proto3-spec`, 2015. [Online; abgerufen am: 31.12.2015].

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web- Informationssystemen.* Aachener Informatik-Berichte, Software Engineering Band (to appear). Shaker Verlag, Aachen, Deutschland, 2016.

[RH06] Ralf Reussner und Wilhelm Hasselbring. *Handbuch der Software-Architektur.* dpunkt Heidelberg, 2006.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems.* Aachener Informatik-Berichte, Software Engineering Band 19. Shaker Verlag, Aachen, Deutschland, 2014.

[RIP11] Davide Di Ruscio, Ludovico Iovino und Alfonso Pierantonio. What is Needed for Managing Co-Evolution in MDE? In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, Seiten 30–38. ACM, 2011.

[RIP12a] Davide Di Ruscio, Ludovico Iovino und Alfonso Pierantonio. Coupled Evolution in Model-Driven Engineering. *Software, IEEE*, 29(6):78–84, 2012.

[RIP12b] Davide Di Ruscio, Ludovico Iovino und Alfonso Pierantonio. Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In *ICGT*, Seiten 20–37. Springer, 2012.

[Roi15] Sebastian Roidl. Tagging Languages. Masterarbeit, RWTH University, Software Engineering, Germany, 2015.

[RPSO07] Gustavo Rossi, Oscar Pastor, Daniel Schwabe und Luis Olsina. *Web Engineering: Modelling and Implementing Web Applications*. Springer Science and Business Media, 2007.

[RR13] Dirk Reiß und Bernhard Rumpe. Using Lightweight Activity Diagrams for Modeling and Generation of Web Information Systems. In Heinrich C. Mayr, Christian Kop, Stephen Liddle und Athula Ginige, Editoren, *Information Systems: Methods, Models, and Applications. 4th International United Information Systems Conference UNISCON 2012, Yalta, Ukraine*, Seiten 61–72. Springer, 2013.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering 2013 Workshopband*, LNI 215, Seiten 155–170. GI, Köllen Druck+Verlag GmbH, Bonn, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Nummer 20 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2014.

[RS01] Erik Ray und Lars Schulten. *Einführung in XML*. O'Reilly, 2001.

[RS08] Gustavo Rossi und Daniel Schwabe. Modeling and implementing web applications with OOHDM. In *Web Engineering: Modelling and Implementing Web Applications*, Seiten 109–155. Springer, 2008.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, Deutschland, 2te Edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring.* Springer, Deutschland, 2te Edition, Juni 2012.

[RW11] Bernhard Rumpe und Ingo Weisemöller. A Domain Specific Transformation Language. In *International Workshop on Models and Evolution (ME)*, Wellington, New Zealand, Oktober 2011.

[RW12] Eric Redmond und Jim Wilson. *Seven Databases in Seven Weeks: a Guide to Modern Databases and the NoSQL Movement.* Pragmatic Bookshelf, 2012.

[SC12] Jane Sleightholme und Ian Chivers. *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77.* Springer Science & Business Media, 2012.

[SCGL14] Jesús Sánchez Cuadrado, Esther Guerra und Juan de Lara. Towards the Systematic Construction of Domain-Specific Transformation Languages. In *Modelling Foundations and Applications*, LNCS 8569, Seiten 196–212. Springer International Publishing, 2014.

[Sch90] Andy Schürr. PROGRES: A VHL-Language Based on Graph Grammars. In *International Workshop on Graph-Grammars and Their Application to Computer Science*, 1990.

[Sch06] Marten Schönherr. Enterprise Application Integration - Serviceorientierung und nachhaltige Architekturen. *Enterprise Architecture Frameworks*, Seiten 3–48, 2006.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Deutschland, 2012.

[Sch13] Sven Schneider. Klassendiagrammsichten zur Generierung spezifischer Datentransferobjekte in Enterprise Information Systems. Masterarbeit, RWTH University, Software Engineering, Germany, 2013.

[Sel07] Bran Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07), Santorini Island, Greece, May 7-9, 2007*, Seiten 2–9, Los Alamitos, California, 2007. IEEE Computer Society.

[Ses97] Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* John Wiley and Sons, Inc., 1997.

[SH93] Steven Spewak und Steven Hill. *Enterprise Architecture Planning: Developing a Blueprint for Data, Applications and Technology.* QED Information Sciences, Inc., 1993.

[Shi16] Apache Shiro. `http://shiro.apache.org/`, 2016. [Online; abgerufen am: 15.02.2016].

[Sir15] Prabath Siriwardena. *Maven Essentials*. Packt Publishing, 2015.

[SM98] Ravi Sandhu und Qamar Munawer. How to Do Discretionary Access Control Using Roles. In *Proceedings of the Third ACM Workshop on Role-based Access Control*, RBAC '98, Seiten 47–54, New York, NY, 1998. ACM.

[Sno14] Monique Snoeck. *Enterprise Information Systems Engineering: The MERODE approach*. Springer, 2014.

[SR95] Daniel Schwabe und Gustavo Rossi. The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, 38(8):45–46, 1995.

[SRS+08] Wieland Schwinger, Werner Retschitzegger, Andrea Schauerhuber, Gerti Kappel, Manuel Wimmer, Birgit Pröll, Cristina Cachero Castro, Sven Casteleyn, Olga De Troyer, Piero Fraternali et al. A Survey on Web Modeling Approaches for Ubiquitous Web Applications. *International Journal of Web Information Systems*, 4(3):234–305, 2008.

[SS11] Christopher Schmitt und Kyle Simpson. *HTML5 Cookbook*. O'Reilly, 2011.

[SSN01] Rahul Sharma, Beth Stearns und Tony Ng. *J2EE Connector Architecture and Enterprise Application Integration*. Addison-Wesley Professional, 2001.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973.

[Sta06] Miroslaw Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In *Model Driven Engineering Languages and Systems*, Seiten 57–72. Springer, 2006.

[SVC06] Thomas Stahl, Markus Voelter und Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley and Sons, 2006.

[SVL13] Eugene Syriani, Hans Vangheluwe und Brian LaShomb. T-Core: A Framework for Custom-Built Model Transformation Engines. *Software and Systems Modeling*, Seiten 1–29, 2013.

[Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, 2003.

[TCP08] Olga De Troyer, Sven Casteleyn und Peter Plessers. WSDM: Web Semantics Design Method. In *Web Engineering: Modelling and Implementing Web Applications*, Seiten 303–351. Springer, 2008.

[Thr16] Apache Thrift. `https://thrift.apache.org/`, 2016. [Online; abgerufen am: 27.03.2016].

[Tiv16] IBM Tivoli Security Policy Manager. `http://www-03.ibm.com/software/products/de/security-policy-manager`, 2016. [Online; abgerufen am: 15.02.2016].

[TMD09] Richard Taylor, Nenad Medvidovic und Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[Tog15] The Open Group. `http://www.opengroup.org/subjectareas/enterprise/togaf`, 2015. [Online; abgerufen am: 22.11.2015].

[Top16] Oracle Toplink Essentials. `https://oss.oracle.com/toplink-essentials-jpa.html`, 2016. [Online; abgerufen am: 27.03.2016].

[Tra16] Trac Open Source Project. `https://trac.edgewall.org/`, 2016. [Online; abgerufen am: 20.04.2016].

[VB07] Dániel Varró und András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3), 2007.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser und Guido Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook. org, 2013.

[Vel15] Apache Velocity. `http://velocity.apache.org/`, 2015. [Online; abgerufen am: 10.11.2015].

[VG07] Markus Völter und Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, Seiten 233–242, Washington, DC, 2007. IEEE Computer Society.

[Vis08] Eelco Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In Ralf Lämmel, João Saraiva und Joost Visser, Editoren, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, LNCS. Springer, 2008.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Deutschland, 2011.

[VWV11] Sander Daniël Vermolen, Guido Wachsmuth und Eelco Visser. Generating database migrations for evolving web applications. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, Seiten 83–92. ACM, 2011.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, Editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, LNCS 4609, Seiten 600–624. Springer-Verlag, July 2007.

[Wed99] Lex Wedemeijer. Semantic Change Patterns in the Conceptual Schema. In Peter Chen, David Embley, Jacques Kouloumdjian, Stephen Liddle und John Roddick, Editoren, *Advances in Conceptual Modeling*, LNCS 1727, Seiten 122–133. Springer Berlin / Heidelberg, 1999.

[Wei11] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, Aachen, Deutschland, 2012.

[WHR14] Jon Whittle, John Hutchinson und Mark Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, 31(3):79–85, 2014.

[Wil03] David Wile. Lessons Learned from Real DSL Experiments. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, Washington, DC, 2003. IEEE Computer Society.

[Wol12] Daniel Wolf. Unterstützung Generativer Entwicklung von J2EE-Anwendungen. Bachelorarbeit, RWTH University, Software Engineering, Germany, 2012.

[WPR10] Jim Webber, Savas Parastatidis und Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly, 2010.

[WS08] Ingo Weisemöller und Andy Schürr. Formal Definition of MOF 2.0 Metamodel Components and Composition. In *MoDELS*, Seiten 386–400, 2008.

[WSM13] Vanessa Wang, Frank Salim und Peter Moskovits. *The Definitive Guide to HTML5 WebSocket*. Springer, 2013.

[WWM+07] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin Van Den Berg, Kim Fleer, David Nelson et al. Experiences in Deploying Model-Driven Engineering. In *SDL 2007: Design for Dependable Systems*, Seiten 35–53. Springer, 2007.

[XML06] Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/2006/REC-xml11-20060816, 2006. [Online; abgerufen am: 10.11.2015].

[Z+87] John Zachman et al. A Framework for Information Systems Architecture. *IBM systems journal*, 26(3):276–292, 1987.

[Zac15] Zachman International. http://www.zachman.com/about-the-zachman-framework, 2015. [Online; abgerufen am: 22.11.2015].

[Zha05] Gefei Zhang. Towards Aspect-oriented Class Diagrams. In *12th Asia-Pacific Software Engineering Conference (APSEC)*, Seite 6. IEEE, 2005.

[ZHJ04] Tewfik Ziadi, Loïc Hélouët und Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering*, LNCS 3014, Seiten 129–139. Springer, 2004.

[Zim16] Matthias Zimmermann. The Eclipse Scout Book. Version vom 04.03.2015. http://wiki.eclipse.org/Scout/Book/, 2016. [Online; abgerufen am: 20.03.2016].

[Zuk05] John Zukowski. *The Definitive Guide to Java Swing*. Apress, 2005.

Teil V

Anhänge

Anhang A

Markierungen in Abbildungen und Listings

Im Folgenden sind die Markierungen, die in Abbildungen und Listings verwendet werden, dargestellt. Dazu zeigt Tabelle A.1 die verwendeten Erkennungsmerkmale, die die Sprache angeben, in der ein Listing oder eine Abbildung verfasst sind. Tabelle A.2 zeigt die in Abbildungen und Listings verwendeten Stereotypen.

Erkennungsmerkmal	**Beschreibung**
CD	Erkennungsmerkmal für Klassendiagramme. Klassendiagramme sind Modelle der Klassendiagrammsprache.
CDV	Erkennungsmerkmal für Sichten. Sichten sind Modelle der Sichtensprache.
DCD	Erkennungsmerkmal für Deltas. Deltas sind Modelle der klassendiagrammspezifischen Deltasprache.
FTL	Erkennungsmerkmal für Templatecode in Freemarker Syntax.
Groovy	Erkennungsmerkmal für Skriptcode in Groovy Syntax.
Java	Erkennungsmerkmal für Quellcode in Java Syntax.
JSON	Erkennungsmerkmal für Quellcode in JSON Syntax.
PRM	Erkennungsmerkmal für Mappingdiagramme. Mappingdiagramme sind Modelle der Mappingsprache.
MCG	Erkennungsmerkmal für Grammatiken in MontiCore Syntax.
MCM	Erkennungsmerkmal für Modelle, deren Sprachen mit MontiCore erstellt wurden.
XML	Erkennungsmerkmal für Skriptcode in XML Syntax.
OD	Erkennungsmerkmal für Objektdiagramme. Objektdiagramme sind Modelle der Objektdiagrammsprache.
PD	Erkennungsmerkmal für Rechtediagramme. Rechtediagramme sind Modelle der Rechtesprache.

Tabelle A.1 wird auf der nächsten Seite fortgesetzt

Erkennungsmerkmal	Beschreibung
Proto	Erkennungsmerkmal für Quellcode, der in Protocol Buffer Syntax geschrieben ist.
RD	Erkennungsmerkmal für Rollendiagramme. Rollendiagramme sind Modelle der Rollensprache.
SQL	Erkennungsmerkmal für Skriptcode, der in SQL Syntax geschrieben ist.
TD	Erkennungsmerkmal für Tagdefinitionen. Tagdefinitionen sind Modelle der Tagdefinitionssprache.
TS	Erkennungsmerkmal für Tagschemas. Tagschemas sind Modelle der Tagschemasprache.
«gen»	Erkennungsmerkmal eines generierten Elements.
«hwc»	Erkennungsmerkmal eines handgeschriebenen Elements.
«rte»	Erkennungsmerkmal eines Elements der Laufzeitumgebung.
...	Erkennungsmerkmale einer unvollständigen Abbildung oder eines unvollständigen Listings.

Tabelle A.1: Erklärung der in Abbildungen und Listings verwendeten Erkennungsmerkmale.

Stereotyp	Beschreibung
«abstract»	Stereotyp einer abstrakten Klasse eines Klassendiagramms.
«derived»	Stereotyp einer Grammatik. Er gibt an, dass die Grammatik methodisch abgeleitet wurde.
«enum»	Stereotyp einer Enumeration in einem Klassendiagramm.
«existing»	Stereotyp einer Grammatik. Er gibt an, dass die Grammatik bereits existent war und nicht veränderbar ist.
«handwritten»	Stereotyp einer Grammatik. Er gibt an, dass die Grammatik handgeschrieben ist.
«interface»	Stereotyp eines Interfaces eines Klassendiagramms.
«predefined»	Stereotyp einer Grammatik. Er gibt an, dass die Grammatik vordefiniert ist und allgemeine Produktionen für Subgrammatiken bereitstellt.

Tabelle A.2: Erklärung der in Abbildungen und Listings verwendeten Stereotypen.

Anhang B

Abkürzungen

ABAC	Attribute-Based Access Control
ACL	Access Control List
AG	Auftraggeber
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
BPMN	Business Process Model and Notation
CD	Klassendiagramm
CDV	klassendiagrammartige Sicht
CORBA	Common Object Request Broker Architecture
CRUD	Create, Read, Update, Delete
DAC	Discretionary Access Control
DAO	Data Access Object
DCD	Delta
DCOM	Distributed Component Object Model
DSL	Domänenspezifische Sprache
DSML	Domänenspezifische Modellierungssprache
DTO	Data Transfer Object
EAR	Enterprise Archive
EBNF	Erweiterte Backus-Naur Form

EIS	Enterprise Informationssystem
ER-Modell	Entity-Relationship-Modell
EJB	Enterprise JavaBeans
FA	funktionale Anforderung
GPL	General Purpose Language
GPML	General Purpose Modeling Language
GUI	Graphische Benutzerschnittstelle
HDF5	Hierarchical Data Format 5
HQL	Hibernate Query Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDL	Interface Definition Language
JAR	Java Archive
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JDBC	Java Database Connectivity
JEE	Java Enterprise Edition
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
LOC	Lines of Code
MAC	Mandatory Access Control
MBE	Model-Based Engineering
MDA	Model-Driven Architecture
MDD	Model-Driven Development

MDE	Model-Driven Engineering
MOF	Meta Object Facility
MOM	Message Oriented Middleware
M2M	Model-to-Model
M2T	Model-to-Text
NFA	nichtfunktionale Anforderung
NIM	Nachbarschaftsinformationsmodell
NIST	National Institute of Standards and Technology
NoSQL	Not only SQL
OCL	Object Constraint Language
OD	Objektdiagramm
OMG	Object Management Group
OOHDM	Object-Oriented Hypermedia Design Method
OOWS	Object-Oriented Web Solution
ORB	Object Request Broker
ORM	Object-Relational Mapper
PD	Rechtediagramm
PE	Produktentwickler
PIM	Platform-Independent Model
POM	Project Object Model
PRM	Mappingdiagramm
PSM	Platform-Specific Model
QoS	Quality of Service
QVT	Query View Transformation
RAP	Remote Application Platform
RBAC	Role Based Access Control
RD	Rollendiagramm

REST	Representational State Transfer
RIA	Rich Internet Application
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SC	Zustandsdiagramm
SD	Sequenzdiagramm
SQL	Structured Query Language
SOAP	Simple Object Access Protokoll
SVN	Subversion
SWT	Standard Widget Toolkit
TC	Testfall
TCP	Transmission Control Protocol
TD	Tagdefinition
TS	Tagschema
UDP	User Datagram Protocol
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Unified Resource Locator
UWE	UML-based Web Engineering
WE	Werkzeugentwickler
WSDM	Web Semantics Design Method
WSDL	Webservice Description Language
XACML	eXtensible Access Control Markup Language
XML	Extensible Markup Language
XSD	XML-Schema Definition
XSLT	Extensible Stylesheet Language Transformations

Anhang C

Modelle und Grammatiken

Im Folgenden sind die vollständigen Modelle und Grammatiken, wie sie im Rahmen dieser Arbeit verwendet werden, aufgeführt.

C.1 Vollständiges Klassendiagramm des sozialen Netzwerks

CD

```
1  import java.util.List;
2  import java.util.Set;
3  import java.util.Date;
4
5  classdiagram SocNet {
6
7    /************************************************************
8     * Profiles, Persons, Commercials
9     * There are two types of Profiles:
10    *                    Persons and Commercials
11   ************************************************************/
12
13   abstract class Profile {
14     String userName;
15     String password;
16   }
17
18   // private people may have profiles
19   class Person extends Profile {
20     String realName;
21     int zip;
22     String street;
23     String city
24   }
25
26   // companies may have commercial profiles
27   class Commercial extends Profile;
28
29   // several persons can have several friends
30   association friends [*] Person -> Person [*];
```

```
31
32    // a person can follow several commercial profiles
33    association follows [*] Person -> Commercial [*];
34
35    // a commercial profile has one of the following tariffs:
36    enum Tariff { GOLD, PLATINUM; }
37
38    association [*] Commercial -> Tariff [1];
39
40    /*************************************************************
41     * Post
42     * Profiles can sent posts that may be
43     * seen by other profiles
44    *************************************************************/
45
46    class Post {
47      boolean hasPhoto;
48    }
49
50    // a profile can sent several posts
51    association sent [1] Profile -> Post [*];
52  }
```

Listing C.1: Darstellung der vollständigen textuellen Syntax des Klassendiagramms des sozialen Netzwerks auf Basis des Szenarios aus Abschnitt 3.5.

C.2 Grammatik des sprachunabhängigen Teils der Tagdefinitionssprache

MCG TD-Common

```
1  package mc.tagging.lang.tags;
2
3  grammar TDCommon extends mc.umlp.common.Common {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8
9    options{
10     compilationunit TagDefinition
11   }
12
13   /*************************************************************
14    * ast rules
15   *************************************************************/
16
```

```
17  ast SubTag =
18    method public mc.types._ast.ASTQualifiedName getName(){}
19    method public void setName
20      (mc.types._ast.ASTQualifiedName name){}
21  ;
22
23  ast ModelElementIdentifier =
24    method public mc.tagging.rte.AMatcher createMatcher(){}
25
26    /*
27      Name and Type are needed for the symbol table
28    */
29    method public String getType(){}
30    method public String getNameAsString(){}
31    method public void setName
32      (mc.types._ast.ASTQualifiedName name){}
33
34    /*
35      If not provided, the Default Equivalence is used
36      when Identifiers have to be compared
37    */
38    method public com.google.common.base.Optional
39      <com.google.common.base.Equivalence<ASTIdentifier>>
40      getEquivalence(){}
41  ;
42
43  /*************************************************************
44   * productions
45  *************************************************************/
46
47  /*
48    Interface that should be extented in Subgrammars to
49    specify how elements in targetgrammars can be addressed.
50  */
51  interface / ModelElementIdentifier astextends
52    /mc.tagging.lang.tags._ast.ISurroundingContextContainer;
53
54  /* Necessary in order to create the ASTIdentifier
55    (which is extended in Sublanguages). Also useful for tests
56  */
57  DefaultIdent implements ModelElementIdentifier astextends
58    /mc.tagging.lang.tags._ast.TagNode = name:QualifiedName;
59
60  ModelElementIdentifierPath =
61    parts:ModelElementIdentifier
62    ("." parts:ModelElementIdentifier)* ;
63
64  interface Tag astextends SubTag;
65
```

```
66    interface SubTag astextends
67      /mc.tagging.lang.tags._ast.IASTSurroundingContext,
68      /mc.tagging.lang.tags._ast.ITagNode;
69
70    /*
71      Start of Grammar.
72    */
73    / TagDefinition
74      astimplements
75        /mc.tagging.lang.tags._ast.IASTSurroundingContext
76      astextends /mc.tagging.lang.tags._ast.TagNode =
77        (hasSchema:["conforms"] "to"
78        schemas:QualifiedName ("," schemas:QualifiedName)*";")?
79        "tags" Name "for" targetModel:QualifiedName "{"
80          (contexts:Context | tags:TargetElement)*
81        "}" ;
82
83    / Context
84      astimplements
85        /mc.tagging.lang.tags._ast.IASTSurroundingContext
86      astextends /mc.tagging.lang.tags._ast.TagNode =
87        "within" ModelElementIdentifierPath "{"
88          (contexts:Context | tags:TargetElement)*
89        "}" ;
90
91    / TargetElement
92      astimplements
93        /mc.tagging.lang.tags._ast.IASTSurroundingContext
94      astextends /mc.tagging.lang.tags._ast.TagNode =
95        "tag" ModelElementIdentifierPath
96        ("," moreTargets:ModelElementIdentifierPath)* "with"
97        tags:Tag ("," tags:Tag)* ";";
98
99    / SimpleTag
100     implements (QualifiedName ("," | ";")) => Tag,
101     (QualifiedName ("," | ";")) => SubTag
102     astextends /mc.tagging.lang.tags._ast.TagNode =
103       name:Name;
104
105   / ValuedTag
106     implements (QualifiedName "=") => Tag,
107     (QualifiedName "=") => SubTag
108     astextends /mc.tagging.lang.tags._ast.TagNode =
109       name:Name "=" value:StringLiteral ;
110
111   / ComplexTag
112     implements (QualifiedName "{") => Tag,
113     (QualifiedName "{") => SubTag
114     astextends /mc.tagging.lang.tags._ast.TagNode =
```

```
115       name:Name "{"
116         (subtags:SubTag ("," subtags:SubTag)* ";")?
117       "}" ;
118
119   / ListSubTag
120     implements (QualifiedName "=" "[") => SubTag
121     astextends /mc.tagging.lang.tags._ast.TagNode =
122       name:QualifiedName "=" "["
123         (listValues:ListValue ("," listValues:ListValue)*)?
124      "]" ;
125
126   / ListValue
127     astextends /mc.tagging.lang.tags._ast.TagNode =
128     (QualifiedName "{") => ComplexTag |
129       SimpleTag  | basicValue:StringLiteral;
130 }
```

Listing C.2: Darstellung der vollständigen Grammatik TDCommon der Tagdefinitionssprache.

C.3 Grammatik des klassendiagrammspezifischen Teils der Tagdefinitionssprache

```
1  package mc.tagging.lang.tags.cd;                                    MCG TD
2
3  grammar TD extends mc.tagging.lang.tags.TDCommon, mc.umlp.cd.CD {
4
5    /*************************************************************
6     * productions
7    *************************************************************/
8
9    / MethodIdentifier
10     implements Identifier
11     astextends /mc.tagging.lang.tags._ast.TagNode =
12       name:QualifiedName ("("
13       ( methodParams:Type ("," methodParams:Type)*)? ")")?;
14
15   / AttributeIdentifier
16     implements Identifier
17     astextends /mc.tagging.lang.tags._ast.TagNode =
```

```
18        name:QualifiedName;
19
20    / AssociationElementIdentifier
21      implements ModelElementIdentifier
22      astextends /mc.tagging.lang.tags._ast.TagNode =
23        (Name | CDAssociation) (["!lefthand"] | ["!righthand"]);
24
25  }
```

Listing C.3: Darstellung der vollständigen Grammatik TD der Tagdefinitionssprache.

C.4 Grammatik des sprachunabhängigen Teils der Tagschemasprache

```
1  package mc.tagging.lang.tagschema;                                    MCG TS-Common
2
3  grammar TSCommon extends mc.umlp.common.Common {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8
9    options {
10     compilationunit TagSchema
11   }
12
13   /*************************************************************
14    * ast rules
15   *************************************************************/
16
17   ast TagType =
18     method public ASTScopeToken getScopeToken(){}
19   ;
20
21   /*************************************************************
22    * productions
23   *************************************************************/
24
25   interface TagType;
26
27   TagSchema
28   astimplements /mc.tagging.lang.tagschema._ast.TagSchemaNode =
29     "tagschema" Name "{" TagType* "}" ;
30
31
```

```
32   /*************************************************************
33    * tag types
34    *************************************************************/
35
36   SimpleTagType
37   implements (Scope? ";") => TagType
38   astimplements /mc.tagging.lang.tagschema._ast.TagSchemaNode =
39     ["inner"]? "tagtype" Name Scope? ";" ;
40
41   EnumeratedTagType
42   implements TagType
43   astimplements /mc.tagging.lang.tagschema._ast.TagSchemaNode =
44     ["inner"]? "tagtype" Name ":" "["
45       enumerationValues:String ("|" enumerationValues:String)*
46     "]" Scope? ";" ;
47
48   ValuedTagType
49   implements TagType
50   astimplements /mc.tagging.lang.tagschema._ast.TagSchemaNode =
51     ["inner"]? "tagtype" Name ":"
52       ("int"|"String"|"Boolean") Scope? ";" ;
53
54   ComplexTagType
55   implements TagType
56   astimplements /mc.tagging.lang.tagschema._ast.TagSchemaNode =
57     ["inner"]? "tagtype" Scope? "{"
58       subTags:Attribute ("," subTags:Attribute)* ";"
59     "}" ;
60
61   Attribute = Name ":"
62     ("int"|"String"|"Boolean"|Name) Cardinality?;
63
64   Cardinality
65   astimplements /mc.tagging.lang.tagschema._ast.TagSchemaNode =
66     zero_to_one:["?"] | one_to_many:["+"] | zero_to_many:["*"];
67
68   interface ScopeIdentifier;
69
70   Scope = "for" (ScopeIdentifier ("," ScopeIdentifier)* | "*");
71 }
```

Listing C.4: Darstellung der vollständigen Grammatik TSCommon der Tagschemasprache.

C.5 Grammatik des klassendiagrammspezifischen Teils der Tagschemasprache

MCG TS

```
1  package mc.tagging.lang.tagschema.cd;
2
3  grammar TS extends mc.tagging.lang.tagschema.TSCommon {
4
5    /*************************************************************
6     * productions
7    *************************************************************/
8
9    CDClassScope implements ScopeIdentifier = "CDClass";
10   CDEnumScope implements ScopeIdentifier = "CDEnum";
11   CDInterfaceScope implements ScopeIdentifier = "CDInterface";
12
13   CDAttributeScope implements ScopeIdentifier = "CDAttribute";
14   CDMethodScope implements ScopeIdentifier = "CDMethod";
15
16   CDAssociationScope
17   implements ScopeIdentifier = "CDAssociation";
18
19   CDAssociationLefthandScope
20   implements ScopeIdentifier = "CDAssociation" "!" "lefthand";
21
22   CDAssociationRighthandScope
23   implements ScopeIdentifier = "CDAssociation" "!" "righthand";
24
25 }
```

Listing C.5: Darstellung der vollständigen Grammatik TS der Tagschemasprache.

C.6 Grammatik der Sichtensprache

MCG CDV

```
1  package mc.montiee.cd.view;
2
3  grammar CDV extends mc.umlp.cd.CD {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8
9
```

```
10  options {
11    compilationunit CDViewDefinition
12  }
13
14  /*****************************************************
15   * ast rules
16  *****************************************************/
17
18  // for editor (analogously to mc.umlp.cd.CD)
19  ast CDAssociationView =
20    arrow:/String
21    method public String getArrow() {
22      if (leftToRight) {
23        return "->";
24      } else if (rightToLeft) {
25        return "<-";
26      } else if (bidirectional) {
27        return "<->";
28      }
29      return "--";
30    };
31
32
33  /*****************************************************
34   * productions
35  *****************************************************/
36
37  /* represents a class diagram view.
38   @attribute stereotype          optional stereotype
39   @attribute name                name of the class diagram view
40   @attribute master              name of the master class
41                                  diagram, i.e., the diagram
42                                  the view is based on
43   @attribute cDVClassViews       list of all class views
44   @attribute cDVInterfaceViews   list of all interface views
45   @attribute cDVEnumViews        list of all enum views
46   @attribute cDVAssociationViews list of all association views
47  */
48  / CDViewDefinition extends CDDefinition =
49    Stereotype? "classdiagramview" Name
50      "of" master:QualifiedName
51    "{"
52      (
53        (Modifier "classview")=>
54        cDVClassViews:CDClassView
55        |
56        (Modifier "interfaceview")=>
57        cDVInterfaceViews:CDInterfaceView
58        |
```

```
59          (Modifier "enumview")=>
60          cDVEnumViews:CDEnumView
61          |
62          (Modifier ("associationview"|"aggregationview"|
63                     "compositionview"))=>
64          cDVAssociationViews:CDAssociationView
65        )*
66      "}";
67
68
69    /* represents a class view in a class diagram view.
70     @attribute modifier       modifier of the class view
71     @attribute name           name of the class view
72     @attribute typeParameters generic type parameters of the
73                               class view
74     @attribute superclasses   list of superclasses of the
75                               class view
76     @attribute interfaces     list of interfaces implemented
77                               by the class view
78     @attribute cDConstructors list of constructors of the
79                               class view
80     @attribute cDMethods      list of methods of the class view
81     @attribute cDAttributes   list of attributes of the class view
82    */
83    / CDClassView extends CDClass =
84        Modifier "classview" Name
85        (options{greedy=true;}: TypeParameters)?
86        ("extends" superclasses:ReferenceType
87                   ("," superclasses:ReferenceType)*)?
88        ("implements" interfaces:ReferenceType
89                   ("," interfaces:ReferenceType)*)?
90        (
91          ("{" (
92            (Modifier Type Name ("=" | ";"))
93            => cDAttributes:CDAttribute
94            |
95            (Modifier (options{greedy=true;}: TypeParameters)?
96              Name "(") => cDConstructors:CDConstructor
97            |
98            cDMethods:CDMethod
99          )* "}")
100         |
101         ";"
102       );
103
104
105   /* represents an interface view in a class diagram view.
106    @attribute modifier       modifier of the interface view
107    @attribute name           name of the interface view
```

```
108     @attribute typeParameters generic type parameters of the
109                               interface view
110     @attribute interfaces     list of interfaces extended by the
111                               interface view
112     @attribute cDMethods      list of methods of the
113                               interface view
114     @attribute cDAttributes   list of attributes of the
115                               interface view
116   */
117   / CDInterfaceView extends CDInterface =
118       Modifier "interfaceview" Name
119       (options{greedy=true;}: TypeParameters)?
120       ("extends" interfaces:ReferenceType
121         ("," interfaces:ReferenceType)*)?
122       (
123         ("{" (
124           (Modifier Type Name ("=" | ";"))
125           => cDAttributes:CDAttribute
126           |
127           cDMethods:CDMethod
128         )* "}")
129         |
130         ";"
131       );
132
133
134   /* represents an enumeration view in a class diagram view.
135     @attribute modifier        modifier of the enum view
136     @attribute name            name of the enum view
137     @attribute interfaces      list of interfaces implemented by
138                                the enum view
139     @attribute cDEnumConstants list of the enum constants
140     @attribute cDConstructors  list of constructors of
141                                the enum view
142     @attribute cDMethods       list of methods of the enum view
143     @attribute cDAttributes    list of attributes of the enum view
144   */
145   / CDEnumView extends CDEnum =
146       Modifier "enumview" Name
147       ("implements" interfaces:ReferenceType
148         ("," interfaces:ReferenceType)*)?
149       (
150         ("{"(
151           cDEnumConstants:CDEnumConstant
152           ("," cDEnumConstants:CDEnumConstant)* ";"
153           (
154             (Modifier Type Name ("=" | ";"))
155             => cDAttributes:CDAttribute
156             |
```

```
157             (Modifier ("<" TypeVariableDeclaration
158             ("," TypeVariableDeclaration)*
159              (">" | ">>" | ">>>")?)?
160             Name "("
161             => cDConstructors:CDConstructor
162             |
163             cDMethods:CDMethod
164           )*
165         )?"}")
166         |
167         ";"
168       );
169
170
171   /* represents an association view in a class diagram view.
172    @attribute modifier                modifier of the
173                                       association view
174    @attribute Association             true if the association
175                                       view is of type
176                                       "associationview"
177    @attribute Aggregation             true if the association
178                                       view is of type
179                                       "aggregation"
180    @attribute Composition             true if the association
181                                       view is of type
182                                       "composition"
183    @attribute type                    type of the association
184                                       view (association,
185                                             aggregation,
186                                             or composition)
187    @attribute derived                 true if the is a derived
188                                       association view
189    @attribute name                    name of the association
190                                       view
191    @attribute leftModifier            optional left side
192                                       modifier
193    @attribute leftCardinality         optional cardinality of
194                                       the left side of
195                                       the association view
196    @attribute leftReferenceName       name of the class or
197                                       interface on the
198                                       left side of
199                                       the association view
200    @attribute leftQualifier           optional qualifier of
201                                       the left side of
202                                       the association view
203    @attribute leftRole                optional role of the class
204                                       or interface on
205                                       the left side of
```

```
206                                          the association view
207    @attribute leftToRight                true if the association
208                                          view is navigable from
209                                          left to right ("->")
210    @attribute rightToLeft                true if the association
211                                          view is navigable from
212                                          right to left ("<-")
213    @attribute bidirectional              true if the association
214                                          view is navigable in both
215                                          directions ("<->")
216    @attribute simple                     true if navigation of
217                                          the association is not
218                                          specified  ("--")
219    @attribute rightRole                  optional role of the
220                                          class or interface on
221                                          the right side of
222                                          the association view
223    @attribute rightQualifier             optional qualifier of
224                                          the right side of
225                                          the association view
226    @attribute rightReferenceName         name of the class
227                                          or interface on the
228                                          right side of
229                                          the association view
230    @attribute rightCardinalityOptional   cardinality of the
231                                          right side of
232                                          the association view
233    @attribute rightModifier              optional right side
234                                          modifier
235    */
236    / CDAssociationView extends CDAssociation =
237        Modifier
238        (
239          Association:["associationview"]
240          | Aggregation:["aggregationview"]
241          | Composition:["compositionview"]
242        )
243        (options{greedy=true;}: Derived:[DERIVED:"/"])?
244        ((Name Modifier Cardinality? QualifiedName)=> Name |)
245        leftModifier:Modifier
246        leftCardinality:Cardinality?
247        leftReferenceName:QualifiedName
248        ("[" leftQualifier:CDQualifier "]")?
249        ("(" leftRole:Name ")")?
250        (
251          leftToRight:["->"]
252          | rightToLeft:["<-"]
253          | bidirectional:["<->"]
254          | simple:["--"]
```

```
255       )
256       ("(" rightRole:Name ")")?
257       ("[" rightQualifier:CDQualifier "]")?
258       rightReferenceName:QualifiedName
259       rightCardinality:Cardinality?
260       rightModifier:Modifier
261       ";";
262
263
264   /* represents a modifier for classes, interfaces, methods,
265    * constructors, attributes, and associations.
266    @attribute stereotype Optional stereotype
267    @attribute public     true if modifier is public
268                          (i.e., modifier written as
269                           "public" or "+")
270    @attribute private    true if modifier is private
271                          (i.e., modifier written as
272                           "private" or "-")
273    @attribute protected  true if modifier is protected
274                          (i.e., modifier written as
275                           "protected" or "#")
276    @attribute final      true if modifier is final
277                          (i.e., modifier written as
278                           "final")
279    @attribute abstract   true if modifier is abstract
280                          (i.e., modifier written as
281                           "abstract")
282    @attribute local      true if modifier is local
283                          (i.e., modifier written as
284                           "local")
285    @attribute full       true if modifier is full
286                          (i.e., modifier written as
287                           "full")
288    @attribute derived    true if modifier is derived
289                          (i.e., modifier written as
290                           "derived" or "/")
291    @attribute readonly   true if modifier is readonly
292                          (i.e., modifier written as
293                           "readonly" or "?")
294    @attribute static     true if modifier is static
295                          (i.e., modifier written as
296                           "static")
297    @attribute flat       true if modifier is flat
298                          (i.e., modifier written as
299                           "flat")
300   */
301   Modifier =
302     Stereotype?
303     (
```

```
304         Public:["public"] | Public:[PUBLIC:"+"]
305       | Private:["private"] | Private:[PRIVATE:"-"]
306       | Protected:["protected"] | Protected:[PROTECTED:"#"]
307       | Final:["final"]
308       | Abstract:["abstract"]
309       | Local:["local"]
310       | Full:["full"]
311       | Derived:["derived"] | Derived:[DERIVED:"/"]
312       | Readonly:["readonly"] | Readonly:[READONLY:"?"]
313       | Static:["static"]
314       | Flat:["flat"]
315     )*;
316 }
```

Listing C.6: Darstellung der vollständigen Grammatik der Sichtensprache.

C.7 Grammatik der Rollensprache

MCG
RD

```
1  package mc.montiee.roles;
2
3  grammar RD extends mc.umlp.common.Common {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8    options {
9      compilationunit Roles
10   }
11
12   /*************************************************************
13    * productions
14   *************************************************************/
15
16   /* represents a role diagram
17    @attribute name           name of the role diagram
18    @attribute roleDefinition role definition list defined in
19                              this role diagram
20   */
21   Roles = "rolediagram" name:Name "{" (RoleDefinition)* "}";
22
23
```

```
24    /* represents a role in the role diagram
25     @attribute roleName        name of the role
26     @attribute parentRoleNames parent role names for this role
27    */
28    RoleDefinition = "role" roleName:Name (["extends"]
29      parentRoleNames:QualifiedName
30        ("," parentRoleNames:QualifiedName)* )? ";"
31    ;
32
33  }
```

Listing C.7: Darstellung der vollständigen Grammatik der Rollensprache.

C.8 Grammatik der Rechtesprache

```
1  package mc.montiee.permission;                                    MCG PD
2
3  grammar PD extends mc.umlp.common.Common {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8    options {
9      compilationunit Permissions
10   }
11
12   /*************************************************************
13    * productions
14   *************************************************************/
15
16   /* represents a permission diagram.
17    @attribute name                 name of the role
18                                    mapping diagram
19    @attribute permissionDefinition permission definitions in
20                                    this diagram
21   */
22   Permissions= "permissiondiagram" name:Name "for"
23      classdiagram:QualifiedName
24     "{"
25       (PermissionDefinition)*
26     "}";
27
28   /* represents a permission set for an entity
29    @attribute instanceNames         used for instance level
30                                     permission definitions
```

```
31    @attribute className             name of the class the
32                                     permission definition
33                                     is derived from
34    @attribute modifyPermission      modify permissions
35    @attribute queryPermission       query permissions
36    @attribute associationPermission association permissions
37   */
38   PermissionDefinition = "context"
39     instanceNames:Name ("," instanceNames:Name)*
40     ":" className:QualifiedName
41     "{"
42       (ModifyPermission | QueryPermission
43         | AssociationPermission)*
44     "}";
45
46   /* ModifyPermission represents modification permissions
47    * for an entity
48    @attribute modifyTypes  type of modification permission
49    @attribute instanceName instance of this modification
50   */
51   ModifyPermission =
52     (modifyTypes:ModifyType) ("," modifyTypes:ModifyType)*
53       instanceName:Name ";";
54
55   /* Type of modification */
56   ModifyType = ["create"] | ["read"] | ["update"] | ["delete"];
57
58  /* QueryPermission represents permissions derived
59   * from a query
60    @attribute queryName name of the query
61   */
62   QueryPermission = ["query"] queryName:Name ";";
63
64   /* AssociationPermission represents permissions derived
65    * from an association
66    @attribute instanceName  instance name that has
67                             this association
68    @attribute attributePath path of the attribute
69   */
70   AssociationPermission = ["association"]
71     instanceName:Name "." attributePath:QualifiedName ";"
72   ;
73 }
```

Listing C.8: Darstellung der vollständigen Grammatik der Rechtesprache.

C.9 Grammatik der Mappingsprache

MCG
PRM

```
1  package mc.montiee.mapping;
2
3  grammar PRM extends mc.umlp.common.Common {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8    options {
9      parser lookahead=2
10     compilationunit RoleMapping
11   }
12
13   /* language embedding */
14   external MappingPermissionDefinition;
15
16   /*************************************************************
17    * productions
18   *************************************************************/
19
20   /* represents a role mapping diagram.
21    @attribute name    name of the role mapping diagram
22    @attribute mapping mapping list defined in this
23                       role mapping diagram
24   */
25   RoleMapping = "rolemappingdiagram" name:Name "{" Mapping* "}";
26
27   /* represents a mapping between one role and
28      target permission definition
29    @attribute roleName                    name of the role for
30                                           permission assignments
31    @attribute mappingPermissionDefinition list of target
32                                           permission definitions
33                                           assigned to given role
34   */
35   Mapping = roleName:QualifiedName
36     "{"
37       MappingPermissionDefinition*
38     "}";
39
40 }
```

Listing C.9: Darstellung der vollständigen Grammatik der Mappingsprache.

C.10 Grammatik der Deltasprache

MCG DCD

```
1  package mc.montiee.cd.delta;
2
3  grammar DCD extends mc.umlp.cd.CD, DCommon {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8    options {
9      compilationunit DeltaCDDefinition
10     parser lookahead=5
11     lexer lookahead=7
12   }
13
14   ast DeltaCDDefinition astextends
15     /mc.umlp.cd._ast.ASTCDDefinition =
16       method public String getName() {
17         return getDelta().getName();
18       }
19   ;
20
21   /*************************************************************
22    * productions
23   *************************************************************/
24
25   DeltarightModifierModifierDeltaOperation implements
26     (DeltaOperand "rightModifier")=> DeltaOperation =
27     operand:DeltaOperand "rightModifier"
28     rightModifier:Modifier ";";
29
30   DeltarightCardinalityCardinalityDeltaOperation implements
31     (DeltaOperand "rightCardinality")=> DeltaOperation =
32     operand:DeltaOperand "rightCardinality"
33     rightCardinality:Cardinality ";";
34
35   DeltarightReferenceNameQualifiedNameDeltaOperation implements
36     (DeltaOperand "rightReferenceName")=> DeltaOperation =
37     operand:DeltaOperand "rightReferenceName"
38     rightReferenceName:QualifiedName ";";
39
40   DeltarightQualifierCDQualifierDeltaOperation implements
41     (DeltaOperand "rightQualifier")=> DeltaOperation =
42     operand:DeltaOperand "rightQualifier"
43     rightQualifier:CDQualifier ";";
44
```

```
45  DeltaleftQualifierCDQualifierDeltaOperation implements
46    (DeltaOperand "leftQualifier")=> DeltaOperation =
47    operand:DeltaOperand "leftQualifier"
48    leftQualifier:CDQualifier ";";
49
50  DeltasimpleDeltaOperation implements (DeltaOperand "simple")
51    => DeltaOperation = operand:DeltaOperand "simple"
52    simple:["--"] ";";
53
54  DeltabidirectionalDeltaOperation implements
55    (DeltaOperand "bidirectional")=> DeltaOperation =
56    operand:DeltaOperand "bidirectional"
57    bidirectional:["<->"] ";";
58
59  DeltarightToLeftDeltaOperation implements
60    (DeltaOperand "rightToLeft")=> DeltaOperation =
61    operand:DeltaOperand "rightToLeft"
62    rightToLeft:["<-"] ";";
63
64  DeltaleftToRightDeltaOperation implements
65    (DeltaOperand "leftToRight")=> DeltaOperation =
66    operand:DeltaOperand "leftToRight"
67    leftToRight:["->"] ";";
68
69  DeltaleftReferenceNameQualifiedNameDeltaOperation implements
70    (DeltaOperand "leftReferenceName")=> DeltaOperation =
71    operand:DeltaOperand "leftReferenceName"
72    leftReferenceName:QualifiedName ";";
73
74  DeltaQualifiedNameDeltaOperation implements
75    (DeltaOperand "QualifiedName")=> DeltaOperation =
76    operand:DeltaOperand "QualifiedName" QualifiedName ";";
77
78  DeltaleftCardinalityCardinalityDeltaOperation implements
79    (DeltaOperand "leftCardinality")=> DeltaOperation =
80    operand:DeltaOperand "leftCardinality"
81    leftCardinality:Cardinality ";";
82
83  DeltaCardinalityDeltaOperation implements
84    (DeltaOperand "Cardinality")=> DeltaOperation =
85    operand:DeltaOperand "Cardinality" Cardinality ";";
86
87  DeltaleftModifierModifierDeltaOperation implements
88    (DeltaOperand "leftModifier")=> DeltaOperation =
89    operand:DeltaOperand "leftModifier"
90    leftModifier:Modifier ";";
91
92  DeltaDerivedDeltaOperation implements
93    (DeltaOperand "Derived")=> DeltaOperation =
```

```
94    operand:DeltaOperand "Derived" Derived:["/"] ";";
95
96  DeltaCompositionDeltaOperation implements
97    (DeltaOperand "Composition")=> DeltaOperation =
98    operand:DeltaOperand "Composition"
99    Composition:["composition"] ";";
100
101 DeltaAggregationDeltaOperation implements
102   (DeltaOperand "Aggregation")=> DeltaOperation =
103   operand:DeltaOperand "Aggregation"
104   Aggregation:["aggregation"] ";";
105
106 DeltaAssociationDeltaOperation implements
107   (DeltaOperand "Association")=> DeltaOperation =
108   operand:DeltaOperand "Association"
109   Association:["association"] ";";
110
111 DeltaCDAssociationOperation implements
112   (DeltaOperand CDAssociation)=> DeltaOperation =
113   operand:DeltaOperand CDAssociation;
114
115 DeltaCDAssociationScopeIdentifier implements ScopeIdentifier =
116   "CDAssociation";
117
118 CDAssociationIdentifier implements
119   ("[" CDAssociation "]")=> ModelElementIdentifier =
120   "[" CDAssociation "]";
121
122 DeltaCDQualifierOperation implements
123   (DeltaOperand CDQualifier)=> DeltaOperation =
124   operand:DeltaOperand CDQualifier;
125
126 DeltaCDQualifierScopeIdentifier implements ScopeIdentifier =
127   "CDQualifier";
128
129 CDQualifierIdentifier implements
130   ("[" CDQualifier "]")=> ModelElementIdentifier =
131   "[" CDQualifier "]";
132
133 DeltaCDAttributeOperation implements
134   (DeltaOperand CDAttribute)=> DeltaOperation =
135   operand:DeltaOperand CDAttribute;
136
137 DeltaCDAttributeScopeIdentifier implements ScopeIdentifier =
138   "CDAttribute";
139
140 CDAttributeIdentifier implements
141   ("[" CDAttribute "]")=> ModelElementIdentifier =
142   "[" CDAttribute "]";
```

```
143
144   DeltaEllipsisDeltaOperation implements
145     (DeltaOperand "Ellipsis")=> DeltaOperation =
146     operand:DeltaOperand "Ellipsis" Ellipsis:["..."] ";";
147
148   DeltaCDParameterOperation implements
149     (DeltaOperand CDParameter)=> DeltaOperation =
150     operand:DeltaOperand CDParameter;
151
152   DeltaCDParameterScopeIdentifier implements ScopeIdentifier =
153     "CDParameter";
154
155   CDParameterIdentifier implements
156     ("[" CDParameter "]")=> ModelElementIdentifier =
157     "[" CDParameter "]";
158
159   DeltaCDConstructorOperation implements
160     (DeltaOperand CDConstructor)=> DeltaOperation =
161     operand:DeltaOperand CDConstructor;
162
163   DeltaCDConstructorScopeIdentifier implements ScopeIdentifier =
164     "CDConstructor";
165
166   CDConstructorIdentifier implements
167     ("[" CDConstructor "]")=> ModelElementIdentifier =
168     "[" CDConstructor "]";
169
170   DeltaexceptionsQualifiedNameDeltaOperation
171     implements (DeltaOperand "exceptions")=> DeltaOperation =
172     operand:DeltaOperand "exceptions"
173     exceptions:QualifiedName ";";
174
175   DeltacDParametersCDParameterDeltaOperation implements
176     (DeltaOperand "cDParameters")=> DeltaOperation =
177     operand:DeltaOperand "cDParameters"
178     cDParameters:CDParameter ";";
179
180   DeltaCDMethodOperation implements
181     (DeltaOperand CDMethod)=> DeltaOperation =
182     operand:DeltaOperand CDMethod;
183
184   DeltaCDMethodScopeIdentifier implements ScopeIdentifier =
185     "CDMethod";
186
187   CDMethodIdentifier implements
188     ("[" CDMethod "]")=> ModelElementIdentifier =
189     "[" CDMethod "]";
190
191   DeltaCDEnumParameterOperation implements
```

```
192     (DeltaOperand CDEnumParameter)=> DeltaOperation =
193     operand:DeltaOperand CDEnumParameter ";";
194
195   DeltaCDEnumParameterScopeIdentifier implements
196     ScopeIdentifier = "CDEnumParameter";
197
198   CDEnumParameterIdentifier implements
199     ("[" CDEnumParameter "]")=> ModelElementIdentifier =
200     "[" CDEnumParameter "]";
201
202   DeltacDEnumParametersCDEnumParameterDeltaOperation implements
203     (DeltaOperand "cDEnumParameters")=> DeltaOperation =
204     operand:DeltaOperand "cDEnumParameters"
205     cDEnumParameters:CDEnumParameter ";";
206
207   DeltaCDEnumConstantOperation implements
208     (DeltaOperand CDEnumConstant)=> DeltaOperation =
209     operand:DeltaOperand CDEnumConstant;
210
211   DeltaCDEnumConstantScopeIdentifier implements ScopeIdentifier =
212     "CDEnumConstant";
213
214   CDEnumConstantIdentifier implements
215     ("[" CDEnumConstant "]")=> ModelElementIdentifier =
216     "[" CDEnumConstant "]";
217
218   DeltaTypeVariableDeclarationDeltaOperation implements
219     (DeltaOperand "TypeVariableDeclaration")=> DeltaOperation =
220     operand:DeltaOperand "TypeVariableDeclaration"
221     TypeVariableDeclaration ";";
222
223   DeltacDEnumConstantsCDEnumConstantDeltaOperation implements
224     (DeltaOperand "cDEnumConstants")=> DeltaOperation =
225     operand:DeltaOperand "cDEnumConstants"
226     cDEnumConstants:CDEnumConstant ";";
227
228   DeltaCDEnumOperation implements
229     (DeltaOperand CDEnum)=> DeltaOperation =
230     operand:DeltaOperand CDEnum;
231
232   DeltaCDEnumScopeIdentifier implements ScopeIdentifier =
233     "CDEnum";
234
235   CDEnumIdentifier implements
236     ("[" CDEnum "]")=> ModelElementIdentifier =
237     "[" CDEnum "]";
238
239   DeltaCDInterfaceOperation implements
240     (DeltaOperand CDInterface)=> DeltaOperation =
```

```
241     operand:DeltaOperand CDInterface;
242
243   DeltaCDInterfaceScopeIdentifier implements ScopeIdentifier =
244     "CDInterface";
245
246   CDInterfaceIdentifier implements
247     ("[" CDInterface "]")=> ModelElementIdentifier =
248     "[" CDInterface "]";
249
250   DeltaTypeParametersDeltaOperation implements
251     (DeltaOperand "TypeParameters")=> DeltaOperation =
252     operand:DeltaOperand "TypeParameters" TypeParameters ";";
253
254   DeltainterfacesReferenceTypeDeltaOperation implements
255     (DeltaOperand "interfaces")=> DeltaOperation =
256     operand:DeltaOperand "interfaces"
257     interfaces:Name ";";
258
259   DeltasuperclassesReferenceTypeDeltaOperation implements
260     (DeltaOperand "superclasses")=> DeltaOperation =
261     operand:DeltaOperand "superclasses"
262     superclasses:Name ";";
263
264   DeltaCDClassOperation implements
265     (DeltaOperand CDClass)=> DeltaOperation =
266     operand:DeltaOperand CDClass;
267
268   DeltaCDClassScopeIdentifier implements ScopeIdentifier =
269     "CDClass";
270
271   CDClassIdentifier implements
272     ("[" CDClass "]")=> ModelElementIdentifier =
273     "[" CDClass "]";
274
275   DeltaStereotypeDeltaOperation implements
276     (DeltaOperand "Stereotype")=> DeltaOperation =
277     operand:DeltaOperand "Stereotype" Stereotype ";";
278
279   DeltaModifierDeltaOperation implements
280     (DeltaOperand "Modifier")=> DeltaOperation =
281     operand:DeltaOperand "Modifier" Modifier ";";
282
283   DeltaCompletenessDeltaOperation implements
284     (DeltaOperand "Completeness")=> DeltaOperation =
285     operand:DeltaOperand "Completeness" Completeness ";";
286
287   DeltaCDDefinitionOperation implements
288     (DeltaOperand CDDefinition)=> DeltaOperation =
289     operand:DeltaOperand CDDefinition;
```

```
290
291   DeltaCDDefinitionScopeIdentifier implements ScopeIdentifier =
292     "CDDefinition";
293
294   CDDefinitionIdentifier implements
295     ("[" CDDefinition "]")=> ModelElementIdentifier =
296     "[" CDDefinition "]";
297 }
```

Listing C.10: Darstellung der vollständigen Grammatik der automatisiert generierten, klassendiagrammspezifischen Deltasprache.

C.11 Grammatik der erweiterten Deltasprache

MCG
EDCD

```
1  package mc.montiee.cd.delta;
2
3  grammar EDCD extends DCD, mc.umlp.cd.CD {
4
5    /*************************************************************
6     * options
7    *************************************************************/
8    options {
9      compilationunit DeltaCDDefinition
10     nostring
11     parser lookahead=5
12     lexer lookahead=7
13   }
14
15   ast DeltaCDDefinition astextends
16     /mc.umlp.cd._ast.ASTCDDefinition =
17     method public String getName() {
18     return getDelta().getName();
19   }
20   ;
21
22   /*************************************************************
23    * productions
24   *************************************************************/
25
26   /*************************************************************
27    * syntax adjustments
28   *************************************************************/
29   DeltaCDClassScopeIdentifier implements ScopeIdentifier =
30     "class";
31
```

```
32  DeltaCDAttributeScopeIdentifier implements ScopeIdentifier =
33    "attribute";
34
35  DeltaCDDefinitionScopeIdentifier implements ScopeIdentifier =
36    "classdiagram";
37
38  DeltaCDConstructorScopeIdentifier implements ScopeIdentifier =
39    "constructor";
40
41  DeltaCDAssociationScopeIdentifier implements ScopeIdentifier =
42    "association";
43
44  DeltaCDEnumScopeIdentifier implements ScopeIdentifier =
45    "enum";
46
47  DeltaCDEnumConstantScopeIdentifier implements ScopeIdentifier =
48    "enumconstant";
49
50  DeltaCDInterfaceScopeIdentifier implements ScopeIdentifier =
51    "interface";
52
53  DeltaCDMethodScopeIdentifier implements ScopeIdentifier =
54    "method";
55
56  /*************************************************************
57   * class operations
58  *************************************************************/
59  DeltaRemoveClassOperation implements
60    (DeltaRemove Flag "class" Name)=> DeltaOperation =
61    operand:DeltaRemove Flag "class" name:Name";";
62
63  DeltaRenameClassOperation implements
64    (DeltaRename "class" Name "to")=> DeltaOperation =
65     operand:DeltaRename "class" name:Name "to" newName:Name";";
66
67  DeltaAddSuperclassClassOperation implements
68    (DeltaAdd "superclass" Name)=> DeltaOperation =
69    operand:DeltaAdd "superclass" name:Name";";
70
71   DeltaRemoveSuperclassClassOperation implements
72    (DeltaRemove "superclass" Name)=> DeltaOperation =
73    operand:DeltaRemove "superclass" name:Name";";
74
75  DeltaAddInterfaceObjectOperation implements
76    (DeltaAdd "reference" Name)=> DeltaOperation =
77    operand:DeltaAdd "reference" name:Name";";
78
79  DeltaRemoveInterfaceObjectOperation implements
80    (DeltaRemove "reference" Name)=> DeltaOperation =
```

```
81    operand:DeltaRemove "reference" name:Name";";
82
83  /*************************************************************
84   * interface operations
85  *************************************************************/
86  DeltaRemoveInterfaceOperation implements
87    (DeltaRemove Flag "interface" Name)=> DeltaOperation =
88    operand:DeltaRemove Flag "interface" name:Name";";
89
90  DeltaRenameInterfaceOperation implements
91    (DeltaRename "interface" Name "to")=> DeltaOperation =
92    operand:DeltaRename "interface" name:Name
93    "to" newName:Name";";
94
95  /*************************************************************
96   * attribute operations
97  *************************************************************/
98  Initializer = " init " " with " fullQualifiedName : Name ";";
99
100 DeltaAddAttributeOperation implements
101   ( DeltaAdd CDAttribute )=> DeltaOperation =
102   operand:DeltaAdd CDAttribute Initializer?;
103
104 DeltaRenameAttributeOperation implements
105   (DeltaRename "attribute" Name "to")=> DeltaOperation =
106   operand:DeltaRename "attribute" name:Name
107   "to" newName:Name";";
108
109 /*************************************************************
110  * parameter, method and enum operations
111 *************************************************************/
112 DeltaRenameParameterOperation implements
113   (DeltaRename "parameter" Name "to")=> DeltaOperation =
114   operand:DeltaRename "parameter" name:Name
115   "to" newName:Name";";
116
117 DeltaRenameMethodOperation implements
118   (DeltaRename "method" Name "to")=> DeltaOperation =
119   operand:DeltaRename "method" name:Name "to" newName:Name";";
120
121 DeltaRemoveEnumOperation implements
122   (DeltaRemove Flag "enum" Name)=> DeltaOperation =
123   operand:DeltaRemove Flag "enum" name:Name";";
124
125 DeltaRenameEnumOperation implements
126   (DeltaRename "enum" Name "to")=> DeltaOperation =
127   operand:DeltaRename "enum" name:Name "to" newName:Name";";
128
129 DeltaRenameEnumConstantOperation implements
```

```
130     (DeltaRename "enumconstant" Name "to")=> DeltaOperation =
131     operand:DeltaRename "enumconstant" name:Name
132     "to" newName:Name";";
133
134   /*********************************************************
135    * association operations
136   *********************************************************/
137   DeltaRemoveAssociationOperation implements
138     (DeltaRemove Flag CDAssociation )=> DeltaOperation =
139     operand:DeltaRemove Flag CDAssociation;
140
141   DeltaRenameAssociationOperation implements
142     (DeltaRename "association" Name "to")=> DeltaOperation =
143     operand:DeltaRename "association" name:Name
144     "to" newName:Name";";
145
146   DeltaAddLeftModifierOperation implements
147     (DeltaSet "leftmodifier" Modifier)=> DeltaOperation =
148     operand:DeltaSet "leftmodifier" Modifier";";
149
150   DeltaAddRightModifierOperation implements
151     (DeltaSet "rightmodifier" Modifier)=> DeltaOperation =
152     operand:DeltaSet "rightmodifier" Modifier";";
153
154   DeltaSetRightReferenceNameOperation implements
155     (DeltaSet "rightreferencename" QualifiedName)
156     => DeltaOperation = operand:DeltaSet "rightreferencename"
157     rightReferenceName:QualifiedName";";
158
159   DeltaSetAssociationTypeOperation implements
160     (DeltaSet "associationtype" "association")=> DeltaOperation =
161     operand:DeltaSet "associationtype" "association"";";
162
163   DeltaSetAggregationTypeOperation implements
164     (DeltaSet "associationtype" "aggregation")=> DeltaOperation =
165     operand:DeltaSet "associationtype" "aggregation"";";
166
167   DeltaSetCompositionTypeOperation implements
168     (DeltaSet "associationtype" "composition")=> DeltaOperation =
169     operand:DeltaSet "associationtype" "composition"";";
170
171   DeltaSetAssociationLeftToRightOperation implements
172     (DeltaSet "associationdirection" "->")=>DeltaOperation =
173     operand:DeltaSet "associationdirection" "->"";";
174
175   DeltaSetAssociationRightToLeftOperation implements
176     (DeltaSet "associationdirection" "<-")=>DeltaOperation =
177     operand:DeltaSet "associationdirection" "<-"";";
178
```

```
179  DeltaSetAssociationbidirectionalOperation implements
180    (DeltaSet "associationdirection" "<->")=>DeltaOperation =
181    operand:DeltaSet "associationdirection" "<->"";";
182
183  DeltaSetAssociationSimpleOperation implements
184   (DeltaSet "associationdirection" "--")=>DeltaOperation =
185    operand:DeltaSet "associationdirection" "--"";";
186
187  DeltaSetLeftReferenceNameOperation implements
188    (DeltaSet "leftreferencename" QualifiedName)
189    => DeltaOperation = operand:DeltaSet "leftreferencename"
190    leftReferenceName:QualifiedName";";
191
192  DeltaSetLeftRoleNameOperation implements
193    (DeltaSet "leftrolename" Name)=> DeltaOperation =
194    operand:DeltaSet "leftrolename" LeftRoleName:Name";";
195
196  DeltaSetRightRoleNameOperation implements
197    (DeltaSet "rightrolename" Name)=> DeltaOperation =
198    operand:DeltaSet "rightrolename" RightRoleName:Name";";
199
200 DeltaSetLeftCardinalityOperation implements
201    (DeltaSet "leftcardinality" Cardinality)
202    => DeltaOperation = operand:DeltaSet "leftcardinality"
203    LeftCardinality:Cardinality";";
204
205 DeltaSetRightCardinalityOperation implements
206    (DeltaSet "rightcardinality" Cardinality)
207    => DeltaOperation = operand:DeltaSet "rightcardinality"
208    RightCardinality:Cardinality";";
209
210  /*************************************************************
211   * move operation
212  *************************************************************/
213  DeltaMoveAttributeOperation implements
214    (DeltaMove "attribute" Name)=> DeltaOperation =
215    operand:DeltaMove "attribute" attributeName:Name
216    oldClass:Name newClass:Name";";
217
218  DeltaMoveMethodOperation implements
219    (DeltaMove "method" Name)=> DeltaOperation =
220    operand:DeltaMove "method" methodName:Name
221    oldClass:Name newClass:Name";";
222
223  /*************************************************************
224   * merge operation
225  *************************************************************/
226  DeltaMergeClassOperation implements
227    (DeltaMerge "class" Name)=> DeltaOperation =
```

```
228     operand:DeltaMerge "class" superClass:Name
229     subClass:Name "to" CDClass;
230
231   /*************************************************************
232    * extract operation
233   *************************************************************/
234   DeltaExtractClassOperation implements
235     (DeltaPull "class" Name)=> DeltaOperation =
236     operand:DeltaPull "class" className:Name
237     "{" (("method" methodName:Name ";") |
238     ("attribute" attributeName:Name ";"))*
239     "}" "as" CDClass;
240
241   DeltaExtractInterfaceOperation implements
242     (DeltaPull "interface" Name)=> DeltaOperation =
243     operand:DeltaPull "interface" className:Name
244     "{" (("method" methodName:Name ";") |
245     ("attribute" attributeName:Name ";"))*
246     "}" "as" CDInterface;
247
248   DeltaExtractSubClassOperation implements
249     (DeltaPull "subclass" Name)=> DeltaOperation =
250     operand:DeltaPull "subclass" className:Name
251     "{" (("method" methodName:Name ";") |
252     ("attribute" attributeName:Name ";"))*
253     "}" "as" CDClass;
254
255   DeltaExtractSuperClassOperation implements
256     (DeltaPull "superclass" Name)=> DeltaOperation =
257     operand:DeltaPull "superclass" className:Name
258     "{" (("method" methodName:Name ";") |
259     ("attribute" attributeName:Name ";"))*
260     "}" "as" CDClass;
261
262   DeltaPullUpAttributeOperation implements
263     (DeltaPullUp "attribute" "to" "from" Name)
264     => DeltaOperation = operand:DeltaPullUp "attribute"
265     attributeName:Name "from" subClasses:Name*
266     "to" superClass:Name";";
267
268   DeltaPullUpMethodOperation implements
269     (DeltaPullUp "method" "to" "from" Name)
270     => DeltaOperation = operand:DeltaPullUp "method"
271     methodName:Name "from" subClasses:Name*
272     "to" superClass:Name";";
273
274   DeltaPushDownAttributeOperation implements
275     (DeltaPushDown "attribute" "from" "to" Name)
276     => DeltaOperation = operand:DeltaPushDown "attribute"
```

```
277     attributeName:Name "from" superClass:Name
278     "to" subClass:Name";";
279
280   DeltaPushDownMethodOperation implements
281     (DeltaPushDown "method" "from" "to" Name)
282     => DeltaOperation = operand:DeltaPushDown "method"
283     methodName:Name "from" superClass:Name
284     "to" subClass:Name";";
285 }
```

Listing C.11: Darstellung der vollständigen Grammatik der erweiterten, klassendiagrammspezifischen Deltasprache.

Abbildungsverzeichnis

Listingsverzeichnis

Tabellenverzeichnis

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design

guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe

variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.