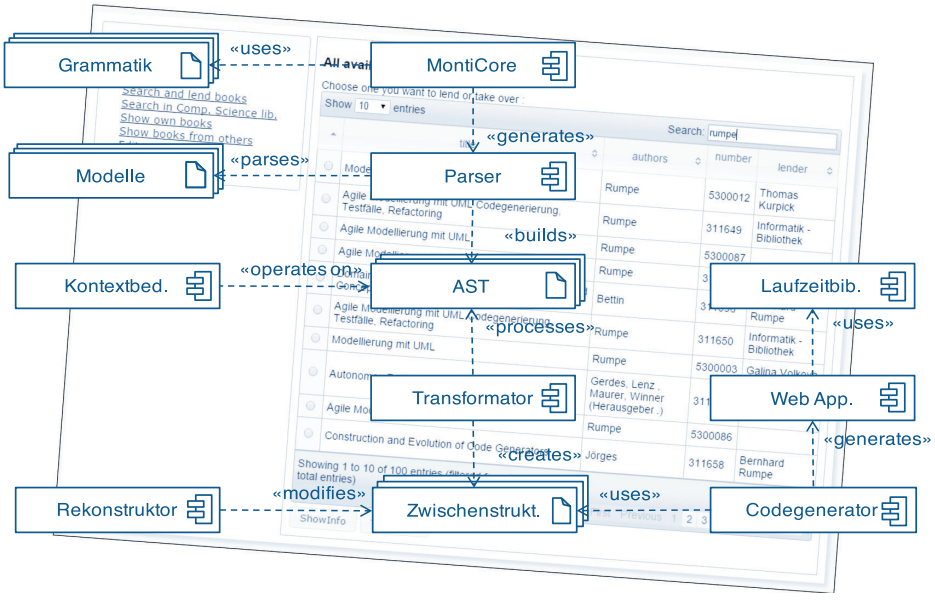
SE | RWTH AACHEN UNIVERSITY

Dirk Reiß

Modellgetriebene generative Entwicklung von Web-Informationssystemen



Aachener Informatik-Berichte,
Software Engineering

Band 22

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

Modellgetriebene generative Entwicklung von Web-Informationssystemen

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-Wirtschaftsinformatiker
Dirk Reiß
aus Braunschweig

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Albert Zündorf

Tag der mündlichen Prüfung: 7. Dezember 2015

[Rei16] D. Reiß:
Modellgetriebene generative Entwicklung von Web-Informationssystemen.
Shaker Verlag, ISBN 978-3-8440-4446-1. Aachener Informatik-Berichte, Software Engineering, Band 22. 2016.
www.se-rwth.de/publications/

Kurzfassung

Die Entwicklung von Web-Informationssystemen, wie sie heutzutage an vielen Stellen im Netz zu finden sind, erfordert auch mit dem Einsatz aktueller Technologien immer noch einen hohen Anteil an repetitiven Tätigkeiten, die nicht nur aufwendig, sondern in der Regel auch fehleranfällig sind. Durch die Identifikation abstrahierbarer Aspekte der Anwendung und deren Abbildung auf Modelle können durch Verwendung des Konzepts der generativen Softwareentwicklung große Teile der Applikation automatisch erzeugt werden. Um möglichst flexibel hinsichtlich modellierbarer Domänen und Anwendungsgebiete zu sein, müssen neben der reinen Abstraktion auf Modelle jedoch auch Mechanismen vorhanden sein, um komplexe Geschäftslogik in die Anwendung integrieren zu können. Die identifizierten modellierbaren Kernaspekte des Web-Informationssystems betreffen die zugrunde liegende Datenstruktur, die Darstellung der verarbeiteten Daten inklusive einfacher aber flexibler Gestaltungsmöglichkeiten, die Berechtigungsstruktur innerhalb der Anwendung und die damit einhergehenden unterschiedlichen Sichten auf die Applikation, die Abfolge von Seitenaufrufen sowie einzel- und mehrbenutzerfähige Abläufe zur Abbildung der Geschäftslogik.

Im Rahmen dieser Arbeit wurden die beschriebenen Aspekte auf eine Menge textueller Modelle abgebildet, die zur Beschreibung und Generierung eines voll funktionsfähigen Web-Informationssystems verwendet werden. Diese sind zum Teil dem UML-Profil UML/P entnommen, zum Teil eine Erweiterung dieses Profils um ein Profil der UML-Aktivitätsdiagramme und zum Teil vollkommen eigenständige, domänenspezifische Sprachen. Die notwendige Flexibilität wird durch die Verwendung von Java als Aktionssprache innerhalb der Aktionen der Aktivitätsdiagramme gewährleistet. Das in dieser Arbeit entwickelte System setzt auf dem MontiCore-Framework auf und nutzt seine Möglichkeiten zur modularen Sprachdefinition für die Prüfung von Abhängigkeiten zwischen den Modellen.

Die Kombination dieser Sprachen erlaubt es sehr agil, auf Basis minimaler Modelle und durch ausgeprägtes, hinzugeneriertes Standardverhalten Prototypen zu erstellen, bietet aber auch genügend Freiheiten zur Modellierung komplexer Anwendungen verschiedenster Anwendungsdomänen.

Abstract

Even though the development of web information systems is supported by a number of actual web frameworks, it still requires a lot of redundant steps that are tedious and error prone in most cases. By identifying common aspects of a web information system and their mapping to models, we can apply mechanisms of generative software development to create major portions of the application automatically. In order to use the approach in various application domains, we further need means to integrate complex business logic into the description of the system. The identified core aspects of a web information system comprise the underlying data structure, the presentation of data including simple but flexible page design, the user rights management within the application alongside with the resulting different views for different user groups, as well as the page- and workflow for the specification of business logic.

The described aspects are mapped to a set of textual modeling languages which are used to describe and generate a complete web information system. Some of these languages are part of the UML profile UML/P, the used UML activity diagrams extend this profile and some of the developed languages are completely standalone and domain specific. In order to achieve the necessary flexibility, Java is used as an action language within the activity diagrams. The languages and generators are implemented based on the MontiCore framework, whose functionalities for modular language definitions are used to verify interdependency of the different models.

The combination of the developed languages is used during the agile development of web information systems, where a minimal set of models and extensive default behavior leads to full-fledged prototypes while iterative extensions allow the generation of complex applications from different problem domains.

Danksagung

Der lange Weg zu meiner Promotion war keiner, den ich allein beschritten habe. Vielmehr wurde ich dabei von mehreren Menschen begleitet, bei denen ich mich an dieser Stelle ganz herzlich bedanken möchte.

Zuallererst gilt mein ganz besonderer Dank Herrn Prof. Dr. Bernhard Rumpe für die Betreuung dieser Dissertation, die er auch nach seinem Wechsel von der TU Braunschweig an die RWTH Aachen weitergeführt hat. Durch das Ermöglichen der Teilnahme an den regelmäßigen Promotionsworkshops und die effektiven Diskussionen sowohl mit ihm als auch den Kollegen ergaben sich wertvolle Hinweise und Sichtweisen, die in diese Arbeit eingeflossen sind.

Ich danke Herrn Prof. Dr. Albert Zündorf für die Übernahme des Zweitgutachtens meiner Arbeit und den Herren Prof. Dr. Martin Grohe und Prof. Dr. Klaus Wehrle für die Bereitschaft, als Mitglieder der Promotionskommission meine Prüfung durchzuführen.

Meinen Kollegen an der RWTH Aachen und der TU Braunschweig verdanke ich eine wunderbare und spannende gemeinsame Zeit. Bezogen auf diese Arbeit sind jedoch einige von ihnen besonders hervorzuheben. Herrn Dr. Holger Krahn, Herrn Dr. Steven Völkel und Herrn Dr. Martin Schindler danke ich für die Grundlagen, die sie im Rahmen ihrer Promotionen gelegt haben und auf denen die vorliegende Arbeit in Teilen aufsetzt. Dank gebührt ebenfalls den Herren Mark Stein und Michael Dukaczewski für die fruchtbaren Diskussionen, die zu Beginn in Vorarbeiten zu dieser Arbeit in Kooperation stattfanden, sich dann aber in unterschiedliche Richtungen entwickelten. Letzterem danke ich herzlich für seine überaus hilfreichen Anmerkungen und die Durchsicht dieser Arbeit. Den Herren Thomas Kurpick, Ludger Steens, Robert Eikermann sowie Dominik Merkelbach gilt mein Dank für die Nutzung von MontiWIS in ihren eigenen Arbeiten und den Herren Florian Leppers, Jerome Pfeiffer und Philipp Schütze für die Betreuung des mit MontiWIS entwickelten Bibliothekssystems. Sie alle lieferten gewinnbringende Anmerkungen, die zu sinnvollen Anpassungen in dieser Arbeit geführt haben.

Abschließend danke ich meinen Eltern, meinem Bruder und meiner Schwägerin sowie all meinen Freunden für jegliche Unterstützung und den Rückhalt, die diese Arbeit überhaupt erst möglich gemacht haben.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Motivation . . . 2
- 1.2 Ziele und Anforderungen . . . 3
- 1.3 Aufbau der Arbeit . . . 5

2 Web-Informationssysteme **7**
- 2.1 Einordnung und Abgrenzung . . . 7
- 2.2 Anforderungen an Web-Informationssysteme . . . 8
  - 2.2.1 Funktionale Anforderungen . . . 8
  - 2.2.2 Qualitative (nicht-funktionale) Anforderungen . . . 11
- 2.3 Web Engineering allgemein und Ansätze der Implementierung . . . 11
  - 2.3.1 Manuelle Implementierung . . . 12
  - 2.3.2 Erhöhte Abstraktion durch den Einsatz von Frameworks . . . 14
  - 2.3.3 Generative Ansätze . . . 16
- 2.4 Durchgängiges Beispiel . . . 16

3 Modellgetriebene generative Softwareentwicklung **19**
- 3.1 Begriffsbildung und Abgrenzungen . . . 19
- 3.2 Werkzeuge für die modellgetriebene Entwicklung . . . 21
  - 3.2.1 Grafische Ansätze . . . 22
  - 3.2.2 Textuelle Ansätze . . . 22
- 3.3 MontiCore . . . 23
  - 3.3.1 Grundlagen der Entwicklung mit MontiCore . . . 23
  - 3.3.2 Symboltabelleninfrastruktur . . . 24
- 3.4 Anforderungen an die Modelle und die Infrastruktur zur Modellverarbeitung 26
  - 3.4.1 Zu modellierende Aspekte . . . 26
  - 3.4.2 Anforderungen an die technische Umsetzung der Sprachen . . . 30
- 3.5 Verwandte Arbeiten im Bereich des modellbasierten Web Engineerings . . . 30

4 Ablauf der Generierung **39**
- 4.1 Gesamtüberblick über den Generierungsprozess . . . 39
  - 4.1.1 Überblick über beteiligte Komponenten . . . 39
  - 4.1.2 Zusammenspiel der Komponenten . . . 41
- 4.2 Sprachverarbeitung mit MontiCore . . . 43

4.3 Verwendung von Symboltabelle und Kontextbedingungen 44
4.4 Verwendung der Zwischenstruktur 45
4.5 Rekonstruktion von Zwischenstrukturelementen 46
4.6 Generierung des Web-Informationssystems 46

5 Generiertes System **49**
5.1 Überblick über das generierte Web-Informationssystem 49
5.2 Datenstruktur und Persistenz 51
5.2.1 Datenklassen 51
5.2.2 Data Access Object Klassen 51
5.3 Nutzerinteraktion und clientseitige Technologien 52
5.3.1 Generelle Funktionsprinzipien 52
5.3.2 JSP Seiten 53
5.3.3 PageEntities 54
5.3.4 Ressourcenklassen 54
5.4 Ablaufsteuerung 55
5.4.1 Generierte Knotenstruktur 56
5.4.2 Kontextklassen zur Speicherung des Ablaufzustands 56
5.4.3 Executorklassen 56
5.4.4 Ressourcenklassen für Abläufe 57
5.5 Applikationssteuerung 57
5.5.1 Menüs 57
5.5.2 Nutzerverwaltung 57

6 Datenstruktur **59**
6.1 Grundlagen und Anforderungen 59
6.1.1 Grundlagen der Datenmodellierung 59
6.1.2 Anforderungen an die Datenmodellierungssprache 59
6.2 Sprache 60
6.2.1 Aufbau der Klassendiagrammsprache 61
6.2.2 Verwendung der Klassendiagrammsprache 63
6.2.3 Vordefinierte Datentypen 69
6.2.4 Profil für MontiWIS und Modellierungsrichtlinien 72
6.3 Generierungsprozess 77
6.3.1 Kontextbedingungen 78
6.3.2 Aufbau der Zwischenstruktur 78
6.3.3 Modellverarbeitung durch einen Visitor 83
6.3.4 Rekonstruktion der Klassendiagramminformationen 86
6.3.5 Erzeugung von Standardseiten für jede Klasse 87
6.3.6 Generator 90
6.4 Implementierung im Zielsystem 94
6.4.1 Datenstruktur und Persistenzmechanismen 94
6.4.2 Generierung der Standardseiten 101
6.5 Andere Ansätze 106

6.5.1 Andere Ansätze der Implementierung . . . 106
6.5.2 Andere Ansätze der Datenmodellierung . . . 107

7 Seitenbeschreibung 109
7.1 Grundlagen und Anforderungen . . . 109
7.1.1 Grundlagen . . . 109
7.1.2 Anforderungen an die Seitenbeschreibungssprache . . . 110
7.2 Sprache . . . 112
7.2.1 Grammatik der Seitenbeschreibungssprache . . . 113
7.2.2 Verwendung der Seitenbeschreibungssprache . . . 116
7.2.3 Darstellung im generierten System . . . 128
7.3 Generierungsprozess . . . 137
7.3.1 Kontextbedingungen . . . 137
7.3.2 Aufbau der Zwischenstruktur . . . 139
7.3.3 Modellverarbeitung durch einen Visitor . . . 144
7.3.4 Rekonstruktion des Seitenaufbaus . . . 144
7.3.5 Generator . . . 146
7.4 Implementierung im Zielsystem . . . 152
7.4.1 Generelle Seitenarchitektur . . . 152
7.4.2 Umsetzung der verschiedenen Datentypen . . . 157
7.4.3 PageEntity- und PageEntityInitializer-Klassen . . . 163
7.4.4 Ressourcenklassen für Seiten . . . 167
7.4.5 Sicherheitsüberlegungen . . . 168
7.5 Andere Ansätze . . . 170
7.5.1 Imperativer Ansatz . . . 170
7.5.2 Deklarativer Ansatz . . . 172

8 Ablaufsteuerung 173
8.1 Grundlagen und Anforderungen . . . 173
8.1.1 Grundlagen . . . 173
8.1.2 Anforderungen an die Ablaufsteuerung . . . 173
8.2 Sprache . . . 176
8.2.1 Aufbau der Aktivitätsdiagrammsprache . . . 176
8.2.2 Verwendung der Aktivitätsdiagrammsprache . . . 181
8.2.3 Beschreibung der einzelnen Sprachelemente . . . 184
8.3 Inhalt von Aktionen . . . 195
8.3.1 Parameter . . . 196
8.3.2 Variablen . . . 197
8.3.3 Aufruf einer Seite . . . 197
8.3.4 Java . . . 198
8.3.5 Kommandos . . . 200
8.3.6 Aufruf anderer Aktivitäten . . . 208
8.4 Generierungsprozess . . . 209
8.4.1 Kontextbedingungen . . . 210

8.4.2 Aufbau der Zwischenstruktur . . . 213
8.4.3 Modellverarbeitung von Aktivitätsdiagrammen durch einen Visitor . 216
8.4.4 Rekonstruktion der Aktivitätsdiagrammelemente . . . 220
8.4.5 Generator . . . 221
8.5 Umsetzung im Zielsystem . . . 225
8.5.1 Übersicht über die Laufzeitumgebung . . . 225
8.5.2 Ausführung von Abläufen . . . 234
8.5.3 Weitere Laufzeitkomponenten . . . 241
8.6 Andere Ansätze . . . 241
8.6.1 Workflowbeschreibungssprachen . . . 241
8.6.2 Web-Modellierungsansätze . . . 243

9 Applikationssprache 245
9.1 Grundlagen und Anforderungen . . . 245
9.1.1 Grundlagen . . . 245
9.1.2 Anforderungen an die Applikationssprache . . . 245
9.2 Sprache . . . 247
9.2.1 Aufbau der Applikationssprache . . . 247
9.2.2 Verwendung der Applikationssprache . . . 249
9.3 Generierungsprozess . . . 253
9.3.1 Kontextbedingungen . . . 254
9.3.2 Aufbau der Zwischenstruktur . . . 255
9.3.3 Modellverarbeitung durch einen Visitor . . . 257
9.3.4 Rekonstruktion . . . 258
9.3.5 Generator . . . 258
9.4 Umsetzung im generierten System . . . 262
9.4.1 Umsetzung des Menüs . . . 262
9.4.2 Umsetzung der Rechte . . . 264
9.5 Andere Ansätze . . . 267
9.5.1 Möglichkeiten der Zugriffssteuerung . . . 267
9.5.2 Methoden zur Spezifikation von Menüs in Webanwendungen . . . 268

10 Anwendungsmethodik 271
10.1 Überblick über MontiWIS-Anwendungsprojekte . . . 271
10.1.1 Anlegen eines neuen Projekts . . . 271
10.1.2 Aufbau eines MontiWIS-Anwendungsprojektes . . . 272
10.2 Durchführung der Generierung . . . 276
10.2.1 Lokale Vorbedingungen . . . 277
10.2.2 Lokale Generierung . . . 277
10.2.3 Generierung über OSTP . . . 278
10.2.4 Schritte nach der Generierung . . . 278
10.3 Iterative Entwicklung des Web-Informationssystems . . . 279
10.3.1 Festlegen von Anwendungsfällen und Akteuren . . . 280
10.3.2 Erweiterung der Datenstruktur . . . 280

10.3.3 Erstellen von Aktivitäten 280
10.3.4 Spezifikation von Seiten 281
10.4 Anpassung / Erweiterung der Implementierung 281
10.4.1 Umbenennen von Buttons und Standardbezeichnungen 281
10.4.2 CSS-Anpassungen 283

11 Fallstudien und Bewertung **285**
11.1 Bibliotheksverwaltung 285
11.1.1 Datenmodell 285
11.1.2 Aktivitäten 285
11.1.3 Applikationsmodell 287
11.2 Modulhandbuch 288
11.2.1 Datenmodell 288
11.2.2 Applikationsmodell 288
11.2.3 Aktivitäten 289
11.3 Robotersteuerung 291
11.3.1 Datenmodell 292
11.3.2 Aktivitäten 292
11.3.3 Applikationsmodell 292
11.4 Weitere Verwendung 292
11.4.1 Feedback bezüglich statischen Komponenten von MontiWIS 292
11.4.2 Feedback bezüglich dynamischer Komponenten von MontiWIS 293
11.4.3 Online-Demonstrator 293
11.5 Bewertung der Ergebnisse 293
11.5.1 Quantitative Bewertung 294
11.5.2 Qualitative Bewertung 295
11.5.3 Modell- und Infrastrukturanforderungen 297
11.5.4 Bewertung der Designentscheidungen 298

12 Zusammenfassung und Ausblick **301**
12.1 Zusammenfassung 301
12.2 Ausblick 303

Literaturverzeichnis **305**

Abbildungsverzeichnis **325**

Tabellenverzeichnis **329**

A Glossar **331**

B Abkürzungen **335**

C Grammatiken **339**
C.1 MontiCore Literals 339

C.2 MontiCore Types 347
C.3 UML/P Common 353
C.4 Java 357
C.5 Klassendiagramme 381
C.6 Seitenbeschreibung 387
C.7 Aktivitätsdiagramme 393
C.8 Applikationsprache 403

D Lebenslauf **407**

Kapitel 1.

Einführung

Am Institut für Software Systems Engineering der Technischen Universität Braunschweig galt es, im Rahmen des Projekts *StuBS - EDV in Studium und Lehre* [Stu, RRSQS08] die Einführung von Bachelor- und Masterstudiengängen durch den Einsatz von Software zu unterstützen bzw. erst mit vertretbarem Aufwand zu ermöglichen. Dabei kristallisierte sich schnell heraus, dass sich die anfallenden Aufgaben in diesem Projekt im Wesentlichen in zwei Bereiche unterteilen ließen.

Einerseits musste bestehende Software, deren Einsatz aufgrund gesetzlicher Rahmenbedingungen und aus Kompatibilitätsgründen obligatorisch war, so angepasst werden, dass die neuen Anforderungen und Gegebenheiten in diesen Systemen abbildbar waren.

Andererseits existierten viele Aufgabenbereiche, für die keine geeignete Software zur Verfügung stand. Die benötigten Funktionalitäten wurden entweder gar nicht oder nur unzureichend unterstützt. In diesen Fällen mussten folglich die spezifischen Anforderungen gesammelt und in Software umgesetzt werden, was einem klassischen Softwareentwicklungsprojekt gleichkam. Im Rahmen des Projekts wurden daher verschiedene webbasierte Anwendungen entwickelt. Darunter befanden sich beispielsweise ein System zur Verwaltung von Studiengängen, Modulen und Lehrveranstaltungen, eine Anwendung zur Unterstützung der Stipendienvergabe sowie eine Applikation, mit der Studienbeitragsmittel verwaltet und zugeteilt werden konnten. Bei der Entwicklung kamen unterschiedliche Technologien wie beispielsweise Apache Struts [Str, CG04] zusammen mit der Apache Velocity Template Engine [Vel] oder auch Apache Tapestry [Tap] zum Einsatz.

Da während dieser Zeit einiges Verständnis hinsichtlich der Gemeinsamkeiten und Charakteristika der erstellten Anwendungen entwickelt wurde, lag es nahe, die am Institut vorhandenen Werkzeuge und Methoden zu verwenden, um darüber mit Hilfe geeigneter Sprachkonstrukte ähnliche Web-Informationssysteme mit deutlich geringerem Aufwand spezifizieren und generieren zu können, als dies bei vollständig manueller Implementierung der Fall ist. Das in dieser Arbeit entwickelte System wird im Folgenden *MontiWIS* genannt[1], wobei *Monti* vom verwendeten Framework MontiCore herrührt und *WIS* sich als Akronym vom Begriff *Web-Informationssystem* ableitet.

Der folgende Abschnitt dient als Einführung in die Thematik der vorliegenden Arbeit und soll einen Überblick darüber geben. In Abschnitt 1.1 wird in die Frage- und Problemstellung der Arbeit eingeführt und die Notwendigkeit und Sinnhaftigkeit des entwickelten Ansatzes

[1] MontiWIS als System werden in dieser Arbeit die Sprachen, die Komponenten zur Sprachverarbeitung, die Generatoren sowie die zusätzlich entwickelten Laufzeitbibliotheken bezeichnet.

aufgezeigt. Im folgenden Abschnitt 1.2 werden die Ziele der Arbeit im Einzelnen herausgestellt und erste Anforderungen an die Sprachen formuliert, die das zu entwickelnde System beschreiben. Abschnitt 1.3 schließlich gibt einen Überblick über den weiteren Aufbau der Arbeit.

1.1. Motivation

Web-Informationssysteme lassen sich traditionell von Grund auf manuell beispielsweise mit den im vorherigen Abschnitt 1 erwähnten Technologien, aber auch prinzipiell jeder anderen Programmiersprache entwickeln, die auf einem Server läuft bzw. die Anfragen per Hypertext Transfer Protocol (HTTP) bedienen kann. Müßig bei manuellen Implementierungen ist allerdings, dass für jede Anwendung stets die gleichen Schritte nötig sind und dass diese in der Regel aufwendig, fehleranfällig und von relativ schematischer Natur sind. Die Datenstruktur muss in jeder Applikation aufs Neue definiert werden, ähnlich strukturierte, der Datenstruktur angepasste Seiten zur Bearbeitung müssen erstellt und Geschäftslogik zur Navigation innerhalb der Applikation und zur Verarbeitung der Daten muss geschrieben werden. Zusätzlich sind Aspekte wie Zugriffsberechtigungen zu definieren und konsequent und konsistent über alle Seiten und Applikationsteile hinweg zu implementieren. Zwar bieten spezielle Frameworks zur Entwicklung von Web-Anwendungen (siehe [Jaz07, SH06] für einen Überblick über einige (Java-basierte) Ansätze) beliebige Flexibilität sowohl bei der Gestaltung der Seiten als auch der bei Implementierung der Geschäftslogik, jedoch geht mit dieser Freiheit auch der erhöhte Aufwand einher, der für die Umsetzung der einzelnen Teile der Applikation notwendig ist.

Der Vorteil von modellbasierten Werkzeugen, wie sie beispielsweise mit MontiCore [Kra10, KRV08] am Institut für Software Engineering entwickelt werden, besteht in dem Produktivitätsgewinn und der letztlichen Zeitersparnis, von denen bei der Entwicklung mehrerer ähnlich gearteter Systeme profitiert werden kann. Zusätzlich wird angestrebt, im Sinne der UML/P [Rum11, Rum12] *programmier-geeignet* zu sein und somit die Generierung vollständiger Anwendungen zu erlauben. Ähnlich wie in der UML/P sollen auch in MontiWIS Java-Codestücke verwendet werden, die sowohl die intuitive Spezifikation von Geschäftslogik als auch die einfache Einbindung extern definierter Codeteile ermöglichen sollen. Die Anlehnung an diese Sprachfamilie bedingt jedoch nicht, dass lediglich bereits existierende Sprachen verwendet werden oder ausschließlich Sprachen der UML zum Einsatz kommen. In Fällen, wo dies sinnvoll erscheint, wird darauf zurückgegriffen, für die Modellierung anderer Aspekte jedoch wird domänenspezifischen Lösungsansätzen der Vorzug gegeben.

Ebenfalls in Anlehnung an den UML/P Entwicklungsansatz ist die in MontiWIS angestrebte agile Entwicklungsmethodik [BBvB$^+$] zu sehen. Insbesondere auf die frühe Auslieferung lauffähiger Software und die iterative Anpassung der Anwendung wird dabei Wert gelegt. Durch die Unterstützung umfangreichen Standardverhaltens und der Möglichkeit, auch aus minimalen Modellen bereits ein lauffähiges Web-Informationssystem generieren zu können, lässt sich prinzipiell direkt von Beginn eines Projekts an eine prototypische Anwendung generieren, die als Grundlage für weitere Entwicklungen dient. Anhand dieser Applikation lassen sich iterativ in enger Kooperation mit potenziellen Auftraggebern oder Kunden Ände-

rungen besprechen, die umgehend in Modellen umgesetzt und in der generierten Anwendung evaluiert werden können. Das generierte Standardverhalten kann als Teil der Applikation Bestand haben, während speziellere Logik hinzugefügt oder das ursprüngliche Verhalten (partiell) ersetzt wird.

Dieses iterative Vorgehen im Sinne einer agilen Methodik impliziert auch, dass keine explizite a priori Anforderungsanalyse gefordert ist, sondern dass direkt mit der Modellierung und damit der Spezifikation des Systems begonnen werden kann. Unvollständige Modelle sind dabei explizit möglich und erwünscht. An Stellen, an denen zu Beginn noch keine vollständige Implementierung vorhanden ist, kann über Kommentare im Modellcode eine informelle Spezifikation vorgenommen werden.

Einen Überblick über die Anforderungen an die Modelle gibt der folgende Abschnitt.

1.2. Ziele und Anforderungen

Die Forschungsfrage, die im Rahmen dieser Arbeit beantwortet werden soll, lässt sich auf den folgenden Satz reduzieren:

> *"Wie lassen sich Web-Informationssysteme effizient und flexibel mit Modellen beschreiben und daraus möglichst vollständig generieren?"*

Ziel dieser Arbeit ist damit die Entwicklung einer Methodik und einer Menge von Sprachen, über die sich Web-Informationssysteme beschreiben lassen. Der Begriff *effizient* impliziert dabei, dass sich das System signifikant kompakter beschreiben lässt, als dies bei einer rein manuellen Implementierung der Fall wäre. *Flexibel* bedeutet in diesem Kontext, dass sich eine Vielzahl von Systemen mit unterschiedlichen Anwendungsdomänen beschreiben lassen, wobei die Sprachen selbst einiges an Flexibilität bieten sollten und zusätzlich die Einbindung beliebigen Codes möglich ist[2]. Auch das Vorgehen bei der Modellierung ist dabei insofern flexibel, als keine strikte Reihenfolge bei der Entwicklung festgelegt ist, sondern dass einzelne Aspekte der Anwendung (mehr oder weniger) unabhängig voneinander modelliert werden können und dabei dennoch ein lauffähiges System entsteht.

Die Aspekte, die bei der Beschreibung eines Web-Informationssystems berücksichtigt werden müssen, lassen sich wie folgt einordnen.

Datenstruktur Da in einem Web-Informationssystem primär Daten verwaltet werden, muss eine Möglichkeit existieren, die Datenstruktur zu spezifizieren. Für diese Daten sollten direkt Eingabemöglichkeiten geschaffen werden, um sie bearbeiten, speichern und verwalten zu können.

Seiten in der Applikation In einem Web-Informationssystem dienen Webseiten als Schnittstelle zwischen Nutzer und Anwendung. Dementsprechend muss es Beschreibungsmittel geben, um Seiten zu spezifizieren und darin festzulegen, wie (Teil-)Datensätze dargestellt oder bearbeitet werden sollen sowie welche statischen Inhalte auf der Webseite präsentiert werden.

[2] Hierbei ist die Einschränkung zu akzeptieren, dass es sich bei dem Code um die Zielsystemsprache handelt, was in diesem Fall Java ist.

Ablaufsteuerung und Geschäftslogik Die Abfolge von Seiten muss ebenfalls über Sprachkonstrukte beschreibbar sein. Dabei müssen Daten von einer Seite zur nächsten übergeben werden können, so dass ihre Verarbeitung schrittweise vor und nach der Anzeige einer Seite erfolgen kann. Zusätzlich sollte quasi beliebige Geschäftslogik eingebunden und ausgeführt werden können, um so der Anforderung der Flexibilität Rechnung zu tragen.

Berechtigungen verschiedener Nutzergruppen Da in einem Web-Informationssystem in der Regel verschiedene Nutzer agieren, die unterschiedliche Sichten auf das System besitzen und denen Zugriff auf bestimmte Teile der Anwendung verwehrt bleibt, müssen Sprachmittel vorhanden sein, um Nutzergruppen und die ihnen zugeordneten Sichten zu beschreiben.

Komposition der Anwendung Die verschiedenen Seiten und Abläufe sollten in einfacher Art und Weise über Menüstrukturen komponiert werden können, so dass die Nutzer der Anwendung darauf zugreifen können. Dazu müssen Sprachmittel und entsprechende Funktionalität entwickelt werden.

Neben den Anforderungen an die Sprachen, über die sich das Web-Informationssystem beschreiben lässt, sollten folgende Aspekte Beachtung finden.

Unterstützung bei der Modellierung Bei der Modellierung sollte dem Nutzer grundlegende Unterstützung wie beispielsweise Syntax-Highlighting in einem Texteditor und das Anzeigen von Warnhinweisen bei offensichtlichen Problemen während der Bearbeitung der Modelle zur Verfügung stehen. Durch die Verwendung verschiedenster Kontextbedingungen lassen sich nicht nur syntaktische Fehler in Modellen finden, sondern auch Probleme hinsichtlich ihrer Sinnhaftigkeit im Kontext mit anderen Modellen entdecken und aufzeigen.

Unterstützung iterativer Methoden Um einen agilen Entwicklungsprozess in mehreren Iterationen zu unterstützen, sollte die Entwicklung der Applikation in einer Art möglich sein, in der bestehende Daten weiterverwendet werden können, während weitere Modelle hinzugefügt oder bestehende Modelle verändert werden.

Generierung umfangreichen Standardverhaltens Ebenfalls im Kontext eines agilen, leichtgewichtigen Prozesses ist die Anforderung zu sehen, umfangreiches Standardverhalten bereitzustellen. Dadurch soll es möglich sein, bereits aus sehr rudimentären Modellen vollständig lauffähige Systeme zu generieren. Neben der standardmäßigen Generierung von vollständiger Standardfunktionalität zur Bearbeitung von Daten sollte es auch an anderen Stellen möglich sein, mit sehr kompakten Modellelementen auszukommen, so dass beispielsweise Seiten noch keinen umfangreichen Inhalt besitzen oder Ablaufschritte lediglich unvollständig spezifiziert sein müssen, ohne dass sie dadurch die Funktionsfähigkeit des generierten Gesamtsystems einschränken.

Softwareentwickler als Zielgruppe MontiWIS richtet sich im Wesentlichen an Personen, die bereits Erfahrung in der Entwicklung von Softwaresystemen haben und mit den Grundlagen der textuellen Spezifikation von Softwaresystemen vertraut sind.

1.3. Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich im weiteren Verlauf wie folgt.

Kapitel 2 erläutert die grundlegenden Funktionsweisen und Charakteristika von *Web-Informationssystemen* und stellt detailliertere Anforderungen an das zu entwickelnde System auf. Diese basieren auf den Zielen der Arbeit, werden jedoch an dieser Stelle noch konkretisiert.

Kapitel 3 führt in Begriffe der *modellbasierten Softwareentwicklung* ein, erläutert grundlegende Konzepte und gibt einen kurzen Überblick über das verwendete Framework MontiCore und seine Komponenten. Ferner werden spezifischere Anforderungen an die zu entwickelnden Modelle aufgestellt sowie der Forschungsstand hinsichtlich modellbasierter Webentwicklung dargestellt.

Kapitel 4 geht überblicksartig auf die Architektur von MontiWIS und den *Ablauf der Generierung* ein. Die Details der einzelnen Schritte werden hingegen in späteren Abschnitten dargestellt, wo der Prozess vom Modell zum generierten System für jede Modellart beschrieben wird.

Kapitel 5 fasst die Architektur und Infrastruktur des *generierten Systems* zusammen. Insbesondere das Zusammenspiel der einzelnen Komponenten wird an dieser Stelle beleuchtet, wobei auch hier auf Details verzichtet wird, da diese in späteren Abschnitten erläutert werden.

Kapitel 6 widmet sich der *Datenmodellierung* in MontiWIS. Hierbei wird zunächst die Sprache und ihre Verwendung erklärt. Nachfolgend wird der Generierungsprozess beschrieben, wobei die eingesetzte Datenstruktur erläutert und auf die verwendeten Kontextbedingungen eingegangen wird. Ferner werden die Abläufe bei der Bearbeitung der Modelle beschrieben und die Schritte der Transformationen auf der Datenstruktur veranschaulicht. Anschließend wird auf die Generierung der Systemteile eingegangen, die ursächlich mit den Datenmodellen zusammenhängen. Schließlich werden Implementierungsdetails der Daten- und Persistenzschicht in der generierten Applikation erläutert sowie ein kurzer Überblick gegeben, wie in anderen Ansätzen Datenmodellierung behandelt wird.

Kapitel 7 beschreibt den gesamten Prozess von der Sprachbeschreibung und -verwendung, über die Sprachverarbeitung bis hin zur Umsetzung im Zielsystem. Betrachtet werden hierbei jedoch die Mechanismen zur *Seitenbeschreibung* innerhalb von MontiWIS.

Kapitel 8 hat die *Ablaufsteuerung* und Abbildung von *Geschäftslogik* zum Thema. Auch hier wird zunächst die Sprache und ihre Verwendung erläutert. Anschließend wird auf die Vorgänge bei der Modellverarbeitung und Generierung sowie die Umsetzung der beschriebenen Aspekte im generierten System eingegangen.

Kapitel 9 behandelt die *Applikationssprache*, über die die Applikationskomposition und die Berechtigungen verschiedener Nutzergruppen beschrieben werden. Nach der Sprachbeschreibung werden Sprachverarbeitung und Generierung sowie die Implementierungsdetails der zugehörigen Anwendungsteile im generierten Web-Informationssystem erläutert.

Kapitel 10 beschreibt die *Anwendungsmethodik*, d. h. den Aufbau eines Projekts zur Verwendung des MontiWIS-Ansatzes sowie die Schritte, die zur Generierung, Installation und Aktualisierung der generierten Anwendung notwendig sind. Das iterative Vorgehen bei der Entwicklung eines Web-Informationssystems sowie die Besonderheiten, die dabei zu beachten sind, werden ebenfalls kurz erläutert. Ferner wird kurz auf vorgesehene Anpassungsmöglichkeiten eingegangen.

Kapitel 11 verdeutlicht anhand einiger *Fallstudien* die Eignung des MontiWIS-Ansatzes zur Entwicklung von Web-Informationssystemen. Dabei werden die Systeme im Überblick beschrieben. Die einzelnen Modelle und Komponenten der Systeme werden kurz erläutert, wobei ein Augenmerk auf Besonderheiten bei der Systemmodellierung gelegt wird.

Kapitel 12 schließlich fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche Entwicklungen und Verbesserungen, die zwar nicht mehr im Rahmen dieser Arbeit umgesetzt wurden, aber als mögliche Forschungsperspektiven dienen können.

Anhang Im Anhang dieser Arbeit sind neben einigen Verzeichnissen insbesondere die vollständigen *Grammatiken* enthalten, die im Hauptteil der Ausarbeitung lediglich in kompakter Form dargestellt oder auch nur erwähnt werden.

Kapitel 2.

Web-Informationssysteme

Ziel dieses Kapitels ist es, Web-Informationssysteme (WIS) als Zieldomäne näher zu beleuchten und damit ein Verständnis zu schaffen, welche Art von Systemen in MontiWIS beschrieben und generiert werden sollen. Obwohl bereits in Kapitel 1 kurz erwähnt, bedarf es einer weiterführenden Erläuterung, welche Anforderungen an ein solches System gestellt werden. Diese werden in Abschnitt 2.2 detaillierter beschrieben. Anschließend wird in Abschnitt 2.3 im Überblick betrachtet, welche verschiedenen Möglichkeiten zur Umsetzung solcher Systeme existieren. Schließlich gibt Abschnitt 2.4 eine kurze Einführung in das Szenario des Beispiels, das im Laufe der Arbeit verwendet wird.

2.1. Einordnung und Abgrenzung

Wie bereits 1998 postuliert, haben die Web-Informationssysteme einen *exponentiell höheren Einfluss auf unsere Leben, einfach, weil das Web das Potenzial hat, ein wesentlich breiteres Publikum zu erreichen als Client/Server Systeme* [IBV98]. Auch in [LF05] wurde bereits die Bedeutung der strukturierten Entwicklung von Websystemen erkannt und eine striktere Methodik bei der Erstellung derartiger Anwendungen angeregt. Konkret wird unter einem *Web-Informationssystem* in dieser Arbeit ein *Informationssystem* verstanden, das über einen Webbrowser bedient wird.

Primäre Aufgabe eines solchen Systems ist damit, Daten in einer strukturierten Art und Weise zu verwalten und sie beliebig zu verarbeiten. Eine grundlegende Klassifizierung von serverbasierten Webanwendungen in *Web Hypermedia Application*, *Web Software Application* und *Web Application* findet in [CZP01] statt. Erstere Art beschreibt eine Anwendung, die ausschließlich aus statischen Seiten und Verknüpfungen zwischen diesen besteht. Der zweite Begriff bezeichnet hingegen eine Applikation, deren Inhalte auf einer Art von Datenschema basieren und auf Daten operierende Funktionalität anbietet. Der letzte Begriff schließlich beschreibt eine Kombination aus beiden Ansätzen, in der sowohl statische Inhalte angeboten als auch Daten verarbeitet werden. Das in MontiWIS generierte System lässt sich damit in die letzte Kategorie einordnen.

Neben serverbasierten Web-Informationssystemen existieren weiterhin sogenannte Rich Internet Applications (RIAs), die sich insbesondere durch verstärkte Konzentration auf clientseitige Ausführung der Anwendung sowie hochperformante Benutzeroberflächen im Webbrowser, die an Desktopanwendungen erinnern, auszeichnen. Durch den Einsatz von Asynchronous JavaScript and XML (AJAX)[Gar05] werden Daten zwischen Client und Ser-

ver transportiert, ohne dass die aktuell dargestellte Seite vollständig neu geladen werden muss, weswegen das Benutzererlebnis dem von nativen Anwendungen ähnelt. Ansätze zur Modellierung und partiellen Generierung solcher Applikationen finden sich beispielsweise in [GGH+10, ADH+09, EGHZ12], wobei das Google Web Toolkit (GWT) [THET13, GWT] als Zielplattform zum Einsatz kommt. Ferner werden einige klassische modellgetriebene Ansätze um RIA Aspekte erweitert, wie etwa für UML-based Web Engineering (UWE) [PLMC+08], Web Modeling Language (WebML) [FCBC10] und Object-Oriented Hypermedia Design Method (OOHDM) [URGS07]. Da sich die Domäne der Anwendung jedoch von der in MontiWIS intendierten unterscheidet, werden diese Ansätze nicht im Detail weiter betrachtet.

2.2. Anforderungen an Web-Informationssysteme

Im folgenden Abschnitt werden die Anforderungen beschrieben, die sich im Vorfeld bei der Entwicklung verschiedener Arten von Web-Informationssystemen als sinnvoll und essenziell herauskristallisiert haben.

2.2.1. Funktionale Anforderungen

Für ein genaueres Verständnis, wodurch sich die Web-Informationssysteme auszeichnen, die Ziel der Entwicklung in MontiWIS sind, müssen ihre Charakteristika definiert werden. Dabei werden zunächst die einzelnen Funktionalitäten beschrieben, die in der Anwendung zur Verfügung stehen sollten.

Flexible Eingabe und Speicherung von Daten

Wie in der Definition in Abschnitt 2.1 beschrieben, werden in Web-Informationssystemen vorrangig Daten verwaltet, so dass dies auch eine vorrangige Anforderung an ein solches System darstellt.

Darunter ist an dieser Stelle zu verstehen, dass beliebige Daten über die Benutzeroberfläche in einem Webbrowser gespeichert werden können. Ebenso sollte es möglich sein, diese Daten zu bearbeiten. Die Art, wie dies geschieht, kann einerseits eng an die Datenstruktur angelehnt sein, so dass sich die Datenstruktur direkt in der Oberfläche widerspiegelt. Andererseits sollte ebenfalls die Möglichkeit bestehen, Teile von verschiedenen Datenstrukturen so miteinander zu kombinieren, dass die interne Struktur nicht mehr ohne Weiteres ersichtlich ist. Dies geschieht beispielsweise, wenn nur einzelne Werte in einem sonst statischen Text ausgegeben werden.

Die Datentypen, die im Web-Informationssystem zur Verfügung stehen, sollten von einfachem Text, numerischen Datentypen für kalkulatorische Operationen bis hin zu spezielleren Datentypen wie beispielsweise E-Mail-Adressen oder Datumsangaben reichen. Sowohl die Eingabe der Daten als auch ihre Ausgabe hat entsprechend ihres Typs stattzufinden. So sollte bei der Eingabe eines Datums Unterstützung durch einen sogenannten Datepicker angeboten werden, bei Datentypen, die einer bestimmten Syntax folgen müssen, sind die Eingaben entsprechend zu validieren.

Für jeden Teil der Datenstruktur, die dem Web-Informationssystem zugrunde liegt, sollte die Anwendung Funktionalität auf Seiten bereitstellen, die einem Nutzer das Anlegen, das Bearbeiten, das Löschen und das Auflisten sämtlicher zum Datentyp gehörenden Objekte ermöglicht. Solche Operationen werden im Allgemeinen als Create / Read / Update / Delete (CRUD) [Mar83] bezeichnet. Die Möglichkeit, diese Funktionalität in der Anwendung zu verwenden, sollte standardmäßig zur Verfügung stehen. Sofern es in das Konzept der Applikation passt, kann die Verwaltung der Daten allein über derartige Standardfunktionalität realisiert werden, anderenfalls kann sie beispielsweise zu Administrationszwecken verwendet werden.

Darstellung statischer und dynamischer Seiten

Auch wenn die primäre Aufgabe eines Web-Informationssystems die Verwaltung von Daten und deren Darstellung auf dynamisch erzeugten Webseiten ist, darf die Bedeutung von statischen Elementen wie Überschriften oder erklärenden Texten auf einer Seite nicht unterschätzt werden. Erst darüber wird die Anwendung benutzbar und hebt sich von einfachen Dateneingabemasken ab. Statische Inhalte sollten auch nicht nur in Kombination mit dynamischen Seiten einsetzbar sein, sondern auch allein stehend auf komplett unveränderbaren Seiten verfügbar sein. Dies ist insofern sinnvoll, als in einem Web-Informationssystem selbstverständlich auch Teile existieren, die bereits bei der Erstellung des Systems feststehen und auch im laufenden Betrieb nicht mehr geändert werden müssen.

Von der Möglichkeit, innerhalb einer Seite auf eine andere Seite zu verlinken, wird explizit Abstand genommen. Stattdessen werden Verknüpfungen zwischen Seiten und Abläufe durch die im nächsten Abschnitt vorgestellten Mechanismen umgesetzt.

Abläufe über mehrere Webseiten

Für die Verarbeitung und Verwaltung von Daten in einem Web-Informationssystem ist die simple Eingabe und Speicherung der Daten auf exakt einer Seite nur in den wenigsten Fällen zielführend. In der Regel werden Daten über ein Webeingabeformular eingetragen und an die Applikation gesendet. Basierend auf den eingegebenen Daten werden eine oder mehrere Folgeseiten aufgerufen, je nachdem, wie die zur Seite gehörige Logik die Daten be- und ausgewertet hat. Diese Art von Abläufen ist essenzieller Bestandteil eines Web-Informationssystems, wie es in MontiWIS intendiert ist. Dabei sollten die Daten, die auf der Seite eingegeben wurden, serverseitig verarbeitet werden können. Dies umfasst sowohl ihre Speicherung, aber auch die Belegung von konsekutiven Seiten und Formularen mit Werten, die entweder eingegeben oder aber anderweitig ermittelt wurden. Weitere Charakteristika der Abläufe werden im Folgenden erläutert.

Zum einen sollte es bei einem Ablauf dem Anwender überlassen werden, welcher Schritt nach Absenden des Formulars als Nächstes ausgeführt werden soll. Dies sollte durch die Anzeige verschiedener Bedienelemente in der Oberfläche realisiert werden, über die je eine Alternative ausgewählt werden kann. Dies können beispielsweise einfache Verweise oder auch Knöpfe zum Absenden des Formulars sein. Die Daten der jeweiligen Seite sollten in jedem

Fall zur weiteren Verwendung an den Server gesendet werden, so dass sie entsprechend verarbeitet werden können.

Zum anderen sollte die Verarbeitung der Daten vollkommen automatisch einzig auf Basis der eingegebenen Werte vonstattengehen können. Derartige lineare Seitenabfolgen kommen beispielsweise bei sogenannten Wizards vor, in denen ein Nutzer linear durch eine Anzahl von Seiten geleitet wird, auf denen er thematisch zusammengehörige Daten bearbeitet.

Ferner ist es wünschenswert, dass Daten von mehreren Personen nacheinander bearbeitet werden können. Derartige verteilte Abläufe sind beispielsweise sinnvoll, wenn an einem Ablauf mehrere Personen beteiligt sind und jede dieser Personen nur eine Teilmenge der benötigten Informationen manipulieren soll.

Ausführen beliebigen (Java-)Codes

Für möglichst große Flexibilität bei der Implementierung eines Web-Informationssystems ist es wünschenswert, dass serverseitig beliebiger Code ausgeführt werden kann. Diese Anforderung mag recht selbstverständlich erscheinen, wird aber aufgrund ihrer Bedeutung hier dennoch diskutiert. Ziel dieser Forderung ist es, sicherzustellen, dass der Funktionsumfang des Web-Informationssystems nicht durch den erlaubten Sprachumfang eingeschränkt wird. Durch die Möglichkeit, sowohl vor der Anzeige einer Seite (und damit der Interaktion mit dem Nutzer der Anwendung) als auch nach Absenden der Seite die eingegebenen Daten beliebig zu verarbeiten und etwa auch extern definierten oder bereits existenten Code aufzurufen, wird ein Maximum an Flexibilität erreicht. Da im Kontext von MontiWIS Java als Zielsprache gewählt wurde, wird an dieser Stelle auch die Unterstützung ebendieser Sprache gefordert. Durch Mechanismen wie Webservice-Aufrufe lassen sich neben der Ausführung von Java-Code auch recht flexibel andere Sprachen und Systeme ansprechen, sollte dies notwendig sein.

Nutzerspezifische Sichten auf die Applikation

Bei der bisherigen Entwicklung von Web-Informationssystemen stellte sich in quasi jedem Fall heraus, dass eine Art Nutzer- und Rechtekonzept obligatorisch ist. Allein die Möglichkeit, dass Nutzer sich am System per Nutzername und Passwort anmelden und danach zusätzliche Funktionalität zur Verfügung haben, ist für sinnvolle Anwendungsszenarien häufig notwendig.

In diesem Zusammenhang sollte das System nicht nur eine Unterscheidung hinsichtlich angemeldet und nicht angemeldet bieten, sondern auch verschiedene Rollen erlauben, in denen eine Person agieren kann. Jede Rolle, die einer Person zugewiesen ist, repräsentiert den Zugriff auf eine bestimmte Menge von Seiten, Abläufen und das erwähnte Standardverhalten in Form von CRUD Operationen zur Verarbeitung bestimmter Datentypen. Die Berechtigungen sollten dabei additiv sein, d. h., dass sich die Zugriffsrechte eines Nutzers aus der Vereinigung der Rechte seiner einzelnen Nutzerrollen ergeben. Neben anderen expliziten Rollen sollte es auch möglich sein, die Applikation unangemeldet zu benutzen, was einer Gastrolle im System gleichkommt.

In diesem Kontext ist es weiterhin wichtig, dass Rollen nicht fest und unveränderbar einem Nutzer zugewiesen werden, sondern dynamisch im System geändert werden können. Auf

diese Weise lassen sich Sichten realisieren, die einem Nutzer, abhängig vom aktuellen Systemzustand und den ihm zu einem bestimmten Zeitpunkt zugewiesenen Rollen, unterschiedliche Möglichkeiten zur Interaktion anbieten.

2.2.2. Qualitative (nicht-funktionale) Anforderungen

Neben den Anforderungen, die die reine Funktionalität des Web-Informationssystems beschreiben (vgl. Abschnitt 2.2.1), werden weiterhin Ansprüche an Webanwendungen gestellt, die eher ihre allgemeine Qualität und Benutzbarkeit betreffen. Diese werden in den folgenden Abschnitten beschrieben.

Einheitliche aber flexible Darstellung

Für die Benutzbarkeit und ein konsistentes Nutzererlebnis ist es wünschenswert, dass die Oberfläche des Web-Informationssystems einheitlich gestaltet ist. Dies umfasst die konsistente Darstellung der einzelnen Elemente, die eindeutig als Navigations- bzw. Interaktionselemente zu erkennen sein sollten [Gar11].

Neben der konsistenten Gestaltung sollte es durch die konsequente Verwendung von Cascading Style Sheets (CSS) [CSS, Mau11] relativ einfach möglich sein, das Gesamterscheinungsbild der Applikation anzupassen. Dies bezieht sich auf sämtliche Elemente der Seite und Oberfläche, also sowohl Textdarstellung als auch Anzeige der Bedienelemente wie Knöpfe, Eingabefelder oder Tabellen.

Intuitive Bedienung

Neben der konsistenten Darstellung der Applikation, wie sie im vorherigen Abschnitt beschrieben und gefordert ist, sollte die Bedienung des Web-Informationssystems möglichst intuitiv, d. h. direkt vom Anwender erschließbar, sein. Hier ist es beispielsweise von Bedeutung, dass zur Navigation und Interaktion mit den Elementen und Datentypen der Applikation die allgemein konventionellen und etablierten Darstellungsmittel zum Einsatz kommen [Gar11]. So werden Texte üblicherweise in Textfeldern eingegeben, ein Datum per Datepicker ausgewählt und Formulare über einen Absenden-Knopf zum Server gesandt. Über verschiedene Knöpfe auf einer Seite kann ebenso eine Auswahl getroffen werden, welcher Schritt als Nächstes auszuführen ist.

2.3. Web Engineering allgemein und Ansätze der Implementierung

Die strukturierte Entwicklung eines Web-Informationssystems wird allgemein auch als *Web Engineering* bezeichnet [Gin98, MDHG01]. Die Autoren erkennen die Notwendigkeit eines quasi formalen Prozesses, der analog zu einem gewöhnlichen Softwareentwicklungsprozess durch die Entwicklungsschritte eines (größeren) Web-Informationssystems führt. Einen Überblick über recht frühe Ansätze zur Entwicklung von Web-Informationssystemen gibt [Fra99]. [MM06] erläutert die Domäne des Web Engineering allgemein, wobei dort auch

modellbasierte Ansätze besprochen werden. [KPRR06a] gibt eine umfassende Einführung in die verschiedenen Teile des Web Engineerings und die daraus resultierenden Anforderungen. Genauer auf die beschriebenen Aspekte wird in den übrigen Teilen von [KPRR06b] eingegangen. [GM08] gibt einen guten Überblick über die Komponenten eines Websystems und [CDDM09] geht auf die generellen Konzepte der Entwicklung von Web-Informationssystemen sowie auf modellbasierte Ansätze dafür ein.

Ein Trend bzgl. der Entwicklung von Web Engineering 1.0 zu 2.0 lässt sich laut [WG08] durch die Entwicklung von AJAX erklären, was zu einer stärkeren Entkopplung einzelner Komponenten einer Webanwendung führt. Dies wird durch die häufigere Verwendung von Webservices, z. B. per Simple Object Access Protocol (SOAP) oder auch Representational state transfer (REST) ermöglicht. [Jaz07] beschreibt den Stand der Technik, wie er 2007 vorlag und größtenteils auch noch heute Anwendung findet. Ferner zeigt er Trends auf, die aus seiner Sicht zukünftig von Bedeutung sein werden (und aus heutiger Sicht größtenteils auch waren). Die dort behandelten Themen zeigen deutlich, dass für die Entwicklung von Webanwendungen im Allgemeinen und Web-Informationssystemen im Speziellen eine Vielzahl von verschiedenen Technologien für die Umsetzung existieren. Der Trend, Content Management Systeme als Plattform zur Entwicklung von Webanwendungen zu nutzen, wird in [NDMN14] diskutiert. Neben dem Profil der Nutzer und Entwickler solcher Systeme werden dort Parallelen und Unterschiede zu modellgetriebenen Ansätzen aufgezeigt.

2.3.1. Manuelle Implementierung

Zunächst werden in den folgenden Abschnitten einige der Möglichkeiten beschrieben, wie sowohl Web-Informationssysteme implementiert als auch auf die Datenhaltungsschicht (d. h. im Fall von MontiWIS auf ein relationales Datenbanksystem) zugegriffen werden kann. Hierbei geht es primär um diejenigen Methoden, die größtmögliche Flexibilität bei der Implementierung bieten und bei denen wenig bis gar keine Funktionalität von Frameworks oder anderen Abstraktionsmitteln übernommen wird, was zu einer Einschränkung der Ausdrucksmächtigkeit führen würde.

Die hier aufgeführten Sprachen und Konzepte stellen eine Auswahl verfügbarer Technologien dar.

Manuelle Implementierung von Web-Informationssystemen

Unter manueller Implementierung wird hier die, mehr oder weniger, direkte Verarbeitung von HTTP-Anfragen und die Rücksendung von Hypertext Markup Language (HTML)-Seiten verstanden. Diese Funktionalität wird im Allgemeinen von Webservern übernommen. Der wohl gebräuchlichste Webserver [Weba] ist Apache Webserver [Apa, Ker12]. In der Standardauslieferung werden von ihm in der Regel statische Inhalte wie beispielsweise HTML-Seiten oder einfache Bilder versandt. Webserver lassen sich jedoch durch Module erweitern, mit denen Sprachen wie PHP ausgeführt und interpretiert oder aber auch externe Programme per Common Gateway Interface (CGI) aufgerufen werden können.

CGI [RC04, McD10] war einer der ersten Ansätze, um Webseiten um dynamische Aspekte anzureichern. Dabei werden aus dem Webserver heraus weitere Programme aufgerufen, die

die Erzeugung der Antwortseiten bzw. Teile davon, übernehmen. In der Regel werden dafür Perl-Programme eingesetzt. In den letzten Jahren tritt dieser Ansatz bei der Entwicklung von Webanwendungen jedoch zugunsten anderer Sprachen und Konzepte immer weiter in den Hintergrund.

Aktuell wesentlich populärer ist hingegen beispielsweise PHP [PHP, Pow10]. Damit lässt sich im einfachsten Fall Programmcode zu ansonsten statischen Webseiten hinzufügen, der dann beim Aufruf der Seite ausgeführt wird. Diese Sprache bietet sämtliche Möglichkeiten, um auf einer geringen Abstraktionsebene Seiten Zeile für Zeile zu erzeugen. Darüber hinaus existieren jedoch auch Erweiterungen und Frameworks, über die die Entwicklung mit PHP wesentlich erleichtert wird und effizienter gestaltet werden kann.

Im Kontext der Java Programmiersprache basiert die Entwicklung von Web-Anwendungen in der Regel auf sogenannten Servlets [Sun10, HB04]. Diese Technologie wird beispielsweise implementiert durch einen Apache Tomcat Server [Tom, VG11], der sowohl für die Verarbeitung statischer Inhalte als auch die dynamischen Servletanteile zuständig ist. Ein Servlet dient im Wesentlichen dazu, HTTP-Anfragen zu verarbeiten und entsprechende Antworten zu generieren. Im einfachsten Fall werden nacheinander die Inhalte der Antwort analog zu gewöhnlichen Java `print`-Anweisungen zusammengestellt und das Ergebnis an den Client zurückgesendet. Eine Abstraktion von diesem sehr aufwendigen Vorgehen bieten JavaServer Pages (JSP)-Seiten. Diese sind ähnlich wie PHP-Seiten aufgebaut und erlauben in ihrer einfachsten Form die direkte Einbindung von Java-Code in das HTML-Gerüst einer Seite. Darüber werden auch in diesem Fall die dynamischen Aspekte der Seite programmatisch zusammengestellt. Von der direkten Verwendung von Java in HTML ist aufgrund der mangelnden Trennung von Inhalt und Logik jedoch Abstand zu nehmen. Stattdessen sollte bei Verwendung von JSP eher auf die JavaServer Pages Standard Tag Library (JSTL) in Verbindung mit der Expression Language (EL) [Sun02, Bay03] bzw. auf selbst geschriebene Tags oder Funktionen [SCR01] gesetzt werden. Die Inhalte einer JSP-Seite werden durch einen JSP- Compiler in Java-Code übersetzt, der letztlich als Servlet ausgeführt wird. Die statischen Teile einer solchen Seite werden in die zugehörigen `print`-Anweisungen umgesetzt, die dynamischen Teile einer JSP-Seite finden sich in ihrer ursprünglichen Form im generierten Servlet-Code wieder.

Ein sehr ähnliches Modell wie JSP verwendet ASP.NET [Wen10], welches jedoch auf die Sprachlandschaft und Infrastruktur von Microsoft und deren .NET Technologie [Wen11] aufsetzt. Diese lehnen sich im einfachsten Fall vom Funktionsumfang her an PHP oder JSP-Seiten an, können jedoch auch fortgeschrittenere Mechanismen und Frameworks verwenden, die einen Teil des Implementierungsaufwands abnehmen.

Einen anderen Ansatz verfolgt GWT[GWT, THET13]. Dabei werden sowohl die clientseitige Logik, die die Oberfläche erzeugt und steuert, als auch die serverseitige Geschäftslogik in Java geschrieben. Durch den GWT-Compiler wird der entstandene clientseitige Code in JavaScript übersetzt, so dass die Benutzeroberfläche im Webbrowser ausgeführt wird und transparent mit den serverseitigen Komponenten kommuniziert. Dadurch lassen sich hochdynamische Anwendungen erzeugen, ohne dass nativer JavaScript-Code bei der Entwicklung geschrieben werden muss.

Manueller Zugriff auf die Datenhaltungsschicht

Eine Möglichkeit der Persistierung von Daten wären Not only SQL (NoSQL) [Pok13] Datenbanken, die primär im Kontext von Cloudanwendungen eingesetzt werden, in der Regel nicht auf einem relationalen Datenmodell, sondern beispielsweise auf Schlüssel-Wert-Paaren, einem dokumentenzentrierten Ansatz oder XML-Formaten basieren und prädestiniert für die Speicherung immenser Datenmengen (*Big Data*) sind. Sie zielen auf hervorragende Skalierung hinsichtlich Speicherplatz aber auch Anfragegeschwindigkeit ab, die sie in der Regel auf Kosten einiger Zusicherungen erreichen, die relationale Systeme bieten. Einen detaillierten Überblick über die verschiedenen Arten von NoSQL-Datenbanken und Speicherschemata sowie eine Diskussion von Vor- und Nachteilen bietet [Hec15]. Da es in MontiWIS nicht Ziel ist, Massendaten zu verarbeiten oder die Anwendung auf Cloudarchitekturen auszurichten und daher die speziellen Eigenschaften von NoSQL-Systemen nicht benötigt werden, wird auf traditionelle relationale Datenbanksysteme gesetzt.

Der manuelle Zugriff auf ein Datenbanksystem geschieht in der Regel durch den direkten Aufruf von SQL-Abfragen, die an die Datenbank gesendet werden. Dazu kommt üblicherweise im Java-Kontext die Java Database Connectivity (JDBC) [Ora11, KE11] zum Einsatz, mit der der Zugriff auf Daten (überwiegend) datenbankagnostisch möglich ist. Durch die Verwendung von dieser oder vergleichbarer Technologien ist in der Regel die manuelle Implementierung von sämtlichen Aspekten des Datenbankzugriffs notwendig. Dies umfasst sowohl die Definition der Datenbankstruktur als auch sämtliche Abfragen, die daran gestellt werden. Hierüber lassen sich explizite Optimierungen an der Datenbankstruktur vornehmen, was jedoch auch einen größeren Aufwand als bei der Verwendung von Frameworks bedeutet. Auch für andere Programmiersprachen lassen sich direkte Zugriffe analog zu JDBC realisieren, worauf an dieser Stelle aufgrund der Ähnlichkeit der Konzepte nicht weiter eingegangen wird.

2.3.2. Erhöhte Abstraktion durch den Einsatz von Frameworks

Im Gegensatz zu den manuellen Ansätzen, die im vorherigen Abschnitt erwähnt wurden, existieren sowohl im Bereich der Entwicklung von Web-Informationssystemen als auch beim Zugriff auf Datenbanken Ansätze, die den Aufwand der Entwicklung verringern, indem sie den Abstraktionsgrad der verwendeten Sprachmittel erhöhen. Derartige Frameworks bieten den Vorteil, dass hier bestimmte Annahmen durch die Intelligenz, die im Framework implementiert ist, festgelegt sind und daher nicht explizit spezifiziert werden müssen. Nachteil eines solchen Vorgehens ist jedoch, dass durch die Verwendung von Frameworks und die damit verbundene erhöhte Abstraktion nicht beliebige Flexibilität zur Verfügung steht, was beispielsweise in [GHKV08] diskutiert wird.

Einige populäre Frameworks sowohl zur Implementierung von Web-Informationssystemen und als auch für den Zugriff auf die Datenhaltungsschicht werden in folgenden Abschnitten kurz erläutert und diskutiert.

Frameworks bei der Implementierung von Webanwendungen

Ein populärer Ansatz für die effiziente Entwicklung von Webanwendungen ist *Ruby on Rails* [RoR, Win13]. Dieses Framework setzt auf die Sprache Ruby auf. Es enthält Erweiterungen wie beispielsweise einen Scaffolding-Mechanismus zur direkten Generierung von CRUD-Funktionalität inklusive Seiten und Datenzugriffslogik. Es bietet dezidierte Mechanismen für Datenbankzugriff per *Active Record*-Muster [Fow03], spezielle Controller- und View-Komponenten sowie Generatoren für einfache Gerüste dieser Programmteile, die die Entwicklung von Web-Informationssystemen relativ stark vereinfachen.

Einen ähnlichen Ansatz wie Ruby on Rails verfolgt *Grails* [Gra]. Dieses basiert im Grundsatz auf Java bzw. der ebenfalls durch die Java Virtual Machine (JVM) ausgeführte dynamische Skriptsprache *Groovy* [Gro]. Grails bietet ähnliche Konzepte wie Ruby on Rails, ebenfalls einen Scaffolding-Generator sowie Möglichkeiten zur Datenmanipulation und -darstellung auf Webseiten. Nachteil bei derartigen Ansätzen ist jedoch, dass die aus dem Scaffolding hervorgegangenen Dateien zwar existieren und bearbeitet werden können, aber in der Regel wieder überschrieben werden, sobald sie bei der Evolution des Systems neu erzeugt werden müssen.

Weniger umfangreich und auch bereits etwas älter ist das *Struts* Framework [Str]. Dabei handelt es sich um ein Model View Controller (MVC) Framework, das insbesondere für die Darstellung der Seiten und den Transport von Daten zur Seite und wieder zurück zur Serverseite Funktionalität bereitstellt. Ferner lassen sich über Extensible Markup Language (XML)-Konfigurationsdateien die Routen zwischen den verschiedenen Seiten und Sichten innerhalb der Applikation spezifizieren und in der Programmlogik über eindeutige Bezeichner referenzieren.

Ebenfalls ein Framework, das auf Java aufsetzt, ist *Tapestry* [Tap]. Bei Tapestry werden weniger offensichtliche Generierungsmechanismen eingesetzt, als vielmehr Konventionen zu Namens- und Dateistruktur sowie Annotationen festgelegt. Beispielsweise wird durch Namensgleichheit von Templates (für die View-Erzeugung) und Java- Klassen (für die Verarbeitung der Daten) festgelegt, wie Daten von Modell zu View übertragen und wieder daraus extrahiert werden können. Tapestry stellt weiterhin Komponenten für die erweiterte Darstellung bestimmter Datentypen sowie die Validierung von Eingaben bereit, ebenso wie Möglichkeiten zum Verweis und zur Anzeige von nachfolgenden Seiten. Zentrale Komponente von Tapestry ist der Inversion of Control (IoC) Container, über den ein Großteil der Funktionalität realisiert wird. Von Vorteil ist auch hier wieder die wesentlich kompaktere Art, die Applikation zu entwickeln. Nachteilig dabei ist jedoch erneut, dass eine Änderung des durch das Framework vorgegebenen Verhalten mit erheblichem Aufwand verbunden ist.

Einen IoC-Container als zentrale Komponente bietet ebenfalls das *Spring-Framework* [Spr]. Neben der Möglichkeit, per IoC die einzelnen Komponenten miteinander zu verschalten, sind darin ebenfalls Möglichkeiten zur Verwendung von aspektorientierten Ansätzen, Funktionalität für den Datenbankzugriff und deren Persistenzhaltung, Nutzerinteraktion per MVC [Ree79], sowie Transaktionsmanagement und Authentifizierung, um nur einige zu nennen, enthalten.

In der Regel wird Spring als Alternative oder Ergänzung zur *Java Platform Enterprise Edition*

(Java EE) [Sun09a, Hef10][3]), welche ebenfalls für eine Vielzahl von Aufgaben Standards vorhält. Auch diese Spezifikation ist nicht als monolithischer Kern zu sehen, sondern bietet einen Überbau für verschiedene andere Spezifikationen wie beispielsweise *Java Persistence API (JPA)* [Sun09b, KS09] für die Persistenz innerhalb des Containers oder auch *JavaServer Faces (JSF)* [Ora13d] für die Erstellung der View-Komponente der Webapplikation sowie der Verarbeitung von Anfragen auf dem Server.

All diese Frameworks besitzen mehr oder weniger enge Vorgaben hinsichtlich der Funktionalität, die durch sie bereitgestellt wird. In der Regel resultieren daraus Einschränkungen, die für die gestellten Anforderungen an das zu erstellende Web-Informationssystem problematisch sind, oder aber die der Implementierung nicht im Weg stehen. Genaueres zur Auswahl der Zielplattform findet sich in Kapitel 5.

Frameworks beim Zugriff auf die Datenhaltungsschicht

Auch bei der Speicherung von Daten in einem Datenbanksystem existieren Ansätze, die sowohl das Anlegen der Datenbankstruktur als auch den Zugriff darauf durch bestimmte Annahmen und Mechanismen vereinfachen.

Ein sehr populärer Ansatz zur Datenspeicherung ist das sogenannte *objektrelationale Mapping*. Dabei werden Klassen, wie beispielsweise Java-Klassen, direkt auf eine relationale Datenbankstruktur abgebildet und durch die Operationen des objektrelationalen Frameworks verwaltet. Im Regelfall werden sowohl die Datenbanktabellen inklusive ihrer Struktur aus der Klassenstruktur erzeugt als auch der Zugriff auf die einzelnen Objekte transparent durch das Framework geregelt. Frameworks, die diese Art von Persistenzfunktionalität bieten, existieren für quasi jede Programmiersprache.

Für Java sind *Hibernate* [Hib] sowie *JPA* [Ora13a, KS09] die weitest verbreiteten Ansätze für objektrelationales Mapping. JPA ist dabei ein Standard als Teil der Java EE, der von unterschiedlichen konkreten Frameworks, dabei auch Hibernate, implementiert und z. T. leicht erweitert wird.

2.3.3. Generative Ansätze

Als weitere Möglichkeit zur Erstellung von Web-Informationssystemen bieten sich generative Ansätze an. Zum Teil lassen sich dazu die bereits beschriebenen Ansätze Ruby on Rails und Grails zählen, da darin Teile der Applikation bzw. im Fall, dass der rudimentäre Funktionsumfang ausreicht, vollständige CRUD-Applikationen generiert werden. Weitere Ansätze werden im folgenden Abschnitt 3.5 erläutert und näher beleuchtet.

2.4. Durchgängiges Beispiel

In Anlehnung an die ursprüngliche Intention dieser Arbeit, die aufgrund des erheblichen Aufwands bei der Entwicklung von Web-Informationssystemen im universitären Kontext

[3] An dieser Stelle wird auf die Java EE v6 verwiesen, weil sie bei der Implementierung von MontiWIS zum Einsatz kam. Während der Ausarbeitung der Arbeit wurde jedoch bereits Version 7 [Ora13c] veröffentlicht.

eine Verringerung eben dessen anstrebt, stammt das Beispiel, das durch diese Arbeit führt, ebenfalls aus der Domäne einer Universität.

Im Wesentlichen orientieren sich die Beispiele an Anwendungsfällen, die in einer Universität vorkommen. Die Nutzer des Systems unterteilen sich in Angestellte der Universität, die einem Institut zugeordnet sind und bei der Bewertung einer Arbeit als Erst- oder Zweitgutachter teilnehmen können, und Studenten, die in einem Studiengang eingeschrieben sind. Die Datenstruktur wird detaillierter in Abschnitt 6.2.2 erläutert. Ein erster Eindruck einer generierten Applikation aus dieser Domäne ist in Abbildung 2.1 dargestellt.

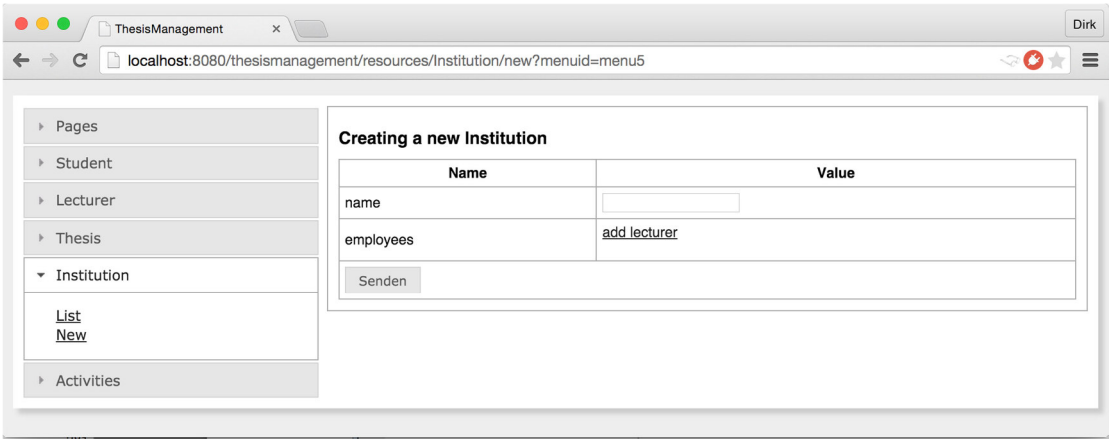


Abbildung 2.1.: Generierte Applikation aus dem durchgängigen Beispiel

Der zur Domäne passende Ablauf befasst sich mit der Bewertung einer studentischen Arbeit. Daran sind zwei Angestellte beteiligt, die nacheinander ihre Noten im Web-Informationssystem eintragen. Detailliertere Ausführungen diesbezüglich finden sich in Abschnitt 8.2.2.

Kapitel 3.

Modellgetriebene generative Softwareentwicklung

Auch wenn, wie in Kapitel 2 dargelegt, viele Ansätze zur Entwicklung von Web-Informationssystemen existieren, so erfordert die Verwendung dieser Methoden und Frameworks in der Regel bei jedem entwickelten System stets einen ähnlichen, doch erheblichen Aufwand. Aus diesem Grund wurden im Rahmen dieser Arbeit modellgetriebene, generative Ansätze in Betracht gezogen um die notwendigen, schematisch ähnlichen Schritte bei der Entwicklung zukünftig zu verringern. In der Literatur werden zum Teil Produktivitätsgewinne bis zum Faktor 10 genannt [KT08, Nok]. Ein direkter Vergleich hinsichtlich des Aufwands (z. B. durch das Gegenüberstellen von Anzahl von Codezeilen) ist allerdings insofern schwierig, als bei manueller Implementierung potenziell andere Frameworks zum Einsatz kommen, als dies bei generativen Ansätzen wie MontiWIS der Fall ist. Dennoch bietet eine derartige Gegenüberstellung einen Anhaltspunkt für die Effizienz des entwickelten Ansatzes. Ein Vergleich von handgeschriebenen Modellzeilen und generiertem Code für MontiWIS findet sich in Abschnitt 11.5.1.

Dieses Kapitel dient dazu, modellbasierte bzw. modellgetriebene Konzepte zu erläutern. In Abschnitt 3.1 werden zunächst die Grundlagen modellbasierter bzw. -getriebener Softwareentwicklung erläutert. Im darauffolgenden Abschnitt 3.2 werden verschiedene Werkzeuge zur modellgetriebenen Entwicklung vorgestellt. Abschnitt 3.3 beschreibt die grundlegenden Funktionen des im Rahmen dieser Arbeit verwendeten Frameworks MontiCore. In Abschnitt 3.4 werden die allgemeinen Anforderungen an die Modelle beschrieben sowie die zu modellierenden Aspekte definiert, die für die erfolgreiche Erfüllung der Anforderungen aus Abschnitt 2.2 sinnvoll und notwendig sind. Abschließend beschreibt Abschnitt 3.5 ähnliche Konzepte, die ebenfalls einen modellgetriebenen Ansatz zur Entwicklung von Web-Informationssystemen verfolgen.

3.1. Begriffsbildung und Abgrenzungen

Für eine sinnvolle Diskussion und Erläuterung modellbasierter bzw. modellgetriebener Konzepte sind zunächst einige Begriffe zu definieren.

Für den Begriff des *Modells* sind in der Literatur verschiedene Definitionen zu finden. Eine sehr weitreichende Sichtweise ist unter *„everything is a model“* zusammengefasst [Béz04]. Etwas konkreter ist da bereits die Definition *„A model is a coherent set of formal elements*

describing something [...] built for some purpose that is amendable to particular form of analysis, such as communicating ideas between people and machines, [...] [or] transformation into an implementation.“[MCF03]. Hier findet bereits eine Einschränkung hinsichtlich der Formalität und des Zwecks eines Modells statt. Auch [Küh05] befasst sich mit der Frage der Definition eines Modells. Er grenzt sich von der Idee, dass alles ein Modell sei, insofern ab, als bestimmte Eigenschaften für technische Modelle gelten sollten, um einen enger gefassten und zu beliebigen Artefakten abgrenzbaren Modellbegriff zu erhalten. Als eine sinnvolle Definition erachtet er diejenige von [Sta73]. Danach muss ein Modell ein *Abbildungsmerkmal* besitzen, womit Modelle Repräsentationen natürlicher oder künstlicher Objekte sein sollen. Des Weiteren sollten Modelle ein *Verkürzungsmerkmal* besitzen und damit eine Abstraktion von bestimmten Eigenschaften des Originals bieten. Schließlich benötigt ein Modell seiner Ansicht nach ein *pragmatisches Merkmal*, womit ein Modell zu einem bestimmten Zweck in einem bestimmten Kontext zu einem bestimmten Nutzen anstelle des Originals verwendet werden kann.

Dieser Definition wird sich im Kontext dieser Arbeit angeschlossen. Modelle werden in MontiWIS dazu eingesetzt, das zu erstellende System zu beschreiben. Dabei werden nur die relevanten Aspekte abstrakt beschrieben, aus denen die Implementierungen abgeleitet werden, so dass bei der Verarbeitung daraus ein vollständiges System generiert werden kann.

Des Weiteren sind die Begriffe *modellbasierte* und *modellgetriebene Ansätze* zu erläutern und abzugrenzen. [MCF03] bezeichnen modellgetriebene Entwicklung provokativ mit *„Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing.“*. Diese Sichtweise wird jedoch von ihnen direkt revidiert aufgrund der Tatsache, dass damit auch das Schreiben von Programmcode und seine Transformation in eine ausführbare Form als modellbasierte Entwicklung betrachtet würde. Der Grad der Abstraktion von Details, wie er bei dieser Sichtweise vorliegt, kann zwar als Ziel, aber in der Regel noch nicht als Status Quo angesehen werden. Dazu müssten Modelle und Modellelemente den first-class Status besitzen, den Objekte und Klassen in den 80er Jahren erhalten haben. Ihre Rolle müsste sich weg von reinen Dokumentationszwecken hin zum Einsatz in der Softwareerstellung entwickeln [Béz04]. Der Erfolg von modellgetriebener Entwicklung kann sich nur einstellen, wenn einerseits vollständige Programme aus Modellen erzeugt werden, andererseits die Qualität und Korrektheit der Modelle, beispielsweise durch ihre Ausführung, verifiziert wird. Damit wird Modellen die Rolle der Implementierungssprache zugestanden, ähnlich wie dies Programmiersprachen der dritten Generation als Substitut für Assemblersprachen taten [Sel03, Sch06]. Daraus resultiert im Wesentlichen auch der Unterschied zwischen (im besten Fall) modellbasierten und modellgetriebenen Ansätzen. *Modellgetriebene Ansätze* werden diejenigen Vorgehen bezeichnet, welche Modelle direkt zur Softwareproduktion einsetzen, wohingegen Modelle bei *modellbasierten Ansätzen* lediglich zur Spezifikation eines Systems verwendet werden, woraus durch verschiedene manuelle Schritte ein Softwaresystem erstellt wird [Sch12]. Bei MontiWIS handelt es sich bei Verwendung dieser Definitionen folglich um einen modellgetriebenen Ansatz, da Modelle hier zur Spezifikation und Generierung eines vollständig lauffähigen Systems eingesetzt werden.

Ein wichtiger Ansatz im Kontext der modellgetriebenen Entwicklung ist die *Model-driven Architecture (MDA)* [Obj03, Fra03, KWB03]. Dieser Ansatz basiert darauf, dass die Modelle von einer sehr abstrakten Sichtweise über Transformationen verfeinert und hin zu einer sehr

implementierungsnahen Repräsentation überführt werden. Hierbei werden die folgenden drei Schichten unterschieden. Das *Computation Independent Model (CIM)* beschreibt das zu modellierende System in einer Art und Weise, die sämtliche Details vernachlässigt, die Implementierungskonzepte betreffen. Diese Modelle beschränken sich damit auf eine domänennahe Sichtweise. Das *Platform Independent Model (PIM)* beinhaltet Aspekte, die nicht auf eine bestimmte Plattform abzielen. Unter einer Plattform ist hier eine Menge von Konzepten zu verstehen, die mehreren Plattformen gemein sind, ohne auf die Details einer speziellen Implementierung einzugehen. Beispiele wären hier etwa die Komponenten einer JavaEE-Anwendung. Das *Platform Specific Model (PSM)* schließlich spezifiziert, wie die Konzepte des PIM im Modell der Plattform umgesetzt werden. Im Kontext der MDA wird überwiegend die Unified Modeling Language (UML) [Obj11c, RQZ07] als Modellierungssprache eingesetzt. Zur Spezifikation der Metamodelle kommt in der Regel *Meta-Object Facility (MOF)* [Obj13c] zum Einsatz, wobei die Speicherung der darauf basierenden Modelle in *XML Metadata Interchange (XMI)* [Obj13a] geschieht. Die Transformation von Modellen in diesem Kontext kann beispielsweise über die *Query View Transformation (QVT)* Spezifikation [Obj11b] geschehen. Eine Diskussion der MDA im Kontext des modellbasierten Web Engineering findet sich in [MRV08].

Einen weiteren Beitrag hin zur Erhöhung des Abstraktionsgrades bei der Softwareentwicklung ist das sogenannte *Domain Specific Modeling (DSM)* [KT08]. Damit soll es möglich sein, die Modelle direkt in bzw. nah an der Domäne des zu modellierenden Systems zu beschreiben. Hierzu sollen sowohl Vokabular als auch Notation der Domäne in Form einer *Domain Specific Language (DSL)* verwendet werden, so dass die Verständigung über die Modelle sowie ihre Semantik eindeutiger ist, als dies bei der Abbildung der Domänenkonzepte auf allgemeine Modellierungssprachen wie die UML der Fall wäre. Die Sinnhaftigkeit der Entwicklung von DSLs sowie die Entwicklungsschritte werden in [MHS05] diskutiert. Einen Überblick über relevante Literatur und Ansätze gibt [vDKV00]. Ebenso werden dort verschiedene Arten von DSLs sowie unterschiedliche Entwicklungsansätze diskutiert. Einen recht vollständigen Überblick über die Entwicklung von (primär textuellen) DSLs bieten sowohl [Voe13] als auch [SVEH07] und [Fow10].

3.2. Werkzeuge für die modellgetriebene Entwicklung

Für die Unterstützung der Entwicklung von domänenspezifischen Sprachen existieren einige Ansätze, die im folgenden Abschnitt kurz beleuchtet werden. Hierbei wird unterschieden in Ansätze, die die Definition von grafischen DSLs erlauben (Abschnitt 3.2.1) und solche, die sich auf textuelle Sprachen beschränken (Abschnitt 3.2.2). Hierbei soll keine Evaluation und Auswahl eines konkreten Ansatzes für die nachfolgende Verwendung im Kontext von MontiWIS stattfinden, sondern lediglich eine Übersicht über einige wichtige und repräsentative Produkte gegeben werden.

3.2.1. Grafische Ansätze

Eine Möglichkeit zur Definition von DSLs sind grafische Editoren, über die grafische Modelle spezifiziert und bearbeitet werden können.

MetaEdit+ [KT08, Met] ist ein Werkzeug der Firma MetaCase. Darüber lassen sich zunächst auf grafische Art und Weise Metamodelle spezifizieren, denen nachfolgend eine konkrete, grafische Symbolik als konkrete Syntax zugewiesen wird. Zu den Modellen lassen sich Codegeneratoren entwickeln, die über eine textuelle Sprache definiert werden. Zu den hervorzuhebenden Funktionen zählen laut [TK09] insbesondere die Möglichkeit, (Meta-)Modelle zu versionieren sowie den Prozess der Codegenerierung durch einen Debugger, analog zu konventionellen Entwicklungsumgebungen, zu unterstützen. Zusammenfassend kann MetaEdit+ als ausgereiftes kommerzielles Werkzeug aus dem industriellen Kontext gesehen werden, mit dem für grafische Sprachen zufriedenstellende Ergebnisse zu erzielen sind.

Poseidon for DSLs sowie *Poseidon for UML* sind beides Werkzeuge von Gentleware. Mit ersterem Werkzeug lassen sich über *ECore*, selbst Teil des *Eclipse Modeling Framework (EMF)* [EMFa], Metamodelle definieren, zu denen ebenfalls grafische Elemente zugeordnet werden können, mit denen letztlich modelliert wird. Das zweitgenannte Werkzeug bietet ebenfalls die Möglichkeit der grafischen Modellierung, beschränkt sich dabei jedoch auf eine Vielzahl der Modelle, die unter der UML zusammengefasst sind. Für beide Werkzeuge ist es möglich, Codegeneratoren zu schreiben, bzw. im Fall von Poseidon for UML, bereits Existierende zu verwenden oder anzupassen.

Eine weitere Diskussion von Werkzeugen, die insbesondere auf UML aufsetzen, ist beispielsweise in [Sch12] zu finden.

3.2.2. Textuelle Ansätze

Neben grafischen Werkzeugen zur Erstellung von Modellierungssprachen existieren auch Ansätze, die die Spezifikation von textuellen Modellen erlauben. Einige dieser Werkzeuge werden im folgenden Abschnitt überblicksartig erläutert. Diese Werkzeuge basieren üblicherweise auf Parsertechnologien, wie sie bereits im Compilerbau vorkommen [ASU86].

In *EMFText* [EMFb] werden Grammatiken in einer auf *Extended Backus Naur Form (EBNF)* basierenden Spezifikationssprache definiert, um auf einem ECore [EMFa]-basierten Metamodell eine konkrete Syntax zu spezifizieren, woraus ein umfangreiches Tooling zur Verarbeitung der Modelle generiert wird. Neben Funktionalität zum Laden und Speichern von Modellinstanzen werden beispielsweise auch Editorunterstützung und Methoden zur Analyse von Modellen erzeugt.

Spoofax [KV10, Spo] bietet ebenfalls die Möglichkeit, per Grammatik eine textuelle Sprache zu definieren. Daraus wird Infrastruktur zur Modellverarbeitung, insbesondere auch umfangreiche Editorfunktionalität, generiert, die bei der Entwicklung von Modellen unterstützt. Mit *Stratego* [BKVV08] besitzt Spoofax eine mächtige Transformationssprache, die sowohl zur Analyse als auch zur Synthese von Modellen bzw. Code genutzt werden kann. Auf Spoofax basiert beispielsweise auch WebDSL, welches in Abschnitt 3.5 näher erläutert wird.

Xtext [Xte, Bet13] ist eine recht populäres Framework zur Entwicklung von DSLs. Es ist als Teil des Eclipse Modeling Framework Projects aus *openArchitectureWare (oAW)* hervorge-

gangen. Über Xtext lassen sich über Grammatiken Sprachen definieren, deren Metamodell ebenfalls über ECore [EMFa] abgebildet wird. Auch hier wird umfangreiche Editorunterstützung generiert.

Einen weiteren, vertiefenden Überblick und einen Vergleich von verschiedenen DSL-Werkzeugen findet sich in [Mer10, PP08]. Generelle Probleme und Herausforderungen bei der Verwendung (textueller) Modelle werden etwa in [FR07, FRS13] diskutiert.

3.3. MontiCore

Nach der überblicksartigen Besprechung vergleichbarer Ansätze zur Erstellung von sowohl grafischen als auch textuellen DSLs wird in diesem Abschnitt auf MontiCore eingegangen, das im Rahmen dieser Arbeit das Werkzeug der Wahl zur Erstellung der Sprachen ist. MontiCore wurde im Rahmen der Dissertation von Holger Krahn entwickelt [Kra10]. Einen kompakteren Überblick über MontiCore geben beispielsweise [KRV10, KRV08].

3.3.1. Grundlagen der Entwicklung mit MontiCore

Mit dem Framework MontiCore ist es möglich, textuelle domänenspezifische Sprachen zu spezifizieren. Dabei werden sowohl die konkrete als auch die abstrakte Syntax der Sprache in einer MontiCore-Grammatik beschrieben [KRV07b], die in einer Datei mit der Endung `mc` zu speichern ist. MontiCore selbst basiert auf ANTLR [Par07], wobei dieser Parsergenerator um einige wesentliche Aspekte erweitert wurde. Die Grammatiken von MontiCore unterstützen verschiedene Mechanismen, die eine modulare Entwicklung von Sprachen und damit auch Modellen ermöglichen [KRV08]. So lassen sich Grammatiken voneinander vererben, wodurch die Produktionen der Elterngrammatik in den Kindgrammatiken ebenfalls verfügbar sind. Ferner besteht die Möglichkeit, über Einbettung einer Grammatik anstelle eines Nichtterminals mehrere Sprachen miteinander zu verknüpfen. Beide Konzepte der Modularität werden im Rahmen von MontiWIS verwendet. Die Vererbung erlaubt beispielsweise die Weiterverwendung von Konzepten der UML/P [Sch12], wohingegen die Einbettung die Verwendung von Java-Ausdrücken durch die Einbettung der in MontiCore umgesetzten Java-Grammatik ermöglicht [FM07].

Die Modelle, die mit MontiCore verarbeitet werden, sind in der Regel in einzelnen Dateien gespeichert, die, ähnlich wie Java Klassen, in Paketen organisiert sind. Diese Paketangaben (spezifiziert über das Schlüsselwort `package`) sind in der Modelldatei anzugeben. Ebenfalls analog zu Java-Klassen lassen sich über `import`-Anweisungen bestimmte Pakete oder explizite Modelle in ein Modell importieren, so dass verkürzte anstatt vollqualifizierte Namen verwendet werden können. Ebenfalls analog zu Java lassen sich in Grammatiken und (in aller Regel ebenfalls in Modellen) Kommentare über die dort übliche Syntax spezifizieren. Einzeilige Kommentare werden eingeleitet über zwei doppelte Schrägstriche (`//`), mehrzeilige Kommentare durch Schrägstrich und Asterisk (`/*`) und abgeschlossen durch die umgekehrte Zeichenfolge (`*/`).

Aus der Grammatik wird sowohl ein Parser als auch eine Reihe von Infrastrukturklassen zur Sprachverarbeitung generiert, über die ein recht einfacher Zugriff auf die Modelle ermöglicht

wird. Einerseits werden für jede Modellart Java-Klassen generiert, die die *Abstract Syntax Tree (AST)*-Struktur der Modelle abbilden. Über diese Klassen lassen sich unter Verwendung von *Visitor*-Klassen [GHJV95] die einzelnen Modellelemente ansprechen. Für die Verarbeitung von Modellen wird in MontiCore das sogenannte *DSLTool*-Framework verwendet [Kra10, KRV10]. Ein `DSLTool` liest Modellinstanzen ein und erzeugt für jede Instanz ein `DSLRoot`-Objekt, über welches der Zugriff auf die zum Modell gehörigen AST-Klassen möglich ist.

Weitere Unterstützung beim Bearbeiten von Modellinstanzen bieten auch die relativ einfach zu erstellenden Eclipse-Editoren, die als Plugin vorliegen und Hilfestellung in Form von Syntax-Higlighting, Folding sowie Outlines und im begrenzten Maße kontextsensitive Codevervollständigung bieten [KRV07a].

3.3.2. Symboltabelleninfrastruktur

Ein besonderes Augenmerk bei der Sprachentwicklung in MontiCore kommt den Bestandteilen zu, die im Rahmen der Dissertation von Steven Völkel [Völ11] entwickelt wurden. Hierbei wurde die Kompositionalität von Sprachen in MontiCore erhöht und durch entsprechende Infrastruktur unterstützt. Zentrale Komponente dabei sind die Symboltabelle und die Auflösungsmechanismen ihrer Einträge. Unter einer Symboltabelle versteht man *eine meist hierarchisch gegliederte Datenstruktur zum Speichern und Auflösen von Bezeichnern in einer Sprache. Kernaufgabe besteht im Auflösen von Namen mit dem Ziel, weitere Informationen wie Typ oder Signatur zu diesem Namen zu erhalten* [Völ11]. Die Klassen, die maßgeblich für die Funktion der Symboltabelle sind, sind überblicksartig in Abbildung 3.1 dargestellt.

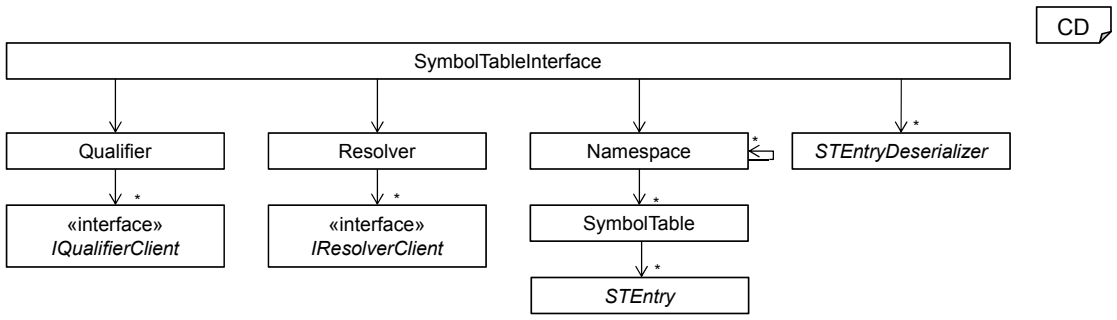


Abbildung 3.1.: Schnittstelle zur Symboltabelle (in Anlehnung an [Sch12])

Die Schnittstellen, die dort angegeben sind, werden in der Regel für jede Modellierungssprache implementiert. Die Funktionen, die sie erfüllen, werden nachfolgend kurz erläutert.

Die Struktur der Symboltabelle wird über die Klasse `SymbolTable` abgebildet. Davon können in einem `Namespace` mehrere Tabellen enthalten sein, wobei auch Namensräume ihrerseits wieder ineinander geschachtelt sein und so zu Hierarchien aufgebaut werden können. In den `SymbolTable`-Objekten werden die eigentlichen Einträge der Symboltabelle gespeichert, die von der abstrakten Klasse `STEntry` abgeleitet werden. Darin befinden sich neben den Informationen zum Eintrag selbst auch `serialize`-Methoden, über die die Einträge serialisiert und ins Dateisystem geschrieben werden können.

Da Symboltabellen über Namensräume miteinander verschachtelt sein können, ist die Auflösung eines Namens zu einem Eintrag nicht trivial, so dass spezielle Logik dafür benötigt wird. Diese ist Teil der `Resolver` Klassen bzw. `IResolverClient`-Implementierungen, die sowohl für jeden Eintrag als auch für jede Kombination von Einträgen vorhanden sind und das Auflösen entsprechend der Struktur der Einträge vornehmen.

Für die Qualifizierung von Einträgen, also das Auflösen von Referenzen und Auffinden von Symboltabelleneinträgen, auf die sich ein Eintrag bezieht, werden die `Qualifier` verwendet. Hierzu steht für jede Kombination aus Symboltabelleneintrag und potenziellem Zieleintrag eine Klasse bereit, die die Schnittstelle `IQualifierClient` implementiert.

Schließlich existieren für jeden Symboltabelleneintrag noch `STEntryDeserializer`-Klassen, über die die serialisierten Objekte wieder eingelesen und zu Einträgen konvertiert werden können. Auch diese Implementierungen existieren für jede Art Eintrag der Symboltabelle.

All diese Schnittstellen bzw. Klassen werden für jede zu entwickelnde Sprache implementiert bzw. erweitert und mit entsprechend angepasster Funktionalität ausgestattet. Dies geschieht auch für sämtliche Sprachen, die innerhalb von MontiWIS entwickelt wurden.

Ein zentrales Merkmal der Symboltabelleninfrastruktur ist die Möglichkeit, zwischen unterschiedlichen Sprachen Verbindungen herzustellen. Darüber ist es beispielsweise möglich, einen Typnamen in einer Sprache als Referenz auf einen Typ oder ein Artefakt einer anderen Sprache zu verstehen und zwischen beiden Elementen eine Verknüpfung zu erstellen. Ein Beispiel für einen solchen Mechanismus ist in Abbildung 3.2 dargestellt.

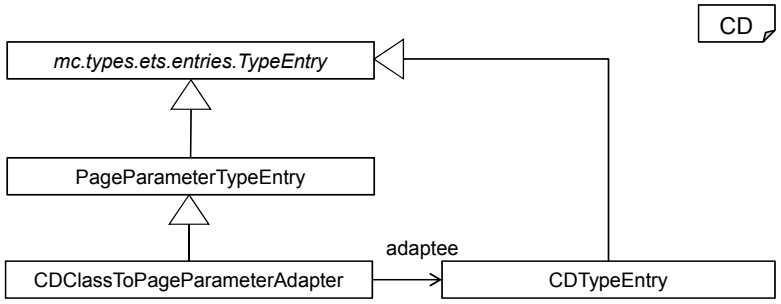


Abbildung 3.2.: Beispiel für Typadaptierung

Wie bereits beschrieben, werden Symboltabelleneinträge für die einzelnen Elemente einer Sprache erstellt und dabei, wie in Abbildung 3.1 gezeigt, per Ableitung von `STEntry` erzeugt[4]. Hier werden sowohl `PageParameterTypeEntry` (zur Repräsentation eines Seitenparameters) als auch `CDTypeEntry` (zur Abbildung einer Klasse in einem Klassendiagramm) davon abgeleitet. Um dem Umstand Rechnung zu tragen, dass der Typ eines Seitenparameters in der Regel in einem Klassendiagramm definiert wurde und um darüber hinaus die Möglichkeit zu bieten, in der Symboltabelle von dem Eintrag des Seitenparameters zum Eintrag der Klasse zu navigieren, werden sogenannte *Adapter* implementiert. Wie im Beispiel zu sehen, wird die

[4]In Abbildung 3.2 nimmt die Klasse `TypeEntry`, die selbst von `STEntry` abgeleitet wird, diese Funktion ein. Sie besitzt etwas größeren Funktionsumfang und wird daher hier im Beispiel (sowie überwiegend bei den Implementierungen in MontiWIS) verwendet.

Klasse `CDClassToPageParameterAdapter` vom Eintrag des Seitenparameters abgeleitet, so dass der Adapter anstelle dieser Klasse verwendet werden kann. Zusätzlich besitzt diese Klasse eine Assoziation zur adaptierten Klasse (`CDTypeEntry` in diesem Fall), so dass vom Adapter aus ein Zugriff auf den referenzierten Eintrag möglich ist. Die Verknüpfung von konkretem `CDTypeEntry`-Objekt mit dem darauf verweisenden `CDClassToPageParameterAdapter`-Objekt geschieht durch einen entsprechenden `IQualifierClient`.

Diese kurze Beschreibung der Symboltabellenfunktionsweise soll lediglich die Mechanismen einführen, die für das spätere Verständnis der Beschreibung der konkreten Implementierungen hilfreich und notwendig sind. Eine detailliertere Beschreibung der Symboltabelle sowie weiterer Mechanismen ist [Völ11] zu entnehmen.

3.4. Anforderungen an die Modelle und die Infrastruktur zur Modellverarbeitung

Anforderungen an Programmiersprachen wurden bereits sehr früh gestellt [Hoa73, Wir74], wobei die Autoren sich dort explizit auf das Design von allgemeinen Programmiersprachen beziehen. Konkreter an die Entwickler von domänenspezifischen Sprachen richtet sich [KKP+09], worin neben den zusammengetragenen Richtlinien aus der Literatur auch weitere hilfreiche Grundsätze des Sprachentwurfs diskutiert werden. Im Rahmen der vorliegenden Arbeit gelten diese Richtlinien als Grundsätze, nach denen die Sprachen und die entsprechende Infrastruktur entwickelt werden. Hierbei wird sowohl Augenmerk auf die Sprachinhalte gelegt, indem beispielsweise lediglich die notwendigen Konzepte der Domäne abgebildet werden (*Guideline 8* aus [KKP+09]), generell auf Einfachheit gesetzt (*Guideline 9*) und keine unnötige Flexibilität (*Guideline 10*) und Komplexität (*Guideline 11*) in die Sprachen eingebaut wird. Hinsichtlich der konkreten Syntax der entwickelten Sprachen wird sich in vielerlei Hinsicht an bestehenden und bekannten Sprachen wie beispielsweise Java orientiert (*Guideline 14*). Ferner wird Augenmerk auf einprägsame und beschreibende Notationen gelegt (*Guideline 15*), wobei ein ähnlicher Stil der Elemente angestrebt wird (*Guideline 21*). Hinsichtlich der abstrakten Syntax der Sprachen wird insbesondere auf Modularität der Sprachen (*Guideline 25*) sowie die Verwendung von Schnittstellen (*Guideline 26*) zur Entkopplung beispielsweise von Aktionsinhalten geachtet.

3.4.1. Zu modellierende Aspekte

Aus den in Abschnitt 2.2 erläuterten Charakteristika der Zieldomäne ergeben sich Anforderungen an die Modelle, die das zu entwickelnde System beschreiben. Auch an anderer Stelle wurden bereits die Aspekte definiert, die bei der Modellierung einer Webanwendung relevant sind. Die Hauptdimensionen, die unter anderem in [Fra99] festgelegt werden, sind *Struktur*, also die in der Anwendung dargestellten und verwalteten Informationen, *Navigation*, also die Möglichkeiten, wie innerhalb und zwischen den dargestellten Informationen navigiert werden kann, sowie schließlich *Präsentation*, worüber die Darstellung und Anordnung des Inhalts beschrieben wird. In [SK06] wird eine ähnliche Einteilung vorgenommen. Für jede dieser Dimensionen werden dort jeweils die Aspekte *Struktur* und *Verhalten* angenommen.

Die genannten Ebenen finden sich ähnlich in einer Vielzahl von Modellierungsansätzen und auch in den zu modellierenden Aspekten in MontiWIS wieder, wie im folgenden Abschnitt erläutert wird. Dennoch existieren einige wichtige Unterschiede, auf die an geeigneter Stelle hingewiesen wird. Nachfolgend wird lediglich überblicksartig diskutiert, welche Sprachen existieren sollten und welche Beschreibungsmöglichkeiten darin benötigt werden. Spezifische Anforderungen an die einzelnen Sprachen werden in den einzelnen Kapiteln 6 - 9 zu den konkreten Sprachen detaillierter diskutiert.

Datenmodell

Zentrale Anforderung an ein Web-Informationssystem ist die Manipulation von Daten, die selbstverständlich modelliert und beschrieben werden müssen. Dementsprechend ist es obligatorisch, dass Möglichkeiten existieren, das Domänendatenmodell beschreiben zu können, was im Wesentlichen dem Aspekt der Strukturmodellierung entspricht, die eingangs in diesem Abschnitt erwähnt wurde.

Als Sprache werden im Rahmen von MontiWIS die UML/P Klassendiagramme [Rum11, Sch12] eingesetzt, da diese den Anforderungen entsprechen und eine weitreichende Implementierung sowohl der Sprache als auch etwaiger Kontextbedingungen bereits vorliegt. Konkretere Anforderungen an die Sprache und die damit abbildbaren Sachverhalte werden entsprechend in Abschnitt 6.1.2 spezifiziert.

Aufbau von Seiten

Die Forderung, sowohl statische als auch dynamische Seiten in einem Web-Informationssystem darstellen und verarbeiten zu können, erfordert die Möglichkeit, diese auch über eine Beschreibungssprache spezifizieren zu können. Für jede darzustellende Seite sollte dabei eine eigene Datei erstellt werden, worüber die einfache Wiederverwendung von ähnlichen Seiten in unterschiedlichen Kontexten möglich wird. Der Seitenbeschreibungsaspekt entspricht der Präsentationsebene der allgemeinen Anforderungen aus der Einführung in diesem Abschnitt.

Die Beschreibung von Seiten sollte dabei möglichst kompakt sein und deklarativ erfolgen, was zur Folge hat, dass in der Spezifikation der Seite keine Logik wie beispielsweise Schleifen existiert, sondern die Kodierung von sämtlichen Elementen einer Seite per Schlüsselwort erfolgt. Dies betrifft neben den Strukturelementen, aus denen eine Seite aufgebaut ist, ebenfalls die Referenzierung von Objekten, die einer Seite übergeben werden. Details dazu finden sich in Abschnitt 7.5.2.

Ferner sollten Seiten in einer eigens dafür entwickelten Sprache spezifiziert werden. In Abschnitt 7.1.2 werden weitere konkrete Anforderungen daran beschrieben und in darauffolgenden Abschnitten, wie deren Umsetzung im Einzelnen erfolgt.

Seitenablauf und Geschäftslogik

Die Sprache zur Beschreibung von Seitenabläufen sollte in möglichst kompakter Art und Weise die Möglichkeit zur Definition der Geschäftslogik der Applikation erlauben. Ein Ablauf besteht in der Regel aus verschiedenen Teilen und sollte in geschlossenen Einheiten

definierbar sein, die untereinander beispielsweise sequenziell oder verzweigt verknüpft sind und unterschiedlichen Nutzern zugeordnet sein können. Die Ablaufbeschreibung entspricht zum Teil der Navigationsebene, die in Abschnitt 3.4.1 erwähnt wurde, geht inhaltlich jedoch wesentlich weiter, da nicht nur die Seiten oder Datenobjekte miteinander verkoppelt, sondern auch Logik darüber abgebildet werden kann.

Die Sprache sollte die Ausführung von Programmlogik vor und nach der Anzeige von Seiten unterstützen, wobei die Geschäftslogik direkt in der Sprache oder extern definiert sein kann und lediglich aufgerufen wird.

Konkretisiert werden die Anforderungen an die Sprache und ihre möglichen Implementierungen in Abschnitt 8.1.2. Dort wird ebenfalls die Wahl von *UML Aktivitätsdiagrammen* für die Spezifikation von Seitenabläufen und Workflows diskutiert sowie die Entscheidung, *Java* als Syntax für die Geschäftslogik zu verwenden.

Menüs

Um eine vollständige Sichtweise auf die Applikation zu ermöglichen und sowohl Startpunkte für Abläufe als auch Verweise auf Seiten zu verorten, muss es eine Möglichkeit zur Spezifikation von Auswahlmenüs für ebensolche Aspekte geben. Dafür werden Sprachmittel benötigt, über die Referenzen auf diese Applikationskomponenten in einer hierarchischen Art und Weise angeordnet werden können. Menüs können hierbei wiederum Untermenüs besitzen, so dass eine inhaltliche Strukturierung möglich ist.

Rechte- und Rollen im System

Neben der Möglichkeit, Menüs per Sprache zu spezifizieren, werden analog zu den Anforderungen in 2.2 ebenfalls Sprachelemente zur Definition von Nutzerrollen in der Applikation und dazu korrespondiere Zugriffsrechte benötigt. Aufgrund der semantischen Nähe von Menü- und Rechtespezifikation sollten derartige Elemente möglichst in einer Datei festgelegt werden. Daraus ergibt sich, dass die letzten beiden Aspekte Teil eines Modells sein sollten. Die zugehörige Sprache wird im Folgenden *Applikationssprache* genannt. Weder die Rechte und Rollen noch die Menüspezifikation des vorherigen Abschnitts finden eine Entsprechung in den Modellebenen aus den zu Beginn des Abschnitts erwähnten Aspekten.

Weitere Anforderungen an die Applikationssprache werden explizit in Abschnitt 9.1.2 erläutert.

Abhängigkeiten zwischen den Modellen

Trotz der expliziten Aufteilung auf unterschiedliche Sprachen existieren zwischen den einzelnen Aspekten, die in MontiWIS zu beschreiben sind, eine Reihe von Abhängigkeiten. Diese sind in Abbildung 3.3 dargestellt und werden im Folgenden näher erläutert.

Im Paketdiagramm in Abbildung 3.3 stehen die Pakete für eine Sprache, die Pfeile zwischen diesen repräsentieren Abhängigkeiten zwischen Sprachen, wobei die Lesart in Pfeilrichtung hier als *wird referenziert bzw. verwendet* zu interpretieren ist.

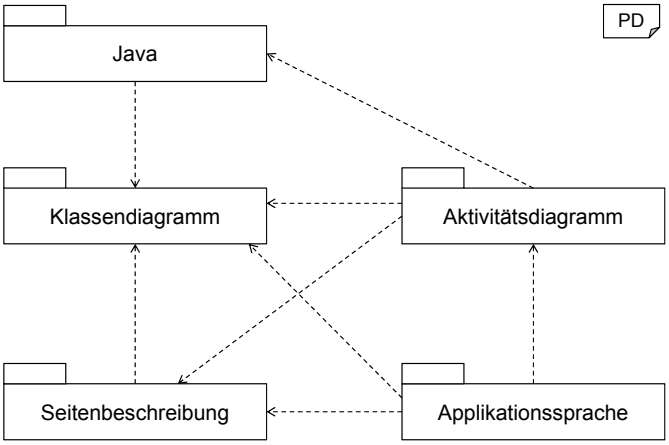


Abbildung 3.3.: Abhängigkeiten zwischen den verschiedenen Sprachen

Elemente eines Klassendiagramms werden aus sämtlichen anderen Sprachen referenziert. In der Seitenbeschreibung werden Objekte von Klassen dargestellt, in Aktivitätsdiagrammen erfolgt die Verarbeitung von Objekten als Teil der Geschäftslogik und in der Applikationssprache schließlich werden die Standardseiten für die CRUD-Operationen sowohl in Menüs eingebunden als auch mit Zugriffsrestriktionen belegt. Aus Klassendiagrammen heraus werden keine weiteren Sprachen referenziert, so dass sich darin lediglich auf die Definition der Domänendatenstruktur beschränkt wird. Schließlich werden die Klassen eines Klassendiagramms auch in der Java-Sprache verwendet, wo sie ihr Äquivalent in Java-Klassen finden.

Modelle, die in der Seitenbeschreibungssprache kodiert sind, werden in Aktivitätsdiagrammen über den Namen der Seite referenziert. Die Referenz bewirkt an dezidierter Stelle den Aufruf der Seite als Teil der Geschäftslogik. In der Applikationssprache werden rein statische Seiten, die ohne die Darstellung von Objekten nutzbar sind, sowohl in Menüs als auch in den Rechtedefinitionen verwendet.

Aktivitätsdiagramme, die zur Beschreibung von Abläufen und Geschäftslogik dienen, werden lediglich in der Applikationssprache referenziert, wo sie in einem Menüeintrag verortet sind, aber auch mit Restriktionen hinsichtlich der Aufrufberechtigung versehen werden können. In den Aktionen der Aktivitäten selbst wird Java als Aktionssprache eingebunden. Ferner existieren weitere sehr kompakte Sprachen für die Spezifikation von Inhalten von Aktionen, die in der Abbildung jedoch nicht explizit mit als Sprache aufgeführt sind, sondern als Teil der Aktivitätsdiagrammsprache gesehen werden (vgl. hierzu Abschnitt 8.2.1)[5].

Wie hier zu sehen ist, werden die verschiedenen Sprachen in der Applikationssprache integriert, indem sie zu einer Anwendung komponiert werden.

[5] In gewisser Art wird auch die Applikationssprache in Aktivitätsdiagrammen referenziert. Die in der Applikationssprache definierten Nutzergruppen, die initial in der Datenstruktur angelegt wurden, können auch in Aktionen der Aktivitätsdiagramme modifiziert werden. Da dies jedoch nur indirekt die Applikationssprache betrifft und direkt über Modifikationen der in der Datenbank abgelegten Objekte geschieht, wird an dieser Stelle auf eine Referenz auf die Applikationssprache verzichtet.

3.4.2. Anforderungen an die technische Umsetzung der Sprachen

Neben den allgemeinen Ansprüchen an den Funktionsumfang der Sprachen wurden weiterhin Anforderungen an die Implementierung der Sprachen identifiziert, die im folgenden Abschnitt kurz beleuchtet werden. Das Ziel der hier besprochenen Funktionalitäten ist es, den Nutzer bei der Erstellung und Bearbeitung der Modelle zu unterstützen.

Implementierung von Interdependenzen zwischen den Modellen

Für syntaktische und statische semantische Korrektheit sollten Kontextbedingungen definiert werden, über die sich Referenzen zwischen Modellelementen realisieren lassen. Diese werden per Adapterimplementierung, wie sie in Abschnitt 3.3.2 beschrieben wurden, umgesetzt. Darüber lässt sich die Existenz der Modellelemente prüfen, auf die an anderer Stelle (per Namenskonvention und Auflösung der vollqualifizierten Namen per Symboltabelle) verwiesen wird. Auf diese Art sollten die Abhängigkeiten, die im vorherigen Abschnitt beschrieben und insbesondere in Abbildung 3.3 dargestellt sind, überprüft werden können, was für die korrekte Funktionsweise der Applikation unerlässlich ist.

Korrektheitsüberprüfungen

Neben der einfachen Referenz von Modellelementen untereinander sowohl innerhalb einer Sprache aber auch aus anderen Sprachen heraus existieren weiterhin Konsistenzbedingungen, die sich nicht ausschließlich über Namensgleichheit und korrekte Auflösung der Elemente überprüfen lassen. Derartige Bedingungen umfassen beispielsweise Parameterkongruenz, wie sie bei Aufruf von Seiten oder Aktionen gelten muss. Auch derartige Bedingungen sollten im Rahmen der Modellüberprüfung gesichert werden.

Die Überprüfung findet in der Regel über die Symboltabelleninfrastruktur statt. Im Falle von fehlschlagenden Konsistenzprüfungen werden generell Warnungen ausgegeben, über die der Modellierer hinreichend verständlich über Probleme in den Modellen informiert wird.

Nutzerunterstützung im Editor

Für die Ersteller der Modelle ist es wünschenswert, dass ihnen bei der Bearbeitung Hilfestellung innerhalb der Entwicklungsumgebung zur Verfügung steht. So ist beispielsweise hilfreich, wenn Schlüsselwörter der Sprache bei der Eingabe hervorgehoben werden (*syntax highlighting*), wenn zentrale Artefakte übersichtlich in einer Baumstruktur dargestellt werden und wenn syntaktische Fehler direkt bei der Eingabe angemerkt werden.

3.5. Verwandte Arbeiten im Bereich des modellbasierten Web Engineerings

Im folgenden Abschnitt wird nun ein kurzer Überblick über andere Ansätze gegeben, die in der gleichen Domäne wie MontiWIS, nämlich der Modellierung und Generierung von Web-

Informationssystemen angesiedelt sind. Die Aktualität dieser Thematik zeigt beispielsweise auch [HKRV11].

Eine Übersicht über modellbasierte Ansätze zur Modellierung und Generierung von Webapplikationen ist in [SRS+08] zu finden. Hier werden einige Ansätze anhand einer Fallstudie verglichen und hinsichtlich ihrer Ausdrucksmächtigkeit, Ausgereiftheit und Verbreitung beurteilt, wobei ein Fokus auf Adaptierbarkeit der Anwendung hinsichtlich verschiedener Szenarien liegt. Ebenso gibt [SK06] einen Überblick über das Thema der Modellierung von Webapplikationen und vergleicht einige Ansätze hinsichtlich des Schwerpunktes der Modelle und ihrer Ausdrucksmächtigkeit. [WD08] untersucht primär die Ansätze UWE und WebML auf ihre Eignung für die Abbildung moderner Webtechnologien und die Unterstützung von Aspekten wie clientseitigem Scripting, Session und Cookies. [JPV09] gibt einen sehr weitgefassten Überblick über verschiedenste Webentwicklungsmethodiken und evaluiert diese hinsichtlich ihrer Eignung für den produktiven Einsatz anhand empirischer Experimente und Interviews mit Studenten. Die Ansätze werden kategorisiert z. B. bezüglich ihres primären Fokus, ihrer Herkunft und ihres Automatisierungsgrades und schlussendlich wird dort festgestellt, dass viele Ansätze nicht für Personen mit eher geringem formalen Hintergrund geeignet sind. [AELH13] vergleicht verschiedene Ansätze insbesondere hinsichtlich ihres Sprach- und Funktionsumfangs und potenzieller Interoperabilitätsmöglichkeiten auf Metamodell-Ebene, wobei insbesondere auch Defizite in diesen Bereichen aufgezeigt werden. Ferner wird darin ein historischer Abriss über die verschiedenen Ansätze gegeben.

Einen Überblick über die verschiedenen Teilbereiche des modellbasierten Web Engineerings gibt [MRV08]. Die behandelten Aspekte sind im Wesentlichen eine Obermenge der Facetten, die in MontiWIS beschrieben werden und umfassen beispielsweise die Datenmodellierung, die Sichtenbeschreibung und die Geschäftslogik, wobei jedoch auch Funktionalitäten wie die Beschreibung der Adaptierung von Inhalten an externe Gegebenheiten aufgeführt werden, die nicht explizit von MontiWIS unterstützt werden. Ferner werden dort Bestrebungen behandelt, die auf eine Interoperabilität von Ansätzen durch deutlicheren Fokus auf die MDA und die Verwendung von UML Profilen abzielen. Ebenfalls mit der Interoperabilität von verschiedenen Ansätzen hat sich [KK03] befasst. Darin wurde ein Metamodell entwickelt, das zunächst auf UWE zugeschnitten ist, aber nach Ansicht der Autoren auch für weitere Webentwicklungsmethoden passen kann.

Als früher Ansatz zur Modellierung von Webapplikationen kann etwa die *Hypermedia Design Method (HDM)* [GPS91, GPS93] gesehen werden. Diese Methodik erlaubt, Daten zu spezifizieren und sie miteinander zu verknüpfen, Sichten darauf zu definieren und letztlich in Code umzusetzen. Hierbei wird primär auf die Hypermediastruktur von Applikationen abgezielt und weniger auf komplexe Geschäftslogik.

HDM evolvierte zu *W2000* [BCMM06], welches auf dem W2000 Metamodell (verwendet wird hier für die Implementierung MOF) basiert. Beschränkungen auf dem Metamodell werden über die Object Constraint Language (OCL) spezifiziert. Die Aufteilung der Modelle erfolgt in die Bereiche *Information* (definiert das Datenmodell über eine Art von Klassendiagrammen), *Navigation* (über sogenannte Cluster, die zusammengehörige Seiten gruppieren) und *Services* (definiert die Applikationslogik durch eine Art OCL) sowie *Presentation* (spezifiziert, wie Daten und Dienste dem Nutzer präsentiert werden). Die Verarbeitung der Modelle erfolgt

über Transformationen. Informationen über lauffähige generierte Anwendungen waren nicht zu finden. Auch dieser Ansatz zielt primär auf die Darstellung von Informationen und die Navigation per Hyperlink zwischen diesen ab.

Die *Relationship Management Methodology (RMM)* [ISB95] ist ebenfalls auf die Entwicklung von Hypermedia-Applikationen ausgelegt. Dort existieren Möglichkeiten zur Datenmodellierung durch die Verwendung von Entity Relationship (ER)-Modellen und zur Definition von Sichten, die sowohl durch besagte ER-Diagramme als auch sogenannte *Slices* realisiert werden, die Attribute zu sinnvollen Teilen gruppieren. Der Zusammenhang wird durch sog. *Access primitives* realisiert, bei denen die Links zwischen verschiedenen Objekten und innerhalb eines Objekts unterschieden werden. RMM schlägt eine Methodik vor, wie und in welchen Schritten die Hypermediaapplikation zu entwickeln ist und bietet damit eine methodische Richtlinie für den Entwicklungsprozess.

Hera [HvdSB+08] verwendet das Resource Description Framework (RDF) zur Modellierung. Dabei wird unterschieden in *domain model* zur Spezifikation der Datenstruktur, *application model* zur Beschreibung der Hypermedia-Navigationsstruktur und *context model* für die explizite Abbildung der nutzerbezogenen Kontextinformationen. Dies wird aufgeteilt in *session* (relevant für eine bestimmte Session), *user* (enthält relevante, sessionübergreifende Nutzerinformationen) sowie *global data* (nutzerübergreifende Daten, die global genutzt werden). Während Daten- und Kontextmodell über RDF abgebildet werden, kommt für das Applikationsmodell eine semantische Abfragesprache zum Einsatz. Weiterhin lassen sich über die *presentation models* verschiedene Sichten für verschiedene Ausgabeformate wie beispielsweise HTML oder WML spezifizieren, die zur prototypischen Generierung der Applikation herangezogen werden. [BFH06] schlagen eine Workflowerweiterung für Hera vor, wobei hierbei Aktivitätsdiagramme zur Modellierung von Abläufen genutzt werden, die dann in die Hera-eigenen Primitive übersetzt werden. Die Modellierung findet unter anderem in Microsoft Visio statt und die Implementierung wird lediglich prototypisch durch verschiedene Generatoren unterstützt.

Autoweb [FP00] verwendet einen Ansatz namens *HDM-lite* (welches, wie der Name vermuten lässt, auf HDM basiert) zur Modellierung und Generierung von Webapplikationen. Ausgehend von einem Datenmodell, spezifiziert über eine Modellart ähnlich den ER-Modellen, werden Navigationsschemata definiert. Zusätzlich lassen sich Seiten- und Seitenaufbau modellieren, was letztlich zur Generierung einer prototypischen Applikation primär zur Dateneingabe- und Modifikation führt. Autoweb evolvierte später zu WebML, was in Kürze in diesem Abschnitt beschrieben wird.

Mit der *Web Application Extension (WAE)* stellt Conallen [Con03] einen sehr umfassenden Ansatz für die Beschreibung von Webapplikationen auf Basis von UML-Profilen vor. Anwendungsfalldiagramme werden zur globalen Spezifikation von Anwendungsfällen verwendet, Aktivitätsdiagramme zur weiteren Verfeinerung dieser. Bildschirmseiten werden mit einem abgewandelten Klassendiagramm modelliert, wobei gerichtete Assoziationen die Navigationsmöglichkeiten zwischen diesen Seiten angeben. Interaktionen zwischen Nutzer und System werden typischerweise über Sequenzdiagramme abgebildet. Verfeinerungen des Designs werden über spezielle Stereotypen ausgedrückt, die implementierungsnahe Artefakte repräsentieren. Für die behandelten Modelle sind Implementierungsbeispiele auf

JSP-Codebasis angegeben, die den jeweiligen Modellkonstrukten entsprechen. Diese müssen jedoch schematisch und vor allem händisch aus den Modellen abgeleitet werden.

Die *Form-Oriented Analysis* [DW05] fokussiert auf die Betrachtung von formularbasierten Applikationen und deren Modellierung. Dazu werden sowohl Daten- als auch Seitenaufbau und vor allem die Navigation zwischen den Seiten zunächst durch eine grafische Notation skizziert, aber final in textueller Notation beschrieben. Hierbei findet explizit die Trennung zwischen server- und clientseitiger Logik statt. Die Modellierung erfolgt über eine eigene Notation, die entfernt an die OCL angelehnt ist. Spezielle Datentypen werden unterstützt und auch die Erzeugung eines Menüs ist möglich, jedoch fehlen Aspekte wie explizite Workflow- und Nutzermodellierung. Die automatische Generierung von JSP-Seiten und zugehöriger Logik sowie das Ableiten von Modellen aus existierenden Anwendungen wird prototypisch unterstützt.

Der Ansatz von Nussbaumer et al. [NFG06b, NFG06a] verwendet domänenspezifische Sprachen, die aber an existierende Sprachen und Notationen angelehnt sind. Dabei wird mit der *Domain Specific Model* genannten DSL ein Konzept vorgeschlagen, das als Metamodell fungiert und über dem mit den sog. *Domain Interaction Models* verschiedene konkrete Notationen spezifiziert werden können, die unter Umständen auch unterschiedliche Notationen (z. B. BPMN oder Petri-Netze) verwenden. Benutzerschnittstellen werden über eine Kombination von XForms [XFo] und Petri-Netzen [Bau96] modelliert, die zur Beschreibung von dynamischen Oberflächen verwendet werden. In einer weiterentwickelten Version des Ansatzes verschiebt sich der Fokus auf die Modellierung und Ausführung von Workflows [FBNG07, Fre09], die in einer proprietären Workflowumgebung verarbeitet werden.

Die *OO-Methode* [PGIP01] ist nicht im Kontext von Web-Applikationen angesiedelt,[6] bietet aber dennoch die Möglichkeit, Datenschemata und das Verhalten von Klassen zu spezifizieren. Dazu werden sowohl Klassendiagramme (für die statischen Aspekte des Systems) als auch *State Transition Diagrams* und *Interaction Diagrams* für die dynamischen Aspekte des Systems verwendet. Diese werden auf eine formale Spezifikationssprache abgebildet, für welche ihrerseits wiederum Richtlinien existieren, wie ihre Elemente in Code zu überführen sind. Sowohl die Modellierung als auch die Generierung wird über ein Werkzeug namens OlivaNova [Oli] unterstützt, das jedoch scheinbar nicht mehr zur Verfügung steht und in ein Nachfolgewerkzeug Integranova aufgegangen ist. Das System, das daraus generiert wird, ist eine Desktop-Applikation.

Die *Object-Oriented-Hypermedia (OO-H)*-Methode basiert auf der OO-Methode, von der sie die Funktionalitäten für die Modellierung von Klassen und deren Verhalten erbt. Zusätzlich dazu werden in OO-H sogenannte *Navigational Access Diagrams* zur Spezifikation von nutzerspezifischen Sichten auf das Datenmodell verwendet, welche ebenfalls Verbindungen zwischen verschiedenen Seiten sowie die hierarchische Dekomposition von Seitenabläufen erlauben. In [KKCM04] werden Mechanismen vorgeschlagen, wie aus Aktivitätsdiagrammen die Navigational Access Diagrams abgeleitet werden können. In [GCP01] werden zusätzlich sogenannte *Abstract Presenation Diagrams* vorgeschlagen, die den Aufbau der generierten Seiten durch die Anwendung von Transformationen verfeinern können. Die Generierung er-

[6] Da jedoch einige weitere Ansätze aus dem Bereich der Webmodellierung darauf basieren, wird hier dennoch kurz darauf eingegangen.

folgt über ein Werkzeug namens VisualWade, welches für verschiedene Plattformen passende Prototypen erzeugen kann.

Die *Object-Oriented Web Solution (OOWS)* [FPP+08, PFPA06] basiert ebenfalls auf der OO-Methode [PGIP01] und erweitert diese um webanwendungsspezifische Aspekte. Zusätzlich zu den Modellen der OO-Methode existieren in OOWS sog. *User Diagram Models*, die eine hierarchische Einordnung von Nutzern ermöglichen. In *Navigational Contexts* wird spezifiziert, was in bestimmten Sichten in welcher Art und Weise dargestellt wird. Darin enthalten sind in *Abstract Information Units* Attribute und Methoden, die letztlich Sichten auf Klassen aus dem Klassendiagramm darstellen. Zusätzlich erlaubt OOWS über sog. *Presentation Models* Seiten und Sichten auf Daten zu definieren. Hierzu werden verschiedene Darstellungsmöglichkeiten angeboten, die bestimmte Paradigmen (z. B. Listenansicht, Baumansicht) unterstützen. Modelle der OO-Methode werden über OlivaNova [Oli] generiert, wobei dieser Generator für OOWS angepasst wurde. Prozesserweiterungen für OOWS wurden in [TP06] und [TGP12] vorgestellt, wobei aus einer Prozessbeschreibungssprache einerseits Oberflächenelemente von OOWS und andererseits Beschreibungen zur Ausführung in einer Prozessengine generiert werden. Die Einbindung externer Applikationen ebenso wie das Bereitstellen von Schnittstellen in das eigene System per Webservice wurde in [TPRV05] erläutert. Dazu wurden die in OOWS vorhandenen Konstrukte um solche erweitert, die die Steuerung und Ausführung von Prozessschritten erlauben.

AndroMDA ist ein Eclipse-basiertes Werkzeug [Ecl], das im Wesentlichen die Möglichkeit zur Verarbeitung von UML-Modellen bietet, die mit einer beliebigen Anwendung erzeugt wurden und als Eingabe in XMI [Obj13a] vorliegen. Sie werden durch eine sog. Cartridge verarbeitet, die die Codegenerierung für eine bestimmte Laufzeitumgebung übernimmt. Cartridges existieren bereits für eine Vielzahl von Zielarchitekturen, so beispielsweise für Java EE-Anwendungen oder für Rich-Client-Applikationen. Ein vollständiges Beispiel zur Entwicklung mit der MDA unter Verwendung von AndroMDA ist in [ZW06] zu finden. AndroMDA scheint stabil zu sein, jedoch findet nach Updateverlauf der Homepage keine aktive Entwicklung mehr statt.

Ebenfalls auf den Prinzipien der MDA basiert *Netsilon* [MSFB05], worin sowohl Datenstruktur, einfache Navigation und Präsentation auf Basis eines einheitlichen Metamodells beschrieben werden können. Die dazu spezifizierte Aktionssprache *Xion* findet Anlehnung an Java und OCL und dient zur Beschreibung einfacher Geschäftslogik. Die Generierung einer vollständigen Anwendung findet über mehrere Transformationsschritte statt und zielt auf PHP und Java als Zielsprachen ab. Workflows oder verschiedene Nutzer werden nicht nativ unterstützt.

Der Ansatz in [HP05] bezieht sich zwar auch auf die MDA, bietet jedoch eine sog. model-driven runtime, über die sich plattformunabhängige Modelle ausführen lassen. Als Eingabe dienen dabei Klassen- und Zustandsdiagramme zur Beschreibung von Objekten und Zuständen darauf sowie OCL zur Beschreibung von Abfragen, die durch die Laufzeitumgebung interpretiert und in Webseiten umgesetzt werden. Die Zieldomäne bei diesem Ansatz sind im Wesentlichen Applikationen, die CRUD-Funktionalität bereitstellen.

Die *Web Semantics Design Method (WSDM)* [DCP08] definiert eine Entwicklungsmethodik, über die nutzerzentriert eine Webanwendung entwickelt werden kann. In dieser werden

zunächst freitextlich die Zielsetzungen des Systems herausgearbeitet. In einem nächsten Schritt werden Nutzerklassen über ein semiformales Vorgehen klassifiziert, wobei jede Nutzerklasse charakterisiert wird durch Aktivitäten, die ein Nutzer ausführen kann oder auch Zugriffsrechte, die ihm zugeordnet sind. In einem nächsten Schritt werden sogenannte *task models* erstellt, die hierarchisch gegliedert die Aufgaben und Aktivitäten der Webapplikation darstellen, wobei diese entweder automatisch durch die Anwendung oder manuell durch den Nutzer ausgeführt oder aber hierarchisch dekomponiert sein können. Diese Aufgaben haben verschiedenartige Abhängigkeiten untereinander, die im Wesentlichen den üblichen Workflowkontrollstrukturen, wie z. B. parallele oder alternative Ausführung, entsprechen. Ausgehend von einzelnen Tasks wird das Objekt- und Datenmodell abgeleitet, welches in der Web Ontology Language (OWL) spezifiziert wird. Tasks werden weiterhin dekomponiert in *navigational models*, in denen konkrete Komponenten und Seitenabläufe beschrieben werden, wobei zwischen einzelnen Seiten und externer Geschäftslogik unterschieden wird. Abschließend müssen die einzelnen Komponenten zu Seiten zusammengefasst werden, welche in nachfolgenden Schritten hinsichtlich der Darstellung einzelner Datenarten über Templates spezifiziert werden. Die Modelle werden prototypisch über verschiedene Transformationsschritte in Code überführt [PCY+05].

Die *Object-Oriented Hypermedia Design Method (OOHDM)* [RS08, RS06] benutzt einen fünfstufigen Prozess zur Beschreibung der Applikation. Im ersten Schritt werden über *User Interaction Diagrams (UID)* Benutzerinteraktionen und insbesondere Informationsflüsse zwischen Benutzern modelliert. Diese sind angelehnt an Anwendungsfalldiagramme, jedoch angereichert um Konstrukte, die den Informationsfluss beschreiben. Im zweiten Schritt (*Conceptual Design*) wird über (leicht modifizierte) UML Klassendiagramme die Datenstruktur der Applikation modelliert. Die Klassendiagramme werden systematisch (aber dennoch manuell) aus den UIDs abgeleitet. Im dritten Schritt (*Navigational Design*) werden Sichten über dem Datenmodell definiert, welche dann zur Anzeige verwendet werden. Zusätzlich existieren *Navigational Contexts*, die die Navigation (vor / zurück) in einer Menge von Objekten sowie die Spezifikation von Menüs und Indizes erlauben. Auch hierzu existieren Handlungsanweisungen, wie sich diese Kontexte aus UIDs ableiten lassen. Im vierten Schritt (*Abstract Interface Design*) werden Widgets und Elemente eingesetzt, die ihrerseits mit dem Navigational Design verknüpft werden. Aus eher abstrakten Blöcken werden Interaktionen zusammengesetzt, die mit den Navigationsklassen verbunden werden. Die Spezifikation erfolgt über die Ontologiesprache OWL. Der fünfte Schritt (*Implementation*) erzeugt Laufzeitobjekte und führt diese in einer Serverumgebung aus, wozu ein Framework namens *HyperDe* verwendet wird, das auf Ruby on Rails aufsetzt. Ferner soll es auch möglich sein, Code für andere Zielplattformen aus den OOHDM Modellen zu generieren. Ein ausgewachsener grafischer Editor zur Erzeugung von grafischen Modellen existiert für OOHDM nicht [Jak11].

UWE basiert überwiegend auf standardkonformen UML Profilen. Ein Überblick darüber ist in [KKZB08] zu finden. Ansätze für die Modellierung von Geschäftsprozessen sind in [KKCM04] beschrieben, zusätzlicher Fokus auf Toolsupport wird in [KKZH04] gelegt. Implementiert wurde UWE beispielsweise über einen Generator, der eine JSF Applikation erzeugt. Dies ist überblicksartig in [KKK09] und detailliert in [Kro08] beschrieben. Anforderungen werden über Anwendungsfalldiagramme modelliert, wobei bei den Aktoren beispielsweise

zwischen einem *anonymen Nutzer*, einem *registrierten Nutzer* und einem *Websystemadministrator* unterschieden wird. Anwendungsfälle werden unterschieden in *Navigationsanwendungsfall* (Browsen der Applikation, Suchen, Ansehen von bestimmten Seiten), *Prozessanwendungsfall* (gewöhnlicher Anwendungsfall, bestehend aus Geschäftslogik, potenziell gekapselt in Transaktionen) und *Personalisierungsanwendungsfall* (ausgelöst durch Nutzerverhalten). Eine Verfeinerung der Anwendungsfälle findet über Aktivitätsdiagramme statt, die ebenfalls über Stereotypen näher spezifiziert werden. Das *Content model* wird modelliert über Klassendiagramme und beinhaltet die Domänendatenstruktur, das *User model* enthält die Struktur, die für Nutzer bzw. Nutzerverhalten relevant ist. Beide Modelle sind an sinnvoller Stelle per Assoziation verbunden. Die *Navigationsstruktur* wird ebenfalls über Klassendiagramme abgebildet, deren Stereotypen zur Unterscheidung genutzt werden, welche Art der Darstellung eine bestimmte Klasse repräsentiert (z. B. Menü, Liste von Objekten). Links werden über gerichtete Assoziationen modelliert. Ein besonderer Stereotyp dient zur Referenzierung von *Prozessklassen*, die als Einstiegs- und Endpunkt für Geschäftsprozesse dienen. Zur weiteren Spezifikation von Abläufen werden Aktivitätsdiagramme verwendet. Die Oberfläche wird zusammengesetzt aus Elementen wie Text, Bild, Link, Formularfeldern und anderen üblichen Elementen. Dazu kommt eine spezielle Darstellung von Kompositionen von Klassen zum Einsatz, die jedoch nicht von allen Editoren unterstützt wird. Der Verfeinerungsprozess in UWE erfolgt analog zum MDA-Ansatz über Transformationen vom CIM, über das PIM hin zum PSM und schlussendlich in Code durch die Verwendung von Query View Transformation (QVT)- oder Atlas Transformation Language (ATL)-Transformationen. Einen Überblick über die in UWE verwendeten Transformationen gibt [Koc07]. Je nach betrachtetem Ansatz im Laufe der Entwicklung von UWE lassen sich die generierten Zwischenergebnisse beispielsweise in ArgoUWE [KKZH04] durch nutzereigene Anteile anreichern, so dass eine erneute Generierung zu Informationsverlust führen würde. In späteren Ansätzen wie UWE4JSF [Kro08, KKK09] findet jedoch eine ausgefeiltere Trennung von generierten Teilen und anzupassenden Artefakten statt. Eine weitere Umsetzung von Transformationen mit Spring [Spr] als Zielplattform fand in [Kra07] statt. Im aktuellen Status findet laut [BKS14] lediglich eine partielle Codegenerierung statt, da die Anpassung des Codegenerators auf neuartige Technologien und Anforderungen mit zu hohem Aufwand verbunden war.

WebDSL basiert als einziger der hier beschriebenen Ansätze ausschließlich auf textuellen, domänenspezifischen Sprachen, die zur Spezifikation von Datenmodell, Seiten, Aktionssprache und Workflows [HVV08] eingesetzt werden. Damit ist er zumindest hinsichtlich Syntax nahe MontiWIS anzusiedeln. Ein genereller Überblick wird in [Vis08a] bzw. [Vis08b] gegeben. Für das Datenmodell wird eine eigene Entity-Notation verwendet. Seiten werden ebenfalls über eine DSL codiert, welche eine Aktionssprache einbindet. WebDSL bietet jedoch wesentlich mehr Elemente zur Spezifikation von Seitenelementen als MontiWIS, was allerdings auch einen größeren Aufwand bedeuten kann. Anders als MontiWIS, welches einzelne Dateien für jedes Modellelement (d. h. beispielsweise für Klassendiagramme, Aktivitäten, Seiten) fordert, erlaubt WebDSL, verschiedene Aspekte innerhalb einer einzigen Datei zu spezifizieren. Eine Aufteilung in mehrere Dateien ist zwar möglich, nicht jedoch die Regel. Die Transformationsmechanismen in WebDSL werden in [HKGV09] näher erläutert, die Implementierung von Zugriffsrechten beschreibt [GV08] und Validierungsmechanismen

von Daten werden detailliert in [GV10] behandelt. Im Gegensatz zu MontiWIS unterstützt WebDSL keine Workflowkonzepte, auch wenn in [HVV08] mit WebWorkFlow Ansätze dafür existieren.

Einer der umfangreichsten und ausgereiftesten Ansätze zur Modellierung von Webapplikationen ist *WebML* [BCFM08, CFB+03]. Dieser Ansatz verwendet generell ein grafisches Modell mit domänenspezifischen Elementen. Zusätzlich existiert eine textuelle Notation, die jedoch in einer überwiegenden Anzahl der Publikationen nicht erwähnt wird, weswegen primär das grafische Pendant verwendet wird. Das *Conceptual Model* in WebML besteht aus Datenstruktur und Hypertextstruktur. Die Datenstruktur wird über ER-Diagramme modelliert. Die Hypertextstrukturen beschreiben Seiten, die aus sog. *Content Units* zur Darstellung von Daten in einer bestimmten Form und *Operation Units* zur Abbildung von Operationen wie Navigation oder Login/Logout oder dem Lesen oder Speichern von Daten, bestehen. Damit lassen sich relativ frei Seiten spezifizieren, wobei dies jedoch eine gewisse Komplexität nach sich zieht und viele Schritte sehr explizit modelliert werden müssen. Als Editor wird das proprietäre Werkzeug *WebRatio* [Webd] verwendet. Ansätze zur Prozessmodellierung wurden in [BCF06] erläutert. Hierbei werden Konstrukte zur Steuerung von Prozessen als Bausteine zusätzlich zu den verfügbaren Komponenten eingeführt, anhand derer sich Prozesse abbilden lassen. Spezifischere Modelle können über Transformationen halb automatisch z. B. aus Business Process Modeling Notation (BPMN)-Modellen erzeugt werden (siehe hierzu z. B. [BCC+02]). WebML wurde in [BC06] mit Möglichkeiten ausgestattet, um Aspekte einer Rich Internet Application (RIA) zu modellieren. [CDMF07] beschreiben Konzepte, wie innerhalb der Datenstruktur Kontextinformationen modelliert werden können, die dann zur Anzeige nutzer- bzw. kontextspezifischer Seiten herangezogen werden. WebML stellt einen sehr mächtigen Ansatz dar, der jedoch die explizite Spezifikation sämtlicher Logik auf den einzelnen Seiten erfordert, wohingegen bei MontiWIS sinnvolle Standardannahmen für die Darstellung von Datentypen getroffen werden.

Kapitel 4.

Ablauf der Generierung

Ziel dieses Kapitels ist es, eine Zusammenfassung des MontiWIS Ansatzes, seiner Komponenten und den allgemeinen Ablauf bei der Modellverarbeitung zu geben. Damit soll es möglich sein, die spezifischeren Erläuterungen der späteren Abschnitte entsprechend in den Gesamtablauf einordnen zu können. Die Details der einzelnen Schritte werden in den Kapiteln 6 - 9 erläutert.

4.1. Gesamtüberblick über den Generierungsprozess

Wie bereits in Abschnitt 3 beschrieben, wird in MontiWIS das Framework MontiCore zur Verarbeitung von textuellen Modellen eingesetzt. Daher orientiert sich der Ablauf der Modellbearbeitung auch an dem Vorgehen, das durch die MontiCore-Infrastruktur induziert wird. Hierbei wird jedoch nicht auf die Methodik gesetzt, die im Rahmen der Dissertation von Martin Schindler [Sch12] entstanden ist, was mehrere Gründe hat. Zum einen fand die Entwicklung von MontiWeb[7][DRRS09] und im Anschluss daran MontiWIS zu einer Zeit statt, als die Entwicklung der Infrastruktur im Rahmen besagter Arbeit noch nicht abgeschlossen war. Zum anderen wird in MontiWIS eine *Zwischenstruktur* als Eingabe für die Codegeneratoren verwendet, die im Gegensatz zu den in [Sch12] verwendeten AST-Strukturen nachträglich noch modifiziert und zum Teil während der Bearbeitung aus anderen Modellelementen erzeugt wird, was nicht dem Konzept der Dissertation von Martin Schindler entspricht.

4.1.1. Überblick über beteiligte Komponenten

Die verschiedenen Komponenten von MontiWIS, die verwendeten Artefakte sowie die bei der Entwicklung beteiligten Akteure sind in Abbildung 4.1 dargestellt.

Die in der Abbildung dargestellten Akteure korrespondieren mit denen, die in [KRV06] eingeführt und erläutert wurden. Unter *Sprachentwickler* (language developer in [KRV06]) ist die Person zu verstehen, die für die Entwicklung der Sprache in Absprache mit den *Produktentwicklern*, also den letztlichen Domänenexperten und Endanwendern, zuständig ist. Produktentwickler (product developers) beschreiben das zu entwickelnde Web-Informationssystem über eine Anzahl von Modellen und nutzen die Werkzeuge, um daraus eine lauffähige Anwendung zu generieren. Die Anwendung zur Modellverarbeitung selbst wird

[7] MontiWeb war ein Vorgängersystem von MontiWIS, das zwar auf eine ähnliche Domäne abzielte und ebenfalls MontiCore verwendete, jedoch auf Apache Tapestry [Tap] als Zielplattform setzte und nur in einigen wenigen Teilen in MontiWIS einfloss.

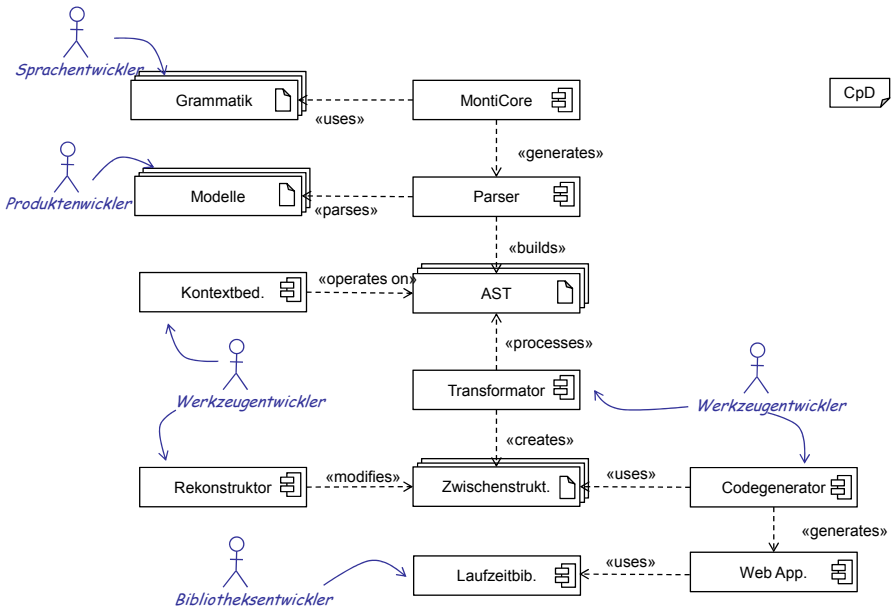


Abbildung 4.1.: Komponenten von MontiWIS und Rollen bei der Entwicklung

durch den sog. *Werkzeugentwickler* erstellt und angepasst, womit dieser Rolle eine zentrale Funktion zukommt. Schließlich übernimmt der *Bibliotheksentwickler* die Aufgabe, statische Komponenten zu implementieren, die in der generierten Applikation zum Einsatz kommen und dort als Infrastruktur zur Verfügung stehen. Bei MontiWIS sind die Aufgaben zwar ebenfalls getrennt zu betrachten und schlagen sich in unterschiedlichen Komponenten nieder, die Entwicklung jedoch wurde primär zentral an einer Stelle vom Autoren dieser Arbeit vorgenommen.

Die Komponenten in Abbildung 4.1 sind im Wesentlichen vertikal chronologisch angeordnet.

Zunächst werden die *Grammatiken* durch das Framework *MontiCore* verarbeitet. Dadurch werden sowohl ein *Parser* für die *Modelle* als auch weitere Infrastruktur für die Verarbeitung von Modellen erzeugt. Eine detailliertere Beschreibung dieser Mechanismen erfolgt in Abschnitt 4.2. Durch das Parsen der *Modelle* wird ein *Abstract Syntax Tree (AST)* erzeugt, auf dem wiederum *Kontextbedingungen* überprüft werden, um (grundlegend) die Korrektheit und Sinnhaftigkeit der Modelle und deren Zusammenspiel mit anderen Modellen zu verifizieren. Näheres dazu wird in Abschnitt 4.3 beschrieben. Auf dem AST operiert ebenfalls die *Transformator*-Komponente, die den AST in eine spezielle *Zwischenstruktur* überführt, welche sich besonders als Eingabe für den templatebasierten Generator eignet. Die Notwendigkeit der Zwischenstruktur und deren Verwendung werden kurz in Abschnitt 4.4 diskutiert. Nach dem Aufbau der Zwischenstruktur werden darauf über den *Rekonstruktor* noch einige Umformungen vorgenommen, wodurch beispielsweise Standardverhalten erzeugt wird. Detaillierter wird darauf in Abschnitt 4.5 eingegangen. Die transformierte und angereicherte Zwischenstruktur dient anschließend als Eingabe für den *Codegenerator*, der aus den Objekten die Dateien des Zielsystems erzeugt, die letztlich die *Webapplikation* bilden. Genauer werden

diese Mechanismen in Abschnitt 4.6 erläutert. Die generierte Webapplikation ihrerseits verwendet *Laufzeitbibliotheken*, die zur Infrastruktur der Anwendung gehören bzw. statische, nicht von den Modellen abhängige Teile enthalten.

4.1.2. Zusammenspiel der Komponenten

Aufgrund der Abhängigkeiten zwischen den Modellarten (vgl. Abbildung 3.3) und damit auch den Modellen selbst werden für die Prüfung von Kontextbedingungen in der Regel mehr Informationen benötigt, als in einem einzelnen Modell vorhanden sind. Der Zugriff auf referenzierte Modelle bei Bedarf wäre zwar über den sogenannten `Modelloader` möglich, durch den unter Angabe eines vollqualifizierten Namens ein Modell sowie die daraus erzeugten Symboltabelleneinträge nachladbar sind, jedoch wurde von der Verwendung dieser Art Verarbeitung von Modellen Abstand genommen, da in der Regel ohnehin Symboltabelleneinträge erzeugt werden müssen, die sich bei Bedarf über die Symboltabelleninfrastruktur nachladen lassen. Stattdessen werden in einem zweistufigen Generierungsprozess zunächst die Einträge der Symboltabelle sämtlicher Modelle erzeugt und im Dateisystem abgelegt, so dass sie in einem zweiten Schritt zur Prüfung der Kontextbedingungen zur Verfügung stehen. Der aus diesem Vorgehen abgeleitete Generierungsprozess in MontiWIS ist im Sequenzdiagramm in Abbildung 4.2 dargestellt.

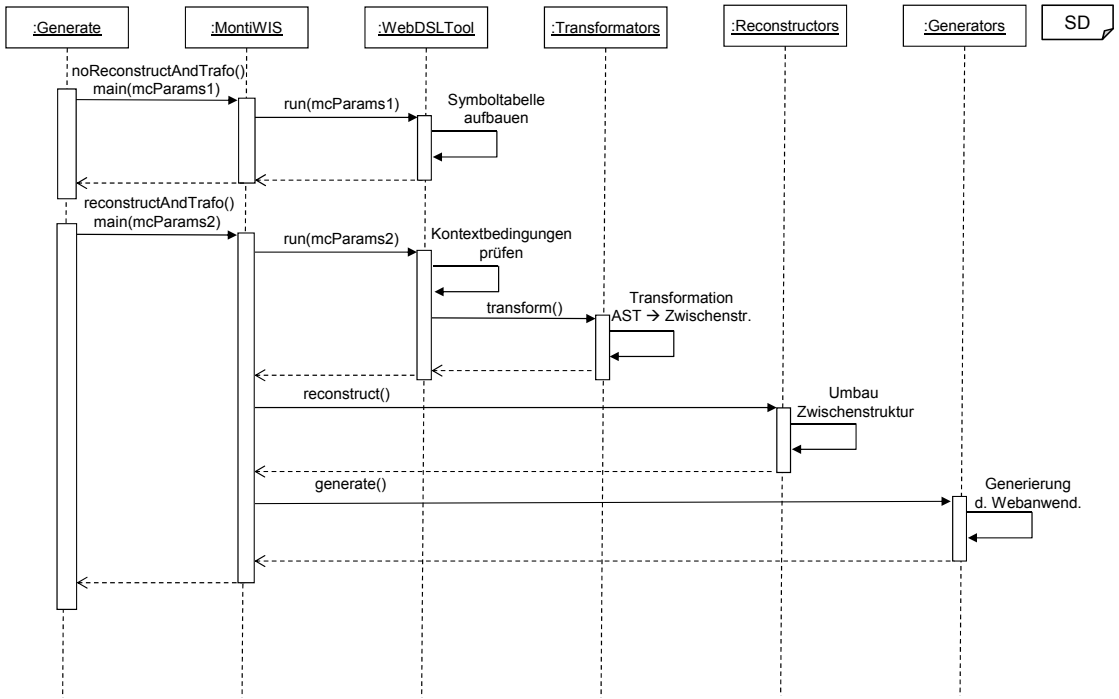


Abbildung 4.2.: Zweistufiger Generierungsprozess in MontiWIS

Die Lebenslinien in Abbildung 4.2 repräsentieren die Klassen bzw. fassen bei `Transformators`, `Reconstructors` und `Generators` die Menge der modellverarbeitenden Klassen zusammen.

Die beiden Prozesse für Modellverarbeitung und Generierung werden jeweils von der Klasse `Generate` aus initiiert. Zunächst wird der Ablauf über den Aufruf der Methode `noReconstructAndTrafo` der Klasse `MontiWis` und anschließendem Aufruf der dortigen `main`-Methode mit einer Reihe von MontiCore-Parametern (angedeutet durch `mcParams1`) gestartet. Darüber wird sichergestellt, dass im ersten Durchlauf lediglich die Teile des Ablaufs ausgeführt werden, die zum Symboltabellenaufbau benötigt werden, nicht jedoch diejenigen Schritte, die den Transformations- oder Generierungsprozess anstoßen. In der `MontiWis`-Klasse werden Klassen der Dependency Injection initialisiert.[8] Über diesen Mechanismus werden aus der `MontiWis`-Klasse die Parameter (`mcParams1`), die ihr bereits übergeben wurden, an die `WebDSLTool`-Klasse weitergereicht. `WebDSLTool` ist eine Klasse, die von `DSLTool` und damit der zentralen Klasse zum Ausführen und Steuern von MontiCore-Abläufen abgeleitet wurde. Die übergebenen Parameter legen fest, welche Teile der im `DSLTool` registrierten Komponenten ausgeführt werden, was im ersten Durchlauf diejenigen MontiCore-Workflows sind, die für den Aufbau der Symboltabelle benötigt werden. Etwas detaillierter behandelt Abschnitt 4.2 die Definition und Verarbeitung von Sprachen mit MontiCore und Abschnitt 4.3 beschreibt, wie die Symboltabelle aufgebaut und zur Prüfung von Kontextbedingungen verwendet wird.

Der zweite Durchgang wird ebenfalls aus der `Generate`-Klasse heraus gestartet, wobei jedoch andere Parameter (angedeutet durch `mcParams2`) zum Einsatz kommen. Auch diese Parameter werden wieder über die Klasse `MontiWis` weitergereicht und im `WebDSLTool` zur Steuerung der Ausführung von MontiCore-Komponenten verwendet. Die Parameter bewirken in diesem Fall, dass sämtliche Schritte ausgeführt werden, die zur Prüfung der Kontextbedingungen notwendig sind. Hierfür wird auf die Informationen der im ersten Durchlauf für alle Modelle aufgebauten Symboltabelle zurückgegriffen, die in geeigneter Form abgelegt wurden und im zweiten Durchlauf geladen werden können. Potenzielle Fehler in den Modellen werden als Warnung ausgegeben. In einem nächsten Schritt wird vom `WebDSLTool` aus über die verschiedenen `Transformator`-Klassen[9] aus den AST-Objekten die Zwischenstruktur aufgebaut (angedeutet durch den Aufruf von `transform`). Etwas genauer wird auf diesen Ablauf in Abschnitt 4.4 eingegangen. Im nächsten Schritt werden die verschiedenen `Reconstructor`-Klassen ausgeführt, die Umbauten in der Zwischenstruktur vornehmen bzw. Standardverhalten darin hinzu generieren (repräsentiert durch den Aufruf `reconstruct`). Da es sich hierbei nicht um Workflows von MontiCore handelt, werden diese aus der `MontiWis`-Klasse heraus gestartet und nicht, wie bislang, aus dem `WebDSLTool` heraus. Ein kurzer Überblick darüber ist in Abschnitt 4.5 zu finden. Die Generierung wird ebenfalls aus der `MontiWis`-Klasse gestartet. Hierbei werden die verschiedenen `Generator`-Klassen aufgerufen, die aus den bislang angelegten Zwischenstrukturobjekten durch den Einsatz verschiedener Templates die benötigten Ausgabeartefakte erzeugen (dargestellt durch den Aufruf von

[8] Im Gegensatz zur MontiCore-Infrastruktur wird in MontiWIS das Dependency Injection Framework Google Guice [Gui] verwendet, über das beispielsweise die Rekonstruktorklassen oder aber auch die zentralen Speicherstellen für die Zwischenstruktur bereitgestellt und während des Generationsprozesses zugänglich gemacht werden.

[9] Unter `Transformators`, `Reconstructors` und `Generators` sind in diesem Fall die Komponenten zu verstehen, die für jede Modellart existieren und jeweils die verschiedenen Modelle bzw. eine Kombination aus diesen verarbeiten. Aufgrund der überblicksartigen Darstellung wird von einzelnen Klassen an dieser Stelle abstrahiert.

generate). Eine weitergehende Erläuterung dieses Vorgangs findet in Abschnitt 4.6 statt.

Die oben beschriebenen Verhaltensweisen und Klassen sind zusammen in dem Eclipse-Projekt MontiWis implementiert, wohingegen die in Abbildung 4.1 erwähnten *Laufzeitbibliotheken* im MontiWis-Target-Projekt umgesetzt wurden. Innerhalb des MontiWis-Projekts existieren einige Sprachpakete bzw. -ordner, innerhalb derer die Funktionalität für eine bestimmte Modellart implementiert ist. Eine Übersicht über die Verzeichnisse sowie eine kurze Erläuterung ist Tabelle 4.3 zu entnehmen.

Paket	Erläuterung
cd	Enthält Code für die Klassendiagrammverarbeitung
page	Enthält Code für die Verarbeitung von Seiten
ad	Enthält Code für die Verarbeitung von Aktivitätsdiagrammen
app	Enthält Code für die Verarbeitung der Applikationssprache
combined	Enthält Code für die Verarbeitung von Kombinationen der oben genannten Sprachen

Tabelle 4.3.: Sprachpakete im MontiWIS-Projekt und zugehörige Modelle

Wie dieser Tabelle zu entnehmen ist, existiert für jede Sprache ein eigenes Paket inklusive diverser Unterpakete, welche allesamt unterhalb des Pakets de.montiwis zu finden sind. Die in den nachfolgenden Abschnitten beschriebenen Mechanismen existieren in der Regel in all diesen Verzeichnissen bzw. Paketen, es sei denn, es wird explizit darauf hingewiesen.

4.2. Sprachverarbeitung mit MontiCore

Wie bereits in Abschnitt 3.3 kurz erläutert, lassen sich über das MontiCore-Framework aus Grammatiken sowohl Lexer als auch Parser für Sprachen erzeugen. Im MontiWis-Projekt werden die Sprachdefinitionen selbst, wie in MontiCore üblich, im Verzeichnis def abgelegt. Darin finden sich in Unterverzeichnissen sowohl die Sprachdefinitionen selbst (in MontiCore-Grammatiken mit dem Suffix mc) sowie die zugehörigen Sprachdateien (mit dem Suffix lng), worin die Metainformationen zur Bearbeitung der Sprachen enthalten sind. Die Sprachverarbeitung findet über die Ausführung der Ant-Datei [Ant] build.xml statt. Hierbei werden durch MontiCore im gen-Verzeichnis des gleichen Projekts eine Reihe von Dateien erzeugt. Dazu zählen beispielsweise AST-Klassen, die die abstrakte Syntax der Sprache repräsentieren und Methoden zur Navigation innerhalb dieser Struktur bieten. Weiterhin wird Parserinfrastruktur erzeugt, die zum Einlesen und zur Weiterverarbeitung von Modellen der jeweiligen Sprache dient.

Die Verarbeitung der Modelle erfolgt im weiteren Verlauf über das sogenannte DSLTool-Framework, in dem Modelle als DSLRoot-Objekte repräsentiert werden, die ebenfalls durch MontiCore generiert werden. Die einzelnen Bearbeitungsschritte werden in sogenannten

`Workflows` organisiert, die durch das `DSLTool` gesteuert und ausgeführt werden. Die soeben beschriebenen Artefakte werden für jede Sprache separat durch MontiCore erzeugt. Wie im vorherigen Abschnitt 4.1 bereits erläutert, werden die zugehörigen Grammatiken und Klassen in den jeweiligen Unterpaketen bzw. -verzeichnissen abgelegt. Für die Bearbeitung der Sprachen ist es im `DSLTool`-Framework notwendig, dass für jede Sprache eine `Language`-Klasse existiert, die die abstrakte Klasse `ModelingLanguage` erweitert. Darin sind sämtliche Komponenten zur Sprachverarbeitung mit MontiCore registriert, was beispielsweise Klassen zum Parsen, PrettyPrinten sowie Klassen für den Aufbau der Symboltabelle und die Transformatorklassen zur Überführung von AST-Objekten in die Zwischenstruktur umfasst. Diese Klassen existieren für jede Modellart, sind im Wurzelpaket der Sprachpakete zu finden und werden in der Klasse `de.montiwis.dsl.DefaultWebDSLTool` registriert.[10] Eine detaillierte Darstellung sowohl von Sprachdefinition als auch von Modellverarbeitung inklusive der dazu getroffenen Annahmen ist in [Kra10] zu finden. Die Sprachdateien wurden im Rahmen von [Völ11] entwickelt und verfeinert und werden dort detaillierter beschrieben.

4.3. Verwendung von Symboltabelle und Kontextbedingungen

Um den Anforderungen aus Abschnitt 3.4.2 zu genügen, wird im Rahmen von MontiWIS auf die Symboltabelleninfrastruktur aus [Völ11] gesetzt. Eine Kurzbeschreibung der dort entwickelten Mechanismen wurde bereits in Abschnitt 3.3.2 gegeben.

Die Implementierung der einzelnen Klassen findet je Sprachpaket im Paket `ets` statt. Dort befinden sich für jede Sprache diejenigen Klassen, die Funktionalität einzig für diese Sprache enthalten. Eine Ausnahme bildet hier das Sprachpaket `cd`, da hier auf die Funktionalität zurückgegriffen wird, die im Rahmen von [Sch12] entwickelt wurde. Die Kombination von Sprachen inklusive der Implementierung von Adaptern zwischen den verschiedenen Sprachen und den zugehörigen Intermodellkontextbedingungen ist im Paket `combined` zu finden.

Wie in Abschnitt 4.1.2 erläutert und in Abbildung 4.2 dargestellt, werden in MontiWIS die Symboltabellen und die dazugehörigen Einträge (implementiert im Unterpaket `entries`) in einem ersten Schritt aufgebaut und zur späteren Verwendung im Dateisystem abgelegt. Dies geschieht über die Visitoren, die je Sprachpaket im Paket `visitors` definiert sind. In einem weiteren Schritt werden die Einträge qualifiziert (über Klassen im Paket `qualifiers`), d. h., dass die Einträge aufgrund ihres Namens aufgelöst werden (über Klassen im Paket `resolvers`) und mit den referenzierten Einträgen über Adapter (implementiert im Paket `adapters`) in Beziehung gesetzt werden. Die entsprechende Umsetzung findet bei Intramodellreferenzen (d. h. bei Referenzen innerhalb eines Modells der gleichen Sprache) im jeweiligen Sprachpaket, bei Intermodellreferenzen (d. h. bei Referenzen zwischen verschiedenen Modellen unterschiedlicher Sprachen) im `combined`-Paket statt. Schließlich erfolgt die Überprüfung von Kontextbedingungen, die ebenfalls entsprechend ihres Bezugs in den Sprachpaketen abgelegt sind. Die ausführbaren `CheckWorkflowClients` sowie einige wenige Hilfsklassen befinden sich im Paket `check` und die Klassen, die die abstrakte Klasse `ContextCondition`

[10] Hierbei handelt es sich um die Implementierung der Klasse, die als `WebDSLTool` in Abschnitt 4.1 bzw. in Abbildung 4.2 bezeichnet ist.

erweitern und damit die Kontextbedingungen selbst implementieren, sind im Paket `coco` der Sprachpakete zu finden.

Dieser sehr oberflächliche Überblick über die Symboltabelleninfrastruktur soll an dieser Stelle genügen, um zusammen mit Abschnitt 3.3 ein grobes Verständnis für die Funktionsweise zu erhalten. Weitere Details sind [Völ11] zu entnehmen. Für Erläuterungen der einzelnen Kontextbedingungen je Sprache sei auf die Abschnitte 6.3.1, 7.3.1, 8.4.1 sowie 9.3.1 verwiesen.

4.4. Verwendung der Zwischenstruktur

In Abschnitt 4.1 wurde bereits darauf eingegangen, dass in MontiWIS eine Klassenstruktur verwendet wird, die zwar relativ ähnlich zur AST- und damit zur Modellstruktur aufgebaut ist, aber wesentliche Vorteile gegenüber der direkten Verwendung von AST-Klassen besitzt. So lassen sich in den Klassen der Zwischenstruktur einfacher Methoden integrieren, die einen komfortablen Zugriff auf die darin enthaltenen Informationen ermöglichen. Derartige Methoden ließen sich zwar ebenfalls in den Grammatiken hinzufügen, jedoch würden die erzeugten Klassen eher unübersichtlich, weswegen die Verwendung separater Klassen bevorzugt wird. Ferner werden durch die Rekonstruktoren in MontiWIS eine nicht unerhebliche Anzahl von Artefakten erzeugt, um beispielsweise Seiten für Standardverhalten in der generierten Anwendung nutzen zu können. Derartige Artefakte ließen sich zwar ebenfalls auf AST- Ebene erzeugen, was jedoch aufgrund des sehr unterschiedlichen Anwendungsfalls (nämlich der Abbildung der Modelle und nicht die Repräsentation der Artefakte im generierten System) recht aufwendig wäre und eben diese Trennung der Anwendungsfälle aufbrechen würde. Grundlegend ist festzuhalten, dass durch die Einführung einer Zwischenstruktur eine Entkopplung zwischen den Eingabemodellen und der zu generierenden Applikation hergestellt wird, die zu mehr Flexibilität und einer saubereren Trennung von Artefakten nach ihren Aufgaben während des Generationsprozesses führt.

Wie in Abbildung 4.1 zu sehen ist, erfolgt die Erstellung der Zwischenstruktur über den sogenannten *Transformator*, der die AST-Struktur in Objekte der Zwischenstruktur überführt. Die Klassen, die für diese Transformation zuständig sind, werden ebenfalls in den Sprachpaketen entsprechend der Sprache abgelegt, wo sie im Unterpaket `intermediate` zu finden sind. Ebenfalls an dieser Stelle sind im Paket `elements` jeweils die Klassen enthalten, die die Zwischenstruktur repräsentieren. Daneben existiert ein Paket `transformator`, in dem die Klassen zur Ausführung der Transformatoren sowie die Transformatoren selbst gespeichert sind. Wie Abbildung 4.2 zeigt, werden die Transformatoren direkt aus dem `WebDSLTool` heraus gestartet, so dass dazu auch Klassen zum Einsatz kommen, die sich von `DSLWorkflow`, einer Klasse des MontiCore-Frameworks, ableiten. Die Transformation selbst geschieht über Klassen, die von `ConcreteVisitor` ihre Funktionalität erben und dem Visitor-Muster [GHJV95] folgen. Ferner befinden sich im Paket `intermediate` sogenannte `GlobalStore`-Klassen, die zur Speicherung der erzeugten Zwischenstruktur dienen.

Obwohl das `DefaultWebDSLTool`, welches in MontiWIS die Funktion des `DSLTools` übernimmt, im Kontext von MontiCore läuft und auch dessen Infrastruktur nutzt, sind für die Verwendung der Zwischenstruktur wenige Besonderheiten zu beachten gewesen. Da es für die

Speicherung notwendig war, möglichst kontrolliert und flexibel auf zentrale `GlobalStore`-Klassen zuzugreifen, die wiederum von den zentralen Klassen bei der Bearbeitung (wie beispielsweise besagten Workflows und Visitoren) aus erreichbar sein müssen, kommt zu diesem Zweck Google Guice [Gui] zum Einsatz. Für die korrekte Funktionsweise dieses Dependency Injection Frameworks müssen die Abhängigkeiten der Klasse in korrekter Art und Weise und vollständig initialisiert zur Verfügung gestellt werden, was beispielsweise per Konstruktorparameter geschehen kann.

Für detailliertere Beschreibungen der Bearbeitung einzelner Modelle sei an dieser Stelle auf die folgenden Abschnitte 6.3.3, 7.3.3, 8.4.3 sowie 9.3.3 verwiesen.

4.5. Rekonstruktion von Zwischenstrukturelementen

Nach dem Aufbau der Zwischenstruktur aus der AST-Struktur, die als Eingabeinformationen für den Generator fungiert, erfolgen Schritte zur Nachbearbeitung einzelner Zwischenstrukturelemente sowie zur Erzeugung von Standardverhalten im generierten Web-Informationssystem. Diese Umbauten werden in MontiWIS *Rekonstruktion* genannt und laufen, wie Abbildung 4.2 zeigt, im zweiten Durchlauf des Generierungsprozesses ab.

Die für die Rekonstruktion zuständigen Klassen finden sich im Sprachpaket `combined`, in dem ein Unterpaket `reconstructors` existiert, das schließlich sämtliche notwendigen Klassen enthält. Die Funktionalität ist nochmals auf einzelne Klassen verteilt. Diese Aufteilung geschieht sowohl hinsichtlich betroffener Modellarten als auch nach der Art der durchgeführten Umbauten an der Zwischenstruktur.

Da die Rekonstruktion auf der Zwischenstruktur stattfindet und nicht mehr mit den Klassen zusammenhängt, die aus MontiCore erzeugt wurden, findet die Ausführung auch nicht mehr innerhalb der MontiCore-Infrastruktur statt, sondern wird über die separate Klasse `Reconstructor` gesteuert, worin die übrigen Rekonstruktorklassen registriert sind und in korrekter Reihenfolge ausgeführt werden. Deren Aufruf erfolgt aus der `MontiWis`-Klasse heraus und nicht über die `WebDSLTool`-Klasse.

Eine nähere Beschreibung der Rekonstruktionsschritte findet sich in den Abschnitten 6.3.4, 7.3.4, 8.4.4 sowie 9.3.4.

4.6. Generierung des Web-Informationssystems

Aus der Zwischenstruktur, die im vorherigen Schritt um Standardverhalten angereichert wurde und worin einige weitere Umbauten erfolgten, wird im letzten Schritt das Web-Informationssystem mit sämtlichen benötigten Dateien generiert. Wie Abbildung 4.2 zeigt, ist auch die Ausführung des Generators Teil des zweiten Verarbeitungsdurchlaufs.

In MontiWIS werden Freemarker Templates [Fre] verwendet, um die Generate zu erzeugen. Dies ist die gleiche Technologie, die auch im UML/P-Framework aus der Dissertation von Martin Schindler [Sch12] zum Einsatz kommt. Im Gegensatz zu diesem Ansatz werden jedoch nicht die dort entwickelten Templates verwendet, die ebenfalls zu einem Großteil den Ablauf der Generierung steuern. Einerseits war zum Zeitpunkt der Entwicklung von MontiWIS das

dort entwickelte Framework noch nicht in einem Zustand, der den problemlosen Einsatz ermöglicht hätte. Andererseits wird das dort verwendete Verfahren der Kombination und Steuerung durch partiell sehr kleinteilige Templates als nicht vorteilhaft erachtet, da die Werkzeugunterstützung bei der Erstellung der Templates nur sehr rudimentär vorhanden ist und etwaige Fehler bei der Generierung erst zur Ausführungszeit der Templates auftreten. Daher wurde im Rahmen von MontiWIS für die Steuerung des Generierungsprozesses ein größtmöglicher Teil der Logik vorrangig in Java-Klassen verlagert, da dort eine starke Typisierung und umfangreiche Werkzeugunterstützung zur Verfügung steht. Da im Fall von MontiWIS die Zieldomäne festgelegt ist (nämlich die Generierung eines Web-Informationssystems) und damit die Anpassbarkeit des Codegenerators auf unterschiedliche Domänen und Einsatzbereiche nicht als vorrangig erachtet wird, ist die Flexibilität und feine Granularität, die der UML/P-Ansatz bietet, nicht in dem Umfang erforderlich und wurde dementsprechend auch nicht als Ziel bei der Entwicklung von Templates und Generator allgemein verfolgt.

Die Generierung erfolgt im Wesentlichen auch aufgeteilt nach Sprachpaketen. Dies hat zur Folge, dass im Sprachpaket `app` keine Dateien existieren. Die Daten, die originär aus der Applikationssprache stammen, werden stattdessen im Paket `combined` verarbeitet, da für deren Bearbeitung Informationen aus einer Reihe weiterer Sprachpakete vorliegen müssen. Im Paket `generator` eines jeden Sprachpakets befindet sich eine `Generator`-Klasse, die die Steuerung des Generierungsprozesses übernimmt. Die Templates, die bei der Erzeugung des Zielsystemcodes zum Einsatz kommen, sind in diesem Paket in einem Unterverzeichnis `templates` abgelegt. Darin existieren potenziell weitere Verzeichnisse, die die Templates hinsichtlich der Art des Artefakts ordnen, zu dessen Erzeugung sie beitragen. So existiert beispielsweise für Java-Klassen das Verzeichnis `java`, für JSP-Seiten das Verzeichnis `jsp` und für die Ablage von Templates für Konfigurationsdateien das Verzeichnis `config`. Diese Aufteilung hat keine funktionale Bedeutung, sondern wurde der Übersichtlichkeit halber eingeführt.

Eine genauere Beschreibung, welche Funktionalität in welchen Templates enthalten ist und wie diese verarbeitet werden, ist in den Abschnitten 6.3.6, 7.3.5, 8.4.5 und 9.3.5 zu finden.

Kapitel 5.

Generiertes System

Nach der Beschreibung des MontiWIS-Projekts und des Ablaufs der Modellverarbeitung und Anwendungsgenerierung im vorherigen Kapitel 4, wird in diesem Kapitel auf die Architektur und die einzelnen Komponenten des generierten Web-Informationssystems eingegangen, so dass die Zusammenhänge der verschiedenen Komponenten deutlich werden und in späteren Kapiteln entsprechend eingeordnet werden können. Hierbei werden ebenfalls potenziell alternative Implementierungen diskutiert und die Entscheidung für eine bestimmte Technologie oder Architektur erläutert.

5.1. Überblick über das generierte Web-Informationssystem

Einen Überblick über die Merkmale und Qualitäten eines Web-Informationssystems gab bereits Kapitel 2. Darin wurden die Anforderungen an die Zieldomäne sowie verschiedene populäre Ansätze zur Implementierung eines solchen Systems diskutiert.

In MontiWIS wurde als Programmiersprache für die Zielimplementierung Java gewählt. Die Entscheidung dazu wurde insbesondere durch die bislang gesammelten Erfahrungen mit dieser Sprache und den zugehörigen Frameworks sowie ihre weite Verbreitung [PYP] beeinflusst. Die Wahl der Serverinfrastruktur fiel in MontiWIS auf die Java EE [Sun09a] im Allgemeinen und auf den Glassfish Server [Gla, Hef10] im Speziellen. Ausschlaggebend waren hier bisherige Erfahrungen mit dieser Technologie und die Tatsache, dass dieser Server als Referenzimplementierung der Java EE gilt.

Die Unterteilung des Kapitels erfolgt hinsichtlich der Komponenten, die zum generierten System gehören bzw. von diesem verwendet werden. Ein grober Überblick über die Architektur ist in Abbildung 5.1 dargestellt.

Die `Entities` sind die zentralen Elemente im generierten System. Darunter sind die Datenklassen zu verstehen, die die Struktur der zu verwaltenden Daten widerspiegeln. Klassen dieser Komponente werden von quasi allen anderen Komponenten verwendet, wie ebenfalls in dieser Abbildung ersichtlich ist. Eng mit den `Entities` hängen die `DAOs` zusammen, die als Schnittstelle zwischen Applikation und Persistenzschicht, d. h. den gespeicherten Daten, dienen. Auch diese werden von einer Vielzahl anderer Komponenten verwendet. Nähere Informationen zu Datenstruktur und Persistenzschicht enthält Abschnitt 5.2, wobei Abschnitt 5.2.1 die Struktur der Datenklassen detaillierter beschreibt und Abschnitt 5.2.2 auf die DAOs eingeht.

Die Schnittstelle zum Nutzer des Web-Informationssystems und dazugehörige Funktiona-

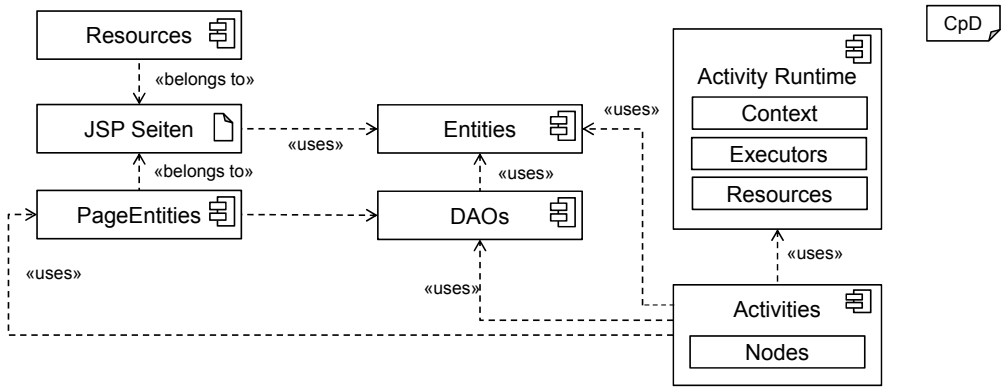


Abbildung 5.1.: Komponenten des generierten Systems

lität ist auf der linken Seite dargestellt. Die Nutzerschnittstelle wird über die `JSP Seiten` realisiert, wozu weiterhin `Resources` und `PageEntities` gehören. Diese Komponenten und Artefakte werden in Abschnitt 5.3 erläutert. In Abschnitt 5.3.1 werden zunächst allgemeine Prinzipien bei der Interaktion mit dem Nutzer beschrieben. Abschnitt 5.3.2 geht näher auf die JSP-Seiten ein. Anschließend erläutert Abschnitt 5.3.3, welche Funktionalität in den `PageEntities` gekapselt ist und Abschnitt 5.3.4 beschreibt, welche Rolle die Ressourcenklassen in MontiWIS spielen.

Schließlich existieren im generierten System noch `Activities` mit all den zugehörigen `Nodes`, die die Abläufe im Web-Informationssystem repräsentieren. Diese nutzen die `Activity Runtime`, welche ihrerseits `Context`-Klassen zur Speicherung von Zuständen, `Executors` zur Ausführung einzelner Knoten sowie `Resources` für die Interaktion mit Nutzern enthält. Eine nähere Beschreibung dieser Komponenten erfolgt in Abschnitt 5.4. Nach einer allgemeinen Einführung wird in Abschnitt 5.4.1 auf die Funktion der generierten Knotenstruktur eingegangen. In Abschnitt 5.4.2 wird das Prinzip der Kontextklassen erläutert, die zusammen mit den Knoten von den Executorklassen zur Ausführung benötigt werden. Letztere werden in Abschnitt 5.4.3 erläutert. Die Funktionalität der Ressourcenklassen, die zur Steuerung von Abläufen durch Interaktion mit Browser und Nutzer dienen, wird in Abschnitt 5.4.4 beschrieben.

Darauffolgend wird in Abschnitt 5.5.1 beschrieben, wie die ausführbaren und anzeigbaren Komponenten in einem Menü verwendet werden. Weitere Aspekte wie die Implementierung von Nutzerverwaltung und -authentifizierung werden schließlich in Abschnitt 5.5.2 erläutert.

Die in den folgenden Abschnitten beschriebenen Komponenten sind in jedem Projekt zu finden, in dem der MontiWIS-Generator zur Generierung der Anwendung verwendet wird bzw. wurde. Ein Überblick über den generellen Aufbau eines MontiWIS-Anwendungsprojekts ist in Abschnitt 10.1.2 zu finden. Die beschriebenen Dateien sind in der Regel im `gen`-Verzeichnis des Projekts zu finden, sofern nicht gesondert darauf hingewiesen wird.

5.2. Datenstruktur und Persistenz

Als zentrales Element eines Web-Informationssystems lässt sich die Domänendatenstruktur sowie die Mechanismen der Persistierung zugehöriger Objekte identifizieren. Wie bereits in den Abschnitten 2.3.1 und 2.3.2 diskutiert, existieren verschiedene Ansätze, wie diese Mechanismen in einem Web-Informationssystem umsetzbar sind. Da in MontiWIS die Entscheidung auf die Java EE als Zielspezifikation und auf Glassfish als Serverinfrastruktur fiel, bot sich die Verwendung von JPA [Sun09b, KS09] als Framework zur Implementierung an. JPA ist laut Java EE 6 [Sun09a] die Standardimplementierung für die objektrelationale Abbildung im Kontext der Java EE und wird damit ausgezeichnet unterstützt.

5.2.1. Datenklassen

Die Klassen, die letztlich das Domänendatenmodell bilden, werden in das Verzeichnis `gen` im Paket `de.montiwis.entities` abgelegt. Klassen, die von JPA direkt verwaltet werden können, müssen mit der Annotation `@Entity` versehen sein und die `Serializable` Schnittstelle implementieren. Diesen Vorgaben folgend werden auch die dort angelegten Klassen generiert. Jede dieser Klassen besitzt vor der Klassendeklaration eine Menge an `NamedQueries`, über die Abfragen hinsichtlich verschiedener Attribute gestellt werden können. Für Attribute existieren ebenfalls `get`- und `set`-Methoden, über die auf die privaten Klassenvariablen zugegriffen werden kann. Weitere Informationen, wie die Klassen generiert werden, enthält Abschnitt 6.4.1.

5.2.2. Data Access Object Klassen

Für die Persistierung der im vorherigen Abschnitt erläuterten `Entity`-Klassen werden in MontiWIS sogenannte Data Access Object (DAO)-Klassen [ACM03] verwendet. Diese sind im Paket `de.montiwis.dao` enthalten. Für jede Klasse aus dem Domänendatenmodell werden im zugehörigen Verzeichnis zwei Dateien angelegt. Eine Schnittstelle, die mit der Annotation `@Remote` versehen wurde und das Suffix `RemoteDao` zusätzlich zum Klassennamen trägt, dient als Definition der nach außen sichtbaren Schnittstelle und erlaubt aufgrund der Annotation den Zugriff auf ihre Funktionalität von außerhalb der Applikation. Die vollständige Implementierung findet in der Klasse statt, die diese Schnittstelle implementiert und neben dem Klassennamen das Suffix `Dao` besitzt. Die Klasse verfügt über die Annotation `@Stateless`, was sie bei ihrer Ausführung zu einem zustandslosen Objekt macht.[11] Über die DAO-Klassen ist es möglich, auf Objekte der Domänendatenstruktur zuzugreifen. Eine detailliertere Beschreibung der DAO-Klassen findet in Abschnitt 6.4.1 statt.

[11] Die hier verwendeten Annotationen stammen aus der Java EE Spezifikation [Sun09a, Hef10], worin auch ihre konkrete Bedeutung und Funktion festgelegt ist.

5.3. Nutzerinteraktion und clientseitige Technologien

Für die Darstellung der Informationen in einer Java EE Anwendung bieten sich verschiedene Möglichkeiten. Über JSF [Ora13d, Mül10] als prominente Methode zur Webseitenerzeugung lassen sich beispielsweise relativ einfach Seiten erstellen, die ihrerseits auf Komponenten für unterschiedliche Datentypen und Darstellungsanforderungen aufbauen. JSF bietet eigene Mechanismen für die Navigation zwischen Seiten, die zwar in Teilen (wie beispielsweise einer linearen Seitenabfolge) mit dem angestrebten Paradigma der Modelle in MontiWIS übereinstimmen (vgl. Abschnitt 3.4.1), jedoch für andere Konzepte wie etwa die Verteilung von Abläufen über mehrere Nutzer kein direktes Äquivalent bieten. Aus diesem Grund wurde in MontiWIS vom Einsatz von JSF Abstand genommen und stattdessen auf JSP für die Implementierung der Seiten gesetzt. Zusätzlich wurden für die Umsetzung der Nutzeroberfläche und die Kommunikation mit den Kontrollkomponenten des Web-Informationssystem einige weitere Entscheidungen getroffen, die im Folgenden erläutert werden.

In Abschnitt 5.3.1 werden die grundlegenden Prinzipien erläutert, denen die einzelnen Komponenten folgen. Abschnitt 5.3.2 gibt einen kurzen Überblick über die generierten JSP-Seiten und Abschnitt 5.3.3 erläutert die zu den Seiten gehörenden Datenspeicher. Schließlich erläutert Abschnitt 5.3.4, wie von Browser- bzw. Nutzerseite mit den Seiten interagiert wird.

5.3.1. Generelle Funktionsprinzipien

Hinsichtlich der Interaktion zwischen Nutzer und Web-Informationssystem wird in MontiWIS generell versucht, dem REST-Paradigma [FT02, Fie00] zu folgen. Damit werden in der Regel Artefakte eines Web-Informationssystems als *Ressource* betrachtet. Dies sind im Kontext von MontiWIS sowohl die Objekte der Domänendatenstruktur als auch die Aktivitäten und Aktionen, die die Abläufe repräsentieren. Laut des REST-Paradigmas sind Ressourcen adressierbar, lassen sich über HTTP-Anfragen in verschiedenster Art manipulieren und die Interaktionen mit ihnen erfüllen dabei bestimmte Bedingungen.

Die Adresse einer Ressource wird über eine Uniform Resource Locator (URL) abgebildet, die sie eindeutig identifizierbar macht. Die Manipulation einer Ressource geschieht über die HTTP-Methoden `GET`, `PUT`, `DELETE` und `POST`. So werden `PUT` bzw. `POST` Anfragen genutzt, um neue Objekte anzulegen oder bestehende Objekte zu aktualisieren. `GET` Anfragen werden zum Abruf einer bestimmten Ressource verwendet, und `DELETE` schließlich dient zum Löschen des adressierten Objekts.

Der HTTP-Spezifikation [FGM+99] folgend, sollte die Umsetzung der Methoden beim Aufruf der URL einer bestimmten Ressource folgende Bedingungen erfüllen. `GET`-Aufrufe sollten *sicher* (engl. *safe*) sein. Dies bedeutet, dass der Abruf einer Ressource per `GET` keine Seiteneffekte haben sollte, sondern lediglich das adressierte Objekt zurückliefert. Weiterhin sollten Aufrufe der `PUT` und `DELETE` Methoden *idempotent* (engl. ebenso) sein. Solche Aufrufe sollen auch bei mehrmaligem Absenden zum gleichen Ergebnis, d. h. zu einer Ressource im gleichen Zustand, führen und damit wiederholbar sein. Dies ist insofern gewährleistet, als das mehrmalige Aktualisieren einer Ressource per `PUT` stets zu den gleichen Werten führt und das mehrmalige Löschen einer Ressource per `DELETE` auch weiterhin zur Nichtexistenz auf Serverseite zur Folge hat. Tabelle 5.2 fasst diese Bedingungen zusammen. Die Behandlung der

HTTP-Aufrufe findet in MontiWIS in sogenannten *Ressourcenklassen* statt, die in der generierten Applikation im Paket `de.montiwis.resources` zu finden sind. Genauere Erläuterungen dazu finden sich in den Abschnitten 5.3.4 sowie 5.4.4.

HTTP-Methode	sicher	idempotent
`GET`	ja	ja
`PUT`	nein	ja
`DELETE`	nein	ja
`POST`	nein	nein

Tabelle 5.2.: HTTP-Methoden und Bedingungen hinsichtlich Seiteneffekten

Für die Implementierung der `PUT` und `DELETE` Methoden ließen sich zwar Methoden per JavaScript übermitteln, zum Zeitpunkt der Umsetzung in MontiWIS (und zum Teil auch heute noch) war die Unterstützung dieser Methoden in Browsern jedoch nicht sichergestellt. Daher wurde entschieden, die entsprechenden Aufrufe per `POST` zu senden und das Attribut `_method="PUT"` bzw. `_method="DELETE"` zu betreffenden Anfragen hinzuzufügen. Auf Serverseite bewirkt die Klasse `de.montiwis.filter.PostReconstructFilter` aus dem `MontiWIS-Target`-Projekt, dass die so gesendeten Anfragen in tatsächliche `PUT` bzw. `DELETE` Anfragen umgeschrieben und von den Ressourcenklassen als originäre Anfragen behandelt werden.

5.3.2. JSP Seiten

Die Darstellung der Seiten im generierten Web-Informationssystem geschieht in MontiWIS über JavaServer Pages (JSP). Der Generator erzeugt die zugehörigen JSP-Dateien im Verzeichnis `webapp/WEB-INF/jsp`. Darin befinden sich sowohl die Seiten, die aus den explizit modellierten Seitenmodellen erstellt wurden, als auch die Seiten zur Umsetzung der Standardfunktionalität für jede Klasse der Domänendatenstruktur.

Innerhalb der JSP-Seiten ist der Teil, der das Menu der Applikation definiert, in eine Datei `menu.jsp` ausgelagert, die in jede der generierten Seiten eingebunden wird. Da die JSP-Technologie nicht von Haus aus die Möglichkeit bietet, dynamische und nutzerfreundliche Elemente (vergleichbar etwa mit Komponenten in JSF) in der Seite zu erzeugen, wird für derartige Funktionalität die jQuery JavaScript Bibliothek [jQua, Ste14] sowie einige Erweiterungen dazu verwendet. Beispiele für verwendete Plugins sind etwa eines für eine erweiterte Eingabevalidierungsfunktionalität [jQub] oder eines für die dynamische Tabellendarstellung einer Liste von Objekten [Dat]. Hinsichtlich der Darstellung der Oberflächenelemente auf der Seite wird im Wesentlichen auf jQueryUi [JQuc] gesetzt, über das sich relativ einfach neue Designs über einen Online CSS Generator erstellen und einbinden lassen. Die Wahl auf diese Plugins und Bibliotheken fiel insbesondere aufgrund des hohen Verbreitungsgrades und der weitgehenden Kompatibilität mit sämtlichen gängigen Browsern. Die Script-

und Style-Dateien sowie zugehörige Bilder sind in den Verzeichnissen `webapp/script` bzw. `webapp/css` und `webapp/images` zu finden. Neben der Funktionalität der genannten Bibliotheken existieren weitere JavaScript- und CSS-Dateien mit MontiWIS-eigenen Ergänzungen, die ebenfalls in den genannten Verzeichnissen abgelegt werden. Änderungen daran sollten, sofern notwendig, jedoch nicht in diesen Verzeichnissen direkt, sondern in ihren Äquivalenten im `res`-Verzeichnis stattfinden, da die Inhalte im `webapp`-Verzeichnis in der Regel von denen aus dem `res`-Verzeichnis überschrieben werden, sobald die Applikation neu generiert und zum Server hochgespielt wird.

An verschiedenen Stellen innerhalb der Applikation existieren weiterhin einige Elemente, deren Beschriftung bzw. Bezeichnung in der generierten Anwendung änderbar sein sollten, ohne dass das komplette System erneut erzeugt werden muss. Dazu existieren im Verzeichnis `gen/resource_bundles` zwei Dateien. In der Datei `langstring.properties` stehen Schlüssel-Wert-Paare, deren Schlüssel in den generierten Seiten anstatt der konkreten Knopf- oder Attributbezeichnungen verwendet werden und deren zugehörigen Werte an den entsprechenden Stellen ersetzt und angezeigt werden. Die andere dort vorliegende Datei `langstring.properties_will_be_overwritten` wird mit jedem Generatorlauf neu erzeugt und mit sämtlichen vorhandenen Schlüssel-Wert-Paaren gefüllt. Änderungen sollten dementsprechend in der `langstring.properties`-Datei erfolgen, da diese nach einmaliger Erzeugung nicht automatisch überschrieben wird. Weitere Details zur Implementierung der Seiten finden sich im Abschnitt 7.4.

5.3.3. PageEntities

Für jede Seite, auf der Daten dargestellt werden, existieren zwei Klassen, die die darzustellenden Daten auf Serverseite verarbeiten und in der Seite bereitstellen. Diese Klassen sind im Verzeichnis `de.montiwis.pageentities` zu finden. Eine `PageEntity`-Klasse setzt sich aus dem Namen der Seite und dem Suffix `PageEntity` zusammen. Sie enthält in jedem Fall sämtliche Attribute, die auf der zugehörigen Seite dargestellt bzw. modifiziert werden. Ferner erben diese Klassen von `de.montiwis.pageentities.DefaultPageEntity` und besitzen damit einige weitere Attribute, die für die Funktionsweise der Applikation vorhanden sein müssen. Ein Attribut `name` einer `PageEntity`-Klasse lässt sich per Konvention aus der Seite heraus per `${it.name}` aufrufen, weitere Attribute analog. Die Namenskonformität wird per Generator sichergestellt.

Die Initialisierung der `PageEntity`-Klasse wird durch eine zweite generierte Klasse übernommen, bei der das Suffix `PageEntityInitializer` an den Seitennamen angehängt. Darin werden die Objekte aus potenziellen involvierten Abläufen in die `PageEntity` übergeben bzw. die Werte, die per HTTP an den Server gesandt wurden, wieder in der Applikation verfügbar gemacht. Ferner findet die serverseitige Validierung hinsichtlich korrekter Datentypen in diesen Klassen statt.

5.3.4. Ressourcenklassen

Wie kürzlich in Abschnitt 5.3.1 beschrieben, werden viele der Elemente im generierten MontiWIS System als Ressourcen betrachtet. Dies umfasst sowohl die Objekte der Domänen-

datenstruktur als auch die Abläufe in der Applikation, die als Aktivitätsdiagramm modelliert werden. Für die Interaktion existieren sogenannte *Ressourcenklassen*, deren Funktionsweise nachfolgend erläutert wird.

Allgemeine Funktionsweise

Die Ressourcenklassen werden in das Paket `de.montiwis.resources`[12] generiert. Für die Implementierung der Klassen kommt Jersey [Jer] als JAX-RS [Ora13b, Bur10] Implementierung zum Einsatz. In einer Klasse, die dieser Spezifikation folgt, wird per `@Path`-Annotation festgelegt, bei welcher URL die Klasse angesprochen wird. Ferner wird pro Funktion festgelegt, bei welcher HTTP-Methode mit welchen abgesandten Parametern sie aufgerufen wird. Funktionen mit der `@GET`-Annotation beispielsweise werden bei Übereinstimmung von URL und HTTP-Methode aufgerufen und abgearbeitet. Die Rückgaben einer so adressierten Methode repräsentieren in der Regel das Ergebnis, das an den aufrufenden Client bzw. Browser zurückgegeben wird. In MontiWIS ist dies üblicherweise ein `Viewable`-Objekt, das eine JSP-Seite und das zugehörige `PageEntity`-Objekt enthält.

Ressourcenklassen für Seiten

Ressourcenklassen existieren unter anderem für jede Seite, die im System vorhanden ist. Dadurch kann prinzipiell jede Seite direkt angesprochen und aufgerufen werden, auch wenn dies lediglich bei den wenigsten Seiten sinnvoll ist. Üblicherweise werden jedoch lediglich Seiten ohne dynamische Anteile direkt aufgerufen, da darüber keine Interaktion mit der Applikation möglich ist. Der Name einer Seitenressource setzt sich zusammen aus dem Namen der Seite und dem Suffix `Resource`. Nähere Informationen zum Aufbau derartiger Klassen finden sich in Abschnitt 7.4.4.

Ressourcenklassen für Domänendatenobjekte

Neben Ressourcenklassen für Seiten werden auch derartige Klassen für jede Domänendatenklasse generiert. Der Name der Ressourcenklasse setzt sich aus dem Namen der Domänendatenklasse und dem Suffix `Resource` zusammen. Dort sind Funktionen spezifiziert, die einen Zugriff auf die Datenobjekte nach dem REST-Paradigma ermöglichen. So lassen sich Objekte anlegen bzw. Seiten zum Anlegen derartiger Objekte abrufen, Objekte verändern oder auch löschen. Wie diese Klassen implementiert sind und welche Funktionalität auf welche Weise im Einzelnen angesprochen werden kann, wird detaillierter in Abschnitt 6.4.2 erläutert.

5.4. Ablaufsteuerung

In einem Web-Informationssystem ist nicht nur die Anzeige einzelner Seiten von Belang, sondern auch die Art und Weise, wie zwischen verschiedenen Seiten navigiert wird und welche

[12] Die Schreibweisen *Ressource* (deutsch) und *resource* (englisch) rühren von der unterschiedlichen länderspezifischen Schreibweisen des Begriffs her und sind beabsichtigt.

Geschäftslogik vor, bei und nach Anzeige der Seiten ausgeführt wird. Für die Steuerung dieser Abläufe wurde in MontiWIS ein eigener Mechanismus geschaffen, der im Wesentlichen die Ausführungssemantik der Aktivitätsdiagramme abbildet.

In den folgenden Abschnitten werden die verschiedenen Komponenten erläutert, die zur Ablaufsteuerung beitragen. Detaillierte Informationen zu all den folgenden Abschnitten finden sich in Abschnitt 8.5.1.

5.4.1. Generierte Knotenstruktur

Die Struktur der Abläufe wird durch Klassen implementiert, die in das Paket `de.montiwis.control` generiert werden. Darin findet nochmals eine Unterteilung hinsichtlich Knotentyp statt, wobei für Aktivitäten ebenfalls ein Verzeichnis vorgesehen ist. All diese Klassen erben von den Klassen aus dem `de.montiwis.workflowengine.elements.impl` Paket des `MontiWIS-Target`-Projekts, die jeweils die Grundfunktionalität der Klassen implementieren. Die Namen der Klassen eines jeden Knotens setzen sich zusammen aus dem Namen der Aktivität, in der sie definiert wurden, gefolgt vom Namen des Knotens innerhalb der Aktivität. Darüber wird Namenseindeutigkeit garantiert.

Im Fall von Aktionen enthalten die generierten Klassen sowohl die Logik, die darin spezifiziert wurde, als auch die Funktionalität zum Setzen und Auslesen von Parametern. Aktivitäten enthalten sowohl Objekte der Knoten als auch die Verschaltung der Knoten, wie sie über Kanten im Modell definiert wurde.

5.4.2. Kontextklassen zur Speicherung des Ablaufzustands

Zu der generierten Knotenstruktur gehören ebenfalls sogenannte `Context`-Klassen, die zur Speicherung des Zustands eines Ablaufs bzw. eines Knotens dienen. Zur Repräsentation des Status eines Ablaufs dient die Klasse `ActivityExecutionContext`, die wiederum einzelne `NodeExecutionContext`-Objekte für jeden Knoten enthält. Diese bilden zusammen den Fortschritt und Zustand der Ausführung in Form von Parameter- und Variablenwerten ab. Ferner werden Informationen über die Nutzer abgelegt, die an der Ausführung beteiligt sind. Diese Objekte werden entweder in einem zentralen Speicher innerhalb der Applikation abgelegt oder aber in einer Datenbank gespeichert. Durch diese Architektur wird die Struktur des Ablaufs von ihrem Zustand getrennt und kann gespeichert und zu einem späteren Zeitpunkt wieder hergestellt werden. Die `Context`-Klassen werden nicht generiert, sondern wurden ebenfalls im `MontiWIS- Target`-Projekt implementiert. Dennoch werden sie hier der Vollständigkeit halber erwähnt, da sie essenziell für das grundlegende Verständnis der Ablaufausführung sind.

5.4.3. Executorklassen

Die Klassen, die Ablauf- und Knotenstruktur und Ausführungskontext zusammenbringen, sind die sogenannten `Executor`-Klassen, die ebenfalls Teil des bereits vorhandenen `MontiWIS-Target`-Projekts sind. Dort sind sie im Paket `de.montiwis.workflowengine.engine.executors` zu finden. Auch hier existieren verschiedene Klassen für unterschiedliche Knotentypen, da

auch deren Ausführungssemantik eine andere ist. Die Klasse `ActivityExecutor` dient zur Ausführung einer Aktivität, wohingegen für jede Knotenart eine eigene `Executor`-Klasse existiert. Generell ist in diesen Klassen die Logik umgesetzt, die die Ausführung eines einzelnen Knotens oder eines ganzen Ablaufs übernimmt. Neben diesen Klassen existiert weiterhin eine Klasse `WorkflowExecutor`, die für das Starten, Stoppen und Fortführen von einzelnen Abläufen zuständig ist und damit den einzelnen Abläufen übergeordnet ist.

Einzelheiten zur Struktur und Funktionsweise dieser Klassen sind in Abschnitt 8.5.1 zu finden.

5.4.4. Ressourcenklassen für Abläufe

Wie in Abschnitt 5.3.1 erläutert, werden neben den Domänendatenklassen und Seiten auch Abläufe als Ressourcen betrachtet. Dementsprechend existieren auch für diese Komponenten Ressourcenklassen, über die einzelne Abläufe oder Teile davon vom Browser aus angesprochen werden. Zum Starten bzw. Fortsetzen eines Ablaufs dienen die `ActivityResource` bzw. `ActivitiesResource`-Klassen. Auf das Ansprechen einer einzelnen Aktion, was sowohl den Abruf einer Aktion als auch das Absenden von Daten an eine bestimmte Aktion betrifft, reagiert die Klasse `ActionResource`. Diese Klassen finden sich ebenso wie die übrigen Ressourcenklassen im Paket `de.montiwis.resources`.

Eine tiefergehende Auseinandersetzung mit Aufbau und Funktionsweise dieser Klassen findet im späteren Abschnitt 8.5.2 statt.

5.5. Applikationssteuerung

Neben Datenstruktur, Seiten und Abläufen zwischen den Seiten werden im generierten Web-Informationssystem weiterhin die Funktionalitäten benötigt, die über die Applikationssprache beschrieben werden. Die Umsetzung dieser Funktionalität wird im folgenden Abschnitt kurz erläutert.

5.5.1. Menüs

Die Ergebnisse, die aus dem Menüteil der Applikationssprache erzeugt werden, finden sich in der Datei `menu.jsp` wieder. Diese enthält sämtliche Logik, um das Menü zum Anzeigen von Seiten und Starten von Abläufen darzustellen. Darin findet sich ebenfalls Funktionalität zur Unterscheidung von Nutzerrollen, so dass lediglich die Menüeinträge angezeigt werden, für die ein Nutzer die Berechtigung besitzt. Dieses Menü wird in sämtliche Seiten eingebunden.

Auch hierzu finden sich im späteren Abschnitt 9.4.1 weitergehende Informationen.

5.5.2. Nutzerverwaltung

Die Nutzerverwaltung im generierten Web-Informationssystem findet sich nicht an einer dezidierten Stelle, sondern über die Applikation verteilt. Dies hängt damit zusammen, dass aufgrund von Sicherheitsüberlegungen in verschiedenen Komponenten wie beispielsweise

Seiten oder Aktivitäten eigenständig überprüft werden sollte, ob ein Zugriff darauf zum gegebenen Zeitpunkt durch einen bestimmten Nutzer erlaubt ist. Dies betrifft sowohl die verschiedenen Ressourcenklassen als auch das generierte Menü.

Genauer wird auf die Funktionsweise in Abschnitt 9.4.2 eingegangen.

Kapitel 6.

Datenstruktur

6.1. Grundlagen und Anforderungen

6.1.1. Grundlagen der Datenmodellierung

Ein zentraler Aspekt bei der Modellierung eines Web-Informationssystems ist die Repräsentation der zugrunde liegenden Domänendatenstruktur. Generell unterscheidet man bei der Modellierung der Datenstruktur drei Ebenen [KE11]. Die *konzeptuelle Ebene* dient der Abbildung des fachlichen Schemas der Anwendung. Hierbei werden die Objekte der Anwendung unabhängig davon spezifiziert, welches Datenbankmanagementsystem letztlich zum Einsatz kommt und welches Paradigma (relational, objektorientiert) diesem zugrunde liegt. Diese Aspekte werden in der *Implementationsebene* näher betrachtet. Noch näher an der Implementierung und auch nicht mehr unabhängig vom verwendeten Datenbankmanagementsystem ist die *physische Ebene*, die letztlich z. B. beschreibt, welche Indizes auf welchen Tabellen definiert werden. In MontiWIS wird bei der Modellierung der Datenstruktur primär die konzeptuelle Ebene betrachtet. Von der Implementationsebene wird sowohl durch den Generator als auch durch die Verwendung eines Frameworks wie JPA abstrahiert, wodurch sämtliche diesbezüglichen Aspekte (wie z. B. Abfragen und Umsetzung des objektorientierten Modells in der Datenbankstruktur) bereits abgebildet werden. Sowohl die Datenbankstruktur als auch potenziell sinnvolle Einstellungen des physischen Entwurfs werden durch das Framework übernommen. Auch hier werden Feineinstellungen, die prinzipiell vorgenommen werden können, durch den MontiWIS-Generator umgesetzt.

Durch das konzeptuelle Modell wird definiert, welche Datenobjekte innerhalb des Systems existieren, welche Attribute und Methoden sie besitzen und wie sie in Beziehung zueinander stehen. Für jedes dieser Objekte stehen im System CRUD-Funktionalitäten zur Verfügung, die primär über Web-Eingabemasken bereitgestellt werden.

6.1.2. Anforderungen an die Datenmodellierungssprache

An die Modelle, über die die Datenstruktur beschrieben wird, existieren einige Anforderungen, die nachfolgend kurz erläutert werden.

Objektorientierte Modelle

Eine Hauptanforderung an die Modelle ist unter dem Thema Objektorientierung zusammenzufassen. Damit ist insbesondere die Möglichkeit gemeint, Vererbung abbilden zu können.

Da dieses Paradigma auch in den Zielsystemen unterstützt wird und zum Einsatz kommt, lassen sich damit flexibel Domänendatenmodelle erstellen, die die gleichen Charakteristika unterstützen.

Assoziationen und Kompositionen

Um zwischen verschiedenen Objekten Beziehungen herstellen zu können, sollten zwei Arten von Relationen unterstützt werden. Assoziationen, die sowohl uni- als auch bidirektional sein können, erlauben einfache Verknüpfungen von Objekten. Hierbei sollten auch Kardinalitäten abbildbar sein können, so dass sich sowohl 1:1, 1:n als auch n:m Beziehungen darstellen lassen. Ferner wären Kompositionen hilfreich, über die enge Beziehungen in einer Art Teil-Ganzes Beziehung abgebildet werden können. So lassen sich Objekte modellieren, deren Existenz lediglich im Kontext anderer Objekte sinnvoll ist, was auch in der Webanwendung entsprechend dargestellt werden sollte.

Abbildung domänenspezifischer Attribute

Die Attribute in den Klassen des Domänendatenmodells sollten einige in Web-Informationssystemen übliche Datentypen unterstützen. Sowohl einfache `Strings`, `Integers` und `Floats`, aber auch `Emails` und `Dates` sollten zur Verfügung stehen. Diese sollten als Attributtyp verwendet werden können und im generierten System zu Überprüfungen hinsichtlich korrektem Format sowie, soweit sinnvoll, Eingabehilfen führen.

6.2. Sprache

Für die Modellierung der Datenstruktur wird ein Profil der Klassendiagramme der Unified Modeling Language / Programmier-geeignet (UML/P) [Rum11, Rum12] verwendet. Die UML/P ist selbst ein Profil der UML und orientiert sich an ihren Kernkonzepten. Sie ist jedoch aufgrund der möglichen Integration von Java-Codestücken und der Anlehnung der Notation an die Java-Syntax für den Einsatz in der programmiernahen Modellierung (daher das Suffix /p) geeignet.

Für die Entscheidung zur Nutzung von Klassendiagrammen als Notation für die Modellierung von Datenstrukturen waren mehrere Gründe ausschlaggebend. Einerseits stellen Klassendiagramme einen etablierten Modellierungsansatz zur Beschreibung von Datenstrukturen dar [KE11]. Insbesondere im Bereich der objektorientierten Modellierung [Mey97] werden Klassendiagramme für den Entwurf und die Spezifikation von Systemen verwendet. Ferner lassen sich die Abhängigkeiten zwischen Datentypen modellieren, die gefordert waren. Sowohl Assoziationen zwischen Klassen als auch Kompositionen und Vererbungsbeziehungen können über die UML/P Klassendiagramme abgebildet werden. Damit stehen dem Modellierer die wichtigsten Beziehungen zwischen Klassen zur Verfügung.

Syntaktisch werden im Kontext von MontiWIS sämtliche Elemente der UML/P Klassendiagramme unterstützt, die im folgenden Abschnitt 6.2.1 erläutert werden. Bei der Generierung

finden jedoch ausschließlich die Konstrukte Beachtung, die in Abschnitt 6.2.2 mit seinen Unterabschnitten beschrieben werden.

6.2.1. Aufbau der Klassendiagrammsprache

Die Grammatik der UML/P Klassendiagramme wurde im Rahmen der Dissertation von Martin Schindler [Sch12] definiert und kommt für MontiWIS in der Version 1.6 zum Einsatz. Eine kompakte Fassung der Grammatik ist in Abbildung 6.1 dargestellt. Die vollständige Grammatik findet sich in Anhang C.5.

MCSimp

```
1  grammar CD extends mc.umlp.common.Common {
2
3    abstract CDElement;
4    abstract CDType extends CDElement;
5    abstract CDCallable;
6
7    external Value;
8    external Body;
9
10   CDDefinition =
11     Completeness? Stereotype? "classdiagram" Name
12     "{" ( CDClass | CDInterface | CDEnum | CDAssociation | Invariant )* "}";
13
14   CDClass astextends CDType =
15     Completeness? Modifier "class" Name TypeParameters?
16     ("extends" superclasses:ReferenceType ("," superclasses:ReferenceType)*)?
17     ("implements" interfaces:ReferenceType ("," interfaces:ReferenceType)*)?
18     ("{" ( CDAttribute | CDConstructor | CDMethod )* "}") |";" );
19
20   CDInterface astextends CDType =
21     Completeness? Modifier "interface" Name TypeParameters?
22     ("extends" interfaces:ReferenceType ("," interfaces:ReferenceType)*)?
23     ("{" ( CDAttribute | CDMethod )* "}") | ";" );
24
25   CDEnum astextends CDType =
26     Completeness? Modifier "enum" Name
27     ("implements" interfaces:ReferenceType ("," interfaces:ReferenceType)*)?
28     ("{"( CDEnumConstant ("," CDEnumConstant)* ";" ( CDAttribute | CDConstructor | ↩
           CDMethod )* )?"}") | ";" );
29
30   CDEnumConstant =
31     Name ( "(" CDEnumParameter ("," cDEnumParameters:CDEnumParameter)* ")" )? ;
32
33   CDEnumParameter =
34      Value ;
35
36   CDMethod astextends CDCallable =
37     Modifier TypeParameters? ReturnType Name "(" ( CDParameter ("," CDParameter)* )? ")"
38     ( "throws" exceptions:QualifiedName ("," exceptions:QualifiedName )* )?
39     Body | ";" ;
```

```
40
41   CDConstructor astextends CDCallable =
42     Modifier TypeParameters? Name "(" ( CDParameter ("," CDParameter)* )? ")"
43     ("throws" exceptions:QualifiedName ("," exceptions:QualifiedName )* )?
44     Body | ";" ;
45
46   CDParameter =
47     Type  Name;
48
49   CDAttribute =
50     Modifier Type Name ("=" Value)? ";" ;
51
52   CDQualifier =
53     Name | Type;
54
55   CDAssociation astextends CDElement =
56     Stereotype? ( ["association"] | ["aggregation"] | ["composition"] )
57     (Derived:["/"])? Name?
58     leftModifier:Modifier leftCardinality:Cardinality?
59     leftReferenceName:QualifiedName("[" leftQualifier:CDQualifier "]")?("(" leftRole:Name↩
           ")")?
60     ( leftToRight:["->"] | rightToLeft:["<-"] | bidirectional:["<->"] | simple:["--"] )
61     ("(" rightRole:Name ")")? ("[" rightQualifier:CDQualifier "]")? rightReferenceName:↩
           QualifiedName
62     rightCardinality:Cardinality? rightModifier:Modifier ";";
63 }
```

Abbildung 6.1.: Vereinfachte Darstellung der Klassendiagramm-Grammatik aus [Sch12]

Wie in Zeile 1 zu sehen, wird die Grammatik der Klassendiagramme von der `mc.umlp.common.Common`-Grammatik (s. Abschnitt C.3) abgeleitet, welche wiederum die Elemente der `Types`-Grammatik (vgl. Abschnitt C.2) und diese ihrerseits die `Literals`-Grammatik (vgl. Abschnitt C.1) erweitert. Darüber stehen einige Nichtterminale zur Verfügung, die in der Klassendiagramm-Grammatik verwendet werden.

Zunächst werden die abstrakten Produktionen `CDElement`, `CDType` und `CDCallable` (Zeilen 3-5) deklariert. `CDType` erbt von `CDElement` und repräsentiert damit sämtliche Elemente eines Klassendiagramms bis auf Methoden (die über das das Nichtterminal `CDMethod` umgesetzt werden) und Konstruktoren (abgebildet über `CDConstructor`). Die letzten beiden Elemente werden über die abstrakte Produktion `CDCallable` repräsentiert. Die beiden Nichtterminale `Value` und `Body` sind als `external` (Zeilen 7-8) definiert, wodurch sie lediglich als Platzhalter innerhalb der Grammatik fungieren und ihr Inhalt per Sprachdatei erst nachträglich festgelegt wird.

Die Nichtterminale, die die eigentlichen Bestandteile eines Klassendiagramms darstellen, werden im übrigen Teil der Grammatik beschrieben.[13] Das Nichtterminal `CDDefinition`

[13] Da MontiWIS nicht sämtliche der dargestellten Elemente auch während der Bearbeitung durch den Visitor berücksichtigt, sei für einen Überblick der tatsächlich unterstützten Elemente auf den nachfolgenden Abschnitt 6.2.2 verwiesen wird.

enthält im Wesentlichen alle weiteren Elemente eines Klassendiagramms. Darin enthalten sind Elemente wie Klassen (über das Nichtterminal `CDClass` abgebildet), Schnittstellen (über `CDInterface`), Aufzählungstypen (durch das Nichtterminal `CDEnum`), sowie Assoziationen (über `CDAssociation`) und Invarianten (über `Invariant`). Klassen (Zeilen 14-18) können sowohl Vererbungs- als auch Implementierungsangaben besitzen (Zeilen 16-17) und sowohl Attribute (über `CDAttribute`), Konstruktoren (über `CDConstructor`) als auch Methoden (über `CDMethod`) beinhalten. Schnittstellen hingegen (Zeilen 20-23) können lediglich Vererbungsbeziehungen besitzen und Attribute und Methoden enthalten. Aufzählungstypen (Zeilen 25-28) können Schnittstellen implementieren und sowohl Aufzählungskonstanten (über das Nichtterminal `CDEnumConstant`, dessen Umsetzung in den Zeilen 30-34 dargestellt ist) als auch Attribute, Konstruktoren und Methoden enthalten. Sowohl Methoden (Zeilen 36-39) als auch Konstruktoren (Zeilen 41-44) besitzen Parameter (über `CDParameter`) und können Ausnahmen werfen. Ihre Implementierung wird stellvertretend durch das externe Nichtterminal `Body` repräsentiert. Parameter (über `CDParameter`), Attribute (über `CDAttribute`) und Qualifizierer (über das Nichtterminal `CDQualifier`) sind in den Zeilen 46-53 definiert. Schließlich werden in den Zeilen 55-62 die verschiedenen Arten von Assoziationen, Kompositionen und Aggregationen definiert. Darin sind etwa Modifier und Kardinalität (Zeilen 58 und 62), Referenz- und Rollennamen (Zeilen 59 und 61) sowie die Verbindungspfeile in jedwede Richtung (Zeile 60) definiert.

Die hier verwendeten Nichtterminale wie beispielsweise `QualifiedName` oder `Name` sind in der `Commons`-Grammatik oder einer der dort bereits vererbten `Types`- oder `Literals`-Grammatiken definiert, die im Anhang aufgeführt sind.

6.2.2. Verwendung der Klassendiagrammsprache

Für die Datenmodellierung kommt in MontiWIS die textuelle Variante der Klassendiagramme zum Einsatz, deren Grammatik in kompakter Form im vorherigen Abschnitt 6.2.1 vorgestellt wurde. Diese Sprache wurde im Kontext der Dissertation von Martin Schindler [Sch12] entwickelt und ist vollständig in Anhang C.5 dargestellt. Im diesem Abschnitt werden nun die konkrete Syntax und die Bedeutung der einzelnen Elemente erläutert. Die Beschreibung erfolgt dabei an einem Beispiel aus der Domäne, die kurz in Abschnitt 2.4 eingeführt wurde.

Aufbau der Klassendiagramme

Der generelle Aufbau eines UML/P Klassendiagramms ist in Abbildung 6.2 dargestellt.

Ein Klassendiagramm sollte wie jedes MontiCore-Modell mit einer Paketdeklaration beginnen (Zeile 1). Auch Klassendiagramme können Import-Statements besitzen, die eine unqualifizierte Nutzung externer Entitäten (hier sind aktuell lediglich Klassen unterstützt) erlauben (Zeile 3). Das vorliegende Beispiel erlaubt die Verwendung aller Klassen, die in Klassendiagrammen des Pakets `tm` definiert sind. Das Klassendiagramm selbst wird mit dem Schlüsselwort `classdiagram` gefolgt vom Namen des Diagramms (Zeile 5) codiert. Innerhalb der geschweiften Klammern werden nun sowohl Klassendefinitionen angegeben als auch Assoziationen und Kompositionen zwischen den Klassen festgelegt.

```
1  package tm;                                                    CD
2
3  import tm.*;
4
5  classdiagram ThesisManagement {
6
7    // Klassendefinitionen
8
9    // Assoziationen und Kompositionen
10
11 }
```

Abbildung 6.2.: Grundaufbau eines UML/P Klassendiagramms

Klassen

Die zentralen Betrachtungsgegenstände dieser Diagrammart sind selbstverständlich Klassen. Der Ausschnitt des Beispiels, der die Datenstruktur der handelnden Akteure beschreibt, ist in Abbildung 6.3 dargestellt. Die zwei Klassen `Lecturer` und `Student` werden hierbei von der Klasse `Person` abgeleitet.

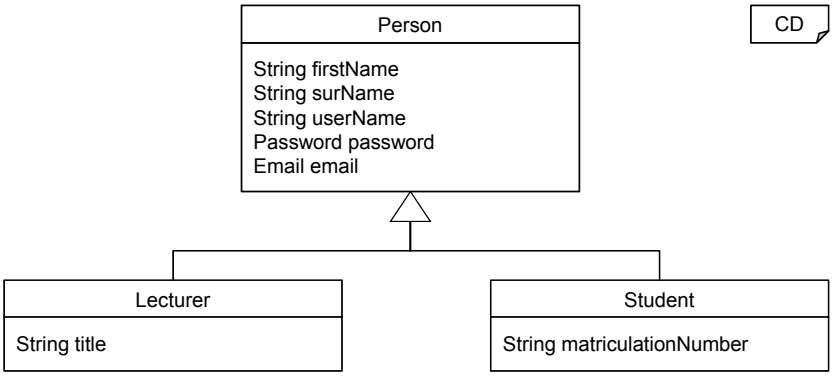


Abbildung 6.3.: Beispiel für ein Klassendiagramm mit Vererbung

In Abbildung 6.4 ist das Klassendiagramm in textueller Form dargestellt. Eine Klasse wird über das Schlüsselwort `class` gefolgt von ihrem Namen definiert. Die Vererbungsbeziehung wird, wie z. B. in Java üblich, per Schlüsselwort `extends` deklariert. Die Attribute einer Klasse werden in einer Syntax ähnlich Java definiert. Zunächst erfolgt die Angabe des Datentyps (z. B. `String` (Zeile 2) oder `Email` (Zeile 6)) gefolgt von dem zu verwendenden Namen (`firstName` bzw. `email`). Für den Datentyp kann generell jeder Klassenname verwendet werden, wobei innerhalb von MontiWIS jedoch lediglich bestimmte Datentypen mit erweiterter Funktionalität unterstützt werden. Eine Auflistung und Erläuterung dieser Datentypen erfolgt in Abschnitt 6.2.3.

CD

```
class Person {
  String firstName;
  String surName;
  String userName;
  Password password;
  Email email;
}

class Student extends Person {
  String matriculationNumber;
}

class Lecturer extends Person {
  String title;
}
```

Abbildung 6.4.: Klassendiagramm aus Abbildung 6.3 in textueller Form

Ferner können auch Assoziationen zwischen Klassen definiert werden, wie im Folgenden erläutert wird.

Assoziationen

Assoziationen zwischen Klassen dienen in der Regel dazu, eine Beziehung zwischen ihnen auszudrücken. In Abbildung 6.5 ist eine solche beispielsweise zwischen den Klassen `Student` und `CourseOfStudy` dargestellt.

Diese besitzt eine *Assoziationsrolle* (hier: `students` und `studentOf`) die angibt, in welcher Beziehung die jeweils andere Klasse steht. Im vorliegenden Beispiel sind die Studiengänge in der Rolle `studentOf` mit der Klasse `Student` verbunden, in der Gegenrichtung ist `Student` in der Rolle `student` einem Studiengang zugeordnet. An einer Assoziation ist es möglich, *Navigationsrichtungen* anzugeben. Dies geschieht durch einen ein- oder beidseitigen Pfeil an der Assoziationslinie. Bei der Assoziation im Beispiel handelt es sich um eine bidirektionale Assoziation, was zu Folge hat, dass von jeder beteiligten Klasse die jeweils andere erreichbar ist. Bei der Verwendung einer unidirektionalen Assoziation kann nur in Pfeilrichtung navigiert werden, d. h. auf die Klasse, die sich am Assoziationsende mit der Pfeilspitze befindet, kann von der anderen Klasse aus zugegriffen werden, andersherum jedoch nicht.

Ferner können Assoziationen *Kardinalitäten* besitzen. Kardinalitäten geben an, wie häufig die assoziierte Klasse maximal verbunden sein kann. Entgegen einer Behandlung gemäß der UML/P [Rum11, Rum12] oder UML [Obj11c, RQZ07] werden innerhalb des Datenmodells von MontiWIS keine Kardinalitäten behandelt, die eine Referenz als obligatorisch kennzeichnen. Diese Besonderheit resultiert aus der Tatsache, dass sich über die Seitenbeschreibungssprache (siehe Kapitel 7) Sichten auf die definierte Datenstruktur erstellen lassen, die potenziell auch obligatorische Attribute bzw. Assoziationen ausblenden könnten.

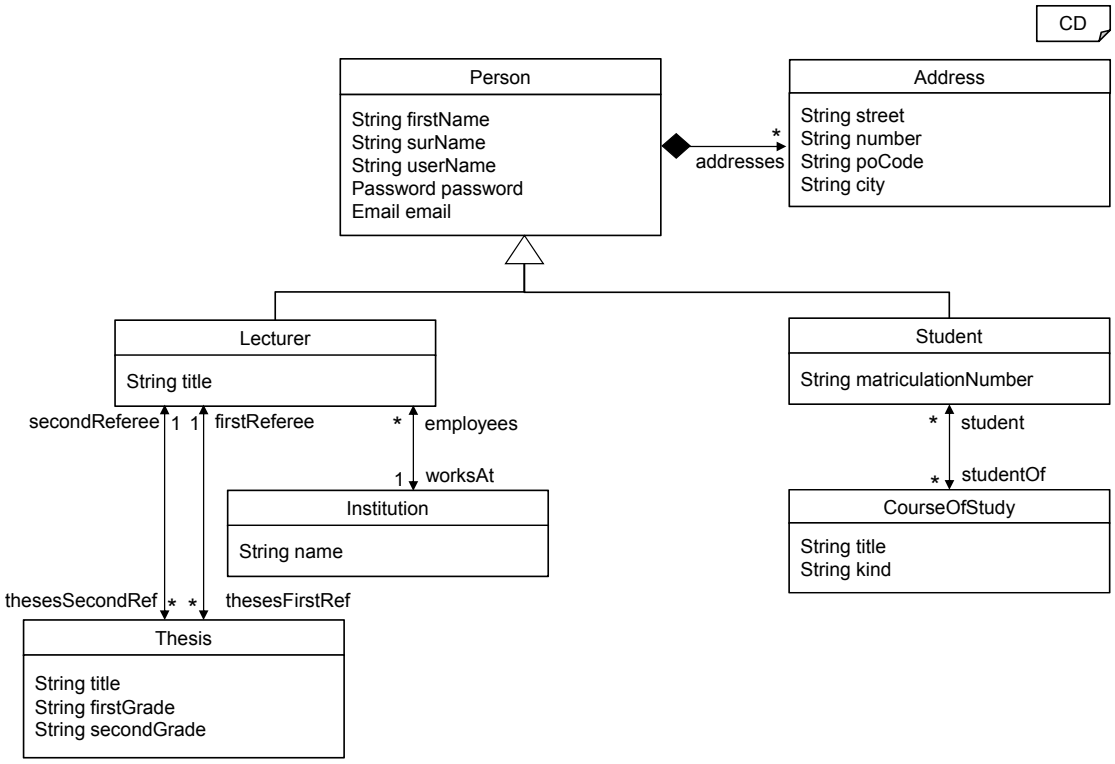


Abbildung 6.5.: Beispiel für ein Klassendiagramm mit Assoziation und Komposition

Dies hätte zur Folge, dass bestimmte Objekte niemals gespeichert werden könnten, da das Datenmodell ein bestimmtes Attribut oder eine Assoziation als obligatorisch kennzeichnet, die Seite, die die Klasse darstellt, das betreffende Attribut aber nicht enthält. Die Kardinalität wird (ebenso wie der Rollenname) am jeweils gegenüberliegenden Assoziationsende angegeben. Eine Kardinalität `1` an einem Assoziationsende (bzw. das Fehlen einer Angabe dort) impliziert, dass die betreffende Klasse maximal einmal referenziert wird. In der generierten Klasse erfolgt eine Abbildung auf ein Attribut der referenzierenden Klasse, das den Namen der Rolle trägt. Fehlt ein Rollenname, wird der Name der Zielklasse übernommen. Bei einer Kardinalität $*$ können beliebig viele Klassen referenziert werden. Im Falle der mehrfachen Assoziation wird in der generierten Klasse ein `Set` aus Objekten der referenzierten Klasse erzeugt. Der Name des Attributs ergibt sich durch Übernahme des Rollennamens. Fehlt dieser, wird der Name der referenzierten Klasse verwendet.

Die textuelle Notation von Assoziationen ist in Abbildung 6.6 dargestellt. Zeile 35 stellt die Assoziation zwischen einem `Student` und einem `CourseOfStudy` dar. Eingeleitet wird die Assoziation über das Schlüsselwort `association`, gefolgt von der Kardinalität auf Seiten der nachfolgenden Klasse, angegeben in eckigen Klammern. Eine Kardinalität wird ebenso wie an der Assoziation in der grafischen Version eines Klassendiagramms codiert. Sie kann entweder `1` sein und so eine maximale Anzahl von einem referenzierten Objekt ausdrücken, $*$ sein, um

CD

```
class Person {
  String firstName;
  String surName;
  String userName;
  Password password;
  Email email;
}

class Student extends Person {
  String matriculationNumber;
}

class Lecturer extends Person {
  String title;
}

class CourseOfStudy {
  String title;
  String kind;
}

class Address {
  String street;
  String number;
  String poCode;
  String city;
}

class Thesis {
  String title;
  String firstGrade;
  String secondGrade;
}

association [*] Student (students) <-> (studentOf) CourseOfStudy [*];
association [1] Student (student) <-> (thesis) Thesis [1];
association [*] Lecturer (employees) <-> (worksAt) Institution [1];
association [1] Lecturer (firstReferee) <-> (thesesFirstRef) Thesis [*];
association [1] Lecturer (secondReferee) <-> (thesesSecondRef) Thesis [*];
composition Person -> (addresses) Address [*];
```

Abbildung 6.6.: Klassendiagramm aus Abbildung 6.5 in textueller Form

eine beliebige Maximalanzahl zu codieren, oder inklusive der eckigen Klammern weggelassen werden, was äquivalent zu einer 1 ist. Der Rollenname, der an der entsprechenden Seite der Assoziation angetragen ist, folgt in runden Klammern. Analog ist die andere an der Assoziation beteiligte Klasse angegeben, deren Rolle und Kardinalität auf der rechten Seite des Assoziationspfeils steht. Die Verbindung inklusive der Navigationsrichtung ist zwischen

den beiden Klassen angegeben. Bidirektionale Assoziationen werden durch einen beidseitigen Pfeil (`<->`) oder eine Assoziation ohne Pfeilspitzen (`-`) codiert. Unidirektionale Assoziationen werden durch einen Pfeil in eine Richtung symbolisiert. Die Richtung wird dabei intuitiv festgelegt. Soll von der Klasse auf der linken Seite der Assoziation zur Rechten navigiert werden können, so ist ein Pfeil mit Spitze auf der rechten Seite (`->`) zu verwenden, bei umgekehrter Richtung ein Pfeil mit Spitze auf der linken Seite (`<-`). In Anwendung ist dies in den weiteren Assoziationen (Zeilen 35-39) zu sehen.

Die Darstellung von Assoziationen in der Benutzeroberfläche des generierten Web-Informationssystems ist für einfache und mehrfache Assoziationen identisch, lediglich das Verhalten variiert. Für jede Assoziation wird in der Oberfläche ein Feld vorgesehen, in dem die referenzierten Objekte aufgelistet sind. Hinter jeder Referenz existiert ein Link zum Löschen der Assoziation. Ferner ist es möglich, neue Verbindungen hinzuzufügen. Dazu kann über einen Link ein Pop-up-Fenster geöffnet werden, das eine Auswahl der möglichen Objekte erlaubt. Diese werden nach Auswahl der Liste der vorhandenen Referenzen hinzugefügt und bei Absenden des Formulars gespeichert. Beispielhaft sind die generierten Elemente in Abbildung 7.26 dargestellt.

Kompositionen

Neben gewöhnlichen Assoziationen existieren in Klassendiagrammen der UML/P und damit auch in MontiWIS, analog zu den in Abschnitt 6.1.2 definierten Anforderungen, *Kompositionen*. Ähnlich wie Assoziationen dienen sie dazu, Relationen zwischen zwei Klassen zu spezifizieren. Ausgedrückt wird jedoch nicht ausschließlich die Verbindung zwischen zwei Klassen, sondern eine Teil-Ganzes-Beziehung. Im Beispiel in Abbildung 6.5 existiert eine Komposition zwischen den Klassen `Person` und `Address`, wobei `Person` das Ganze ist, `Address` der inkludierte Teil. Dargestellt wird eine Komposition über eine Verbindungslinie, die mit einer Raute endet. Die so bezeichnete Seite markiert die zusammengesetzte Klasse. Inhaltlich ist die Existenz eines Teils an die Existenz eines Ganzen gekoppelt. Eine Adresse kann also nur existieren, wenn auch eine Person existiert, deren Teil sie ist. Wie in [Rum11] vorgeschlagen, werden Lebenszyklus und die Austauschbarkeit im Kontext von MontiWIS definiert. Hinsichtlich des Lebenszyklus wird sich sowohl an der UML/P als auch an [Obj11c, S. 38] orientiert. Teile existieren in dem Kontext, d. h. in Verbindung zu dem Ganzen, in dem sie erzeugt wurden. Dies bedeutet auch, dass sie bei Löschen des Ganzen ebenfalls gelöscht werden. Hinsichtlich der Austauschbarkeit von Objekten einer Klasse wird in MontiWIS jedoch vom UML-Standard abgewichen. Objekte der Klasse, die die Rolle des Teils einnimmt, können nicht von ihrem Ganzen gelöst und in anderem Kontext weiterverwendet werden, sondern können lediglich angelegt und gelöscht werden. Ist es erwünscht, dass Objekte auch unabhängig vom assoziierten Objekt existieren können, so sind gewöhnliche Assoziationen zu verwenden.

Die textuelle Notation von Kompositionen ist in Abbildung 6.6 in Zeile 40 dargestellt. Eingeleitet wird eine Komposition durch das Schlüsselwort `composition` gefolgt von dem Namen der Klasse, die das Ganze (hier: `Person`) repräsentiert (theoretisch wäre auch die Angabe des Teiles zuerst möglich, allerdings muss die Assoziation in diesem Fall in die andere Richtung weisen). Eine Angabe einer Kardinalität auf Seiten der Komposition erfolgt nicht.

Da die eingebundene Klasse lediglich Teil eines Ganzen sein kann, ist die Kardinalität per Definition `1`, daher kann darauf verzichtet werden. Von der komponierten Klasse erfolgt durch Verwendung des gerichteten Assoziationspfeils (`->`) nun die Referenz auf die eingebundene Klasse (im Beispiel: `Address`) unter Angabe des Rollennamens (hier: `addresses`). Dieser Name dient als Variablenname in der generierten Klasse und gibt an, wie von Seiten der einbindenden Klasse aus auf seine Komponenten zugegriffen werden kann. Sollte kein Rollenname angegeben sein, wird der Name der Komponentenklasse mit angehängtem `s` übernommen.

Im generierten System werden Kompositionen direkt in der Oberfläche der einbindenden Klasse angezeigt. Zu jeder Komponente existiert ein Link, der zum Hinzufügen eines neuen Objekts der Komponentenklasse dient und eine Eingabemaske innerhalb der existierenden erzeugt. Hier können die Werte der Komponente belegt und zusammen mit der einbindenden Klasse gespeichert werden. Die Darstellung von Kompositionen in der generierten Seite ist beispielhaft in Abbildung 7.27 dargestellt.

6.2.3. Vordefinierte Datentypen

Obwohl prinzipiell jede Klasse, die in einem Klassendiagramm deklariert ist, auch als Datentyp für Attribute verwendet werden kann, existieren in MontiWIS spezielle *vordefinierte Datentypen*. Durch solch spezielle Datentypen ist eine explizit definierte Behandlung während des gesamten Generierungsprozesses und auch im generierten System möglich.

Um bereits auf Modellebene angemessen mit den vordefinierten Datentypen arbeiten zu können und sie korrekt als Klasse referenzieren zu können, wird in jedem MontiWIS-Anwendungsprojekt ein Klassendiagramm mitgeliefert, welches bereits Klassendeklarationen für sämtliche vordefinierten Datentypen enthält. Damit das Klassendiagramm als Definitionsort für derartige Datentypen erkannt und entsprechend verwendet wird, ist die Auszeichnung des Klassendiagramms mit einem Stereotypen notwendig. Darauf wird in Abschnitt 6.2.4 genauer eingegangen. Für eine Nutzung dieser Klassen ist lediglich ein Import des entsprechenden Klassendiagramms erforderlich, was in Abbildung 6.2 dargestellt ist. Im Folgenden werden die einzelnen Datentypen und deren Behandlung überblicksartig beschrieben. Eine detaillierte Beschreibung der Umsetzung auf Ebene der Benutzeroberfläche erfolgt in Abschnitt 7.2.3, wo auch Ausschnitte der generierten Seiten zu finden sind.

In Tabelle 6.7 werden die Eigenschaften der vordefinierten Datentypen zusammengefasst und im weiteren Verlauf dieses Abschnitts erläutert.

String

Der Datentyp `String` entspricht im Wesentlichen einem Freitext, wie er in quasi jeder Programmiersprache vorkommt. Im generierten System wird dieser Datentyp auf ein Attribut vom Typ `java.lang.String` abgebildet. Ein `String` kann im generierten System über ein einzeiliges Textfeld eingegeben und verändert werden. Eine feste Längenbeschränkung existiert nicht, jedoch legt die optische Beschränkung des Eingabefeldes bereits nahe, dass hier eher kurze Texte eingegeben werden sollen. Eine Überprüfung hinsichtlich korrekter Werte erfolgt bei diesem Datentyp nicht, da jede beliebige Zeichenkette als valide angesehen wird.

Attributtyp	Zieldatentyp	Benutzeroberfläche	Anmerkung
`String`	`java.lang.String`	Darstellung über ein einzeiliges Textfeld	keine Validierung der Eingabe
`Text`	`java.lang.String`	Darstellung über ein mehrzeiliges Eingabefeld	keine Validierung der Eingabe
`Date`	`java.util.Date`	Darstellung über ein Datumsauswahlfeld	
`Integer`	`java.lang.Integer`	Darstellung über ein einzeiliges Textfeld	
`Float`	`java.lang.Float`	Darstellung über ein einzeiliges Textfeld	
`Bool`	`java.lang.Boolean`	Darstellung über einen Radiobutton	
`Email`	`java.lang.String`	Darstellung über ein einzeiliges Textfeld	
`Password`	`java.lang.String`	Darstellung über ein Passwortfeld	Speicherung in gehashter Form

Tabelle 6.7.: Vordefinierte Datentypen in MontiWIS

Text

Ebenso wie `String` dient der Datentyp `Text` dazu, beliebige Zeichenketten zu speichern und darzustellen. Der Unterschied zwischen beiden Datentypen ist jedoch nicht der Datentyp im generierten System (hier kommt ebenfalls `java.lang.String` zum Einsatz), sondern die Darstellung in der generierten Benutzeroberfläche. Ein `Text` wird hier in einem mehrzeiligen Textfeld (`textarea` in HTML) eingegeben und bearbeitet. Diese Unterscheidung wurde vorgenommen, um auch längere Texte komfortabel verarbeiten zu können und dies in der Oberfläche entsprechend anzudeuten. Identisch ist bei diesem Datentyp die nicht stattfindende Validierung der Eingabe.

Date

Über den Datentyp `Date` wird ein Datum abgebildet. Dies besteht, wie allgemein üblich, aus einer Angabe von Tag, Monat und Jahr. Die Speicherung im generierten System erfolgt durch ein `java.util.Date`, worüber die regulären Funktionen zur Verarbeitung dieses Datentyps bereitstehen. Um die Eingabe in einem Browser zu erleichtern, wird an entsprechender Stelle in der Bedienoberfläche ein Datumsauswahlfeld verwendet. Solche Elemente ermöglichen durch Einblendung eines Pop-Up-Fensters die grafische Auswahl eines Datums, so dass ein

Nutzer bei der Wahl sowohl des Datumswertes als auch seines korrekten Formats unterstützt wird. Die Validierung des eingegebenen Wertes findet client- und serverseitig statt. Sollte ein Wert in inkorrektem Format in das Feld eingetragen werden, so wird der Nutzer bereits vor Absenden des Formulars darauf hingewiesen. Auf Seite des Servers wird erneut geprüft, ob die Werte einem Datum entsprechen und das Formular gegebenenfalls vor der Weiterverarbeitung der Daten mit der Bitte um Korrektur zurückgesandt.

Integer

Ganze Zahlen können in MontiWIS über den Datentyp `Integer` dargestellt werden. Als Datentyp im generierten System kommt `java.lang.Integer` zum Einsatz. Für die Darstellung in einem Formular wird ein einfaches Textfeld verwendet, welches jedoch bereits clientseitig die Eingaben validiert. Auch hier wird ebenfalls auf dem Server überprüft, ob die Eingabe einer ganzen Zahl entspricht.

Float

Neben ganzen Zahlen können auch Gleitkommazahlen verarbeitet werden. Der hierfür im Klassendiagramm zu verwendende Datentyp ist `Float`. Im generierten System wird ein solcher Datentyp auf `java.lang.Float` abgebildet. Die Eingabe eines Wertes erfolgt in einem regulären, einzeiligen Textfeld. Es wird bereits im Browser überprüft, ob es sich dabei um eine Dezimalzahl handelt. Auf Seite des Servers wird vor einem Abspeichern nochmals das Format der Eingabe überprüft.

Boolean

Um einen booleschen Wert als Attribute in einer Klasse zu verwenden, ist der Datentyp `Boolean` im Klassendiagramm anzugeben. Im generierten System wird für diesen Datentyp die Klasse `java.lang.Boolean` verwendet. In der Benutzeroberfläche werden dem Nutzer zwei Radiobuttons mit den Auswahlmöglichkeiten *yes* oder *no* angezeigt. Diese Werte werden auf Serverseite entsprechend zu `true` und `false` umgewandelt und verarbeitet.

Email

In Web-Informationssystemen tritt häufig der Fall auf, dass eine E-Mail-Adresse gespeichert werden soll, z. B. als Kontaktinformation zu einem Nutzer. Damit hier ausschließlich korrekte und funktionierende Daten eingegeben werden, ist eine Validierung sowohl auf Client- als auch auf Serverseite wünschenswert. Um dieser Anforderung gerecht zu werden, existiert in MontiWIS der eigenständige Datentyp `Email`. Als Datentyp im generierten System wird die Klasse `java.lang.String` verwendet. In der Benutzeroberfläche erfolgt die Eingabe in ein gewöhnliches Textfeld. Eine Verifizierung, ob der gegebene Domain- Teil der E-Mail-Adresse zu einer existierenden Internetadresse gehört, findet jedoch nicht statt.

Password

Nutzer in einem Web-Informationssystem verwenden für die Anmeldung am System in der Regel eine Kombination aus einem Nutzernamen und einem Passwort. Ersteres kann über ein Datum vom Typ `String` oder auch per Identifikation über eine `Email` erfolgen, Letzteres geschieht üblicherweise über ein `Password`. Im Kontext einer Webapplikation nimmt ein Passwort eine Sonderstellung ein. Passwörter sollten derart gespeichert werden, dass niemand den Klartext des Passworts auslesen kann. Dies betrifft die Speicherung auf Serverseite, aber auch die Eingabe in ein Formularfeld. Diese Anforderungen wurden in MontiWIS berücksichtigt. Die Abbildung des Datentyps erfolgt im generierten System auf ein `java.lang.String`, der Wert des Datums entspricht jedoch nicht dem Klartext des eingegebenen Wertes. Beim Belegen eines solchen Datentyps mit einem Wert wird standardmäßig ein Message-Digest Algorithm 5 (MD5)-Hash über den gegebenen Wert ausgeführt und erst danach dieser Hashwert gespeichert. Dadurch wird etwa auch verhindert, dass durch Auslesen der Werte aus der Datenbank Rückschlüsse auf das Passwort gezogen werden können. Weitere Überlegungen hinsichtlich der Speicherung von Passwörtern sind in Abschnitt 7.4.5 zu finden. In der Benutzeroberfläche wird zur Eingabe eines Passworts ein Passwortfeld genutzt. Dies verschleiert die Eingabe durch Darstellung von grafischen Symbolen (in der Regel Punkte) anstelle des Klartextes. Auch bei der Darstellung eines Passworts werden grafische Symbole stets gleicher Anzahl dargestellt.

6.2.4. Profil für MontiWIS und Modellierungsrichtlinien

Die vorgestellten Sprachelemente stellen die Grundlage für die Modellierung der Domänendatenstruktur innerhalb von MontiWIS dar. Sie sind lediglich eine Teilmenge der UML/P und die beschriebene Bedeutung ist domänenspezifisch auf die Modellierung eines Web-Informationssystems zugeschnitten. Die vordefinierten Datentypen sind ebenso auf die Domäne und das Gesamtkonzept abgestimmt wie die restringierte Verwendung von Kardinalitäten an Assoziationen und Kompositionen. Die vordefinierten Klassen selbst lassen sich mit Sprachmitteln der UML/P ausdrücken, benötigen jedoch die Angabe eines Stereotyp, damit das Klassendiagramm als eines erkannt wird, das eben diese vordefinierten Klassen enthält.

Neben diesem existieren weitere Stereotypen mit spezieller Bedeutung in MontiWIS sowie weitere Anforderungen an die Modelle. Sowohl mögliche Stereotypen als auch notwendige Klassen bzw. Assoziationen inklusive deren Hintergründe werden in folgenden Abschnitten erläutert. In Tabelle 6.8 werden die in MontiWIS möglichen Stereotypen zusammengefasst.

Vordefinierte Datentypen durch den Stereotyp implicit

Wie bereits in Abschnitt 6.2.3 beschrieben, werden vordefinierte Datentypen in einem eigenständigen Klassendiagramm spezifiziert. Dieses sollte in das Klassendiagramm, das die Domänendatenstruktur definiert, importiert werden, um die Datentypen nutzen zu können. Zur eindeutigen Identifizierung muss vor die Deklaration des Klassendiagramms der Stereotyp «`implicit`» gesetzt werden. Abbildung 6.9 zeigt, wie das Klassendiagramm somit aufgebaut sein sollte.

Stereotyp	Verwendung	Erläuterung
`implicit`	vor einem Klassendiagramm	Dient der Spezifikation des Klassendiagramms, in dem die vordefinierten Datentypen definiert werden
`user`	vor einer Klasse	Spezifiziert die Klasse, die einen Nutzer innerhalb des Systems identifiziert
`inList`	vor einem Attribut	Markiert die Verwendung des Attributs als Teil des Labels in der Listendarstellung

Tabelle 6.8.: Stereotypen in MontiWIS

CD

```
1 package tm;
2
3 <<implicit>>
4 classdiagram Standardtypes {
5
6   class String;
7
8   // weitere implizite Klassendeklarationen
9 }
```

Abbildung 6.9.: Spezifikation von vordefinierten Klassen über den Stereotyp `implicit`

Zeile 3 enthält den relevanten Stereotyp. Innerhalb des Klassendiagramms sind die Klassen definiert, ohne jedoch Attribute oder Methoden zu nennen. Die Spezifikation der vordefinierten Datentypen an einen Stereotyp zu koppeln hat den Vorteil, dass anstelle des dargestellten Klassendiagrammnamens `Standardtypes` jeder beliebige andere Name verwendet werden kann. Lediglich auf den korrekten Import in die Klassendiagramme der Domänendatenstruktur ist zu achten.

Nutzerdeklaration über Stereotyp user

Ein Web-Informationssystem bietet in der Regel die Möglichkeit, Nutzerkonten anzulegen. Klassen, die einen Nutzer repräsentieren, sind gewöhnlich Teil des Domänendatenmodells und daher auch im Klassendiagramm zu spezifizieren. Die Klasse, die ein Nutzerkonto repräsentiert, wird in MontiWIS über den Stereotypen «`user`» festgelegt. Auch hier stellt die Verwendung eines Stereotyps gegenüber einer Namenskonvention einen Vorteil dar, da eine beliebige Klasse der modellierten Domäne zur Verwendung als Nutzerklasse annotiert werden kann und diese nicht auf einen Namen festgelegt ist. In Abbildung 6.10 ist dargestellt, wie eine Klasse auszusehen hat, um als ein Nutzer behandelt und für die Authentifizierung verwendet

zu werden.

CD

```
1 <<user>>
2 class Person {
3   String userName;
4   Password password;
5
6   // domänenspezifische Attribute
7   <<inList>> String firstName;
8   <<inList>> String surName;
9 }
```

Abbildung 6.10.: Grundaufbau einer Klasse für Nutzerkonten

Zeile 1 enthält den notwendigen Stereotyp hierfür. Weitere Spezifika, die für eine funktionierende Nutzerverwaltung bei der Modellierung der Datenstruktur zu beachten sind, werden in Kürze erläutert.

Festlegung der Beschriftungen über den Stereotyp inList

Generell ist es möglich, aus einem Klassendiagramm und den darin enthaltenen Klassen CRUD-Standardfunktionalität im generierten Web-Informationssystem abzuleiten. Dinge wie Eingabemasken und die Bezeichnung von Feldern sind allein aus dem Datentyp und dem Namen eines Attributs herzuleiten (für die Beschreibung der grundlegenden Funktionsweise siehe Abschnitt 6.4). Für bestimmtes Verhalten ist jedoch spezielles Domänenwissen und seine explizite Festlegung im Modell notwendig, da hier Standardannahmen in der Regel nicht zu zufriedenstellenden Ergebnissen führen können. Ein Anwendungsfall, wo eben solches Domänenwissen notwendig ist, ist die Listendarstellung von Objekten oder auch die Referenz auf ein bestimmtes Objekt in der Web-Oberfläche. Ein Generator kann allein aus dem Domänendatenmodell nicht ableiten, welche Attribute ein Objekt für den Nutzer eines Systems hinreichend identifizierbar machen. Sollten Seiten manuell spezifiziert sein, so stellt dies kein Problem dar. In diesem Fall ist es möglich, bei der Angabe des Listenelements bzw. bei der Referenz auf ein anderes Objekt (siehe hierzu Abschnitt 7.2.2) zu spezifizieren, welche Attribute an speziell dieser Stelle angezeigt werden sollen. So können in unterschiedlichen Anwendungskontexten unterschiedliche Darstellungen eines Objekts realisiert werden.

Sollte lediglich die Klassendefinition mit einer großen Anzahl von Attributen vorliegen, ist es für den Generator des Systems nicht entscheidbar, welche Attribute davon in der standardmäßig generierten Listenansicht angezeigt werden sollen. Eine vollständige Anzeige des Objekts ist sicherlich selbst bei einer geringen Anzahl an Attributen nicht praktikabel. Auch Datentypen wie beispielsweise `Text` lassen sich nicht in übersichtlicher Art und Weise in einer Listenansicht darstellen.

Um hier eine Möglichkeit zu schaffen, bestimmte Attribute bereits im Datenmodell als Teil der Darstellung zu wählen, existiert in MontiWIS die Möglichkeit, durch Verwendung

des Stereotypen «inList» die gewünschten Attribute festzulegen. Diese werden in jedem Fall in der generierten Standardansicht zur Darstellung herangezogen. Auf einer manuell spezifizierten Seite werden sie verwendet, sofern nicht explizit andere Attribute festgelegt wurden. Wurden keine der Attribute mit dem betreffenden Stereotypen ausgezeichnet und soll dennoch eine Liste angezeigt werden, wird der eindeutige Bezeichner (id) des Objekts in der Liste dargestellt, was zwar nicht sehr benutzerfreundlich ist, aber immerhin zu einer zweifelsfreien Unterscheidbarkeit der Objekte führt.

Als Beispiel soll die Klasse Person aus Abbildung 6.10 dienen. Für eine Klasse, die Daten einer Person speichert, ist es sicherlich sinnvoll, den Vor- und Nachnamen für die Anzeige zu verwenden. Aus diesem Grund wurden die Attribute firstName (Zeile 7) und surName (Zeile 8) mit dem Stereotypen «inList» versehen. Damit werden eben diese Attribute angezeigt, wenn auf eine Person referenziert wird oder mehrere Personen in einer Liste angezeigt werden.

Notwendige Attribute der Nutzerklasse

Wenn eine Klasse als Nutzerklasse fungiert, sind üblicherweise auch Anmeldeinformationen notwendig, um einen Nutzer eindeutig zu identifizieren. Wie gewohnt erfolgt auch in MontiWIS die Authentifizierung eines Nutzers über einen Nutzernamen in Kombination mit einem geheimen Passwort. Das Attribut, das den Nutzernamen repräsentiert, muss hierbei vom Datentyp String sein, das Passwort ist über den Datentyp Password abzubilden. Diese Voraussetzungen sind für eine funktionierende Authentifizierung im generierten System notwendig. Anders als im Falle der Nutzerklasse, bei der es je nach Anwendungsdomäne unterschiedliche Klassen geben kann, ist bei den Anmeldeinformationen sowohl Datentyp als auch Bedeutung der Attribute festgelegt. Zwar wäre es denkbar gewesen, einen Stereotypen «userName» zur Festlegung des Attributs, das den Nutzernamen repräsentiert, und einen Stereotypen «password» zur Deklaration des Passwortattributs zu definieren, jedoch würde sich dies in der überwiegenden Zahl der Fälle ohnehin in dem Attributnamen wiederfinden. Daher wurde für diesen Aspekt der Datenstruktur darauf verzichtet, gesondert Stereotypen zu definieren, die die Attribute als Nutzername bzw. Password deklarieren. Stattdessen wird davon ausgegangen, dass die Nutzerklasse analog zu Abbildung 6.10 aufgebaut ist und die Attribute userName vom Datentyp String (Zeile 3) und password vom Datentyp Password exakt so vorliegen.

Notwendige Klasse GroupInternal zur Spezifikation von Gruppen

Neben der Authentifizierung eines Nutzers ist ebenfalls von Bedeutung, welcher *Nutzergruppen* ein Nutzer angehört. Prinzipiell ist die Rollenzuordnung als eine Assoziation eines Objekts der Nutzerklasse zu einem oder mehreren Objekten einer Gruppenklasse zu betrachten. Obwohl eine Gruppe nicht originär Teil des Domänenmodells ist, sondern zur Konfiguration des Web-Informationssystems gehört, wird die Klasse im Domänendatenmodell berücksichtigt. Dieses Vorgehen wurde insofern als sinnvoll erachtet, als auch die Nutzerklasse [14] bereits

[14] Streng genommen ist auch diese Klasse nicht ausschließlich Teil des Domänendatenmodells, sondern, da sie auch zur Konfiguration des Web-Informationssystems verwendet wird, zumindest teilweise auf einer

dort modelliert wird. Durch die explizite Verwendung der Gruppenklasse wird ebenfalls erreicht, dass sämtliche Standardfunktionalität direkt im generierten Web-Informationssystem zur Verfügung steht. Objekte einer Gruppe können angelegt, verändert und Nutzern zugeordnet werden, ohne dass dafür separat Funktionalität oder Eingabemasken generiert werden müssten. Auch die Zuordnung zu einem Nutzer lässt sich so realisieren.

Die Klasse, die zur Repräsentation einer Nutzergruppe vorgesehen ist, muss den Namen `GroupInternal` tragen und analog zu Abbildung 6.11 aufgebaut sein. Dies ist als Minimalanforderung zu verstehen, d. h. die dargestellten Attribute müssen exakt so vorkommen. Weitere Attribute sind jedoch prinzipiell möglich.

CD

```
1 class GroupInternal {
2   <<inList>> String groupName;
3   String groupDesc;
4 }
```

Abbildung 6.11.: Struktur der Klasse `GroupInternal` zur Gruppendefinition

Der Name der Klasse wurde bewusst etwas sperrig gewählt, da so die Wahrscheinlichkeit, dass es zu Namenskollisionen innerhalb des Domänendatenmodells kommt, geringer ist als wenn die Klasse beispielsweise lediglich `Group` hieße. Neben dem festgelegten Namen muss ebenfalls das Attribut `groupName` vom Datentyp `String` vorhanden sein. Dessen Werte werden zur Identifikation der Gruppe genutzt und durch die Verwendung des Stereotyps «`inList`» in Referenzen und Listen angezeigt. Das Attribut `groupDesc` vom Datentyp `String` kann optional dazu dienen, eine Beschreibung oder kurze Erläuterung anzugeben. Für die korrekte Funktionsweise der Nutzergruppen ist es jedoch nicht relevant.

Da auch hier der Ansatz verfolgt wird, dass auch Klassen, die theoretisch implizit hinzu generiert werden könnten, explizit im Klassendiagramm spezifiziert werden, ist diese Klasse in dieser Form anzugeben. Ein entsprechendes Muster ist daher beim Anlegen eines neuen MontiWIS-Anwendungsprojekts als Teil des dort vorliegenden Klassendiagramms bereits vorhanden.

Nutzergruppen könnten zwar manuell von einem Nutzer (oder einem Administrator) des generierten Systems angelegt werden, sinnvoll wäre dieses Vorgehen jedoch nicht, da die Berechtigungen, die für die Nutzergruppen festgelegt werden, auf Modellebene bereits zum Zeitpunkt der Generierung spezifiziert werden. Die Nutzergruppen werden aus dem Applikationsmodell abgeleitet und bei der Initialisierung der Datenbank erzeugt. Details sowohl zur Modellierung der Gruppen als auch der Umsetzung sind in Kapitel 9 beschrieben. Die manuelle, über die Zeit änderbare Zuordnung von Nutzern zu Nutzergruppen ist jedoch sehr wohl sinnvoll und der festen Verdrahtung von Nutzern zu Gruppen vorzuziehen. Die Voraussetzungen für die Zuordnung werden im nachfolgenden Abschnitt beschrieben.

Meta-Ebene angesiedelt.

Notwendige Assoziation zwischen Nutzerklasse und -gruppe

Die im vorherigen Abschnitt beschriebene Klasse `GroupInternal`, die zur Repräsentation einer Nutzergruppe im Web-Informationssystem dient, ist tatsächlich nur sinnvoll, wenn sie auch zu der Klasse, die einen Nutzer darstellt, in Relation steht. Daher ist es notwendig, dass eine Assoziation zwischen diesen beiden Klassen existiert, so wie sie in Abbildung 6.12 dargestellt ist.

```
1 association [*] Person (users) <-> (groups) GroupInternal [*];                CD
```

Abbildung 6.12.: Assoziation zwischen Nutzerklasse und -gruppe

Sollte für die Nutzerklasse eine andere Klasse gewählt worden sein (durch Annotation der Klasse über den Stereotyp «`user`»), ist diese an dieser Stelle zu verwenden. Da die Klasse `GroupInternal` vorgeschrieben ist, kann diesbezüglich nicht variiert werden.

Bei der Assoziation ist es sinnvoll, dass sie bidirektional ist, so dass sich einerseits die Gruppenzugehörigkeit von Seiten eines Nutzers abfragen lässt, aber ebenfalls von einer Gruppe aus ermittelt werden kann, welche Nutzer ihr angehören. Die Kardinalitäten sollten ebenfalls wie dargestellt gewählt werden, da eine Gruppe stets mehrere Mitglieder haben, ein Nutzer aber auch zu mehreren Gruppen gehören kann.

6.3. Generierungsprozess

Wie bereits in Kapitel 4 allgemein beschrieben, erfolgt eine Bearbeitung von Modellen unter Verwendung des Frameworks MontiCore. Im Folgenden werden die Schritte, die bei der Bearbeitung von UML/P Klassendiagrammen ausgeführt werden, detaillierter betrachtet. Zunächst wird kurz in Abschnitt 6.3.1 auf die Kontextbedingungen eingegangen, die nach dem Parsen geprüft werden und die hinreichende Konsistenz innerhalb der Modelle gewährleisten (sofern Verstöße dagegen entsprechend beseitigt werden). Anschließend wird in Abschnitt 6.3.2 die Zwischenstruktur erläutert, die für die weitere Bearbeitung im Rekonstruktions- und Generierungsprozess die Grundlage liefert. Anschließend erfolgt in Abschnitt 6.3.3 die Beschreibung des Visitors, da bei der Modellverarbeitung einige Besonderheiten zu beachten waren. Auf der entstandenen Zwischenstruktur werden auch einige Rekonstruktionsschritte ausgeführt, die im Anschluss in Abschnitt 6.3.4 beschrieben werden. Abschnitt 6.3.5 geht dabei gesondert auf die Erzeugung von Standardfunktionalität als Teil des Rekonstruktionsprozesses ein. Abschließend wird in Abschnitt 6.3.6 auf die Besonderheiten des Generators und der verwendeten Templates eingegangen.

6.3.1. Kontextbedingungen

Im ersten Schritt bei der Verarbeitung von Klassendiagrammen werden die Modelle vom Parser eingelesen und die Symboltabellen aufgebaut. Hierfür wird die Infrastruktur verwendet, die im Rahmen der Dissertation von Steven Völkel entstanden ist [Völ11]. Da die UML/P Klassendiagramme sowohl auf den Grammatiken als auch der Implementierung der Symboltabellenïnfrastruktur aufsetzen, die in der Dissertation von Martin Schindler [Sch12] entstanden sind, werden auch diese Komponenten in MontiWIS verwendet. Dies beinhaltet nicht nur den Aufbau der Symboltabelle, sondern auch eine Reihe von Kontextbedingungen, die zur Prüfung der Intra- und Intermodellkonsistenz Verwendung finden.

Intramodellkonsistenz bezeichnet hierbei die Sinnhaftigkeit eines Modells innerhalb eines einzigen Modells, wohingegen Intermodellkonsistenz die Korrektheit zwischen verschiedenen Modellen bezeichnet.[15] Diese Arten der Kontextbedingungen werden durch die Implementierung von Martin Schindler geprüft. Hierbei werden diejenigen Konsistenzbedingungen übernommen, die in dieser Arbeit empfohlen wurden und damit standardmäßig aktiviert sind. Dazu zählt beispielsweise die Eindeutigkeit von Klassen-, Schnittstellen- und Attributnamen innerhalb eines Modells, was auch im Fall von MontiWIS sehr sinnvoll ist.

6.3.2. Aufbau der Zwischenstruktur

In diesem Abschnitt wird der Aufbau der Zwischenstruktur für Klassendiagramme näher erläutert. Im Wesentlichen orientiert sich die Zwischenstruktur an der Struktur, wie sie sowohl im Klassendiagramm vorliegt als auch im generierten System vorhanden sein wird. Der Aufbau der Zwischenstruktur ist im Klassendiagramm in Abbildung 6.13 dargestellt.

Klasseninformationen in der Klasse Entity

Ein Objekt der Klasse `Entity` wird pro Klasse erzeugt, die im Klassendiagramm vorhanden ist. Daher sind in ihr auch Informationen gespeichert, die direkt aus dem Klassendiagramm abgeleitet werden oder auch in späteren Schritten berechnet wurden. Direkt in einem `Entity` wird beispielsweise gespeichert, welchen Namen die Klasse besitzt (Attribut `name` in Abbildung 6.13), in welchem Paket sie deklariert wurde (`packageName`), sowie in welchem Klassendiagramms (`classDiagramName`) sie definiert wurde. All diese Informationen sind bei der Generierung wichtig, aber auch im Verlauf des Generierungs- und Rekonstruktionsprozesses sind sie von Belang.

Neben diesen Namensinformationen besteht weiterhin die Möglichkeit, die Namen der standardmäßig generierten Seiten anzugeben, die sich aus dem Klassennamen ergeben. Diese werden im Rahmen des Rekonstruktionsprozesses gesetzt (stellvertretend für weitere Seitennamen repräsentiert über das Attribut `listPageName` in Abbildung 6.13).

[15] In der Arbeit von Martin Schindler wird zusätzlich unterschieden, ob die Modelle aus der gleichen Sprache stammen. Da in MontiWIS jedoch ausschließlich Klassendiagramme und keine anderen der dort spezifizierten Sprachen verwendet werden, findet diese Unterscheidung zumindest in Bezug auf die Klassendiagramme keine Anwendung. In anderen Sprachen werden jedoch sehr wohl Verweise auf Klassen verwendet. Diese werden in den späteren Abschnitten 7.3.1, 8.4.1 sowie 9.3.1 entsprechend erläutert.

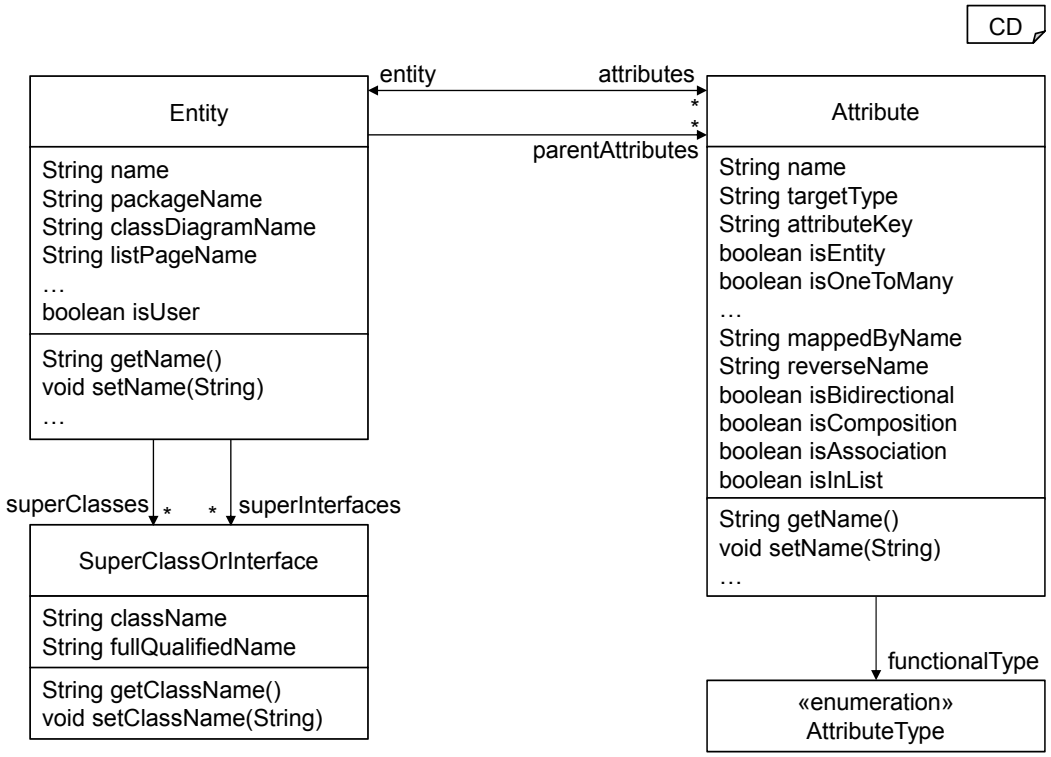


Abbildung 6.13.: Zwischenstruktur für Klassendiagramme

Ferner existiert ein boolesches Feld, welches angibt, ob das Entity eine Klasse repräsentiert, die als Nutzerklasse im generierten System fungieren soll (`isUser`). Dieses wird im Laufe des Rekonstruktions- und Generierungsprozesses benötigt, um entsprechende Umformungen vorzunehmen bzw. bestimmtes Verhalten zu ergänzen.

Schließlich besitzt die Klasse `Entity` Attribute, die aus den Assoziationen in Abbildung 6.13 hergeleitet sind. Die Klasse `SuperClassOrInterface` dient zur Speicherung von Informationen über Elternklassen oder Schnittstellen, die ein `Entity` besitzt. Das Attribut `superClasses` beinhaltet dabei Objekte, die die Superklassen repräsentieren und das Attribut `superInterfaces` enthält ebensolche Objekte, die jedoch Informationen über implementierte Schnittstellen enthalten. Hier wurde in beiden Fällen entschieden, dass eine Menge angegeben werden kann, obwohl in der aktuellen Zielsprache Java lediglich eine Superklasse pro Klasse erlaubt ist.

Weitere Attribute, die sich aus den im Klassendiagramm dargestellten Assoziationen ergeben, ist zum einen die Menge von `Attribute`-Objekten, die im Attribut `attributes` abgelegt sind. Diese stellen die Attribute der Klasse dar, die im Klassendiagramm spezifiziert wurden. Hierbei werden sowohl Attribute enthalten sein, die sich aus vordefinierten Datentypen herleiten, als auch Attribute, die aus Assoziationen abgeleitet werden. Ebenfalls Attribute enthält die Menge `parentAttributes`, wobei diese jedoch nicht direkt im Klassendiagramm definiert werden, sondern im Laufe des Rekonstruktionsprozesses hinzugefügt werden. Hierfür existiert

ein gesondertes Attribut, da nicht in jedem Fall alle Attribute gleich behandelt werden. Bei der Generierung von Seiten beispielsweise sind sowohl die eigenen als auch die Elternattribute von Bedeutung, wohingegen in der generierten Datenstruktur die Elternattribute durch die Vererbungsbeziehung zur Verfügung stehen.

Für all diese Attribute existieren in der Klasse `Entity` gewöhnliche getter-, setter- und adder-Methoden sowie einige Funktionen für eine komfortablere Benutzung. Stellvertretend für die get- und set- Methoden sind `getName` und `setName` dargestellt.

Speicherung der Attribute

Neben der Klasse `Entity`, die zur Speicherung sämtlicher Informationen dient, die direkt die Klassen des Domänendatenmodells betreffen, ist die Klasse `Attribute` zentraler Bestandteil der Zwischenstruktur für Klassendiagramme. Sie repräsentiert ein Attribut einer Klasse und enthält die Informationen, die zur komfortablen Verarbeitung der Attribute notwendig sind.

Zunächst enthält die Klasse `Attribute` Attribute, die Informationen beinhalten wie den Attributnamen (Attribut `name` in Abbildung 6.13), den hinsichtlich seiner Funktion bestimmten Datentyp (z. B. `Email` oder auch `Password`, repräsentiert durch das Attribut `functionalType`), den Zieldatentyp (vollqualifiziert, z. B. `java.lang.String`, durch das Attribut `targetType` abgebildet) oder auch den Schlüssel, der für die Ersetzung der Standardbezeichnung in der Oberfläche verwendet wird (`attributeKey`). All diese Attribute sind als einfache `Strings` realisiert.

Da ein `Attribute` sowohl ein einfaches Attribut sein kann, das durch einen vordefinierten Datentyp modelliert wurde, aber auch aus einer Assoziation zu einer anderen Domänenklasse entstanden sein kann, muss an geeigneter Stelle gespeichert sein, zu welcher Art das jeweilige Objekt gehört. Hierfür existiert das boolesche Feld `isEntity`, welches eben diesen Fakt speichert. Zusätzlich kann im Falle, dass es sich um ein Assoziations-Attribut handelt, das referenzierte `Entity` im Attribut abgelegt werden (Assoziation `entity`). Informationen daraus werden im weiteren Verlauf der Rekonstruktion und Generierung benötigt.

Da erklärtes Ziel der Zwischenstruktur ist, Informationen für die Templates in möglichst aufgearbeiteter Form bereitzustellen und sich damit sehr eng an der Zielinfrastruktur zu bewegen (vgl. hierzu Abschnitt 4.4), werden Informationen darüber, welche Annotationen der JPA [Ora13a, KS09] mit welchen Parametern innerhalb der generierten Klassen zu erzeugen sind, bereits in der `Attribute`-Klasse gespeichert. Wie in Abschnitt 6.3.3 beschrieben wird, werden diese Informationen bereits während der Visitor-Bearbeitung gesetzt. Dies hätte prinzipiell auch in einem der nachgelagerten Schritte wie beispielsweise der Rekonstruktion oder erst direkt während der Generierung des Zielsystems geschehen können, jedoch drücken die booleschen Attribute für die jeweilige Annotation sehr gut aus, welche Art der Relation über das Attribut repräsentiert wird (stellvertretend für weitere Attribute hier lediglich `isOneToMany`).

Neben den Annotationen in einem Attribut ist im generierten System ebenfalls von Bedeutung, wie das Attribut der Gegenseite gestaltet ist (beispielsweise durch den Parameter `mappedBy`). Auch diese Information wird in der `Attribute`-Klasse gespeichert (`mappedByName` bzw. `reverseName`). Die Belegung dieser Attribute erfolgt ebenfalls bereits während der Be-

arbeitung durch den Visitor. Zusätzlich wird in einer booleschen Variablen gespeichert, ob die Assoziation bidirektional ist (`isBidirectional`).

Wie in Abschnitt 6.2.2 beschrieben, unterstützt MontiWIS die beiden in den UML/P-Klassendiagrammen vorhandenen Relationen Assoziation und Komposition. Da es sowohl in der generierten Datenklasse als auch in der generierten Oberfläche einen erheblichen Unterschied macht, welche dieser Assoziationsarten ein Attribut repräsentiert, sind in der Klasse `Attribute` je ein boolesches Attribut dafür vorgesehen (`isAssociation` bzw. `isComposition`).

In Abschnitt 6.2.4 wird der Stereotyp «`inList`» erwähnt, der die Anzeige bestimmter Attribute in bestimmten Listenstrukturen zur Folge hat. Diese Eigenschaft wird ebenfalls über ein boolesches Feld abgebildet (`isInList`).

Neben diesen Attributen und den zugehörigen getter-, setter- und adder-Methoden existieren in der Klasse `Attribute` einige weitere Funktionen, die hilfreich bei der Verarbeitung der Daten sowohl in späteren Bearbeitungsschritten als auch den Templates zur Generierung sind. Stellvertretend dafür sind lediglich `getName` und `setName` in Abbildung 6.13 dargestellt.

Speicherung weiterer Informationen aus den Klassendiagrammen

Die Klasse `SuperClassOrInterface` dienst als Containerklasse, um Informationen über Vererbungsstrukturen innerhalb der Klassen abbilden zu können. Eine Unterscheidung, ob es sich bei dem Objekt um eine Klasse oder ein Interface handelt, wird hier nicht gespeichert, sondern lediglich dadurch bestimmt, in welcher Verbindung das jeweilige Objekt der Klasse `Entity` zugeordnet wurde. Dies genügt für eine korrekte Generierung. Der Name der Klasse oder des Interfaces wird in zweierlei Form (lediglich der Klassenname und in vollqualifizierter Form) in den Attributen `className` und `fullQualifiedName` gespeichert. Diese beiden Angaben werden im nachfolgenden Generierungsprozess verwendet, um die entsprechenden `Entity`-Objekte zu erhalten und weiter zu verarbeiten.

Die letzte Klasse, die zur Zwischenstruktur der Klassendiagramme zählt, ist der Aufzählungstyp `AttributeType`. Dieser wird innerhalb eines `Attribute` verwendet um zu spezifizieren, welchen funktionalen Typ ein Attribut hat, also ob es sich etwa um eine E-Mail-Adresse mit entsprechenden Konsistenzüberprüfungen oder um ein Textfeld zur Eingabe langer Texte handelt.

Zwischstrukturspeicher in CDGlobalStore

Mehrfach erwähnt und in Abschnitt 4.4 erläutert wurde bereits der Ansatz, zentrale Container für die Ablage der Informationen für die Zwischenstruktur zu verwenden. Auch im Falle der Klassendiagramme existiert ein solcher. Die Klasse `CDGlobalStore` definiert die Schnittstelle für den Zugriff und die Benutzung dieses Containers, implementiert wird er durch die Klasse `DefaultCDGlobalStore`. Sowohl Schnittstelle als auch implementierende Klasse sind in Abbildung 6.14 dargestellt.

Primäre Aufgabe eines `CDGlobalStore`s ist es, `Entity`-Objekte, die bei der Bearbeitung der Klassendiagramm-Modelle entstehen, zwischenzuspeichern und komfortablen Zugriff darauf zu ermöglichen. Daher bietet die Schnittstelle Methoden, um `Entity`-Objekte zum

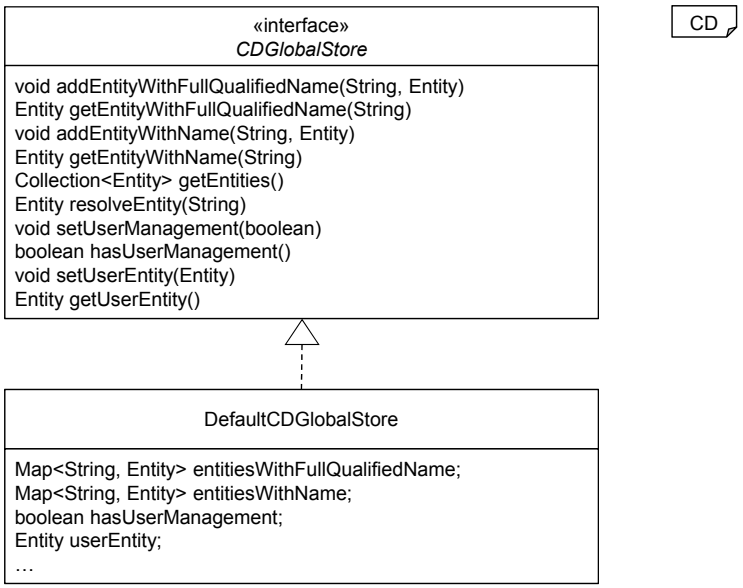


Abbildung 6.14.: Zentraler Speicher für Datenstrukturinformationen

Speicher hinzuzufügen und diese wieder daraus abzurufen. Da im nachgelagerten Rekonstruktions- und Generierungsprozess über unterschiedliche Schlüssel auf die Entity-Objekte zugegriffen wird, erfolgt auch das Ablegen in verschiedener Art. So werden die Objekte einerseits unter ihrem einfachen Namen gespeichert (repräsentiert durch die Funktionen `add-` bzw. `setEntityWithName` in Abbildung 6.14), andererseits aber ebenfalls unter ihrem vollqualifizierten Namen (der aus Paket-, Klassendiagramm- und Klassennamen besteht) abgelegt (`get-` bzw. `addEntityWithFullQualifiedName`).

Da sowohl im Rekonstruktions- als auch im Generierungsablauf die Notwendigkeit besteht, über sämtliche Entity-Objekte zu iterieren, bietet der zentrale Speicher die Möglichkeit, eine Sammlung aller Objekte abzurufen (`getEntities`). Diese Menge wird erzeugt aus der Vereinigung sämtlicher Objekte, die zum Speicher hinzugefügt wurden, also sowohl derer, die per Namen als auch die per vollqualifiziertem Namen identifizierten Objekte. Dies sollten jedoch in der Regel die selben sein.

Für eine komfortablere und einfachere Art des Zugriffs auf die gespeicherten Objekte existiert die Methode `resolveEntity`, die als Parameter eine Zeichenkette erhält. Diese dient für einen einheitlichen Zugriff auf die `Entity`-Objekte, da der Parameter sowohl einfacher als auch vollqualifizierter Name sein kann.

Zusätzlich dient der zentrale Speicher zusätzlich zum Ablegen der Information, ob innerhalb der Applikation Nutzermanagement und Authentifizierungsmethoden mit generiert werden sollen. Die Speicherung erfolgt im boolesche Attribut (`hasUserManagement`).[16]

[16] Auch wenn die Existenz eines Nutzermanagements logisch eher zum Speicher für Informationen über die Applikation in ihrer Gesamtheit (`ApplicationGlobalStore`) gehört, wird sie hier gespeichert, da sie aus der Existenz des Stereotypen im Klassendiagramm abgeleitet wird und auch, wie nachfolgend beschrieben, zur separaten Ablage weiterer Informationen führt.

Für den Fall, dass ein Nutzermanagement zur Verfügung stehen soll, lässt sich das `Entity` der Domänenklasse, die einen Nutzer repräsentiert, separat im zentralen Speicher ablegen (`userEntity`). Darüber ist es an geeigneter Stelle im Generierungsprozess möglich, direkt auf die Informationen zur Nutzerklasse zuzugreifen, ohne sämtliche `Entity`-Objekte durchsuchen zu müssen. Das Setzen dieses Attributs ebenso wie des soeben erwähnten booleschen Attributs geschieht über zugehörige `get`- und `set`-Methoden.

6.3.3. Modellverarbeitung durch einen Visitor

Die Bearbeitung von Modellen erfolgt, wie in MontiCore üblich, unter Verwendung von Visitoren [GHJV95], die jedes relevante Modellelement bearbeiten, dabei Transformationen ausführen und, wie im Fall von MontiWIS, eine Zwischenstruktur aufbauen, die zur Weiterverarbeitung der Modellinformationen dient. Das Vorgehen bei Klassendiagrammen bildet hierbei keine Ausnahme. Nachfolgend werden die relevanten Schritte des Visitors beschrieben. Dabei wird nicht auf jedes Detail, sondern nur generell auf den Ablauf der Bearbeitung eingegangen, wie er in der Klasse `CDTransformatorConcreteVisitor` umgesetzt ist. Da nicht jedes in den UML/P Klassendiagrammen vorhandene Element im Kontext von MontiWIS relevant ist, erfolgt die Erläuterung anhand der AST-Klassen, die vom Visitor bearbeitet werden.

Ebenso wie in der überwiegenden Anzahl der MontiWIS-Visitoren werden auch bei der Bearbeitung der Klassendiagramme `ownVisit`-Methoden verwendet. Darüber ist es möglich, die Reihenfolge der Verarbeitung eines Elements explizit festzulegen, wovon in diesem Fall auch Gebrauch gemacht wird. An entsprechender Stelle wird auf dieses Vorgehen und seine Sinnhaftigkeit hingewiesen.

Auf die explizite Darstellung der Klasse `CDTransformatorConcreteVisitor` in Form eines Klassendiagramms wird an dieser Stelle bewusst verzichtet, da sich deren Aufbau im Wesentlichen aus `ownVisit`-Methoden für die einzelnen AST-Elemente zusammensetzt.

Verarbeitung von ASTCDDefinition

Die Verarbeitung von Klassendiagrammen beginnt, wie innerhalb der Klassendiagramm-Grammatik festgelegt, mit der Bearbeitung des Wurzelelements der AST-Struktur (`ASTCDDefinition`). Innerhalb der `visit`-Methode wird zunächst der Name des Klassendiagramms für den Zeitraum der Verarbeitung gespeichert. Dieser wird an späterer Stelle benötigt.

Im nächsten Schritt wird geprüft, ob das aktuell verarbeitete Klassendiagramm die Klassen der vordefinierten Datentypen enthält (siehe hierzu auch den entsprechenden Unterabschnitt in 6.2.4), da aus diesen keine Zwischenstruktur zur weiteren Verarbeitung aufgebaut wird. Bei allen anderen Klassendiagrammen werden anschließend `ASTCDClass` und `ASTCDAssociation`-Klassen besucht und verarbeitet. Innerhalb des Visitors kommen nur die `ownVisit`-Methoden zum Einsatz und nicht die standardmäßigen `visit`-Methoden zum Besuchen sämtlicher AST-Elemente nacheinander.

Verarbeitung von ASTCDClass

Der zentrale Betrachtungsgegenstand innerhalb der UML/P-Klassendiagramme sind sicherlich die Klassen. Diese werden auf AST-Ebene repräsentiert über die Klasse `ASTCDClass`. Zu Beginn der Bearbeitung eines `ASTCDClass`-Objekts wird zunächst ein Objekt der Klasse `Entity` der Zwischenstruktur (siehe 6.3.2) angelegt und innerhalb des Visitors zwischengespeichert, so dass weitere nachgelagerte Schritte wie das Bearbeiten der Klassenattribute auf das aktuell bearbeitete `Entity`-Objekt zugreifen können. Darin werden zu Beginn einige Attribute wie beispielsweise Klassen-, Paket- und Klassendiagrammname gesetzt. Innerhalb der Zwischenstruktur wird exakt der Name der Klasse übernommen. Da die Klassennamen bereits im Modell eindeutig sein müssen, ist es sinnvoll, sie auch während des Generierungsprozesses beizubehalten.

Im nächsten Schritt werden die Superklassen und implementierte Schnittstellen der aktuell bearbeiteten Klasse ermittelt. Im aktuellen Arbeitsschritt wird lediglich der Superklassen- bzw. Schnittstellenname im `Entity`-Objekt gespeichert. Weitere Schritte wie etwa Vererbung potenzieller Assoziationen erfolgen an dieser Stelle noch nicht. Ebenso werden noch keine potenziellen Attribute der Elternklasse bearbeitet. Problematisch hier wäre, dass die Elternklasse möglicherweise noch gar nicht durch den Visitor bearbeitet wurde und daher noch keine Aussage über die Struktur etwaiger Elternklassen getroffen werden kann. Daher werden diese Rekonstruktionen erst zu einem späteren Zeitpunkt vorgenommen, was detaillierter in Abschnitt 6.3.4 beschrieben wird.

Anschließend wird geprüft, ob die aktuell bearbeitete Klasse als Nutzerklasse verwendet werden soll, was, wie in Abschnitt 6.2.4 erläutert, durch den Stereotyp «`user`» angezeigt wird. Ist die aktuelle Klasse derartig ausgezeichnet, wird sowohl im zugehörigen `Entity`-Objekt als auch im `CDGlobalStore` gespeichert, dass sie einen Nutzer repräsentiert bzw. dass ein Nutzermanagement im Web-Informationssystem gewünscht ist. Um im generierten System als Nutzerklasse zu fungieren, ist es notwendig, dass sie von der Klasse `de.montiwis.workflowengine.elements.WorkflowUser` erbt. Diese Klasse ist Teil der MontiWIS-Infrastruktur. Für eine Erläuterung ihrer Funktion hinsichtlich der Authentifizierung innerhalb des Systems sei auf den Abschnitt 9.4.2 und für die Beschreibung der Verwendung dieser Klasse im Kontext von verteilten Workflows sei auf Abschnitt 8.5 verwiesen.

Nachdem die notwendigen Werte im `Entity`-Objekt gesetzt wurden, wird nun das `Entity`-Objekt dem zentralen Speicher der Zwischenstruktur hinzugefügt. Da während des Generierungsprozesses nicht nur der einfache Name eines `Entity`-Objekts, sondern zum Teil auch weitere Informationen wie Paket- und Klassendiagrammname einer Klasse zur Identifikation herangezogen werden, wird das `Entity`-Objekt unter mehreren Schlüsseln abgespeichert, die auch Klassen-, Klassendiagramm- und Paketnamen mit einbeziehen.

Als letzter Schritt bei der Bearbeitung eines `ASTCDClass`-Objekts wird die Verarbeitung weiterer darin referenzierter Elemente angestoßen. Im aktuellen Fall ist dies die Bearbeitung der Attribute der Klasse. Diese werden, wie in MontiCore üblich, über den Aufruf der `startVisit`-Methode auf der Liste der untergeordneten Attribute verarbeitet.

Verarbeitung von ASTCDAttribute

Nachdem die Verarbeitung der zu einer Klasse selbst gehörenden Informationen abgeschlossen ist, werden ihre Attribute durch den Visitor in Augenschein genommen. Attribute werden in der AST-Struktur über die Klasse `ASTCDAttribute` repräsentiert und werden nacheinander, wie im Folgenden beschrieben, behandelt.

Die Speicherung in der Zwischenstruktur erfolgt in einem `Attribute`-Objekt, das zu Beginn der Bearbeitung angelegt wird. Wie bereits bei der Verarbeitung der Klassen wird auch hier der Name des Attributs im weiteren Verlauf übernommen. Dies ist einerseits hinsichtlich einer späteren Wiedererkennung der Attribute sinnvoll, andererseits erfüllen die Namen von Attributen in Klassendiagrammen bereits Eigenschaften wie Eindeutigkeit innerhalb einer Klasse, die ebenfalls im generierten System erfüllt sein müssen.

Im nächsten Schritt wird im `Attribute`-Objekt der genutzte vordefinierte Datentyp gespeichert, was über einen Wert wie `String` oder `Email` geschieht. Diese Angabe wird jedoch nicht zwangsläufig im Zielsystem so übernommen, sondern üblicherweise auf einen vollqualifizierten Datentyp in der Zielsprache abgebildet, was in Abschnitt 6.2.3 beschrieben ist. Diese Information wird ebenfalls bereits durch den Visitor gesetzt. Ferner wird gespeichert, ob das Attribute im Modell mit dem Stereotyp «`inList`» ausgezeichnet ist (vergleiche hierzu Abschnitt 6.2.4).

Als weiterer Parameter im `Attribute`-Objekt wird der Schlüsselwert für das `ResourceBundle` gesetzt, unter dem die Beschriftung in externen Dateien abgelegt wird. Eine Anforderung für diesen Schlüssel ist, dass er global nicht mehrfach vorkommen darf, um eindeutig das Attribut identifizieren zu können. Da sowohl Klassen innerhalb des Klassendiagramms als auch Attribute innerhalb einer Klasse eindeutig sein müssen, wird ein Schlüssel entsprechend der Form *<Klassename>_<Attributname>* erzeugt und im `Attribute`-Objekt abgelegt. Eine Erläuterung, wie in MontiWIS mit Beschriftungen wie diesen umgegangen wird, findet sich in Abschnitt 10.4.1.

Nach erfolgreicher Belegung der Attribute im `Attribute`-Objekt wird das erzeugte `Attribute` noch zum letzten (und aufgrund der Reihenfolge bei den Visitoraufrufen auch korrekten) `Entity`-Objekt hinzugefügt. Als Schlüssel für dieses Attribut wird der Name verwendet, der innerhalb der Klasse und des `Entity`-Objekts eindeutig sein sollte.

Das soeben beschriebene Vorgehen wird für jedes deklarierte Attribut in einer Klasse ausgeführt. Damit sind bislang alle Attribute abgedeckt, die einen vordefinierten Datentyp besitzen und daher als Klassenattribut zu behandeln sind. Noch nicht erfasst sind jedoch Attribute, die sich aus den Assoziationen zwischen Domänenklassen ableiten. In einem UML/P Klassendiagramm werden solche Sachverhalte über Assoziationen definiert, deren Bearbeitung im Folgenden beschrieben wird.

Bearbeitung von ASTCDAssociation

Wie in Abschnitt 6.2.2 erläutert, existieren in den UML/P Klassendiagrammen zwei Arten von Assoziationen. Zum einen sind gewöhnliche Assoziationen zwischen zwei Klassen möglich, zum anderen ist auch die Deklaration einer Teil-Ganzes-Beziehung über die Verwendung

einer Komposition erlaubt. Beide Möglichkeiten werden innerhalb der Grammatik der Klassendiagramme über ein AST-Element des Typs `ASTCDAssociation` abgebildet.

Da aus einer Assoziation potenziell zwei Attribute resultieren, werden in einem ersten Schritt zwei Objekte der Klasse `Attribute` erzeugt. Da `ASTCDAssociations` ebenfalls per `ownVisit`-Methode aufgerufen werden und der Aufruf innerhalb von `ASTCDDefinition` nach der Bearbeitung der Klassen stattfindet, ist davon auszugehen, dass die Klassen, zwischen denen eine Assoziation definiert wird, bereits verarbeitet wurden und damit im zentralen Speicher die zugehörigen `Entity`-Objekte existieren. Aus diesem Grund ist es auch möglich, die `Entity`-Objekte aus dem zentralen Speicher abzurufen und die notwendigen Modifikationen daran durchzuführen. Hierfür werden die Namen der an der Assoziation beteiligten Klassen benötigt, über die aus dem `CDGlobalStore` im nächsten Schritt die entsprechenden `Entity`-Objekte abgerufen werden. Im `Attribute` wird das `Entity`-Objekt der Zielklasse sowie ihr Name gespeichert.

Da sowohl Assoziationen als auch Kompositionen über das AST-Element `ASTCDAssociation` abgebildet werden, wird nachfolgend ermittelt, um welche Art der Assoziation es sich aktuell handelt und das Ergebnis im `Attribute`-Objekt gespeichert.[17].

Im nächsten Schritt erfolgt die Festlegung des Attributnamens. Dieser kann auf mehrere Arten festgelegt werden. Gewöhnlich wird der Name des Attributs aus dem Rollennamen der Assoziation abgeleitet. Dieser Wert lässt sich direkt aus dem Modell ablesen, sofern er dort angegeben ist. Sollte kein Rollenname angegeben worden sein, wird der (mit erstem Buchstaben klein geschriebene) Name der referenzierten Klasse als Attributname verwendet.

Soll im generierten System eine andere Bezeichnung als der Rollen- bzw. Zielklassenname angezeigt werden, so ist dies über die Modifikation der `ResourceBundle`-Dateien möglich, wozu jedoch auch wie bei vordefinierten Attributen ein eindeutiger Schlüssel nötig ist. Dieser wird ebenfalls im Zuge der Bearbeitung der Assoziation erzeugt und setzt sich wie bei gewöhnlichen Attributen aus dem Klassen- und dem Attributnamen zusammen.

Die nächsten Schritte bei der Bearbeitung der Assoziationsattribute ergeben sich aus der Art der Assoziation. Hierbei wird beispielsweise unterschieden, ob es sich um eine uni- oder bidirektionale Assoziation handelt. Davon ist etwa abhängig, ob die Attribute zu beiden beteiligten `Entity`-Klassen oder nur einer hinzugefügt werden. Da in MontiWIS, wie in Abschnitt 5.2 beschrieben, JPA als Framework für objektrelationale Abbildung verwendet wird, orientieren sich die in den Attributen gesetzten Parameter explizit an diesem Framework und den dort verwendeten Annotationen. Auf eine detaillierte Beschreibung, welche Attribute im Einzelnen existieren und wie diese besetzt werden, wird an dieser Stelle verzichtet.

6.3.4. Rekonstruktion der Klassendiagramminformationen

Bereits in Abschnitt 4.5 wurde erwähnt, dass und warum in MontiWIS Rekonstruktoren eingesetzt werden, um die aus den Modellen extrahierten Informationen vor der Generierung der Applikation umstrukturieren zu können oder auch weitere Konstrukte aus den bereits vorliegenden Informationen zu erzeugen. Auch die Objekte der Zwischenstruktur, die aus

[17] Obwohl in der Grammatik prinzipiell vorgesehen, wird nicht geprüft, ob die Assoziation eine Aggregation ist, da dieses Konstrukt in MontiWIS nicht unterstützt wird (vgl. hierzu Abschnitt 6.2.2).

der Bearbeitung der Klassendiagramme entstanden sind, dienen als Eingangsdaten für einige solche Umformungen. Im Folgenden wird auf die verschiedenen Rekonstruktionen eingegangen, die auf der Zwischenstruktur der Klassendiagramme ausgeführt werden. Hierbei ist ausschlaggebend, welche Daten primär als Eingabe für den beschriebenen Rekonstruktor dienen, nicht jedoch, was aus ihnen erzeugt wird.

Hinzufügen von Elternattributen

Die nachfolgend beschriebene Rekonstruktion ist in der Klasse `de.montiwis.combined.reconstructor.EntityReconstructor` enthalten. Durch die Verwendung von Vererbungsbeziehungen in einem Klassendiagramm ist es notwendig, einige Informationen aus der Elternklasse an eine Kindklasse weiter zu reichen. Dies ist nicht zwangsläufig notwendig, wenn es einzig um die Weitergabe der Attribute an die Kindklasse ginge, was bereits durch die im generierten System verwendete Vererbungsbeziehung realisiert wird. Anders verhält es sich jedoch zum Beispiel bei der Generierung von Eingabemasken, in denen die Elternattribute nicht automatisch mit übernommen werden. Dazu müssen dem Generator explizit die Attribute vorliegen, die aus der Elternklasse an die Kindklasse vererbt werden.

Um Elternattribute hinzufügen zu können, werden die `Entity`-Objekte aus dem zentralen Zwischenstrukturspeicher abgerufen. Für jedes dieser Objekte wird geprüft, ob zu ihm Elternklassen gespeichert sind. Sollte dies der Fall sein, wird die identifizierte Superklasse per Namen aus dem zentralen Speicher abgerufen und ihre Attribute der Kindklasse hinzugefügt.

Wie bereits angedeutet, werden diese Attribute nicht den Standardattributen der Klasse hinzugefügt, sondern separat gespeichert. Dadurch kann beim Generierungsprozess an verschiedenen Stellen unterschiedlich damit umgegangen werden.

Umbenennung boolescher Attribute

In MontiWIS sollten boolesche Attribute einen Namen besitzen, der dem Schema `is...` folgt, d. h. mit „is“ beginnt, um dann mit einem Großbuchstaben und CamelCase-Schreibweise fortgesetzt zu werden. Um diese Konvention durchzusetzen, erfolgt eine Bearbeitung boolescher Attribute. Dies geschieht ebenfalls in der Klasse `de.montiwis.combined.reconstructor.EntityReconstructor`.

Dabei wird über sämtliche `Entity`-Objekte iteriert und innerhalb dieser wiederum über alle Attribute. Sollte eines davon ein boolescher Typ sein, erfolgt eine Ersetzung des Namens durch Voranstellen von `is`, sofern der Name nicht bereits mit „is“ beginnt, da anderenfalls eine Dopplung des Präfixes vorläge.

6.3.5. Erzeugung von Standardseiten für jede Klasse

Ein zentrales Konzept von MontiWIS ist es, für jede Klasse im Klassendiagramm vollständige CRUD-Funktionalität bereitzustellen. Dies umfasst zum einen sämtliche Infrastruktur für die Speicherung von Objekten einer Klasse, aber auch sämtliche Seiten für die Bearbeitung von

Objekten sowie die Möglichkeit, diese in der Applikation direkt zu verwenden. [18]

Für die erforderlichen Schritte ist die Klasse `de.montiwis.combined.reconstructor.DefaultPageBuilder` zuständig. Hierin werden für jedes `Entity`-Objekt im globalen Speicher dieselben Schritte durchgeführt, die nachfolgend erläutert werden. Der Name der letztlich erzeugten Seite ergibt sich aus dem Klassennamen plus dem Suffix, das in Tabelle 6.15 dargestellt ist. Dieser Name wird unter anderem zur Identifikation der generierten Seite verwendet.

Klassenname plus ...	Erläuterung
`View`	Enthält eine Seite für die Anzeige eines Objekts
`New`	Enthält eine Seite für das Neuanlegen eines Objekts
`Edit`	Enthält eine Seite für das Bearbeiten eines Objekts
`List`	Enthält eine Seite für das Auflisten aller Objekte

Tabelle 6.15.: Erzeugung von Seitennamen für Standardseiten

Bei der Beschreibung der Seiten wird lediglich auf die an dieser Stelle erzeugten Elemente eingegangen. Grundsätzlich lässt sich sagen, dass für Seiten, ähnlich wie für `Entity`-Objekte, ebenfalls Objekte in einer ähnlichen Zwischenstruktur erzeugt werden, die letztlich als Eingabe für die Templates zur Seitenerzeugung dienen. In diesem Fall repräsentiert die `Page`-Klasse eine Seite in der Zwischenstruktur, aus denen die notwendigen Artefakte im Zielsystem generiert werden. Einen Überblick über den Aufbau und die unterstützten Elemente von Seiten liefert Abschnitt 7.2. Ein Überblick über die Seiten im generierten System wurde in Abschnitt 5.3 gegeben. Eine weiterführende Beschreibung der Zwischenstruktur von Seiten findet sich in Abschnitt 7.3.2 und die Generierung der Seitenelemente wird in Abschnitt 7.3.5 beschrieben. Screenshots von solchen Standardseiten sind in Abbildung 6.21 zu finden.

Generierung von Seiten zum Anzeigen von Objekten

Die Standardseite zur Darstellung einer Klasse muss die Möglichkeit bieten, ein Objekt aufzunehmen und darzustellen, ohne dass daran Änderungen möglich sind.

Aus diesem Grund wird ein `Page`-Objekt angelegt, dessen Name sich analog der Darstellung in Tabelle 6.15 aus dem Klassennamen mit angehängtem Suffix `View` ergibt. Hier werden auch einige Attribute gesetzt, etwa, dass es sich um eine Standardseite handelt, die exakt zu einer Klasse gehört und um welche Klasse es sich dabei handelt. Ferner wird festgehalten, ob zu dieser Seite eine separate `PageEntity`-Klasse generiert werden muss, was in diesem Fall nicht notwendig ist, da einige `PageEntity`-Klassen auch seitenübergreifend verwendet

[18] Obwohl während dieses Bearbeitungsschrittes Seiten erzeugt werden, was per se Thema des nächsten Kapitels 7 ist, erfolgt eine kurze Beschreibung an dieser Stelle, da die ausschlaggebenden Informationen aus den Klassendiagrammen stammen.

werden können. Um der Seite ein Objekt der zugehörigen Klasse übergeben zu können, wird ein entsprechender Parameter zum `Page`-Objekt hinzugefügt.

Die eigentliche Struktur der Seite besteht aus einer Überschrift, die standardmäßig „Displaying“ gefolgt von dem Klassennamen lautet. Für die Darstellung des Objekts wird ein `AllAttributesElement` hinzugefügt, welches im späteren Verlauf der Rekonstruktion durch die einzelnen Attribute der Klasse ersetzt wird. Da es sich um eine Seite zur Anzeige handelt, wird bei diesem Element festgelegt, dass die Attribute in nicht-änderbarer Weise lediglich ausgegeben werden sollen.

Generierung von Seiten zum Erstellen oder Editieren

Zwei weitere Seiten, die für jede Klasse erstellt werden, dienen zum Anlegen eines neuen Objekts und zum Bearbeiten eines bereits existierenden. Diese ähneln sich hinsichtlich ihres Aufbaus und werden daher hier gemeinsam betrachtet, wobei auf die Unterschiede entsprechend eingegangen wird.

Auch für die beiden Seiten wird je ein `Page`-Objekt angelegt. Diese erhalten einen Namen, der sich entsprechend des Schemas aus Tabelle 6.15 herleitet. Der Seitenname für das Neuanlegen besteht aus dem Klassennamen sowie dem Suffix `New`, wohingegen das Suffix für Seiten zum Verändern von Objekten `Edit` ist. Bei beiden zugehörigen Zwischenstrukturobjekten wird ebenfalls eine Überschrift hinzugefügt, wobei bei der Seite zum Editieren die Überschrift „Editing a“ und bei der Seite zum Anlegen eines Objekts die Überschrift „Creating a new“ verwendet wird, jeweils gefolgt vom entsprechenden Klassennamen. Das nächste Element ist ebenso wie bei der erzeugten Anzeigeseite ein `AllAttributesElement`. Hier wird jedoch die Variante verwendet, die editierbare Eingabeelemente in der Seite erzeugt.

Einzig bei dem `Page`-Objekt, das die Seite zum Erstellen eines Objekts repräsentiert, wird ein Schalter gesetzt, der bewirkt, dass daraus eine `PageEntity`-Klasse generiert wird. Dies genügt insofern, als alle drei bislang beschriebenen Seiten von der Struktur her exakt ein Objekt einer Klasse darstellen und somit in ihrer Struktur der darzustellenden Elemente ebenfalls übereinstimmen.

Generierung von Seiten zum Auflisten von Objekten

Die bisher beschriebenen Seiten ähneln sich stark hinsichtlich ihres Aufbaus, da bei jeder Seite genau ein Objekt einer Klasse in unterschiedlicher Art und mit unterschiedlicher Funktionalität dargestellt wird. Zusätzlich zu diesen beschriebenen Seiten wird für jede Klasse eine Seite generiert, die sämtliche Objekte in einer Listenform darstellt und für jedes der dargestellten Objekte standardmäßige Aktionen wie Editieren, Anzeigen oder Löschen bereithält (unter der Voraussetzung, dass einem Nutzer entsprechende Rechte gewährt wurden).

Gemein ist der Listenseite mit den bislang beschriebenen Seiten, dass auch sie über einen Namen identifiziert wird, welcher sich in diesem Fall zusammensetzt aus dem Klassennamen mit angehängtem Suffix `List`. Auch diese Seite wird nicht dazu verwendet, ein `PageEntity` zu erzeugen, da das notwendige Verhalten bereits in der Klasse vorhanden ist, die zur `New`-Seite generiert wurde.

Da in der Listenansicht eine Menge von Objekten der zugehörigen Klasse dargestellt werden muss, ist auch für diese Seite notwendig, dass ein Eingabeparameter vorhanden ist. Dieser repräsentiert jedoch nicht wie bei den anderen Seiten ein einzelnes Objekt der Klasse, sondern eine Menge davon. Daher wird der Listenseite ein Parameter hinzugefügt, der allerdings mengenwertig Attribut ist, wodurch der Seite nach der Generierung eine Menge von Objekten übergeben werden kann.

Der Aufbau der Seite zur Listenansicht wird wie bei den anderen Seiten auch durch eine Überschrift eingeleitet, die im Standard „Displaying a list of“ mit jeweiligem Klassennamen lautet. Gefolgt wird diese Überschrift von einem `ListEntityElement`, das eine Listendarstellung repräsentiert. An dieser Stelle werden zusätzlich noch die Attribute festgelegt, die zur Anzeige in der Liste vorgesehen sind. Wie in Abschnitt 6.2.4 beschrieben, werden diese über einen Stereotyp (`«inList»`) deklariert.

Nach der Erläuterung, welche Rekonstruktionen auf den aus den Klassendiagrammen gewonnenen Zwischenstrukturobjekten durchgeführt werden, wird im folgenden Abschnitt darauf eingegangen, auf welche Weise welche Dateien daraus konkret im Zielsystem erzeugt werden.

6.3.6. Generator

In Abschnitt 4.6 wurde beschrieben, wie die eigentliche Generierung aus der Zwischenstruktur in MontiWIS abläuft und welche Technologien dabei zum Einsatz kommen. Im folgenden Abschnitt wird die Generierung der Artefakte erläutert, die direkt aus den Klassendiagrammen abgeleitet werden. Dabei werden sowohl programmatische Teile der Generierung als auch im Überblick die Schritte innerhalb der Template-Bearbeitung beschrieben. Die eigentlichen Teile der Applikation, die sich aus dem Datenmodell ergeben, werden näher in Abschnitt 6.4 beleuchtet.

Die Klasse `de.montiwis.cd.generator.CDGenerator` enthält den Code, der die Generierung der zum Datenmodell gehörigen Artefakte steuert. Sämtliche nachfolgend beschriebene Funktionalität ist dort angesiedelt. In dieser Klasse werden Templates aufgerufen, die die übergebenen Daten in Applikations- oder Konfigurationscode überführen. Die Templates und eine kurze Beschreibung dazu sind in Tabelle 6.16 angegeben. Die Templatenamen enthalten eine Pfadangabe, die relativ zum Verzeichnis `templates` im Paket `de.montiwis.cd.generator` zu sehen ist. Anders verhält es sich mit dem Template `java/ResourceFileForEntity.ftl`, welches Teil des Pakets `de.montiwis.combined.generator` ist. Dieses Template wird aufgerufen aus der Klasse `de.montiwis.combined.generator.CombinedGenerator`, da dort nicht ausschließlich Informationen aus dem Klassendiagramm, sondern zusätzlich Informationen aus der Applikationsbeschreibungssprache verarbeitet werden. Zur konsequenten Anwendung des Sicherheitsmechanismus werden darin ebenfalls die in der Applikationssprache deklarierten, erlaubten Rollen verwendet, um sie als Teil der Seite zu verarbeiten. Trotz dieses Umstands wird dieses Template an dieser Stelle mit aufgeführt, da es inhaltlich stark mit den Artefakten aus dem Klassendiagramm verzahnt ist.

Template-Datei	Erläuterung
`config/persistence.xml.ftl`	dient zur Generierung der Datenbank-Konfigurationsdatei
`config/init_db.sql.ftl`	dient zur Initialisierung der Datenbank
`java/Entity.ftl`	wird für die Generierung der `Entity`-Klassen verwendet
`java/EntityDAO.ftl`	dient zur Generierung der DAO-Klassen
`java/EntityRemoteDAO.ftl`	dient zur Generierung der zur DAO-Klasse gehörenden Remoteschnittstelle
`java/ResourceFileForEntity.ftl`	dient zur Generierung der Ressourcenklasse für die Bearbeitung von Entities; enthalten im Paket `de.montiwis.combined.generator`

Tabelle 6.16.: Die Template-Dateien für Klassendiagramme und ihre Bedeutung

Generierung der Datenbankkonfiguration

Eine Datei, für deren Generierung zwar nicht direkt Informationen aus dem Klassendiagramm-Modell notwendig sind, die aber dennoch ursächlich mit dem Datenmodell in Verbindung steht, ist die Datei `persistence.xml`. Sie dient zur generellen Konfiguration der Datenbankverbindung.

Das Template `config/persistence.xml.ftl` wird dazu verwendet, die entsprechende Konfigurationsdatei zu erstellen. Der Großteil der Datei ist statisch und nicht abhängig von Informationen, die für jedes Projekt spezifisch wären. Lediglich der *Projektname* wird benötigt und in der Klasse `CDGenerator` gesetzt. Dieser wird dazu verwendet, innerhalb der Konfigurationsdatei eine separate Datenbank für jedes Projekt zu definieren und die Verbindung dazu entsprechend zu spezifizieren. Dies umfasst sowohl die Konfiguration des Java Transaction API (JTA)-Zugriffs als auch die Konfiguration einer manuellen Verbindung, um beispielsweise die Datenbankinitialisierung und Belegung mit Standardwerten zu ermöglichen. Für die manuelle Konfiguration wird ebenfalls der Projektname als *Nutzername* und *Passwort* verwendet. Ferner wird darin unterschieden, welches Datenbankmanagementsystem in der Zielplattform zum Einsatz kommt. Dies wird in einer Konfigurationsdatei bestimmt (s. Abschnitt 10.1.2) und im Template berücksichtigt.

Generierung einer Datei zur Initialisierung der Datenbank

Neben der Konfiguration des Datenbankzugriffs selbst wird ebenfalls eine Datei erzeugt, die per Ant-Skript ausgeführt werden kann und Anweisungen enthält, um die Datenbank zu initialisieren. Die Erzeugung der Initialisierungsdatei erfolgt über das Template `config/init_db.sql.ftl`.

Über das Initialisierungsskript geschieht das Setup der funktionsfähigen Datenbank, das Anlegen eines Nutzers sowie das Setzen der korrekten Berechtigungen für den erzeugten Nutzer auf die erzeugte Datenbank.

Da ebenso wie bei der Datenbankkonfiguration im vorherigen Abschnitt üblicherweise nicht nur eine Datenbank auf einem Datenbankserver existiert, wird der Name von Datenbank und Datenbanknutzer ebenfalls aus dem *Projektnamen* abgeleitet. Dies ist auch der einzige Parameter, der über das Template verarbeitet wird. Der Rest der erzeugten Datei ist rein statisch und ändert sich in der Regel nicht.

Generierung der Entity-Klassen

Nach der Beschreibung der Konfigurations- und Skriptdateien, die zwar auch für das Funktionieren der Applikation notwendig sind, werden in den folgenden zwei Abschnitten die zentralen Artefakte beschrieben, die aus den UML/P Klassendiagrammen generiert werden. Im aktuellen Abschnitt liegt der Fokus auf den *Entity-Dateien*, die die eigentliche Domänendatenstruktur repräsentieren.

Für die Generierung von JPA Entity-Klassen wird über die `Entity`-Objekte aus der Zwischenstruktur iteriert. Für jedes Zwischenstrukturobjekt werden die Templates mit den notwendigen Informationen versorgt, so dass daraus die entsprechenden Klassen generiert werden können. Neben dem `Entity`-Objekt aus der Zwischenstruktur, das als primäre Eingabe dient, werden der Paketname sowie die Superklassen und -schnittstellen übergeben.

Für die Generierung wird das Template `java/Entity.ftl` verwendet. In diesem werden primär die Informationen des übergebenen `Entity`-Objekts aus der Zwischenstruktur verarbeitet. Da bereits während der Bearbeitung durch den Visitor und im Rekonstruktionsprozess Wert darauf gelegt wurde, dass die Generierung ohne komplexe Berechnungen in den Templates selbst auskommt, ist der Templateaufbau im Falle der Entity-Klassen relativ einfach. Dadurch kann direkt auf Klassen- und Attributnamen in qualifizierter und unqualifizierter Form zugegriffen werden. Ebenso können die JPA-Annotationen direkt aus der Zwischenstruktur abgeleitet werden. Hierfür stehen boolesche Funktionen zur Verfügung, über die die Existenz bestimmter Annotationen abgefragt werden kann. Durch die derartige Vorbereitung der Zwischenstruktur lassen sich die Kontrollstrukturen auf einfache Schleifen und Bedingungsabfragen beschränken.

Der Aufbau der generierten Entity-Klasse gestaltet sich ebenfalls relativ unspektakulär. Die Klasse leitet sich aus der entsprechenden `Entity`-Klasse ab, die übergebenen Superklassen und -schnittstellen werden direkt auf die entsprechenden Konstrukte in einer Klasse abgebildet. Aus `Attributes`-Objekten werden, wie zu erwarten, Klassenattribute generiert, ebenso wie entsprechende getter- und setter- bzw. adder-Methoden im Falle von mengenwertigen Attributen, die aus Assoziationen entstanden sind. Die JPA-Annotationen, die die Assoziationen zwischen Klassen näher spezifizieren, werden aus den zugehörigen Informationen der `Attribute`-Objekte abgeleitet und an korrekter Stelle im Template in Annotationen umgesetzt. Zusätzlich werden noch einige Methoden generiert, die hilfreich beim Umgang mit den JPA-Entities sind. Eine detailliertere Beschreibung der Implementierung der Entity-Klasse selbst wird in Abschnitt 6.4 vorgenommen.

Generierung der DAO-Klassen

Die zweite Art von Artefakten, die von zentraler Bedeutung für die Funktionsweise der MontiWIS Datenverarbeitung und -persistierung ist, sind die *DAO-Klassen* [ACM03], die eine Möglichkeit schaffen, gespeicherte Objekte einer Klasse zu laden, sie nach Verändern zu speichern, sie zu löschen und danach zu suchen. Die Generierung dieser Klassen soll im Folgenden kurz beschrieben werden. Detaillierter wird auf die Funktionsweise der DAO-Klassen in Abschnitt 6.4 eingegangen.

Zur Generierung einer DAO-Klasse wird das Template `java/EntityDAO.ftl` verwendet. Ebenso wie bei der Erzeugung der JPA-Entity-Klassen wird auch bei der Generierung der DAO-Klassen über sämtliche `Entity`-Objekte aus der Zwischenstruktur iteriert. Für jedes dieser Objekte wird die Generierung einer DAO-Klasse angestoßen. Dazu werden neben dem `Entity`-Objekt selbst zusätzlich noch der Paketname und der Paketname der generierten Entity-Klassen sowie Informationen über potenziell vorhandene Superklassen benötigt. Mit der Übergabe dieser Informationen an das Template wird für jedes `Entity`-Objekt eine entsprechende DAO-Klasse generiert.

Da die vorausgegangenen Bearbeitungsschritte in Visitor und Rekonstruktoren dieselben waren wie zur Generierung der JPA-Entities, stehen auch diesem Template weitgehend aufgearbeitete Informationen zur Verfügung. Daher ist auch hier die Logik innerhalb des Templates relativ einfach gehalten. Für jedes `Entity`-Objekt wird eine DAO-Klasse angelegt, deren Name sich aus dem Namen der Klasse mit angehängtem Suffix `DAO` ergibt. Sollte die Klasse aus einer Superklasse abgeleitet worden sein, so wird auch die DAO-Klasse von der der Superklasse vererbt. Darüber stehen beispielsweise Funktionen wie das Suchen über Attribute des Eltern-DAO ebenfalls im Kind-DAO zur Verfügung und müssen nicht erneut generiert werden.

Zusätzlich zur DAO-Klasse selbst wird über das Template `EntityRemoteDAO.ftl` die zugehörige Schnittstelle generiert. Diese Schnittstelle ist für das korrekte Abrufen der Klassen im Applikationsserver notwendig.

Die eigentliche DAO-Funktionalität wird über die generierten Methoden abgedeckt. Das *Speichern* eines Objekts geschieht über die Methode `save`, der ein Objekt der zu persistierenden JPA-Entity-Klasse übergeben wird. Die Logik dieser Methode ist insofern etwas komplexer, als die Relationen zu potenziell gelöschten Objekten und deren gegenseitigen Referenzen darin behandelt werden. Das *Laden* eines Objekts erfolgt über die Methode `getForId`, die als Parameter die `id` des gewünschten Objekts erhält. Zusätzlich wird die Methode `getAll` generiert, die die Menge sämtlicher Objekte einer Klasse zurückliefert. Um Objekte nicht nur aufgrund ihres eindeutigen Bezeichners finden zu können, sondern auch hinsichtlich anderer Attribute filtern zu können, wird je Attribut eine Methode generiert, die dies erlaubt. Der Name der Methode ergibt sich aus dem Präfix `getBy` gefolgt vom jeweiligen Attributnamen. Der Rückgabewert dieser Methode ist ebenfalls eine Menge der zum DAO gehörenden Klasse.

Generierung der Ressourcenklassen

Für die Bereitstellung der CRUD-Funktionalität ist es notwendig, dass für jede Klasse eine sog. *Ressourcenklasse* erzeugt wird, die zur Verarbeitung der Clientanfragen nach dem

REST-Architekturmuster dienen (vgl. hierzu Abschnitt 5.3.1). Für jede Klasse des Domänendatenmodells, die nicht Teil der Menge der vordefinierten Datentypen ist, wird eine solche Ressourcenklasse erzeugt. Generiert wird dabei eine Klasse, deren Name sich zusammensetzt aus dem Namen der behandelten Klasse sowie dem Suffix `Resource`. Für die Klasse `Person` wäre die zugehörige Ressourcenklasse folglich `PersonResource`. Hierzu wird das Template `ResourceFileForEntity.ftl` verwendet, welches durch durch die Klasse `de.montiwis.combined.generator.CombinedGenerator` mit den notwendigen Daten befüllt wird. Da für jede Klasse des Domänendatenmodells eine Ressourcenklasse erzeugt wird, wird in `CombinedGenerator` über sämtliche nicht vordefinierten Klassen des Domänendatenmodells iteriert. Dem Template wird das `Entity`-Objekt der Zwischenstruktur übergeben. Zusätzlich dazu wird ein Objekt des `ApplicationGlobalStore` zur Verfügung gestellt, über das die Zugriffsrechte für die Standardseiten ermittelt werden können und so direkt in der Ressourcenklasse überprüft werden kann, ob eine bestimmte Nutzerrolle Zugriff auf eine bestimmte Funktionalität hat. Daneben werden dem Template einige Daten zugänglich gemacht, die eine einfachere Generierung ermöglichen. Hierzu zählen beispielsweise die Namen der Seiten für Neuanlegen, Editieren oder Auflisten aller Objekte einer Klasse.

6.4. Implementierung im Zielsystem

Wie in Abschnitt 5.2 diskutiert, wird für die Umsetzung der Datenpersistenz des Systems JPA im Kontext einer JEE Architektur verwendet. In diesem Abschnitt wird nun etwas detaillierter auf die Implementierung eingegangen. Hierbei wird nicht jede Einzelheit beschrieben, sondern der Fokus auf Besonderheiten gelegt, die sich für die Funktionsweise von MontiWIS als essenziell erwiesen haben.

Zunächst wird im folgenden Abschnitt 6.4.1 auf die Aspekte des Systems eingegangen, die direkt oder indirekt mit der Definition der Domänendatenstruktur und ihrer Persistierung zu tun haben. Anschließend gibt Abschnitt 6.4.2 einen Überblick über die standardmäßig generierten Seiten und den Zugriff darauf.

6.4.1. Datenstruktur und Persistenzmechanismen

Im folgenden Abschnitt wird zunächst auf die Umsetzung der Datenstruktur selbst und anschließend auf die Implementierung der Datenzugriffsmechanismen über das DAO-Muster eingegangen.

JPA Entity-Klassen

Zentraler Bestandteil der JPA-Infrastruktur sind die Entity-Klassen, von denen für jede Klasse aus dem Domänendatenmodell eine Klasse generiert wird. Ihr Aufbau ist schematisch in Abbildung 6.17 dargestellt. Als Beispiel ist dort ein stark vereinfachter Ausschnitt der Klasse gezeigt, die aus der Klasse `Person` aus Abbildung 6.5 erzeugt wurde. Paketangaben und Import-Deklarationen wurden auch hier der Übersichtlichkeit halber weggelassen.

Java

```
1  @XmlRootElement
2  @Entity
3  @Table(name = "PERSON")
4  @NamedQueries({
5    // benannte Abfragen
6  })
7  public class Person
8  implements Serializable {
9
10   @Id
11   protected String id = UUID.randomUUID().toString();
12
13   // Attributfelder
14
15   @OneToMany
16   Set<Address> addresses = new HashSet<Address>();
17
18   // getter- / setter- / adder- / sowie Hilfsmethoden
19
20   @JsonIgnore
21   public Set<Address> getAddresses() {
22       return this.addresses;
23   }
24
25   @Override
26   public int hashCode() {
27     // hashCode über id-Attribut
28   }
29
30   @Override
31   public boolean equals(Object obj) {
32     // equals über id-Attribut
33   }
34
35   public void assignValuesOf(Person person) {
36     // Zuweisen der Werte der übergebenen Person
37   }
38
39
40 }
```

Abbildung 6.17.: Schematischer Aufbau einer generierten JPA-Klasse

Die Klasse `Person` selbst wird in Zeile 7 deklariert und implementiert die Schnittstelle `Serializable`, wie es notwendig für ein JPA-Entity ist. Über die Annotation `@Entity` (Zeile 2) wird angezeigt, dass es sich bei dieser Klasse um ein solches handelt. Die Annotation `@Table` (Zeile 3) und ihr Parameter gibt an, welchen Namen die Tabelle besitzt, in der die Objekte der Klasse abgelegt werden sollen. Obwohl JPA Tabellen selbstständig benennt im Falle, dass kein

expliziter Name spezifiziert wurde, wird während der Generierung ein solcher angegeben. Auf diese Weise ist der generierte Name unabhängig von einer JPA-Implementierung und in Konfigurationsdateien und Initialisierungsskripten kann sicher darauf verwiesen werden. Über die Annotation `@NamedQueries` lassen sich Abfragen spezifizieren, auf die aus den DAO-Klassen zugegriffen wird. In diesem Fall sind es Abfragen, über die nach bestimmten Attributen gesucht werden kann. Die Annotation `@XmlRootElement` (Zeile 1) stammt aus der API- Sammlung Java Architecture for XML Binding (JAXB) [Sun06], die es ermöglicht, aus der Klassenstruktur direkt eine XML-Struktur zu generieren. In MontiWIS wird diese Bibliothek jedoch dazu verwendet, JavaScript Object Notation (JSON)-Repräsentationen [JSO] eines Objekts zu erstellen, was über die gleichen Mechanismen geschieht. JSON-Repräsentationen werden benötigt, um per AJAX [Gar05] Objekte auf die Client-Seite zu transportieren, was z. B. beim Hinzufügen von Objekten bei der Verknüpfung von Assoziationen zum Einsatz kommt (eine Beschreibung dieses Vorgehens ist in Abschnitt 7.2.3 zu finden).

Für die Speicherung von Entity-Objekten in einer Datenbank ist es obligatorisch, dass eine Klasse einen *Primärschlüssel* besitzt. In JPA wird ein Attribut zur Verwendung als Schlüssel über die Annotation `@Id` (Zeile 10) deklariert. Obwohl prinzipiell möglich, wird in MontiWIS von einer automatischen Belegung des Attributs Abstand genommen. In einigen Fällen ist es notwendig, dass der Bezeichner bereits nach der Erzeugung eines Objekts vorhanden ist und nicht erst beim erstmaligen Abspeichern in der Datenbank belegt wird. Aus diesem Grund wird als Bezeichner ein Universally Unique Identifier (UUID) verwendet, der zwar theoretisch nicht garantiert, aber in der Praxis mit ziemlicher Sicherheit eindeutige Zeichenketten generiert. Der verwendete Aufruf dazu ist in Zeile 11 zu finden.

Der Kommentar *Attributfelder* in Zeile 13 steht stellvertretend für sämtliche Attribute, die sich aus den *vordefinierten Datentypen* (vgl. Abschnitt 6.2.3) ergeben. Die Umsetzung eines solchen Datentyps auf eine Klasse im generierten System ist in Tabelle 6.7 festgeschrieben und wird hier entsprechend verwendet. Da diese Attribute nach einfachem Schema als Klassenvariablen generiert werden, wird an dieser Stelle auf eine nähere Erläuterung verzichtet.

Der Quellcode in den beiden nachfolgenden Zeilen repräsentiert *Assoziationen* in einer generierten Klasse. Zusammengefasst werden die Objekte der Zielklasse über eine Menge (implementiert als `java.util.Set`), was Dopplungen und insbesondere eine bestimmte Reihenfolge der Zuordnung nicht zulässt (Zeile 16). Wie von JPA verlangt, wird die Art der Zuordnung hier über der Annotation `@OneToMany` festgelegt. Die Attribute innerhalb der Annotation unterscheiden sich je nach Anwendungsfall und sind hier der Übersichtlichkeit halber weggelassen. Je nach Art der Assoziation werden unterschiedliche Annotationen verwendet. Kardinalität wird dabei ebenso unterschieden wie die Richtung der Assoziation. Damit einhergehend lässt sich ebenfalls spezifizieren, welches Attribut der Klasse am anderen Assoziationsende auf das aktuelle Objekt referenziert. Auf die Einzelheiten, für welche Konstrukte im Klassendiagramm welche Annotationen mit welchen Attributen generiert werden, soll an dieser Stelle nicht näher eingegangen werden. Eine Beschreibung der Annotationen ist beispielsweise in [KS09] zu finden.

Wie im Kommentar in Zeile 18 angedeutet, werden zu jedem Attribut die üblichen Zugriffsmethoden generiert. Für vordefinierte Attribute sind dies die standardmäßigen `get`- und

`set`-Methoden, wohingegen für Assoziationen zusätzlich Methoden zum Hinzufügen und Entfernen von Objekten erzeugt werden. Ein Beispiel für eine `get`-Methode für Assoziationen ist in Zeilen 21-23 dargestellt. Diese Methode gibt die Menge an `Adress`-Objekten zurück, die der Person zugeordnet wurden. Bemerkenswert an dieser Methode ist die Annotation `@JsonIgnore`. Diese entstammt dem Kontext von JAXB und bewirkt, dass bei der Serialisierung eines Objekts in eine JSON-Struktur die so bezeichnete Assoziation nicht weiter verfolgt wird. Dies ist insbesondere bei bidirektionalen Assoziationen notwendig, da bei Standardverhalten die Verbindungen unendlich häufig verfolgt würden und somit im schlimmsten Fall durch Überlauf des Puffers das Programm zum Absturz bringen würden. Da bei der Serialisierung in der Regel ohnehin lediglich die Attributwerte eines Objekts von Interesse sind, ist das Verfolgen der Assoziation und die Serialisierung des referenzierten Objekts nicht notwendig.

Die Methoden `hashCode` (Zeilen 25-28) sowie `equals` (Zeilen 30-33) überschreiben die Standardmethoden aus `java.lang.Object`. Diese sind notwendig, da Objekte einer Klasse üblicherweise in Mengen abgelegt werden, die auf der jeweiligen Hash-Methode aufbauen. Die Standardimplementierung von `equals` prüft auf Objektidentität, in MontiWIS jedoch sind zwei Objekte auch logisch gleich, wenn sie nicht auf das gleiche Objekt referenzieren. Es ist hingegen ausreichend, wenn die Bezeichner zweier Klassen gleich sind. Dies entspricht auch der Implementierung sowohl der `equals` als auch der `hashCode` Methoden, die in der dargestellten Entity-Klasse generiert wurden. Hier wird, neben den Standardtests auf Gleichheit von z. B. Objekttyp, lediglich die Gleichheit des Attributs `id` vorausgesetzt bzw. auch darüber der Hashwert gebildet.

Die letzte Methode, die hier etwas näher beschrieben werden soll, ist die in Zeilen 35-37 angedeutete Methode `assignValuesOf`. Diese erhält ein Objekt der jeweiligen Klasse als Parameter und dient dazu, der aktuellen Klasse dessen Werte zuzuweisen. Hier wird insbesondere beachtet, ob ein Attribut oder auch eine Referenz auf ein oder mehrere andere Objekte berücksichtigt werden müssen. Dies ist der Fall, wenn sie zusammen auf einer Seite dargestellt wurden und somit potenzielle Änderungen an diesem Objekt relevant sind und entsprechend gespeichert werden müssen. Anderenfalls kann es durch das Konzept von Seiten und dem Anzeigen einer Teilmenge von Attribute durchaus vorkommen, dass nur einige der Attribute überhaupt belegt sein können, was dazu führen würde, dass einige Attribute des übergebenen Objekts mit `null` belegt wären. Für Attribute, die Assoziationen repräsentieren, ist dies nicht eindeutig. Hier kann ein übergebene leere Menge einerseits bedeuten, dass die jeweilige Assoziation nicht Teil einer Seite war. Andererseits wäre es aber auch durchaus denkbar, dass einfach keine Assoziation zwischen den Objekten erstellt wurde oder sämtliche vormals vorhandenen Assoziationen gelöscht wurden. Aus diesem Grund existieren boolesche Methoden, die anzeigen, ob eine Assoziation geändert wurde oder nicht. Dies ist ausschlaggebend dafür, ob die jeweilige Assoziation aktualisiert oder ob sie ignoriert wird.

Neben den hier angegebenen Methoden enthalten die generierten Entity-Klassen noch weitere Hilfsmethoden, die an dieser Stelle nicht näher erläutert wurden. Im nachfolgenden Abschnitt werden nun die *DAO*-Klassen erläutert, die ebenfalls direkt aus den `Entity`-Klassen der Zwischenstruktur generiert werden.

DAO Klassen

Wie bereits in Abschnitt 6.3.6 erwähnt, werden neben den eben beschriebenen JPA-Entity-Klassen ebenfalls DAO-Klassen je `Entity`-Klasse aus der Zwischenstruktur generiert. Diese korrespondieren wiederum mit einer Klasse im UML/P-Klassendiagramm. Im folgenden Beispiel handelt es sich um die Klassen, die aus der Klasse `Person` aus Abbildung 6.5 generiert wurden.

Pro Klasse wird sowohl eine Schnittstelle, wie sie in Abbildung 6.18 dargestellt ist, als auch eine Klasse, wie in Abbildung 6.19 gezeigt, generiert, die diese Schnittstelle implementiert. Diese Klassen werden nachfolgend kurz beschrieben.

Die Schnittstelle in Abbildung 6.18 dient dazu, die Funktionalität der DAO-Klasse auch extern, d. h. von außerhalb des Java EE Servers verfügbar zu machen. Damit können DAO-Klassen auch verwendet werden, um z. B. aus manuell geschriebenem Code die Datenbank abzufragen oder zu befüllen (etwa mit Daten aus einer bereits vorhandenen Datenbank).

Java

```
1 @Remote
2 public interface PersonRemoteDao {
3
4   String save(Person entity);
5
6   // weitere Methoden in der DAO-Klasse
7
8 }
```

Abbildung 6.18.: Remoteschnittstelle einer generierten DAO-Klasse

Der Name der Schnittstelle setzt sich zusammen aus dem Namen der Klasse (`Person` im Fall des Beispiels in Abbildung 6.18) sowie dem Suffix `RemoteDao` (Zeile 2). Die Annotation `@Remote` (Zeile 1) bewirkt, dass diese Schnittstelle und ihre Methoden nach außen verfügbar sind. Die Methode in Zeile 4 steht hier lediglich stellvertretend für weitere Funktionen, die im Kontext der implementierenden Klasse in Abbildung 6.19 und im Folgenden erläutert werden.

In der schematischen Darstellung der DAO-Klasse in Abbildung 6.19 sind sowohl Paket- als auch Importdeklarationen ausgelassen. Die Klassendeklaration in Zeile 2, die, wie bereits angesprochen, die Schnittstelle aus Abbildung 6.18 implementiert, wird von der Annotation `@Stateless` in Zeile 1 begleitet. Diese stammt aus dem Enterprise Java Bean (EJB)-Kontext und deutet an, dass die so gekennzeichnete Klasse eine Enterprise Bean ist, die keinen Zustand zwischen zwei verschiedenen Aufrufen speichert. Dies ist in einer DAO-Klasse nicht notwendig, da jeder Aufruf einer ihrer Methoden in sich abgeschlossen und nicht auf Informationen vorhergehender Aufrufe angewiesen ist. Mit der Implementierung der DAO-Klasse als Bean ist es möglich, komfortabel auf diese Funktionalität von anderer Stelle aus zuzugreifen, ohne dass Erzeugung des Objekts und Ressourcenzuordnung problematisch sind, da diese Aspekte standardmäßig vom Enterprise-Server übernommen werden.

Java

```
1 @Stateless
2 public class PersonDao implements PersonRemoteDao {
3
4   @PersistenceContext(unitName="MontiWISContainer")
5   private EntityManager em;
6
7   public PersonDao() {
8   }
9
10  public String save(Person entity) {
11    // ...
12  }
13
14  public boolean delete(String id) {
15    // ...
16  }
17
18  public Person getForId(String id) {
19    // ...
20  }
21
22  public Collection<Person> getAllPerson() {
23    // ...
24  }
25
26  public Collection<Person> getByUserName(java.lang.String userName) {
27    // ...
28  }
29
30  // weitere finder-Methoden pro Attribut
31 }
```

Abbildung 6.19.: Schematischer Aufbau einer generierten DAO-Klasse

Ebenfalls vom Enterprise-Server bereitgestellt wird der in Zeile 5 dargestellte `EntityManager`, dessen Parameter über die Annotation `@PersistenceContext` in Zeile 4 spezifiziert wird. Der Annotations-Parameter `unitName` legt fest, unter welchem Namen die Datenbankverbindung im Server konfiguriert wurde. Dessen Wert leitet sich aus dem Projektnamen ab und korrespondiert mit der Serverkonfiguration, die an anderer Stelle generiert wird. `EntityManager` sind zum Laden, Speichern von JPA-Entities oder auch Ausführen beliebiger Structured Query Language (SQL) Ausdrücke notwendig. Durch die hier verwendete Art des Injizierens des `EntityManagers` werden die Ressourcen und Zugriffe zur Datenbank ebenfalls über den Server verwaltet.

Das nächste Konstrukt in einer DAO-Klasse ist ein leerer, parameterloser Konstruktor, wie er per Konvention im Enterprise-Kontext für Beans gefordert wird. Dieser ist für das korrekte Injizieren eines Objekts dieser Klasse bzw. einen erfolgreichen Lookup notwendig.

Die erste dargestellte Methode, die tatsächlich Funktionalität bereitstellt, ist `save` in Zeilen 10-12. Sie erhält als Parameter ein Objekt der zum DAO gehörenden Klasse und speichert das übergebene Objekt in der Datenbank. Hierbei ist es unerheblich, ob es sich bei dem übergebenen Objekt um ein neu angelegtes handelt oder eines, das bereits in der Datenbank persistiert wurde. Sollte das Objekt bereits in der Datenbank existieren, so wird es zunächst geladen. Dies ist notwendig, um potenziell vorhandene Assoziationen auch in den referenzierten Objekten löschen zu können im Fall, dass das übergebene Objekt die Assoziation nicht mehr besitzt. Ferner wird dafür gesorgt, dass potenzielle Rückreferenzen auf das übergebene Objekt die aktuellen Werte besitzen um mögliches Überschreiben mit nicht mehr aktuellen Objekten zu verhindern. Nach der entsprechenden Bearbeitung des Objekts und seiner potenziellen Referenzen wird es per `merge`-Methode des `EntityManagers` gespeichert.

Die nächste aufgeführte Methode `delete` in Zeilen 14-16 dient zum Löschen eines Objektes. Als Parameter wird ihr der eindeutige Bezeichner des Objekts übergeben. Über diesen wird das entsprechende Objekt geladen und direkt per `EntityManager` aus der Datenbank entfernt.

Die Methode `getForId` in Zeilen 18-20 dient zum Laden eines einzelnen Objekts. Wie der Name vermuten lässt, wird sie mit dem Bezeichner des Objekts aufgerufen, das angefordert wird. In einem ersten Schritt wird dieses über den `EntityManager` geladen. In MontiWIS erfolgt die Verarbeitung von Objekten von der Datenbank *detached* und damit entkoppelt vom Persistenzkontext. Wäre dies nicht der Fall, würden sämtliche Änderungen an einem Objekt im Persistenzkontext direkt in einer Transaktion auf die Datenbank abgebildet werden. Dies ist in der Regel jedoch nicht gewünscht, da Änderungen zu dezidierten Zeitpunkten gespeichert werden sollen. Anderenfalls wäre beispielsweise die Bearbeitung eines Objekts über mehrere Seiten problematisch.

Über die Methode `getAll`, dargestellt in Zeilen 22-24, die keinen Parameter erfordert und als Rückgabewert eine Menge von Entity-Objekten liefert, können sämtliche Objekte eines Typs geladen werden. Hierfür wird im Falle des Beispiels eine Anfrage der Form `SELECT e FROM Person e` erzeugt, was eben sämtliche vorhandenen Objekte lädt.

Die letzte dargestellte Methode `getByUserName` in Zeilen 26-28 steht stellvertretend für alle Methoden, die zum Filtern bzw. Suchen nach einem bestimmten Attribut generiert werden. Im dargestellten Beispiel wurde für das Attribut `userName` eine Filtermethode generiert. Als Parameter wird hier der Nutzername erwartet, nach dem in der Funktion gesucht wird. Analog wird für jedes Attribut eine entsprechende Methode generiert, deren Parameter auch entsprechend des Attributdatentyps erzeugt wird. Darin wird je nach Attribut das entsprechende `NamedQuery` ausgeführt, das in der zugehörigen JPA Entity-Klasse generiert wurde. Gemein ist all den Methoden, dass die Menge der Objekte zurückgegeben wird, auf die das entsprechende übergebene Kriterium passt. Auch in jeder dieser Filtermethoden wird jedes gefundene Objekt vom Persistenzkontext entkoppelt, so dass problemlos und ohne Gefahr des versehentlichen Aktualisierens damit gearbeitet werden kann.

Wie soeben beschrieben, ist es möglich, die DAO-Klassen von extern aufzurufen. Innerhalb der Applikation können sie wie üblich über den Schnittellennamen (hier: `PersonRemoteDao`) und der `@EJB`-Annotation injiziert werden. Der Aufruf des DAOs von extern muss jedoch analog zu dem Code aus Abbildung 6.20 erfolgen.[19]

[19]Die Aufrufe können dabei geringfügig variieren, da der vollqualifizierte Name in Zeile 3 des Codeausschnitts

```java
InitialContext ctx = new InitialContext();
PersonRemoteDao dao = (PersonRemoteDao)
    ctx.lookup("java:global/TM/PersonDao!de.montiwis.dao.PersonRemoteDao");
```

Abbildung 6.20.: Aufruf einer DAO-Klasse

Nach der Erläuterung der JPA Entity- und DAO-Klassen, wird im folgenden Abschnitt auf die Standard-Seiten und die Mechanismen eingegangen, die zur Funktionsweise dieser notwendig sind.

6.4.2. Generierung der Standardseiten

In einem Web-Informationssystem, wie MontiWIS eines generiert, sind die bislang beschriebenen Artefakte nur sinnvoll, wenn es auch eine Möglichkeit gibt, Daten in der modellierten Struktur einfach und effizient manipulieren zu können. Dies umfasst Standardseiten für CRUD-Funktionalität sowie Klassen, die die Verarbeitung der Daten auf Serverseite übernehmen. Dies sind einerseits die `PageEntity` bzw. die `PageEntityInitializer`-Klassen, die zum Transportieren von Daten in eine Seite hinein und daraus hinaus dienen. Darüber hinaus werden weiterhin *Ressourcenklassen* benötigt, die Anfragen von einer Seite entgegennehmen und diese entsprechend verarbeiten. Sowohl diese Klassen als auch die Seiten werden in diesem Abschnitt erläutert.

Pages

Welche Seite aus einer Klasse des Klassendiagramms generiert werden, wurde bereits in Abschnitt 6.3.5 erläutert. Dort ist ebenfalls beschrieben, wie diese durch Bearbeitung der aus den Klassendiagrammen erzeugten Zwischenstruktur generiert wurde. Die einzelnen Elemente jeder Seite ergeben sich direkt aus dem Aufbau der Seite, wie sie ebenda zusammengesetzt wurde, weswegen an dieser Stelle nicht mehr auf jedes Detail eingegangen werden soll. Die Struktur der Seite in Zusammenhang mit der Erläuterung, wie diese Elemente in eine Seite umgesetzt werden, ist hinreichend für das Verständnis des Seitenaufbaus. Dennoch existieren für Seiten, die für das Bearbeiten von Objekten einer Klasse erzeugt wurden, einige Besonderheiten, auf die im Folgenden eingegangen wird.

Die Standardseiten, die generiert werden, sind in Screenshots in Abbildung 6.21 dargestellt. Diese sind *List*-Seiten (a), die zur Anzeige einer Liste aller Objekte eines Typs dienen, *New*-Seiten (b) zum Anlegen eines neuen Objekts, *Edit*-Seiten (c) zum Verändern eines Objekts sowie *View*-Seiten (d) zum Anzeigen eines Objekts. Diese Seiten, die bereits in Tabelle 6.15 und deren Kontext erwähnt wurden, finden sich ebenfalls im Statechart in Abbildung 6.22 als Zustände wieder. Dieses stellt die verschiedenen Möglichkeiten dar, über die ein Nutzer von

von Projekt zu Projekt unterschiedlich sein wird, da er den Projektnamen bzw. den Kontext, in dem die Anwendung installiert wurde, enthält.

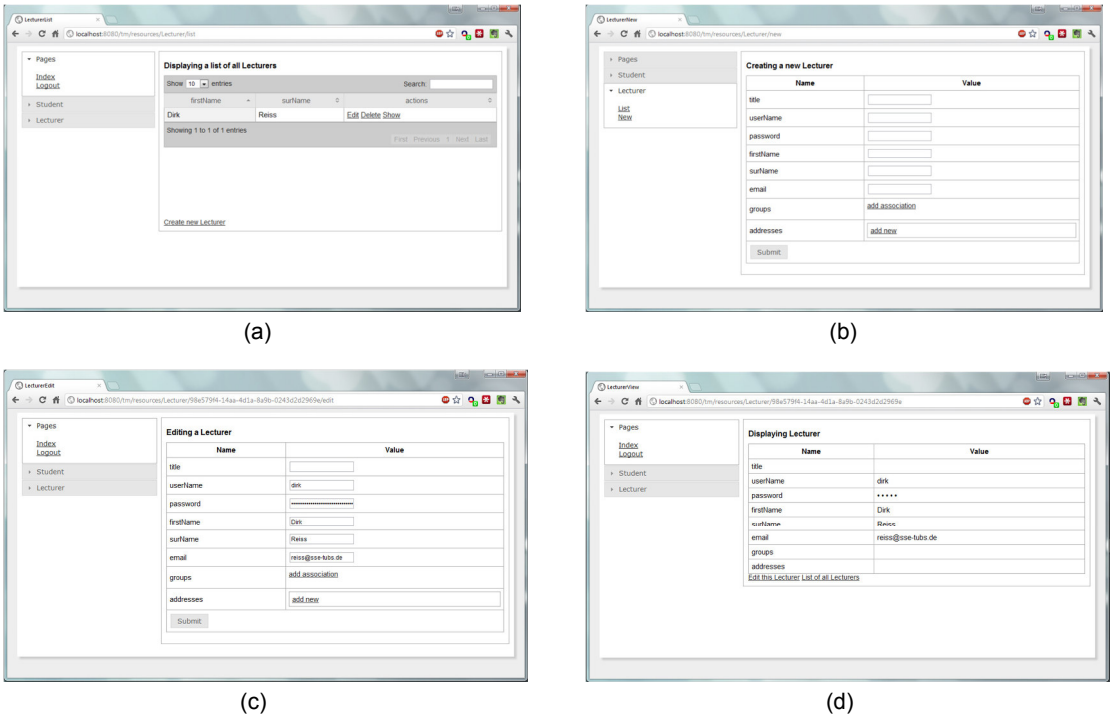


Abbildung 6.21.: Beispiel für Standardseiten

einer Seite zu einer anderen gelangen kann. Die Transitionen von einer Seite auf eine andere sind als Methodenaufruf wie beispielsweise `editLink()` repräsentiert, wobei die eigentliche Aktion durch einen Nutzer mit Klick auf einen Link aufgerufen wird. Anzumerken ist an dieser Stelle, dass die nachfolgend beschriebene Funktionalität durch Zugriffsbeschränkungen restringiert sein kann. Diese können je Klasse spezifiziert werden, so dass ein Nutzer unterschiedliche Berechtigungen auf Objekten unterschiedlicher Klassen besitzen kann. Restriktionen betreffen hier auch bereits die Darstellung von Links auf den Seiten und werden ebenfalls serverseitig geprüft. Eine Erläuterung findet im späteren Abschnitt 9.2.2 und dort insbesondere in Tabelle 9.5 statt.

Obwohl bis auf die `Edit`-Seite auch alle anderen Seiten als Startpunkt der Betrachtung dienen könnten, sei an dieser Stelle zunächst die `List`-Seite betrachtet. Auf ihr werden sämtliche vorhandenen Objekte einer Klasse aufgelistet, wobei hinter jedem Listeneintrag Links zum Anstoßen verschiedener Aktionen angezeigt werden. Zusätzlich dient ein weiterer Link zum Anlegen eines neuen Objekts. Wird dieser betätigt (`newLink()` im Statechart), so gelangt der Nutzer auf die `New`-Seite.

Von der `New`-Seite aus ist es lediglich möglich, durch Absenden des Formulars (`saveButton()`) das Speichern zu veranlassen. Nach Absenden des Formulars erfolgt eine abschließende serverseitige Validierung. Ist diese erfolgreich (Vorbedingung `[valid]`), wird auf die `View`-Seite weitergeleitet, im Falle eines Misserfolgs der Validierung (Vorbedingung `[invalid]`) wird der Nutzer zurück auf die `New`-Seite geleitet, wo erneut das gesendete aber

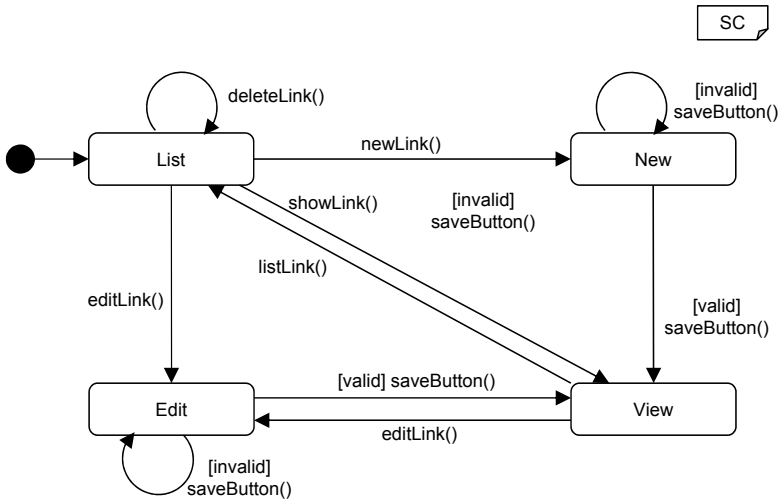


Abbildung 6.22.: Navigationsmöglichkeiten zwischen den Seiten

nicht korrekte Objekt angezeigt wird.

Auf der `View`-Seite wird ein Objekt angezeigt, ohne jedoch Möglichkeiten zum Ändern des Objekts direkt auf der Seite zu bieten. Um zur `Edit`-Seite zu gelangen, muss ein Link auf der Seite betätigt werden (`editLink()`). Zurück zur `List`-Seite gelangt der Nutzer ebenfalls über einen entsprechenden Verweis (`listLink()`).

Die `Edit`-Seite erlaubt selbst keine Navigation zu einer anderen Seite. Ebenso wie die `New`-Seite wird das dortige Formular nach Absenden (`saveButton()`) serverseitig validiert. Bei erfolgreicher Validierung (`[valid]`), wird der Nutzer zur `View`-Seite geleitet, auf der das gespeicherte Objekt mit all seinen Werte ausgegeben wird. Bei fehlgeschlagener Validierung (`[invalid]`) werden die fehlerhaften Daten wieder auf der `Edit`-Seite angezeigt und können erneut bearbeitet werden.

Eine weitere Besonderheit bei den Standardseiten, die zu jeder Klasse generiert werden, ist die Art, wie und wohin Anfragen gesendet werden. Hier bestehen Unterschiede zu Seiten, die als Teil eines Ablaufs oder eines Wizards aufgerufen werden. Worin diese Unterschiede bestehen und wie dies programmatisch umgesetzt wurde, erläutert der folgende Abschnitt.

Ressourcenklassen für Klassen und Zugriff über eine REST Schnittstelle

Wie in Abschnitt 5.3 erläutert, wird in MontiWIS weitestgehend dem REST-Architekturmuster [Fie00, FT02] gefolgt. Dieses Prinzip wird auch beim Zugriff auf Objekte einer bestimmten Klasse angewandt. Welche Mechanismen dabei zum Tragen kommen und wie diese umgesetzt wurden, wird im Folgenden dargelegt.

Da es in REST üblich ist, Objekte generell als *Ressource* zu behandeln, werden auch die Objekte der Klassen des Domänendatenmodells als solche betrachtet. Diese sollten eindeutig per URL identifizier- und adressierbar und über die Methoden des HTTP-Protokoll manipulierbar sein. Ein Beispiel für eine URL in MontiWIS, die die Ressource `Person` adressiert, ist die

Adresse `http://montiwis.de/tm/resources/Person/o2h92`. Über eine `GET`-Anfrage an diese Adresse wird das Objekt mit dem Bezeichner „o2h92“ aufgerufen, was dem REST-Paradigma entspricht. Tabelle 6.23 zeigt den *Pfad* in einer URL sowie die zugehörige HTTP-Methode, über die die URL aufgerufen wird und bietet eine kurze Erläuterung des entsprechenden Zugriffs. Der Pfad, der dort angegeben ist, muss als Anhang zu einer URL folgenden Schemas gesehen werden. Für den Fall der oben angegebenen Beispiel-URL ist `http://montiwis.de-/tm/resource/` der fixe Teil, der stets für jede Ressource gleich bleibt. `tm` ist dabei der Kontext, der sich aus dem Projektnamen ergibt, `resources` der Teil des Pfades, unter dem sich REST-Ressourcen ansprechen lassen. Der Pfadteil `Person` in diesem Beispiel ergibt sich aus dem Klassennamen, so dass dieser variabel für jede zu adressierende Klasse ist.

Pfad	HTTP-Methode	Besonderheit	Erläuterung
`{id}`	`GET`		Aufrufen des Objekts mit der übergebenen `id` über `View`-Seite
`/new`	`GET`		Aufrufen einer Seite zum Neuanlegen eines Objekts über `New`-Seite
`/list`	`GET`		Aufruf der Übersichtsseite über sämtliche Objekte über `List`-Seite
`/list`	`GET`	Ausgabeformat: JSON	Rückgabe der Liste sämtlicher Objekte im JSON Format
	`POST`		Anlegen eines neuen Objekts vom entsprechenden Typ
`{id}`	`PUT`		Aktualisieren des Objekts mit der übergebenen `id`

Tabelle 6.23.: Zugriff auf Objekte über REST-Schnittstelle

Die Angabe der `id` in Verbindung mit einer GET-Anfrage führt dazu, dass das Objekt mit dem übergebenen Bezeichner `id` zurückgeliefert wird. Erfolgt diese Anfrage durch einen Browser, so wird standardmäßig die zugehörige `View`-Seite mit den Werten des angefragten Objekts dargestellt. Änderungen an diesem Objekt werden durch diesen Aufruf nicht vorgenommen.

Auch der Aufruf des Pfads `/new` über die `GET`-Methode hat zunächst keine Auswirkungen auf eine Ressource des betreffenden Typs. Hierdurch wird die `New` Seite der Ressource aufgerufen, was durch einfache Rückgabe der JSP-Seite für neue Objekte geschieht.

Der Aufruf des Pfads `/list` hat in zwei Kontexten unterschiedliche Bedeutung und unterschiedliche Effekte. In beiden Fällen wird der Aufruf über die HTTP-Methode `GET` erwartet. Im gewöhnlichen Fall, d. h., wenn wie üblich ein `Response`-Objekt mit entsprechendem `Viewable`-Inhalt zurückgegeben wird, erfolgt die Darstellung der `List`-Seite. Auf dieser werden sämtliche Objekte des entsprechenden Typs in Listenform angezeigt. Im zweiten Fall jedoch, wenn clientseitig JSON-Inhalt angefordert wird, erfolgt die Rückgabe eines Ar-

rays vom jeweiligen Objekttyp. Durch eine spezielle Annotation in der Ressourcenklasse (`@Produces("application/json")`) wird ausgedrückt, dass bei Anforderung von JSON als Antwort die entsprechende Funktion aufgerufen werden soll. Diese Methode wird insbesondere bei der Zuordnung von Objekten per Assoziation über die Benutzerschnittstelle (vgl. den entsprechenden Abschnitt in 7.2.3) aufgerufen. Hierüber werden die JSON- Repräsentationen der referenzierten Klasse geliefert, die dann in der Auswahlliste angezeigt werden können. Die Menge sämtlicher Objekte wird per DAO-Klasse aus der Datenbank geladen und in ein Array umgewandelt. Die Serialisierung der Objekte ins JSON-Format erfolgt automatisch mit Hilfe der JAXB-Funktionalität, die in den JPA Entity-Klassen per Annotation verfügbar gemacht wurde.

Der Aufruf der oben beschriebenen URL ohne Parameter (also beispielsweise `http://montiwis.de/tm/resource/Person`) per HTTP `POST` führt zur Erzeugung eines neuen Objekts, in diesem Fall also einer neuen Person. Hierbei werden die übermittelten Formulardaten per `PageEntity`-Objekt in das entsprechende Objekt umgewandelt. Sollte dabei ein Fehler auftreten, beispielsweise ein Datum nicht im korrekten Format vorliegen, erfolgt die erneute Rückgabe der `New`-Seite, belegt mit den übermittelten inkorrekten Daten. Im Normalfall, d. h. wenn das Objekt über das `PageEntity`-Objekt erfolgreich mit den Werten aus dem Formular belegt werden konnte, erfolgt eine Weiterleitung per `Response.seeOther`-Methode, was einen HTTP-Status von 303 zur Folge hat. Das Ziel der Weiterleitung ist die URL für den entsprechenden Objekttyp mit angehängter `id` des soeben angelegten Objekts. Das Verhalten dieses Aufrufs ist weiter oben in diesem Abschnitt beschrieben.

Ein ähnliches Verhalten wie das Anlegen eines Objekts per `POST`-Methode tritt beim Aktualisieren eines Objekts per `PUT`-Methode auf. Wie in Abschnitt 5.3.1 erläutert, wird innerhalb der Ressourcenklasse zwar auf die HTTP-Methode geprüft und damit auch die zugehörige Funktion innerhalb der Ressourcenklasse ausgewählt, clientseitig wird jedoch aktuell noch ein zusätzliches Attribut im Formular übertragen und nach Abschicken des Formulars per Filter in die korrekte HTTP-Methode umgewandelt. Dies ist notwendig, da eine vollständige Browserunterstützung für andere HTTP-Methoden als `GET` und `POST` nicht zwangsläufig gegeben ist. Entsprechend des REST-Paradigmas erfolgt durch `PUT` an eine bestimmte Ressource das Ersetzen der adressierten Ressource mit den übermittelten Werten. Daher wird nach Absenden des Formulars und korrektem Aufruf der mit `PUT` annotierten Methode in der Ressourcenklasse zunächst das Laden des zu aktualisierenden Objekts angestoßen. Dieses Objekt wird per `id` Pfadparameter der Methode identifiziert. Die Werte des geladenen Objekts werden in einem `PageEntity`-Objekt initialisiert und anschließend mit den aktuellen Daten aus dem Formular belegt. Dieses zweistufige Vorgehen ist notwendig, um potenziell gelöschte Assoziationen in allen betreffenden Objekten löschen zu können. Sollte beim Belegen der Werte trotz clientseitiger Validierung ein Fehler auftreten, wird die `Edit`-Seite des jeweiligen Objekts, belegt mit den inkorrekten Werten, über den HTTP-Status von 400 (*bad request*) zurückgeliefert. Im Fall, dass sämtliche Werte korrekt waren, wird das entsprechende Objekt aus der `PageEntity` per DAO gespeichert und analog zum Speichern eines neuen Objekts per `seeOther`-Methode auf die URL zum Betrachten des Objekts umgeleitet.

PageEntities für Standardseiten

Neben den Ressourcenklassen werden selbstverständlich auch `PageEntity`-Klassen benötigt, um die abgesandten Daten aus den Formularen der Seiten entsprechend entgegen zu nehmen und serverseitig zu verarbeiten bzw. die Daten aus der Datenbank in der Seite verfügbar zu machen. Im Gegensatz zu den Ressourcenklassen, für die speziell für die Verarbeitung einzelner Entities generiert werden, entsprechen die PageEntity-Klassen exakt denen, die ohnehin für jede Seite generiert werden. Aus diesem Grund sei an dieser Stelle für eine Beschreibung ihres Aufbaus auf Abschnitt 7.4.3 verwiesen.

6.5. Andere Ansätze

In diesem Abschnitt wird beleuchtet, wie andere Ansätze mit dem Aspekt der Datenmodellierung bzw. mit der Implementierung von Datenbankzugriffen umgehen. Zunächst werden in Abschnitt 6.5.1 unterschiedliche Arten der Implementierung von Datenmodell und Persistenz vorgestellt und mit der Implementierung in MontiWIS verglichen. Eine Betrachtung anderer Ansätze der Datenmodellierung und insbesondere, wie andere modellbasierte Ansätze diesen Aspekt behandeln, wird anschließend in Abschnitt 6.5.2 vorgenommen.

6.5.1. Andere Ansätze der Implementierung

In diesem Abschnitt wird zunächst auf unterschiedliche Ansätze zur Implementierung des Datenbankzugriffs eingegangen. Nachfolgend werden kurz andere Methoden erläutert, die ebenfalls einen direkten Zugriff auf CRUD-Funktionalität für eine vorgegebene Datenstruktur bieten.

Umsetzung von Datenbankzugriffen

MontiWIS verwendet die JPA [Sun09b, KS09] zur Umsetzung der Datenmodellierung und Persistenz. Durch die Verwendung von objektrelationalem Mapping ist es nicht mehr notwendig, die Datenbankstruktur manuell (oder im Fall von generativer Softwarenentwicklung: manuell und explizit durch den Generator) anzulegen. Allein die Umsetzung der Datenstruktur (modelliert über Klassendiagramme) in die Klassenstruktur in Java genügt, um daraus über das Framework die Datenbankstruktur abzuleiten.

Der Zugriff auf die gespeicherten Daten wird in MontiWIS über das DAO-Muster [ACM03] realisiert. Darüber kann auch manuell auf die in der Datenbank gespeicherten Objekte zugegriffen werden. Direkter Zugriff über z. B. einen `EntityManager` wäre sicherlich ebenfalls möglich, ebenso beispielsweise die Generierung der Funktionalität im Sinne des *Active Record* Musters [Fow03]. Hierbei wäre die Funktionalität zum Datenbankzugriff (sowie weitere Geschäftslogik) direkt an der generierten Domänenklasse angesiedelt, was jedoch eine recht starke Kopplung von Logik und Datenhaltung bedingen würde, weswegen davon abgesehen wurde.

Eine Alternative zum Einsatz eines Frameworks wie JPA wäre natürlich, die Daten durch die direkte Verwendung von JDBC [Ora11], inklusive manueller Erzeugung der Datenbankstruk-

tur, sämtlicher Abfragen und Funktionalität. In [GHKV08] wird argumentiert, dass dies der Weg der größtmöglichen Flexibilität ist, der eingeschlagen werden sollte. Dies ist zweifelsohne richtig und auch in den Fällen vorteilhaft, in denen hochperformante, auf bestimmte Szenarien optimierte Anforderungen zu erfüllen sind. Diese Notwendigkeit stellte sich bei der Implementierung von MontiWIS jedoch noch nicht, so dass hier der Ansatz, die sinnvollen und für einen allgemeinen Fall stark optimierten Funktionalitäten des Frameworks zu nutzen, präferiert wurde.

Andere Möglichkeiten zur Erzeugung von CRUD-Funktionalität

Naked Objects [Paw04, PM02, PM01] stellen einen Ansatz dar, der auf die automatische Generierung von Benutzerschnittstellen auf Basis von Reflection abzielt. Ausgehend von einem Datenmodell (*Model*) werden die für die Interaktion notwendigen Komponenten *View* und *Controller* transparent hinzu generiert. Hierbei wird zunächst eine Java Rich Application generiert. Eine Implementierung dieses Ansatzes findet sich aktuell z. B. in Apache Isis [Isi], wobei hier sowohl eine REST Webservice Application Programming Interface (API) als auch eine Webapplikation erzeugt wird.

In diesen Kontext fallen auch sogenannte *Scaffolding*-Ansätze, wie sie auch bereits in Abschnitt 2.3.2 kurz erwähnt wurden. Damit lassen sich ausgehend von einem Daten- bzw. Klassenmodell komplette CRUD-Anwendungen generieren. Derartige Ansätze existieren für eine Vielzahl von Programmiersprachen, wie beispielsweise Ruby on Rails (für Ruby) [RoR, Win13], Grails (für Groovy bzw. Java) [Gra] oder Django (für Python) [Dja]. Diese Ansätze lassen sich, ebenso wie die Standardseiten in MontiWIS, dazu nutzen, grundlegende Administrationsmasken für ein gegebenes Datenmodell zu erzeugen. Die Weiterverarbeitung der Generate ist jedoch aufgrund der in Abschnitt 2.3.2 erwähnten Probleme des Überschreibens generierter Dateien eher schwierig.

6.5.2. Andere Ansätze der Datenmodellierung

Ansätze zur Modellierung von Datenstrukturen existieren verschiedene. Die wohl bekanntesten und verbreitetsten sind ER Modelle [Che76] sowie UML Klassendiagramme [Obj11c]. Hinsichtlich der Aspekte, die abgebildet werden können, sind Klassendiagramme insofern mächtiger, als auch Verhalten mit modelliert werden kann, was bei ER-Modellen nicht möglich ist.

Zunächst sollen an dieser Stelle überblicksartig die Web-Modellierungsansätze betrachtet werden, die ER-Modelle (oder leicht modifizierte Abarten davon) verwenden. Autoweb [FP00], welches seinerseits den Ansatz HDM-lite verwendet, benutzt beispielsweise ein solches Modell zur Spezifikation des Strukturmodells der Webanwendung. Standardmäßig werden bei Autoweb sogenannte *data interfaces* generiert, die zur Eingabe und Modifikation der gespeicherten Daten dienen, was als eine Art Administrationsschnittstelle (ähnlich zu den Standardseiten in MontiWIS) gesehen werden kann. Als einer der Nachfolger von Autoweb werden im Ansatz WebML [BCFM08] ebenfalls ER-Diagramme zur Modellierung der verarbeiteten Datenstruktur eingesetzt. Auch RMM [ISB95] setzt ER-Modelle für die Strukturmodelle ein. In MontiWIS wurde, wie in Abschnitt 6.1.2 erwähnt, Wert auf die Möglichkeiten

objektorientierter Sprachen gelegt, weswegen von der Nutzung von ER-Modellen abgesehen wurde. [20]

Die UML stellt einen natürlichen Kandidaten für die Modellierung von Web-Informationssystemen dar, weswegen auch einige Ansätze diesen Weg beschreiten und sich in ihrer Modellierungssprache daran anlehnen. UWE [KKZB08] verwendet ebenso wie MontiWIS UML Klassendiagramme, um die Datenstruktur der Applikation abzubilden. Hier werden jedoch Domänenmodell und Nutzermodell getrennt betrachtet und modelliert, die Verknüpfungen zwischen beiden Modellen werden über gewöhnliche Assoziationen in den Klassendiagrammen abgebildet. Auch der Ansatz OOWS [FPP+08], welcher auf der OO-Methode [PGIP01] basiert, verwendet UML Klassendiagramme zur Spezifikation des Strukturmodells. Hierbei werden sowohl Klassen als auch deren Attribute und Assoziationen zwischen diesen berücksichtigt. Gleiches gilt für OO-H [GCP01, KKCM04], welches ebenfalls auf der OO-Methode basiert und dementsprechend ähnliche Modelle verwendet. Die OOHDM [RS06, RS08] verwendet eine angepasste Form von Klassendiagrammen für die Modellierung von Datenstrukturen. Diese werden systematisch (aber dennoch manuell) aus vorhergehenden Modellen abgeleitet.

Zusätzlich zu den relativ etablierten Ansätzen wie ER-Modell und UML existieren jedoch auch Ansätze, die exotischere oder domänenspezifische Sprachen für die Spezifikation heranziehen. Auf diese wird im Folgenden eingegangen.

HERA [HvdSB+08] verwendet RDF für die Datenmodellierung. Hierbei werden Konzepte (äquivalent zu Klassen), Attribute und Assoziationen unterstützt. Sowohl Domänendatenstruktur als auch Kontextinformationen werden im Datenmodell abgebildet. Ebenfalls eine Ontologiesprache mit ähnlicher Ausdrucksmächtigkeit zur Datenmodellierung verwendet *WSDM* [DCP08], wobei hier die OWL zum Einsatz kommt.

WebDSL [Vis08b, Webc] verwendet eine DSL zur Spezifikation der Datenstruktur. Ebenso wie MontiWIS setzt WebDSL spezielle, domänenspezifische Datentypen ein, die im Datenmodell Verwendung finden. Die Validierung von Daten und Eingaben wird detailliert in [GV10] erläutert.

Zusammenfassend ist zu sagen, dass die beschriebenen Ansätze im Wesentlichen die Möglichkeiten zur Datenmodellierung bieten, die auch MontiWIS unterstützt. Die Unterstützung für domänenspezifische Attribute und den aus dem Datenmodell abgeleiteten Darstellungen wird jedoch nur in den genannten Fällen angeboten. Leider sind in den seltensten Fälle Informationen über die Art der Implementierung eines Ansatzes verfügbar, so dass hierzu folglich auch keine gesicherte Aussage getroffen werden kann.

[20] Zwar existieren mit dem Extended Entity-Relationship Model [TYF86] Erweiterungen in diese Richtung, die sich jedoch nicht vollständig durchgesetzt haben und nur marginal unterstützt werden.

Kapitel 7.

Seitenbeschreibung

7.1. Grundlagen und Anforderungen

7.1.1. Grundlagen

In Kapitel 6 wurde beschrieben, wie die Datenstruktur des Web-Informationssystems zu beschreiben ist. Allein aus der Domänendatenstruktur werden standardmäßig Seiten generiert, die einem Nutzer CRUD-Funktionalität für sämtliche Klassen bieten. Die dazugehörenden Seiten enthalten jedoch nur rudimentäre Informationen und sind von ihrer Struktur her eher einfach und sehr schematisch gehalten, wie in Abschnitt 6.4.2 erläutert wurde. Bis auf angepasste Überschriften für jede Klasse enthalten die Standardseiten lediglich die Darstellung ihrer Attribute.

Für ein vollständiges Web-Informationssystem, das neben der beschriebenen CRUD-Funktionalität auch erlaubt, Abläufe mit individuellen Seiten darzustellen, ist die Möglichkeit, komplexere Seiten mit feingranularen Sichten auf die Daten zu spezifizieren, obligatorisch. Dies wurde bereits als Grundanforderung in Abschnitt 2.2.1 festgelegt. Aus diesem Grund wurde in MontiWIS dafür eine eigene Sprache entwickelt, über die Seiten spezifiziert werden können.

Aus der Erfahrung bei der Entwicklung mit MontiWeb [DRRS09] zeigte sich, dass die Möglichkeit, statische Anteile einer Seite zu spezifizieren zu können, unabdingbar ist. In MontiWeb war es beispielsweise schwer möglich, mehr als eine Klasse auf einer Seite darzustellen. Über sog. *Classviews* war zwar eine eingeschränkte Sicht auf eine Klasse definierbar, jedoch ließen sich mehrere Objekte einer oder verschiedener Klassen nicht gemeinsam darstellen. Ebenso waren rudimentäre statische Elemente wie erklärende Texte oder strukturierende Überschriften nur unzureichend umsetzbar. Diese Unzulänglichkeiten wurden bei MontiWIS vermieden, indem statische Elemente bei der Definition der Seitenbeschreibungssprache berücksichtigt wurden.

Generell können Seiten in MontiWIS in zweierlei Kontexten auftreten. Zum einen können sie eigenständig angezeigt werden, ohne dass Teile der im System verwalteten Daten darin enthalten sind, was sie zu rein statischen Artefakten macht. Zum anderen können die gespeicherten oder anderweitig festgelegten Daten darauf dargestellt und potenziell verändert werden, womit sie dynamische Anteile enthalten.

Statische Seiten können prinzipiell von jeder beliebigen Stelle aus der Anwendung aufgerufen werden. In ihnen werden keine Daten angezeigt, so dass es folglich auch nicht notwendig ist, diese Daten vorher zu ermitteln bzw. in der Seite zu belegen. Sie können sowohl direkt

aus einem Menü oder beim ersten Aufruf der Applikation angezeigt werden, aber auch Teil eines Workflows sein. In ersterem Fall lassen sich damit Seiten wie das Impressum, Kontaktinformationen oder auch die Begrüßung darstellen. In letzterem Fall dienen sie z. B. zur Anzeige von Statusinformationen, Nachrichten an den Nutzer oder textuellen Erläuterungen.

Dynamische Seiten müssen in MontiWIS grundsätzlich als Teil eines Workflows aufgerufen werden. Ihnen werden in der Regel Objekte übergeben, deren Attribute dargestellt oder mit neuen Werten belegt werden können. Diese Möglichkeit impliziert natürlich auch, dass die übergebenen Objekte vor der Anzeige der Seite geladen und nach Anzeige und Absenden der Seite weiterverarbeitet und gespeichert werden müssen. Genau diese Funktionalität wird üblicherweise in den Aktionen einer Aktivität definiert und ausgeführt. Die Verwendung von Seiten als Teil eines Ablaufs wird in Abschnitt 8.3.3 erläutert.

In diesem Zusammenhang ist noch zu bemerken, dass MontiWIS bewusst auf die Möglichkeit verzichtet, von einer Seite auf eine andere Seite zu verweisen zu können.[21] Eine derartige Möglichkeit ließe sich zwar einfach nachrüsten, widerspricht jedoch dem Paradigma, dass der Übergang zwischen Seiten per Ablaufbeschreibung über Aktivitätsdiagramme modelliert wird. Die leichtfertige Verwendung von Links auf einer Seite und damit außerhalb des Mechanismus der Ablaufsteuerung würde einen derartig ausgeführten Workflow aufbrechen, was explizit nicht gewünscht ist.

7.1.2. Anforderungen an die Seitenbeschreibungssprache

Bei der Entwicklung von MontiWIS wurden verschiedene Möglichkeiten abgewogen, wie Seiten zu beschreiben sind. Wie soeben erläutert, wurde eine Möglichkeit bereits in MontiWeb verwendet und für nicht hinreichend befunden. Daher sollen an dieser Stelle die Anforderungen aufgezeigt werden, die a priori ermittelt wurden.

Darstellung von statischen Inhaltsanteilen

Da in MontiWIS ein vollständiges Web-Informationssystem beschrieben werden soll, muss es auch möglich sein, Seiten zu beschreiben, die lediglich *statischen Inhalt* anzeigen. Dazu muss die Seitenbeschreibungssprache Elemente anbieten, die die Spezifikation statischer Teile auf einer Seite erlauben. Solche Elemente sollten jedoch nicht nur in rein statischen Seiten verwendbar sein, sondern darüber hinaus auch auf dynamischen Seiten, um beispielsweise Erläuterungen zu Eingabefeldern hinzuzufügen oder die Seite generell zu strukturieren.

Sichten auf (Teil-)Attributmengen

Da die vorrangige Aufgabe eines Web-Informationssystems die Anzeige und Bearbeitung von Daten ist, sollte eine Seitenbeschreibungssprache dies auch möglichst gut unterstützen. Die Definition der Datenstruktur in MontiWIS erfolgt, wie in Kapitel 6 erläutert, über Klassendiagramme. Daher sollte es auch auf den einzelnen Seiten der Anwendung möglich sein, die zugehörigen Daten darzustellen und darauf sowohl in lesender als auch schreibender Art

[21] Ausnahme bildet hier der Umweg, in HTML-Abschnitten direkt Links einzufügen. Dies ist selbstverständlich möglich, aber weder zielführend noch empfohlen.

und Weise zuzugreifen. Hierbei müssen nicht nur ganze Objekte einer Klasse, sondern auch Teilmengen der Attribute in unterschiedlichem Kontext mit unterschiedlicher Bezeichnung darstellbar sein. Über solche Sichten lässt sich der (lesende und schreibende) Zugriff auf ausgewählte Attribute einer Klasse realisieren. Nicht nur können darüber bestimmte Attribute vollständig vor den Nutzern verborgen werden, indem sie niemals für ihn sichtbar sind, sondern zusätzlich lassen sich bestimmte Teile einer Klasse auch ausschließlich in einem bestimmten Kontext als Teil eines Ablaufs anzeigen, wobei in einem anderen Ablauf hingegen andere Teile der Klasse sichtbar sind.

Editier- und Darstellungssicht auf einzelne Attribute

Neben der Möglichkeit, die Anzeige lediglich auf bestimmte Attribute zu beschränken, sollte es weiterhin möglich sein, pro Attribut die Art des Zugriffs festlegen zu können. Einerseits sollten Daten in editierbarer Form, z. B. über ein Formularfeld, dargestellt werden können, andererseits ist es ebenso wünschenswert, Objekte lediglich anzuzeigen, ohne diese verändern zu können. Diese beiden Möglichkeiten sollten ebenfalls kombinierbar sein, so dass nur Teile eines Objekts veränderbar sind, während andere Attribute lediglich dargestellt werden, um den aktuellen Kontext deutlich zu machen.

Wiederverwendbarkeit der Seiten

Über die Seitenbeschreibungssprache soll zwar die Struktur und Zusammensetzung der darzustellenden Seite beschrieben werden können, jedoch sollten diejenigen Informationen, die eine Seite in exakt einen Kontext einordnen, anderweitig spezifiziert werden. Dazu zählt insbesondere die Einordnung, an welcher Stelle die Seiten zu erreichen sind und in welcher Reihenfolge sie mit welchen Werten angezeigt werden. Daher sind Seiten frei von Ablauf- und Reihenfolgeinformationen (wie z. B. einen Link auf die nächste anzuzeigende Seite) zu halten, so dass sie in unterschiedlichen Kontexten eingebunden und verwendet werden können. Um eben diese Wiederverwendbarkeit zu unterstützen, sind auch Informationen über Zugriffsberechtigungen bzw. -beschränkungen nicht innerhalb der Seiten, sondern extern zu spezifizieren. Im Falle von MontiWIS werden Reihenfolgebeziehungen und die dazu notwendige Abfolgelogik in den *Aktivitätsdiagrammen* (vgl. Kapitel 8), Zugriffsbeschränkungen und direkte Verlinkungen statischer Seiten aus einem Menü heraus über die sog. *Applikationssprache* (vgl. Kapitel 9) spezifiziert.

Kompakte Syntax der Sprache

Wie in MontiWIS generell angestrebt wird, soll auch die Seitenbeschreibungssprache mit möglichst kompakter, klarer und präziser Syntax auskommen. Dies beinhaltet, dass für die Verwendung eines Attributs lediglich eine Referenz auf das entsprechende Attribut aus dem Domänendatenmodell angegeben werden muss um entweder ein Formularfeld dafür zu erzeugen oder seinen Wert auszugeben. Da die Art der Darstellung eines Attributs, wie in Abschnitt 6.2.3 beschrieben, ohnehin durch seinen vordefinierten Datentyp determiniert wird,

ist es hinreichend, lediglich den Attributnamen zu referenzieren, um daraus die entsprechende Darstellung (sowohl in editierbarer als auch nicht veränderbarer Form) abzuleiten.

Abstraktion von Implementierungsdetails

Durch den modellgetriebenen Ansatz, der in MontiWIS verfolgt wird, ist es generell erklärtes Ziel, den Grad der Abstraktion zu erhöhen, um den Aufwand bei der Softwareentwicklung zu verringern bzw. zu minimieren. Dies sollte auch bei der Spezifikation der Seiten Beachtung finden, so dass in der Seitenbeschreibungssprache keine Details der Implementierung abgebildet werden müssen, sondern die Verwendung der Elemente der Seitenbeschreibungssprache für den Seitenaufbau ausreicht. Wie auch bei der Entwicklung der Webbeschreibungssprache Hypertext Markup Language 5 (HTML5) [HTM] zu bemerken ist, geht die Tendenz hin zu einer stärkeren Trennung zwischen Struktur eines Dokuments (und der Funktion ihrer Elemente) und deren Darstellung. Diese Abstraktion sollte auch in MontiWIS eingehalten werden, so dass über die Seitenbeschreibung keine Darstellungsdetails, sondern lediglich die semantischen Funktionen über die Modellkonstrukte abgebildet werden.

Vollständige Flexibilität durch Einbindung von HTML

Auch wenn generell Ziel sein sollte, von den Konstrukten der HTML zu abstrahieren, ist es in seltenen Fällen dennoch notwendig, marginale Anteile der Seite direkt per HTML-Konstrukt zu spezifizieren. Einerseits sollte der HTML-Ausdruck direkt und ohne dynamische Anteile ausgegeben werden können, andererseits ist es wünschenswert, dass Anteile des Konstrukts dynamisch durch die Applikationslogik spezifiziert werden können. Der Einsatz dieser Möglichkeit ist jedoch mit Vorsicht zu genießen, da invalide HTML-Anteile potenziell zu sehr drastischen Effekten bei der letztlich generierten Seite führen können, was eine vollkommen unleserliche Darstellung nach sich ziehen kann.

7.2. Sprache

Wie später im Abschnitt 7.5 beschrieben wird, existieren mehrere Ansätze für die Beschreibung von Seiten. Im Fall von MontiWIS überwiegen die Vorteile eines deklarativen Ansatzes, weswegen entschieden wurde, eine deklarative Art der Seitenspezifikation zu entwickeln. Hierfür wurde eine eigene Sprache definiert, die den in Abschnitt 7.1.2 gestellten Anforderungen genügt. Der folgende Abschnitt erläutert, wie diese Sprache aufgebaut ist, erklärt, welche Elemente zur Verfügung stehen und beschreibt letztlich, wie die Elemente der Sprache im generierten System erscheinen. Dazu wird zunächst in Abschnitt 7.2.1 ein Überblick der Grammatik gegeben, über die die Sprache spezifiziert wurde. Im nachfolgenden Abschnitt 7.2.2 wird die konkrete Verwendung der einzelnen Elemente der Sprache beschrieben.

Um den Kontext der verwendeten Beispiele zu erläutern, sei auf die Aktivität in Abbildung 7.1 verwiesen.

Die Beispielseiten, die im Verlauf des Kapitels zur Verdeutlichung der Sprachkonstrukte herangezogen werden, entstammen der Aktivität aus Abbildung 7.1. Hier dargestellt ist der

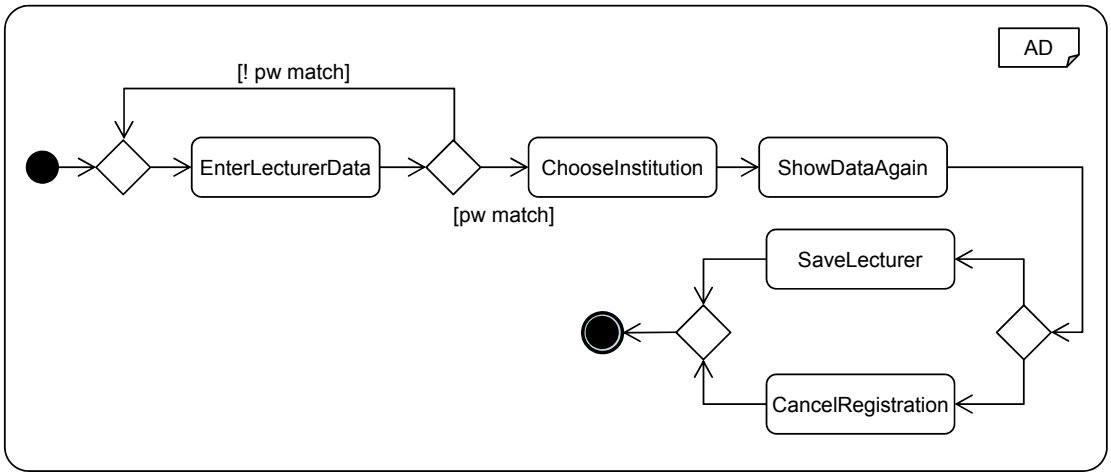


Abbildung 7.1.: Workflow als Beispiel für Seitenbeschreibung

Ablauf, der die Registrierung eines neuen Dozenten beschreibt. Die Einzelheiten, insbesondere was den Datenfluss zwischen den Aktionen betrifft, sind hier aus Gründen der Übersichtlichkeit weggelassen. Die Aktivität beginnt mit der Aktion `EnterLectureData`, in der ein Nutzer seine Daten eingeben muss. Neben seinen Daten, welche auch sein Passwort beinhalten, wird eine erneute Eingabe des Passworts verlangt. Innerhalb der Aktion wird auf Gleichheit der beiden Passwörter geprüft und die Weiterleitung geschieht basierend auf dem Ergebnis des Tests. Im Fall, dass die Passwörter nicht übereinstimmen (`[! pw match]` in der Abbildung), wird erneut die Seite zur Korrektur angezeigt, anderenfalls ist die Aktion `ChooseInstitution` die nächste aufgerufene. Dort muss aus der Liste aller möglichen Institutionen diejenige gewählt werden, der der Dozent angehört. Hierzu wird eine Liste mit Objekten präsentiert, aus der eine Institution zu wählen ist, die im nächsten Schritt zum Dozenten hinzugefügt wird. In der Aktion `ShowDataAgain` werden sämtliche Attribute zur Kontrolle erneut angezeigt, zusammen mit einer Aufforderung zur Überprüfung der Daten und der Wahlmöglichkeit, ob im nächsten Schritt die Daten gespeichert (`SaveLecturer` im Beispiel) oder die Registrierung abgebrochen (`CancelRegistration` in der Abbildung) werden soll. In letzteren beiden Aktionen sind die Seiten insofern personalisiert, als der Name des Dozenten innerhalb der Seite ausgegeben wird.

7.2.1. Grammatik der Seitenbeschreibungssprache

Bevor detailliert beschrieben wird, wie in MontiWIS Seiten über die Seitenbeschreibungssprache spezifiziert werden und welche Regeln bei der Modellierung gelten, wird zunächst überblicksartig kurz auf die Implementierung der Sprache eingegangen. Dazu ist in Abbildung 7.2 eine vereinfachte Version der Grammatik der Seitenbeschreibungssprache dargestellt. Die vollständige Darstellung der Grammatik inklusive aller Elemente, Details und Kommentare findet sich in einem späteren Abschnitt in Anhang C.6.

MCSimp

```
grammar Page extends mc.umlp.common.Common {

  interface PageElement;
  interface TextContents;

  PageDefintion =
    Stereotype?
    "page" name:Name ( Parameters? "{" PageElement* "}" ) | (";")
    ;

  Parameters =
    "(" param:Parameter? ("," param:Parameter )* ")";

  Parameter =
    Type Name;

  StringContents implements TextContents =
    text:StringLiteral;

  AttributeReferenceContents implements TextContents =
    Output:["<"] pageParameterName:Name "." attributeName:Name;

  SimpleAttributeReferenceContents implements TextContents =
    Output:["<"] pageParametername:Name;

  Head implements PageElement =
    "head" "{" TextContents* "}";

  Text implements PageElement =
    "text" "{" TextContents* "}";

  HTMLElement implements PageElement =
    "html" "{" TextContents* "}";

  Table implements PageElement =
    "table"
    "{" TableHead?
      tableRows:TableRow+
    "}";

  TableHead =
    "head" "{" tableCells:TableCell ("|" tableCells:TableCell )* "}";

  TableRow =
    "row" "{" tableCells:TableCell ("|" tableCells:TableCell )* "}";

  TableCell =
    langString:QualifiedName | text:StringLiteral | attributeRef:AttributeReference;

  AttributeReference =
    (Edit:[">"] | Output:["<"] | InfoOutput:["<>"])
```

```
52    entityName:Name ("." attributeName:Name)?
53      ("(" attributesToDisplay:AttributeToDisplay (attributesToDisplay:AttributeToDisplay↩
          )* ")")?;
54
55  AllAttributesShortcut implements PageElement =
56    "allAttributes" "{" (Edit:[">"] | Output:["<"]) entityName:Name "}";
57
58  AttributeToDisplay =
59    simpleAttribute:Name
60    |
61    attributeName:Name "(" referencedAttributeName:Name (referencedAttributeName:Name)* ↩
        ")";
62
63  ListEntities implements PageElement =
64    "list" "{"
65      (Edit:[">"] | Output:["<"])
66      pageVariableName:Name ("(" attributeToDisplay:AttributeToDisplay (↩
          attributeToDisplay:AttributeToDisplay)* ")")?
67    "}";
68
69  ListEntitiesOnly implements PageElement =
70    "listOnly" "{"
71        pageVariableName:Name ("(" attributeToDisplay:AttributeToDisplay (↩
            attributeToDisplay:AttributeToDisplay)* ")")?
72    "}";
73
74  ChooseEntities implements PageElement =
75    "choose" "{"
76      pageVariableName:Name ("(" attributeToDisplay:AttributeToDisplay (↩
          attributeToDisplay:AttributeToDisplay)* ")")?
77    "}";
78
79  ChooseOneEntity implements PageElement =
80    "chooseOne" "{"
81      pageVariableName:Name ("(" attributeToDisplay:AttributeToDisplay (↩
          attributeToDisplay:AttributeToDisplay)* ")")?
82    "}";
83 }
```

Abbildung 7.2.: Vereinfachte Darstellungen der Seitenbeschreibungs-Grammatik

Wie sämtliche anderen Sprachen in MontiWIS wird auch die Seitenbeschreibungssprache von der `mc.umlp.common.Common` Grammatik abgeleitet, womit sämtliche der dort definierten (bzw. auch dort bereits vererbten) Nichtterminale auch in der Seitensprache zur Verfügung stehen. In den Zeilen 3-4 werden die Schnittstellen `PageElement` und `TextContents` deklariert, die in der Grammatik als Stellvertreter für weitere Elemente fungieren. Das Nichtterminal `PageDefintion` (Zeilen 6-9) ist das Startelement, das optional `Parameter` und Inhalt in Form von einer Menge von `PageElement`-Elementen beinhalten oder aber auch leer sein kann. Die Parameter sind in `Parameters` und `Parameter` (Zeilen 11-15) definiert, wobei sich ersteres aus einem oder mehreren von letzteren Elementen zusammensetzt.

Die nächsten drei Nichtterminale implementieren alle die `TextContents` Schnittstelle und lassen sich damit an Stellen verwenden, an denen Textinhalt auf einer Seite gewünscht wird. `StringContents` (Zeilen 17-18) kommt dabei zum Einsatz, wenn gewöhnliche Zeichenketten benötigt werden, `AttributeReferenceContents` (Zeilen 20-21) lässt die Referenz eines Attributs eines der übergebenen Objekte und `SimpleAttributeReferenceContents` (Zeilen 23-24) schließlich lässt die Ausgabe eines vordefinierten Datentyps zu. All diese Elemente können in den Überschriften (über das Nichtterminal `Head` (Zeilen 26-27)), einfachem Text (über `Text` (Zeilen 29-30)) sowie beliebigem HTML-Code (über das Nichtterminal `HTMLElement` (Zeilen 32-33)) in beliebiger Anzahl und Reihenfolge verwendet werden. Die letzten drei Nichtterminale sind auch diejenigen, die die Schnittstelle `PageElement` implementieren und damit ebenfalls in beliebiger Reihenfolge in einer Seite vorkommen können. Neben diesen Elementen sind ebenfalls Tabellen (implementiert durch das Nichtterminal `Table` (Zeilen 35-39)) und die Darstellung sämtlicher Attribute einer Klasse (`AllAttributesShortCut` (Zeilen 55-56)) möglich. Listendarstellungen von Objekten werden sowohl mit Editierfunktionalität (über das Nichtterminal `ListEntities` (Zeilen 63-67)) als auch ohne die Möglichkeit der Interaktion (über das Nichtterminal `ListEntitiesOnly` (Zeilen 69-72)) angeboten. Schließlich existiert die Auswahlmöglichkeit von mehreren Objekten (`ChooseEntities` in Zeilen 74-77) oder lediglich einem Objekt über das Nichtterminal `ChooseOneEntity` in Zeilen 79-82. Weiterhin werden in den Zeilen 41-48 die Nichtterminale `TableHead`, `TableRow` und `TableCell` zur Konstruktion von Tabelleninhalten definiert. In den Zeilen 50-53 wird mit dem Element `AttributeReference` der Inhalt einer Tabellenzelle spezifiziert, über den in verschiedener Art auf Seitenparameter zugegriffen werden kann. Referenzen auf Seitenparameter werden über das Nichtterminal `AttributeToDisplay` (Zeilen 58-61) abgebildet.

7.2.2. Verwendung der Seitenbeschreibungssprache

Nach dem kurzen Überblick über die Grammatik der Seitenbeschreibungssprache werden in diesem Abschnitt konkrete Beispiele zur Verwendung der Seitensprache besprochen. Ein minimales Beispiel dieser Sprache ist in Abbildung 7.3 dargestellt.

Page

```
1 page Index {
2   head { "Welcome!" }
3   text { "These pages show an overview of the possibilities of MontiWIS." }
4   text { "Please select an action on the left hand side." }
5 }
```

Abbildung 7.3.: Beispielhafter Aufbau einer Page-Sprachdatei

Bei diesem Beispiel ist zu Beginn der Datei die Paketdeklaration ausgelassen. Ferner sind in diesem Beispiel keine `import` Anweisungen angegeben, was aufgrund fehlender Referenzen zu anderen Modellen hier nicht notwendig ist. Wie Paket- und Importdeklarationen zu spezifizieren sind und welche Bedeutung ihnen zukommt, ist in Abschnitt 4.2 beschrieben.

Ein Seitenbeschreibungsmodell beginnt mit dem Schlüsselwort `page` gefolgt vom *Namen* der Seite (in diesem Fall `Index`). Der Inhalt der Seite ist eingeschlossen zwischen geschweiften Klammern anzugeben (Zeilen 1, 5). Zur Demonstration sind im Beispiel lediglich einfache statische Elemente enthalten. Zunächst wird über das Element `head` eine Überschrift spezifiziert (Zeile 2), gefolgt von zwei `text` Elementen (Zeilen 3 und 4), die zur Definition von Fließtext-Absätzen innerhalb einer Seite dienen. Der Inhalt des jeweiligen Abschnitts (sowohl Überschriften als auch Absätzen) wird umschlossen von geschweiften Klammern. Im Beispiel wird lediglich einfacher Text (eingefasst von Anführungszeichen) verwendet. Weitere Möglichkeiten der Spezifikation von Inhalten sind weiter unten in diesem Abschnitt beschrieben.

Um zu verdeutlichen, wie das einfache Beispiel aus Abbildung 7.3 in der generierten Applikation dargestellt wird, ist der Seitenausschnitt in Form eines Screenshots in Abbildung 7.4 dargestellt.

Abbildung 7.4.: Generierte Seite der Sprachdatei aus Abbildung 7.3

Der dargestellte Teil der Seite entspricht der Formatierung, wie sie aus dem Modellausschnitt generiert wurde. Der Textteil, der als Überschrift definiert wurde, wird im Zielsystem standardmäßig in Fettdruck dargestellt. Die Textabsätze (ausgezeichnet über `text` in der Seitensprache) werden als Absätze auf der Seite dargestellt.

Nachdem nun anhand eines Beispiels ganz generell der Aufbau eines Seitenmodells dargestellt wurde, soll nun auf die einzelnen Elemente der Seitensprache eingegangen werden. Die in den folgenden Abschnitten beschriebenen Konstrukte können alle in beliebiger Anzahl und Reihenfolge innerhalb der Seitendeklaration vorkommen und werden entsprechend der Reihenfolge ihres Auftretens übernommen.

Seitendeklaration

Neben den üblichen Teilen eines MontiCore-Modells (Paket- und Importdeklarationen) enthält jede Seitenbeschreibung exakt eine Seitendeklaration. Diese ist in Abbildung 7.5 dargestellt.

Wie bereits beschrieben, beginnt eine Seite mit dem Schlüsselwort `page`, gefolgt vom Namen der Seite (in diesem Fall `EnterLecturerData` in Zeile 5). Auch hier gilt, wie in MontiCore üblich, dass der Name des Modells dem der Datei entsprechen muss, in der es gespeichert ist. Das Suffix für Seitenbeschreibungen ist `pg`, weswegen das Beispiel in einer Datei mit Namen `EnterLecturerData.pg` zu speichern ist.

Seitenparameter

Optional können Seiten *Parameter* besitzen, wie ebenfalls im Beispiel in Abbildung 7.5 zu sehen ist. Parameter werden, wie auch in der überwiegenden Zahl der Programmiersprachen

Page

```
1 package tm;
2
3 import tm.*;
4
5 page EnterLecturerData (Lecturer lec, Password pw) {
6
7   // ...
8
9 }
```

Abbildung 7.5.: Genereller Aufbau einer Seitenbeschreibung

und u. a. in Java, in Klammern angegeben und über Kommata getrennt. Ein Parameter besteht aus dem Typ des Parameters (`Lecturer` bzw. `Password` im Beispiel) und seinem Namen (`lec` oder auch `pw` im Beispiel). Ein Parametertyp kann entweder auf eine Klasse aus dem Domänendatenmodell eines Projekts verweisen, oder aber ein *vordefinierter Datentyp* sein (vgl. Abschnitt 6.2.3). Beide Fälle sind ebenfalls im Beispiel dargestellt, wo mit `Lecturer` auf eine Klasse des Klassendiagramms aus dem Domänendatenmodell verwiesen wird (vgl. auch Abschnitt 6.2.2 und insbesondere Abbildung 6.5 bzw. 6.6). Ein Parameter in Form eines vordefinierten Datentyps ist beispielhaft über das `Password pw` dargestellt. Durch die Möglichkeit, einfache vordefinierte Datentypen ebenfalls als Parameter und damit im Folgenden auch als Eingabe- bzw. Ausgabefeld in einer Seite verwenden zu können, lassen sich deren Werte innerhalb von Abläufen verwenden, ohne dass sie Teil einer Klasse des Domänendatenmodells sein müssen.

Die Parameter einer Seite lassen sich aus einer Aktion heraus belegen, so dass die übergebenen Werte auf der Seite dargestellt werden können. Andererseits kann auch auf die (potenziell veränderten) Werte aus der Seite zugegriffen werden, nachdem die Seite abgesandt wurde. Der Mechanismus, wie dieser Zugriff erfolgt, ist in Abschnitt 8.3 beschrieben.

Damit die Seitenparameter Elementen des Datenmodells korrekt zugeordnet werden können, ist es notwendig, dass die entsprechenden Klassendiagramme in das Seitenmodell *importiert* werden, wie es im Beispiel in Zeile 3 geschieht. Damit können sowohl die Klassen aus dem Klassendiagramm als auch die vordefinierten Klassen über die Kontextbedingungen aufgelöst und damit die Korrektheit des Modells sichergestellt werden.[22] Diese Mechanismen sind in Abschnitt 7.3.1 detaillierter beschrieben.

Neben einzelnen Objekten einer Klasse können ebenfalls *mengenwertige Parameter* verwendet werden. Dies geschieht über die Verwendung einer `Collection` von Objekten. Für die Übergabe einer Menge von `Lecturers` wäre der Parameter `Collection<Lecturer>`, gefolgt von einem sinnvollen Namen, zu verwenden, worüber die Sammlung von Objekten in der Seite angesprochen werden kann. Die mengenwertigen Parameter können als eine Liste

[22] Voraussetzung dafür ist, dass beide Modelle im Paket `tm` definiert sind und auch in diesem Verzeichnis liegen. Anderenfalls müssen die referenzierten Pakete entsprechend angepasst werden.

von Objekten dargestellt werden, oder auch in einer Tabelle, in der über Radiobuttons oder Checkboxen exakt eines oder mehrere Objekte ausgewählt werden können.

Überschriften

Um eine Seite in bestimmte visuelle Abschnitte aufteilen zu können, erlaubt die Seitenbeschreibungssprache von MontiWIS die Verwendung von *Überschriften*. Deren Verwendung ist beispielhaft in Abbildung 7.6 dargestellt.

```
page Registered (Lecturer lec) {                                    Page

  head { "Welcome " < lec.firstName " " < lec.surName "!" }
  // ...

}
```

Abbildung 7.6.: Verwendung einer Überschrift

Die Seite, deren Teil die Überschrift ist, wird innerhalb der Aktion `SaveLecturer` aufgerufen und dient als Bestätigung, dass die Registrierung erfolgreich war. Eine Überschrift wird generell codiert über das Schlüsselwort `head` und enthält innerhalb geschweifter Klammern den Inhalt, der als Überschrift formatiert werden soll. Die Überschrift selbst besteht aus Teilen, die statisch übernommen werden (eingeschlossen in Anführungszeichen analog zur gewöhnlichen `String`-Repräsentation), kann aber auch Teile enthalten, die aus den übergebenen Parametern übernommen werden. Die Syntax für die Ausgabe eines Attributs ist angelehnt an das Konstrukt, wie es innerhalb von Tabellen für die Ein- bzw. Ausgabe von Attributen verwendet wird. Der dynamische Teil wird codiert durch die Verwendung eines *kleiner*-Zeichens (`<`) sowie der Referenz auf das auszugebende Attribut des Parameters. Im Fall des Beispiels werden sowohl der Vorname (`lec.firstName`) als auch der Nachname (`lec.surName`) ausgegeben. Ein Verbindungsliteral für die Verknüpfung von statischem String-Inhalt und dynamischem Inhalt aus Attributen, wie es beispielsweise über ein `+` in Java üblich ist, wird hier in einer Überschrift nicht benötigt. Statischer Inhalt und Attributwerte können mehrfach und in beliebiger Reihenfolge hintereinandergeschrieben werden.

Freitext-Absatz

Ähnlich zu Überschriften können auch Absätze aus Freitext spezifiziert werden. Ein Beispiel eines Textabschnitts ist in Abbildung 7.7 dargestellt.

```
1 page Registered (Lecturer lec) {                                    Page
2
3   head { "Welcome " < lec.firstName " " < lec.surName "!" }
4   text { < lec.firstName ", you can now log in to the system on the left hand side
5       and select your next actions in the menu as well." }
6
7 }
```

Abbildung 7.7.: Verwendung eines Textabsatzes

Textabsätze werden über das Schlüsselwort `text` definiert und schließen ihren Inhalt in geschweiften Klammern ein. Wie in den Zeilen 4-5 zu sehen ist, können auch Textabschnitte sowohl statischen `String`-Inhalt als auch dynamische Anteile aus den Attributen von Seitenparametern enthalten. Die Syntax des Inhalts eines Absatzes entspricht der einer Überschrift. `String`-Inhalte und Attributreferenzen werden einfach hintereinander angegeben und lassen sich so beliebig kombinieren.

Die Möglichkeit, beliebigen Textinhalt aus einem Parameter zu extrahieren, erlaubt auch die dynamische Kombination von Seiteninhalten analog zu einem Content-Management-System. Inhalte, die in der Datenstruktur als `Text`-Datentyp deklariert wurden, können als Inhalte in einem Textabsatz einer Seite eingebunden und so dynamisch zu Fließtexten kombiniert werden.

Tabellen mit Parameterreferenzen

Eine zentrale Bedeutung in MontiWIS kommt der Tabellendarstellung innerhalb einer Seite zu. Abbildung 7.8 zeigt das Beispiel einer typischen Verwendung einer Tabelle inklusive der möglichen Inhalte der einzelnen Tabellenzellen. Ein Screenshot einer sehr ähnlichen Seite ist in Abbildung 11.1 zu sehen, wobei dort die Attribute leicht von den hier verwendeten differieren. Das Prinzip der Tabellendarstellung sollte jedoch darin erkennbar sein.

Wie in Zeile 8 zu sehen, beginnt eine Tabelle mit dem Schlüsselwort `table`, worin zwischen geschweiften Klammern (Zeilen 8-13 sowie 15-19) der eigentliche Inhalt der Tabelle definiert ist.

Eine Tabelle kann optional, wie in Zeile 9 dargestellt, eine *Kopfzeile* besitzen. Eine solche wird syntaktisch codiert über das Schlüsselwort `head`, wobei der Inhalt ebenfalls wieder in geschweiften Klammern angegeben wird. Innerhalb der Kopfzeile werden die einzelnen Spalten getrennt durch ein Pipezeichen (`|`). Der Inhalt einer Tabellenzelle kann verschiedenartig sein. Eine Übersicht über die verschiedenen Möglichkeiten ist in Tabelle 7.9 aufgelistet.

Zunächst kann in einer Tabellenzelle einfacher Text stehen. Hierfür wird der Freitext in gewöhnlichen Anführungszeichen in einer `String`-Schreibweise angegeben. Dies ist beispielsweise in Zeilen 11-12 und 16-18 in Abbildung 7.8 der Fall. Dieser Text wird in exakt der spezifizierten Form übernommen und so in der generierten Tabellenzelle dargestellt.

Eine andere Möglichkeit für die Darstellung von Text in einer Tabelle ist die Verwendung

Page

```
1  package tm;
2
3  import tm.*;
4
5  page EnterLecturerData (Lecturer lec, Password pw) {
6
7    head { "Please enter your credentials" }
8    table {
9      head { tableheading.attribute | tableheading.value }
10     row { Lecturer_userName | > lec.userName }
11     row { "Password" | > lec.password }
12     row { "Password again" | > pw }
13   }
14   head { "... and the rest of your data" }
15   table {
16     row { "First name" | > lec.firstName }
17     row { "Surname" | > lec.surName }
18     row { "Title" | > lec.title }
19   }
20 }
```

Abbildung 7.8.: Verwendung einer Tabelle

von `ResourceBundle`-Referenzen. Eine solche Referenz wird über einen Schlüsselwert definiert, der nicht in Anführungszeichen gesetzt ist und möglichst aussagekräftig seine Funktion beschreiben sollte. Im Beispiel in Abbildung 7.8 wird eine solche Referenz in Zeile 9 in der Tabellenkopfzeile verwendet. Der Vorteil einer solchen ausgelagerten Definition von Tabelleninhalt ist die Möglichkeit, in der generierten Applikation bestimmte Ausdrücke einheitlich und ebenfalls in verschiedenen Sprachen spezifizieren zu können. Werden, wie im Beispiel dargestellt, die Kopfzeilen derartig angegeben und die zugehörigen Schlüsselwerte konsistent in jeder Tabelle verwendet, so kann zentral in einer Datei die zugehörige Beschriftung ausgetauscht werden. Ebenfalls sinnvoll kann der Einsatz des standardmäßig generierten Schlüsselwertes für Attribute einer Klasse sein. Wie in Abschnitt 7.3.4 beschrieben werden aus dem Element `allAttributes` Standardansichten zur Darstellung aller Attribute einer Klasse generiert, die für die Bezeichnung der Attribute einen einheitlichen Schlüssel verwenden, der somit zentral für die Namensgebung eines Attributs verwendet werden kann. Dies ist jedoch nur sinnvoll, wenn in einer Seite keine kontextspezifischen Namen der Felder notwendig oder sinnvoll erscheinen. Der generierte Schlüsselwert besteht aus dem Klassennamen (beginnend mit einem Großbuchstaben) und dem Attributnamen (klein beginnend), welche durch einen Unterstrich miteinander verbunden sind. Ein Beispiel für die Verwendung eines solchen Bezeichners ist in Zeile 10 zu finden. Der Mechanismus der `ResoureBundles` wird detaillierter in Abschnitten 5.3.2 und 10.4.1 beschrieben.

Neben der Spezifikation von Text auf zwei verschiedene Arten ist in einer Tabelle auch der Zugriff auf die Parameter einer Seite möglich.

Art der Zelle	Beispiel	Erläuterung
Freitext	`"Vorname"`	Einfacher Freitext in `String`-Form, der direkt übernommen wird
`ResourceBundle`-Referenz	`tableheading.attribute`	Schlüsselwert, der innerhalb einer `ResourceBundle`-Datei aufgelöst wird
Ausgabe eines Attributs	`< lec.firstName`	Ausgabe des Attributs eines Seitenparameters
Ausgabe einer Referenz	`< lec.worksAt (name)`	Ausgabe einer Referenz auf ein anderes Objekt mit Angabe, welche Attribute dargestellt werden sollen
Eingabe eines Attributs	`> lec.firstName`	Eingabe des Attributs eines Seitenparameters
Ausgabe eines Links	`<> lec.worksAt (name)`	Ausgabe eines Links auf ein anderes Objekt mit Angabe, welche Attribute dargestellt werden sollen

Tabelle 7.9.: Möglicher Inhalt von Tabellenzellen

Um ein Attribut auszugeben, ohne dass eine Möglichkeit besteht, dieses zu verändern, ist das *kleiner*-Zeichen (<), gefolgt von der Referenz auf das entsprechende Attribut zu verwenden. Die Referenz gliedert sich in den Namen des Parameters (`lec` im Beispiel für die Referenz auf den `Lecturer`), gefolgt von einem Punkt und dem Namen des auszugebenden Attributs (beispielsweise `firstName` in Zeile 16 oder Tabelle 7.9). Die Darstellung erfolgt entsprechend des Datentyps des referenzierten Attributs, also beispielsweise `Text`, `Password` oder `Date`.

Analog zur Ausgabe eines Attributs kann über das gleiche Schema ein veränderbares Feld für das Attribut des Parameters erzeugt werden. Statt das *kleiner*-Zeichens ist jedoch das *größer*-Zeichen (>) als Präfix zu verwenden. Auch hier wird im generierten System entsprechend des Datentyps Zugriff auf das Attribut gewährt, also beispielsweise über ein Text- oder Passwortfeld in der Webseite.

Für den Fall, dass das Attribut eine Referenz auf ein anderes Objekt ist, lässt sich optional in Klammern festlegen, welche Attribute in der Ausgabedarstellung angezeigt werden sollen, wobei mehrere Attribute durch Leerzeichen getrennt werden. Bei derartiger Spezifikation von Attributen in der Editieransicht werden die gelisteten Attribute sowohl bei der Bezeichnung bereits assoziierter Objekte ausgegeben, als auch in der Liste zur Auswahl neuer Objekte zur Festlegung der darzustellenden Spalten herangezogen. Dies ist insofern sinnvoll, als nicht

sämtliche Attribute eines referenzierten Objekts ausgegeben werden können und auch in der Regel nicht sollen. Ähnlich der Darstellung in einer Liste werden bei Fehlen der Attribute diejenigen verwendet, die im Klassendiagramm mit dem Stereotyp «`inList`» annotiert wurden. Sollte auch dieser Stereotyp fehlen, wird in der Darstellung der Referenz der eindeutige Bezeichner (`id`) verwendet. Ist das referenzierte Attribut Teil einer Komposition, so wird die entsprechende Darstellung an dieser Stelle in der Tabelle eingebunden. Bei einem Attribut, das eine Assoziation repräsentiert, ist es zusätzlich möglich, über die Angabe des *größer- und kleiner-Symbols* (`<>`), einen Link zu erzeugen, der von der Beschriftung her analog zur Anzeigeansicht des Attributs dargestellt wird. Dadurch wird in einem separaten Pop-up-Fenster die Anzeigeansicht des referenzierten Objekts dargestellt. Somit steht eine Vorschau des Objekts zur Verfügung, die ohne Verlassen der Seite sämtliche Attribute eines referenzierten Objektes anzeigt.

Ein etwas anderes Schema wird bei der Referenz auf einen Parameter verwendet, wenn dieser kein Element des Domämendatenmodells des Klassendiagramms, sondern einer der vordefinierten Datentypen ist. Hier ist der einfache Name des Seitenparameters zu verwenden, wobei hinsichtlich Editier- oder Ausgabeansicht des Parameters die gleichen Symbole wie bei Attributreferenzen gelten. Ein Beispiel für den Zugriff auf einen vordefinierten Datentyp ist in Zeile 12 dargestellt, wo die erneute Eingabe des Passworts ebenfalls als Teil der Tabelle gefordert wird.

Die beschriebenen Zelleninhalte einer Tabelle gelten nicht ausschließlich für den Tabellenkopf, sondern ebenfalls und insbesondere auch für die einzelnen Tabellenzeilen. Eine Zeile wird codiert über das Schlüsselwort `row` und enthält, ebenso wie der Tabellenkopf, einzelne Zellen innerhalb von geschweiften Klammern und getrennt voneinander durch ein Pipesymbol. Attribute und Seitenparameter sollen in einer Tabellenzeile ausgegeben und editiert werden, innerhalb einer Tabellenüberschrift wird davon abgeraten.

Darstellung aller Attribute

Für den Fall, dass keine besonderen Ansprüche an die Darstellung eines Objekts gestellt werden und lediglich sämtliche Attribute auf einer Seite dargestellt werden sollen, existiert in MontiWIS ein Seitenelement, das genau diese Funktionalität bietet. Abbildung 7.10 demonstriert, wie es verwendet wird.

```
1 page ShowLecturer(Lecturer lec) {                                    Page
2
3   head { "Hello, " < lec.firstName " " < lec.surName ","  }
4   text { "shall this data be saved like this?" }
5   allAttributes { < lec }
6
7 }
```

Abbildung 7.10.: Verwendung des Elements zur Darstellung aller Attribute

Nach einer Überschrift und einem kurzen erklärenden Text auf der Seite sollen sämtliche Attribute des übergebenen Objekts ausgegeben werden. Dazu wird das Element `allAttributes` verwendet. In geschweiften Klammern wird der Parameter, dessen Attribute ausgegeben werden sollen, deklariert. Ferner wird über die bekannten Symbole (*kleiner*-Zeichen für die Ausgabe wie im Beispiel bzw. *größer*-Zeichen für Editierbarkeit) festgelegt, ob die Attribute veränderbar sein sollen. Das verwendete Symbol bezieht sich auf sämtliche Attribute der Klasse, es besteht folglich keine Möglichkeit, einzelne Attribute auszublenden oder einige in editierbarer und andere in nicht-editierbarer Form darzustellen. Sollte dies gewünscht sein, sind die bekannten Mittel zur Seitenspezifikation in Tabellenform sowie die explizite Referenz der einzelnen Attribute zu verwenden.

Durch die Möglichkeit, alle Attribute einer Klasse in standardisierter, d. h. tabellarischer Form mit Attributnamen in der linken Spalte und je nach Darstellungsart Attributwert oder entsprechendem Eingabefeld in der rechten Spalte darzustellen, entspricht die generierte Darstellung im Wesentlichen der für jede Klasse erzeugten Standardseiten (`View` bzw. `Edit` oder `New`) mit angepasster Überschrift.

Neben der Darstellung einzelner Objekte innerhalb einer Seite existieren verschiedene Möglichkeiten, Mengen von Objekten zu verarbeiten. Die dazu verwendbaren Seitenelemente werden in den nächsten Abschnitten beschrieben.

Listendarstellung zur Auswahl eines Objekts

Innerhalb eines Web-Informationssystems kommt es häufig vor, dass eine Liste von Objekten angezeigt werden soll, beispielsweise, wenn sämtliche Objekte eines Typs oder auch lediglich eine eingeschränkte Menge von Objekten als Ergebnis einer Suche dargestellt werden sollen. Für diesen Zweck können in MontiWIS verschiedene Elemente genutzt werden, wovon zunächst diejenigen beschrieben werden, die die Möglichkeit zur Auswahl eines oder mehrerer Elemente und einer nachgelagerten Weiterverarbeitung bieten. Ein Beispiel findet sich in Abbildung 7.11.

```
1 page ChooseInstitution(Collection<Institution> institutions) {                Page
2
3   head { "Please select one of the institutions this lecturer belongs to:" }
4   chooseOne { institutions (name) }
5
6 }
```

Abbildung 7.11.: Verwendung der Listendarstellung zur Auswahl eines Objekts

Hier dargestellt ist die Seite zur Auswahl einer `Institution` aus einer Menge von übergebenen Objekten. Über das Element `chooseOne` (Zeile 4) wird festgelegt, dass die Menge in einer Art und Weise dargestellt wird, bei der ein Nutzer exakt ein Element auswählen kann. Bei diesem Element wird in geschweiften Klammern der Seitenparameter angegeben, der

die Sammlung von Objekten enthält. Da in einer Liste von Objekten in der Regel nicht alle Attribute angezeigt werden können, und dies gewöhnlich auch nicht gewünscht ist, kann bei der Listendarstellung spezifiziert werden, welche der Attribute in der Liste in welcher Reihenfolge spaltenweise angezeigt werden sollen. Dies geschieht in Klammern nach der Referenz auf den Parameter. Im Beispiel soll der `name` der `Institution` in der Liste angezeigt werden. Sollen mehrere Attribute dargestellt werden, so werden diese über ein Leerzeichen getrennt zwischen den Klammern aufgelistet. Jedes Attribut wird in der Reihenfolge seines Auftretens in einer eigenen Spalte angezeigt, so dass die Tabelle auch nach jedem dieser Attribute sortiert werden kann. Die Spezifikation der Attribute ist optional. Sollten keine Attribute vorhanden sein, werden diejenigen, die im Klassendiagramm mit dem Stereotyp «`inList`» annotiert wurden, an dieser Stelle verwendet. Sollten auch keine solchen Stereotypen festgelegt worden sein, so wird der eindeutige Bezeichner (`id`) des Objekts verwendet.

Zusätzlich zu den Attributen auf der Seite wird zu Beginn der Liste eine Spalte hinzugefügt, in der ein *Radiobutton* eingefügt wird. Dies stellt sicher, dass von den angezeigten Objekten exakt eines ausgewählt werden kann. Eine Beschreibung, wie die Tabelle im generierten System umgesetzt wird, ist Abschnitt 7.2.3 zu entnehmen, die programmatische Umsetzung wird in Abschnitt 7.4 erläutert.

Listendarstellung zur Auswahl mehrerer Objekte

Soll nicht ein Objekt ausgewählt werden, sondern ist erwünscht, dass mehrere Objekte aus der Liste weiterverarbeitet werden, ist auch dies über ein Seitenelement möglich. Dies ist in Abbildung 7.12 dargestellt.

```
1 page ChooseInstitutions(Collection<Institution> institutions) {
2
3   head { "Please select the institutions this lecturer belongs to:" }
4   choose { institutions (name) }
5
6 }
```

Page

Abbildung 7.12.: Verwendung der Listendarstellung zur Auswahl mehrerer Objekte

Der einzige Unterschied zum Element zur Darstellung mehrerer Objekte und Auswahl *eines* Objekts besteht im Schlüsselwort, welches dafür in der Seite zum Einsatz kommt. Hierfür wird das Schlüsselwort `choose` (im Gegensatz zu `chooseOne`) verwendet. Identisch ist die Behandlung der weiteren Teile des Elements. Die Referenz auf einen `Collection`-Parameter hat ebenso in geschweiften Klammern zu stehen und auch die Spezifikation anzuzeigender Attribute in der Liste erfolgt in Klammern nach der Parameterreferenz.

In der generierten Applikation unterscheidet sich diese Art von Auswahl dadurch, dass statt Radiobuttons Checkboxen verwendet werden, welche eine mehrfache Auswahl ermöglichen.

Listendarstellung mehrerer Objekte

In einem Web-Informationssystem kann es erwünscht sein, dass eine Menge von Objekten direkt in einer Liste ausgegeben wird und darin Funktionalität zur Anzeige bzw. Bearbeitung eines jeden Objekts zur Verfügung steht. Solches Verhalten ist in der Regel genau das, was in der Listenansicht einer generierten Standardseite Verwendung findet (vgl. hierzu Abschnitt 6.4.2). In MontiWIS können diese Elemente jedoch auch in jeder beliebigen Seite eingebunden werden.

Das hier beschriebene Verhalten wird nicht im Kontext des Ablaufs verwendet, der zur Veranschaulichung in diesem Abschnitt dient. Grund hierfür ist, dass über die Möglichkeit zur Auflistung von Objekten keine Interaktion mit dem aufrufenden Workflow möglich ist. Dies bedingt auch, dass die Listendarstellung, wie sie hieraus erzeugt wird, in der Regel nicht häufig im Rahmen eines Seitenablaufs eingesetzt wird. Dennoch ist ein Beispiel für die hier beschriebenen Möglichkeiten in Abbildung 7.13 dargestellt.

Page

```
1 page InspectInstitutions(Collection<Institution> institutions) {
2
3   head { "Here you can see a collection of all existing institutions:" }
4   list { < institutions (name) }
5
6 }
```

Abbildung 7.13.: Verwendung der Listendarstellung zur Ansicht mehrerer Objekte

Generell wird die hier beschriebene Listenfunktionalität über das Schlüsselwort `list` in eine Seite eingebunden. In geschweiften Klammern wird auf den mengenwertigen Parameter verwiesen, dessen Inhalt dargestellt werden soll. Auch bei diesem Element ist es möglich, in Klammern die darzustellenden Attribute festzulegen.

Die Listendarstellung erlaubt sowohl die reine Darstellung der Menge der übergebenen Objekte als auch die Auflistung mit dem Zugriff auf Editier- bzw. Löschfunktionalität. Soll lediglich ein Link auf die Standardseite zur Darstellung eines Objekts dargestellt werden, so ist das *kleiner*-Zeichen, wie im Beispiel zu sehen, zu verwenden. Sollen zusätzlich Verweise auf die Editier- und Löschfunktionalität eingebunden werden, muss das *größer*-Zeichen, wie in MontiWIS üblich für Editierdarstellung, gewählt werden.

Ferner besteht die Möglichkeit, über das Schlüsselwort `listOnly` die Objekte in einer derartigen Liste darzustellen, weiterführende Links jedoch vollständig auszublenden. Darüber ist weder die Seite zur Darstellung eines Objekts noch diejenige zur Veränderung erreichbar.

Spezielle Beachtung soll an dieser Stelle dem Thema Sicherheitsmechanismen und deren Auswirkung auf das Verhalten dieses Elements geschenkt werden. Sollte eines dieser Elemente auf einer Seite verwendet werden, finden die Sicherheitsmechanismen, die im Applikationsmodell definiert wurden, auch an dieser Stelle Anwendung. Da das Standardverhalten für die Seiten zur Anzeige von Objekten in MontiWIS generell Zugriff erlaubt, ist auch stets der Link zu dieser Seite verfügbar. Auch ein Klick darauf führt zur Anzeige des jeweiligen

Elements auf der Standardseite. Anders verhält es sich mit den Editier- bzw. Löschverweisen. Hier wird bei der Anzeige ebenfalls geprüft, ob der aktuell aufrufende Nutzer in einer Rolle im System agiert, die das Editierrecht auf Objekten des aufgelisteten Typs gewährt. Im Fall des dargestellten Beispiels wäre es folglich notwendig, dass der Nutzer Editierrechte auf `Institution`-Objekte besitzt, damit die entsprechenden Links in der Seite dargestellt werden. Dies ist insofern sinnvoll, als über die Verweise ohnehin die Standardseiten (bzw. die Standardfunktionalität im Falle des Löschlinks) aufgerufen werden und spätestens dort der MontiWIS-übergreifende Berechtigungsmechanismus einen Zugriff verhindern würde. Die Spezifikation der Berechtigungen in MontiWIS und deren Auswirkungen werden ausführlich in Kapitel 9 beleuchtet.

Dieses Verhalten ist auch ein Grund, weswegen vom Einsatz der hier beschriebenen Elemente abgeraten wird, es sei denn, der Nutzer ist sich bewusst über die Implikationen, die insbesondere die Anwendung der Berechtigungsrestriktionen im generierten System darauf hat.

Einbindung von beliebigem HTML-Code

Um der Anforderung aus Abschnitt 7.1.2 zu genügen, die die Einbindung beliebigen HTML-Codes fordert, kann über das Seitenelement `html` eben solcher Inhalt eingebunden werden. Abbildung 7.14 zeigt, wie dies vonstattengeht.

Page

```
1 page ImageDisplay (String image) {
2
3   html {
4     "<img src='" < image "' width='800px' height='600px' />"
5   }
6
7 }
```

Abbildung 7.14.: Verwendung der Einbindung von beliebigem HTML-Code

Wie in dieser Seitendefinition zu sehen, lässt sich bei Verwendung des `html`-Elements in geschweiften Klammern analog zu Text oder Überschriften beliebiger Text einbinden. Dabei ist auch die Verwendung von dynamischen Anteilen (`image` in Abbildung 7.14) möglich. Im Beispiel wird durch ein `img`-Tag ein Bild in die Seite eingebunden, wobei die konkrete Spezifikation, welches Bild verwendet werden soll, per Seitenparameter festgelegt wird. In `html`-Seitenelementen lassen sich beliebige Inhalte auf diese Weise spezifizieren, was quasi beliebigen (statischen) Seiteninhalt ermöglicht.

Bei der Verwendung dieses Elements ist jedoch insofern Vorsicht geboten, als die dort eingebundenen Inhalte an sich valide sein müssen, so dass die generierte Seite nicht durch inkorrekten Code unbrauchbar wird.

Generell sollte dieses Element nur in solchen Fällen eingesetzt werden, in denen nicht über die anderen Elemente zu einem zufriedenstellenden Ergebnis gelangt werden kann.

7.2.3. Darstellung im generierten System

Nach der Beschreibung der unterstützten Elemente der Seitenbeschreibungssprache in MontiWIS soll nun darauf eingegangen werden, wie die einzelnen Elemente einem Nutzer in der generierten Anwendung präsentiert werden. Hierbei werden zunächst die überwiegend statischen Elemente wie Überschriften und Freitext beschrieben, nachfolgend wird auf die Darstellung jeder einzelnen Tabellenzelle (abhängig vom referenzierten vordefinierten Datentyp) eingegangen, wobei auch Attribute erläutert werden, die Assoziationen oder Kompositionen repräsentieren und damit andere Objekte referenzieren bzw. einbinden. Schlussendlich werden die verschiedenen Listendarstellungen betrachtet, die entweder die Auswahl eines oder mehrerer Objekte erlauben, oder aber die auf die Anzeige- bzw. Editieransicht eines Objekts verweisen.

Die hier abgebildeten Elemente lassen sich, wie es in Web-Informationssystemen üblich ist bzw. sein sollte, über die Verwendung von CSS in ihrem Aussehen verändern und anpassen. Nähere Erläuterungen, welche Schritte in welchen Dateien dazu notwendig sind, lassen sich Abschnitt 7.4 entnehmen.

Darstellung von Überschriften und Freitext

Im generierten System werden die Überschriften, die zur Strukturierung des Inhalts eingesetzt werden, wie in Abbildung 7.15 dargestellt. Hierbei handelt es sich um die Standarddarstellung von Überschriften, welche etwas größer als der normale Text und in fettem Schriftstil dargestellt werden. Potenziell eingebundene Werte von Attributen werden ebenfalls in der Formatierung der Überschrift angezeigt.

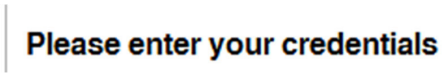


Abbildung 7.15.: Generierte Darstellung für Überschriften (`head`)

Textabsätze, die in der Seitenbeschreibungssprache über das Element `text` spezifiziert wurden, sind im generierten System in der Art dargestellt, wie in Abbildung 7.16 gezeigt wird. Die Textdarstellung erfolgt in normalem Text. Auch hier werden Attributwerte direkt in den Text integriert, ohne dass hier ein Unterschied zum statisch deklarierten Text erkennbar wäre.

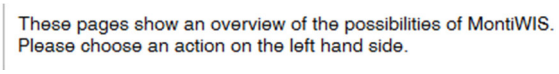


Abbildung 7.16.: Generierte Darstellung für Freitext (`text`)

Darstellung von Tabellenzellen je nach Datentyp

Der Inhalt von Tabellen, die zur Darstellung von statischem Inhalt aber vor allem auch für die Ein- und Ausgabe von Parameterattributen verwendet werden, unterscheidet sich in der

Darstellung insbesondere durch die Art der Tabellenzelle und die Art des vordefinierten Datentyps, der ausgegeben wird bzw. dessen Wert verändert werden kann.

Tabellen allgemein werden durch eine dünne, graue Umrandung dargestellt, wie beispielsweise in Abbildung 7.17 zu sehen ist. Dort ist auch dargestellt, wie ein Tabellenkopf in der Standardansicht angezeigt wird. Der Inhalt einer Zelle der Kopfzeile wird fett und zentriert formatiert. Potenziell ausgegebene Attributwerte werden ebenfalls in dieser Formatierung dargestellt.

Als Nächstes wird die Darstellung der Tabellenzellen erläutert, die für die Eingabe und Ausgabe eines bestimmten vordefinierten Datentyps generiert werden. Eine Zusammenfassung der Eingabefelder ist bereits in Tabelle 6.7 aufgelistet worden.

Abbildung 7.17.: Generierte Darstellung für Tabellenkopfzeilen

`String` Die Eingabefelder für Attribute, die vom vordefinierten Datentypen `String` sind, entsprechen denen, die in Abbildung 7.18 (a) dargestellt sind. Für die Eingabe wird ein einfaches Eingabefeld in HTML verwendet. Da hier beliebiger Text eingegeben werden kann, erfolgt keine Validierung der Eingabe hinsichtlich eines bestimmten Formats. Die Ausgabe eines `String`s geschieht durch einfaches Einbinden des Wertes, wie in Teil (b) gezeigt.

Abbildung 7.18.: Generierte Eingabe- und Ausgabedarstellung für `String`

`Text` Wenn ein Attribut einer Klasse den vordefinierten Datentyp `Text` besitzt, so wird ein HTML-Textfeld wie in Abbildung 7.19 (a) zu sehen, generiert. Ein solches Feld kann nahezu beliebig in der Größe verändert werden und daher eine größere Menge Text aufnehmen, als dies bei einem einzeiligen Textfeld der Fall wäre. Da der Inhalt eines `Text`-Attributs keiner besonderen Syntax genügen muss, findet auch hier keine Validierung der Eingabe hinsichtlich eines bestimmten Schemas statt. In der Ausgabeansicht eines solchen Felds wird der enthaltene Text eingebunden, ohne in spezieller Weise formatiert zu werden, wie dies in Teil (b) der Abbildung dargestellt ist.

Abbildung 7.19.: Generierte Eingabe- und Ausgabedarstellung für `Text`

`Date` Für Attribute im Klassendiagramm, die vom Datentyp `Date` sind, wird im Gegensatz zu den beiden Textdatentypen nicht lediglich ein Texteingabefeld generiert, sondern im Browser zusätzlich eine Eingabehilfe in Form eines *Datepickers* angeboten. Dieser ist in Abbildung 7.20 (a) dargestellt. Beim Klick auf das Texteingabefeld für das Datum öffnet sich das dargestellte Fenster. Darüber kann ein Nutzer bequem das gewünschte Datum auswählen, indem das entsprechende Feld in der Kalenderansicht angeklickt wird. Über die Pfeile in der Kopfzeile dieses Fensters lassen sich die Monate wechseln. Nach Auswahl des gewünschten Datums erscheint es in der Form im Textfeld, die auf Serverseite problemlos verarbeitet werden kann. Da es sich bei einem Datum um einen speziellen Datentyp handelt, der ein bestimmtes Format verlangt, um korrekt gelesen und interpretiert werden zu können, findet bereits bei Verlassen des Textfelds eine entsprechende Validierung des Wertes statt. Sollte der Feldinhalt nicht der korrekten Syntax folgen, wird der Nutzer durch Einblendung eines Hinweises darauf aufmerksam gemacht. Erst nach Korrektur des Feldes lässt sich die Seite zum Server abschicken, vorher ist dies nicht möglich. Die Darstellung in der Ausgabeansicht ist in dem rechten Teil (b) der Abbildung dargestellt. Hierbei wird das Datum in einem Standardformat ausgegeben.

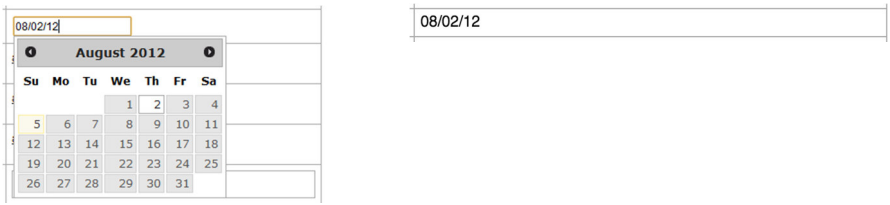


Abbildung 7.20.: Generierte Eingabe- und Ausgabedarstellung für `Date`

`Integer` Der Datentyp `Integer`, der ebenfalls als vordefinierter Datentyp im Domänendatenmodell als Attribut vorkommen kann, repräsentiert einen ganzzahligen Wert, wie er auch in einer Vielzahl von Programmiersprachen vorkommt. Das zugehörige Eingabefeld ist in Abbildung 7.21 (a) zu sehen. Da auf Serverseite sichergestellt sein muss, dass es sich tatsächlich um eine ganze Zahl handelt, wird auch hier bereits auf Clientseite geprüft, ob der Inhalt des Felds syntaktisch der Anforderung entspricht. Sollte dies nicht der Fall sein, wird der Nutzer entsprechend darauf hingewiesen und das Absenden des Formulars verhindert. Der Wert eines `Integers` wird in der Anzeigeansicht als Zahl einfach ausgegeben, wie in Teil (b) der Abbildung zu sehen ist.

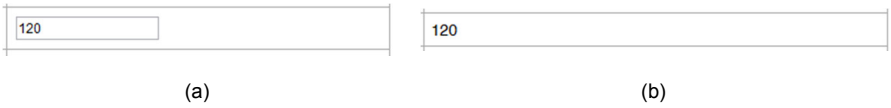


Abbildung 7.21.: Generierte Eingabe- und Ausgabedarstellung für `Integer`

`Float` Sehr ähnlich zum Datentyp `Integer` verhält es sich beim Datentyp `Float`, der eine Fließkommazahl repräsentiert. Auch hier fungiert ein einfaches Textfeld als Formularelement für die Eingabe des Wertes, und auch hier wird die Eingabe hinsichtlich syntaktischer Gesichtspunkte vor Absenden (bzw. bereits beim Verlassen des Felds) validiert. Abbildung 7.22 (a) zeigt das Eingabeelement für einen `Float`, Teil (b) zeigt, wie der Wert in der Anzeigeansicht dargestellt wird.

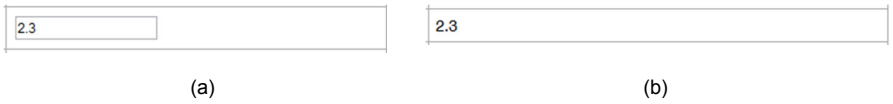


Abbildung 7.22.: Generierte Eingabe- und Ausgabedarstellung für `Float`

`Boolean` Wenn ein Attribut einer Klasse im Domänendatenmodell den Datentyp `Boolean` besitzt, repräsentiert dieses einen Wahrheitswert. Typische Anzeigemöglichkeiten für solch einen Wert könnten Dropdown-Auswahlfelder sein, die genau eine Auswahl erlauben, oder auch eine Checkbox, die beim Wert `wahr` angehakt ist, im anderen Fall entsprechend nicht. In MontiWIS wurde jedoch entschieden, die Eingabe über die in Abbildung 7.23 gezeigten Elemente zu realisieren. Hierbei dient ein Paar von Radiobuttons zur Auswahl, wobei einer dieser Buttons zur Auswahl des Wertes `wahr` (in der Abbildung über *Yes* repräsentiert), der andere für den Wert `falsch` (in der Abbildung entsprechend über *No* dargestellt) steht. Durch die Verwendung einer Gruppe (oder eines Paares) von Radiobuttons ist sichergestellt, dass jeweils nur einer der Werte ausgewählt werden kann. Die Darstellung in der Anzeigeansicht geschieht über Ausgabe von `true` im Falle von *wahr* und `false` im Falle von *falsch*.

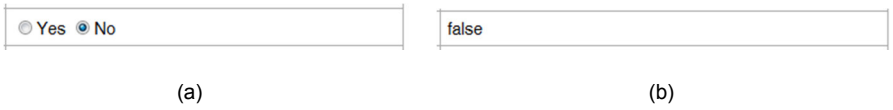


Abbildung 7.23.: Generierte Eingabe- und Ausgabedarstellung für `Boolean`

`Email` Der vordefinierte Datentyp `Email` stellt eine E-Mail-Adresse dar, wie sie sehr häufig in Web-Informationssystemen, z. B. bei der Verarbeitung von Nutzer- oder Personendaten allgemein als primäre Kontaktmöglichkeit verwendet wird. Die Eingabe einer Emailadresse erfolgt, wie in Abbildung 7.24 (a) dargestellt, ebenfalls über ein einfaches Texteingabefeld. Obwohl auf Serverseite kein spezieller Datentyp für die Speicherung einer E-Mail-Adresse verwendet wird, erfolgt dennoch eine Validierung der Syntax des eingegebenen Wertes bei Verlassen und vor Absenden des Formulars. Dies stellt sicher, dass bei tatsächlicher Verwendung der E-Mail-Adresse (z. B. für die Benachrichtigung) ein zumindest syntaktisch korrektes Datum verwendet werden kann (ohne sicherstellen zu können, dass die Adresse tatsächlich existiert). Die Ausgabe einer `Email` erfolgt durch einfache Ausgabe des Attributwertes, wie in Teil (b) der Abbildung dargestellt ist.

Abbildung 7.24.: Generierte Eingabe- und Ausgabedarstellung für `Email`

`Password` Der vordefinierte Datentyp `Password` impliziert ein etwas komplexeres Verhalten aufgrund seiner speziellen Bedeutung in einem Web-Informationssystem. Er repräsentiert ein eingegebenes Datum, welches zur sicheren Authentifizierung eines Nutzers dient. Die Eingabe des Passworts erfolgt, wie allgemein üblich, über die Verwendung eines Passwortfelds wie in Abbildung 7.25 (a) dargestellt. Dies verhindert, dass potenzielle Beobachter bei der Festlegung den eingegebenen Wert mitlesen können und damit in Besitz dieses eigentlich geheimen Datums gelangen. Bei der „Ausgabe“ in der Anzeigeansicht werden lediglich die in Teil (b) der Abbildung gezeigten *Bullets* dargestellt, die das Passwort repräsentieren. Da nicht der Klartext, sondern lediglich ein Hashwert des Passworts gespeichert wird, über den beim Login die Korrektheit der Eingabe überprüft wird, könnte eine Ausgabe des Passworts ohnehin nicht erfolgen und ist auch nicht sinnvoll. Sollte ein Objekt (z. B. eine `Person`) editiert werden, so wird anstelle des Passworts lediglich ein leeres Feld dargestellt, über das ein neues Passwort gesetzt werden kann. Sollte hier keine Eingabe erfolgen, so bleibt das bisherige Passwort erhalten. Dies wird auf Serverseite entsprechend sichergestellt.

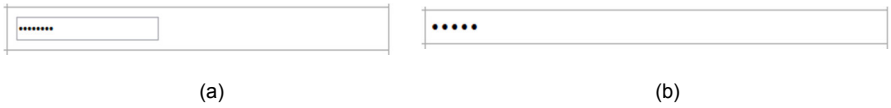


Abbildung 7.25.: Generierte Eingabe- und Ausgabedarstellung für `Password`

Assoziationen

Anders als Attribute, die als vordefinierter Datentyp spezifiziert wurden, sind Attribute zu behandeln, die aus *Assoziationen* hervorgegangen sind. Diese stellen Verbindungen zu existierenden, anderen Objekten dar und müssen entsprechend dargestellt werden. Abbildung 7.26 (a) zeigt, wie eine Assoziation in der Editieransicht dargestellt wird. Für jedes bereits assoziierte Objekt existiert ein Aufzählungspunkt. Es werden jeweils die Attribute dargestellt, die entweder in der Seitensprache in Klammern angegeben wurden, oder aber im Domänendatenmodell über den Stereotyp «`inList`» annotiert wurden. Sollte über keine dieser Alternativen eine Menge von Attributen gewählt worden sein, wird zur eindeutigen Unterscheidbarkeit die `id` des Objekts angezeigt. Jedes dieser angezeigten Objekte besitzt zusätzlich einen Link, der zum Löschen der jeweiligen Assoziation dient. Für jede Assoziation existiert ein Link, der zum Hinzufügen (im Falle von mengenwertigen Assoziationen) bzw. Setzen (im Fall von einfachen Assoziationen) dient. Die Beschriftung ergibt sich aus dem Verb *add* gefolgt vom Namen der Klasse, die jeweils referenziert wird (`Institution` im Beispiel). Ein Klick auf diesen Link führt zum Öffnen eines Fensters, wie es in Teil (b) der Abbildung dargestellt ist. Dies enthält

eine Liste aller existierenden Objekte des Zieldatentyps. Die angezeigten Spalten entsprechen den Attributen, die entweder in der Seitenbeschreibung oder im Klassendiagramm per Stereotyp spezifiziert wurden. Die Tabelle bietet sämtliche Funktionalität, wie sie generell in MontiWIS bei Listendarstellungen angeboten wird. Die Anzahl der angezeigten Elemente kann über die Dropdown-Box angepasst werden, über Knöpfe kann durch die verschiedenen Seiten der aufgelisteten Objekte navigiert werden und nach dem Inhalt sämtlicher dargestellten Attribute kann durch Eingabe eines Wertes in das Suchfeld gefiltert werden. Über die Betätigung des Links *[add]* kann das jeweilige Objekt hinzugefügt werden bzw., im Fall einer Assoziation mit einfacher Kardinalität, das bisherige Objekt ersetzt werden. Gespeichert werden die Assoziationen nach Absenden der gesamten Seite. Weder Löschen noch Zuordnen von Objekten haben vorher eine Auswirkung.

Die Ausgabe der zugeordneten Objekte erfolgt, wie in Teil (c) der Abbildung dargestellt, ebenfalls über Aufzählungen und mit den spezifizierten Attributen. Hier fehlen lediglich die Verweise, die sowohl zum Hinzufügen bzw. Setzen von neuen Assoziationen als auch zum Löschen einer Assoziation in der Editieransicht vorhanden sind.

Teil (d) der Abbildung zeigt, wie die Spezifikation des *größer- und kleiner-Symbols* (<>) im generierten System umgesetzt wird. Der dargestellte Link führt zum Öffnen eines Pop-up-Fensters, welches die standardmäßige `View`-Seite einbindet und damit eine Vorschauansicht auf sämtliche Attribute des referenzierten Objekts bietet.

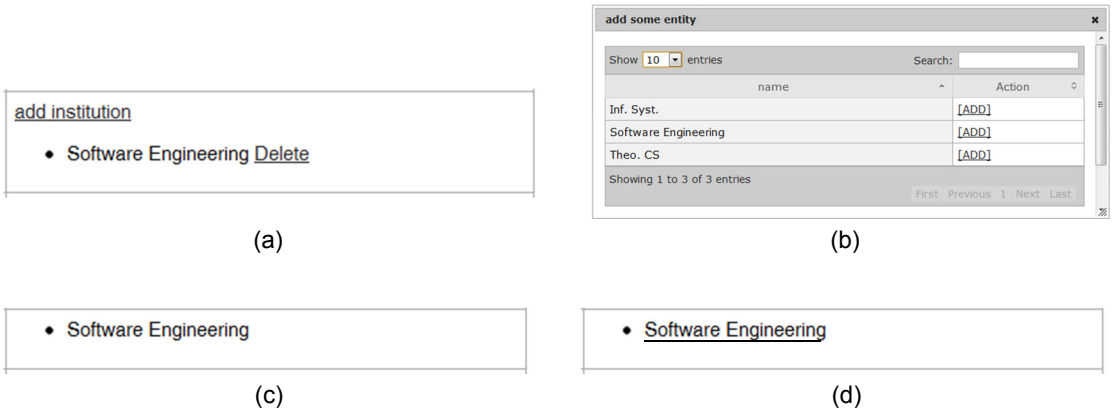


Abbildung 7.26.: Generierte Eingabe- und Ausgabedarstellung von Assoziationen

Kompositionen

Im Gegensatz zu Assoziationen, bei denen existierende Objekte hinzugefügt werden, ist die Existenz einer Komponente explizit an den Lebenszyklus des Kompositums gekoppelt. Entsprechend werden diese auch in MontiWIS dargestellt. Wie Abbildung 7.27 zeigt, werden die Komponenten direkt innerhalb der Darstellung des Kompositums erstellt und können dort auch wieder gelöscht werden. Hierzu existiert ein Link *add new*, über den eine neue Komponente angelegt werden kann. Die Eingabefelder für das referenzierte Objekt werden direkt innerhalb einer Kindtabelle erzeugt. Das Verhalten dieser Eingabefelder entspricht

dem der jeweiligen vordefinierten Datentypen in einem normalen Objekt. Jede Komponente besitzt zusätzlich einen *remove*-Link zum Löschen des jeweiligen Objekts.

In einer Anzeigeansicht werden die Komponenten ebenfalls in einer Kindtabelle dargestellt, wie Teil (b) der Abbildung zeigt.

Abbildung 7.27.: Generierte Eingabe- und Ausgabedarstellung Kompositionen

Darstellung von Mengen von Objekten

Generell ist die Darstellung von mehreren Objekten in Listenform in von MontiWIS relativ ähnlich, wie die Abbildungen 7.28 - 7.31 zeigen. Daher soll zunächst der generelle Aufbau von Listendarstellungen erläutert werden, anschließend wird auf die jeweiligen Besonderheiten der einzelnen Einsatzszenarien eingegangen.

Generell werden die Mengen von Objekten in Tabellenform angezeigt, wie sie in den folgenden Abbildungen dargestellt sind. Diese Tabellen besitzen jedoch einige nützliche Zusatzfunktionen, die die Verwendung wesentlich vereinfachen und für den Nutzer hilfreich sind.

Gemein ist den verschiedenen Ansichten, dass sämtliche in der `Collection` (übergeben als Seitenparameter) enthaltenen Objekte in der Liste dargestellt werden. Die Funktionalität der Tabelle hilft lediglich, diese Menge verwaltbar und zugreifbar zu machen. Dazu kann in der Tabelle oben links festgelegt werden, wie viele Objekte gleichzeitig dargestellt werden sollen. Durch die verschiedenen Kontingente der Objekte kann über die Schaltflächen unten rechts in der Tabelle navigiert werden. Hierüber können die nächsten *n* Objekte aufgerufen werden, ebenso wie relativ wahlfrei die verschiedenen Seiten angewählt werden. Über das *Search*-Feld oben rechts in der Ecke der Tabelle lassen sich die angezeigten Tabellenzellen filtern, so dass nur diejenigen Objekte angezeigt werden, deren Text mit dem im Suchfeld eingegebenen Wert übereinstimmt. Die Auswahl, welche Attribute in Spalten dargestellt werden, erfolgt über ihre Spezifikation in der Seitenbeschreibung (in Klammern) oder über die Stereotypen an bestimmten Attributen im Klassendiagramm. Die Spalten lassen sich, wie in der Abbildung durch vertikale Pfeile am Tabellenkopf angedeutet, auf- bzw. absteigend sortieren.

Über die beschriebenen Mechanismen lässt sich bequem auf die dargestellten Objekte zugreifen, weswegen sie auch in den nachfolgend beschriebenen Anwendungsfällen zur Verfügung stehen. Auch in den Fenstern, die zur Auswahl von zu assoziierenden Objekten dienen, wird die gleiche Art Tabelle verwendet.

Auswahl von Objekten

Die Auswahl von Objekten dient dazu, aus einer Menge von Objekten eines oder mehrere zu bestimmen. Als Grundkonstrukt dient hierbei die Tabelle, die soeben detailliert beschrieben wurde.

Für die Auswahl *eines* Elements (resultierend aus der Verwendung eines `chooseOne`-Elements in der Seitenbeschreibung) werden zusätzlich zu den potenziell spezifizierten Attributen *Radiobuttons* in einer Spalte hinzugefügt, die die Auswahl exakt eines Elements fordern. Die resultierende Tabelle ist in Abbildung 7.28 zu sehen.

Abbildung 7.28.: Generierte Listendarstellung für die Auswahl eines Objekts

Sehr ähnlich ist die Tabelle, die einem Nutzer zur Auswahl mehrerer Elemente präsentiert wird (resultierend aus der Verwendung von `choose` in der Seitenbeschreibung). Wie in Abbildung 7.29 zu sehen, werden anstelle der Radiobuttons hier *Checkboxen* verwendet. Diese erlauben die Auswahl einer beliebigen Anzahl von Objekten.

Abbildung 7.29.: Generierte Listendarstellung für die Auswahl mehrerer Objekte

Listendarstellung von Objekten

Neben den Ansichten zur Auswahl von Objekten existieren zusätzlich zwei Ansichten, die lediglich die Objekte auflisten und, je nach Art der gewünschten Darstellung, bestimmte Aktionen für jedes einzelne Objekt anbieten. Die nun beschriebenen Darstellungen werden generiert, wenn das Element `list` in der Seitenbeschreibung verwendet wurde. Die Ansichten variieren hinsichtlich der Interaktionsmöglichkeiten, werden aber zusätzlich eingeschränkt

aufgrund der Berechtigungen eines Nutzers. In beiden Ansichten wird auf der rechten Seite der Tabelle eine separate Spalte erzeugt, die die verfügbaren Aktionen als Links enthält.

Die erste beschriebene Darstellung in Abbildung 7.30 wird durch die Verwendung des *kleiner*-Symbols (<) erreicht. Hierbei wird in der Aktionsspalte lediglich der *show* Link erzeugt, der zur standardmäßig generierten `View`-Ansicht des jeweiligen Objekts führt. Da die Ansicht eines Objekts implizit für jede Klasse erlaubt ist, müsste die Auswahl, welche Objekte hier zur Verfügung gestellt werden, explizit innerhalb des Ablaufs (der ja letztlich für die Übergabe der gewünschten Objekte als Menge zuständig ist) geregelt werden.

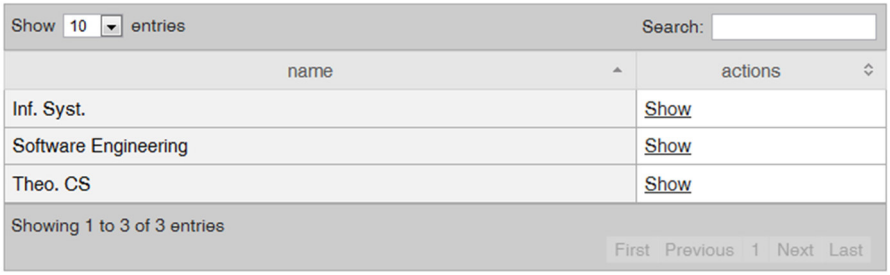


Abbildung 7.30.: Generierte Listendarstellung im Anzeigemodus

Die zweite Darstellung aus Abbildung 7.31 wird generiert, wenn das *größer*-Zeichen (>) genutzt wurde. Für den Fall, dass der aufrufende Nutzer die entsprechenden Berechtigungen für den dargestellten Objekttyp besitzt, werden zusätzlich zum *Show*-Link die Verweise zum Editieren (*Edit*) und Löschen (*Delete*) angezeigt. Diese führen entweder zur standardmäßig generierten `Edit`-Seite bzw. zum direkten Löschen des Objekts.

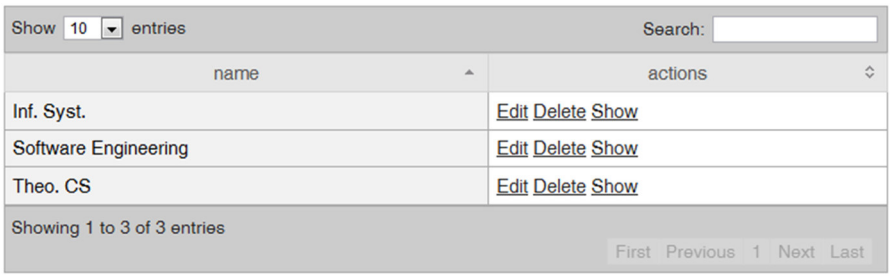


Abbildung 7.31.: Generierte Listendarstellung im Editiermodus

Auf eine separate Abbildung wird im Falle der Darstellung eines `listOnly`-Elements einer Seite verzichtet. Die Darstellung erfolgt analog zu den anderen Elementen, wobei jedoch keinerlei Link dargestellt wird, sondern lediglich die Objekte aufgelistet werden.

Nach der Beschreibung, wie die einzelnen Sprachelemente einer Seite im generierten System umgesetzt werden, soll nun genauer auf den Generierungsprozess selbst eingegangen werden.

7.3. Generierungsprozess

Nach den eher anwendungsorientierten Aspekten der Sprachentwicklung werden nun die eher entwicklernahen Themen behandelt. Zunächst wird in Abschnitt 7.3.1 auf die implementierten Kontextbedingungen eingegangen, die helfen, korrekte Modelle zu erstellen und im Fehlerfall auf problematische Modellteile hinweisen. Als Nächstes wird in Abschnitt 7.3.2 die Zwischenstruktur erläutert, in die die Seitenmodelle überführt werden. Anschließend werden in Abschnitt 7.3.3 die Schritte beschrieben, durch die Modelle per Visitor verarbeitet werden und wie dabei die Zwischenstruktur aufgebaut wird, die als Eingabe für nachfolgende Schritte dient. Die Veränderungen, die an der erzeugten Objektstruktur vorgenommen werden, finden ihre Erläuterung in Abschnitt 7.3.4, worauf schlussendlich in Abschnitt 7.3.5 der eigentliche Generierungsprozess inklusive der verwendeten Templates beschrieben wird.

7.3.1. Kontextbedingungen

Der erste Schritt, der bei der Verarbeitung eines Seitenmodells ausgeführt wird, ist die Konsistenzprüfung der Modelle. Hierzu wird insbesondere die Infrastruktur verwendet, die in der Dissertation von Steven Völkel entwickelt wurde [Völ11]. Die grundlegenden Mechanismen für die Konsistenzprüfung sind in Abschnitt 4.3 beschrieben.

Für die Seitensprache existieren mehrere Arten von Kontextbedingungen. Zum einen werden bestimmte Dinge innerhalb eines einzelnen Modells geprüft, zum anderen wird überprüft, ob die Referenzen zwischen verschiedenen Modellen korrekt sind. In der Regel werden die Bedingungen jedoch gemeinsam geprüft, weswegen an dieser Stelle keine explizite Unterscheidung diesbezüglich getroffen wird.

Verwendete Symboltabelleneinträge

Bei den Konsistenzprüfungen der Seitensprache wird auf die Symboltabelleninfrastruktur von MontiCore gesetzt. Für die Funktionsweise werden Symboltabelleneinträge benötigt, die (a) Einträge anderer Sprachen referenzieren und (b) an anderer Stelle von anderen Einträgen referenziert werden (vgl. hierzu Abschnitt 9.3.1). Die verwendeten Einträge und welche Teile der Seite sie repräsentieren, soll nur kurz erläutert werden. Abbildung 7.32 gibt einen Überblick über die Einträge und deren Verwendung.

Der Eintrag `PageEntry` repräsentiert eine Seite in der Symboltabelle. Einzig der Name der Seite wird gespeichert. Darüber hinaus werden die `PageParameterEntry`-Einträge dort gespeichert (mengenwertige Assoziation `parameters`). Referenziert werden diese Seiten in der Klasse `ActivityOrPageToAppReferenceAdapter`, wo eine Seite innerhalb der Applikationsbeschreibungssprache sowohl im Menü als auch bei der Berechtigungsvergabe referenziert werden kann und für die folglich ein Adapter zu den dortigen Symboltabelleneinträgen existiert. Für eine Erläuterung dieses Adapters siehe Abschnitt 9.3.1.

Über den Symboltabelleneintrag `PageParameterEntry` werden die Artefakte abgebildet, die als Parameter einer Seite übergeben werden. Diese leiten sich von `TypeEntry` ab, welcher als abstrakte Klasse bereits Attribute wie `name`, `packageName` und `typeParameters` bereitstellt und damit, analog zu Klassendiagrammeinträgen, die Informationen enthält, die auch für

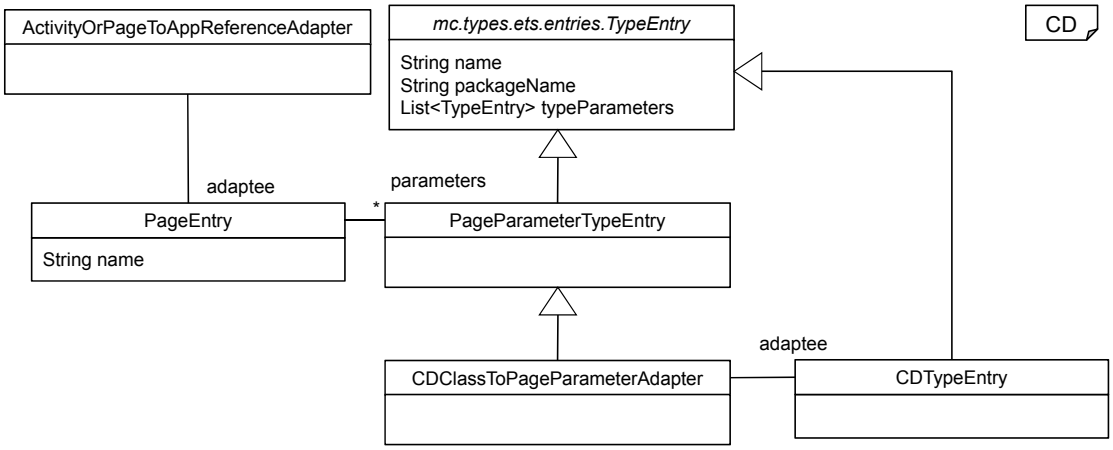


Abbildung 7.32.: Übersicht über Symboltabelleneinträge für Seiten

die Parameter einer Seite relevant sind. Da ein Seitenparameter eine Klasse referenziert, wurde mit der Klasse `CDClassToPageParameterAdapter` ein Adapter implementiert, der nach korrekter Auflösung auf die jeweilige, adaptierte Klasse verweist (`adaptee`). Darüber sind die Strukturinformationen der Klasse in diesem Adapter verfügbar, so dass die für Kontextbedingungen notwendigen Abfragen möglich sind.

Existenz von Parameterreferenzen als Parameter der Seite

Wenn innerhalb einer Seite auf einen Parameter verwiesen wird, so ist in jedem Fall notwendig, dass dieser Parameter auch in der Liste der Parameter der jeweiligen Seite existiert. Dies wird im Rahmen von konkreteren Überprüfungen implizit mit getestet, beispielsweise während überprüft wird, ob ein spezielles Attribut der referenzierten Klasse existiert.

Korrekte Referenz des Parametertyps im Klassendiagramm

Wie bereits bei der Beschreibung der Seitenparameter in Abschnitt 7.2 erwähnt, können die Parameter entweder eine Klasse innerhalb des Domänendatenmodells oder aber eine Klasse im Klassendiagramm der vordefinierten Datentypen referenzieren. Es sollte selbstverständlich auch überprüft werden, ob die Typen, die in der Parameterliste einer Seite angegeben werden, auch tatsächlich existieren. Wie in Abschnitt 4.3 erläutert, werden beim Aufbau der Symboltabelle alle registrierten `Qualifier` ausgeführt, die die Vollqualifizierung der in der Symboltabelle vorhandenen Einträge vornehmen. Hierbei werden auch die Zieleinträge der Adapter aufgelöst, wobei ein Fehler auftritt, sofern der als Seitenparameter angegebene Name (in Kombination mit den angegebenen importierten Paketnamen) nicht zu einer Qualifizierung der Einträge führt. Durch die Ausführung des `CDClassToPageParameterQualifier` werden derartige Fehler im Modell abgefangen und eine entsprechende Warnung an den Autoren eines Modells ausgegeben.

Korrekte Referenz von Parameterattributen innerhalb einer Seite

Als logische Konsequenz nach der Überprüfung, ob ein Parametertyp auch in einem Klassendiagramm existiert, ist es, sicherzustellen, dass auch die Attribute, die innerhalb einer Seite verwendet werden, in der referenzierten Klasse vorhanden sind. So wird über die Kontextbedingung `AttributeReferenceExistsChecker` festgestellt, ob das referenzierte Attribut auch tatsächlich Teil der Klasse ist, die über den Adapter aufgelöst wurde. Sollte dies nicht der Fall sein, wird eine entsprechende Warnung ausgegeben.

Korrekte Referenz eines Parameters bei der Ausgabe aller Attribute

Neben der Verwendung von Attributen in der Seite sollte ebenso überprüft werden, ob die verwendeten Parameter und Attribute innerhalb eines `allAttributes`-Objekts korrekt verwendet werden, d. h., dass sowohl der Parameter als auch die Attribute existieren. Dies wird über die Klasse `AllAttributesShortcutIsCorrectChecker` überprüft.

Korrekte Referenz eines Parameters bei der Listendarstellung

Da auch bei der Listenanzeige einer Menge von Objekten sowohl ein Seitenparameter (der zusätzlich auch eine `Collection` sein muss) als auch dessen Attribute referenziert werden, sind auch bei diesem Element die Existenz und korrekte Verwendung sicherzustellen. Dies geschieht für das `list`-Element innerhalb der Kontextbedingung `ListElementReferenceChecker`. Sollte die Überprüfung fehlschlagen, wird eine entsprechende Warnung ausgegeben.

Korrekte Referenz eines Parameters bei Auswahl von Objekten

Ähnlich wie bei der Listendarstellung werden auch bei der Auswahl von Objekten aus einer Menge (über die Seitenelemente `choose` bzw. `chooseOne`) sowohl mengenwertige Seitenparameter als auch Attribute der referenzierten Klasse verwendet. Die korrekte Referenz wird über die Kontextbedingungen `ChooseElementReferenceChecker` (für `choose`-Elemente) bzw. `ChooseOneElementReferenceChecker` (für `chooseOne`-Elemente) geprüft.

All diese Kontextbedingungen werden vor Durchlauf der eigentlichen Visitorbearbeitung, die nachfolgend beschrieben ist, geprüft. Die Warnungen, die bei der Überprüfung dieser Bedingungen erzeugt werden, dienen als Hinweis für den Entwickler, wo Probleme in den Modellen bestehen.

7.3.2. Aufbau der Zwischenstruktur

Wie bereits in Abschnitt 4.4 erläutert, werden in MontiWIS aufgrund der besseren Verarbeitbarkeit durch Templates und aufgrund umfangreichen Standardverhaltens, was während der Bearbeitung der Modellinformationen hinzu generiert wird, eigene Zwischenstruktur-Klassen verwendet, die alle notwendigen Informationen in leicht zugreifbarer Form beinhalten.

Ein Überblick über die Zwischenstruktur der Seitenbeschreibungssprache ist in Abbildung 7.33 dargestellt.

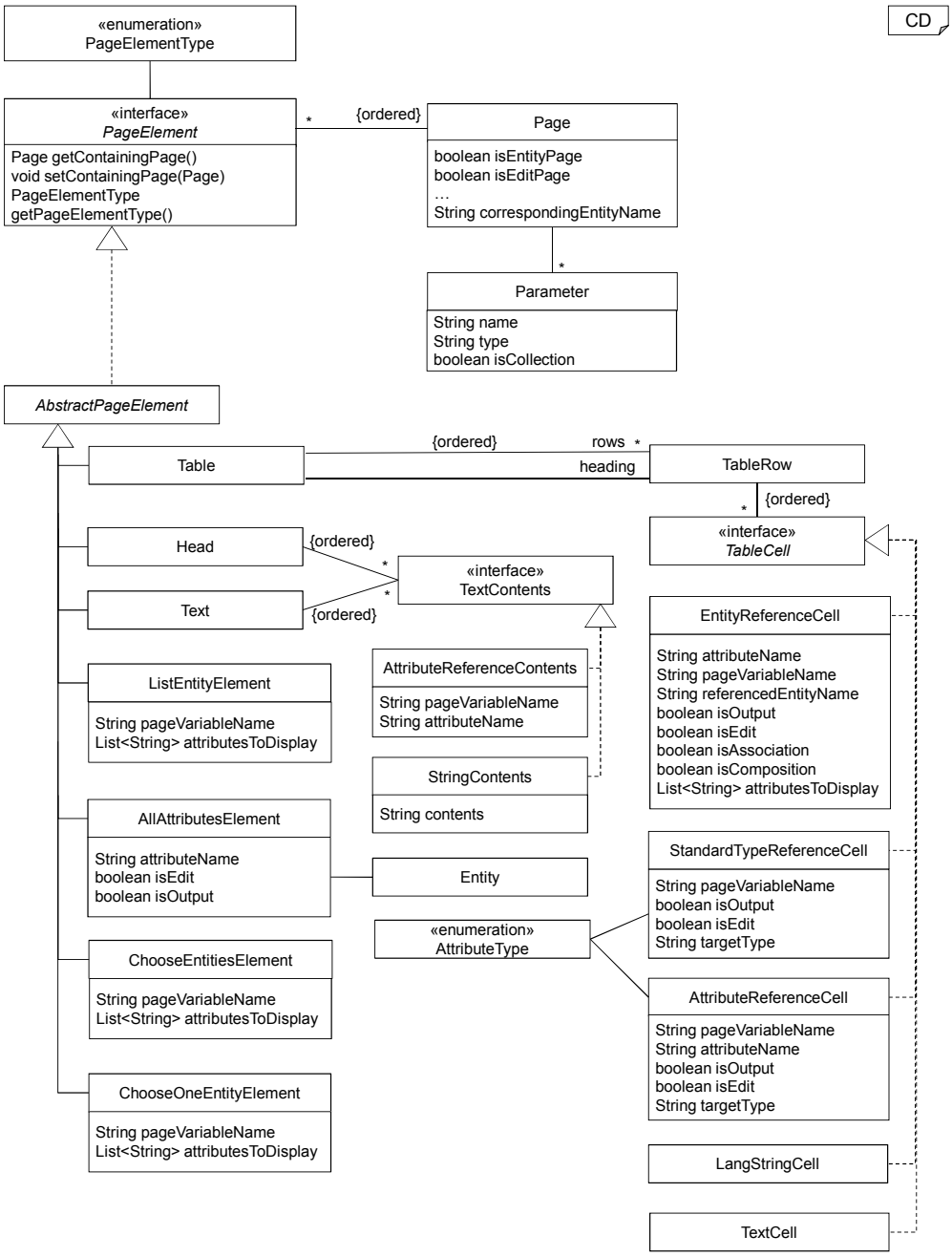


Abbildung 7.33.: Übersicht über die Zwischenstruktur für die Seitenbeschreibungssprache

Page

Die zentrale Klasse in der Zwischenstruktur für die Seitenbeschreibung ist die Klasse `Page`. Sie repräsentiert eine vollständige Seite mit einer beliebigen Anzahl von Seitenelementen.

Eine Seite kann für eine Klasse generiert worden sein und damit Standardfunktionalität dafür bereitstellen, was über den Schalter `isEntityPage` ausgedrückt wird. Die Art der Seite, die aktuell gerade generiert wird, lässt sich über die Attribute `isNewPage`, `isEditPage`, `isViewPage` und `isListPage` ermitteln (dargestellt über entsprechende Attribute in Abbildung 7.33). Für den Fall, dass eine Seite eine dieser Standardseiten ist, wird ebenfalls gespeichert, für welche `Entity` sie generiert wurde. Hier wird lediglich der Name der Klasse gespeichert, worüber das entsprechende Entity im Bedarfsfall aus dem zentralen Speicher für Klassen abgerufen werden kann (Attribute `correspondingEntityName`). Für eine einfache Verarbeitung der referenzierten Attribute werden diese, zusätzlich zur Speicherung als Teil einer Tabelle, doppelt und separat in der Seite abgelegt. Dies ist insbesondere für die Erzeugung von `PageEntity`- und `PageEntityInitializer`-Klassen (vgl. Abschnitt 7.4.3) hilfreich, wo die statische Struktur einer Seite nicht von Belang ist, sondern lediglich die Ein- bzw. Ausgabestruktur benötigt wird. Die zugehörigen `get`- und `set`-Funktionen sind der Übersichtlichkeit halber hier nicht dargestellt.

Für jede Seite werden sämtliche `Parameter` gespeichert, die ihr übergeben werden. Dies geschieht über die Klasse `Parameter`. Diese Klasse enthält Attribute für Namen (`name`) und Typ (`type`) des Parameters, sowie Attribute, um zu speichern, ob es sich um einen mengenwertigen Parameter (`isCollection`) handelt.

Neben diesen Informationen enthält eine Seite eine Liste von `PageElements`, die den eigentlichen Inhalt der Seite darstellen. Hier gibt die Reihenfolge des Auftretens der Objekte die Reihenfolge an, in der sie in der Seite definiert wurden (d. h. von oben nach unten in der Seite). `PageElement` ist dabei eine Schnittstelle, die von `AbstractPageElement` implementiert wird. Letztere enthält die Funktionalität, die allen davon abgeleiteten Seitenelementen gemein ist. Wiederum davon abgeleitet sind alle Elemente, die als Seitenelement vorkommen können. Diese werden nachfolgend kurz erläutert.

Seitenelemente über das Interface PageElement

Die Schnittstelle `PageElement` stellt lediglich Zugriff auf die Seite bereit (`get-` bzw. `setContainingPage`), deren Teil ein Seitenelement ist, und besitzt eine Methode zur Ermittlung des Seitenelementtyps (`getPageElementType`), welche in der Klasse `AbstractPageElement` implementiert werden bzw. als abstrakt an Kindklassen weiter vererbt werden. Alle Kindklassen, die `PageElement` implementieren bzw. von `AbstractPageElement` erben, können austauschbarer Teil einer Seite sein.

Das Seitenelement `AllAttributesElement` repräsentiert das äquivalente Element der Seitenbeschreibungssprache und dient dazu anzuzeigen, dass alle Attribute eines bestimmten `Entitys` ausgegeben werden sollen. Ein Objekt dieser Klasse enthält ein Attribut, das das entsprechende `Entity` enthält. Zusätzlich existiert ein Attribut, das den Namen des referenzierten Parameters enthält (`attributeName`). Notwendig ist ebenfalls zu speichern, ob die Attribute in editierbarer Form oder lediglich zur Ausgabe dargestellt werden sollen (`isEdit` bzw. `isOutput`). Diese Informationen sind ausreichend, um dieses Seitenelement in einem späteren Rekonstruktionsschritt durch die einzelnen Attribute zu ersetzen (vgl. hierzu Abschnitt 7.3.4).

Ein Objekt der Klasse `ChooseOneEntityElement` repräsentiert das Element, welches zur Auswahl eines Elements aus einer Menge mehrerer Objekte dient. Darin werden lediglich der Name des Seitenparameters (`pageVariableName`) sowie die Liste der darzustellenden Attribute gespeichert (`attributesToDisplay`). Diese Attribute werden als `String` bzw. als Liste von `String`s gespeichert, was genügt, um daraus die entsprechenden Seitenteile zu generieren.

Ähnlich ist die Klasse `ChooseEntitiesElement` aufgebaut, die letztlich die gleichen Informationen benötigt, aber dennoch leicht anderes Verhalten repräsentiert.

Auch die Klasse `ListEntityElement` beinhaltet Informationen für die Darstellung von sehr ähnlichen Darstellungen wie die zwei kürzlich beschriebenen Elemente. Daher ist es strukturell auch sehr ähnlich zu ihnen.

Die beiden Seitenelemente `Head` und `Text` sind sehr ähnlich aufgebaut. Sie beinhalten eine Liste von `TextContents`-Elementen. Diese Schnittstelle wird implementiert von den beiden Klassen `AttributeReferenceContents` und `StringContents`. Erstere repräsentiert die Ausgabe eines Attributwertes, und beinhaltet folglich Parameter- und Attributnamen, um die Ausgabe entsprechend generieren zu können. Letztere Klasse beinhaltet jedoch lediglich den Textinhalt, der ausgegeben werden soll.

Als letztes Seitenelement ist die Klasse `Table` zu nennen, welches eine Tabelle und deren Struktur beschreibt. Da sich diese als relativ komplex darstellt, wird etwas detaillierter im folgenden Abschnitt darauf eingegangen.

Tabellenstruktur

Das Seitenelement `Table`, welches die Schnittstelle `PageElement` implementiert, stellt eine Tabelle dar, die aus verschiedenartigen Tabellenzellen bestehen kann. Diese werden über die Schnittstelle `TableCell` in der Art zusammengefasst, dass die austauschbar innerhalb der Tabelle verwendet werden können. Tabellenzellen werden in Objekten der Klasse `TableRow` zusammengefasst, worin sie geordnet (von links nach rechts in der Reihenfolge ihres Auftretens) in Listenform gespeichert werden. Eine `TableRow` kann in der Tabelle einerseits als Überschrift gesetzt (`heading`) oder in einer Liste gespeichert werden, die die einzelnen Zeilen repräsentieren (`rows`).

Die Schnittstelle `TableCell` wird implementiert von den Klassen, die die unterschiedlichen Inhalte einer Tabellenzelle repräsentieren. Als einfachste Implementierung kann die Klasse `TextCell` angesehen werden, worin lediglich der darzustellende Text gespeichert wird. Wenig komplexer ist die Klasse `LangStringCell`, die den Schlüssel enthält, der über den `ResourceBundle`-Mechanismus zur Auflösung eines ausgelagerten `String`s dient.

Die Klasse `StandardTypeReferenceCell` dient dazu, eine Referenz auf einen Parameter zu repräsentieren, dessen Typ einer der vordefinierten Datentypen ist, also etwa ein Parameter der Form `String s` (`targetType`). Die Art und Funktion des Parameters wird über die Attribute `isEdit` bzw. `isOutput` abgebildet. Neben dem Namen des Parameters, der referenziert wird (`pageVariableName`), werden ebenfalls Typinformationen sowohl des *Zieldatentyps* (z. B. `String`) als auch der *funktionale Datentyp* (z. B. `Email` für korrekte Verarbeitung, abgebildet durch den Aufzählungstypen `AttributeType`) des Parameters gespeichert.

Ähnliche Informationen werden in Objekten der Klasse `AttributeReferenceCell` abgelegt, da hierbei auf ein bestimmtes Attribut eines Parameters referenziert wird, was in der Regel ebenfalls sowohl Ziel- als auch funktionalen Datentyp erfordert, um diese korrekt im Generator verarbeiten zu können.

Bei einer Zelle des Typs `EntityReferenceCell` werden nicht Attribute eines vordefinierten Datentyps dargestellt, sondern Attribute, die aus einer Assoziation im Klassendiagramm herrühren. Dementsprechend werden sowohl die üblichen Attribute wie der Name des referenzierten Parameters (`pageVariableName`) oder der Fakt, ob es sich um ein editierbares Feld oder lediglich die Anzeige des Attributs handelt (`isEdit` bzw. `isOutput`), als auch Informationen über den Zieldatentyp (`referencedEntityName`), was in diesem Fall eine Klasse aus dem Klassendiagramm ist, gespeichert. Da diese Klasse sowohl Referenzen in Form von Assoziationen als auch Referenzen, die als Kompositionen spezifiziert wurde, speichert, wird ebenfalls per Attribut gespeichert (`isAssociation` bzw. `isComposition`), um welche der Arten es sich dabei handelt. Schließlich wird zusätzlich abgelegt, welche Attribute bei der Darstellung der Referenz bzw. beim Hinzufügen eines neuen Objekts angezeigt werden sollen. Dies geschieht über eine `String`-Liste, was hierfür hinreichend ist.

Zwischenstrukturspeicher in PageGlobalStore

Wie in Abschnitt 4.4 erläutert, ist es in MontiWIS üblich, die Ergebnisse der Visitorbearbeitung in einer zentralen Datenstruktur abzulegen, die von beliebigen Stellen aus aufgerufen und verwendet werden kann. Auch für die Zwischenstruktur der Seitenbeschreibungssprache existiert eine solche. Die Schnittstelle `PageGlobalStore` beschreibt die Methoden, die eine implementierende Klasse besitzen muss. Im aktuellen System ist dies die Klasse `DefaultPageGlobalStore`, deren Funktionalität nun kurz beschrieben wird. Ein Überblick ist in Abbildung 7.34 dargestellt.

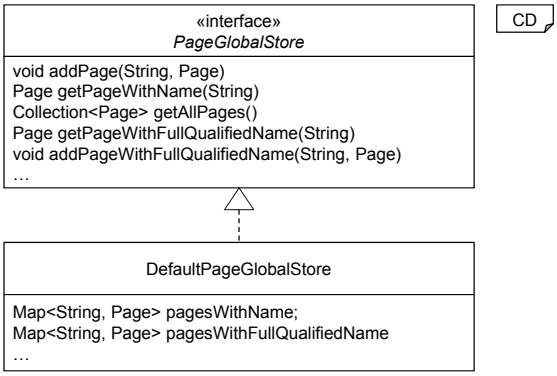


Abbildung 7.34.: Zentraler Speicher für Seiteninformationen

`Page`-Objekte können über die Methode `addPage` durch Angabe des Seitennamens und Übergabe des `Page`-Objekts selbst hinzugefügt werden. Zusätzlich können Seiten auch per vollqualifiziertem Namen abgelegt und wieder abgerufen werden (`addPageWithFullQualifiedName` bzw. `getPageWithFullQualifiedName`), was sinnvoll in einigen nachfolgenden Bearbeitungs-

schritten ist. Ferner existiert eine Methode, um alle gespeicherten Seiten abzurufen, was insbesondere beim Iterieren über alle Seiten hilfreich ist. Die Implementierung der Schnittstelle erfolgt in der Klasse `DefaultPageGlobalStore`, worin sowohl für die Ablage von `Page`-Objekten per Name als auch per vollqualifiziertem Namen eine `Map` existiert (`pagesForName` bzw. `pagesForFullQualifiedName`), wie ebenfalls in Abbildung 7.34 dargestellt ist.

7.3.3. Modellverarbeitung durch einen Visitor

Ebenso wie bei den anderen Teilsprachen, die in MontiWIS verwendet werden, wird auch die Bearbeitung der Sprachebeschreibungsmodelle über die Verwendung eines Visitors ausgeführt.

Generell werden die einzelnen Klassen des AST, wie auch bei der Bearbeitung des Klassendiagramms, über die `ownVisit`-Methoden verarbeitet. Obwohl die Reihenfolge der Bearbeitung bei der Seitenbeschreibungssprache nicht in der Art relevant ist, wie dies bei Klassendiagrammen der Fall ist (wo wichtig ist, dass die aus Klassen erzeugten Zwischenkonstrukte bei der Bearbeitung der Assoziationen bereits existieren), wird diese Strategie aus Konsistenz- und Flexibilitätsgründen dennoch verfolgt.

Jede Seite und jedes Seitenelement wird über eine Visitormethode bearbeitet, wobei für jedes Element der Seitensprache das entsprechende Äquivalent aus der Zwischenstruktur erzeugt und mit den Informationen (wie z. B. Name, Typ) aus dem Modell belegt wird. Da bei der Bearbeitung keine wesentlich bemerkenswerten Schritte erfolgen, soll an dieser Stelle nicht weiter auf den Bearbeitungsvorgang eingegangen werden. Aus dem Aufbau der Zwischenstruktur erschließt sich relativ eindeutig, woher die abgelegten Informationen stammen und wie diese belegt wurden.

Wie generell in MontiWIS üblich, werden die generierten Zwischenstruktur-Objekte in einer zentralen Datenstruktur abgelegt. Im Falle der Seitenbeschreibung ist dies die Schnittstelle `PageGlobalStore` bzw. die diese implementierende Klasse `DefaultPageGlobalStore`.

7.3.4. Rekonstruktion des Seitenaufbaus

Nach dem Aufbau der Zwischenstruktur und ihrer Speicherung in der zentralen Datenstruktur für Seitenbeschreibungen erfolgt auch bei der Verarbeitung der Seiten ein umfangreicher Rekonstruktionsschritt. Dies ist notwendig, um potenziell fehlende Informationen aus anderen Modellen nachzupflegen. Ferner werden Elemente wie `AllAttributesElement` umgebaut und durch die Elemente ersetzt, die letztlich im Generierungsschritt verarbeitet werden.

Ein Überblick über die an der Rekonstruktion der Seiten beteiligten Klassen ist in Abbildung 7.35 dargestellt. Die einzelnen Klassen und die darin ausgeführten Operationen werden nachfolgend erläutert.

Der Rekonstruktionsprozess der Seiten wird aus der Klasse `Reconstructor` (nicht im Klassendiagramm in Abbildung 7.35 dargestellt) durch den Aufruf der Methode `reconstructAllPages` der Klasse `AllPagesReconstructor` gestartet. Darin wird über alle Seiten, die im `PageGlobalStore` abgelegt sind, iteriert und jede Seite wird über die Methode `reconstructPage` des `SinglePageIterators` bearbeitet. Die Schritte, die innerhalb dieser Methode für jede Seite vorgenommen werden, sind in den folgenden Unterabschnitten erläutert.

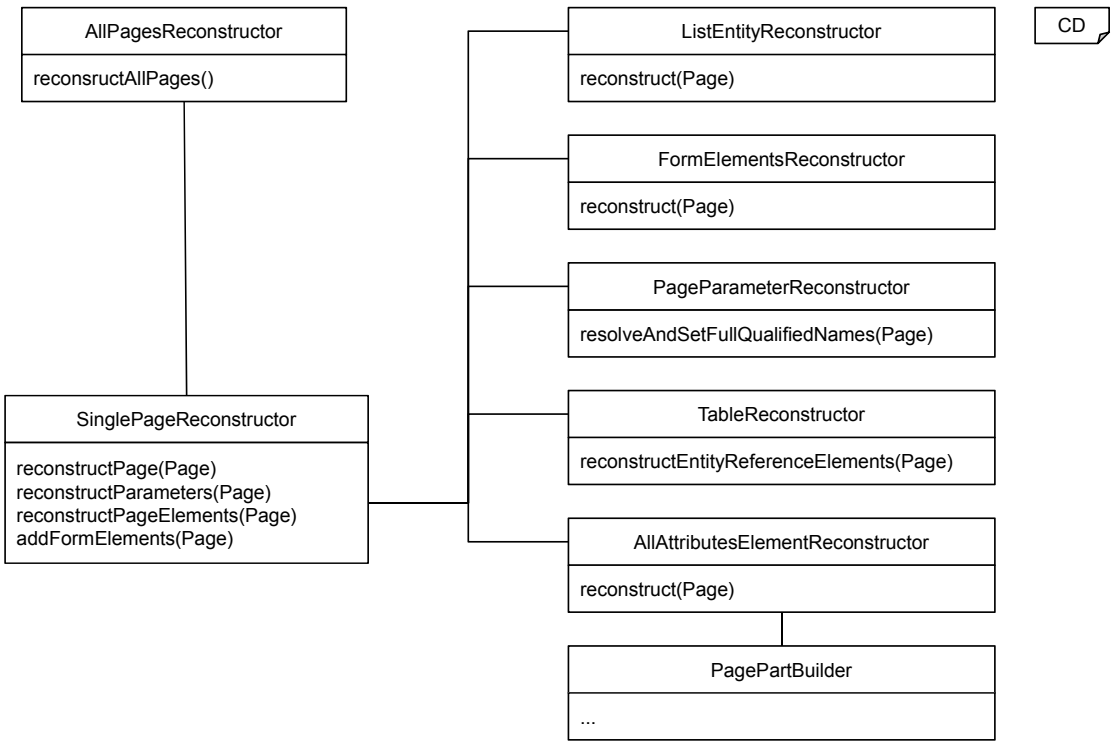


Abbildung 7.35.: Übersicht über die an der Rekonstruktion von Seiten beteiligten Klassen

Rekonstruktion der Seitenparameter

Aufgrund der Tatsache, dass zunächst sämtliche Informationen über alle im System vorhandenen `Entity`-Objekte vorliegen müssen, bevor dieser Schritt ausgeführt werden kann, wird dieser Rekonstruktionsschritt nötig. Hierbei werden in der Methode `resolveAndSetFullQualifiedName` der Klasse `PageParameterReconstructor` die vollqualifizierten Namen der Parameter gesetzt, damit sie entsprechend bei der Generierung verwendet werden können.

Erzeugung der Tabelle für die Darstellung aller Attribute

In der Methode `reconstructPageElements` der Klasse `SinglePageReconstructor` wird über sämtliche `PageElement`-Objekte der zu bearbeitenden Seite iteriert. Zunächst werden alle potenziell vorhandenen `AllAttributesElement`-Objekte bearbeitet, um daraus die entsprechende Tabellenstruktur zu erzeugen. Dies geschieht über den Aufruf `reconstruct` der Klasse `AllAttributesElementReconstructor`. Darin werden anhand der Information, welche Klasse dargestellt werden soll, die entsprechenden Tabellenzellen generiert. Dazu werden die Attribute des `Entitys`, die ohnehin bereits Teil des `AllAttributesElement` ist, durchlaufen und, je nach Typ des Attributs, eine entsprechende Tabellenzeile generiert. Zusätzlich werden die Schlüssel für `ResourceBundle`-Beschriftung eines jeden Attributs erzeugt und dem `CombinedGlobalStore` hinzugefügt. Das Erzeugen der genannten Strukturen wird analog

zum Builder-Pattern [GHJV95] durch die Klasse `PagePartBuilder` realisiert. Hier werden die einzelnen Tabellenzeilen inklusive ihrer Zellen erzeugt, die entsprechenden Parameter belegt und letztlich zusammengesetzt.

Die erzeugte Tabelle wird schließlich anstelle des `AllAttributesElement` in die ursprüngliche Seite eingesetzt, so dass potenzielle andere Elemente weiterhin Teil der Seite sind und nur das relevante Element ersetzt wird.

Rekonstruktion der Assoziationen

Nach der Bearbeitung potenzieller `AllAttributesElemente` werden diejenigen Teile einer Tabelle, die eine Assoziation darstellen, durch Verwendung der Methode `reconstructEntityReferenceElements` der Klasse `TableReconstructor` bearbeitet. Dies geschieht sowohl für die neu erzeugte Tabelle als auch für alle Tabellen, die explizit in einer Seite im Seitenmodell spezifiziert wurden.

Für jede Tabellenzelle, die über ein `EntityReferenceCell`-Objekt repräsentiert ist und damit also eine Assoziation zu einer anderen Klasse darstellt, werden die `Attribute`-Objekte, die laut Spezifikation (entweder aus der Seitenbeschreibung oder aus Stereotypen des Klassendiagramms) dargestellt werden sollen, der Zelle hinzugefügt, so dass beim Generierungsschritt die entsprechenden Informationen zur Verfügung stehen. Zusätzlich werden für alle Kompositionen die entsprechenden Attribute hinzugefügt. Hierzu wird zunächst ein `AllAttributesElement` erstellt und daraus dann die entsprechende Tabellenstruktur erzeugt.

Rekonstruktion aller Formularelemente

Nach dem Umbau der Tabellenzellen werden sämtliche Formularelemente durch die Methode `reconstruct` der Klasse `FormElementsReconstructor` in der Art bearbeitet, dass die Informationen für das generierte System vollständig sind. Hierzu werden beispielsweise die Klassen- und Paketnamen korrekt gesetzt, ebenso wie die funktionalen Attributtypen (also etwas wie z. B. `Email`).

Umbau der Listenelemente

Für die `ListEntityElemente` innerhalb einer Seite werden keine umfangreichen Rekonstruktionen vorgenommen. In der Methode `reconstruct` der Klasse `ListEntityElementReconstructor` wird lediglich der vollqualifizierte Klassenname im Zielsystem gesetzt, der im generierten System genutzt wird.

7.3.5. Generator

Nach der Aufbereitung der Zwischenstruktur für die einfache Bearbeitung bei der Generierung wird der Generierungsschritt selbst ausgeführt. Hierzu kommt ebenfalls der in MontiWIS übliche Templatemechanismus zum Einsatz. Da für die Generierung von Seiten in der Regel nicht ausschließlich Informationen aus der Seitenspezifikation notwendig sind, sondern zusätzlich

auch Informationen aus dem Klassendiagramm sowie aus der Applikationsbeschreibungssprache hinzugezogen werden müssen, existieren die Templates im Paket `de.montiwis.combined`. Dort liegen generell die Artefakte, die zur Bearbeitung Informationen aus mehr als einer Modellart als Eingabe erfordern. In Tabelle 7.36 sind die Templates aufgelistet, die bei der Generierung der Seiten zum Verwendung finden. Die angegebenen Dateinamen- und Pfade sind relativ zum Verzeichnis `templates` angegeben. Hier sind die Templates erwähnt, die primär die Informationen der Seitenbeschreibung bzw. deren Zwischenstruktur verarbeiten.

Template-Datei	Erläuterung
`jsp/Page.ftl`	Erzeugt die JSP-Seiten für jede Seite
`jsp/header.ftl`	Steuert den Header für jede JSP-Seite bei
`jsp/footer.ftl`	Steuert den Footer für jede JSP-Seite bei
`jsp/chooseElements.ftl`	Generiert die Seitenteile für `chooseOne` oder `choose`-Elemente
`jsp/head.ftl`	Generiert die Seitenteile für `head`-Elemente
`jsp/text.ftl`	Generiert die Seitenteile für `text`-Elemente
`jsp/listEntity.ftl`	Generiert die Seitenteile für `list`-Elemente
`jsp/table.ftl`	Generiert die Seitenteile für `table`-Elemente
`jsp/table_composition_js.ftl`	Generiert JavaScript-Code für das Hinzufügen neuer Kompositionstabellen
`jsp/table_heading.ftl`	Generiert Tabellenteile für einen Tabellenkopf
`jsp/table_attribute_ref_cell.ftl`	Generiert die Tabellenteile für eine `AttributeReferenceCell`
`jsp/table_standard_ref_cell.ftl`	Generiert Tabellenteile für eine `StandardTypeReferenceCell`
`jsp/table_entity_ref_cell.ftl`	Generiert Tabellenteile für eine `EntityReferenceCell`
`java/PageEntity.ftl`	Erzeugt die PageEntity-Dateien für jede Seite
`java/PageEntityInitializer.ftl`	Erzeugt die PageEntityInitializer-Dateien für jede Seite
`java/ResourceFileForPage.ftl`	Erzeugt die Ressourcenklasse für jede Seite

Tabelle 7.36.: Die Template-Dateien für Seiten und ihre Verwendung

Generierung der Seite

Zur Erzeugung einer JSP-Seite wird das Template `jsp/Page.ftl` über die Klasse `CombinedGenerator` für jedes in der Zwischenstruktur enthaltene `Page`-Objekt aufgerufen und verarbeitet. Da im Zuge der Rekonstruktion der Seitenelemente bereits ein Großteil der notwendigen Informationen zum `Page`-Objekt und seinen Elementen hinzugefügt wurde, sind relativ wenige Zusatzinformationen notwendig. Lediglich Informationen über `ResourceBundle`-Dateien sowie die Information, ob Feedback-Client-Funktionalität[23] hinzu generiert werden soll, werden dem Template zusätzlich zur Verfügung gestellt. Ferner wird eine Referenz auf den `ApplicationGlobalStore` übergeben und damit Zugriff auf die modellierten Zugriffsberechtigungen gewährt, so dass diese bei der Generierung der Seite mit einbezogen werden können.

Die Generierung der Seite selbst geschieht unter Verwendung des Templates `Page.ftl`, welches jedoch eine Reihe weiterer Templates einbindet. Diese wurden der Übersichtlichkeit halber ausgelagert und erlauben so eine etwas fokussiertere Sicht auf einzelne Teile der Seite. Einerseits wurden die Kopf- und Fußteile der Seite, die in sämtlichen Seiten identisch sind, in separaten Templates gespeichert. Eine weitere Aufteilung erfolgt nach der Art der `PageElements`, was zu eigenen Dateien für Elemente wie `text` oder `table` führt. Die relevanten Teile der Templates sollen nachfolgend kurz erläutert werden.

In das zentrale Template `Page.ftl` wird das Template `header.ftl` eingebunden, das den Kopf einer jeden Seite enthält. Darin sind die Teile spezifiziert, die sämtlichen Seiten gemein sind. Dies sind zum Beispiel die Deklarationen von Taglibrary und `ResourceBundle`, die in der generierten JSP-Seite notwendig sind, die Grundstruktur der HTML-Seite inklusive des `head`-Teils mit seinen CSS-Links sowie der erste Teil des `body`-Bereichs der HTML-Seite. Darin sind Referenzen auf die verwendeten JavaScript-Dateien sowie einige später gefüllte `div`-Tags, beispielsweise zur Darstellung potenzieller Meldungen an den Nutzer, enthalten. Analog zum Kopfteil der Seite wird über das Template `footer.ftl` ein minimaler Fußteil der Seite eingebunden, in dem primär die im Kopfteil geöffneten HTML-Elemente wieder geschlossen werden.

Im Template `Page.ftl` selbst werden lediglich die korrekten HTML `form`-Elemente sowie die Knöpfe zum Absenden des Formulars erzeugt. Die Behandlung der weiteren Elemente, die letztlich aus den `PageElement`-Objekten generiert werden, ist in separate Templates ausgelagert. Darin erfolgt eine Überprüfung hinsichtlich des Elementtyps, so dass das jeweilige Element in korrekter Weise ausgegeben werden kann.

Im Template `head.ftl` werden die einzelnen `TextContents`-Objekte, aus denen die Überschrift besteht, durchlaufen und entsprechend ihres Typs ausgegeben. Sollte es sich um ein `StringContents` handelt, wird einfach der Inhalt des dargestellt, wobei im Falle eines `AttributeReferenceContents` der Wert des referenzierten Attributs ausgegeben wird. Entsprechend seiner Funktion innerhalb einer Seite wird der Inhalt der Seite durch Verwendung eines `h1` HTML-Elements formatiert.

[23] Der Feedback-Client ist eine Funktionalität, über die Bemerkungen eines Nutzers auf einem Server gesammelt werden können. Weitere Erläuterungen sind in Abschnitt 10.1.2, [LHK+12] sowie in der in Entstehung befindlichen Dissertation von Thomas Kurpick zu finden.

Sehr ähnlich verhält es sich mit dem Template `text.ftl`, welches den Inhalt eines `Text` Seitenelements verarbeitet und ebenfalls die `TextContents`-Objekte ausgibt.

Der bei Weitem komplexeste Generierungsschritt wird bei der Erzeugung einer Tabelle inklusive ihrer verschiedenen Tabellenzellen durchgeführt. Aus diesem Grund wurde das Template `table.ftl` der Übersichtlichkeit halber nochmals aufgeteilt. Im Template für die Tabellenerzeugung selbst wird über die Zeilen einer Tabelle sowie über deren einzelne Tabellenzellen iteriert und für jede Zellenart ein entsprechendes Template eingebunden, das für die Generierung des Inhalts der jeweiligen Tabellenzelle zuständig ist.

Zunächst wird in dem Template `table_composition_js.ftl` jedoch über sämtliche Tabellenzellen iteriert und für den Fall, dass eine Referenz auf ein Kompositionsattribut existiert, eine JavaScript-Variable generiert, die bei Hinzufügen eines neuen Kompositionsobjekts an entsprechender Stelle in die Seite eingefügt wird. Der Inhalt dieser Variablen wird erzeugt durch das Durchlaufen der sog. `compositionTable` der `EntityReferenceCell` generiert. Diese Tabelle wurde im Rahmen des Rekonstruktionsschrittes erzeugt und dieser Zelle hinzugefügt (vgl. Abschnitt 7.3.4, wo dieses Vorgehen im Kontext der Rekonstruktion von Assoziationen erläutert wurde). Die dort generierten Tabellenelemente entsprechen im Wesentlichen denen von gewöhnlichen `AttributeReferenceCell`-Elementen, so dass auch das Verhalten dieser beiden Tabellen korrespondiert. Da die Deklaration dieser JavaScript-Variablen außerhalb der Tabelle geschehen muss, findet dieser Generationsschritt vor der eigentlichen Tabellenerzeugung statt.

Zu Beginn der Tabellengenerierung wird das Template `table_heading.ftl` eingebunden. Darin wird geprüft, ob ein Tabellenkopf definiert wurde. Sollte dies der Fall sein, wird über die einzelnen Zellen des Kopfes iteriert und jede Zelle entsprechend ihres Typs verarbeitet. Mögliche Inhalte sind an dieser Stelle Attributreferenzen, ausgelagerte `ResourceBundle`-Variablen oder auch einfacher Textinhalt, der ohne Weiteres übernommen werden kann.

Die Generierung sämtlicher gewöhnlicher Attributreferenzen ist ausgelagert in das Template `table_attribute_ref_cell.ftl`. Darin wird zunächst unterschieden, ob es sich um ein editierbares Attribut oder lediglich um die Ausgabe seines Wertes handelt. Im ersten Fall wird, entsprechend des funktionalen Attributtyps (also etwa `Email` oder `Date`), das passende Eingabeelement inklusive Validierungsmechanismen generiert. Im Falle, dass das referenzierte Attribut lediglich ausgegeben werden soll, wird das passende Konstrukt zur Ausgabe eines Attributwertes erzeugt.

Sollte es sich bei dem referenzierten Attribut nicht um ein Attribut des Parameters, sondern um den Seitenparameter selbst und damit um einen vordefinierten Datentyp handeln, wird das Template `table_standard_ref_cell.ftl` eingebunden und durchlaufen. Die generierten Eingabe- bzw. Ausgabeartefakte sind sehr ähnlich zu denen, die bei gewöhnlichen Attributreferenzen erzeugt werden, jedoch wird selbstverständlich nicht auf ein Attribut, sondern den Parameter direkt verwiesen.

Wenn die bearbeitete Tabellenzelle eine Assoziation repräsentiert und damit auf ein oder mehrere andere Objekte verweist, erfolgt die Bearbeitung über das Template `table_entity_ref_cell.ftl`. Darin wird zunächst unterschieden, ob es sich bei der Assoziation um eine gewöhnliche Assoziation (also tatsächlich eine Referenz auf ein existierendes Objekt) oder um eine Komposition (und damit auf ein direkt auf der Seite erzeugtes Element)

handelt. Im ersten Fall wird, sofern die Attributreferenz eine editierbare ist, ein Link zum Hinzufügen bzw. Setzen eines neuen bzw. anderen Objekts erzeugt. Dessen Bezeichnung besteht aus dem Verb *add* gefolgt vom Klassennamen des Zielobjekts, also für das Hinzufügen eines Personenobjekts etwa *add Person*. Die Funktionalität, die dazu generiert wird, ist in einer JavaScript-Funktion ausgelagert, die entsprechenden Parameter der Funktion werden an dieser Stelle mitgeneriert. Als Nächstes werden potenziell bereits vorhandene Objekte dargestellt. Hierbei wird unterschieden, ob es sich um eine Referenz zu exakt einem Objekt handelt, oder ob mehrere Objekte referenziert werden können. Im ersten Fall wird lediglich die Referenz auf ein Objekt ausgegeben, in zweitem Fall werden JSP-Konstrukte generiert, die bei Ausführung der Seite die Ausgabe der übergebenen Objekte vornehmen. Hinsichtlich der Art der Darstellung wird unterschieden, ob ein Attribut editierbar, lediglich anzuzeigen oder als Link zur Standarddarstellungsseite zu generieren ist. Die schlichte Anzeige einer Assoziation führt zur Ausgabe der spezifizierten Attribute, welche auch bei beiden anderen Ansichten verwendet werden. Im Falle der Editieransicht wird zusätzlich ein Link zum Löschen der Assoziation erzeugt, der den Link zunächst auf der Seite und nach Speichern der Seite ebenfalls auf Serverseite entfernt.

Im gleichen Template `table_entity_ref_cell.ftl` werden ebenfalls Kompositionen behandelt. Die Funktionalität für das Hinzufügen eines neuen Objekts wurde bereits in dem Template `table_composition_js.ftl` behandelt. Code für bereits existierende Objekte, die zum dargestellten Objekt in entsprechender Kompositionsbeziehung stehen, wird durch dieses Template erzeugt. Die Darstellung der einzelnen Attribute erfolgt ebenfalls analog zu den gewöhnlichen Attributreferenzen in Tabellenform.

Die Ausgabe von `LangStringCell` sowie `TextCell` Tabellenzellen wird aufgrund geringer Komplexität direkt im Template `table.ftl` vorgenommen. Dafür wird die entsprechende `ResourceBundle`-Funktionalität bzw. der spezifizierte Text direkt in die Seite generiert.

Nach der kurzen Beschreibung der Generierung der JSP-Seiten wird nachfolgend die Generierung der restlichen, für die Seitendarstellung notwendigen Artefakte, erläutert.

Generierung der PageEntities

Wie in Abschnitt 5.3.3 motiviert und in Abschnitt 7.4.3 detailliert erläutert, werden zu jeder Seite sog. `PageEntity`- und `PageEntityInitializer`-Klassen generiert, die für den Transfer von Werten in die Seite und aus dem Seitenkontext heraus in ein Objekt der jeweiligen Klasse zuständig sind. Da deren Struktur primär vom Aufbau der Seite abhängt, enthält die Zwischenstruktur der Seite auch die maßgeblichen Informationen für die Generierung. In der Klasse `CombinedGenerator` wird über sämtliche `Page`-Objekte aus dem Zwischenstrukturspeicher iteriert und dabei den beiden Templates eine Instanz der jeweiligen Seite übergeben sowie einige Attribute, deren Werte bei der Generierung verwendet werden. Der Name der generierten Klasse resultiert aus dem Namen der Seite, erweitert um das Suffix `PageEntity`- bzw. `PageEntityInitializer`.

Für die Generierung einer `PageEntity`-Klasse wird das Template `PageEntity.ftl` verwendet, die `PageEntityInitializer`-Klasse wird über das Template `PageEntityInitializer.ftl` erzeugt.

Da die Aufgabe der `PageEntity`-Klasse hauptsächlich der Transfer von Attributwerte in eine Seite bzw. daraus hinaus ist, werden die Parameter und Attributreferenzen einer Klasse verarbeitet. Für jeden Parameter wird ein Feld in der Klasse angelegt, welches mit den übergebenen Werten belegt wird. Dabei werden sowohl gewöhnliche Klassen als auch mengenwertige `Collections` von Klassen unterstützt.

Die Logik zum Transfer der Werte aus der Seite heraus in Objekte, die auf Serverseite weiterverwendet werden, ist in der `PageEntityInitializer`-Klasse gekapselt, worin für jedes Attribut und jede Attributart die entsprechende Logik generiert. Diese Logik ist insbesondere dann komplex, wenn Assoziationen oder Kompositionen enthalten sind, da hier potenziell Objekte per `Dao` nachgeladen bzw., im Falle von Kompositionen, erzeugt und mit den entsprechenden Werten belegt werden müssen.

Neben dieser zentralen Funktionalität existieren in der `PageEntityInitializer`-Klasse weitere Methoden, beispielsweise zum Setzen der Felder von außerhalb der `PageEntity` sowie zum Abruf der Felder zur Weiterverwendung.

Eine detailliertere Beschreibung der Implementierung der `PageEntity`- und `PageEntityInitializer`-Objekte kann Abschnitt 7.4.3 entnommen werden.

Generierung der Ressourcenklassen für Seiten

Über das Template `ResourceFileForPage.ftl` wird für jede Seite eine Ressourcenklasse erzeugt, so dass diese theoretisch direkt per URL aufgerufen werden kann. Dies geschieht jedoch in den seltensten Fällen, nämlich nur bei Seiten, die ausschließlich statischen Inhalt aber keine Parameter enthalten. Selbst Seiten, die standardmäßig für jede Klasse im Domänendatenmodell erzeugt wurde, werden nicht über die hierüber generierte Ressourcenklasse aufgerufen, sondern über die REST-Schnittstelle der jeweiligen Klasse (vgl. hierzu Abschnitt 6.4.2 sowie Tabelle 6.23). Dennoch soll hier kurz beschrieben werden, welche Besonderheiten sich bei der Generierung dieser Seiten ergeben.

Das Template wird, wie alle hier beschriebenen Templates, über die Klasse `CombinedGenerator` mit den notwendigen Werten befüllt. Dies geschieht auch bei diesem Template für jede Seite, die im zentralen Zwischenstrukturspeicher existiert. Übergeben werden im Wesentlichen das jeweilige `Page`-Objekt, eine Referenz auf den `ApplicationGlobalStore` (für den Abruf der dort gespeicherten Zugriffsrestriktionen) sowie einige weitere Variablen, deren Belegung aufwendigere Operationen innerhalb des Templates ersparen (wie beispielsweise der Name der referenzierten JSP-Seite).

Im Template selbst werden verhältnismäßig wenig dynamische Teile berücksichtigt. Primär werden die Zugriffsbeschränkungen, die zu verwendenden JSP-Seitennamen sowie Seitennamen in einem ansonsten überwiegend statischen Template mit zwei Methoden zur Generierung herangezogen. Eine dieser Methoden dient zum Aufruf der Seite über die HTTP-Methode `GET`, eine weitere zum Verarbeiten der aus dem Formular abgesendeten Werte durch `POST`, was aber in der Regel nicht vorkommt.

7.4. Implementierung im Zielsystem

Nach der Erläuterung, wie aus dem AST eine Zwischenstruktur aufgebaut wird, wie diese Zwischenstruktur aufgebaut ist und welche Rekonstruktionen darauf ausgeführt werden, um letztlich eine möglichst einfache Generierung über Templates zu erlauben, wird im folgenden Abschnitt auf die tatsächliche Struktur der generierten Artefakte eingegangen. Zunächst wird in Abschnitt 7.4.1 die allgemeine Struktur der JSP-Seiten beschrieben, erläutert welche Technologien in welcher Art und Weise eingesetzt werden und wie diese anzupassen sind. Nachfolgend beschreibt Abschnitt 7.4.2, wie die einzelnen Attributarten im generierten System umgesetzt sind, was ebenfalls Assoziationen und Kompositionen einschließt. Abschnitt 7.4.3 beschreibt die relevanten Teile der `PageEntity`-Klassen und Abschnitt 7.4.4 geht kurz auf die Ressourcenklassen von Seiten ein. Schließlich wird in Abschnitt 7.4.5 diskutiert, welche Sicherheitsüberlegungen bei der Generierung der Seiten berücksichtigt werden und wie diese umgesetzt wurden.

7.4.1. Generelle Seitenarchitektur

Dieser Abschnitt beschreibt überblicksartig, wie die JSP-Seiten aufgebaut sind. Insbesondere werden die verwendeten Technologien und deren Einsatz im generierten System beschrieben und erläutert. Ein Überblick über Seiten, ihre Komponenten und ihre Erzeugung ist in Abbildung 7.37 dargestellt.

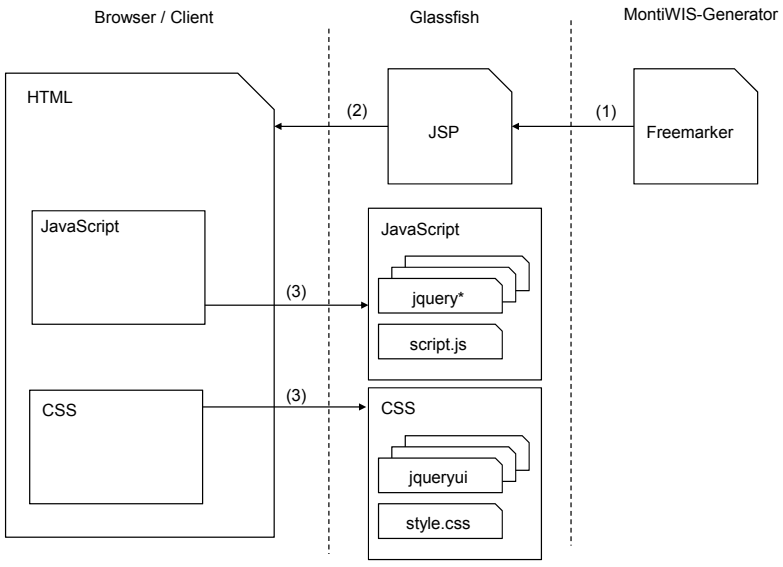


Abbildung 7.37.: Zusammenhang zwischen Templates, Seiten- und ihren Anteilen

Wie dort unter (1) zu sehen werden *Freemarker-Templates* auf Seiten des *MontiWIS-Generators* verwendet, um *JSP-Seiten* zu generieren. Diese werden im *Glassfish Server* ausgeführt und zu *HTML*-Seiten generiert, welche wiederum auf *Client-Seite* im Browser dargestellt werden (2). Darin eingebunden und nachgeladen werden, wie unter (3) dargestellt, sowohl

JavaScript- als auch *CSS*-Dateien. Diese Dateien werden im Applikationsserver bereitgehalten. Sie wurden nicht durch den MontiWIS-Generator erzeugt, sondern liegen als statische Inhalte bei der Zusammenstellung der Webapplikation vor. Auf diese einzelnen Komponenten wird im Folgenden näher eingegangen.

JSP-Seiten

Der Aufbau einer generierten JSP-Seite orientiert sich im Wesentlichen an der Struktur, wie sie in Abbildung 7.38 dargestellt ist.

JSP

```
<!DOCTYPE html>
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<%@ taglib uri="http://montiwis.de/taglib" prefix="m" %>

<fmt:bundle basename="resource_bundles.langstring">

<html>
  <head>
    <meta charset="utf-8" />
    <title>EnterLecturerData</title>
    <link rel="stylesheet" href="${pageContext.request.contextPath}/css/custom/jquery-ui.↩
        custom.css"/>
    <link rel="stylesheet" href="${pageContext.request.contextPath}/css/demo_table_jui.↩
        css"/>
    <link rel="stylesheet" href="${pageContext.request.contextPath}/css/demo_page.css"/>
    <link rel="stylesheet" href="${pageContext.request.contextPath}/css/style.css"/>
  </head>
  <body>
    <script src="${pageContext.request.contextPath}/script/jquery.js"></script>
    <script src="${pageContext.request.contextPath}/script/jquery-ui.custom.min.js"></↩
        script>
    <script src="${pageContext.request.contextPath}/script/jquery.validate.js"></script>
    <script src="${pageContext.request.contextPath}/script/jquery.validate-add.js"></↩
        script>
    <script src="${pageContext.request.contextPath}/script/jquery.dataTables.js"></script↩
        >

    <script src="${pageContext.request.contextPath}/script/script.js"></script>

    <div id="dialog" title="add some entity">
      <ul id="dialog-list"></ul>
    </div>

    <div id="info"></div>

    <div id="all">
```

```
36        <%@ include file="/WEB-INF/jsp/menu.jsp" %>
37        <div id="content">
38
39          <!-- Inhalt der Seite -->
40
41        </div>
42      </div>
43    </body>
44  </html>
45  </fmt:bundle>
```

Abbildung 7.38.: Grundaufbau der generierten JSP-Seiten

Wie in Zeile 1 zu sehen, wird als Dokumententyp derjenige verwendet, der in HTML5 vorgeschrieben ist [HTM, Krö11]. Dies führt dazu, dass Webbrowser die generierte Seite entsprechend als HTML5 interpretieren und damit auch kompaktere Konstrukte erlauben. Die Zeilen 2-3 enthalten eine Seitendirektive, die der JSP Ausführungsengine mitteilt, wie die folgende Seite zu behandeln ist, was in diesem Fall die verwendete Sprache sowie Inhalt und Zeichencodierung der zu generierenden Seite festlegt. Die Zeilen 4-7 deklarieren, welche Taglibraries in der Seite verwendet werden können. Hierzu wird über die `uri` eindeutig identifiziert, welche Implementierung herangezogen werden soll und über den `prefix`, wie die jeweilige Taglibrary in der JSP-Seite aufgerufen werden kann, wobei die ersten drei Referenzen zu den JSTL gehören und die vierte Referenz Funktionalität bereitstellt, die in der Laufzeitbibliothek von MontiWIS enthalten ist (weitere Erläuterungen bzgl. JSP Taglibraries siehe [SCR01], zu JSTL siehe [Bay03]). In Zeile 9 wird auch das erste der JSTL-Tags verwendet um festzulegen, welche `ResourceBundle`-Datei innerhalb dieser der generierten JSP zum Einsatz kommt. Geschlossen wird dieses Element wieder in Zeile 45, so dass der gesamte Inhalt der JSP-Datei dadurch abgedeckt ist. Der HTML-Inhalt selbst befindet sich in den Zeilen 11-44. Zunächst werden im Kopf der Datei die Zeichencodierung (Zeile 13) sowie der Seitentitel (Zeile 14) festgelegt und CSS-Stylesheets eingebunden. Diese sind überwiegend Dateien von Drittanbietern, die für bestimmte Funktionalität in MontiWIS genutzt werden. Dazu zählen jQuery UI in Zeilen 15-16 [JQuc] sowie jQuery Datatables in Zeile 17 [Dat]. Das Stylesheet, das die Styles von MontiWIS enthält, wird in Zeile 18 eingebunden.

Der eigentliche `body` der HTML-Datei erstreckt sich von Zeile 20-43. Darin werden zunächst die notwendigen JavaScript-Dateien eingebunden, die in der Regel in einer JSP-Seite von MontiWIS zum Einsatz kommen. Auch hier handelt es sich überwiegend um Script-Dateien von Drittanbietern, die für die z. T. dynamische Funktionalität benötigt werden. Hierbei wären zu nennen die jQuery-Bibliothek in Zeile 21 [jQua], jQueryUI in Zeile 22 [JQuc], jQuery Validation Plugin in Zeilen 23-24 [jQub] sowie jQuery Datatables in Zeile 25 [Dat]. In Zeile 27 schließlich wird die JavaScript-Datei eingebunden, die MontiWIS-spezifische Funktionalität enthält. Die nachfolgenden `div`-Elemente werden im Falle des Hinzufügens eines Entities (Zeilen 29-31) sowie zur Ausgabe von dynamischen Meldungen (z. B. Fehlermeldungen, Zeile 33) genutzt und bei Bedarf durch JavaScript-Logik entsprechend modifiziert.

Das `div`-Element mit `id="all"` (Zeile 35-42) umschließt den kompletten Inhalt der ge-

nerierten Seite. Über die `include`-Direktive in Zeile 36 wird ein JSP-Teil eingebunden, der das generierte Menü erzeugt und an entsprechender Stelle in der Seite erzeugt. Näheres zur Modellierung und Generierung des Menüs ist Abschnitt 9.2.2 zu entnehmen. Das `div`-Element mit `id="content"` enthält schließlich den Inhalt der Seite, der über die Seitenbeschreibungssprache bzw. letztlich durch deren Zwischenstruktur beschrieben wird und an exakt diese Stelle generiert wird.

Die bislang beschriebene Struktur entstammt primär dem, was per `header.ftl` bzw. `footer.ftl` in jede Seite generiert wird.

Verwendung von JavaScript

Wie bereits mehrfach erwähnt, wird ein Großteil der clientseitigen Logik in MontiWIS durch JavaScript realisiert. Hierbei wird sowohl auf Standardfunktionalität durch Drittanbieter (vornehmlich jQuery und dem nahestehende Plugins) als auch auf für MontiWIS geschriebene, angepasste Funktionalität zurückgegriffen. Nachfolgend soll kurz die jeweils genutzte bzw. entwickelte Funktionalität erläutert werden.

jQuery Ein Großteil der verwendeten Funktionalität in MontiWIS basiert auf der *jQuery*-Bibliothek [jQua]. Diese bietet einen umfangreichen Satz an Funktionen, die es erleichtern, bestimmte Teile einer HTML-Seite zu selektieren und zu modifizieren. Darüber lassen sich wesentlich bequemer und insbesondere auch relativ sicher browserübergreifend bestimmte Funktionen wie beispielsweise das dynamische Generieren und einfügen von HTML-Elementen in eine Seite realisieren, was wesentlich zur plattformübergreifenden Nutzbarkeit von MontiWIS beiträgt.

jQuery UI Sehr eng verknüpft mit dem Einsatz von jQuery ist auch *jQuery UI* [JQuc], was einige sehr funktionale und ebenfalls browserübergreifend nutzbare Oberflächenelemente wie beispielsweise das in MontiWIS verwendete dynamische Menü, den Datepicker oder auch die Pop-up-Fenster zur Auswahl von zu assoziierenden Objekten bereitstellt. Neben der Funktionalität zur Unterstützung der Interaktivität und für die Bereitstellung dynamischer Funktionalität erlaubt jQuery UI ebenfalls über einen einfachen Konfigurationsdialog auf seiner Webseite, das Aussehen der Widget-Elemente in vielfältiger Form anzupassen. Das gewählte Design kann einfach in Form von CSS-Dateien und zugehörigen Bilddateien heruntergeladen und in MontiWIS integriert werden, was dazu führt, dass sich das Aussehen von Seite und Bedienelementen konsistent anpasst. Auf die Verwendung bestimmter Funktionalitäten wird nachfolgend bei der Beschreibung der einzelnen Elemente noch eingegangen.

jQuery Validation Plugin Für die clientseitige Validierung der Eingaben kommt ebenfalls ein Plugin eines Drittanbieters zum Einsatz, was in diesem Fall das Plugin *jQuery Validation Plugin* [jQub] ist. Hierüber ist durch einfache Deklaration einer bestimmten `class` an einem `input`-Element möglich, dass durch Aufruf der `validate`-Methode auf dem selektierten `form`-Element innerhalb der Webseite die syntaktische Korrektheit der eingegebenen Werte überprüft wird. Dies geschieht direkt beim Verlassen eines jeden (zu validierenden) Feldes sowie bei Absenden des Formulars. Sollten noch inkorrekte Werte

in den Eingabefeldern vorhanden sein, wird das Absenden des Formulars verhindert. Der Aufruf des Plugins wird an entsprechender Stelle bei der Beschreibung der einzelnen Eingabeelemente erläutert.

jQuery Datatables Plugin Da in MontiWIS an verschiedenen Stellen Tabellen eingesetzt werden, um eine Menge von Objekten in möglichst einfacher und ansprechender Form darzustellen, wird auch für deren Darstellung ein Plugin eingesetzt. Hierbei handelt es sich um das *jQuery Datatables Plugin* [Dat]. Die Verwendung dieses Plugins führt bei korrektem Aufbau der Tabellenstruktur zum Hinzufügen der Funktionalität, wie sie in u. a. in Abschnitt 7.2.3 beschrieben ist. Auch für dieses Plugin wird die Verwendung an entsprechender Stelle erläutert.

script.js Die Datei `script.js` enthält die Funktionalität in JavaScript, die einzig für das Verhalten von MontiWIS geschrieben wurde. Sie ist komplett statisch und wird als Teil der Webanwendung mitgeliefert. An dieser Stelle soll nicht die komplette Datei erläutert werden, sondern lediglich kurz ein Überblick über die Funktionen und deren Sinn und Zweck gegeben werden. Die Funktion `$()` (die Teil von jQuery ist) wird ausgeführt, sobald das Dokument vollständig im Browser geladen wurde. Darin werden Initialisierungen vorgenommen, beispielsweise um die Validierung, die Menü-Funktionalität sowie bestimmte Darstellungen wie Button-Formatierung oder auch die Datatables-Funktionalität zu aktivieren. Die Funktion `loadRunningActivities` dient zum Nachladen von Informationen über die Aktivitäten, an denen der aktuell agierende Nutzer beteiligt ist und in denen aktuell für ihn Handlungsbedarf besteht. Dazu wird per AJAX-Funktion eine Anfrage an die `ActivitiesResource` gestellt und das Ergebnis dynamisch in die Seite eingebaut. Die Methode `add` dient dazu, bei der Auswahl eines Objekts als Assoziation das entsprechende Pop-up- Fenster zu erzeugen. Dazu werden die möglichen Objekte per AJAX von der Ressourcenklasse der Zielklasse abgerufen (über den `/list`-Pfad mit JSON-Rückgabewert, vgl. auch Tabelle 6.23). Die Ergebnisse werden in einer Tabelle dargestellt, die ebenfalls durch diese Methode erzeugt und dem Pop-up-Dialog hinzugefügt wird. Die Methoden `remove` bzw. `removeComposition` entfernen die betroffenen Elemente aus einer Seite und erlauben damit das clientseitige Löschen von Assoziation. Die Methoden `sendPost` bzw. `sendDelete` senden das Formular der Seite zum Server, wobei beim Löschen noch der Eintrag zum Behandeln der Anfrage als HTTP `DELETE`-Methode hinzugefügt wird (für eine Erläuterung dieser Problematik und dem verwendeten Lösungsansatz siehe Abschnitt 5.3.1). Die Methode `preview` schließlich erzeugt ein Dialogfeld und lädt die übergebene URL dort hinein, was in der Regel zum Laden der Standardansicht eines Objekts dient. Ferner sind in dieser JavaScript-Datei noch einige Hilfsmethoden enthalten, auf die hier nicht näher eingegangen werden soll, da sie lediglich unterstützende Funktion bei der Ausführung der beschriebenen Methoden haben.

Verwendung von Cascading Style Sheets

Wie allgemein in der Entwicklung von Websystemen üblich, wird auch im generierten System in MontiWIS die optische Umsetzung, d. h. die stilistische Darstellung, über Cascading Style Sheets (CSS) realisiert. Dadurch bleibt die Struktur einer Seite unabhängig davon, in welcher Größe, Farbe oder Ausrichtung die einzelnen Elemente der Seite dargestellt werden. Wie im vorherigen Abschnitt erläutert, werden einige Bibliotheken von Drittanbietern eingesetzt, die in der Regel auch eigene CSS-Dateien mitbringen. Primär erfolgt die Darstellung angelehnt an die Stilangaben, wie sie von *jQuery UI* geliefert werden. Neben diesen orientiert sich ebenfalls das *Datatables Plugin* an diesen Stilvorgaben, bringt jedoch noch einige eigene Dateien mit. Auf all diese Dateien soll an dieser Stelle nicht gesondert eingegangen werden, da die hier spezifizierten Angaben lediglich ergänzend zur bereits erläuterten Funktionalität der JavaScript-Bibliotheken existieren.

Neben den jQuery UI Stylesheets, die maßgeblich die Darstellung der Nutzerschnittstelle in MontiWIS bestimmen, existiert mit der Datei `style.css` eine Styledatei, die das globale Layout der generierten Applikation festlegt. Darin werden globale Schriftvorgaben ebenso definiert wie die Breite der Seite sowie ihrer einzelnen Teile wie Menü oder Inhaltsbereich, der rechts vom Menü dargestellt wird. Auch die Hintergrundfarbe ist hier spezifiziert.

Die Anzahl der hier definierten Stilangaben ist recht überschau- und nachvollziehbar, weswegen an dieser Stelle nicht im Einzelnen darauf eingegangen wird.

7.4.2. Umsetzung der verschiedenen Datentypen

Nach der Erläuterung, unter Zuhilfenahme welcher Technologien die clientseitige Funktionalität bereitgestellt wird, sind nun die Umsetzungen der einzelnen Datentypen zu erläutern. Hierbei wird anhand von Beispielen erläutert, wie diese im generierten Code umgesetzt werden und welche (potenziell umfangreiche) Funktionalität dabei verwendet wird.

Umsetzung der einzelnen Seitenelemente

Wie beispielsweise in Abschnitt 7.3.5 bzw. Abschnitt 6.2.3 erläutert, werden für die Eingabefelder in der Regel Texteingabefelder generiert, über die die Attributwerte eingegeben bzw. verändert werden können. Abbildung 7.39 zeigt eine Auswahl der generierten Eingabefelder in HTML Quellcode.

Die Eingabefelder, die in einer generierten Seite verwendet werden, sind (bis auf ein `textarea`-Feld) in der Regel gewöhnliche HTML `input`-Felder. Der Unterschied bei den verschiedenen Datentypen besteht im Wesentlichen aus den `type`- und den `class`-Attributen des jeweiligen Elements. Erstere geben an, welcher Art das jeweilige Eingabeelement ist, Letztere werden verwendet, um festzulegen, wie jQueryUI [JQuc] und das jQuery validation plugin [jQub] das Element darstellen bzw. im Schritt der clientseitigen Validierung behandeln sollen. Die einzelnen Attributwerte und deren Auswirkung werden nachfolgend in aller Kürze, aufgeteilt nach vordefinierten Datentypen, erläutert. Für die umfassende Darstellung sämtlicher möglichen Attribute sind in diesem Beispiel typische Fälle dargestellt, ohne dass

JSP

```
1 <input class="" type="text" name="student.matriculationNumber"
2   value="${fn:escapeXml(it.student.matriculationNumber)}" />
3
4 <textarea class="" rows="8" name="lecturer.resume">
5   ${fn:escapeXml(it.lecturer.resume)}</textarea>
6
7 <input class="datepicker date" type="text" name="student.birthDate"
8   value="<fmt:formatDate pattern="MM'/'dd'/'yy"
9     value="${it.student.birthDate}"/>" />
10
11 <input class="integer" type="text" name="student.overallCreditPoints"
12   value="${fn:escapeXml(it.student.overallCreditPoints)}"/>
13
14 <input class="number" type="text" name="student.averageGrade"
15   value="${fn:escapeXml(it.student.averageGrade)}"/>
16
17 <input type="radio" name="lecturer.isActive" value="true"
18   <c:if test="${it.lecturer.isActive()}">checked</c:if>>Yes
19 <input type="radio" name="lecturer.isActive" value="false"
20   <c:if test="${not it.lecturer.isActive()}">checked</c:if>>No
21
22 <input class="email" type="text" name="student.email"
23   value="${fn:escapeXml(it.student.email)}"/>
24
25 <input class="" type="password" name="student.password" value=""/>
```

Abbildung 7.39.: Grundaufbau von Eingabefeldern

hierbei vollständige und sinnvolle Seiten erzeugt werden. Dargestellt sind einige Attribute der Klassen `Student` und `Lecturer` und ihre generierte Repräsentation.

In sämtlichen nachfolgend beschriebenen Konstrukten wird im Attribut `name` auf eine Seitenvariable verwiesen, die sich nach dem Schema zusammensetzt, wie es im folgenden Abschnitt 7.4.3 beschrieben ist. Ferner wird bei der Ausgabe potenziell vorhandener Werte (durch die Verwendung des Attributs `value`) aus Gründen der Sicherheit über die Funktion `fn:escapeXml` (Teil der JSTL Functions Taglibrary [Bay03]) die Maskierung potenziell gefährlicher Zeichen vorgenommen. Erläutert werden diese Überlegungen in Abschnitt 7.4.5.

`String` Die Zeilen 1-2 zeigen, wie das Eingabefeld für einen `String` generiert wird. Dies ist auch gleichzeitig das rudimentärste Eingabefeld, da hier ein einfaches `text`-Feld zum Einsatz kommt und dabei keine `class` angegeben wird. Dies ist insofern ausreichend, als keine Validierung vorgenommen werden muss und somit auch keine spezielle clientseitige Funktionalität notwendig ist.

`Text` Ähnlich erfolgt die Generierung von Elementen für den vordefinierten Datentyp `Text` (Zeilen 4-5). Auch hier ist keine `class` angegeben, jedoch wird statt eines einfachen

Eingabefeldes eine `textarea` erzeugt, für die 8 Zeilen in der Standardansicht angezeigt werden.

`Date` Eines der komplizierteren Elemente wird aus dem Datentyp `Date` generiert (Zeilen 7-9). Hier werden zwei Klassenattribute angegeben. Die CSS-Klasse `datepicker` wird von jQuery UI verwendet, um die entsprechenden Modifikationen für die Darstellung eines Auswahlfensters vorzunehmen. Die Klasse `date` kommt im Rahmen des Validierungsplugins von jQuery zum Einsatz, um das Format der Eingabe sowohl zum Zeitpunkt des Absendens als auch beim Verlassen des Eingabefeldes zu überprüfen. Bei der Darstellung von bereits vorhandenen Werten wird über die Funktion `fmt:formatDate` (Teil der JSTL Format Taglibrary [Bay03]) der `Date`-Datentyp in der angegebenen Art und Weise formatiert. Ebenso wird das Datum dargestellt, das über das Datumsauswahlfenster gewählt und damit in das Textfeld eingefügt wurde.

`Integer` Ein `Integer`-Datentyp wird wie in Zeilen 11-12 gezeigt umgesetzt. Dazu wird ein einfaches `input`-Feld verwendet, welches über die CSS-Klasse `integer` für die Validierung vorbereitet wird. Dies bewirkt, dass lediglich positive oder negative ganze Zahlen innerhalb des Feldes vorkommen dürfen.

`Float` Ähnlich wird der vordefinierte Datentyp `Float` behandelt (Zeilen 14-15). Hier wird allerdings die CSS-Klasse `number` angegeben, was zur Validierung einer (positiven wie negativen) Dezimalzahl führt.

`Boolean` Für einen `Boolean`-Datentyp werden die Konstrukte generiert, wie sie in Zeilen 17-20 zu sehen sind. Hierbei handelt es sich um zwei Radiobuttons, die abhängig vom aktuellen Wert entweder markiert oder nicht markiert sind. Für die Überprüfung werden `c:if` Konstrukte der JSTL Core Taglibrary [Bay03] verwendet, über die entsprechend des Attributwertes `checked`-Attribute generiert werden.

`Email` Das vordefinierte Attribut `Email` wird ebenfalls über ein `input`-Textfeld eingegeben bzw. verändert (Zeilen 22-23). Durch die CSS-Klasse `email` wird über das jQuery validation plugin [jQub] sichergestellt, dass die Eingabe syntaktisch einer E-Mail-Adresse entspricht.

`Password` Ein `Password` wird über ein Textfeld vom Typ `password` verarbeitet (Zeile 25). Hier findet keine besondere Validierung statt, lediglich die übliche Maskierung der Eingabe über symbolhafte Punkte erfolgt durch diese Darstellung. Wie in Abschnitt 6.2.3 erläutert, werden Passwörter nicht im Klartext gespeichert, weswegen eine Darstellung des vorherigen Wertes weder wünschenswert noch möglich ist.

An dieser Stelle sei angemerkt, dass die Darstellung und Validierung potenziell auch auf andere Art und Weise erfolgen könnte. HTML5 [HTM, Krö11] bietet neben den bisherigen Eingabefeldern einige neue Typen (wie z. B. `date` oder `email`). In MontiWIS wurde jedoch davon abgesehen, diese Feldtypen zu verwenden, da eine durchgängige Unterstützung durch alle Webbrowser in verschiedenen Versionen nicht gegeben ist. Selbst im Falle einer vollständigen Unterstützung bieten die Browser nicht exakt das anvisierte Verhalten. So werden

beispielsweise Attribute vom Typ `number` in der Regel über einen Spinner (d. h. zwei Pfeilknöpfe, über die eine Zahl um ein bestimmtes Intervall manipuliert werden kann) dargestellt, was auch die Eingabe und Manipulation von Dezimalzahlen erschwert. Die Darstellung eines `date` Datentyps erfolgt, sofern unterstützt, ebenfalls über ein Datumsauswahlfeld, dessen Darstellung jedoch browserspezifisch und damit nicht konsistent mit den übrigen Oberflächenelementen ist. Da die partielle Unterstützung durch die native Implementierung der Elemente der Benutzeroberfläche jedoch auch dazu führt, dass die Darstellung der Elemente nicht einheitlich ist, wurde generell vom Einsatz der HTML5-Alternativen abgesehen. Die Verwendung der eingesetzten jQuery Bibliotheken garantiert eine ausgezeichnete Unterstützung durch eine Vielzahl verschiedener Browser und ein einheitliches und konfigurierbares Erscheinungsbild durch die Verwendung von jQuery UI.

Umsetzung von Assoziationen

Nach den Attributen, die aus den vordefinierten Datentypen resultieren, soll nun auf die Attribute, die aus Assoziationen und Kompositionen herrühren, eingegangen werden. Wie bereits in Abschnitt 7.2.3 beschrieben, werden existierende Assoziationen als Auflistung der assoziierten Objekte dargestellt und sowohl das Löschen als auch das Hinzufügen von Objekten per Link ermöglicht. Abbildung 7.40 zeigt den generierten Teil einer JSP-Seite, der die Assoziation eines Studenten zu seinen assoziierten Studiengängen darstellt. Hierbei handelt es sich um die Editierdarstellung, in der ein Nutzer sowohl neue Objekte hinzufügen als auch bestehende Assoziationen löschen kann.

Dargestellt ist hier die Tabellenzeile, die zur Darstellung der Assoziation generiert wird. Zeile 2-4 enthält hierbei die Beschriftung des Attributs, welche in diesem Fall die standardmäßig generierte `RessourceBundle`-Referenz ist. Die zweite dargestellte Tabellenzelle enthält zunächst einen Link, der das Hinzufügen eines neuen Objekts erlaubt (Zeilen 6-13). Hierzu kommt die JavaScript-Methode `add` zum Einsatz, die insgesamt fünf Parameter benötigt. Der erste (`student.studentsof`) bezeichnet die `id` des `ul` Elements (Zeile 14), zu der die neuen Assoziationen hinzugefügt werden sollen, der zweite Parameter (beginnend mit `${pageContext.`) bezeichnet dir URL, unter der die Liste der JSON-Objekte mit einer Liste sämtlicher Objekte der referenzierten Klasse abgerufen werden kann, der dritte Parameter (`['title','kind']`) gibt an, welche Attribute der Objekte in der Auswahltabelle angezeigt werden sollen, der vierte Parameter (`student.studentof`) spezifiziert, unter welchem Namen die versteckten `input`-Elemente die gewählten Objekte speichern sollen und der fünfte Parameter schließlich bestimmt, ob es sich bei der Assoziation um eine mengenwertige handelt, was das Verhalten der Funktion beeinflusst. Im Falle einer mengenwertigen Assoziation werden die Objekte zur Aufzählungsliste hinzugefügt, anderenfalls werden potenziell vorhandene Objekte beim Hinzufügen ersetzt. Die Funktion sorgt für die Darstellung der Liste verfügbarer Objekte in einem Pop-up-Fenster und für das korrekte Hinzufügen der ausgewählten Objekte zur Liste der Objekte.

Die zugeordneten Objekte werden als ungeordnete Liste dargestellt, wie dies in den Zeilen 14-24 zu sehen ist. Dabei wird für jedes Objekt ein verstecktes Eingabefeld mit der `id` des Objekts (Zeilen 17-18), die darzustellenden Attributwerte (Zeile 19) sowie ein Link zum

JSP

```
1  <tr>
2    <td>
3      <fmt:message key="Student_studentOf" />
4    </td>
5    <td>
6      <a href="javascript:void(0);" onclick="javascript:add(
7        'student.studentOf',
8        '${pageContext.request.contextPath}/resources/CourseOfStudy/list',
9        ['title','kind'],
10       'student.studentOf',
11       true
12       );">add courseOfStudy
13     </a>
14     <ul id="student.studentOf">
15       <c:forEach items="${it.student.studentOf}" var="studentOf">
16         <li>
17           <input type="hidden" name="student.studentOf"
18             value="${studentOf.id}"/>
19             ${fn:escapeXml(studentOf.title)}  ${fn:escapeXml(studentOf.kind)}
20           <a href="javascript:void(0);"
21             onclick="javascript:remove(this)">Delete</a>
22         </li>
23       </c:forEach>
24     </ul>
25   </td>
26 </tr>
```

Abbildung 7.40.: Umsetzung von Assoziationen

Entfernen des jeweiligen Objekts (Zeilen 20-21) generiert. Ein ebensolcher Aufbau wird für neu ausgewählte Objekte in die Struktur der HTML-Seite durch Verwendung von JavaScript eingebaut. Das Entfernen eines solchen Konstrukts von der Seite erfolgt ebenfalls über eine Funktion, die die jeweiligen Teile aus der Seite entfernt.

Letztlich relevant dafür, welche Objekte zugeordnet sind, ist, welche bei Absenden auf der Seite vorhanden sind. Die korrekte Behandlung, also das Hinzufügen neuer bzw. das Entfernen nicht mehr zugeordneter Objekte geschieht vollständig auf Serverseite sowohl in den `PageEntity`- (siehe dazu Abschnitt 7.4.3) als auch den `DAO`-Klassen (siehe Abschnitt 6.4.1).

Umsetzung von Kompositionen

Die zweite Art von Assoziationen, die in MontiWIS behandelt wird, sind die Kompositionen. Hierbei werden die Objekte, die einem Objekt zugeordnet werden, direkt im Kompositum erzeugt und sind damit auch an den Lebenszyklus von diesem gekoppelt. Wie in Abschnitt 7.2.3 beschrieben, werden für die Darstellung separate Tabellen erzeugt, die jeweils die Attribute einer der Komponenten enthält. Die Implementierung in der JSP-Seite erfolgt, wie in

Abbildung 7.41 schematisch dargestellt. Hierbei handelt es sich um die Einbindung von Adressobjekten in eine Person (oder eine der daraus abgeleiteten Klassen) und die für das Editieren dieser Komposition notwendigen Konstrukte.

JSP

```
<script>
  var addresses = '<tr><td colspan="2">' + // Neue Eingabetabelle für Komponente
</script>

<!-- beliebige Teile der Seite -->

<tr>
  <td>
    <fmt:message key="Person_addresses" />
  </td>
  <td>
    <table id="addresses">
      <tr>
        <td colspan="2">
          <a href="javascript:void(0);" onclick="javascript:appendToElement(
            '#addresses',addresses);">add new</a>
        </td>
      </tr>
      <c:if test="${!empty it.student.addresses}">
      <c:set var="addresses" value="${fn:escapeXml(it.student.addresses)}" />
      <tr>
        <td colspan="2">
          <table>
            <tr>
              <td colspan="2">
                <a href="javascript:void(0);"
                  onclick="javascript:removeComposition(this);">remove</a>
              </td>
            </tr>
            <tr>
              <td>
                <fmt:message key="Address_street" />
              </td>
              <td>
                <input class="" type="text" name="addresses.street"
                  value="${fn:escapeXml(addresses.street)}" />
              </td>
           </tr>
         </table>
      </c:if>
    </table>
  </td>
</tr>
```

Abbildung 7.41.: Umsetzung von Kompositionen

Die Teile der generierten JSP-Seite, die für Darstellung und Funktionalität von Kompositionen, lassen sich in zwei Teile einteilen. Zunächst wird pro Komposition in einem `script`-Abschnitt eine Variable deklariert, die als Wert den HTML-Code enthält, der zum Hinzufügen einer neuen Komponente in die HTML-Seite eingebaut werden muss. Dies ist dargestellt in den Zeilen 1-3, wobei der Inhalt der Variablen hier nur angedeutet ist. Er entspricht aber im Wesentlichen den Konstrukten aus den Zeilen 21-38. Der eigentliche Teil der JSP-Seite für die Modifikation von Kompositionen ist in den Zeilen 7-43 dargestellt. Hier wird nach der generierten Bezeichnung des Attributs (Zeilen 8-10) eine Tabelle generiert (Zeilen 12-41), welche die `id addresses` besitzt. Diese leitet sich aus dem Attribut- bzw. Rollennamen der Komposition ab. Innerhalb dieser Tabelle wird als erste Zeile ein Link eingebunden, der zum Hinzufügen einer neuen Komponente dient (Zeilen 13-18). Hierzu wird die JavaScript-Funktion `appendToElement` verwendet, der die vorher generierte Variable mit den hinzuzufügenden HTML-Konstrukten sowie die `id` des HTML-Elements, dem dieser Inhalt angehängt werden soll, übergeben wird. Als Nächstes werden potenziell bereits zugeordnete Objekte ausgegeben, was auch in Tabellenform geschieht. Hierbei erhält jedes Objekt einen Link, mit dem das entsprechende Objekt aus der Seite entfernt werden kann (Zeilen 25-28). Hierzu wird die JavaScript-Funktion `removeComposition` aufgerufen, der das aktuelle HTML-Element übergeben wird. Dies wird von der Funktion verwendet, um selbiges inklusive der korrekten Elternelemente aus der Seite zu entfernen. Die restlichen Tabellenzellen sind hier nur angedeutet, entsprechen aber im Wesentlichen den gewöhnlichen Attributen, die aus den vordefinierten Datentypen generiert werden.

Auch bei den Kompositionen werden sämtliche Modifikationen zunächst clientseitig vorgenommen und erst beim Absenden der Seite die entsprechenden Objekte erzeugt bzw. gelöscht.

7.4.3. PageEntity- und PageEntityInitializer-Klassen

In Abschnitt 5.3.3 wurde bereits erläutert, welchen Sinn die `PageEntity` und die dazugehörigen `PageEntityInitializer`-Klassen haben. Nachfolgend wird nun der Aufbau dieser Klassen beschrieben.

PageEntity-Klassen

Die `PageEntity`-Klassen dienen im Wesentlichen dem Datentransfer in eine JSP-Seite hinein und daraus hinaus. Für jede Seite existiert eine ihr entsprechende `PageEntity`-Klasse, deren Aufbau derjenigen in Abbildung 7.42 entspricht.

Die darin dargestellte Klasse ist diejenige, die für eine Seite zur Darstellung bzw. zum Editieren eines `Student`-Objekts generiert wird. Die Klasse selbst wird von `DefaultPageEntity` abgeleitet (Zeile 1), welche sowohl Felder u. a. zur Übermittlung möglicher anzuzeigender und betätigter Knöpfe und Fehlermeldungen sowie die notwendigen Zugriffsmethoden wie getter- und setter-Methoden enthält, die in jeder `PageEntity`-Klasse zur Verfügung stehen müssen. Für jeden der Seitenparameter (im Falle einer Seite zum Anzeigen bzw. Editieren eines einzelnen Objekts einer Klasse also genau dieses) wird eine Variable (Zeile 3) sowie eine Menge, die sämtliche Objekte einer Klasse enthält (Zeile 5) als Klassenvariable generiert.

```java
public class StudentPageEntity extends DefaultPageEntity {

  private de.montiwis.entities.Student student;

  private Collection<Student> studentCollection;

  StudentPageEntity() {
    this.student = new de.montiwis.entities.Student();
  }

  public Collection<Student> getStudentCollection() {
    return this.studentCollection;
  }

  public void setStudentCollection(Collection<Student> coll) {
    this.studentCollection = coll;
  }

  public de.montiwis.entities.Student getStudent() {
    return this.student;
  }

  public void setStudent(de.montiwis.entities.Student e) {
    this.student = e;
  }
}
```

Java

Abbildung 7.42.: Umsetzung von `PageEntity`-Klassen

Im Konstruktor (Zeilen 7-9) wird die einzelne Klassenvariable mit einem neuen Objekt initialisiert. Ferner existieren sowohl für das einzelne Objekt als auch für die Menge aller Objekte getter- und setter-Methoden (Zeilen 11-25).

Eine `PageEntity`-Klasse besitzt damit lediglich Datenstrukturcharakter und hält die Daten vor, die auf einer bestimmten Seite angezeigt werden bzw. die Daten, die von einer Seite aus abgesandt wurden. Die Logik für die Belegung der Klassenvariablen und deren Attribute ist in der zur `PageEntity` gehörigen `PageEntityIntitializer`-Klasse ausgelagert und wird nachfolgend beschrieben.

PageEntityInitializer-Klassen

Abbildung 7.43 zeigt das `PageEntityInitializer`-Pendant für die Seite zur Anzeige oder zum Editieren eines `Student`-Objekts.

Eine `PageEntityInitializer`-Klasse ist, wie in Zeile 1 zu sehen, eine *Stateless Session Bean*, so dass der Java EE Server die Verwaltung von Objekten dieser Klasse übernehmen kann und sie damit ressourcensparend eingesetzt werden können. In den Zeilen 4-5 ist exemplarisch

Java

```
1  @Stateless
2  public class StudentPageEntityInitializer {
3
4    @EJB
5    private StudentRemoteDao studentDao;
6
7    // weitere DAO-Deklarationen
8
9    public StudentPageEntity createNewPageEntity() {
10     // erzeugt und initialisiert ein PageEntity-Objekt
11   }
12
13   public void setPageEntityValuesInGivenEntities(Student student,
14     StudentPageEntity pe) {
15     // weist die Werte der PageEntity den übergebenen Objekten zu
16   }
17
18   public boolean setFormValuesInPageEntity(MultivaluedMap<String, String> form,
19     StudentPageEntity studentPageEntity)  {
20     // initialiisiert die übergebene PageEntity mit den übergebenen Formwerten
21   }
22
23   private void initParameters(MultivaluedMap<String, String> form,
24     StudentPageEntity pe) throws ValidatorException {
25     // initialiisiert die übergebene PageEntity mit den übergebenen Formwerten
26   }
27
28   public StudentPageEntity setEntitiesInPageEntity(Student student,
29     StudentPageEntity entity) {
30     // setzt das übergebene Objekt in dem  übergebenen PageEntity-Objekt
31   }
32 }
```

Abbildung 7.43.: Umsetzung von `PageEntityInitializer`-Klassen

dargestellt, wie eine DAO-Klasse per `@EJB`-Annotation injiziert wird. Dies geschieht sowohl für jede Klasse, die als Klassenvariable in der zugehörigen `PageEntity`-Klasse behandelt wird (also in der Regel für jede Klasse, die als Seitenparameter einer Seite übergeben wird) sowie für jede daraus referenzierte Klasse. DAOs sind notwendig für das Laden von Klassenvariablen und referenzierten Klassen, etwa um Referenzen herstellen oder auch auflösen zu können. Die Fabrikmethode `createNewPageEntity` (Zeilen 9-11) dient zur Erzeugung und Initialisierung eines `PageEntity`-Objekts. Die Methode `setPageEntityValuesInGivenEntities` (Zeilen 13-16) dient dazu, die Klassenvariablen der `PageEntity`-Klasse an die entsprechenden Objekte, die der Methode übergeben werden, zuzuweisen. Über die Methode `setFormValuesInPageEntity` (Zeilen 18-21) werden die Werte, die in der `MultivaluedMap` enthalten sind, im übergebenen `PageEntity`-Objekt gesetzt. Die Logik hierzu ist in der priva-

ten Methode `initParameters` (Zeilen 23-26) enthalten, die von ihr aufgerufen wird. Das Gros der Logik dieser Klasse ist Teil eben dieser Methode. Hier werden die Werte zugewiesen, potenzielle Referenzen auf andere Objekte gesetzt bzw. diese im Bedarfsfall gelöscht sowie Komponentenklassen im Falle von Kompositionen erzeugt bzw. wieder entfernt.

Anzumerken ist an dieser Stelle, dass in dem vorliegenden Beispiel lediglich eine Klasse (`Student`) behandelt wird. Für `PageEntity`- bzw. die zugehörigen `PageEntityInitializer`-Klassen, die mehr als einen Seitenparameter besitzen, werden entsprechend mehrere Objekte den jeweiligen Methoden übergeben. Die Funktionsweise ist jedoch äquivalent zu dem hier beschriebenen.

Die Zuordnung von Formularwerten zu `PageEntity`-Klassenvariablen erfolgt über die Seitenvariablen, deren Aufbau und Namensschema im folgenden Abschnitt erläutert wird.

Aufbau der Seitenvariablen

Damit Inhalte auf Clientseite korrekt dargestellt werden können, ist es notwendig, in den einzelnen Formularfeldern bzw. bei der Ausgabe von Werten in einem Formular oder auch einer Überschrift oder Textabschnitt eindeutig auf die Werte auf Serverseite zugreifen zu können. Dazu dienen, wie bereits in Abschnitt 5.3.3 erläutert, die `PageEntity`-Klassen, die die für die Darstellung notwendigen Informationen und insbesondere die Seitenparameter enthalten. Diese stellen somit die Schnittstelle zwischen den Daten auf Serverseite und den JSP-Seiten dar. Der Zugriff in der JSP-Seite erfolgt über die Seitenvariable `${it}` [Bur10], die das `PageEntity`-Objekt repräsentiert. Da für jeden Seitenparameter in der `PageEntity` ein Feld mit gleichem Namen existiert und da für jedes dieser Felder getter-Methoden existieren, kann in einer Seite auf den Seitenparameter `Lecturer lec` über den Aufruf von `${it.lec}` zugegriffen werden. Da dessen Attribute wiederum ebenfalls getter-Metoden besitzen, lässt sich der `firstName` über `${it.lec.firstName}` auslesen. Auf Seitenparameter, die direkt vordefinierte Datentypen sind, wird ebenfalls über den Namen zugegriffen. Ein Parameter `String s` wird also folglich über `${it.s}` ausgegeben.

Ähnlich verhält es sich mit Assoziationen, die über einen eindeutigen Bezeichner identifiziert werden. Im Falle einer Assoziation eines `Lecturers` zu mehreren Gruppen (abgebildet über die Variable `groups`) erfolgt dies über die Seitenvariable `${it.lec.groups}`, auf die bei mehrfachem Vorkommen in einer JSP-Seite per Liste auf Serverseite zugegriffen werden kann. Diese Variable enthält die eindeutigen Bezeichner (`id`) der assoziierten Objekte, welche in der `PageEntityInitializer`-Klasse zu den zugehörigen Objekten aufgelöst und ihren konkreten Objektrepräsentationen zugeordnet werden.

Kompositionen nehmen auch hierbei einen weiteren Sonderfall ein. Ähnlich wie bei Assoziationen werden diese ebenfalls über den Variablennamen identifiziert, wobei nicht der Bezeichner, sondern jedes Attribut der Komponente abgebildet wird. Im Fall einer Komposition mit mehreren Adressen als Komponente (`addresses`) werden deren Straßenattribute (`street`) über die Seitenvariable `${it.lec.addresses.street}` in einer JSP-Seite identifiziert. Auch hier wird bei mehrfachem Vorkommen eines solchen Formularelements auf Serverseite per Liste in der Reihenfolge ihres Auftretens in der Seite zugegriffen.

7.4.4. Ressourcenklassen für Seiten

In Abschnitt 6.4.2 wurde bereits erläutert, wie die Ressourcenklassen für die Bearbeitung von Objekten aus dem Domänendatenmodell erzeugt werden, wie diese aufgebaut sind und welche Funktionen sie bereitstellen. Diese Ressourcenklassen besitzen insofern spezielle Funktionalität, als sie für das Laden und Speichern einzelner Objekte oder Mengen davon zuständig sind. Ferner sind die Seiten, die sie zur Darstellung verwenden, durch den Transformationsprozess aus dem Domänendatenmodell hervorgegangen.

Im Gegensatz dazu kommen die Seiten, die explizit über die Seitenbeschreibungssprache spezifiziert werden, in zweierlei Ausprägungen vor. Einerseits können Seiten Parameter besitzen, die gewöhnlich innerhalb einer Aktion als Teil einer Aktivität belegt werden. Diese Seiten werden daher auch als Teil der Ausführung einer Aktivität aufgerufen und bedürfen daher keiner Ressourcenklasse, die ja letztlich die Schnittstelle zum Nutzer darstellt und den direkten Aufruf der Seite erlaubt. Andererseits gibt es ebenfalls Seiten, die keine Seitenparameter besitzen und daher lediglich statischen Inhalt darstellen. Dies gilt beispielsweise für Seiten, die Kontaktinformationen darstellen, ein Impressum oder auch allgemeine Statusmeldungen enthalten. Diese Seiten können zwar potenziell auch innerhalb der Ausführung einer Aktivität als Teil einer Aktion aufgerufen, aber auch ebenso direkt in einem Menü als Teil der Applikationssprache verlinkt werden, so dass die generierte Seite über eine URL erreichbar sein muss. Für diese Erreichbarkeit existiert für jede Seite eine Ressourcenklasse, wie sie in Abbildung 7.44 dargestellt ist.

Java

```
1  @Path("/page/Contact")
2  public class ContactResource {
3
4    @Context
5    private HttpServletRequest request;
6
7    @GET
8    public Response getPage() {
9      if (IsUserInRolesTag.isUserInRoles(request, "guest")) {
10       Viewable viewable = new Viewable("/Contact.jsp", null);
11       return Response.ok(viewable).build();
12     } else {
13       Viewable viewable = new Viewable("/ResourceNotAllowed.jsp", null);
14       return Response.status(403).entity(viewable).build();
15     }
16   }
17 }
```

Abbildung 7.44.: Umsetzung von Ressourcenklassen für Seiten

Die Ressourcenklasse, die hier dargestellt ist, wurde erzeugt aus der Seite `Contact.pg`, die jedem nicht eingeloggten Nutzer zugänglich ist und deren Inhalt allgemeine Kontaktinformationen enthält. Wie in Zeile 1 zu sehen, ist die Seite unter der URL `/page/Contact`

zu erreichen, jeweils angehängt an die URL, unter der sämtliche Ressourcen innerhalb der generierten Applikation zu finden sind. Die Zeilen 4-5 enthalten Code, der innerhalb der Klasse Zugriff auf den mit dem Aufruf assoziierten `HttpServletRequest` gewährt. Dieser wird im Folgenden für die Prüfung der Berechtigungen benötigt. Da statische Seiten keine Seitenparameter besitzen und bei ihrem direkten Aufruf auch keine Möglichkeit zum Absenden bieten, ist lediglich die Anzeige der Seite relevant und muss in der Ressourcenklasse behandelt werden. Dies geschieht über die Methode `getPage` (Zeilen 8-16). Annotiert ist diese Methode mit `@GET`, wodurch sie bei gewöhnlichem HTTP `GET`-Aufruf ausgeführt wird. Die Logik innerhalb dieser Methode ist sehr überschaubar. Hierin wird geprüft, ob der aufrufende Nutzer die Berechtigung besitzt, die Seite zu sehen (Zeile 9). Sollte dies der Fall sein, wird die JSP-Seite entsprechend als Antwort (gekapselt in einem `Response`-Objekte) zurückgesandt, anderenfalls wird eine Seite zusammen mit dem HTTP-Statuscode 403 (was einem *forbidden* entspricht) zurückgegeben, die den Nutzer auf fehlende Berechtigung hinweist (`ResourceNotAllowed.jsp` in Zeile 13).

Die Berechtigungen, die innerhalb dieser Methode geprüft werden, resultieren aus den Zugriffsbeschränkungen, die innerhalb der Applikationssprache deklariert wurden. Eine Erläuterung der möglichen Sprachmittel dazu ist in Kapitel 9 zu finden.

7.4.5. Sicherheitsüberlegungen

Gerade bei der Entwicklung von Web-Informationssystemen ist es von essenzieller Bedeutung, dass diese im Hinblick auf Sicherheit einen bestimmten Grundschutz gewährleisten. An dieser Stelle soll darauf eingegangen werden, wie diese Aspekte in MontiWIS berücksichtigt werden.

Zeichenmaskierung von Ausgaben

Cross Site Scripting (XSS) Angriffe sind eine ernst zu nehmende Bedrohung [OWA]. Hierbei handelt es sich um das Einschleusen von ausführbarem JavaScript-Code in eine Webseite, um dadurch die Webseite zu manipulieren oder Informationen daraus zu extrahieren. Dies kann auf unterschiedliche Art und Weise geschehen. Einerseits kann der Code über eine URL in der Art an eine Webseite gesendet werden, dass Teile der URL in der Seite dargestellt werden, andererseits kann auch in den gespeicherten Daten in der Datenbank Code enthalten sein, der bei Aufruf und Ausgabe der Daten in der Webseite dargestellt und ausgeführt wird. Dies geschieht beispielsweise über das Einfügen von `<script>`-Tags (zum Teil auch auf verschlüsselte oder bereits maskierte Art), die den entsprechenden Code enthalten und ausgeführt werden.

Um dies zu verhindern, werden in MontiWIS sämtliche Ausgaben, die potenziell von Nutzereingaben herrühren, bei der Ausgabe auf der JSP-Seite maskiert, d. h. sämtliche Zeichen, die als Teil von HTML interpretiert werden könnten (und damit beispielsweise auch das soeben erwähnte `<script>`-Tag), werden durch sogenannte *named character entities* ersetzt. Damit werden sie zwar in gewöhnlicher Art und Weise in der HTML-Seite dargestellt, aber nicht als Teil von HTML interpretiert. Dies geschieht durch die Anwendung der `fn:escapeXml()`-Funktion der JSTL-Formatierungs-Tagbibliotheken [SCR01]. Dieser Funktion wird der auszugebende Ausdruck übergeben, woraufhin eine Ersetzung sämtlicher

potenziell gefährlicher Zeichen vorgenommen wird. Wie dieser Ausdruck umgesetzt wurde, ist beispielsweise in Abbildung 7.39 zu sehen.

Verwendung von Objektrelationalem Mapping

Ungeprüfte Eingaben in Webformularen können auf Serverseite auch in anderer Hinsicht problematisch sein. Sollten die Eingaben verwendet werden, um in SQL-Ausdrücken beispielsweise als `WHERE` oder `LIKE` eingesetzt zu werden, können über bestimmte Ausdrücke die resultierenden Befehle manipuliert und damit potenziell Schadcode ausgeführt werden. Diese Art der Manipulation wird im Allgemeinen *SQL Injection* genannt.

Als Gegenmaßnahme können, wie in MontiWIS, sogenannte *prepared statements* bzw. die Abfragesprache des JPA-Frameworks zum Einsatz kommen. Dabei werden die Datenbankabfragen in der Art vorbereitet, dass die `WHERE` oder `LIKE` Ausdrücke als Parameter in einen bereits vorkompilierten Ausdruck eingesetzt werden, so dass dieser nicht mehr nachträglich verändert werden kann.

Verschlüsselter Passwortspeicher

Wie bereits in Abschnitt 6.2.3 erwähnt, werden Passwörter in MontiWIS nicht in einer Klartextrepräsentation in der Datenbank gespeichert. Stattdessen erfolgt die Anwendung einer MD5-Hashfunktion auf das Passwort, was dazu führt, dass das Passwort nicht rekonstruierbar ist, aber dennoch durch erneute Anwendung der Hashfunktion auf ein eingegebenes Passwort überprüft werden kann. Problematisch ist bei einfachen Hashfunktionen jedoch, dass sog. *Rainbowlisten* (vgl. [Oec03]) existieren, die für ganze Wörterbücher von Passwörtern die zugehörigen Hashwerte enthalten, so dass durch einfachen Vergleich Rückschlüsse auf das ursprüngliche Passwort gezogen werden können. Um dies zu verhindern, sollten sogenannte *gesalzene* Passwörter verwendet werden. Darunter versteht man das Anhängen einer beliebigen Zeichenkette, die den Vergleich von Wörtern eines Wörterbuchs mit den standardmäßigen Hashes verhindert.

Im optimalen Fall wäre es wünschenswert, dass für jedes Passwort ein eigener *Seed* (also die Zeichenkette, die an das eigentliche Passwort angehängt wird) verwendet und separat gespeichert wird. Dies wird theoretisch auch von dem in MontiWIS verwendeten Framework Flexible JDBC Realm unterstützt, da es darin möglich ist, bei der Überprüfung der Zugangsdaten anzugeben, über welche SQL-Anweisung der jeweilige Seed geladen werden soll, so dass diese Zeichenkette zu einem Nutzer gespeichert werden kann. Dies geschieht auch in MontiWIS.

Problematisch ist jedoch, dass über diese Methode der Seed direkt über ein Nutzerobjekt bezogen werden muss. Das Beispiel in Abbildung 7.8 verdeutlicht dies. Hier werden zwei Passwörter abgefragt. Eines, welches direkt einem `Lecturer`-Objekt (und damit einem Nutzerobjekt) zugeordnet ist, und eines, welches losgelöst davon existiert. Da für die erfolgreiche automatische Berechnung des Hashwertes das Passwortattribut zu einem Nutzerobjekt zugeordnet sein muss, kann dieser ausschließlich im ersten Fall berechnet werden. Im zweiten Fall existiert diese Möglichkeit nicht. Daher wird in MontiWIS stets der gleiche Seed für die Berechnung aller Hashes verwendet. Dieser wird ebenfalls in der Tabelle zum Nutzerobjekt

gespeichert, damit er bei der Authentifizierung eines Nutzers bereitsteht. Innerhalb des Quellcodes kann jedoch statisch darauf zugegriffen werden, so dass auch der Vergleich der beiden Passwörter aus dem Beispiel möglich ist und das erläuterte Problem damit umgangen wird.

Dieser pragmatische Ansatz bietet nicht den Grad an Sicherheit, wie er bei unterschiedlichen Seedwerten für jedes Nutzerobjekt bestehen würde. Dennoch wurde dieser Weg in MontiWIS beschritten, so dass die Überprüfung zweier Passwörter bzw. deren Hashes auf Gleichheit innerhalb des Programmcodes möglich ist.

7.5. Andere Ansätze

[dS01] kategorisiert Ansätze für Modellierungsansätze von Benutzerschnittstellen und gibt einen Überblick über eine Vielzahl solcher Ansätze. [dSP03, dS02] spezifiziert eine Benutzerschnittstelle über erweiterte UML Notationen. Nutzerinteraktionen auf Clientseite werden dabei über Aktivitätsdiagramme modelliert, wohingegen für den Aufbau der Seite eine domänenspezifische, grafische Notation verwendet wird. Ähnlich wie bei MontiWIS können Input-/Output- Felder separat angegeben werden.

Während der Entwicklung der Seitenbeschreibungssprache in MontiWIS wurden einerseits die Lehren aus der Entwicklung von MontiWeb gezogen, andererseits aber auch andere Möglichkeiten der Seitenspezifikation betrachtet und abgewägt.

Generell kann man bei der Spezifikation unterscheiden, ob ein *imperativer Ansatz* verfolgt wird, wobei beschrieben wird, *wie* die Seite aufgebaut werden soll, oder ob *deklarativ* beschrieben wird, *was* auf einer Seite dargestellt werden soll. Diese beiden Möglichkeiten werden im Folgenden betrachtet und deren Vor- und Nachteile diskutiert.

7.5.1. Imperativer Ansatz

Unter einem *imperativen Ansatz* zur Seitenbeschreibung werden Methoden verstanden, bei denen explizit beschrieben wird, wie die Seite erstellt und dargestellt wird. Ein Beispiel für eine imperative Seitenbeschreibung ist in Abbildung 7.45 dargestellt. Hier dient als Beispiel ein minimaler Ausschnitt aus einer HTML-Seite, die durch Verwendung der Freemarker [Fre] Templatesprache um dynamische Anteile erweitert wird.

Anhand dieses Beispiels wird deutlich, was eine imperative Spezifikation der Seite ausmacht. Bezeichnend für diese Art der Sprache ist, dass Konstrukte wie beispielsweise in Zeilen 4 bzw. 6-8 elementarer Bestandteil der Beschreibung sind. Der Ausdruck `${heading}` in Zeile 4 stellt eine Referenz auf einen skalaren Wert dar, an dessen Stelle z. B. der Wert der entsprechenden Variablen einzusetzen ist. Die Zeilen 6-8 enthalten Kontrollstrukturen der Freemarker-Sprache, die das Iterieren über eine mengenwertige Variable (in diesem Fall `entities`) bewirken, was dazu führt, dass der Inhalt der Kontrollstruktur pro enthaltenem Objekt ausgegeben wird. Neben der Verwendung besagter Kontrollstrukturen ist es ebenfalls typisch für imperative Ansätze, dass die HTML-Konstrukte direkt in der Seitenspezifikation eingebunden sind. Dies hat zwar zur Folge, dass prinzipiell alle Möglichkeiten der Seitengestaltung zur Verfügung stehen, aber auch, dass hier keine Abstraktion von der eigentlichen Struktur einer Seite stattfindet. Neben dem erhöhten Aufwand, den ein solches Vorgehen

FM

```
<html>
  <head> <!-- ... --> </head>
  <body>
    <h1>${heading}</h1>
    <ul>
      <#list entities as entity>
      <li>${entity.text}</li>
      </#list>
    </ul>
  </body>
</html>
```

Abbildung 7.45.: Beispielhafter Aufbau eines Freemarker-Templates

bedeutet, lassen sich darüber auch relativ schwer konsistente Darstellungen innerhalb der Applikation erreichen. Diese Problematik tritt besonders deutlich hervor, wenn, wie in Abbildung 7.45, direkt HTML als Beschreibungssprache zusätzlich zu den Template-Konstrukten verwendet wird. Hier ist nicht nur die Gefahr inkonsistenter Darstellung über Seiten hinweg sehr groß, sondern auch die Möglichkeiten, Seiten syntaktisch falsch zu codieren, indem beispielsweise die HTML-Struktur nicht valide ist oder, sofern Seiten mit anderen Teilen wie Kopf- oder Fußzeilen ergänzt werden, in deren Kontext kein syntaktisch korrektes Dokument bilden oder dessen vorgesehene Struktur zerstören. Neben der im Beispiel verwendeten Freemarker-Syntax wäre es ebenfalls möglich, Apache Velocity [Vel], JSP [Ber03], eine beliebige andere Templatesprache oder eigens geschriebene Konstrukte zu verwenden.

Zu beachten ist hierbei, dass die Variablen, die innerhalb dieser Seite referenziert werden, an anderer Stelle entsprechend mit Werten belegt werden müssen. Wie dies im Einzelnen geschieht und wie die Referenz syntaktisch realisiert ist, bleibt hier offen und wird nicht weiter diskutiert.

Die angesprochenen Probleme hinsichtlich Aufwand und Fehleranfälligkeit bei der direkten Verwendung von HTML lassen sich jedoch entscheidend verringern, indem anstatt direkt HTML eine domänenspezifische Sprache eingesetzt wird, die wiederum von den konkreten HTML-Elementen abstrahiert. Dieser Ansatz wird beispielsweise im Fall von WebDSL [Vis08b, Webc] unterstützt. Hier werden zwar domänenspezifische, deklarative Konstrukte zur Spezifikation der Seitenstruktur eingesetzt, jedoch sind zusätzlich Kontrollstrukturen wie z. B. Schleifen möglich. Dieser Ansatz stellt somit eine Zwischenlösung zwischen deklarativem und imperativem Vorgehen dar.

Einen komplexeren Ansatz bietet die Interaction Flow Modeling Language (IFML) [IFM, Obj13b]. Dieser Ansatz befindet sich aktuell in einem Standardisierungsprozess durch die Object Management Group (OMG) und basiert im Wesentlichen auf Konzepten, die aus WebML stammen, weswegen WebRatio (die Firma hinter WebML) auch eine treibende Kraft hinter dieser Spezifikation ist. Über diese Sprache lassen sich Nutzerschnittstellen beschreiben, wobei explizit das Verhalten auf Clientseite mit modelliert wird. Einzelne Komponenten werden

per Event- und Datenfluss mit potenziellen Geschäftslogikabschnitten verknüpft, worüber das Verhalten von Komponenten der Benutzerschnittstelle spezifiziert wird. Im Gegensatz zu MontiWIS ist diese (grafische) Modellierungsmöglichkeit zwar wesentlich vollständiger, aber auch wesentlich komplizierter und schwergewichtiger zu verwenden.

7.5.2. Deklarativer Ansatz

Im Gegensatz zum soeben beschriebenen imperativen Ansatz verfolgt ein *deklarativer Ansatz* eine wesentlich abstraktere Strategie. Hierbei wird nicht mehr codiert *wie* die Seite erzeugt und dargestellt wird, sondern lediglich, *was* darauf in welcher semantischen Funktion erscheinen soll. Ein Beispiel für eine grafische, deklarative Syntax ist in Abbildung 7.46 gegeben.

Abbildung 7.46.: Beispiel einer deklarativen Seitenbeschreibung in UWE (vgl. [KKZB08])

Das Beispiel entstammt einer Präsentationsklasse, wie sie in UWE [KKZB08] verwendet wird um festzulegen, wie bestimmte Attribute auf einer Seite angeordnet und dargestellt werden. Auf die einzelnen Attribute wird durch Angabe ihres Namens verwiesen, die Piktogramme im jeweiligen Feld geben an, in welcher Weise ein Attribut dargestellt werden (z. B. als Text im Falle von `title` oder `year` und als Link im Falle von `officialSite`). Wie hier gut ersichtlich ist, wird bei einem deklarativen Ansatz spezifiziert, welche Attribute dargestellt werden sollen und wie dies geschehen soll. Zwar ist hierbei nicht zwangsläufig notwendig, dass die Art, wie ein Attribut präsentiert werden soll, explizit mit angegeben wird (wie im Beispiel in Abbildung 7.46 zu sehen), aber in manchen Fällen kann dies dennoch sinnvoll sein, etwa wenn es mehrere Möglichkeiten zur Darstellung eines bestimmten Attributs geben könnte und zwischen den Alternativen ausgewählt werden soll. Dies widerspricht jedoch nicht dem deklarativen Charakter des Beispiels.

Neben dem im Beispiel verwendeten Ansatz UWE existieren noch weitere Ansätze, die ebenfalls eine deklarative Strategie verfolgen. So werden auch in WebDSL [Vis08b, Webc] deklarative Konzepte verwendet, die jedoch mit imperativen Konzepten zusammengebracht wurden. Auch in *OOHDM* [RS08] werden mit den Abstract Interface Designs deklarative Ansätze zur Seitenbeschreibung eingesetzt. Ebenso lässt sich Seiten in *RMM* [ISB95] über sogenannte Slices spezifizieren, was auch deklarativ erfolgt.

Kapitel 8.

Ablaufsteuerung

8.1. Grundlagen und Anforderungen

8.1.1. Grundlagen

Mit den bislang beschriebenen Sprachmitteln ist es zum einen möglich, über *Klassendiagramme* die Datenstruktur des zu entwickelnden Web-Informationssystems zu beschreiben und durch entsprechend generiertes Standardverhalten einzelne Objekte von Klassen des Domänendatenmodells anzulegen, zu verändern und anzuzeigen. Zum anderen bietet die *Seitenbeschreibungssprache* die Möglichkeit, sowohl statische Seiten zu spezifizieren, als auch Objekte bzw. partielle Sichten darauf anzuzeigen und sie zu verändern, wobei auch zu diesen Seiten statische Anteile hinzugefügt werden können. Mit den vorliegenden Mitteln gibt es jedoch keine Möglichkeit, die spezifizierten Seiten zu dezidierten Zeitpunkten und in festgelegter Reihenfolge anzuzeigen. Dazu sind Sprachmittel zu schaffen, die das Navigieren zwischen den Seiten nicht nur in linearer Art und Weise, sondern auch mit alternativen Wegen aufgrund von Auswahl des Nutzers oder der Evaluierung anderweitiger Bedingungen, erlauben. Auch die Objekte des Domänendatenmodells lassen sich, abgesehen von den generierten Standardseiten, nicht den Seiten übergeben und damit darstellen oder bearbeiteten geschweige denn aus der Datenbank lesen, speichern oder in anderer Art verarbeiten. Dies muss vor und nach dem Anzeigen einer Seite möglich sein. All diese Aspekte können unter dem Oberbegriff der *Geschäftslogik* der Anwendung subsumiert werden. Damit geht auch einher, dass sämtliche Funktionalität in den Modellen spezifiziert ist oder aber aus diesen heraus aufgerufen werden kann. All diese Aspekte werden im folgenden Kapitel beleuchtet.

8.1.2. Anforderungen an die Ablaufsteuerung

Der nachfolgende Abschnitt definiert, welche Anforderungen weiterhin an die sprachlichen Mittel zur Spezifikation von Geschäftslogik und Ablaufsteuerung gestellt werden. Dies betrifft sowohl die Ausdrucksfähigkeit der Sprache als auch die Erfordernisse, die an die intuitive Semantik der einzelnen Sprachelemente gestellt werden.

Verlinkung von Seiten

Die Seitenbeschreibungssprache bietet keine Möglichkeit, einzelne Seiten untereinander zu verlinken. Dies ist der bewussten Entscheidung geschuldet, dass Seiten nicht nur in einem,

sondern in mehreren Kontexten und an verschiedenen Stellen in unterschiedlichen Aktionen eingesetzt werden sollten. Ferner würden Verknüpfungen direkt auf einer Seite das Verlassen von vorgesehenen Abläufen erlauben, was nicht gewünscht ist. Aus diesen Gründen sollte es über eine externe Ablaufspezifikation möglich sein, Reihenfolgebeziehungen zwischen Seiten definieren zu können. Hierbei sollten sowohl lineare Abläufe, in denen in Art eines Wizards eine Seite nach der anderen angezeigt wird, als auch komplexere Konstrukte möglich sein. Wünschenswert ist beispielsweise, einem Nutzer Wahlfreiheit zu lassen, welche Seite als Nächstes angezeigt werden soll, aber auch, die Anzeige einer Seite abhängig von beliebigen anderen Kriterien (beispielsweise von den eingegebenen Werten) zu machen.

Generell sollte eine Seite im Kontext einer in sich geschlossenen Logikeinheit aufgerufen werden, die sowohl Eingabe- und Ausgabeparameter besitzt und die neben oder anstatt der Seitenpräsentation auch Geschäftslogik enthalten kann. Diese Aspekte werden im Folgenden beleuchtet.

Abbildung von Datenfluss

Neben der Möglichkeit zu spezifizieren, in welcher Reihenfolge Seiten angezeigt werden sollen, sollten auch Daten von Seite zu Seite bzw. Einheit zu Einheit transportiert werden können. Damit lassen sich Datenobjekte entlang eines Ablaufs weitergeben, ohne dass diese zwangsläufig zwischenzeitlich in der Datenbank gespeichert werden müssen. Ferner können aufgrund der transferierten Objekte die nächsten Schritte im Ablauf determiniert werden, indem ihre Werte hinsichtlich bestimmter Bedingungen evaluiert werden.

Um explizit Daten von einer Komponente zu einer anderen weitergeben zu können, sollten daran sowohl Eingabe- als auch Ausgabeparameter vorhanden sein, die explizit verknüpft werden.

Einbinden von beliebiger Geschäftslogik

Um ein Web-Informationssystem vollständig beschreiben zu können, sind nicht nur Datenstruktur und Seiten von Belang, sondern auch potenziell komplexe Geschäftslogik, die vor, nach oder auch insbesondere statt der Anzeige einer Seite ausgeführt wird. Geschäftslogik kann zusammen mit der Anzeige einer Seite in einer Komponente gekapselt sein, oder aber auch als eigenständige Einheit existieren, die in sich abgeschlossen ist und die ebenfalls die Möglichkeit der Aufnahme und Weitergabe von Datenobjekten bietet.

An dieser Stelle wird nicht zwangsläufig festgelegt, welcher Syntax die eingebundene Aktionssprache zu folgen hat. Potenziell möglich wären beispielsweise Aktionssprachen wie die *Action Language for Foundational UML (Alf)*[Obj13a], aber auch beliebige andere, eher abstrakte Sprachen. Da MontiWIS jedoch im Kontext der UML/P existiert und dementsprechend auch *programmier-geeignet* sein soll, wird sich an Java als Syntax orientiert.

Dennoch existieren für die Spezifikation der Geschäftslogik einige Anforderungen. So sollte es zum einen möglich sein, nicht nur per Aktionssprache einfache Ausdrücke direkt in der Ablaufsprache zu spezifizieren, sondern auch externen Code (z. B. in Form von API Aufrufen) einzubinden. Somit lassen sich beliebig komplexe, auch extern definierte, Algorithmen ausführen, deren Spezifikation in der Ablaufbeschreibung der Übersichtlichkeit halber nicht

angebracht wäre. Konzepte zur Integration handgeschriebenen und generierten Codes, wie sie beispielsweise in [GHK+15] beschrieben wurden, sind in MontiWIS insofern nicht notwendig, als die Integration und der Aufruf externen Codes direkt innerhalb der Modelle erfolgt.

Weiterhin sollten in der Aktionssprache soweit wie möglich Referenzen auf Konstrukte vermieden werden, die auf Modellebene nicht existieren, sondern erst über den Generator erzeugt werden. Sollte Zugriff auf generierte Funktionalität benötigt werden, so sind dafür idealerweise spezielle Konstrukte zu verwenden, die unabhängig von generierter Syntax sind. Auf diese Weise werden Abhängigkeiten zwischen Modell und Generator bzw. generiertem Code minimiert, so dass die Implementierung potenziell austauschbar ist.

Verteilte Abläufe

In einem Web-Informationssystem, das explizit Unterstützung für Abläufe bietet, ist es wünschenswert, dass Abläufe zwischen mehreren Akteuren verteilt werden können. Über derartige Konstrukte ist es beispielsweise möglich, einen Ablauf schrittweise unter verschiedenen Personen und Rollen innerhalb der Applikation zu verteilen. Anwendungsfälle für verteilte Abläufe sind beispielsweise die Bearbeitung eines Objekts, bei der unterschiedliche Akteure unterschiedlichen Zugriff auf bestimmte Attribute haben, oder aber auch die Abwicklung von Kauftransaktionen zwischen Käufer und Verkäufer in einem Onlineshop.

Speicherung und Wiederaufnahme von Abläufen

Damit ein Ablauf über mehrere Nutzer verteilt werden kann, sollte sein Zustand gespeichert werden können. Dies beinhaltet den Stand des Ablaufs sowie die Werte, die zum aktuellen Zeitpunkt auf Eingabeparametern einer Komponente vorliegen. Darüber können mehrere Nutzer gemeinsam an einem Ablauf arbeiten, aber auch ein Nutzer einen Ablauf zu einem späteren Zeitpunkt wieder aufnehmen und fortsetzen.

Zugriff auf Metafunktionalität

Um in den Modellen möglichst unabhängig von generiertem Code zu sein und möglichst wenig Annahmen über die Ergebnisse des Generators vorauszusetzen, sollte nach Möglichkeit kein Zugriff auf die Syntax von generierter Funktionalität in den Modellen der Ablaufbeschreibung erfolgen. Dies hat zur Folge, dass beispielsweise für die Speicherung oder auch das Laden von Objekten (was im generierten System über DAO Klassen geschieht, vgl. hierzu Abschnitt 6.4) ein separates Sprachkonstrukt existieren sollte, das diesen Zugriff erlaubt. Solche Konstrukte sollten auch für Funktionalität existieren, die beispielsweise den Zugriff auf Serverfunktionen (wie etwa den Abruf des aktuellen Nutzers) oder Funktionen der Laufzeitumgebung (z. B. um Rollenzuordnungen vorzunehmen) ermöglichen.

Die Konstrukte sollten in den Komponenten der Ablaufbeschreibungssprache verwendbar sein, so dass sie in der chronologischen Ausführung der Geschäftslogik eingebunden werden können.

Hierarchische Abläufe

In Web-Informationssystemen existieren häufig Abläufe, die derart komplex sind, dass eine Aufteilung in mehrere Unterabläufe sinnvoll erscheint. Dies sollte über die Ablaufbeschreibungssprache in MontiWIS ebenfalls möglich sein. Hierzu sollte das Verhalten einzelner Komponenten durch andere Abläufe definierbar sein, so dass bei Ausführung eben dieser Komponente der separate Ablauf gestartet und nach dessen Ende der ursprüngliche aufrufende Ablauf fortgesetzt wird. Dies entspricht der hierarchischen Dekomposition von Komponenten, was auch die Wiederverwendung bestimmter Teile von Abläufen erlaubt.

8.2. Sprache

Nach der Definition von Anforderungen an die Sprache und die Funktionalität, die damit in der generierten Applikation abgebildet werden soll, wird nun auf die Grammatik der Sprache und die Sprache selbst eingegangen, die in MontiWIS entwickelt wurde.

Da mit den UML/P Klassendiagrammen bereits ein Profil der UML zum Einsatz kommt, wurde entschieden, auch für die Ablaufsteuerung eine Sprache der UML zu nutzen. Hierbei fiel die Wahl auf *Aktivitätsdiagramme*, da sie von ihrer Ausführung zum Paradigma der Ablaufsteuerung passen und den Anforderungen, die in Abschnitt 8.1.2 aufgestellt wurden, genügen. Auf die Anforderungen, die bestimmte Konzepte erfüllen, wird an entsprechender Stelle hingewiesen.

Aktivitätsdiagramme selbst sind Teil des UML Standards [Obj11c]. Eine sehr umfangreiche Einführung in die Elemente und die Semantik von UML Aktivitäten ist in [Boc03a, Boc03b, Boc03c, Boc04a, Boc04b] gegeben. [RvdATW06] untersucht Aktivitätsdiagramme hinsichtlich ihrer Eignung für die Modellierung von Geschäftsprozessen anhand der Unterstützung von Workflow Mustern [vdATKB03]. Da in MontiWIS lediglich eine Teilmenge der in der UML definierten Konstrukte verwendet wird, lassen sich deren Ergebnisse nicht ohne Weiteres auf die hier verwendeten Aktivitätsdiagramme übertragen.

8.2.1. Aufbau der Aktivitätsdiagrammsprache

In diesem Abschnitt wird zu Beginn der Aufbau der Aktivitätsdiagrammsprache beschrieben. Dazu werden zunächst kurz die Grammatiken erläutert, die zur Definition der Sprache verwendet wurden, wie diese aufgebaut sind und welche Designentscheidungen dabei getroffen wurden.

Abbildung 8.1 zeigt eine vereinfachte Form der Grammatik für die Aktivitätsdiagramme, wie sie im Kontext dieser Arbeit entwickelt wurde[24].

Diese Grammatik wird über den Vererbungsmechanismus von MontiCore für Sprachdefinitionen (vgl. Abschnitt 3.3 bzw. [Kra10, Völ11]) von den UML/P Common (`mc.umlp.common.Common`) abgeleitet (Zeile 1, entwickelt in [Sch12] und vollständig abgebildet in Abschnitt C.3). Damit sind in der Aktivitätsdiagrammsprache bereits Konstrukte wie Stereotypen verfügbar,

[24] Insbesondere wurden der Übersichtlichkeit halber einige Konstrukte wie beispielsweise syntaktische Prädikate weggelassen. Die vollständige Grammatik mit sämtlichen Details ist im Anhang in Abschnitt C.7 zu finden.

aber auch die kompletten `mc.types.Types` (s. Abschnitt C.2) und `mc.literals.Literals` (s. Abschnitt C.1), von denen die `Common`-Grammatik transitiv ihre Elemente erbt. An dieser Stelle werden nicht sämtliche Elemente der Grammatik und deren exakter Aufbau beschrieben, vielmehr erfolgt lediglich ein Überblick über ihre generelle Struktur und Designentscheidungen, die bei ihrer Entwicklung getroffen wurden. Für eine detaillierte Erläuterung der einzelnen möglichen Elemente der Sprache sei auf Abschnitt 8.2.2 verwiesen.

MCSimp

```
1  grammar Activitydiagram extends mc.umlp.common.Common {
2
3    external ADContent;
4
5    interface ADElement;
6    interface ADAssignment;
7
8    ADDefinition =
9      Stereotype? "activity" name:Name
10     "{" ADInputParameters? ADOutputParameters? (aDElements:ADElement)* "}";
11
12   ADAction implements ADElement =
13     Stereotype? "action" name:Name
14     ( "{"
15       ADInputParameters? ADOutputParameters? ADVariableDeclaration?
16       ( ADIncludeStatement | (ADActionContent)* )
17       (ADAssignment)*
18     "}" )
19     |
20     ";";
21
22   ADVariableDeclaration  =
23     "var" ":" params:ADParameter ("," params:ADParameter)* ";";
24
25   ADInputParameters =
26     "in" ":" params:ADParameter ("," params:ADParameter)* ";";
27
28   ADOutputParameters  =
29     "out" ":" params:ADParameter ("," params:ADParameter)* ";";
30
31   ADParameter =
32     type:Type name:Name;
33
34   ADIncludeStatement =
35     "includes" includedActivity:Type ";";
36
37   ADActionContent =
38     kind:Name ":" ADContent(parameter kind);
39
40   ADHierarchicalInputAssignment implements ADAssignment =
41     (inputParam:Name "->" childActivityName:Name "." parameterName:Name) ";";
42
43   ADHierarchicalOutputAssignment implements ADAssignment =
```

```
44    (childActivityName:Name "." parameterName:Name "->" outputParam:Name) ";";
45
46  ADRole implements ADElement =
47    Stereotype? "role" name:Name
48    "{" actionNames:Name ("," actionNames:Name)* ";" "}";
49
50  ADDecisionNode implements ADElement =
51    Stereotype? "decision" name:Name ( ";" | "{" ADBasicAssignment* "}" );
52
53  ADMergeNode implements ADElement =
54    Stereotype? "merge" name:Name  ( ";" | "{" ADBasicAssignment* "}" );
55
56  ADJoinNode implements ADElement =
57    Stereotype? "join" name:Name ( ";" | "{" ADBasicAssignment* "}" );
58
59  ADForkNode implements ADElement =
60    Stereotype? "fork" name:Name ( ";" | "{" ADBasicAssignment* "}" );
61
62  ADEdge implements ADElement =
63    ADLeftEdgeSide "->" ADRightEdgeSide ";";
64
65  ADLeftEdgeSide =
66    "initial" | ADJoinEdgeSide | ADMergeEdgeSide | ADNode;
67
68  ADRightEdgeSide =
69    "final" | "flowfinal" | ADForkEdgeSide |
70    ADSplitEdgeSide | ADConditionalSplitEdgeSide | ADNode;
71
72  ADForkEdgeSide =
73    forkNodes:ADNode ("&" forkNodes:ADNode)+;
74
75  ADJoinEdgeSide =
76    joinNodes:ADNode ("&" joinNodes:ADNode)+;
77
78  ADSplitEdgeSide =
79    terms:ADSplitSideTerm ("|" terms:ADSplitSideTerm)+;
80
81  ADSplitSideTerm =
82    ADNode | "final" | "flowfinal";
83
84  ADMergeEdgeSide =
85    mergeNodes:ADNode ("|" mergeNodes:ADNode)+;
86
87  ADConditionalSplitEdgeSide =
88    terms:ADConditionalSplitTerm ("|" terms:ADConditionalSplitTerm)+ ;
89
90  ADConditionalSplitTerm =
91    (kind:Name ":")? "[" InvariantContent(parameter kind) "]" ADSplitSideTerm;
92
93  ADNode =
94    name:Name ("." pinName:Name)? ;
95
96  ADBasicAssignment =
```

```
97      outputParam:Name "=" inputParam:Name ";";
98
99  }
```

Abbildung 8.1.: Vereinfachte Darstellung der Aktivitätsdiagramm-Grammatik

Die Grammatik der Aktivitätsdiagramme in MontiWIS beginnt mit dem Nichtterminal `ADDefinition` (Zeilen 8-10), welches die Definition einer Aktivität mit all ihren Elementen repräsentiert. Dieses enthält neben Eingabe- und Ausgabeparametern insbesondere eine beliebige Anzahl von `ADElements`. Diese Schnittstelle (Zeile 5) steht stellvertretend für sämtliche Elemente, die die Struktur und Funktionalität einer Aktivität ausmachen, also beispielsweise Aktionsdefinitionen, Rollendeklarationen, die Definition weiterer Kontrollknoten oder die Verknüpfung von Knoten über Kanten.

Eine Aktion (Zeilen 12-20) als elementarer Bestandteil einer Aktivität beinhaltet zunächst optional Ein- und Ausgabeparameter sowie Variablendeklarationen. Der Logikanteil der Aktion wird entweder extern, d. h. in einer anderen Aktivität (über das Nichtterminal `ADIncludeStatement`, spezifiziert in den Zeilen 34-35) oder aber direkt in der Aktion definiert, was über das Nichtterminal `ADActionContent` geschieht. Ferner können Parameterverknüpfungen der Aktion zu der eingebundenen Aktivität enthalten sein (`ADAssignment` in Zeile 17). Der Aktionsinhalt (Zeilen 37-38) besteht aus einem Schlüsselwort, das angibt, welcher Art der Inhalt ist, sowie dem Inhalt selbst, welcher durch das externe Nichtterminal `ADContent` repräsentiert wird. Die Parameterverknüpfungen werden über die Schnittstelle `ADAssignment` (Zeile 17 bzw. definiert in Zeile 6) nochmals unterschieden in solche, die die Eingabeparameter der eingebundenen Aktivität belegen und solche, die deren Ausgabeparameter mit Ausgabeparametern der deklarierten Aktion verbinden (Zeilen 40-44). Eingabe- und Ausgabeparameter sowie Variablendeklarationen dienen der Übergabe und Verwendung von Objekten innerhalb einer Aktion und sind ähnlich aufgebaut (Zeilen 22-32).

Die Syntax der Rollen sind ebenso wie die Deklaration weiterer Kontrollknoten relativ unspektakulär (Zeilen 46-60). In den Kontrollknoten `ADMergeNode`, `ADDecisionNode`, `ADJoinNode` sowie `ADForkNode` (Zeilen 50-60) existiert die Möglichkeit, die Eingangs- und Ausgangsparameter miteinander zu verknüpfen, was über ein `ADBasicAssignment` (Zeilen 96-97) geschieht.

Aktivitätskanten sind spezifiziert in Zeilen 62-63. Diese bestehen aus einer linken Kantenseite, die den oder die Ausgangsknoten repräsentieren und einer rechten Kantenseite, die den oder die Zielknoten darstellen. Sowohl linke (Zeilen 65-66) als auch rechte Kantenseiten (Zeilen 68-70) bestehen aus einem oder mehreren Knoten, die entweder alleinstehend oder über Kontrollknoten miteinander verknüpft sind. Dazu existieren verschiedene Nichtterminale, die diese Kurzform ermöglichen (Zeilen 72-94). Jede dieser Seiten kann entweder einen Knoten mit einem Knoten oder einen Parameter eines Knoten mit einem Parameter eines Zielknoten verbinden.

Wie bereits beschrieben, werden die Inhalte einer Aktion über externe Nichtterminale abgebildet. Davon können mehrere in unterschiedlicher Reihenfolge kombiniert werden, wobei der Typ des `ADContent` über das Terminal `kind` (Zeile 38) festgelegt wird, welcher

Inhalt an der Stelle folgt. Für den Wert `java` wird die Grammatik verwendet, die in kompakter Form in Abbildung 8.2 dargestellt ist, für den Wert `view` erfolgt die Einbindung der Grammatik aus Abbildung 8.3 und schließlich führt der Wert `cmd` zur Ersetzung durch die Sprache, die in Abbildung 8.4 definiert ist.

MCSimp

```
1 grammar JavaContent extends mc.umlp.common.Common {
2
3   external BlockStatement;
4
5   Java =
6     blockStatement:BlockStatement;
7
8 }
```

Abbildung 8.2.: Vereinfachte Darstellung des `java` Aktionsinhalts

Bei der Einbindung von Java-Inhalt, der der Grammatik aus Abbildung 8.2 folgt, wird der eigentliche Inhalt über ein externes Nichtterminal realisiert. Das Nichtterminal `Java` wird durch das externe Nichtterminal `Blockstatement` ersetzt, welches seinerseits aus der Java-Sprache [FM07] stammt und eine beliebige Anzahl von Java-Statements enthalten kann, die in geschweiften Klammern eingeschlossen sind (die Java-Grammatik ist im Anhang in Abschnitt C.4 zu finden).

MCSimp

```
1 grammar ViewContent extends mc.umlp.common.Common {
2
3   View  =
4     (pageName:Name "(" params:Name? ("," params:Name)* ")" ";") ;
5
6 }
```

Abbildung 8.3.: Vereinfachte Darstellung des `view` Aktionsinhalts

Die Syntax für die Einbindung von Seiten durch das Verwenden des Schlüsselworts `view` wird über die Grammatik in Abbildung 8.3 realisiert. Hierbei wird direkt spezifiziert, welchen Aufbau die Referenz einer Seite inklusive potenzieller Parameterübergaben besitzt (Zeilen 3-4).

Die Verwendung von speziellen Kommandos wird über die Syntax vorgenommen, die in Abbildung 8.4 definiert ist. Hierbei stehen verschiedene Kommandos zur Verfügung, die alternativ verwendet werden können (Zeilen 3-5). Jedes dieser Kommandos ist nachfolgend über eine eigene Produktion definiert (Zeilen 8-34).

MCSimp

```
grammar CommandContent {

  Command =
    FindCommand | SaveCommand | LoadCommand | AssignRolesCommand |
    LoadAllCommand | GetActualUserCommand | SaveAllCommand |
    DeleteCommand | SetErrorsCommand ;

  FindCommand  =
    (variableName:Name "=")?  entityName:Name "." "find"
      "(" attributeName:Name "," value:Name ")" ";";

  SaveCommand  =
    entityName:Name "." "save" "(" attributeName:Name ")" ";";

  SaveAllCommand =
    entityName:Name "." "saveAll" "(" attributeName:Name ")" ";";

  LoadCommand  =
    (variableName:Name "=")?  entityName:Name "." "load" "(" id:Name ")" ";";

  LoadAllCommand =
    (variableName:Name "=")?  entityName:Name "." "loadAll" "(" ")" ";";

  AssignRolesCommand =
    "assignRole" "(" roleName:Name "," variableName:Name ")" ";";

  GetActualUserCommand =
    variableName:Name "=" "getActualUser" "(" ")" ";";

  DeleteCommand =
      entityName:Name "." "delete" "(" variableName:Name ")" ";";

  SetErrorsCommand =
      "setErrors" "(" variableName:Name ")" ";";

}
```

Abbildung 8.4.: Vereinfachte Darstellung des `cmd` Aktionsinhalts

8.2.2. Verwendung der Aktivitätsdiagrammsprache

Nachdem im vorherigen Abschnitt bereits der Aufbau der Grammatiken erläutert wurde, wird im folgenden Abschnitt die Verwendung der Sprache näher beschrieben. Eine Einführung in das Gesamtkonzept der Aktivitätsdiagramme, wie sie in MontiWIS verwendet werden, erfolgt anhand des Beispiels in Abbildung 8.5, das ebenfalls in [RR13] diskutiert wurde.

Das Beispiel zeigt die Aktivität `GradeThesis`. Darin ist der Ablauf abgebildet, der bei Bewertung einer Abschlussarbeit abläuft. An der dargestellten Aktivität sind zwei Nutzer

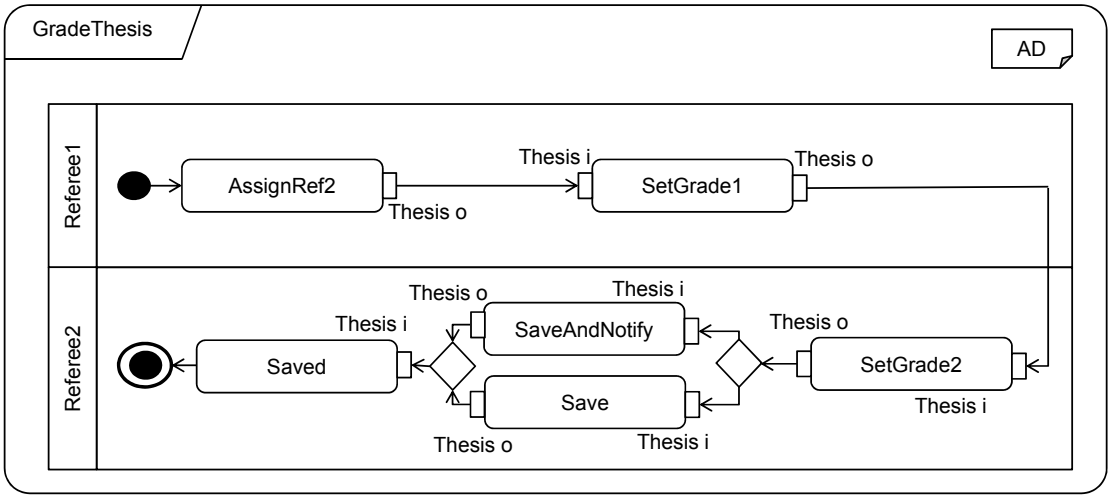


Abbildung 8.5.: Grafisches Beispiel einer Aktivität

beteiligt, was durch die beiden Aktivitätsbereiche (oder Partitionen) ausgedrückt wird. Hierbei handelt es sich um die beiden Gutachter (Erstgutachter (`Referee1`) und Zweitgutachter (`Referee2`)), die jeweils die Bewertung der Arbeit vornehmen. Der Ablauf beginnt mit der Aktion `AssignRef2`, in der durch den Erstgutachter festgelegt wird, welcher Nutzer die Rolle des Zweitgutachters innerhalb dieses Ablaufs übernimmt. Dies ist dargestellt durch die Aktion `AssignRef2`. In dieser Aktion wird ferner ein neues `Thesis`[25] Objekt erstellt, das nach Abschluss der Aktion zum nächsten Schritt `SetGrade1` weitergereicht wird. In dieser Aktion vergibt der Erstgutachter seine Note, welche über das `Thesis` Objekt zum nächsten Schritt `SetGrade2` übergeben wird. Nach dem Empfang der Daten in dieser Aktion ist für den Erstgutachter der Ablauf beendet, da sich die nächste Aktion im Aktivitätsbereich des Zweitgutachters befindet. In `SetGrade2` wird dem Zweitgutachter die Möglichkeit gegeben, seine Note einzutragen. Danach sind potenziell zwei unterschiedliche Aktionen möglich. Einerseits können die eingegebenen Daten lediglich gespeichert werden (über die Aktion `Save`), andererseits ist es möglich, die Daten zu speichern und gleichzeitig Benachrichtigungen darüber zu versenden (durch die Verwendung der Aktion `SaveAndNotify`). Da keine Wächterbedingungen an den ausgehenden Kanten der Aktion `SetGrade2` angegeben sind und somit auch keine automatische Evaluierung möglich ist, wird die Auswahl, welche der Aktionen als Nächstes ausgeführt werden soll, dem Zweitgutachter bei der Anzeige in `SetGrade2` überlassen. Innerhalb der Aktionen `Save` und `SaveAndNotify` werden keine Seiten präsentiert, sondern lediglich die beschriebenen Aktionen verarbeitet. Damit können diese Aktionen vollständig automatisch (d. h. ohne Nutzerinteraktion) ablaufen. Schließlich wird das `Thesis`-Objekt zur Aktion `Saved` weitergeleitet. In dieser Aktion wird lediglich eine Seite angezeigt, die über die erfolgreiche Speicherung informiert und das `Thesis`-Objekt erneut angezeigt.

Abbildung 8.6 stellt die Syntax des Beispiels aus Abbildung 8.5 dar. Hierbei werden lediglich

[25] Die in diesem Ablauf verarbeiteten Objekte beziehen sich auf die Klassen des Domänendatenmodells aus Abbildung 6.5 bzw. 6.6.

AD

```
activity GradeThesis {

  role Referee1 { AssignRef2, SetGrade1 ;}
  role Referee2 { SetGrade2, SaveAndNotify, Save, Saved; }

  action AssignRef2 {
    out : Thesis o;
    // weiterer Inhalt
  }

  action SetGrade1 {
    in : Thesis i;
    out: Thesis o;
    // weiterer Inhalt
   }

  action SetGrade2 {
    in : Thesis i;
    out : Thesis o;
    // weiterer Inhalt
  }

  action SaveAndNotify {
    in : Thesis i;
    out : Thesis o;
    // weiterer Inhalt
  }

  action Save {
    in : Thesis i;
    out : Thesis o;
    // weiterer Inhalt
  }

  action Saved {
    in : Thesis i;
    // weiterer Inhalt
  }

  initial -> AssignRef2;
  AssignRef2.o -> SetGrade1.i;
  SetGrade1.o -> SetGrade2.i;
  SetGrade2.o -> SaveAndNotify.i | Save.i;
  SaveAndNotify | Save -> Saved;
  Saved -> final;
}
```

Abbildung 8.6.: Textuelle Repräsentation des Beispiels aus Abbildung 8.5

die Teile dargestellt, die auch im grafischen Pendant zu sehen sind. Inhalte wie beispielsweise Seitenreferenzen oder Variablendeklarationen innerhalb der Aktionen wurden an dieser Stelle der Übersichtlichkeit halber weggelassen. Ferner wurde auf Paket- und Importdeklarationen verzichtet. Wie in Abschnitt 3.3.1 erläutert und in MontiCore üblich werden Modelle in einer Datei gespeichert, die dem Namen des Modells entspricht. Dies führt dazu, dass die Aktivität `GradeThesis` in einer Datei mit dem Namen `GradeThesis.ad` gespeichert werden muss, da `ad` das Suffix für Aktivitätsdiagramme in MontiWIS ist.

Die dargestellte Aktivität enthält Rollendeklarationen, die denen des grafischen Beispiels entsprechen (Zeilen 3-4). In den Aktionen (Zeilen 6-35) werden lediglich Ein- und Ausgabeparameter deklariert, die ebenfalls im Beispiel dargestellt sind. Schließlich werden die Kanten in den Zeilen 40-45 definiert.

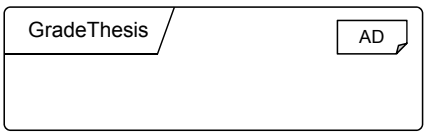
Detaillierter soll auf die Umsetzung dieses Beispiels an dieser Stelle nicht eingegangen werden. Darauf wird an entsprechender Stelle bei der Erläuterung der einzelnen Konstrukte zurückgekommen.

8.2.3. Beschreibung der einzelnen Sprachelemente

Nach der sehr oberflächlichen Beschreibung der Umsetzung des Beispiels wird im folgenden Abschnitt auf die einzelnen Elemente detailliert eingegangen. Hierbei wird zum Teil auf Konstrukte des Beispiels aus Abbildung 8.6 verwiesen. Zusätzlich werden jedoch die beschriebenen Elemente anhand expliziter Gegenüberstellung von grafischer Repräsentation auf der linken Seite und textueller Syntax auf der rechten Seite der jeweiligen Abbildung veranschaulicht. Sprachelemente, die keine grafische Repräsentation besitzen (wie beispielsweise der Aktionsinhalt), werden lediglich textuell angegeben.

Aktivitäten

Abbildung 8.7 zeigt die Umsetzung von Aktivitäten. Wie bereits im Beispiel in Abbildung 8.6 zu sehen ist, werden Aktivitäten über das Schlüsselwort `activity` und einen Namen definiert. Eine Aktivität wird stets in einer eigenen Datei gespeichert, die den gleichen Namen wie die Aktivität besitzt und mit dem Suffix `ad` erweitert wird.




```
1 activity GradeThesis {                    AD
2
3   // weiterer Inhalt
4
5 }
```

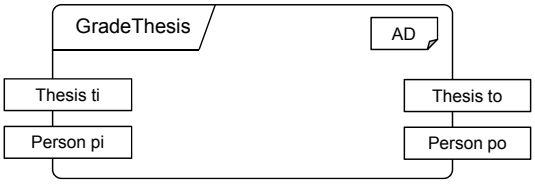
Abbildung 8.7.: Syntax einer Aktivität

Der eigentliche Inhalt einer Aktivität wird in geschweiften Klammern angegeben. Zunächst werden potenzielle *Eingangs- und Ausgangsparameter* der Aktivität angegeben. Diese sind optional und relevant, wenn die Aktivität das Verhalten einer anderen Aktion beschreiben

soll. Der Aktivitätsinhalt besteht aus den Elementen, die im Folgenden erläutert werden. Dies können *Aktionen*, *Aktivitätsbereiche*, *Verzweigungsknoten*, *Verbindungsknoten*, *Parallelisierungsknoten*, *Synchronisationsknoten*, *Startknoten*, *Endknoten* sowie verschiedene Arten von *Kurzschreibweisen* für Kanten sein. Diese Elemente können in beliebiger Anzahl und in beliebiger Reihenfolge vorkommen. Es ist jedoch der Übersichtlichkeit halber empfehlenswert, die Elemente in sinnvoller Weise zu gruppieren. Dies ist beispielsweise in der textuellen Darstellung des Beispiels in Abbildung 8.6 der Fall, wo zunächst Aktivititätsbereiche definiert werden, anschließend Aktionen und abschließend die Kanten zwischen den Knoten.

Parameter

Abbildung 8.8 stellt dar, wie Parameter von Aktivitäten spezifiziert werden. Parameter besitzen einen Typ und einen Namen und können entweder Eingangs- oder Ausgangsparameter sein. Ein Eingangsparameter nimmt ein Objekt seines Typs entgegen und leitet dieses innerhalb der Aktivität zur Bearbeitung an Aktionen weiter.




```
activity GradeThesis {                                    AD
  in : Thesis ti, Person pi;
  out : Thesis to, Person po;

  // weiterer Inhalt

}
```

Abbildung 8.8.: Syntax von Aktivitätsparametern

Im Beispiel in Abbildung 8.8 sind die Parameter `Thesis ti` und `Person pi` Eingangsparameter und `Thesis to` und `Person po` Ausgangsparameter der Aktivität. Dies wird in der grafischen Repräsentation lediglich durch die Lage der Parameter angedeutet, wobei die Eingänge auf der linken Seite der Aktivität liegen, die Ausgänge hingegen auf der rechten Seite. In der textuellen Repräsentation der Parameter wird diese Unterscheidung explizit über die Schlüsselwörter `in` und `out` modelliert. Nach diesen Schlüsselwörtern folgt ein Doppelpunkt, hinter dem die Parameter durch Kommata getrennt und mit einem Semikolon abgeschlossen werden. Zu beachten ist, dass die Parameter in diesem Beispiel konstruiert sind, da die dargestellte Aktivität zwar den gleichen Namen trägt wie im vorher eingeführten Beispiel in Abbildung 8.5 bzw. 8.6, dieses jedoch keine Eingangs- und Ausgangsparameter besitzt.

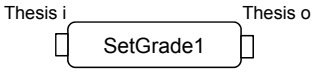
Generell müssen Parameter sowohl bei Aktivitäten als auch, wie nachfolgend beschrieben, in Aktionen, einen Typ und einen Namen besitzen. Der Typ kann einer derjenigen sein, die innerhalb von MontiWIS als vordefinierte Datentypen zur Verfügung stehen (siehe hierzu Abschnitt 6.2.3) oder aber selbst im Domänendatenmodell als Klasse definiert worden sein. Ferner sind mengenwertige Parameter in Form von `Collection<T>`s erlaubt, die auch wieder mit einer vordefinierten Klasse oder einem Bestandteil des Domänendatenmodells parame-

triert sein kann. Der Name eines Parameters sollte seine Funktion widerspiegeln und muss eindeutig innerhalb der Aktivität oder der Aktion sein.

Sollten Eingangsparameter einer Aktivität Teil der Kantendeklaration sein, so fungieren sie ebenfalls als Startpunkt für diese Aktivität, d. h. bei Start der Aktivität werden die Objekte, die diesem Parameter zugeordnet wurden, den Zielparametern der Kante übergeben. Die Verwendung von Parametern in Kantendefinitionen wird im Abschnitt über Aktivitätskanten detaillierter erläutert.

Aktionen

Eine Aktion ist in MontiWIS das komplexeste Konstrukt einer Aktivität. In ihr wird die Geschäftslogik ausgeführt, sie dient zur Anzeige von Seiten und ihr Status wird in der Regel gespeichert. Ein Teil der Elemente, die zu Aktionen gehören, ist in Abbildung 8.9 dargestellt.




```
1 action SetGrade1 {                                    AD
2   in : Thesis i;
3   out : Thesis o;
4
5   // weiterer Inhalt
6
7 }
```

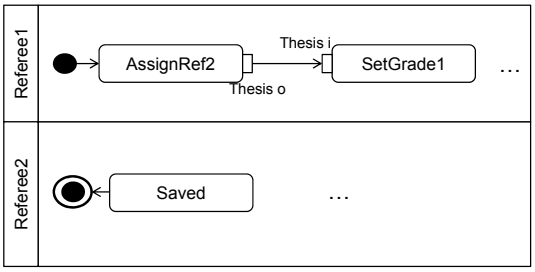
Abbildung 8.9.: Syntax von Aktionen

Hierbei handelt es sich um die Aktion `SetGrade1` aus dem Beispiel aus Abbildung 8.5 bzw. 8.6. Eine Aktion wird in der textuellen Syntax der Aktivitätsdiagramme eingeleitet über das Schlüsselwort `action` und besitzt einen Namen. Der Inhalt einer Aktion ist eingeschlossen in geschweiften Klammern. Die dargestellte Aktion besitzt Eingangs- (`Thesis i`) und Ausgangsparameter (`Thesis o`). Eingangsparameter werden hierbei hinter dem Schlüsselwort `in` und einem Doppelpunkt angegeben, bei Ausgangsparametern ist dies das Schlüsselwort `out`. Mehrere Parameter werden über Kommata getrennt und die Liste der Parameter wird mit einem Semikolon beendet. Nach den Parametern ist es weiterhin möglich, Variablen zu deklarieren, die innerhalb der Aktion Gültigkeit besitzen. Diese werden über das Schlüsselwort `var` identifiziert und gleichen ansonsten den eben beschriebenen Parametern. Hinsichtlich Typisierung und Namensgebung folgen sowohl die Parameter als auch die Variablendeklarationen den Richtlinien, die im Abschnitt mit Erläuterungen zu Parametern kürzlich beschrieben wurden.

Da Aktionen das zentrale Element von Aktivitätsdiagrammen sind und in MontiWIS zur Spezifikation von Geschäftslogik dienen, wird ihr Inhalt sehr ausführlich behandelt. Dies geschieht im nächsten Abschnitt 8.3.

Aktivitätsbereiche

Die Definition von Aktivitätsbereichen (welche im Allgemeinen auch als sog. *Swimlanes* bekannt sind) ist in Abbildung 8.10 dargestellt.



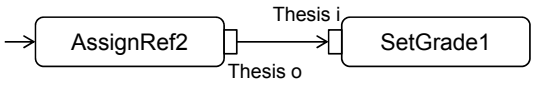

```
1 role Referee1 { AssignRef2, ↩        AD
      SetGrade1, // ...
2   ;}
3 role Referee2 { SetGrade2, // ...
4   ;}
```

Abbildung 8.10.: Syntax von Aktivitätsbereichen

Über Aktivitätsbereiche wird spezifiziert, welche Rolle, die an einer Aktivität beteiligt ist, eine bestimmte Aktion ausführt. Dementsprechend werden auch die Aktionen in der textuellen Syntax einem bestimmten Aktivitätsbereich zugeordnet. Dies geschieht über das Schlüsselwort `role` gefolgt vom Namen der Rolle (in diesem Fall `Referee1` bzw. `Referee2`). In geschweiften Klammern werden nachfolgend die Aktionen angegeben, die dem jeweiligen Aktivitätsbereich zugeordnet sind. Mehrere Aktionen werden durch Kommata getrennt und mit einem Semikolon abgeschlossen.

Aktivitätskanten

Aktivitätskanten stellen Verbindungen zwischen beliebigen Knoten in einer Aktivität dar. Hierbei können einerseits Knoten verknüpft werden (im Falle, dass es sich bei der Kante lediglich um Kontrollfluss handelt), oder aber direkt Ausgangs- mit Eingangsparametern eines Knotens. Abbildung 8.11 stellt letztere Variante dar. Wie im Beispiel gezeigt, werden Parameter von Aktionen über einen Punkt von dem Namen der Aktion getrennt. Derartige Parameterverknüpfungen werden in der Regel auf beiden Seiten der Kantendeklaration angegeben. Ausnahme bildet jedoch eine Verknüpfung, in der auf einer Seite der Kante ein Parameter einer Aktion angegeben ist, auf der anderen Seite aber der Parameter der einschließenden Aktivität. In diesem Fall wird lediglich der Name des Eingangs- bzw. Ausgangsparameters der Aktivität angegeben, da dies bereits einen Parameter adressiert.




```
1 AssignRef2.o -> SetGrade1.i;          AD
```

Abbildung 8.11.: Syntax von Kanten

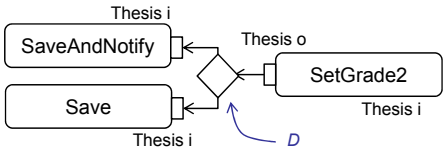
Im Fall des Beispiels wird der Ausgangsparameter (`Thesis o`) der Aktion `AssignRef2` mit dem Eingangsparameter (`Thesis i`) der Aktion `SetGrade1` verknüpft. Die gerichtete Kante

wird hierbei symbolisiert durch ein Pfeilsymbol (->), wobei links davon der Ausgangsparameter (bzw. bei Kontrollfluss entsprechend der Ausgangsknoten) angegeben wird, auf der rechten Seite des Pfeils der Eingangsparameter (bzw. Knoten), der das Ziel der Kante darstellt. Der Typ des Parameters wird bei der Definition einer Kante nicht angegeben, dieser ist lediglich innerhalb der Ausgangs- bzw. Zielaktion festgelegt. Typkompatibilität ist dennoch vonnöten, da im generierten System eine entsprechende Zuweisung anderenfalls nicht möglich ist.

Bei der Definition von Kanten ist es stets möglich, Parameter bzw. Knoten direkt 1:1 miteinander zu verbinden. Zusätzlich existieren jedoch auch Kurzformen, die die Verknüpfung von *n* Knoten über eine bestimmte Art von Kontrollknoten repräsentieren. Sowohl diese Kurzformen als auch Verknüpfung von Aktionen über die explizite Verwendung von Kontrollknoten werden in den folgenden Abschnitten beschrieben.

Verzweigungsknoten

Für die Spezifikation von Verzweigungsknoten (engl. *decision node*) existieren in MontiWIS verschiedene Möglichkeiten. In Abbildung 8.12 sind zunächst Verzweigungsknoten dargestellt, die keine Wächter besitzen, was in MontiWIS implizit die Entscheidung, welche Aktion als Nächstes ausgeführt werden soll, auf den Nutzer der Applikation überträgt.




```
1 SetGrade2.o -> SaveAndNotify.i | Save.i;   AD
2
3 decision D {
4   o = i;
5 }
6
7 SetGrade2.o -> D.i;
8 D.o -> SaveAndNotify.i;
9 D.o -> Save.i;
```

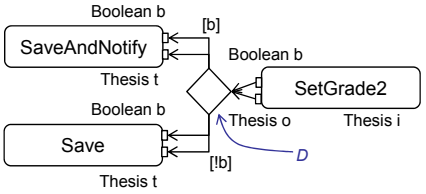
Abbildung 8.12.: Syntax von Verzweigungsknoten

Syntaktisch bieten sich hierfür zwei Möglichkeiten. Einerseits ist es möglich, durch die in Zeile 1 dargestellte Kurzform mehrere Knoten über einen Entscheidungsknoten zu verbinden. Auch in diesem Fall können sowohl Knoten als auch explizit Parameter miteinander verbunden werden. Wie im Beispiel dargestellt, werden die Parameter (in diesem Fall die Parameter `Thesis i` der Aktionen `SaveAndNotify` und `Save`) auf der rechten Seite einer Kante aufgelistet, jeweils getrennt durch das Pipe Symbol (`|`) und abgeschlossen mit einem Semikolon. In diesem Fall ist lediglich ein Ausgangs- mit einem Eingangsparameter zu verknüpfen. Sind hiervon mehrere vorhanden, so sind diese explizit in einer weiteren Kante analog zu spezifizieren.

Die zweite Möglichkeit, Verzweigungsknoten zu definieren, geschieht über die explizite Definition des Verzweigungsknotens, wie im Beispiel in den Zeilen 3-5 zu sehen ist. Ein Verzweigungsknoten wird über das Schlüsselwort `decision` definiert und besitzt einen Namen,

über den er im weiteren Verlauf in Kantendeklarationen referenziert wird. Sollten Verzweigungsknoten mit Datenfluss zusammen verwendet werden, wie dies im Beispiel der Fall ist, sind in dem Knoten ebenfalls die Verknüpfungen zu spezifizieren. Generell werden dabei stets Eingangsparameter mit Ausgangsparametern verknüpft, so dass in den Kantendeklarationen explizit auf diese Parameter verwiesen werden kann. In diesem Fall geben sie an, dass die Variable `o` (auf der linken Seite der Zuweisung) mit dem Wert der Variablen `i` (auf der rechten Seite der Zuweisung) belegt wird und somit der Eingangsparameter zum Ausgangsparameter durchgeschleust wird. Dies ist notwendig, um eventuellen Nichtdeterminismen vorzubeugen, die entstünden, wenn beispielsweise zwei Datenobjekte desselben Typs entlang der Kante transportiert würden. Explizit definierte Verzweigungsknoten und deren Parameter sind wie andere Knotentypen auch in Kanten zu verwenden. Dies ist in Zeilen 7-9 dargestellt. Hier wird zunächst eine Kante vom Ausgangsparameter `o` zum impliziten Eingangsparameter `i` des Verzweigungsknotens gezogen. Von dessen Ausgangsparameter `o` werden die Eingangsparameter `i` der Aktionen `SaveAndNotify` und `Save` verknüpft.

Bislang wurden lediglich Verzweigungsknoten betrachtet, bei denen dem ausführenden Nutzer der Aktivität die Entscheidung zukommt, welche Aktion als Nächstes auszuführen ist. Alternativ dazu besteht jedoch auch die Möglichkeit, automatisch auf Serverseite zu entscheiden, welche Kante des Verzweigungsknotens gewählt werden soll. Dies ist in Abbildung 8.13 dargestellt.




```
1  SetGrade2.b -> [b] SaveAndNotify.b | [!b] Save.i;
2  SetGrade2.t -> SaveAndNotify.t | Save.t;
3
4  decision D {
5    bo = bi;
6    to = ti;
7  }
8
9  SetGrade2.b -> D.bi;
10 SetGrade2.t -> D.ti;
11
12 D.bo -> [b] SaveAndNotify.b;
13 D.bo -> [!b] Save;
14 D.to -> SaveAndNotify;
15 D.to -> Save.t;
```

AD

Abbildung 8.13.: Syntax von Verzweigungsknoten mit Bedingungen

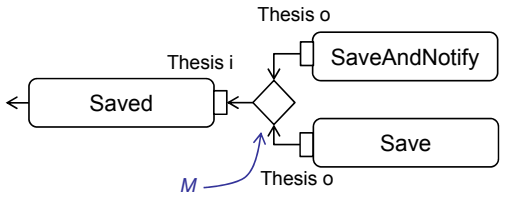
Das Beispiel in Abbildung 8.13 stellt den gleichen Sachverhalt wie das vorherige Beispiel dar. In der Aktion `SetGrade2` werden jedoch nicht explizit zwei Auswahlmöglichkeiten präsentiert, die dem ausführenden Nutzer die Wahl des nächsten Schrittes überlassen. Die Entscheidung wird stattdessen vom Wert eines Parameters abhängig gemacht, was im Fall des Beispiels der boolesche Parameter `b` ist. Neben diesem Parameter wird weiterhin das Objekt `Thesis t` von Aktion zu Aktion weitergegeben.

Auch für die Definition von Verzweigungsknoten existiert wiederum eine verkürzte Darstellung, die die explizite Definition des Verzweigungsknotens überflüssig macht. Dies ist in Zeilen 1-2 dargestellt. Hier werden ebenfalls die Ausgangs- mit den Eingangsparametern verknüpft, wobei das Pipe-Symbol (`|`) als abkürzende Schreibweise für Verzweigungsknoten zum Einsatz kommen. Die Bedingung, unter der eine bestimmte Kante ausgewählt wird, ist in eckigen Klammern zu spezifizieren. Da es sich bei dem Parameter `b` um ein `Boolean` handelt, kann in der Bedingung direkt nach seinem Wert geprüft werden. Alternativ wäre es auch möglich, beliebige Ausdrücke als Bedingung anzugeben, die einen Wahrheitswert zurück liefern (beispielsweise ein `equals`-Vergleich). Da in diesem Fall zwei Parameter entlang der Kante(n) weitergegeben werden, muss eine weitere Kantendefinition zur Verknüpfung des zweiten Parameters angegeben werden. Hierbei darf keine weitere Bedingung spezifiziert werden, da sonst potenziell Konflikte zwischen den unterschiedlichen Kanten entstünden.

Die längere Art der Darstellung ist in den Zeilen 4-15 abgebildet. Hierbei wird zunächst explizit der Verzweigungsknoten definiert und darin festgelegt, wie die eingehenden zu ausgehenden Parametern verknüpft werden. Weiterhin werden die jeweiligen Parameter explizit zum Verzweigungsknoten hin und davon weg verknüpft. Die Bedingungen werden an den Kanten angegeben, deren Parameter davon betroffen sind. Dies ist beispielsweise in den Zeilen 12-13 der Fall, wo die booleschen Parameter vom Verzweigungsknoten hin zu den nachfolgenden Aktionen geleitet werden. Der Parameter `Thesis` wird in einer separaten Kantendeklaration analog verknüpft.

Verbindungsknoten

Als das Gegenstück zu Verzweigungsknoten können sogenannte *Verbindungsknoten* (engl. *merge nodes*) gesehen werden. Während Verzweigungsknoten den Kontroll- oder Datenfluss innerhalb einer Aktion aufspalten und alternativ eine der möglichen Optionen ausgeführt werden, dienen Verbindungsknoten dazu, alternative Abläufe (oder auch beliebige andere) fortzusetzen. Um erfolgreich weitergeleitet zu werden, muss bei einem Verbindungsknoten einer der alternativen Kontrollflüsse ankommen. Ein Beispiel ist in Abbildung 8.14 dargestellt.




```
1  SaveAndNotify.o | Save.o -> Saved.i;
2
3  merge M {
4    o = i;
5  }
6
7  SaveAndNotify.o -> M.i;
8  Save.o -> M.i;
9  M.o -> Saved.i;
```

AD

Abbildung 8.14.: Syntax von Verbindungsknoten

Auch bei Verbindungsknoten existieren wiederum zwei Möglichkeiten, das gleiche Konstrukt in textueller Syntax abzubilden. Beide Möglichkeiten sind im Beispiel angegeben. Zunächst ist

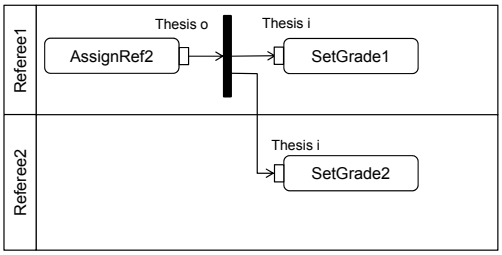
in Zeile 1 die Kurzschreibweise einer Kante mit Verbindungsknoten zu sehen. Hierbei werden auf der linken Seite die Knoten, von denen die Kanten ausgehen, die per Verbindungsknoten zusammengeführt werden, per Pipe Symbol (`|`) getrennt aufgelistet (in diesem Fall die Aktionen `SaveAndNotify` und `Save`). Auf der rechten Seite der Kantendefinition ist der Knoten zu sehen, der Ziel der (zusammengeführten) Kante ist.

Die zweite Möglichkeit der Verknüpfung von Knoten über einen Verbindungsknoten ist in den Zeilen 3-9 dargestellt. In Zeile 3-7 wird zunächst explizit der Verbindungsknoten über das Schlüsselwort `merge` und einen Namen definiert. Dieser Name wird nachfolgend in den Kantendefinitionen verwendet. Die Kanten werden separat von jeder alternativen Aktion zum Verbindungsknoten hin (Zeilen 7-8) und davon weg zur nachfolgenden Aktion (Zeile 9) definiert.

Wie im Beispiel zu sehen, besteht auch bei Verbindungsknoten die Möglichkeit, Parameter sowohl in der Kurzform als auch in der verbosen Form zu verknüpfen. Dies geschieht wie bisher in den Beispielen gezeigt. Auch bei Verbindungsknoten kann es notwendig sein, dass innerhalb des Knotens die Eingangs- und Ausgangsparameter explizit durch den Knoten hindurch geschleust werden. Analog zu Verzweigungsknoten werden die Parameter direkt bei der Knotendefinition in geschweiften Klammern definiert und verknüpft (Zeilen 3-5).

Parallelisierungsknoten

Sowohl Verzweigungs- als auch Verbindungsknoten stellen von ihrer Ausführungssemantik her Alternativen dar. Bei Verzweigungsknoten sollte lediglich eine Bedingung wahr sein, wohingegen zum Fortsetzen der Ausführung bei Verbindungsknoten das Vorliegen von Kontrollfluss auf einem Eingang des Knotens notwendig ist. Bei *Parallelisierungsknoten* (engl. *fork nodes*) hingegen werden sämtliche ausgehenden Kanten weiter verfolgt. Ein Beispiel dazu ist in Abbildung 8.15 dargestellt.




```
1 AssignRef2.o -> SetGrade1.i & SetGrade2.i; AD
2
3 fork F {
4   o = i;
5 }
6
7 AssignRef2.o -> F.i;
8 F.o -> SetGrade1.i;
9 F.o -> SetGrade2.i;
```

Abbildung 8.15.: Syntax von Parallelisierungsknoten

Hierbei handelt es sich nicht um einen Teil der bisherigen Beispielaktivität innerhalb dieses Abschnitts, sondern eine Umdeutung von Teilen davon. Anstatt dass das Festlegen der Note zunächst durch den Erst- und danach durch den Zweitgutachter erfolgt, geschieht dies parallel. Dazu wird zunächst der Zweitgutachter durch den Erstgutachter bestimmt (in der Aktion

`AssignRef2`). Aus dieser Aktion wird ein `Thesis` Objekt heraus zu einem Parallelisierungsknoten geleitet, worin es implizit dupliziert wird und je ein Objekt an die Aktion `SetGrade1` und eines an `SetGrade2` weitergereicht wird. Erstere wird vom Erstgutachter (`Referee1`), letztere vom Zweitgutachter (`Referee2`) ausgeführt. Da in der Aktion `AssignRef2` sowohl der Erst- als auch der Zweitgutachter festgelegt wurden, sind die nachfolgenden Aktionen einerseits von dem aktuellen Nutzer, andererseits von dem zugeordneten Zweitgutachter ausführbar. Nach Zuweisung wird zunächst direkt die Aktion `SetGrade1` aufgerufen, da diese ebenfalls im gleichen Aktivitätsbereich beheimatet ist. Parallel dazu kann direkt die Aktion `SetGrade2` bearbeitet werden, sofern der zugeordnete Nutzer aktuell im System aktiv ist.

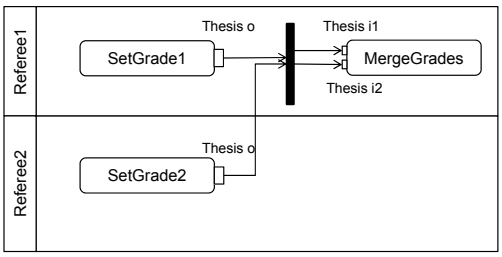
Eine besondere Behandlung erfährt hierbei der Datenfluss entlang der Kante. Der Ausgangsparameter `Thesis o` aus der Aktion `AssignRef2` wird durch den Parallelisierungsknoten dupliziert, so dass damit unterschiedliche Objekte zu den nachfolgenden Knoten weitergereicht werden. Dies erlaubt die unabhängige Bearbeitung der jeweiligen Objekte, impliziert jedoch auch, dass das Zusammenführen der Objekte unter Umständen relativ aufwendig sein kann und somit explizit behandelt werden muss. Auf diesen Umstand und wie damit umzugehen ist, wird im nächsten Abschnitt zu *Synchronisationsknoten* eingegangen.

Syntaktisch orientieren sich Parallelisierungsknoten an anderen Kanten- bzw. Knotendefinitionen. Wie in Zeile 1 in Abbildung 8.15 zu sehen, existiert auch für diese Art von Knoten eine Kurzform, die keine separate Knotendeklaration benötigt, sondern direkt aus der Definition der Kante abgeleitet wird. Auf der linken Seite der Kantendeklaration wird wie üblich der Knoten oder der Parameter notiert, von dem die Kante ausgeht (im Beispiel Parameter `o` der Aktion `AssignRef2`), auf der rechten Seite der Kante werden diejenigen Knoten oder Parameter angegeben, die Ziel der Kanten sind, die vom Parallelisierungsknoten ausgehen (im Beispiel die Parameter `i` der Aktionen `SetGrade1` und `SetGrade2`). Als Verbindungssymbol zwischen mindestens zwei Knoten kommt dabei ein Et-Zeichen (&) zum Einsatz, ein Semikolon schließt auch hier die Kantendeklaration ab. Sollen zwischen den gleichen Knoten mehrere Parameter verknüpft werden, so sind auch hierfür mehrere Kanten zu definieren, die jeweils Ausgangs- mit Eingangsparametern verknüpfen und dabei ebenfalls das Et-Zeichen als Trennzeichen verwenden.

Die zweite Möglichkeit, Parallelisierungsknoten zu verwenden, ist die explizite Definition des Knotens mit dem Schlüsselwort `fork` und dessen Verwendung in Kantendeklarationen (Zeilen 3-9). Hierbei werden analog zu den Beispielen zu Verzweigungsknoten in Abbildungen 8.12 oder 8.13 innerhalb des Knotens die Verknüpfungen von Eingangs- zu Ausgangsparametern spezifiziert und in den Kantendeklarationen (Zeilen 7-9) entsprechend verwendet.

Synchronisationsknoten

Nachdem der Ablauf einer Aktivität durch einen Parallelisierungsknoten in mehrere gleichzeitige Abläufe aufgeteilt wurde, müssen diese auch wieder zusammengeführt werden können. Um explizit die verschiedenen Abläufe wieder in einen weiterführenden Ablauf überführen zu können, werden sog. *Synchronisationsknoten* (engl. *join nodes*) verwendet. Diese bewirken, dass der Ablauf erst nach Ende der vorgelagerten Knoten weiter fortgeführt werden kann.



AD

```
1 join J {
2   o1 = i1;
3   o2 = i2;
4 }
5
6 SetGrade1.o -> J.i1;
7 SetGrade2.o -> J.i2;
8 J.o1 -> MergeGrades.i1;
9 J.o2 -> MergeGrades.i2;
```

Abbildung 8.16.: Syntax von Synchronisationsknoten

Bei dem in Abbildung 8.16 dargestellten Beispiel ist es nicht möglich, eine kompakte Notation zu verwenden, wie sie in den vorherigen Beispielen möglich war. Dies gilt jedoch nur für Fälle, wo Objekte entlang des Ablaufs weitergereicht werden. Für Ausnahmefälle, wo lediglich zwischen den Aktionen weiter geschaltet wird, ist eine Kurzform durchaus möglich. Im Falle des Beispiels würde diese durch `SetGrade1 & SetGrade2 -> MergeGrades` codiert werden. Damit ist die Syntax dieser Definition analog zur Kurzform von Verbindungsknoten, wobei als Verknüpfungssymbol das Et-Zeichen (&) dient.

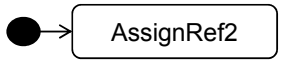
Auch wenn ein Synchronisationsknoten laut Spezifikation ([Obj11c], Abschnitt 12.3.34) lediglich eine ausgehende Kante besitzt, ist es in MontiWIS nicht möglich, nur eine Kante bzw. einen Parameter aus dem Synchronisationsknoten heraus zu leiten. Eine sinnvolle Kombination von Objekten, die auf den eingehenden Kanten eintreffen, ist nicht ohne Weiteres möglich. Dies lässt sich bereits am dargestellten Beispiel einleuchtend demonstrieren. Da die Objekte, die aus den beiden Aktionen `SetGrade1` und `SetGrade2` in den Synchronisationsknoten übergeben werden, ursprünglich durch Duplikation aus einem einzigen Objekt entstanden sind, kann ohne zusätzliche Information nicht erschlossen werden, wie beim Zusammenführen der beiden Objekte zu einem einzigen damit verfahren werden soll. Stattdessen werden, wie in der grafischen Darstellung des Beispiels bereits angedeutet, explizit zwei Parameter aus dem Synchronisationsknoten weitergegeben, welche auch wieder explizit mit je einem der Eingangsparameter der nachgelagerten Aktion `MergeGrades` verknüpft sind. Darin kann das eigentliche Verschmelzen der beiden Parameter aus den parallelen Aktionen geschehen, was in diesem Fall durch das Übernehmen der beiden Noten in eines der Objekte geschieht. Allgemein ist für die Verwendung von Synchronisationsknoten in MontiWIS festzuhalten, dass insbesondere beim Zusammenführen von vorher in der Aktivität aufgeteilten Abläufen und den weitergereichten Objekten in der Regel eine separate Aktion notwendig ist, in der die Logik dieses Verschmelzens explizit spezifiziert ist.

Wie in Abbildung 8.16 zu sehen, ist die Syntax von Synchronisationsknoten analog zu den übrigen Kontrollknoten definiert. Über das Schlüsselwort `join` und einen Namen wird ein Knoten definiert, der in geschweiften Klammern die Zuordnung von Eingangs- zu Ausgangsparametern enthält (Zeilen 1-4). Die Verknüpfung sowohl von den vorgelagerten Aktionen zu den Eingangsparametern des Synchronisationsknotens (Zeilen 6-7) sowie von seinen

Ausgangsparametern zu den Eingangsparametern der nachgelagerten Aktion (Zeilen 8-9) erfolgt analog zu gewöhnlichen Kantendefinitionen in MontiWIS.

Startknoten

Um anzuzeigen, welche Knoten zu Beginn einer Aktivität ausgeführt werden, werden *Startknoten* (engl. *initial nodes*) verwendet. Diese werden nicht explizit als Knoten modelliert, sondern lediglich per Schlüsselwort in einer oder mehreren Kantendeklarationen angegeben. Dies ist in Abbildung 8.17 dargestellt. Dies ist insofern einleuchtend, als bei Startknoten nicht die Unterscheidung in unterschiedliche Knoten, sondern lediglich die Spezifikation der zuerst auszuführenden Knoten von Bedeutung ist.




```
1 initial -> AssignRef2;                                    AD
```

Abbildung 8.17.: Syntax von Startknoten

Startknoten werden über das Schlüsselwort `initial` direkt innerhalb einer Kantendeklaration angegeben. Dieses hat selbstverständlich auf der linken Seite der Kantendeklaration zu stehen, da ein Startknoten nicht Ziel einer Kante sein kann.

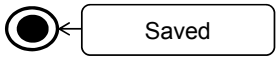
Eine Aktivität muss nicht zwangsläufig einen Startknoten besitzen, sofern sie stattdessen Eingangsparameter enthält, die implizit zu Beginn der Ausführung einer Aktivität mit passenden Objekten belegt werden. Diese werden bei Start der Aktivität entlang ihrer Kantendeklarationen weitergereicht. Sollten mehrere Startknoten deklariert worden sein, was entweder durch die Verwendung des Schlüsselworts `initial` zusätzlich zu Aktivitätsparametern oder durch die mehrfache Verwendung dieses Schlüsselworts geschehen kann, so ist das Startverhalten insofern nichtdeterministisch, als einer der Zielknoten der Kanten ausgeführt wird, ohne dass festgelegt ist, welcher dies ist. Daher wird empfohlen, hier möglichst kein nichtdeterministisches Verhalten zu spezifizieren bzw. sich über die Konsequenz daraus im Klaren zu sein.

Endknoten

Neben der Spezifikation, welche Knoten als erstes in einer Aktivität ausgeführt wird, lässt sich ebenfalls festlegen, wann eine Aktivität beendet ist. Auch hierfür werden keine expliziten Knoten deklariert, da bei *Endknoten* (engl. *final nodes*) lediglich das Verhalten, nicht aber eine Unterscheidung von verschiedenen Endknoten relevant ist.

Endknoten werden generell unterschieden in *Aktivitätsendknoten* (engl. *activity final node*) und *Kontrollflussendknoten* (engl. *flow final node*). Ein Aktivitätsendknoten ist in Abbildung 8.18 dargestellt.

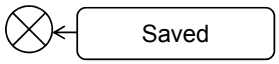
Die textuelle Spezifikation eines Aktivitätsendknotens geschieht über das Schlüsselwort `final`, welches ausschließlich auf der rechten Seite einer Kante vorkommen darf. Sobald der Ablauf einer Aktivität bei einem Aktivitätsendknoten anlangt, wird die Ausführung der




```
1 Saved -> final;
```
AD

Abbildung 8.18.: Syntax von Endknoten

Aktivität beendet, was ebenfalls potenziell weitere aktive Abläufe einschließt. Im Gegensatz dazu kann für die Beendigung eines Teils des Aktivitätsablaufs auch ein Kontrollflussendknoten verwendet werden, welcher beispielhaft in Abbildung 8.19 dargestellt ist.




```
1 Saved -> flowfinal;
```
AD

Abbildung 8.19.: Syntax von Ablaufendknoten

Kontrollflussendknoten sind ebenfalls lediglich über das Schlüsselwort `flowfinal` innerhalb einer Kantendeklaration zu spezifizieren. Wenn der Ablauf einer Aktivität einen solchen Knoten erreicht, wird lediglich der aktuelle Ablauf, nicht aber die komplette Aktivität beendet. Potenziell existierende weitere Abläufe werden weiterhin ausgeführt, so dass beispielsweise andere Nutzer den Ablauf fortführen können.

8.3. Inhalt von Aktionen

Die Geschäftslogik von Aktivitäten wird primär in Aktionen spezifiziert. Aus diesem Grund wird im folgenden Abschnitt detailliert auf die verschiedenen Möglichkeiten eingegangen, wie die Logik in MontiWIS ausgedrückt werden kann. In Aktionen lassen sich unterschiedliche Arten von Instruktionen spezifizieren. Diese können in der Regel in beliebiger Anzahl und beliebiger Reihenfolge darin vorkommen. Sollten diesbezüglich Restriktionen existieren, so wird im entsprechenden Abschnitt darauf hingewiesen.

Neben der eigentlichen Festlegung der Logik innerhalb einer Aktion besitzt sie in der Regel Eingangs- und Ausgangsparameter, über die Objekte von Aktion zu Aktion transportiert werden und die durch die Geschäftslogik in der Aktion erzeugt und modifiziert werden. Neben den Parametern, auf die über Kantendeklarationen zugegriffen werden kann, besteht weiterhin die Möglichkeit, Variablen zu deklarieren, auf die ausschließlich innerhalb der Aktion, aber dort global zugegriffen werden kann. Parameter werden in Abschnitt 8.3.1 beschrieben, Variablen in Abschnitt 8.3.2 erläutert. Hierbei ist zu beachten, dass bei der Deklaration von Parametern und Variablen zunächst Eingangs- und Ausgangsparameter und danach Variablen angegeben werden müssen, was der Übersichtlichkeit dienen soll. Diese sind zu Beginn der Aktion in genau dieser Reihenfolge zu deklarieren, sofern sie vorhanden sind. Allerdings sind sowohl Parameter als auch Variablendeklarationen optional und können so ggf. auch weggelassen werden.

Ferner ist es natürlich im Kontext eines Web-Informationssystems obligatorisch, dass Seiten

angezeigt werden können. Die Syntax für das Einbinden von Seitenreferenzen in einer Aktion wird in Abschnitt 8.3.3 erläutert.

Neben dem Aufruf von Seiten in einer Aktion sollte weiterhin möglich sein, sehr generische Geschäftslogik auszuführen und dabei insbesondere auch externen Code nutzen zu können. Aus diesem Grund erlaubt MontiWIS das Einbinden von Java-Code, der direkt beim Erreichen einer Aktion abläuft. Die Syntax und die Besonderheiten, die für das Einbinden von Java-Code beachtet werden müssen, werden in Abschnitt 8.3.4 beschrieben.

Obwohl Java-Aufrufe an sich einen Großteil benötigter Funktionalität abdecken, existieren dennoch Bereiche, wo auf Funktionalität zugegriffen werden muss, die zum Zeitpunkt der Modellierung noch nicht vorhanden ist bzw. die Bereiche berührt, auf die nicht einfach per gewöhnlichem Quellcodeaufruf zugegriffen werden kann. Aus diesem Grund existieren in MontiWIS domänenspezifische *Kommandos*, die eben solche Zugriffe erlauben. Welche Kommandos existieren und wie sie verwendet werden, wird in Abschnitt 8.3.5 erläutert.

Letztlich kann die Geschäftslogik einer Aktion auch in einer separaten Aktivität beschrieben sein. Hierbei wird eine andere Aktivität gestartet, die auf diese Weise auch in mehreren Kontexten aufgerufen werden kann. Sie muss dementsprechend nicht zwangsläufig durch einen Nutzer explizit gestartet werden, sondern wird für diesen transparent ausgeführt. Die Syntax für den Aufruf anderer Aktivitäten und deren Anwendung wird in Abschnitt 8.3.6 beschrieben.

8.3.1. Parameter

Parameter in Aktionen werden, analog zu Parametern von Aktivitäten, innerhalb ihrer Elemente beschrieben. Ein Beispiel für Aktionsparameter ist in Abbildung 8.20 gegeben.

AD

```
1 action SetGrade1 {
2   in : Thesis i;
3   out : Thesis o;
4   // weiterer Inhalt
5 }
```

Abbildung 8.20.: Parameter von Aktionen

Auch hier werden Eingangsparameter durch das Schlüsselwort `in` gefolgt von einem Doppelpunkt und der Deklaration der Parameter spezifiziert. Ein Parameter besitzt einen Typ (was entweder ein vordefinierter Datentyp (vgl. Abschnitt 6.2.3) oder ein Klassenname aus dem Domänendatenmodell (hier: `Thesis`) sein kann) sowie einem Namen, unter dem der Parameter innerhalb der Aktion verwendet wird (hier: `i` bzw. `o`). Mehrere Parameter werden durch Kommata getrennt und mit einem Semikolon abgeschlossen. Dieselbe Syntax gilt für Ausgangsparameter, wobei hierbei jedoch das Schlüsselwort `out` zum Einsatz kommt.

Eingangs- und Ausgangsparameter werden innerhalb einer Aktion deklariert und besitzen innerhalb dieser Einheit Gültigkeit. Sie sind außerhalb der Aktion insofern sichtbar, als sie in

Kantendeklarationen verwendet werden, um Objekte von einer Aktion zur nächsten entlang der Kanten einer Aktivität weiterzureichen.

8.3.2. Variablen

Neben Parametern, die zum Transport von Objekten einerseits entlang der Kanten einer Aktivität und andererseits in eine Aktion hinein und daraus hinaus dienen, existiert die Anforderung, innerhalb von Aktionen globale *Variablen* nutzen zu können, auf die innerhalb der Aktion aus jeder der folgenden Logikdefinitionen zugegriffen werden kann. Diese folgen syntaktisch den Parameterdeklarationen, die soeben beschrieben wurden, allerdings dient hier als Schlüsselwort `var`. Abbildung 8.21 zeigt dies beispielhaft. Wie bereits eingangs in Abschnitt 8.3 beschrieben, müssen Variablendeklarationen an dritter Stelle einer Aktion vorkommen, sofern sowohl Eingangs- als auch Ausgangsparameter in einer Aktion enthalten sind.

AD

```
1 action SetGrade1 {
2   // Parameterdeklarationen
3   var : Thesis other;
4 }
```

Abbildung 8.21.: Variablen in Aktionen

Wie bereits erwähnt sind Variablen innerhalb der Aktion sicht- und zugreifbar, außerhalb der Aktion sind sie nicht erreichbar. Abgesehen davon gelten die gleichen Regeln hinsichtlich möglicher Datentypen und -namen und der Auflistung mehrerer Variablen in der gleichen Aktion wie dies bei den Parametern sowohl in Aktionen als auch Aktivitäten der Fall ist.

8.3.3. Aufruf einer Seite

Eine zentrale Funktionalität in einem Web-Informationssystem, wie es in MontiWIS modelliert wird, ist selbstverständlich die Präsentation von Seiten und die Interaktion mit dem Nutzer darüber, entweder durch das Präsentieren von Informationen, das Abfragen von ebendiesen oder einer Kombination aus beidem. Seitenaufrufe können, müssen aber nicht in einer Aktion vorkommen. Abbildung 8.22 zeigt ein Beispiel, wie Seitenaufrufe in Aktionen eingebunden und verwendet werden.

Um eine Seite als Teil einer Aktion anzuzeigen, wird ihr Name (hier: `SetGrade1`) hinter dem Schlüsselwort `view`, gefolgt von einem Doppelpunkt, angegeben (Zeile 4). Der Name ist an dieser Stelle zufällig identisch mit dem Namen der Aktion, was nicht der Fall sein muss. In Klammern nach dem Namen der Seite werden die Parameter spezifiziert, die der Seite übergeben werden sollen. Mehrere Parameter werden innerhalb der Klammern per Komma getrennt. Die Parameter müssen hinsichtlich Anzahl und Typ mit denen übereinstimmen, die in der Seitendefinition spezifiziert wurden, was in Abschnitt 7.2.2 erläutert wurde. Wie

AD

```
1 action SetGrade1 {
2   in: Thesis i;
3
4   view : SetGrade1(i);
5   // weiterer Inhalt
6  }
```

Abbildung 8.22.: Seitenaufruf in Aktionen

bereits erwähnt, wird der Inhalt einer Aktion in der Reihenfolge seiner Definition ausgeführt. Dementsprechend gliedert sich eine Aktion in die Instruktionen, die *vor der Anzeige* der Seite ausgeführt werden, und diejenigen, die erst *nach Absenden* der Seite zum Tragen kommen. Dies bedingt, dass die Parameter der Seite innerhalb der Aktion vor Aufruf der Seite initialisiert worden sein müssen. Sie können entweder als Parameter (wie im Beispiel der Eingangsparameter `i`) oder aber als Variablen definiert werden. Vor Übergabe an eine Seite müssen sie in der Aktion bereits erzeugt bzw. mit entsprechenden Werten belegt worden sein. Die Parameter werden bei Anzeige der Seite entsprechend ihres Typs und der Spezifikation der Seite dargestellt.

Nach Absenden der Seite besitzen die Parameter die Werte, die vom Nutzer eingegeben, gesetzt und abgesandt wurden. Der Zugriff darauf erfolgt weiterhin über die Objekte, die der Seite übergeben wurden. Die Parameter werden damit sowohl als Eingabeparameter als auch Rückgabewerte verwendet. Dies ist insofern sinnvoll, als in MontiWIS auf einer Seite keine Objekte erzeugt, sondern lediglich mit Werten belegt werden. Ein neues Objekt wird somit zunächst in einem vorgelagerten Schritt (in der Regel in einem Java-Abschnitt innerhalb der Aktion) instanziiert und danach der Seite übergeben, um es mit entsprechenden Werten zu belegen. Über derartiges Vorgehen ist auch die Manipulation von mehreren Objekten auf der Seite möglich, was beispielsweise bei Verwendung eines `return`-Konstrukts nicht ohne Weiteres möglich wäre.

8.3.4. Java

In MontiWIS wurde das Ziel verfolgt, möglichst flexibel hinsichtlich der Einbindung von beliebiger Geschäftslogik zu sein. Hierzu ist es möglich, beliebigen *Java-Code* in Aktionen zu verwenden. Ein Beispiel für die Verwendung ist in Abbildung 8.23 zu finden.

Für die Einbindung von Java-Quellcode wird das Schlüsselwort `java` gefolgt von einem Doppelpunkt verwendet. Der Java-Inhalt selbst wird innerhalb von geschweiften Klammern angegeben (Zeilen 6-8). Der Inhalt zwischen den beiden Klammern wird behandelt wie ein Blockstatement (wodurch es auch in der Java Grammatik [FM07] umgesetzt ist). Dies impliziert, dass beliebige Java Ausdrücke innerhalb dieser Klammern stehen dürfen, was neben einer Zuweisung wie im Beispiel etwa auch Instanziierungen, Methodenaufrufe oder Ähnliches sein kann. Da die Verwendung von Quellcodeanteilen direkt in Aktionen im generierten

AD

```
1 action SetGrade1 {
2     in : Thesis i;
3     out: Thesis o;
4
5     view : SetGrade1(i);
6     java : {
7         o = i;
8     }
9
10 }
```

Abbildung 8.23.: Einbindung von Java in Aktionen

Code potenziell problematisch sein kann und zu Inkompatibilitäten führen kann, werden im Folgenden einige Hinweise gegeben, die bei der Einbindung zu beachten sind. Dies ist insbesondere relevant, da der Java-Inhalt in der Form, wie er in der Aktion spezifiziert ist, untransformiert in die generierte Webapplikation übernommen wird.

Dieses Vorgehen bricht zwar mit dem generellen Anspruch, dass im Modell (und auf dieser Ebene bewegt sich der Java-Code in den Aktionen an dieser Stelle) kein Wissen über generierten Code existiert, aber pragmatisch betrachtet würde eine solche Vermischung ohnehin nicht vollständig zu vermeiden sein. Vordefinierte Datentypen wie beispielsweise `Email` (s. Abschnitt 6.2.3 und dort insbesondere Abbildung 6.7) werden im generierten Code des Web-Informationssystems auf gewöhnliche Java-Klassen abgebildet, was spätestens beim Aufruf externen Codes zu einer impliziten Konvertierung führen würde.

Unqualifizierter Zugriff auf Domänenobjekte

Um auf Klassen aus dem Domänendatenmodell (vgl. Kapitel 6) zuzugreifen, beispielsweise um ein neues Objekt einer Klasse zu erzeugen, genügen auch innerhalb von Java-Abschnitten einer Aktion die unqualifizierten Namen aus dem Domänendatenmodell. Entsprechende Qualifizierungen bzw. Importe werden implizit in dem Bereich vorgenommen. Um beispielsweise ein neues `Thesis` Objekt zu erzeugen, genügt etwa der Aufruf des `Thesis()`-Konstruktors innerhalb des Java-Blocks.

Attributzugriff durch get- und set-Methoden

Neben dem Zugriff auf Klassen aus dem Domänendatenmodell an sich ist es auch essenziell, deren Attribute abfragen bzw. setzen zu können. Dies geschieht, wie in Java üblich, über die Verwendung von `get`- und `set`-Methoden sowie im Fall von mengenwertigen Assoziationen über `add`-Methoden zum Hinzufügen neuer Objekte. Die entspricht den Methoden, die im generierten System verfügbar sind. Besonderheiten der Typisierung der Rückgabetypen von `get`-Methoden und der Parameter von `set`-Methoden werden nachfolgend erläutert.

Typkompatibilität mit den Zieldatentypen

Bei der Einbindung von Java-Code in Aktionen in MontiWIS werden sowohl Objekte des Domänenmodells und deren Attribute als auch potenziell extern definierter Code vermischt. Domänenobjekte werden, wie oben beschrieben, auf entsprechende Klassen im generierten Code abgebildet. Anders verhält es sich mit den Attributen der Domänenobjekte. Hier wird beispielsweise ein vordefiniertes `Email`-Attribut im generierten Java-Code nicht auf eine Klasse `Email`, sondern auf einen `java.lang.String` abgebildet. Zusätzlich werden an dezidierten Stellen entsprechende syntaktische Überprüfungen vorgenommen. Die Abbildung von vordefinierten Datentypen auf bestimmte Java Klassen ist in Tabelle 6.7 zusammengefasst.

Diese Abbildung resultiert auch darin, dass die getter- und setter-Methoden von Attributen mit vordefiniertem Datentyp die Klassen des Zielsystems zurückliefern, also beispielsweise ein `java.lang.String` für eine `get`-Methode eines `Email`-Attributs. Folglich ist auch ein `String` als Parameter für die zugehörige `set`-Methode und im weiteren Verlauf des eingebundenen Java-Codes ein `String` zu verwenden.

Verwendung von vollqualifizierten Namen

Da es relativ unpraktikabel wäre, innerhalb von Aktivitäten sämtliche importierten Paketnamen gesammelt anzugeben oder diese pro Aktion zu spezifizieren, sind Klassennamen innerhalb der Java-Abschnitte in vollqualifizierter Art anzugeben. Dies gilt für alle Klassen, die nicht im Paket `java.util` zu finden sind. Dessen Klassen sind implizit innerhalb des Java Abschnittes verfügbar, also vor allem auch die Standard `Collection`-Klassen, die häufig genutzt werden. Alle anderen Klassen, insbesondere die selbst geschriebenen, sind vollqualifiziert anzugeben.

8.3.5. Kommandos

Die Möglichkeit, beliebigen Java-Code nach den soeben beschriebenen Regeln einzubinden bietet ein Höchstmaß an Flexibilität bei der Spezifikation von Geschäftslogik in MontiWIS. Es existieren jedoch Anwendungsfälle, die sich nicht ohne Weiteres mit den damit zur Verfügung stehenden Mitteln umsetzen lassen.

Dies ist zum einen Funktionalität, die zum Zeitpunkt der Modellierung noch nicht zur Verfügung steht und erst aus den Modellen generiert wird. So wird zwar DAO-Funktionalität zum Laden, Speichern und Suchen von Objekten des Domänendatenmodells erzeugt, jedoch ist die Verwendung dieser Funktionalität nur über zielplattformspezifische Operationen verfügbar. Da auf Modellebene insbesondere davon abstrahiert werden soll, wie bestimmte Funktionen letztendlich umgesetzt werden, existieren sogenannte *Kommandos* (engl. *commands*), die es im Modell sehr einfach machen, auf die benötigten Operationen zuzugreifen, ohne explizit nicht existente bzw. komplizierte, zielplattformspezifische Funktionen aufzurufen.

Zum anderen wurde Funktionalität identifiziert, die ebenfalls stark von der letztlichen Zielplattform abhängt. Dies ist beispielsweise die Möglichkeit, auf den aktuell agierenden Nutzer und damit seine Integration in das Gesamtsystem, etwa durch Verknüpfung mit anderen Domänenobjekten, zuzugreifen. Damit zusammenhängend ist auch die Zuweisung

eines Nutzers zu bestimmten Aktivitätsbereichen (also Rollen innerhalb eines verteilten Workflows) zu sehen, was ebenfalls relativ aufwendig und spezifisch für die verwendete Technologie im generierten Web-Informationssystem ist.

Um hier eine Entkopplung zwischen der Modellebene und der Zielplattform (also sowohl der zielplattformspezifischen Infrastruktur wie Serverfunktionalität als auch den vom Generator erzeugten Klassen) zu erreichen, sind die folgenden Kommandos definiert worden.

Generell werden Kommandos eingeleitet durch das Schlüsselwort `cmd`, gefolgt von einem Doppelpunkt und schließlich dem eigentlichen Kommando. Dies ist beispielsweise in Abbildung 8.24 in Zeile 4 dargestellt. Vor und nach dem Kommando können quasi beliebige weitere Aktionsinhalte spezifiziert sein, also beispielsweise weitere Kommandos, Java-Inhalte oder ein Aufruf einer Seite. Wie bereits erläutert, werden auch diese Inhalte in der Reihenfolge ihres Auftretens ausgeführt.

Laden aller Objekte eines Typs

Bei der Spezifikation von Geschäftslogik ist es häufig notwendig, Zugriff auf sämtliche Objekte eines bestimmten Typs zu haben, um nachfolgend entweder weitere Filterungen vorzunehmen oder anderweitig Operationen darauf auszuführen. Dies lässt sich mit dem Kommando `loadAll` realisieren, wie es in Zeile 4 von Abbildung 8.24 zu sehen ist.

AD

```
1 action AssignRef2 {
2   var : Collection<Lecturer> allStaff;
3
4   cmd : allStaff = Lecturer.loadAll();
5   // weiterer Inhalt
6 }
```

Abbildung 8.24.: Kommando zum Laden von allen Objekten eines Typs

Dieses Kommando wird im Kontext einer bestimmten Klasse verwendet, welche im Rahmen des Domänendatenmodells definiert wurde (in diesem Fall `Lecturer`). Wie im Beispiel zu sehen ist, handelt es sich bei dem Ergebnis der Operation um eine `Collection`. Hierbei wurde darauf verzichtet, eine spezifischere `Collection`-Art (wie beispielsweise eine `List` oder ein `Set`) zu verwenden, um noch keine Festlegung zu treffen, wie die letztliche Implementierung erfolgen wird. Die Variable, der das Ergebnis zugewiesen wird, muss entweder als Eingangs- oder Ausgangsparameter oder aber, wie im Beispiel ersichtlich, als Variable deklariert worden sein, so dass im weiteren Verlauf der Aktion darauf zugegriffen werden kann. Im Falle der Definition als Ausgangsparameter und seiner Belegung per Kommando wird das Ergebnis direkt zur nächsten Aktion weitergegeben, was aber in der Regel eher selten der Fall sein wird.

Laden eines Objekts anhand seines Bezeichners

Wenn statt sämtlichen Objekten explizit eines geladen werden soll, so ist dies über das Kommando `load` möglich. Die Verwendung wird beispielhaft in Abbildung 8.25 dargestellt.

AD

```
1 action LoadLecturerWithId {
2     in : String id;
3     var: Lecturer lecturer;
4
5     cmd : lecturer = Lecturer.load(id);
6     // weiterer Inhalt
7 }
```

Abbildung 8.25.: Kommando zum Laden eines Objekts per Id

Das Kommando zum Laden eines einzelnen Objekts wird ebenfalls eingeleitet üben den Namen der Klasse (hier: `Thesis`), gefolgt von einem Punkt und dem Schlüsselwort `load`. Darauf folgend wird eine Variable in Klammern übergeben (hier: `id`), die mit dem eindeutigen Bezeichner des Objekts belegt sein muss. Diese `id` ist in diesem Fall per Eingangsparameter der Aktion übergeben worden. Alternativ wäre es beispielsweise auch möglich, eine Variable zu deklarieren und sie in einem vorgelagerten Java-Abschnitt mit einem festen Wert zu belegen.

Das Ergebnis des Kommandos ist im weiteren Verlauf der Aktion verfügbar über die Variable `lecturer`, die als Variable der Aktion deklariert wurde. Da zum Laden hier der eindeutige Bezeichner des Objekts verwendet wird, handelt es sich bei der Variable um ein einzelnes Objekt. Im Falle, dass das Objekt nicht gefunden wurde, wird die Variable `null` sein, weswegen in der Regel eine Überprüfung im weiteren Verlauf der Aktion empfohlen wird.

Die Anwendung dieses Kommandos wird in MontiWIS vermutlich eher selten auftreten, da beispielsweise bei der Auswahl eines bestimmten Objekts aus einer Liste von mehreren (oder auch allen) Objekten eines Typs dieses bereits geladen wird und in der übergebenen Variable der Seite vorhanden ist. Die Logik hierfür ist bereits im generierten Code enthalten und muss nicht separat durch explizites Laden implementiert werden. Dennoch kann das `load`-Kommando sinnvoll sein, beispielsweise wenn es um das Laden eines bestimmten Objektes geht, dessen `id` der Nutzer einer Seite eingegeben hat. Aus diesem Grund wird es in MontiWIS angeboten.

Suchen und Laden von Objekten

Neben dem Laden von allen Objekten eines Typs sowie eines explizit identifizierten ist es wünschenswert, eine Menge von Objekten anhand eines bestimmten Attributwertes laden zu können. Ein Beispiel für solch ein Kommando ist in Abbildung 8.26 zu finden.

Zum Suchen von Objekten wird zunächst die Klasse festgelegt, deren Objekte gesucht und geladen werden sollen. Im Falle des Beispiels ist dies die Klasse `Lecturer` (Zeile 5), gefolgt

AD

```
1 action FindLecturerWithTitle {
2     in : String titleName;
3     var: Collection<Lecturer> lecturers;
4
5     cmd : lecturers = Lecturer.find(title, titleName);
6     // weiterer Inhalt
7 }
```

Abbildung 8.26.: Kommando zum Suchen und Laden von Objekten

von einem Punkt und dem Schlüsselwort `find`. Nachfolgend in Klammern wird zunächst das Attribut der vorher genannten Klasse angegeben, nach dem gesucht werden soll (im Beispiel: `title`) gefolgt von der Variable, die den Wert enthält, mit dem das Attribut übereinstimmen muss. Im Beispiel ist dies der Eingangsparameter `titleName`, der mit einem entsprechenden `String`-Wert belegt sein muss. Alternativ könnte hier auch ein `String`-Attribut eines Objekts aus dem Datenmodell über seine getter-Methode übergeben werden oder aber eine Variable, deren Wert in einem vorgelagerten Java-Abschnitt fest belegt wurde.

Das Ergebnis eines `find`-Kommandos ist, wie im Beispiel in Zeile 3 zu sehen, ebenfalls wieder mengenwertig und vom Typ `Collection`, da potenziell mehrere Objekte bei der Suche gefunden und geladen werden können. Sollte die Suche kein Ergebnis liefern, so wird `null` zurückgegeben.

Speichern eines Objekts

Bei der Ausführung von Aktivitätsdiagrammen in MontiWIS werden Objekte erst bei explizitem Aufruf eines Kommandos in der Datenbank gespeichert bzw. aktualisiert. Dies geschieht über das `save`-Kommando, dessen Verwendung in Beispiel 8.27 dargestellt ist.

AD

```
1 action Save {
2   in : Thesis i;
3   cmd : Thesis.save(i);
4 }
```

Abbildung 8.27.: Kommando zum Speichern eines Objekts

Auch bei diesem Kommando wird zunächst die Klasse angegeben, dessen Objekt gespeichert werden soll (im Beispiel `Thesis` (Zeile 3)), gefolgt von einem Punkt und dem Schlüsselwort `save`. In Klammern wird die Variable übergeben, deren Inhalt in der Datenbank gespeichert oder aktualisiert werden soll. Im Beispiel ist dies der Eingangsparameter `i`. Beim Aufruf des Kommandos spielt es dabei inbesondere keine Rolle, ob das Objekt zu diesem Zeitpunkt

bereits in der Datenbank vorhanden ist oder ob lediglich geänderte Attributwerte gespeichert werden sollen.

Speichern einer Menge von Objekten

Da es in Aktionen möglich ist, eine nach einer Menge von Objekten zu suchen und diese zu Laden, sollte es auch möglich sein, eine Menge von Objekten gleichzeitig zu speichern. Dies ist über das Kommando `saveAll` möglich. Ein Beispiel hierzu ist in Abbildung 8.28 dargestellt.

AD

```
1 action SaveAllTheses {
2   in : Collection<Thesis> theses;
3
4   // weiterer Inhalt
5   cmd : Thesis.saveAll(theses);
6 }
```

Abbildung 8.28.: Kommando zum Speichern einer Menge von Objekten

Wie bei einer überwiegenden Anzahl der Kommandos wird auch dieses mit dem Namen der Klasse begonnen, dessen Objekte gespeichert werden sollen. Auch in diesem Fall handelt es sich hierbei um die Klasse `Thesis` (Zeile 5). Nach dem Schlüsselwort `saveAll` wird die mengenwertige Variable übergeben, deren Inhalt gespeichert werden soll (in diesem Fall die `Collection theses` in Zeile 2). Jedes Objekt dieser Sammlung wird separat gespeichert bzw. in der Datenbank aktualisiert.

Löschen eines Objekts

Außer dem Laden und Speichern von Objekten in einem Web-Informationssystem ist weiterhin das *Löschen* von Objekten von essenzieller Bedeutung. Hierzu wurde ein Kommando definiert, dessen Verwendung in Abbildung 8.29 zu sehen ist.

AD

```
1 action DeleteThesis {
2   in : Thesis thesisToDelete;
3
4   cmd : Thesis.delete(thesisToDelete);
5   // weiterer Inhalt
6 }
```

Abbildung 8.29.: Kommando zum Löschen eines Objekts

Beim Kommando zum Löschen eines Objekts wird ebenfalls zunächst der Name der zu löschenden Klasse angegeben. Darauffolgend wird, von einem Punkt getrennt, das Schlüsselwort `delete` verwendet, um in Klammern danach die Variable bzw. den Parameter zu spezifizieren, dessen Inhalt gelöscht werden soll (`thesisToDelete` im Fall des Beispiels). Dies bewirkt, dass das übergebene Objekt aus der Datenbank gelöscht wird.

Zu beachten ist, dass hier keine weiterführenden Konzepte wie beispielsweise das Löschen von potenziell verwaisten Objekten unterstützt werden. Soll derartiges realisiert werden, so ist dies manuell und separat per Löschkommando umzusetzen.

Abfragen des aktuell agierenden Nutzers

Neben Kommandos zum Suchen, Laden und Speichern sowie Löschen von Objekten existieren weiterhin einige Kommandos, die mit der Laufzeitumgebung des generierten Systems interagieren und somit nicht einfach per API-Aufruf ausgeführt werden können. Eines dieser Kommandos dient zum Abfragen des aktuell eingeloggten Nutzers. Dies ist beispielsweise dann sinnvoll, wenn Attributwerte oder Assoziationen zu anderen Objekten ermittelt oder gesetzt werden sollen. Ein Beispiel für dieses Kommando ist in Abbildung 8.30 zu finden.

AD

```
1 action AssignRef2 {
2   var : Lecturer actualUser;
3
4   cmd : actualUser = getActualUser();
5   // weiterer Inhalt
6 }
```

Abbildung 8.30.: Kommando zum Abfragen des aktuellen Nutzers

Bei diesem Kommando ist keine Spezifikation einer expliziten Klasse notwendig, sondern es genügt die Angabe des Schlüsselworts `getActualUser`. Diese Methode erfordert keine Parameter, da bei der Abfrage des aktuellen Nutzers keine Notwendigkeit existiert, diese in irgendeiner Art und Weise zu parametrieren.

Das `getActualUser`-Kommando liefert als Rückgabewert ein Objekt des Typs, der auf der linken Seite der Zuweisung verwendet wird, was in der Regel kontextabhängig ist. Im Fall des Beispiels handelt es sich um die Variable `actualUser` vom Typ `Lecturer` (Zeile 2), welche nach Ausführung des Kommandos den aktuellen Nutzer enthält. Allgemeingültig ist zu sagen, dass der Typ der Variablen von der Klasse `Person` oder generell derjenigen abgeleitet worden sein muss, die im Domänendatenmodell mit dem Stereotyp «`user`» annotiert wurde (vgl. hierzu Abschnitt 6.2.4).

Das zurückgelieferte Objekt kann im Folgenden weiterbearbeitet werden, beispielsweise in einem Java-Abschnitt oder aber per Übergabe an eine Seite, so dass darin entsprechende Änderungen vorgenommen werden können.

Zuweisen von Nutzern zu einer Rolle im Ablauf

Ein weiterer Anwendungsfall für das Objekt, welches den aktuellen Nutzer repräsentiert, ist dessen Zuweisung zu einem Aktivitätsbereich. Eine solche Zuweisung ist notwendig, damit Aktivitäten von verschiedenen Nutzern bzw. in Teilen auch parallel ausgeführt werden können. In Abbildung 8.31 ist dargestellt, wie eine solche Zuweisung syntaktisch zu spezifizieren ist.

AD

```
1  activity GradeThesis {
2
3      role Referee1 { AssignRef2, SetGrade1 ;}
4
5      action AssignRef2 {
6        var : Collection<Lecturer> actualUserSet, Lecturer actualUser;
7
8        cmd : actualUser = getActualUser();
9        java : {
10         actualUserSet = new HashSet<Lecturer>();
11         actualUserSet.add(actualUser);
12       }
13       cmd : assignRole(Referee1, actualUserSet);
14       // Auswahl des zweiten Gutachters und Zuordnung
15     }
16
17     // weitere Aktionen und Kanten
18 }
```

Abbildung 8.31.: Kommando zum Zuweisen von Nutzern zu Rollen

In diesem Beispiel sind diejenigen Teile der Aktivität dargestellt, die an die Zuweisung von Nutzern zu Aktivitätsbereichen beteiligt sind. Zunächst wird in Zeile 3 der Aktivitätsbereich `Referee1` definiert, dem im Folgenden ein Nutzer zugeordnet werden soll. Dies geschieht innerhalb der Aktion `AssignRef2` (Zeilen 5-15), worin neben der dargestellten Zuweisung des Erstgutachters, wie der Name der Aktion schon andeutet, auch der Zweitgutachter festgelegt wird. Dies ist aus Gründen der Übersichtlichkeit an dieser Stelle nicht weiter dargestellt, funktioniert aber analog.

Das Kommando zur Zuordnung von Nutzern zu Aktivitätsbereichen ist in Zeile 13 dargestellt. Nach dem Schlüsselwort `assignRole` werden in Klammern zwei Parameter übergeben. Der erste Parameter (`Referee1` im Beispiel) gibt an, zu welchem Aktivitätsbereich die Nutzer zugeordnet werden sollen, der zweite Parameter muss eine Menge von Objekten sein, die dem vorgenannten Bereich zugewiesen werden. Im Fall des Beispiels wird hierzu zunächst der aktuelle Nutzer abgerufen (Zeile 8), welcher in Zeile 11 zur in Zeile 10 erzeugten Menge `actualUsers` hinzugefügt wird. Dieser zusätzliche Schritt ist notwendig, da prinzipiell mehrere Nutzer alternativ Ausführende sein können. Sollten mehrere Nutzer dem Bereich zugeordnet werden, so kann jeder dieser Nutzer die entsprechenden Aktionen bearbeiten. Hierbei gilt jedoch, dass eine Aktion lediglich einmalig ausgeführt werden kann und da-

nach auch potenziellen weiteren Nutzern nicht mehr zur Verfügung steht. Die Objekte der übergebenen Menge müssen Nutzerobjekten sein (d. h. vom Typ der Klasse oder abgeleitet von der Klasse, die per Stereotyp «`user`» annotiert wurde, vgl. Abschnitt 6.2.4), so dass sie auch tatsächlich registrierte Nutzer innerhalb des generierten Web-Informationssystems sein können. Die jeweiligen Nutzer erhalten die entsprechende Aktion zur Ausführung angeboten, sobald der Ablauf der Aktivität diese erreicht hat.

Hinzufügen von Fehlermeldungen zu einer Seite

Bei der Ausführung von Aktionen kann es vorkommen, dass Nachrichten, beispielsweise über aufgetretene Fehler während dem Abarbeiten der Programmlogik, an den Nutzer weitergegeben werden sollen. Hierzu ist ein konsistenter Mechanismus wünschenswert, auf den von jeder Aktion aus zugegriffen werden kann. Zu diesem Zweck wurde in MontiWIS das `addErrors`-Kommando definiert, über das Meldungen an eine Seite übergeben und darin dargestellt werden können. Die Verwendung dieses Kommandos ist in Abbildung 8.32 dargestellt.

AD

```
1  action SetGrade1 {
2    in : Thesis i;
3    var : Collection<String> errors;
4
5    java {
6      errors = new ArrayList<String>();
7      if (i == null) {
8        errors.add("Thesis == null!");
9      }
10   }
11   cmd : addErrors(errors);
12   view : SetGrade1(i);
13
14   // weiterer Inhalt
15 }
```

Abbildung 8.32.: Kommando zum Darstellen von Fehlermeldungen

Das Kommando selbst ist in Zeile 11 dargestellt. Es besteht im Wesentlichen aus dem Schlüsselwort `addErrors` und einem Parameter. Dieser muss eine `Collection<String>` (Zeile 3) sein, in der die einzelnen (Fehler)meldungen gesammelt werden. Im Beispiel wird in den Zeilen 5-10 zunächst ein neues `ArrayList<String>`-Objekt angelegt und dann, im Fall, dass der Eingangsparameter `i` `null` ist, eine entsprechende Nachricht zur Menge der Fehlermeldungen hinzugefügt. In jedem Fall werden die Nachrichten in Zeile 11 zur Darstellung dem Kommando übergeben. Dies hat innerhalb einer Aktion *vor* dem Aufruf der Seite selbst (im Beispiel in Zeile 12) zu geschehen, so dass diese auch noch dargestellt werden können.

Eine weitere Möglichkeit ist die Weitergabe der Nachrichten in eine nachgelagerte Aktion,

um beispielsweise Hinweise hinsichtlich fehlgeschlagener Konsistenzprüfungen darstellen zu können. Hierbei muss es sich nicht zwangsläufig um eine andere als die aktuelle Aktion handeln, sondern kann prinzipiell auch die gleiche Aktion sein. Auf diesem Weg können Meldungen, die erst nach Absenden einer Seite ermittelt werden können, ebenfalls dargestellt werden.

Neben der Möglichkeit, per Kommando Meldungen an eine Seite zu übergeben, bestünde prinzipiell auch die Möglichkeit, diese Meldungen innerhalb einer Seite direkt zu codieren oder aber in dezidierten Seiten einen Parameter vorzuhalten, der entsprechend mit Meldungen belegt werden kann, jedoch wurde davon Abstand genommen, um über das Kommando eine ubiquitäre und konsistente Möglichkeit in sämtlichen Seiten zur Verfügung zu haben.

8.3.6. Aufruf anderer Aktivitäten

Bislang wurden die Inhalte einer Aktion beschrieben, die direkt innerhalb der Aktion festgelegt wurden. Abgesehen davon existiert die Möglichkeit, das Verhalten einer Aktion in einer anderen Aktivität zu spezifizieren, die dann anstelle der Aktion ausgeführt wird. Dies wird in der UML auch *CallBehaviourAction* (siehe Abschnitt 12.3.14 in [Obj11c]) genannt. Darüber ist es möglich, komplexes Verhalten in verschiedenen Kontexten wiederzuverwenden. Ein solcher Aufruf ist in Abbildung 8.33 dargestellt.

AD

```
action SaveAndNotify {
    in : Thesis i;
    out : Thesis o;

    calls SaveThesisAndNotify;

    i -> SaveThesisAndNotify.i;
    SaveThesisAndNotify.o -> o;
}
```

AD

```
activity SaveThesisAndNotify {
  in : Thesis i;
  out : Thesis o;

  action SaveThesis {
    in : Thesis i;
    out : Thesis o;

    // Logik zum Speichern
  }

  action Notify {
    in : Thesis i;
    out : Thesis o;
    // Parameter und Logik
    // für Email-Benachrichtigung
  }

  i -> SaveThesis.i;
  // weitere Kantendefinitionen
  Notify.o -> o;
}
```

Abbildung 8.33.: Aufruf einer Aktivität (rechts) aus einer Aktion heraus (links)

Auf der linken Seite von Abbildung 8.33 ist die Aktion `SaveAndNotify` dargestellt, deren

Verhalten in der Aktivität `SaveThesisAndNotify` auf der rechten Seite des Beispiels definiert ist. Eine solche Aktion darf in MontiWIS keine Logik enthalten, sondern lediglich den Verweis auf die aufgerufene Aktivität. Dies geschieht nach dem Schlüsselwort `calls` über den Aufruf des Namens der Aktivität (siehe Zeile 5). Neben dem eigentlichen Aufruf der Aktivität beinhaltet die Aktion weiterhin sowohl Eingangs- als auch Ausgangsparameter (Zeilen 2-3), die in den Zeilen 7-8 den Eingangs- und Ausgangsparametern der Aktivität zugeordnet werden. Dies geschieht durch Verknüpfung des Aktionseingangsparameters auf der linken Seite der Zuweisung mit dem zugehörigen Eingangsparameter der Aktivität. Als Zuweisungssymbol wird analog zu Kantendefinitionen ein Pfeil (`->`) genutzt und der Aktivitätseingangsparameter wird per Namen der Aktivität und dem Namen des Parameters, getrennt von einem Punkt, spezifiziert. Ähnlich wird die Zuweisung des Ausgangsparameters der Aktivität zum Ausgangsparameter der Aktion vorgenommen, wobei hierbei die Reihenfolge jedoch entsprechend vertauscht ist.

Die Aktivität, die das Verhalten der Aktion spezifiziert, ist auf der rechten Seite von Abbildung 8.33 dargestellt. Diese Aktivität ist eine derjenigen, die Eingangs- und Ausgangsparameter auf Aktivitätsebene besitzt, so dass Objekte der aufrufenden Aktion hierin weiterverarbeitet und ebenso Ergebnisse an die aufrufende Aktion zurückgegeben werden können. Die Typen der Parameter müssen mit denen der aufrufenden Aktion übereinstimmen, die Namen entsprechen sich im Beispiel zwar, müssen es aber natürlich nicht zwangsläufig. Wie im Beispiel zu sehen, müssen Aktivitäten, die von extern aufgerufen werden und über Parameter verfügen, weder `initial` noch `final`-Knoten besitzen, da deren Funktion bereits durch die Parameter mit übernommen wird. Dementsprechend werden die Parameter auch per Kanten verbunden (Zeilen 19-21), was dazu führt, dass einerseits die Objekte, die der Aktivität von der aufrufenden Aktion übergeben wurden, entlang der Kanten zur ersten Aktion bzw. deren Parametern weitergereicht werden, und andererseits die auf den Ausgangsparametern befindlichen Objekte der aufrufenden Aktion wieder zur Verfügung stehen.

Neben Aktivitäten mit Parametern aus Aktionen heraus aufzurufen, können auch Aktivitäten mit gewöhnlichen `initial`- und `final`-Knoten in hierarchischer Weise gestartet werden, was allerdings aufgrund der Tatsache, dass in diesem Fall keine Funktionalität der übergeordneten Aktivität und seiner Objekte übernommen werden kann, eher selten vorkommen dürfte.

8.4. Generierungsprozess

Die Verarbeitung der Modellinstanzen von Aktivitäten erfolgt in mehreren Schritten, wie bereits in Kapitel 4 sowie im Kontext von anderen Modellen in den Abschnitten 6.3 und 7.3 beschrieben wurde. Zunächst werden einige Kontextbedingungen geprüft, wodurch Hinweise auf fehlerhafte Modelle entdeckt und gemeldet werden. Ein Überblick darüber wird in Abschnitt 8.4.1 gegeben. In einem zweiten Schritt werden die Modelle per Visitor bearbeitet und daraus eine Zwischenstruktur (erläutert in Abschnitt 8.4.2) erzeugt (beschrieben in Abschnitt 8.4.3), die nachfolgend als Eingabe für einige Rekonstruktionsschritte dienen, die in Abschnitt 8.4.4 beschrieben sind. Schließlich beschreibt Abschnitt 8.4.5, wie aus der modifizierten Zwischenstruktur die zugehörigen Teile des Web-Informationssystems generiert werden.

8.4.1. Kontextbedingungen

Bei der Spezifikation von Modellen existieren in der Regel nicht nur diejenigen Fehler, die bereits durch syntaktische Überprüfungen gefunden und angezeigt werden können, sondern auch diejenigen, die sich erst im Kontext von mehreren Modellen einer oder auch mehrerer Arten prüfen lassen. Die entsprechenden Überprüfungen werden im Folgenden erläutert.

Verwendete Symboltabelleneinträge

Ein Teil der Kontextbedingungen wird über die Symboltabelle geprüft, die für die Aktivitätsdiagramme aufgebaut werden. Darüber wird einerseits die Struktur der Modelle abgebildet, andererseits werden aber auch die Zusammenhänge von Elementen eines Modells zu Elementen anderer Modelle nachvollziehbar und vor allem überprüfbar gemacht. Die dafür verwendete Klassenstruktur, die eben diese Zusammenhänge abbildet, ist in Abbildung 8.34 dargestellt.

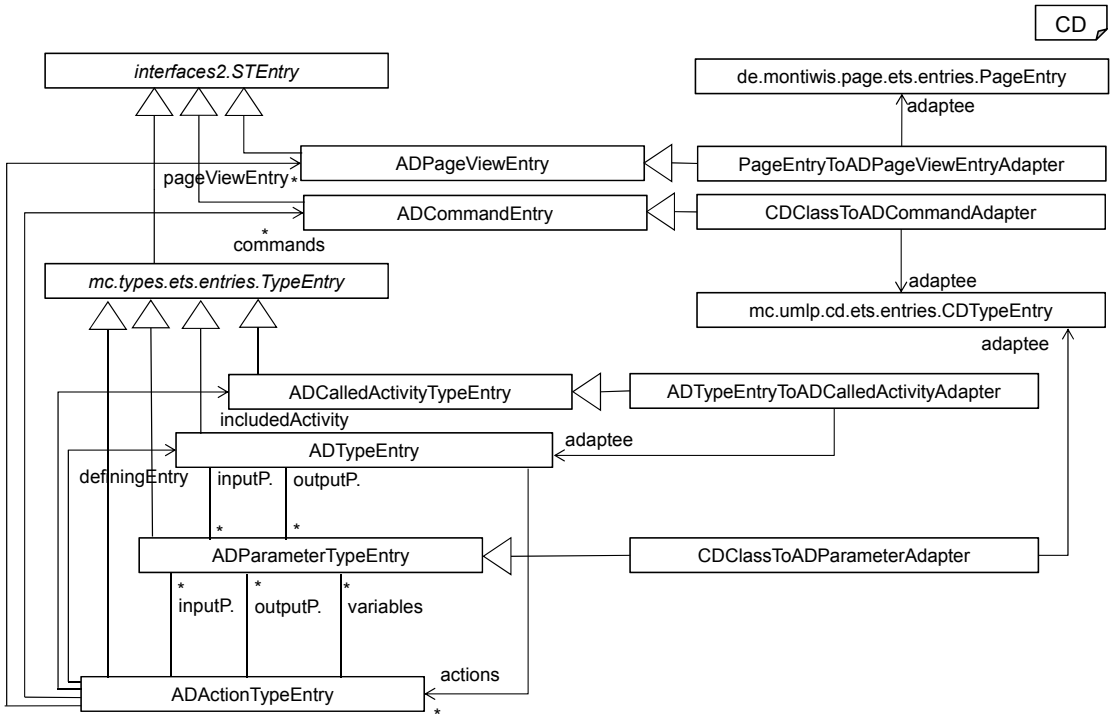


Abbildung 8.34.: Symboltabelleneinträge für Aktivitätsdiagramme

Wie in Abschnitt 3.3.2 erläutert, gehören die Klassen `TypeEntry` und `STEntry` zur Symboltabelleninfrastruktur. Von ihnen werden im Wesentlichen sämtliche Einträge abgeleitet, die die Struktur der Aktivitätsdiagramme und deren Inhalte abbilden. `TypeEntry` ist von `STEntry` abgeleitet und besitzt zusätzlich Attribute wie `name` und `packageName`, wohingegen `STEntry` die rudimentären Funktionalitäten eines Symboltabelleneintrags bereitstellt.

Die zentralen Symboltabelleneinträge für Aktivitäten sind `ADTypeEntry`, welcher eine Aktivität repräsentiert sowie `ADActionTypeEntry`, welcher zu jeder Aktion einer Aktivität angelegt wird. Zur Abbildung von sowohl Eingangs- und Ausgangsparametern (`inputP.` sowie `outputP`) als auch Variablen (`variables`) in einer Aktion wird der Eintrag `ADParameter"TypeEntry` verwendet, der in beiden Einträgen mehrfach in diesen Rollen vorkommen kann. `ADActionTypeEntries` sind selbstverständlich potenziell mehrfach in einem `ADTypeEntry` enthalten (`actions`), wohingegen ein Aktionseintrag einem Aktivitätseintrag zugeordnet ist (`definingEntry`).

Der Inhalt von Aktionen (`ADActionTypeEntry`) wird über verschiedene Symboltabelleneinträge abgebildet. Kommandos werden über `ADCommandEntry` repräsentiert, Seitenaufrufe lassen sich über `ADPageViewEntry` darstellen und Aufrufe einer anderen Aktivität werden über `ADCalledActivityTypeEntrys` in der Symboltabelle abgelegt. Zu jedem dieser Einträge existiert ein Adaptereintrag, über den eine Verbindung mit den referenzierten Einträgen hergestellt wird. Bei Kommandos, die in Verbindung zu einer bestimmten Domänenklasse definiert sind, wird über einen `CDClassToADCommandAdapter` eine Referenz auf den entsprechenden `CDTypeEntry` abgelegt. Bei Seitenaufrufen wird über den `PageEntryToADPageViewEntryAdapter` eine Verbindung zum Symboltabelleneintrag `PageEntry` (vgl. hierzu Abschnitt 7.3.1) hergestellt. Die Parameter von Aktionen und Aktivitäten, die über `ADParameterTypeEntry` abgebildet werden, können per `CDClassToADParameterAdapter` ebenfalls mit einem `CDTypeEntry` in Verbindung gebracht werden, welcher eine Klasse aus dem Domänendatenmodell repräsentiert. Innerhalb der Aktivitätsdiagrammeinträge lässt sich über einen `ADTypeEntryToADCalledActivityAdapter` der Aktivitätseintrag (`ADTypeEntry`) referenzieren, der in einer Aktion per `ADCalledActivityTypeEntry` aufgerufen wurde.

Unter Zuhilfenahme der beschriebenen Symboltabelleneinträge lässt sich die Konsistenz der Aktivitätsdiagramme überprüfen. In den folgenden Abschnitten werden diejenigen Sachverhalte beschrieben, die aktuell in MontiWIS automatisch sichergestellt werden und bei deren Verletzung entsprechende Warnungen während der Modellverarbeitung ausgegeben werden.

Korrekte Referenz von Klassen aus dem Domänendatenmodell

Wie in Abbildung 8.34 zu sehen, werden sowohl Parameter innerhalb von Aktivitäten und Aktionen als auch einige Kommandos auf Klassen aus dem Domänendatenmodell adaptiert. Während der Qualifizierung der Einträge (für eine kurze Erläuterung dieses Mechanismus siehe Abschnitt 4.3) durch die Klassen `CDClassToADParameterQualifier` sowie `CDClassToADCommandQualifier` wird bereits überprüft, ob die entsprechend referenzierte Klasse existiert und der zugehörige `CDTypeEntry` zugeordnet werden kann. Sollte dies fehlschlagen, wird eine entsprechende Warnung ausgegeben.

Korrekter Aufruf von Seiten aus einer Aktion

Der korrekte Aufruf einer Seite wird auf zweierlei Arten überprüft. Einerseits stellt die Qualifizierung der Seitenadapter durch die Klasse `PageEntryToADPageViewQualifier` sicher, dass ein namentlich passender `PageEntry` existiert und dieser korrekt gesetzt werden kann.

Andererseits wird über die Kontextbedingung `ADCorrectParameterNumberOnViewChecker` überprüft, dass beim Aufruf der Seite aus einer Aktion heraus die gleiche Anzahl von Parametern aufgerufen wird, wie sie in der Seite definiert wurde. Eine Zuweisungskompatibilität wird aktuell nicht überprüft, ließe sich jedoch einfach nachrüsten.

Korrekter Aufruf von Aktivitäten aus Aktionen heraus

Auch der Aufruf einer Aktivität aus einer Aktion heraus wird über verschiedene Mechanismen überprüft. Zum einen qualifiziert die Klasse `ADTypeEntryToADCalledActivityQualifier` den `ADTypeEntryToADCalledActivityAdapter` in der Art, dass der zugehörige `ADTypeEntry` aufgelöst und im Adapter gesetzt wird. Dies stellt die Existenz der referenzierten Aktivität sicher. Andererseits wird in der Klasse `ADCorrectParameterUsageInHierarchicalActivitiesChecker` sichergestellt, dass die referenzierten Parameter in der aufgerufenen Aktivität tatsächlich existieren und ebenfalls korrekt als Eingangs- bzw. Ausgangsparameter definiert und auch in dieser Rolle verknüpft sind. Auch hier findet zunächst keine Überprüfung hinsichtlich Typkompatibilität statt, was aber auch an dieser Stelle mit begrenztem Aufwand möglich wäre.

Eindeutige Namen von Parametern und Variablen in einer Aktion

Sowohl Parameter als auch Variablen werden innerhalb von Aktionen analog zu Klassenvariablen in Programmiersprachen wie beispielsweise Java verwendet. Damit dienen sie unter anderem zum Austausch von Objekten über die verschiedenen Elemente einer Aktion hinweg, so dass beispielsweise geladene Objekte Seiten übergeben oder sie in Abschnitten mit Java-Code manipuliert werden können. Ebenso wie bei Klassenvariablen müssen folglich auch Parameter und Variablen innerhalb einer Aktion einen eindeutigen Namen besitzen. Dies wird durch die Kontextbedingung `ADDuplicateParameterNameInActionChecker` überprüft.

Eindeutige Namen von Parametern und Knoten in einer Aktivität

Auch wenn in einer Aktivität nicht die Möglichkeit besteht, Variablen zu deklarieren, gilt dennoch die Anforderung der Namenseindeutigkeit. Dies betrifft in diesem Fall jedoch nicht nur potenzielle Eingangs- und Ausgangsparameter von Aktionen, sondern schließt ebenfalls die Namen der Kontrollknoten mit ein. Wie in Abschnitt 8.2.3 beschrieben, werden Aktivitätsparameter ebenfalls in Kanten analog zu Knoten verwendet, weswegen hier Ambiguitäten zu vermeiden sind. Diese Anforderung wird über die Kontextbedingung `ADDuplicateParameterOrNodeNamesChecker` geprüft.

Korrekte Zuordnung von Aktionen zu Aktivitätsbereichen

Wie in Abschnitt 8.2.3 erläutert, dienen Aktivitätsbereiche dazu, bestimmte Aktionen durch einen bestimmten Nutzer innerhalb der Applikation ausführen zu lassen. Bei der Definition von Aktivitätsbereichen ist es daher nicht zulässig, andere Knoten als existierende Aktionen

zu Aktivitätsbereichen zuzuordnen. Diese Einschränkung wird in der Kontextbedingung `ADExistingActionsReferencedInRoleChecker` überprüft.

Korrekte Verwendung von Knoten und Parametern in Kantendeklarationen

Bei der Spezifikation von Kanten in einer Aktivität können leicht Verwechselungen bzw. Tippfehler auftreten, weswegen hier ebenfalls eine Überprüfung der korrekten Verwendung von Knoten- und Parametern stattfinden sollte. Hierbei wird sowohl die Existenz von Knoten als auch die korrekte Verwendung ihrer Parameter überprüft. Für diese Aufgabe ist die Kontextbedingung `ADCorrectNodeInTransitionChecker` zuständig.

8.4.2. Aufbau der Zwischenstruktur

Wie in MontiWIS üblich (siehe Abschnitt 4.4) wird auch aus den Aktivitätsdiagrammen eine Klassenstruktur erzeugt, die eine einfachere Verarbeitung durch die Templates erlaubt. Diese Klassenstruktur ist in Abbildung 8.35 dargestellt und wird nachfolgend erläutert.

Speicherung von Aktivitäten und deren Struktur

In der Zwischenstruktur dient die Klasse `Activity` dazu, Information über den Aufbau einer Aktivität aufzunehmen und zu verarbeiten. Neben dem Namen und dem Paketnamen (`getName` bzw. `getPackageName`) werden darin primär die Objekte gespeichert, die die Knoten und Kanten repräsentieren. Knoten werden in `Map`-Strukturen, unterteilt nach Knotentyp, abgelegt, wobei der Name des Knotens hier als Schlüssel dient. Der Zugriff erfolgt über `get`-, `set`- und `add`-Funktionen, angedeutet über `getDecisionNode` bzw. `setDecisionNode` und `addDecisionNode`. Kanten werden über die Klasse `Edge` abgebildet und enthalten zum einen die Knoten, zwischen denen die Kante gezogen werden und zum anderen, sofern zutreffend, die Namen der Parameter, die mit der Kante verbunden werden (stellvertretend repräsentiert über die Funktionen `getLeftHandSidePinName` und `setLeftHandSidePinName`). Ferner enthält eine Kante den Schlüssel, unter dem die Bezeichnung des Knopfes für diese Kante in der `ResourceBundle`-Datei abgelegt wird. Auch in Aktivitäten werden ihre Parameter gespeichert, jeweils unterteilt nach Eingangs- (`inputParam.`) und Ausgangsparametern (`outputParam.`). Dies sind diejenigen Informationen, die für die erfolgreiche Generierung über Templates ausreichen.

Abbildung von Knoten

Den Knoten einer `Activity` in der Zwischenstruktur sind diejenigen Methoden gemein, die in der Schnittstelle `Node` definiert sind. Diese bieten überwiegend Zugriffsmethoden auf Namen und Parameter (`addInputParameter` bzw. `getInputParameter`) sowie einige Hilfsmethoden, die bei der Generierung Zugriff auf Klassen- oder Objektnamen (`getName` bzw. `getInstanceName`) bieten. Implementiert wird ein Großteil dieser Methoden in der Klasse `AbstractNode`, die sowohl die entsprechenden Attribute enthält, als auch die Funktionalität für viele dieser Methoden bereitstellt.

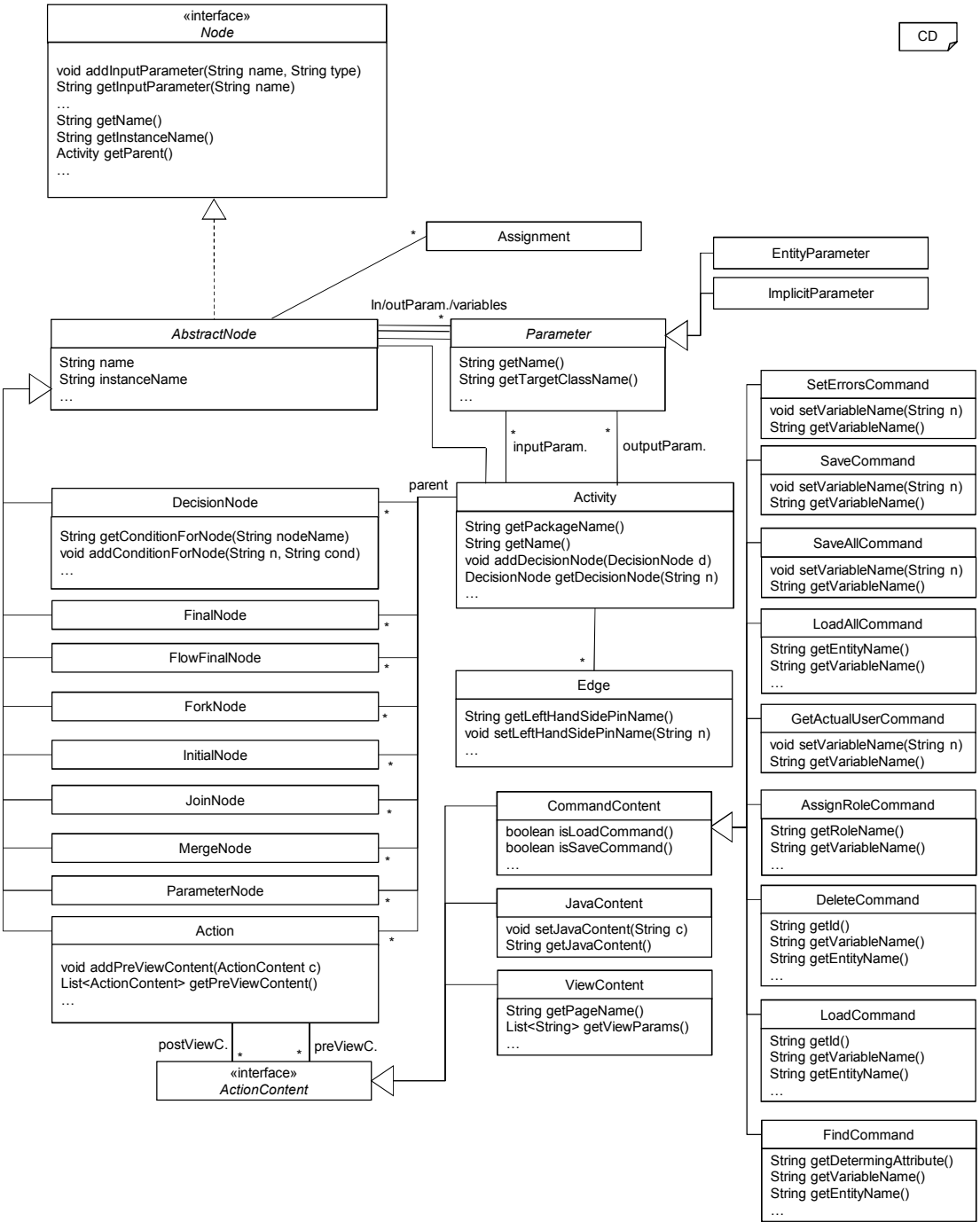


Abbildung 8.35.: Zwischenstruktur für Aktivitätsdiagramme

Von `AbstractNode` abgeleitet werden sämtliche Klassen, die die konkreten Knoten repräsentieren. Dies sind `FinalNode`, `FlowFinalNode`, `InitialNode`, `JoinNode` und `MergeNode`, die intuitiv die notwendigen Informationen zu diesen Knoten repräsentieren. Ein `DecisionNode` enthält zusätzlich zu den gewöhnlichen Informationen weiterhin Felder, um die zugehörigen Bedingungen aufzunehmen und zu verarbeiten (`getConditionForNode` bzw. `setConditionForNode`). Ein `ParameterNode` repräsentiert die Eingangs- und Ausgangsparameter einer Aktivität und enthält Informationen über seine Art (Eingangs- oder Ausgangsparameter) und Domänendatentyp des Parameters.

Aktionen und Aktionsinhalt

Aktionen werden über die Klasse `Action` abgebildet. Diese besitzt neben den bereits in `AbstractNode` spezifizierten Methoden zusätzlich Informationen über den eigentlichen Inhalt der Aktion. Der Inhalt einer Aktion wird über die Klassen spezifiziert, die die Schnittstelle `ActionContent` implementieren. Dies sind sowohl Seitenaufrufe, die über `ViewContent` abgebildet werden, Java-Code, der in `JavaContent` gespeichert wird sowie Kommandos, die über die Kindklassen von `CommandContent` in der Zwischenstruktur abgelegt werden. Die Zuordnung und der Abruf der jeweiligen Inhalte geschieht, unterteilt nach Geschäftslogik, vor und nach dem Aufruf potenzieller Seitenaufrufe (stellvertretend dargestellt über die Funktionen `addPreViewContent` bzw. `getPreViewContent`).

`ViewContent` enthält im Wesentlichen den Seitennamen (`getPageName`) sowie die Liste mit den Parameternamen (`getViewParams`), die der Seite übergeben werden.

In `JavaContent` wird der Inhalt des Java-Bereiches in der Form, wie er in der Aktion angegeben wurde, als `String` gespeichert (`getJavaContent` bzw. `setJavaContent`). Umbauten dieser Inhalte erfolgen zum aktuellen Stand von MontiWIS nicht.

Schließlich werden Kommandos in Aktionen über diejenigen Klassen abgebildet, die von `CommandContent` abgeleitet sind. Diese Klasse enthält Funktionsrümpfe, über die sich die Art des abgeleiteten Kommandos abfragen lässt (`isLoadCommand` und `isSaveCommand` stellvertretend für alle anderen). Die abgeleiteten Klassen sind im Einzelnen die Klassen `SetErrorsCommand`, `SaveCommand`, `SaveAllCommand`, `LoadAllCommand`, `GetActualUserCommand`, `AssignRoleCommand`, `DeleteCommand`, `LoadCommand` und `FindCommand`. In jedem dieser Zwischenstrukturelemente werden entsprechend die Informationen abgelegt, die für das jeweilige Kommando relevant sind. So wird beispielsweise beim `LoadAllCommand` der Name der Variablen gespeichert, der das Ergebnis zugewiesen wird, sowie der Name der Klasse, deren Objekte geladen werden sollen. Analog dazu werden die Informationen in den anderen `Command`-Klassen gespeichert, die in deren Kontext genötigt werden.

Zwischenstrukturspeicher in ADGlobalStore

Wie auch im Fall von Domänendatenmodell mit der Klasse `CDGlobalStore` und Seiten mit der Klasse `PageGlobalStore` werden die Objekte der Zwischenstruktur, die aus Aktivitätsdiagrammen erzeugt werden, in einer zentralen Klasse abgelegt und können darüber verwaltet und abgerufen werden. Die Schnittstelle dieses Speichers wird definiert in der Schnittstelle `ADGlobalStore` und die eigentliche Speicherung erfolgt in der Klasse

`DefaultADGlobalStore`. Auch Letztere ist als `@Singleton` annotiert, so dass beim Verwenden des Objekts sichergestellt wird, dass lediglich nur eines davon existiert. Ein Überblick über Schnittstelle und implementierende Klasse ist in Abbildung 8.36 dargestellt.

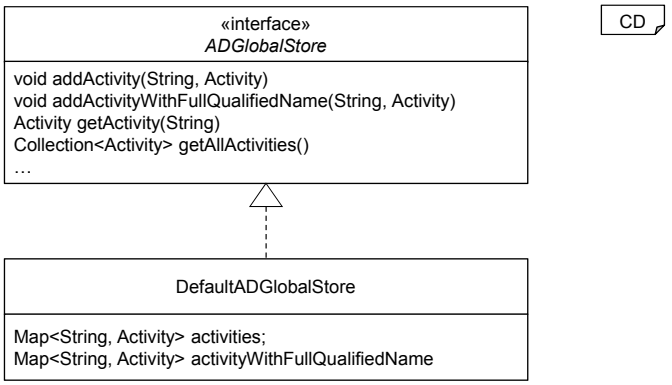


Abbildung 8.36.: Zentraler Speicher für Aktivitätsinformationen

Hinsichtlich der Funktionalität dieser Klasse ist über die Speicherung und den Zugriff auf die verschiedenen Objekte hinaus keine erwähnenswerte weitere Funktionalität enthalten. Anzumerken ist hierbei dennoch, dass hier nicht jede Art von Objekten der Zwischenstruktur aufgenommen werden, sondern lediglich Aktivitäten (`addActivity` bzw. `getActivity` und `getAllActivities` zum Iterieren über sämtliche Objekte). Im Verlauf der Bearbeitung erfolgt ein Zugriff auf andere Knoten in der Regel über die Zugriffsmethoden, die eine `Activity` zur Verfügung stellt. Dies rührt insbesondere daher, dass innerhalb einer Aktivität die Namen der Knoten ohnehin eindeutig sein müssen, was aktivitätsübergreifend nicht der Fall ist.

8.4.3. Modellverarbeitung von Aktivitätsdiagrammen durch einen Visitor

Nach der Überprüfung von Syntax und den eben beschriebenen semantischen Kontextbedingungen erfolgt die Verarbeitung der Aktivitätsdiagramme durch einen Visitor, um aus der AST-Struktur die in Abschnitt 4.4 erläuterte Zwischenstruktur aufzubauen.

Die Funktionalität zur Bearbeitung der Aktivitätsdiagramme ist implementiert in der Klasse `ADTransformatorConcreteVisitor`. Auch dort wird statt der standardmäßigen `visit`-Methode die `ownVisit`-Methode verwendet, durch die die Steuerung der Reihenfolge der Bearbeitung möglich ist. Diese ist bei Aktivitätsdiagrammen insofern relevant, als dabei zunächst Knoten und erst danach Aktivitätsbereiche und Kanten bearbeitet werden sollten. Knoten sollten in der Regel bereits in der Zwischenstruktur angelegt worden sein, bevor sie in weiteren Konstrukten verwendet werden.

Nachfolgend wird kurz auf die Bearbeitung der einzelnen Teile einer Aktivität eingegangen und erläutert, welche Besonderheiten dabei zu beachten sind.

Bearbeitung von ASTADDefinition

Das Element `ASTADDefinition` repräsentiert innerhalb der AST-Struktur der Aktivitätsdiagramme dasjenige, das die Aktivität mit ihrem Inhalt abbildet. Bei dessen Bearbeitung durch den Visitor wird sowohl das `Activity`-Element erzeugt und mit Attributen wie Namen und Paketnamen belegt, als auch die Parameter der Aktivität bearbeitet. Hierzu werden die AST-Elemente, die den Eingangs- und Ausgangsparametern entsprechen, innerhalb der Methode des Visitors verarbeitet und die Parameter dem Zwischenstrukturelement der Aktivität hinzugefügt. Neben den eigentlichen Parametern werden ebenfalls `ParameterNode`-Objekte erzeugt, die zu einem Knoten in einer Kantendeklaration werden und entsprechend verwendet werden können.

Das AST-Element enthält neben Parametern eine beliebige Anzahl von `ASTADElement`-Kindknoten. Da dies als Schnittstelle implementiert ist, können verschiedene AST-Elemente an dieser Stelle in beliebiger Reihenfolge auftreten. Dies sind zum einen sämtliche Knoten in der Aktivität, zum anderen jedoch auch Kantendeklarationen, die zwischen den Knoten hergestellt werden. Bei der Bearbeitung dieser Elemente ist es sinnvoll, dass zunächst sämtliche Knoten bearbeitet werden, so dass Informationen über deren Namen und Parameter bereits vorliegen, wenn Kanten oder auch Aktivitätsbereiche bearbeitet werden. Da in MontiWIS der Ansatz verfolgt wird, dass Kindknoten des AST per explizitem Aufruf der `ownVisit`-Methoden besucht werden, lassen sich diese Vorrangbeziehungen einfach über die Reihenfolge der expliziten Knotenbesuche abbilden. Dazu wird über `ASTADElement`-Knoten iteriert, wobei Knoten direkt besucht, Kanten und Aktivitätsbereich jedoch für einen zweiten Durchlauf zwischengespeichert und erst im Anschluss daran verarbeitet werden.

Bearbeitung von ASTADAction

Bei der Verarbeitung eines `ASTADAction`-Elements werden in einem neuen `Action`-Objekt zunächst die notwendigen Attribute wie beispielsweise Name und Elternaktivität gesetzt. Der Name entspricht hierbei dem Namen der Aktion, da dieser innerhalb der Aktivität bereits eindeutig sein muss. Anschließend wird der eigentliche Inhalt verarbeitet, was abhängig von der Art des Inhalts ist. Ferner werden in der Aktion sowohl Eingangs- und Ausgangsparameter als auch Variablendeklarationen hinzugefügt.

Zunächst wird überprüft, ob die Aktion eine andere Aktivität aufruft. In diesem Fall kann eine Aktion keinen weiteren Inhalt besitzen und es werden sowohl Informationen bzgl. der Kindaktivität als auch potenzielle Zuweisungen von Variablen in die Zwischenstruktur übernommen.

Sollte dies nicht der Fall sein und die Aktion Java-Code enthalten, Seiten anzeigen oder eines oder mehrere der domänenspezifischen Kommandos ausführen, so werden die entsprechenden Visitoren aufgerufen, die für den jeweiligen Aktionsinhalt spezifiziert wurden. Für Java-Code kommt hierfür der `JavaContentTransformator` zum Einsatz, Seitenaufrufe werden über den `ViewTransformator` bearbeitet und Kommandos werden per `CommandContentTransformator` in die zugehörige Zwischenstruktur überführt. Sofern ein Seitenaufruf Teil einer Aktion ist, werden die Kommando- und Java-Inhalte einer Aktion unterschieden in diejenigen, die vor dem Seitenaufruf erfolgen und die, die nach dem Aufruf

der Seite ausgeführt werden. Die jeweiligen Zwischenstrukturelemente werden dem `Action`-Objekt über die Methoden `addPreViewContent` bzw. `addPostViewContent` hinzugefügt, um sie auch bei der Generierung entsprechend unterscheiden zu können.

Bearbeitung der weiteren Knoten

Die Transformation der übrigen Knotentypen aus dem AST in die Zwischenstruktur erfolgt relativ linear. In der Regel wird aus einem AST-Knoten der entsprechende Zwischenstrukturknoten per Visitor erzeugt. Der Name wird dabei direkt vom Namen des Knotens innerhalb der Aktivität übernommen, da auf diese Art sämtliche Anforderungen hinsichtlich Eindeutigkeit erfüllt sind. Bei Knoten, die die Angabe von Zuweisungen innerhalb des Knotens erlauben, um die Durchleitung von Eingangs- zu Ausgangsparametern bei der Ausführung des Knotens zu realisieren, werden diese Informationen dem jeweiligen Zwischenstrukturelement hinzugefügt.

Eine Sonderbehandlung erfahren jedoch Start- und Endknoten, die nicht explizit als Knoten angegeben werden, sondern implizit per Schlüsselwort in einer Kante angegeben werden. In der Zwischenstruktur werden dennoch explizite Objekte hierfür angelegt, was aber nicht durch eine eigene `ownVisit`-Methode geschieht, sondern während der Verarbeitung der Kantendeklarationen durchgeführt wird.

Bearbeitung von Kantendeklarationen

Eine recht umfangreiche Bearbeitung erfahren die Kantendeklarationen in einer Aktivität. Da die Syntax der Aktivitätskanten in MontiWIS recht vielfältige Konstruktionen erlaubt, sollten diese zu einer möglichst einheitlichen Abbildung in der Zwischenstruktur führen. Aus diesem Grund werden sämtliche Kanten zu 1:1 Verknüpfungen zwischen Knoten bzw. deren Ausgangs- und Eingangsparametern transformiert, so dass nur einheitliche Kanten zwischen zwei Knoten bzw. Parametern im Zwischenstrukturspeicher abgelegt werden. Dies hat zur Folge, dass auch die Kurzformen, die für die implizite Verwendung von Kontrollknoten erlaubt sind, auf eben solche 1:1 Kanten abgebildet werden. Die verschiedenen Möglichkeiten von Kantendeklarationen und deren Behandlung wird nachfolgend kurz erläutert.

Analog zur Grammatik der Aktivitätsdiagrammsprache (vgl. hierzu Abschnitt 8.2.1), wo Kanten (Nichtterminal `ADEdge`) aufgeteilt wurden in deren linke Seite (Nichtterminal `ADLeftHandSide`) und rechte Seite (Nichtterminal `ADRightHandSide`), erfolgt die Bearbeitung der Kanten auch in dieser Reihenfolge.

Nachdem ein `Edge`-Objekt der Zwischenstruktur erstellt wurde, wird zunächst die linke Seite der Kante verarbeitet. Im einfachsten Fall ist auf dieser der Name eines Knotens (und evtl. eines Parameters) zu finden, was dazu führt, dass dieser Knoten (Aktion oder Parameterknoten) aus der Zwischenstruktur abgerufen und dem neuen Kantenobjekt hinzugefügt wird.

Umfangreicher sind die Bearbeitungsschritte im Fall, dass es sich bei der linken Seite um die Kurzformen von Kanten mit Verbindungs- oder Synchronisationsknoten handelt. Dabei sind auf der linken Seite der Kantendeklaration mehrere Knoten angegeben, die implizit per Kontrollknoten verknüpft sind. Die Behandlung der Kurzformen ist schematisch in Tabelle 8.37 dargestellt. Diese Umformungen finden nicht, wie dargestellt, auf konkreten Modellen statt,

sondern werden direkt in die Zwischenstruktur übertragen. Der Einfachheit der Darstellung halber sind auch die Namen der verwendeten Knoten und Parameter nicht zwangsläufig identisch mit den dargestellten.

	kompakte Kantendeklaration	Ergebnis

1.	`A | B -> C`	`A -> G_Merge`
		`B -> G_Merge`
		`G_Merge -> C`

2.	`A & B -> C`	`A -> G_Join`
		`B -> G_Join`
		`G_Join -> C`

3.	`A.o -> [o == true] B.i |`
	`[o == false] C.i`
		`A.o -> G_Decision.i`
		`G_Decision.o -> [o == true] B.i`
		`G_Decision.o -> [o == false] C.i`
		
		`decision G_Decision {`
		`  o = i`
		`}`

4.	`A -> B & C`	`A -> G_Fork`
		`G_Fork -> B`
		`G_Fork -> C`

Tabelle 8.37.: Umformungen bei der Bearbeitung von Kantendeklarationen

In 1. und 2. von Tabelle 8.37 ist zu sehen, wie mit impliziten Verbindungs- bzw. Synchronisationsknoten auf der linken Seite umgegangen wird. Diese werden in einen expliziten Knoten (`G_Merge` bzw. `G_Join`) umgesetzt, der von den Knoten `A` und `B` mit Kanten verbunden wird. Dieser neu eingeführte Kontrollknoten wird seinerseits mit dem Zielknoten `C` verknüpft, was jedoch erst im zweiten Teil bei der Bearbeitung der rechten Seite der Kante geschieht. Für den Fall, dass die Kanten nicht lediglich Knoten, sondern auch Parameter verknüpfen, würden im erzeugten Knoten selbstverständlich die entsprechenden Durchleitungen der Parameter durch den Knoten festgelegt, wie es etwa in 3. angedeutet ist.

Ebenfalls auf der linken Seite kann das Schlüsselwort `initial` stehen, was den Startknoten einer Aktivität repräsentiert. In diesem Fall wird in `InitialNode`-Objekt der Zwischenstruktur erzeugt und der Kante zugewiesen.

Nach der Bearbeitung der linken Seite einer Kante und der Erzeugung potenziell mehrerer Kanten erfolgt die Behandlung ihrer rechten Seite. Auch diese können wiederum von verschiedener Art sein, wobei hier eine einfache Aktion (oder allgemein ein singulärer Knoten) auf der linken Seite die einfachste denkbare Art ist. Diese kommt beispielsweise in 1. und

2. in Tabelle 8.37 vor. In diesem Fall werden sämtliche bislang erzeugten Kanten mit dem Knoten belegt, der in der Aktivität erzeugt wurde.

Komplizierter sind die Fälle, die in 3. und 4. dargestellt sind. Hier wird auf der rechten Seite ein Zwischenknoten (`G_Decision` bzw. `G_Fork`) erzeugt, der als Zielknoten für die ursprüngliche linke Seite dient (s. erste Zeile in 3. und 4.). Zusätzlich werden weitere Kanten erzeugt, die von diesem Zwischenknoten ausgehend die Knoten als Zielknoten adressieren, die in der ursprünglichen Kantendeklaration auf der rechten Seite zu finden waren.

Auch auf der rechten Seite einer Kantendeklaration können Schlüsselwörter stehen, die implizit Knoten repräsentieren. Diese werden ebenfalls in entsprechende Knoteninstanzen umgedeutet, die an dieser Stelle entweder erzeugt oder aus der Aktivität bezogen werden, sofern sie in einer anderen Kante bereits vorkamen. Hierbei handelt es sich um End- bzw. Ablaufendknoten, die durch `FinalNode` bzw. `FlowFinalNode` in der Zwischenstruktur repräsentiert werden.

Bearbeitung von Aktivitätsbereichen

Als letzter Schritt bei der Verarbeitung von Aktivitäten werden die *Aktivitätsbereiche* behandelt. Für jeden Aktivitätsbereich werden die Aktionsnamen durchlaufen, die in ihm angegeben sind. Bei der Bearbeitung werden die Aktionen aus der Aktivität abgefragt und darin der entsprechende Rollenname in der Aktion gesetzt. Dies genügt für die nachfolgende Verarbeitung durch den Codegenerator.

8.4.4. Rekonstruktion der Aktivitätsdiagrammelemente

Nach der Überprüfung der syntaktischen Korrektheit und Sinnhaftigkeit der Modelle und dem erfolgreichen Aufbau der Zwischenstruktur erfolgt auch bei der Ablaufsteuerung der geringfügige Umbau der erstellten Zwischenstruktur. Für den Umbau von Aktivitäten ist die Klasse `ActivityReconstructor` zuständig, deren Funktionalität im Folgenden erläutert wird. Da sämtliche Schritte in dieser Klasse enthalten sind, wird bewusst auf die Darstellung eines Klassendiagramms verzichtet.

Zieldatentyp in Parametern setzen

In einem ersten Umbauschritt werden sämtliche Parameter der Knoten und Aktivitäten durchlaufen und die Zieldatentypen der Parameter gesetzt. Dies geschieht in der Methode `setParameterTargetClassName`. Hierbei werden die entsprechenden Klassen aus dem zentralen Speicher abgefragt und deren vollständiger Zielklassenname im Parameter übernommen. Dies ist insofern notwendig, als die Information über diesen Datentyp in den `Entity`-Klassen der Zwischenstruktur der Klassendiagramme vorhanden ist und in den Aktivitäten nicht als bekannt vorausgesetzt werden kann.

Erzeugen von Knopfbezeichnungen

Als Nächstes werden die externen `ResourceBundle`-Zeichenketten erzeugt, die für die Aktivitäten und deren Bestandteile relevant sind. Dies geschieht durch den Aufruf der Methode `addMultiLanguageLangstring`. Darin werden zum einen die Aktivitätsnamen für die Bezeichnung der Links zum Fortsetzen von Aktivitäten gespeichert. Zum anderen werden auch die Aktionsbezeichnungen externalisiert, die auf den Knöpfen angezeigt werden, sofern ein Nutzer auswählen kann, welche Aktion als Nächstes auszuführen ist (resultierend aus der Verwendung von Entscheidungsknoten ohne Bedingungen).

Parameter von Kontrollknoten komplettieren

Schließlich werden die Kontrollknoten einer jeden Aktivität bearbeitet. Dies geschieht jeweils in den Methoden `reconstructDecisionNode`, `reconstructJoinNode` usw. In diesen Methoden werden den Kontrollknoten Parameter hinzugefügt, die sich aus den internen Zuweisungen von Parametern ableiten. Diese können entweder automatisch beim Aufbau der Zwischenstruktur hinzugefügt oder aber explizit in einer Knotendeklaration im Modell angegeben worden sein.

8.4.5. Generator

Die Zwischenstruktur, die in den vorherigen Schritten aufgebaut bzw. angereichert und umgebaut wurde, dient als Eingabedatenstruktur für die verschiedenen Templates, die letztlich den Code für die Ablaufsteuerung des Web-Informationssystems in MontiWIS erzeugen. Wie üblich werden dazu Freemarker-Templates verwendet, über die die Inhalte der Zwischenstruktur verarbeitet und in lauffähigen Code übersetzt werden. Die verwendeten Templates sind in Tabelle 8.38 inklusive einer kurzen Erläuterung gelistet.

Template-Datei	Erläuterung
`a.../java/decisionnode.ftl`	Erzeugt eine Klasse für jeden Entscheidungsknoten einer Aktivität
`a.../java/finalnode.ftl`	Erzeugt eine Klasse für jeden Aktivitätsendknoten einer Aktivität
`a.../java/flowfinalnode.ftl`	Erzeugt eine Klasse für jeden Ablaufendknoten einer Aktivität
`a.../java/forknode.ftl`	Erzeugt eine Klasse für jeden Parallelisierungsknoten einer Aktivität
`a.../java/initialnode.ftl`	Erzeugt eine Klasse für jeden Startknoten einer Aktivität
`a.../java/joinnode.ftl`	Erzeugt eine Klasse für jeden Synchronisationsknoten einer Aktivität

`a.../java/mergenode.ftl`	Erzeugt eine Klasse für jeden Verbindungsknoten einer Aktivität
`a.../java/parameternode.ftl`	Erzeugt eine Klasse für jeden Parameterknoten einer Aktivität
`c.../java/action.ftl`	Erzeugt eine Klasse für jede Aktion einer Aktivität
`c.../java/ActionResource.java.ftl`	Erzeugt die Ressourcenklasse zur Verarbeitung von Anfragen an eine Aktion
`c.../java/ActivitiesResource.java.ftl`	Erzeugt die Ressourcenklasse zur Verarbeitung von Anfragen an Aktivitäten
`c.../java/activity.ftl`	Erzeugt eine Klasse für jede Aktivität
`c.../java/ActivityFactory.java.ftl`	Erzeugt die Klasse zur Instanziierung von Aktivitäten nach ihren Namen
`c.../java/ActivityResource.java.ftl`	Erzeugt die Ressourcenklasse zur für Anfragen an eine bestimmte Aktivität
`c.../jsp/ActivityFinished.jsp.ftl`	Erzeugt die JSP-Seite für abgeschlossene Workflows
`c.../jsp/InvalidState.jsp.ftl`	Erzeugt die JSP-Seite für ungültige Zustände eines Workflows
`c.../jsp/RunningActivities.jsp.ftl`	Erzeugt die Seite zur Anzeige von laufenden Workflows

Tabelle 8.38.: Die Template-Dateien für Aktivitäten und ihre Bedeutung beim Generierungsprozess

Aufgrund des begrenzten Platzes innerhalb der Tabelle werden die Stellen der Templates im Quellcodebaum abgekürzt. Pfade, die mit `a.../` beginnen, beziehen sich auf das Verzeichnis `de/montiwis/ad/generator/templates/` wohingegen diejenigen, deren Anfang `c.../` lautet, im Verzeichnis `de/montiwis/combined/generator/templates` zu finden sind. Für die Verarbeitung der Templates des ersteren Ortes zeichnet sich die Klasse `de.montiwis.ad.generator.WorkflowGenerator` verantwortlich, letztere Templates werden verarbeitet durch die Klasse `de.montiwis.combined.generator.CombinedGenerator`, die neben diesen Templates beispielsweise auch solche für die Generierung von Seiten aufruft.

Die Aufteilung der Templates in diese verschiedenen Paketstrukturen lässt sich durch die unterschiedlichen Inhalte der Zwischenstruktur erklären. Die Templates, die im `de/montiwis/ad` Paket liegen, benötigen lediglich diejenigen Informationen, die direkt aus den Aktivitätsmodellen gewonnen werden. Für die Templates, die in der Paketstruktur unter `de/montiwis/-`

`combined` zu finden sind, werden zusätzlich Informationen aus anderen Modellen benötigt. Beispielsweise werden darin Seiten referenziert, so dass deren Namen und Parameter aus dem `PageGlobalStore` (vgl. Abschnitt 7.3.2) benötigt werden.

In den nachfolgenden Abschnitten werden die einzelnen Schritte bei der Generierung kurz beschrieben. Die eigentliche Funktionalität erschließt sich bei vielen der generierten Klassen jedoch erst im Kontext des generierten Systems und dessen Aufbau, weswegen für eine Einführung auf Abschnitt 5.4 und für eine detailliertere Beschreibung dessen auf Abschnitt 8.5 verwiesen wird.

Generierung von Klassen für Kontrollknoten

Die Templates, die im Verzeichnis `de/montiwis/ad/generator/templates/java` abgelegt sind, werden für die Generierung eines überwiegenden Teils der Kontrollknotenklassen verwendet.

Die Klassen werden in der Regel im Paket `control.nodetype` abgelegt, wobei `nodetype` der Knotentyp ist. Dies ist `decisionnodes` im Fall von Entscheidungsknoten, `finalnodes` bzw. `flowfinalnodes` im Fall von Aktivitäts- oder Ablaufendknoten, `intialnodes` bei Startknoten, `forknodes` bei Parallelisierungsknoten, `mergenodes` bei Verbindungsknoten, `joinnodes` im Fall von Synchronisationsknoten und `parameternodes` für Parameterknoten. Der Name der Klasse setzt sich standardmäßig zusammen aus dem Namen der Elternaktivität, dem Namen des Kontrollknotens sowie der Art des Kontrollknotens als Suffix. Der Name des Kontrollknotens kann entweder explizit angegeben oder aber bei der Bearbeitung durch einen Visitor generiert worden sein. Als Beispiel soll hier angenommen werden, dass eine Aktivität `GradeThesis` einen Entscheidungsknoten mit Namen `D` enthält. Daraus würde eine Klasse mit Namen `GradeThesis_DDecision` generiert werden, die im Paket `control.decisionnodes` abgelegt wird.

Der Inhalt der Klassen besteht bei Start- und beiden Arten von Endknoten lediglich aus der Logik, die von den Elternklassen vererbt wird. Die übrigen Klassen enthalten, sofern zutreffend, die interne Zuordnung von Eingangs- zu Ausgangsparametern. Bei Entscheidungsknoten kommt weiterhin Funktionalität zur Auswertung von potenziell vorhandenen Bedingungen hinzu.

Generierung von Aktionsklassen

Umfangreicheren Inhalt enthalten diejenigen Klassen, die aus dem Template `action.ftl` im `combined` Verzeichnis generiert werden. Die generierten Klassen werden im Verzeichnis `control/actions` abgelegt und ihr Name setzt sich ebenfalls aus dem Namen der Elternaktivität, dem Aktionsnamen sowie dem Suffix `Action` zusammen.

Als Inhalt wird für jeden Parameter und jede Variable die Logik generiert, durch die die Werte gesetzt und ausgelesen werden können. Ferner werden die Inhalte in der Klasse verankert, die die eigentliche Geschäftslogik repräsentieren. Hierzu werden zweierlei Methoden generiert, so dass die Logik sowohl vor dem Aufruf einer Seite als auch danach ausgeführt werden kann. Die Logik selbst setzt sich aus den Java-Anteilen sowie den in Java-Ausdrücke übersetzten Kommandos zusammen.

Generierung von Klassen für jede Aktivität

Für jede Aktivität im Modell wird eine Klasse erzeugt, die die Struktur und Funktionalität der Aktivität im generierten System festlegt. Bei der Generierung wird das Template `activity.ftl` für jedes `Activity`-Objekt der Zwischenstruktur aufgerufen. Diesem Template werden nicht nur das zugehörige Objekt der Zwischenstruktur selbst, sondern auch sämtliche darin enthaltenen Knoten und Kanten übergeben, da dies eine einfachere Bearbeitung ermöglicht.

Im Template selbst werden sämtliche Knoten durchlaufen und daraus entsprechende Objekte sowie deren Instanziierung generiert. Ferner wird über jede Kante (`Edge` in der Zwischenstruktur) iteriert und davon ein Äquivalent in der generierten Klasse angelegt.

Ähnlich den Aktionen existieren auch in Aktivitäten Methoden zum Setzen und Auslesen von Parametern. Deren Inhalt wird entsprechend vorhandener Parameterknoten mit Inhalt gefüllt, so dass auf potenziell vorhandene Objekte zugegriffen werden kann.

Generierung von Ressourcenklassen und Fabriken

Wie in Tabelle 8.38 zu sehen, existieren auch Templates für die Generierung von Ressourcenklassen. Dies sind im Einzelnen `ActionResource.java.ftl`, `ActivitiesResource.java.ftl` sowie `ActivityResource.java.ftl`, die allesamt im `combined` Zweig des Quellcodes abgelegt sind. Diese Templates besitzen lediglich marginal dynamische Teile (wie beispielsweise den Import von potenziell variierenden Aktivitäts- bzw. Aktionsklassen), welche ebenfalls durch sinnvolles Standardverhalten bei der entstehenden Projektstruktur eliminiert werden könnten. In solch einem Fall wären die erzeugten Klassen vollständig statisch und könnten an eine entsprechende Stelle im generierten System kopiert werden. Da die generierten Klassen für die Verarbeitung per Jersey-Schnittstelle jedoch in einem speziellen Verzeichnis liegen müssen, um erkannt und verarbeitet zu werden, wurde an dieser Stelle entschieden, die Klassen dorthin zu generieren anstatt sie zu kopieren.

Etwas anders verhält es sich mit der Generierung der Aktivitätsfabrik, die über das Template `ActivityFactory.java.ftl` erzeugt wird. Für ihre Funktion sind primär sämtliche Aktivitäten bzw. deren Namen notwendig, weswegen dem Template sämtliche `Activity`-Objekte aus der Zwischenstruktur übergeben werden. Darin wird darüber iteriert und für jede einzelne Aktivität der entsprechende Code zur Instanziierung eines Objekts erzeugt.

Generierung von JSP-Seiten für die Ausführung von Aktivitäten

Schließlich werden im Kontext der Ablaufsteuerung einige JSP-Seiten generiert. Ähnlich der Ressourcenklassen sind auch hier die dynamischen, vom Modell abhängigen Anteile der Seite eher rudimentär. Der Einheitlichkeit halber und aufgrund der Tatsache, dass die erzeugten Seiten an den gleichen Stellen wie die mit dynamischen Anteilen generierten Seiten liegen müssen, wurde auch in diesem Fall entschieden, dass der Aufruf eines Templates einfachem Kopieren vorzuziehen sind.

Die Templates, die hierbei zum Einsatz kommen, sind `ActivityFinished.jsp.ftl`, `InvalidState.jsp.ftl` sowie `RunningActivities.jsp.ftl`. Die ersten beiden Seiten enthalten

neben Verweisen auf die Templates für einen einheitlichen Kopf- und Fußteil (vgl. hierzu Abschnitt 7.3.5 sowie Tabelle 7.36) tatsächlich lediglich Referenzen auf Nachrichten, die per `ResourceBundle`-Datei ausgetauscht werden können. Letzteres Template enthält Anteile, die in der generierten Seite per Javascript-Aufruf während ihrer Anzeige ersetzt werden und durch Links zum Fortführen eines Workflows angereichert werden.

8.5. Umsetzung im Zielsystem

Die Klassen, die als Resultat des in Abschnitt 8.4.5 beschriebenen Generierungsprozess erzeugt wurden, sind als in Kombination mit den Klassen aus der Laufzeitumgebung zu sehen und auch nur in deren Kontext funktionsfähig. Ihre Funktionalität wird im folgenden Abschnitt detaillierter beschrieben.

Die Klassen, die zur Laufzeitumgebung gehören, sind im Projekt `montiwis-target` enthalten. Dessen Inhalt wird zur Ausführung benötigt und muss folglich als Abhängigkeit (beispielsweise als jar-Archiv) im generierten Projekt vorhanden sein.

Im Folgenden werden zunächst in Abschnitt 8.5.1 überblicksartig die einzelnen Komponenten der Laufzeitumgebung dargestellt. Abschnitt 8.5.2 beschreibt, wie mit den Aktivitäten per REST-Ressource interagiert wird und wie sie weiter ausgeführt werden und Abschnitt 8.5.3 schließlich erläutert, welche weiteren Klassen zur Unterstützung an der Ausführung von Aktivitäten beteiligt sind.

8.5.1. Übersicht über die Laufzeitumgebung

Für ein grundlegendes Verständnis der Funktionsweise der Laufzeitumgebung zur Ausführung von Abläufen und Geschäftslogik wird in diesem Abschnitt auf die zentralen Komponenten und deren Zusammenspiel eingegangen. Ein kurzer Überblick darüber wurde bereits in Abschnitt 5.4 gegeben. Nachfolgend werden nun Details der Ausführungsschicht erläutert. Dafür zeigt Abbildung 8.39 einen Überblick über die verschiedenen Teilkomponenten der Laufzeitumgebung und ihren Zusammenhang.

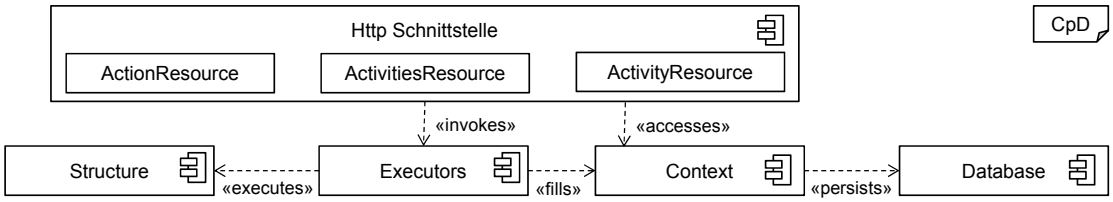


Abbildung 8.39.: Komponentenstruktur der Aktivitätsausführungsschicht

Wie dort zu sehen besteht die Laufzeitumgebung aus verschiedenen Teilen, die gemeinsam zur Ausführung von Abläufen benötigt werden. Oben in der Abbildung ist die *Http Schnittstelle* dargestellt, über die die Interaktion mit den Komponenten der Ablaufsteuerung stattfindet. Zu ihr gehören die Klassen `ActivityResource`, `ActionResource` und `ActivitiesResource`, die, wie in Abschnitt 8.4.5 erläutert, bei der Generierung des Web-Informationssystems mit

erzeugt werden. Von diesen Klassen aus werden primär die `Executors` sowie bedingt die `Contexts` verwendet, um die Steuerung des Systems anzustoßen. Unter `Structure` sind die Klassen zusammengefasst, deren Generierung in Abschnitt 8.4.5 erläutert wurde und die im Wesentlichen die Struktur der Aktivitäten repräsentieren. Unter `Context` werden die Klassen verstanden, die den Zustand der Ausführung einer Aktivität repräsentieren und insbesondere auch die Werte von Variablen enthalten. Durch die `Executors` werden die Strukturelemente und der Kontext zusammengeführt und zur Ausführung gebracht. Schließlich steht die Komponente `Database` für den Datenbankzugriff, über den der Kontext gespeichert und zu einem späteren Zeitpunkt wieder geladen werden kann.

Die einzelnen Teile und deren Zusammenspiel werden im Folgenden genauer erläutert.

Knotenstruktur

Unter *Knotenstruktur* werden die Klassen verstanden, die die Struktur und den Inhalt der Modellelemente widerspiegeln. Die Abbildung der Modellinformationen erfolgt durch die generierten Klassen aus Abschnitt 8.4.5, die einen Teil ihrer Funktionalität durch Vererbung der Klassen aus Abbildung 8.40 erhalten. Die Implementierung der Logik und der Strukturinformationen wird durch Kombination dieser Klassen und durch Erweiterung an dezidierten Stellen umgesetzt.

Die in der Übersicht dargestellten Schnittstellen befinden sich im Paket `de.montiwis.workflowengine.elements`, die diese Schnittstellen implementierenden Klassen sind im Paket `de.montiwis.workflowengine.elements.impl` zu finden. Die `Node`-Schnittstelle wird erweitert durch die Schnittstellen, die die einzelnen Knotenarten innerhalb einer Aktivität beschreiben. Sie spezifiziert lediglich die von allen Knoten benötigten Funktionen wie den Zugriff auf Name (`getName` bzw. `setName`) und Elternaktivität (`getParent` bzw. `setParent`) des jeweiligen Objekts. Diese Funktionalität wird in der Klasse `AbstractNode` implementiert. Davon werden ihrerseits wieder die Klassen abgeleitet, die lediglich diese Attribute benötigen, welches `InitialNode`, `FinalNode` und `FlowFinalNode` sind.

Für Knoten, die Eingangs- und Ausgangsparameter besitzen, existiert die Schnittstelle `ParametrizedElement`. Diese beinhaltet Methoden zum Setzen und Auslesen von Parametern (stellvertretend für beide Arten von Parametern `addInputParameter` zur Definition eines Parameters und `set-` bzw. `getInputParameterValue` zum Setzen bzw. Abrufen des konkreten Wertes). Die Implementierung eines Großteils dieser Funktionalität erfolgt in der Klasse `AbstractParametrizedElement`. Diese Klasse wiederum wird von den Klassen `ActionImpl`, `ForkNodeImpl`, `DecisionNodeImpl`, `JoinNodeImpl`, `MergeNodeImpl`, `ParameterNodeImpl` sowie `ActivityImpl` erweitert. In `AbstractParametrizedElement` existiert für Eingangs- und Ausgangsparameter je eine `Map` für die Zuordnung von Parameternamen zu Klasse (`Class<?>`) für die Überprüfung des korrekten Typs und eine `Map` für die Zuordnung vom Namen des Parameters zum eigentlichen `Object`, also dem Wert des Parameters. Wie der Namensbestandteil `Abstract` andeutet, werden einige Methoden nicht implementiert, sondern als `abstract` zur Implementierung der Kindklassen markiert. Auch die eben genannten `Impl`-Klassen, die zwar die generell notwendige Funktionalität bereitstellen, reichen diese Methoden an deren zu generierenden Kindklassen weiter. Dies sind insbesondere die Funk-

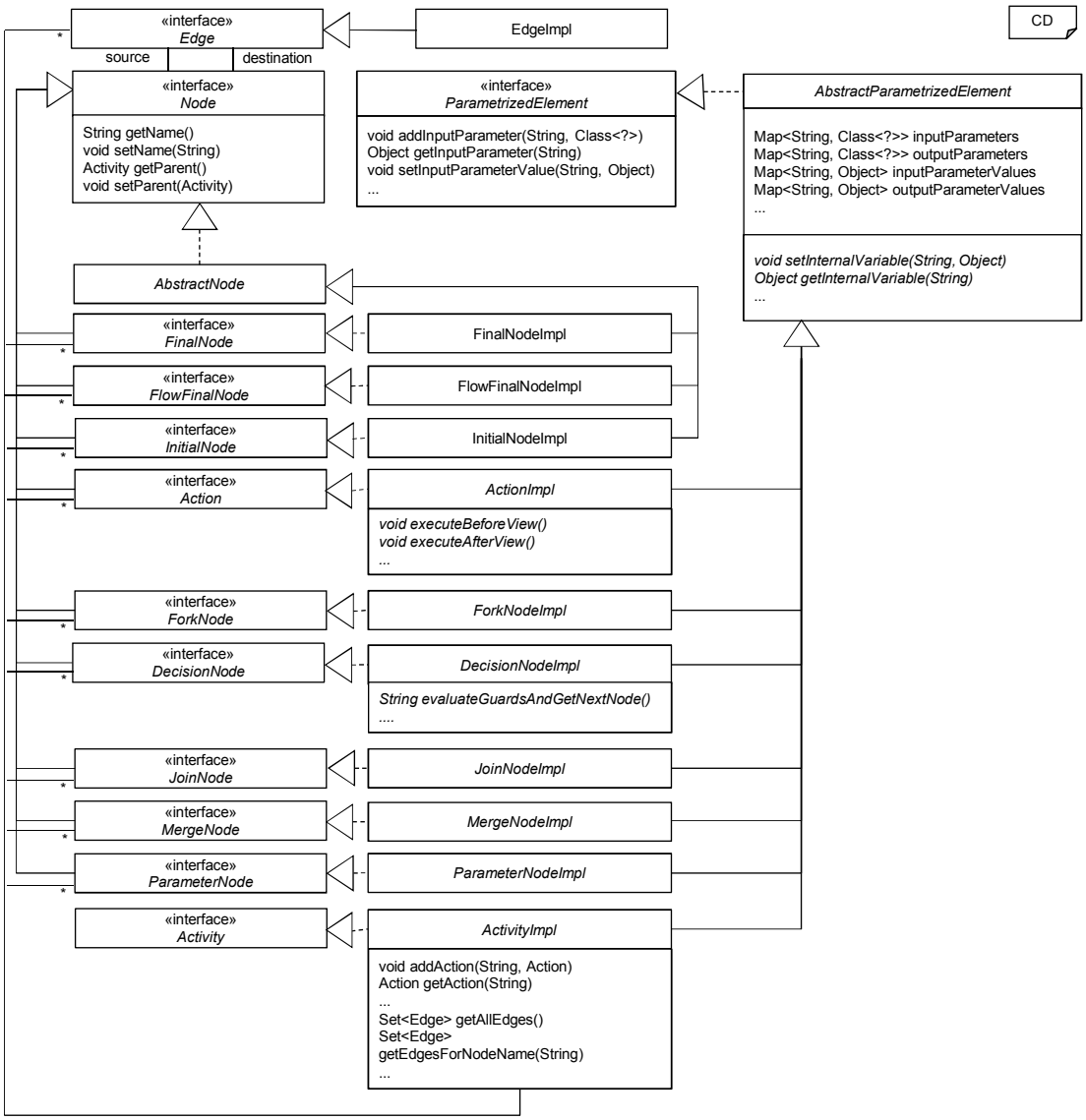


Abbildung 8.40.: Klassen zur Repräsentation von Aktivitäten in der Laufzeitumgebung

tionen `setInternalVariable` für das Setzen eines Feldes und `getInternalVariable`, die ein `Object` zurückliefert. An diese Methoden werden Aufrufe zum Setzen oder Abrufen eines Parameters mit bestimmtem Namen delegiert. Durch den Generator wird die konkrete Logik dazu erzeugt. Beispielhaft ist dies in Abbildung 8.41 anhand eines Ausschnitts zur generierten Klasse der Aktion `SetGrade1` aus Abbildung 8.6 dargestellt.

Wie dort zu sehen ist, existieren in der Klasse die jeweiligen Parameter und Variablen als Klassenvariablen. Dies ist sinnvoll, damit insbesondere in Aktionen direkt ohne größere Umbauten des im Modell spezifizierten Java-Codes direkt auf die Objekte zugegriffen werden kann. Im

Java

```
1  public class GradeThesis_SetGrade1Action extends ActionImpl {
2
3    private Thesis o;
4    private Thesis i;
5
6    public GradeThesis_SetGrade1Action(String name) {
7      super(name, false);
8      this.addOutputParameter("o", Thesis.class);
9      this.addInputParameter("i", Thesis.class);
10   }
11
12   @Override
13   protected void setInternalVariable(String parameterName, Object value) {
14     if (parameterName.equals("o")) {
15       this.o = ((Thesis)value);
16     }
17     if (parameterName.equals("i")) {
18       this.i = ((Thesis)value);
19     }
20   }
21
22   @Override
23   protected Object getInternalVariable(String parameterName) {
24     Object ret = null;
25     if (parameterName.equals("o")) {
26       return this.o;
27     }
28     if (parameterName.equals("i")) {
29       return this.i;
30     }
31     return ret;
32   }
33 }
```

Abbildung 8.41.: Setzen und Abrufen von Parametern in Knoten

Konstruktor wird der Typ des Parameters zur Kontrolle in der oben beschriebenen `Map` gesetzt, damit die Typkompatibilität beim Aufruf (in der Klasse `AbstractParametrizedElement`) überprüft werden kann. In der Implementierung der abstrakten Funktion `setInternalVariable` wird auf Basis des Namens das entsprechende Feld in der Klasse gesetzt (Zeilen 13-20) bzw. in der Funktion `getInternalVariable` das korrekte Objekt zurückgeliefert (Zeilen 23-32). Der Umweg über diese Methoden ist einerseits aus dem Grund sinnvoll, dass die Variablen direkt als Felder verfügbar sind, andererseits erlauben sie aber auch den Zugriff auf die Parameter sowohl beim Weiterschalten des Ablaufs als auch bei der Speicherung der Werte und der späteren Fortsetzung der Aktivität. Dies gilt neben den Eingangs- und Ausgangsparametern im Fall von Aktionen auch für Variablenwerte, die in der Regel auch mit

dem Zustand einer Aktion im zugehörigen `Context`-Objekt gespeichert werden müssen.

Neben Variablen enthalten Aktionen weitere Funktionen, die zur Erfüllung ihrer Aufgaben notwendig sind. Zum Großteil existieren die Funktionen bereits in den `-Impl`-Klassen und werden lediglich durch die Belegung von Variablen in den generierten Klassen mit Funktionalität hinterlegt. So werden, wie auch schon in Abbildung 8.41 dargestellt, Parameter oder auch Seitennamen im Konstruktor der generierten Klassen gesetzt und können über bereits existierende Methoden abgerufen werden. Zusätzlich wird der Inhalt der Aktionen aus den Modellen bei der Generierung in die Funktionen `executeBeforeView` und `executeAfterView` übernommen, die letztlich den Java-Code enthalten, der in den Modellen als Java spezifiziert wurde sowie den Code, der aus den Kommandos erzeugt wurde.

Neben den Aktionen besitzen auch Entscheidungsknoten (`DecisionNodes`) eine besondere Funktion zusätzlich zu den bereits beschriebenen Eingangs- und Ausgangsparametern und deren Verarbeitung. Hier werden, sofern vorhanden, die Bedingungen, die an der Kante und damit implizit am Entscheidungsknoten angetragen waren, übernommen. Dies geschieht in der Methode `evaluateGuardsAndGetNextNode`, in der die Werte (die ebenfalls als Feld in der Klasse vorhanden sind) hinsichtlich der gegebenen Bedingung evaluiert werden. Als Rückgabewert wird der Name des als Nächstes auszuführenden Knotens zurückgegeben, so dass dieser entsprechend aufgerufen werden kann.

Nach der Erläuterung des grundlegenden Aufbaus der verschiedenen Knotentypen und deren Besonderheiten wird nun der Aufbau von Aktivitäten über Kindklassen von `ActivityImpl` beschrieben. Diese Klasse besitzt zunächst einige `Map`-Attribute zur Speicherung sämtlicher Knoten, jeweils aufgeteilt nach Typen und abgespeichert unter ihrem Namen. Ferner existieren Methoden zum Hinzufügen und Abrufen von Knotenobjekten (stellvertretend dargestellt durch die Funktionen `addAction` bzw. `getAction`) zu diesen Strukturen. Der eigentliche Aufbau der Aktivität wird in der generierten Klasse im Konstruktor erzeugt. Hierbei werden sowohl die Knoten angelegt und unter ihrem Namen in besagten `Maps` abgelegt als auch die Kanten (`Edges`) erzeugt, die die Knoten verbinden. `EdgeImpl` Klassen enthalten sowohl Start- als auch Zielknoten sowie die Namen von potenziellen Eingangs- und Ausgangsparametern, die an der jeweiligen Kante beteiligt sind. Kanten werden ebenfalls in einer `Map` abgelegt, wobei als Schlüssel der Name des Ausgangsknotens dient und als Wert die Menge von Kanten abgelegt wird, die von diesem Knoten ausgehen. Diese können sowohl qualifiziert über Angabe des Ausgangsknotennamens (`getNodesForNodeName`) als auch in ihrer Gesamtheit gemeinsam abgerufen werden (`getAllEdges`).

Ausführungskontext

Neben der Struktur der Aktivitäten gehören zur Ausführungsschicht weiterhin sog. `Executors` und `Contexts`. Letztere werden im folgenden Abschnitt etwas näher beleuchtet. Eine Übersicht über die verschiedenen Arten von `Context` ist in Abbildung 8.42 dargestellt.

Die `Context`-Klassen befinden sich im Paket `de.montiwis.workflowengine.context`, orientieren sich am Kontext-Muster aus [ACM03] und dienen im Wesentlichen zum Auslagern und Speichern des Aktivitäts- bzw. Knotenzustands zu einem bestimmten Zeitpunkt. Für jeden Knoten wird ein `NodeExecutionContext` erzeugt. Darin gespeichert werden unter

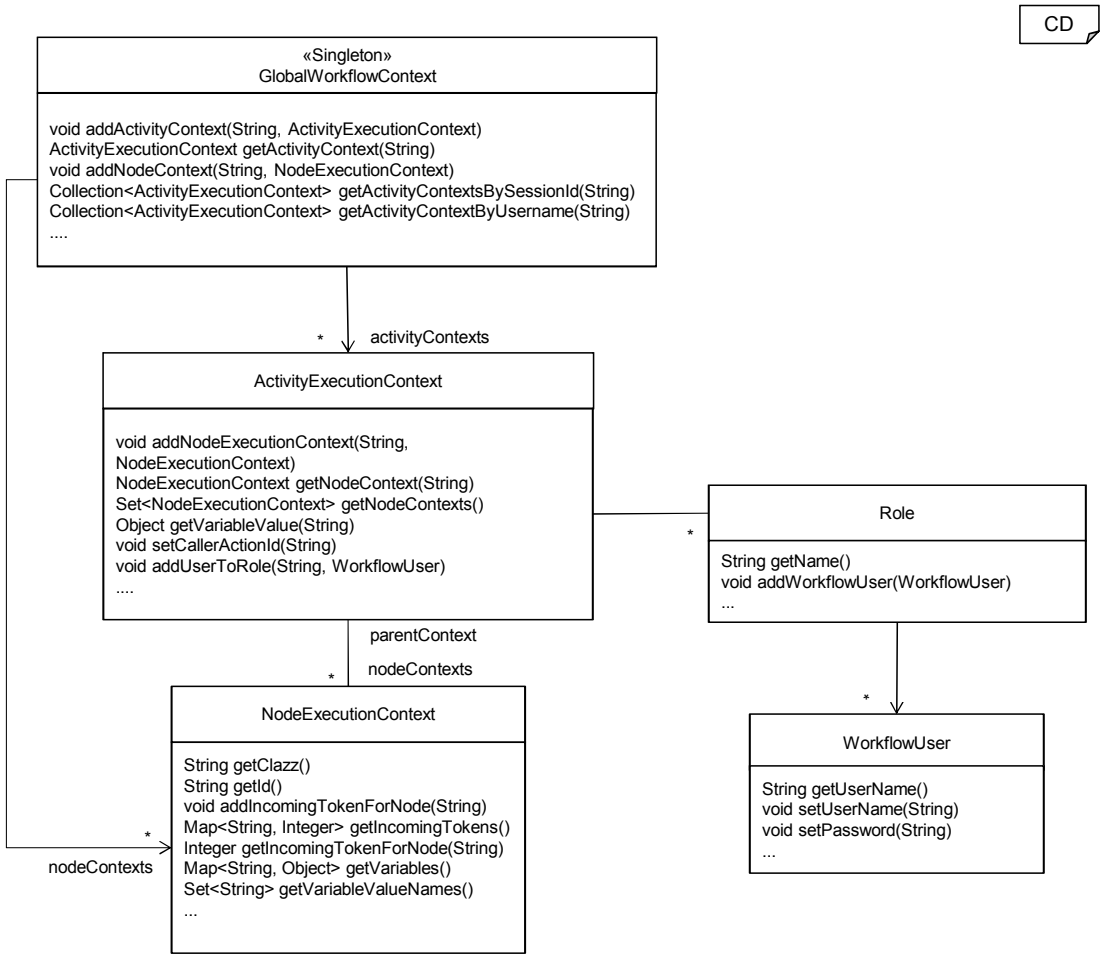


Abbildung 8.42.: Klassen zur Repräsentation von Ausführungskontext

anderem der Name des Knotens, die Klasse des Knotens, zu der der jeweilige Kontext den Zustand speichert und einen eindeutigen Bezeichner zur Identifikation des Kontextes, welcher per Instanziierung einer neuen UUID belegt wird (angedeutet über die Funktionen `getId` und `getClazz`). Ferner existiert eine `Map`, in der die Anzahl vorhandener Token[26] abgespeichert wird. Dort dient als Schlüssel der Name des vorhergehenden Knotens und als Wert ein `Integer`, der die Anzahl der Token repräsentiert. Hinzugefügt werden Token über die Methode `addIncomingTokenForNode`. Für den Zugriff auf die Anzahl der Tokens sowie deren Manipulation existieren verschiedene Methoden, die dies ermöglichen (stellvertretend dargestellt über die Funktionen `getIncomingTokensForNode` und `getIncomingTokens`). Ein elementarer Be-

[26] Bei der Ausführung orientiert sich MontiWIS im Wesentlichen an der Ausführungssemantik der UML Aktivitätsdiagramme, deren zentrales Konzept sogenannte Token sind, die den Bearbeitungszustand von Aktivitäten repräsentieren. Nähere Informationen dazu finden sich beispielsweise in [RQZ07, Obj11c]. In formaler Art und Weise wurden diese Konzepte beispielsweise in [GRR10] oder auch [Kni11, Stö04, VK05] behandelt.

standteil eines Kontextes sind die Werte von Parametern und Variablen. Diese werden ebenfalls in einer `Map` abgelegt, bei der der Name der Variablen oder des Parameters als Schlüssel dient, der zugehörige Wert wird in Form eines allgemeinen `Object` darin gespeichert. Der Zugriff darauf erfolgt über Funktionen wie `getVariables` oder `getVariableValueNames` zum Abfragen der Variablen- bzw. Parameternamen. Zusätzlich wird in einem `NodeExecutionContext` eine Referenz auf den übergeordneten `ActivityExecutionContext` abgelegt sowie, soweit zutreffend, der eindeutige Bezeichner des Kontextes der aufgerufenen Aktivität. Dies ist jedoch nur von Belang, sofern es sich bei dem zugehörigen Knoten um eine Aktion handelt, deren Inhalt in einer anderen Aktivität spezifiziert (vgl. hierzu Abschnitt 8.3.6).

Die Klasse `ActivityExecutionContext` dient zur Speicherung des Zustands einer gesamten Aktivität. Auch in dieser Klasse werden sowohl der Name der zugehörigen Klasse, ein eindeutiger Bezeichner (ebenfalls per UUID belegt, worüber über Funktionen wie `getNodeContext` bzw. `addNodeExecutionContext` zugegriffen wird) sowie ebenfalls die Variablen, die potenziell einer Aktivität bei ihrem Aufruf übergeben worden sein können (über Funktionen wie `getVariableValue` abrufbar). Letztere werden an die entsprechenden Parameterknoten weitergereicht und im Verlauf der Verarbeitung der Aktivität entlang des Ablaufs weitergegeben. Außerdem besteht die Möglichkeit, den Bezeichner der aufrufenden Aktion zu speichern (über die Funktion `setCallerActionId`). Überwiegend wird der Zustand einer Aktivität jedoch über den Zustand der einzelnen darin enthaltenen Knoten determiniert. Dementsprechend existieren in einem `ActivityExecutionContext` Referenzen zu den `NodeExecutionContext`-Klassen der darin enthaltenen Knoten. Diese werden in einer `Map` abgelegt, wobei der Name des Knotens als Schlüssel verwendet wird.

In der Klasse `ActivityExecutionContext` existiert weiterhin eine `Map` zur Speicherung einer Menge von `Role`-Objekten, die zur Zuordnung von bestimmten `WorkflowUsern` (bzw. die davon abgeleiteten Klassen, vgl. Abschnitt 6.2.4 bzw. Abbildung 6.10) zu expliziten Aktivitätsbereichen gedacht sind. `Role`-Objekte werden mit dem Namen des Aktivitätsbereichs als Schlüssel abgelegt. In der `Role`-Klasse wird neben dem Namen der Rolle (abrufbar über `getName`) die Menge von konkreten `WorkflowUser`-Objekten abgelegt (über die Methode `addWorkflowUser`), die zur Ausführung eines bestimmten Aktivitätsbereichs berechtigt sind. Auch `Role` und `WorkflowUser` Klassen besitzen einen eindeutigen UUID Bezeichner, eine `Role` zusätzlich eine Referenz auf den zugeordneten `ActivityExecutionContext`. Ein `WorkflowUser` besitzt neben Nutzernamen und Passwort ein Feld für eine Salt-Zeichenkette, die jedoch aktuell aus pragmatischen Gründen per Konstante gesetzt ist (vgl. hierzu Abschnitt 7.4.5).

Als letzte Kontextklasse ist der `GlobalWorkflowContext` zu erwähnen. Hierbei handelt es sich um ein EJB-Singleton, welches per `@Singleton` annotiert ist und damit innerhalb des Applikationskontextes nur einmal vorkommen kann. Diese Klasse besitzt eine `Map` zur Ablage von `NodeExecutionContext`-Objekten mit ihrem Bezeichner als Schlüssel sowie zwei `Maps` zur Ablage von `ActivityExecutionContext`-Objekten, einmal per eindeutigem Bezeichner des Kontextes selbst und einmal per eindeutigem Bezeichner der Serversession, in der diese Aktivität ausgeführt wird. Dies rührt daher, dass Abläufe, die von nicht registrierten bzw. eingeloggten Nutzern ausgeführt werden, ebenfalls gespeichert werden müssen, was jedoch nicht persistent in der Datenbank geschehen sollte. Die `Context`-Objekte lassen sich über die Funk-

tionen addNodeContext bzw. addActivityContext hinzufügen und abrufen über verschiedene Funktionen, denen entweder die id des Contexts übergeben wird (getActivityContext) oder aber der Sessionbezeichner (getActivityContextsBySessionId) oder Nutzername (getActivityContextsByUserName). Neben den genannten Speichern und Methoden zum Zugriff darauf existieren weiterhin Referenzen auf die DAO-Klassen von NodeExecutionContext und ActivityExecutionContext, so dass die zwischengespeicherten Kontexte auch persistiert bzw. wieder geladen werden können. Letzteres geschieht automatisch im Fall, dass der angeforderte Kontext nicht bereits in den Maps des GlobalExecutionContext vorhanden ist.

Bis auf den GlobalWorkflowContext sind all die Kontextklassen per @Entity annotiert und lassen sich damit über die üblichen JPA-Mechanismen speichern, laden und aktualisieren. Hierfür existieren sowohl für ActivityExecutionContext mit der Klasse ActivityExecutionContextDao und für die Klasse NodeExecutionContext mit der Klasse NodeExecutionContextDao Klassen, die Methoden für diese Aktionen anbietet.

Executors

Neben der Struktur der Aktivitäten und dem dazugehörigen Laufzeitzustand, der in Kontexten gespeichert wird, werden weiterhin Komponenten benötigt, die letztlich die Ausführung steuern und als Schnittstelle zwischen den Strukturklassen und den Zustandsinformationen stehen. Dies ist in Abbildung 8.39 überblicksartig dargestellt und wird in Abbildung 8.43 detaillierter gezeigt.

Auf der linken Seite der Abbildung sind die Schnittstellen dargestellt, wie sich auch bereits in Abbildung 8.40 gezeigt wurden und welche die Struktur einer Aktivität über verschiedene Knotentypen repräsentieren.

Zu jeder dieser Schnittstellen existiert eine Executor-Klasse, wie sie in der mittleren Spalte der Abbildung dargestellt sind. Diese Klassen sind alle von der abstrakten Klasse NodeExecutor abgeleitet. Sie ist Teil des *Observer*-Musters [GHJV95] und erweitert die Klasse Obervable, wodurch sie bei Observer-Klassen registriert werden kann. Diese Klassen bzw. Schnittstellen sind Teil des Java Development Kit (JDK) und stellen die Standardimplementierung dieses Musters in Java dar. Die NodeExecutor-Klasse besitzt bereits Referenzen zur zugehörigen Node sowie zum zugehörigen NodeExecutionContext und dem übergeordneten ActivityExecutor. Generell stellt die Klasse Methoden bereit, um die Ausführung von Aktivitäten und Knoten zu übernehmen. Dazu zählt beispielsweise die Manipulation von vorhandenen Tokens an Kanten oder auch das Setzen und Auslesen von Variablen sowohl in den Strukturknoten als auch den zugehörigen Kontexten. Ferner wird Logik bereitgestellt, um potenziellen Inhalt von Knoten auszuführen und gemäß Observer-Muster die Beendigung der Ausführung bekannt zu geben. Da es sich bei der Klasse NodeExecutor um eine abstrakte handelt, existiert natürlich auch eine solche Methode. Dies ist die boolesche Funktion canExecute, die je nach Knotentyp unterschiedlich implementiert bzw. durch einen Generator mit Logik gefüllt werden muss.

Nachdem NodeExecutor-Klassen für die Ausführung eines einzelnen Knotens zuständig sind, wird deren übergeordnete Steuerung durch die Klasse ActivityExecutor übernommen.

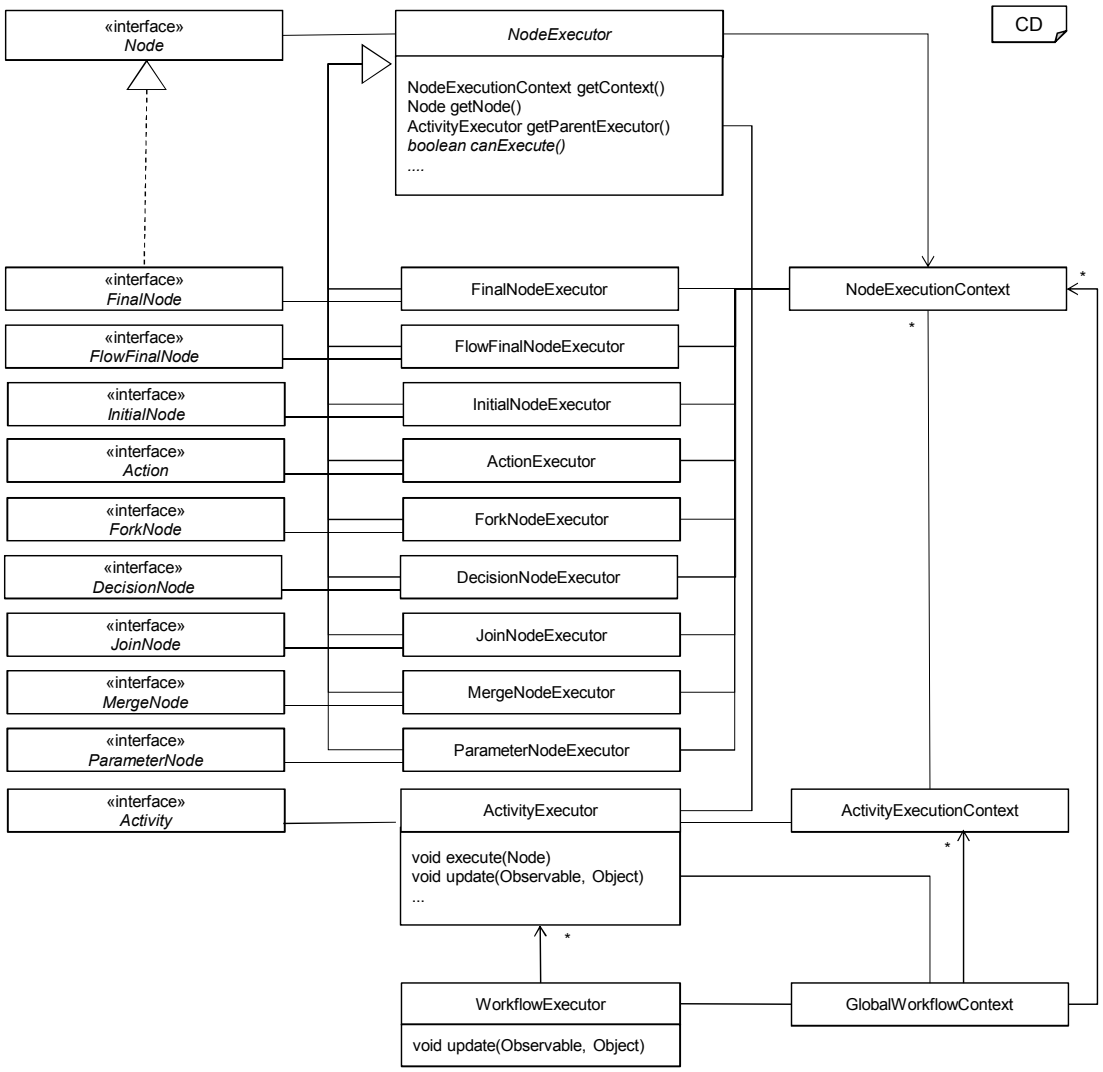


Abbildung 8.43.: Überblick über Klassen von Aktivitätsstruktur, ihrem Kontext und ihren Executor-Klassen

Auch diese Klasse besitzt Referenzen auf eine `Activity` als zugehörigen Knoten- bzw. Strukturtyp sowie auf dessen Laufzeitzustandsinformationen in Form des `ActivityExecutionContexts`. Ferner werden ebenfalls die `NodeExecutor`-Objekte der zur Aktivität gehörenden Knoten vorgehalten. Auf den `GlobalExecutionContext` wird zum Speichern und Abrufen der verwendeten Kontexte zugegriffen. In der Klasse `ActivityExecutor` ist in verschiedenen Funktionen die Logik implementiert, die die Ausführung einer Aktivität umsetzt. Für die Kommunikation zwischen den `Executor`-Klassen der enthaltenen Knoten dient auch hier das Observer-Muster, wobei ein `ActivityExecutor` sowohl die `Observable`-Klasse erweitert als auch die `Observer`-Schnittstelle implementiert. Ersteres ist notwendig um selbst wieder bei

einem `Observer` registriert werden zu können. In diesem Fall ist dies die `WorkflowExecutor`-Klasse, die bei Änderungen benachrichtigt wird. Letzteres wird nötig, da die verschiedenen `NodeExecutor`-Klassen bei einem `ActivityExecutor` registriert werden und diesen bei Abschluss der Ausführung benachrichtigen.

Schließlich werden die verschiedenen `ActivityExecutor`-Objekte über die Klasse `WorkflowExecutor` gesteuert. Wie bereits erwähnt, implementiert diese Klasse die Schnittstelle `Observer`, womit `ActivityExecutor`-Objekte bei ihr registriert werden können. Innerhalb dieser Klasse werden Referenzen zu den registrierten `ActivityExecutor`-Objekten und zum `GlobalWorkflowContext` gehalten. Der überwiegende Teil der Funktionalität dieser Klasse ist in der `update`-Funktion implementiert, welche bei Benachrichtigung von `ActivityExecutor`-Objekten ausgeführt wird und primär zum Starten neuer Aktivitäten als Teil von Aktionen (*CallBehaviour*) bzw. dem Fortführen von Aktionen nach der Beendigung von derartig gestarteten Aktivitäten dient.

8.5.2. Ausführung von Abläufen

Der generelle Aufbau und das Zusammenspiel der verschiedenen Elemente zur Beschreibung der Aktivitätsstrukur, den `Executor`-Klassen sowie den Laufzeitstatusinformationen wurde bereits im vorherigen Abschnitt beschrieben. Im folgenden Abschnitt wird detaillierter darauf eingegangen, wie die Ausführung vonstattengeht. Insbesondere werden auch die Mechanismen beschrieben, über die per HTTP-Schnittstelle mit den Abläufen interagiert werden kann.

Starten einer neuen Aktivität

Für einen Nutzer muss es möglich sein, eine Aktivität vom Webbrowser aus zu starten. Dazu wird in der Regel ein Link im Menü der Webanwendung geklickt (siehe hierzu Abschnitt 9.2.2). Die URL wird, wie in Abschnitt 5.3 erklärt, über die in Abbildung 8.44 dargestellte Ressourcenklasse verarbeitet.

Java

```
1  @Stateless
2  @Path("/Activity")
3  public class ActivityResource  {
4
5    @POST
6    @Path("{name}")
7    public Response startNewActivity(@PathParam("name") String name, @QueryParam("menuid") ↩
         String menuid) {
8      // Erstellen und Initialisieren des ActivityExecutors ae
9
10     if (IsUserInRolesTag.isUserInRoles(request, ae.getAllowedRolesString())) {
11       // starten der Aktivität und setzen notwendiger Parameter
12
13       if (ae.getChildActivityId() != null) {
14         URI target = new URI("/Activity/"+ae.getChildActivityId());
```

```
15          return Response.seeOther(target).contentLocation(target).build();
16        }
17
18        String nextActionId = ae.getNextActionId();
19        if (nextActionId == null) {
20          Viewable viewable = new Viewable("/ActivityFinished.jsp", null);
21          return Response.ok(viewable).build();
22        }
23
24        URI target = new URI("/Action/"+nextActionId+"?menuid="+menuid);
25        return Response.seeOther(target).contentLocation(target).build();
26      }
27      return Response.seeOther(new URI("/page/Index")).contentLocation(new URI("/page/Index↩
            ")).build();
28    }
29
30    @GET
31    @Path("{id}")
32    public Response continueActivity(@PathParam("id") String id, @QueryParam("menuid") ↩
          String menuid) {
33      // Erstellen und Initialisieren des ActivityExecutionContexts ac
34
35      if (ac != null) {
36        // Initialisieren des ActivityExecutors ae mit notwendigen Parametern
37        String nextActionId = ae.getNextActionId();
38
39        if (nextActionId == null) {
40          Viewable viewable = new Viewable("/ActivityFinished.jsp", null);
41          return Response.ok(viewable).build();
42        }
43
44        URI target = new URI("/Action/"+nextActionId+"?menuid="+menuid);
45        return Response.seeOther(target).contentLocation(target).build();
46      }
47      return Response.seeOther(new URI("/page/Index")).contentLocation(new URI("/page/Index↩
            ")).build();
48    }
49  }
```

Abbildung 8.44.: Ressourcenklasse zum Starten und Fortführen von Aktivitäten

Die Klasse `ActivityResource` besitzt eine Annotation `@Path("/Activity")`, wodurch ihre Methoden beim Aufruf einer URL der Form `resources/Activity` angesteuert werden. Das Starten einer neuen Aktivität geschieht über die Funktion `startNewActivity`, die per HTTP `POST` aufgerufen wird und als Pfaderweiterung den Namen der zu startenden Aktivität erwartet. Dieser Name entspricht dem Namen der Aktivitätsklasse (und damit in der Regel auch dem Namen der Aktivität auf Modellebene). Der `QueryParam` `menuid` dient lediglich zur Speicherung, welches Menü zum Zeitpunkt des Aufrufs gerade aktiv war, so dass dieses auch bei der Anzeige der nächsten Seite wieder geöffnet sein kann. Dies ist auch bei den im Folgenden erklärten Funktionen der Fall.

In der Funktion `startNewActivity` wird in einem ersten Schritt der `ActivityExecutor` erzeugt, der im weiteren Verlauf verwendet wird. Dies geschieht in der Klasse `WorkflowExecutor` unter Verwendung der generierten `ActivityFactory`, wo für jeden Namen ein Objekt der zugehörigen konkreten Klasse zurückgegeben wird. Dies sind die Schritte, die sich hinter dem Kommentar in Zeile 8 verbergen. Nachfolgend wird in Zeile 10 überprüft, ob der potenziell eingeloggte Nutzer (oder auch der nicht-eingeloggte Gastnutzer) die aktuelle Aktivität starten darf. Sollte dies der Fall sein, wird die Aktivität gestartet und einige Parameter wie der aktuelle Sessionbezeichner und Name des aktuellen Nutzers (sofern vorhanden) gesetzt, anderenfalls wird zur Hauptseite der Webapplikation umgeleitet (Zeile 27). Für den Fall, dass beim Start der Aktivität eine Aktion mit *CallBehaviourAction* als Inhalt ausgeführt wurde, wird per Umleitung die soeben gestartete Aktivität fortgesetzt. Dies geschieht per HTTP Status 303 (aufgerufen über die Funktion `seeOther`), wobei die URL mit dem eindeutigen Bezeichner der gestarteten Aktivität[27] aufgerufen wird (Zeilen 13-16). Durch diese Umleitung wird die Funktion `continueActivity` aufgerufen, die in einem späteren Unterabschnitt erläutert wird.

Im Fall, dass keine Kindaktivität gestartet wurde, wird der Bezeichner der als Nächstes anzuzeigenden Aktion ermittelt (Zeile 18). Sollte diese `null` sein, wird die `ActivityFinished.jsp`-Seite zurückgegeben, was immer der Fall ist, wenn keine nachfolgende Aktion ausgeführt werden kann. Diese Seite wird zusammen mit dem HTTP Status 200 (durch Aufruf der Funktion `ok`) zurückgesandt, da dies ein normales Verhalten für eine Aktivität ist (Zeilen 19-22).

Im Normalfall wird jedoch der Bezeichner der anzuzeigenden Aktion ermittelt und dorthin umgeleitet. Auch hierfür wird wiederum der Statuscode 303 (`seeOther`) verwendet und Ziel ist eine URL der Form `/Action/`, gefolgt vom eindeutigen Bezeichner der Aktion[28]. Das Verhalten beim Aufruf und bei der Ausführung von Aktionen wird etwas später in diesem Abschnitt detaillierter erläutert.

Fortführen einer Aktivität

Wie soeben bereits erwähnt, existiert mit `continueActivity` eine Funktion innerhalb der `ActivityResource`, die zum Fortsetzen von bereits gestarteten Aktivitäten dient. Diese ist ebenfalls schematisch in Abbildung 8.44 dargestellt. Wie an der Annotation `@GET` und den Parametern der Funktion zu sehen, wird diese über eine URL der Form `/Activity/` mit angehängtem Bezeichner aufgerufen. Dies entspricht insofern der REST-Konvention, als in der Regel lediglich der aktuelle Status der Aktivität zurückgegeben wird und keine Änderungen daran vorgenommen werden.

In einem ersten Schritt wird innerhalb der Funktion der `ActivityExecutionContext` mit dem gegebenen Bezeichner aus dem `GlobalWorkflowContext` abgerufen. Dieser ist entweder bereits darin vorhanden oder wird aus der Datenbank abgerufen. Sollte der `ActivityExecutionContext` erfolgreich abgerufen worden sein (Zeile 35), wird der zu-

[27] Die eindeutigen Bezeichner beziehen sich auf diejenigen, die das `Context`-Objekt identifizieren. Darüber lässt sich ermitteln, welche konkrete Instanz des Ablaufs adressiert wird.

[28] Auch bei diesem Bezeichner handelt es sich um die `id` des zugehörigen `Context`-Objektes.

gehörige `ActivityExecutor` erzeugt, Parameter wie der Bezeichner der Session gesetzt (Zeile 36) und letztlich der Bezeichner der als Nächstes anzuzeigenden Aktion ermittelt (Zeile 37). Sollte keine Aktion ausführbar sein, wird die `ActivityFinished.jsp` dargestellt (Zeilen 39-42). Im Normalfall wird jedoch zur Ausführung und Darstellung der nächsten Aktion umgeleitet (Zeilen 44-45). Sollte bereits der Versuch, den Zustand der gewünschten Aktivität zu finden, fehlgeschlagen sein, wird auf die Hauptseite der Webapplikation verwiesen (Zeile 47).

Aufruf und Absenden von Aktionen

Von besonderer Bedeutung bei er Abarbeitung von Aktivitäten ist die Ausführung der darin enthaltenen *Aktionen*. Zum besseren Verständnis des Gesamtablaufs der Ausführung von Aktionen sind die einzelnen Schritte im Sequenzdiagramm in Abbildung 8.45 aufgeführt. Da bei der Anzeige von Seiten eine Interaktion mit einem Nutzer stattfindet, existiert auch hier eine Ressourcenklasse (in der Abbildung `ActionResource` in der mittleren Lebenslinie), über die vom Browser aus mit Aktionen kommuniziert werden kann. Die Struktur dieser Klasse ist im Quellcodeausschnitt in Abbildung 8.46 dargestellt.

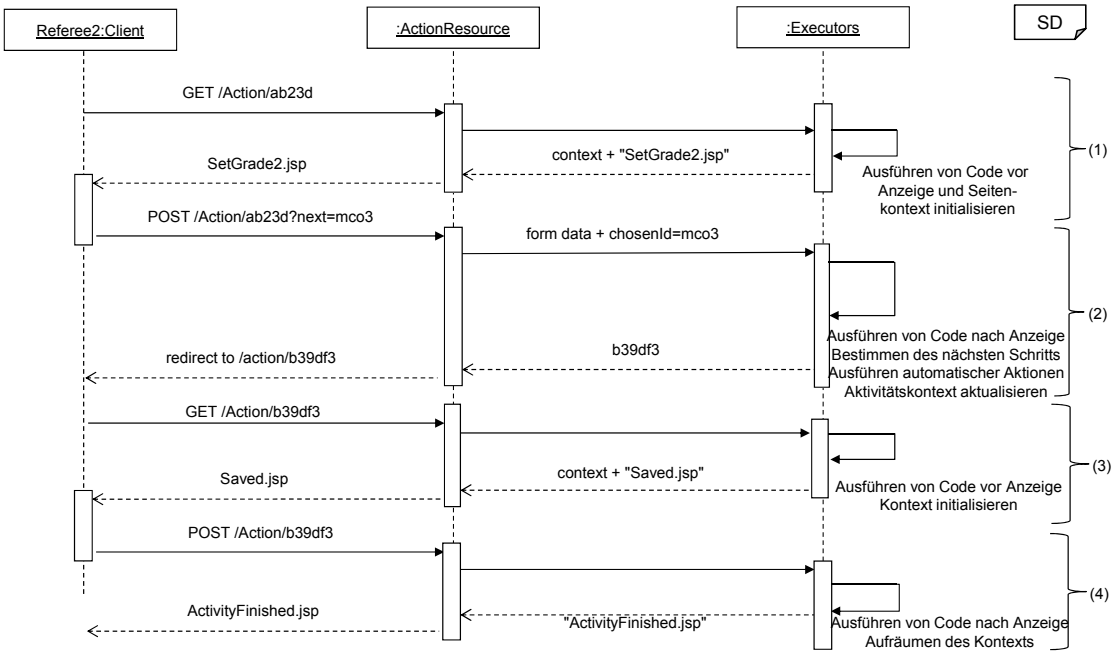


Abbildung 8.45.: Ablauf bei Aufruf und Absenden von Aktionen

Der in Abbildung 8.46 gezeigte Ablauf entspricht der Anzeige der Aktion `SetGrade2` aus Abbildung 8.5. Die an der Sequenz beteiligten Kommunikationspartner sind hier der `Client` `Referee2`, die `ActionResource` sowie die `Executors`, die letztlich die Ausführung vornehmen. Unter `Client` ist hier nicht eine Klasse zu verstehen, sondern der Webbrowser, der die entsprechenden Aufrufe an die `ActionResource`-Klasse stellt und davon auch die

Antworten erhält. Die `ActionResource` ist diejenige Klasse, an die die HTTP-Anfragen des Browsers weitergeleitet werden und worin die Bearbeitung einer einzelnen Aktion angestoßen wird. Die Lebenslinie `Executors` fasst sämtliche `Executor`-Klassen zusammen, die an der Ausführung einer Aktion bzw. der Aktivität, deren Teil die Aktion ist.

Zu Beginn eines typischen Ablaufs, wie er in Abbildung 8.45 dargestellt ist, wird eine Anfrage per HTTP `GET` gesendet, die der Form `/Action/`, gefolgt von einem eindeutigen Bezeichner der aufzurufenden Aktion, entspricht. Eine solche Anfrage wird beispielsweise durch die Umleitung in Zeile 24 bzw. 44 in Abbildung 8.44 erzeugt. Entsprechend der Anforderungen des REST-Paradigmas bzw. des HTTP-Protokolls werden, sofern der Code *vor Anzeige der Seite* innerhalb der Aktion nichts Gegenteiliges definiert wurde, keine Änderungen am Zustand der *Ressource Aktion* vorgenommen. Andersartige Änderungen, etwa das Manipulieren und Speichern von Objekten in der Datenbank, sollten vor der Anzeige einer Seite ebenfalls vermieden werden, da der Aufruf einer Aktion in diesem Fall nicht mehr *sicher* (d. h. ohne Nebeneffekte) wäre. Die Funktion, die durch die `GET`-Anfrage aufgerufen wird, ist schematisch in Abbildung 8.46 in den Zeilen 5-20 dargestellt.

Java

```
1  @Stateless
2  @Path("/Action")
3  public class ActionResource {
4
5    @GET
6    @Path("{id}")
7    public Response getAction(@PathParam("id") String id, @QueryParam("menuid") String ↩
         menuid) {
8      // Initialisieren des ActivityExecutors ae und Setzen der Parameter
9      if (IsUserInRolesTag.isUserInRoles(request, ae.getAllowedRolesString())) {
10       // Setzen von Parametern
11       ViewContext vc = ae.executeBeforeViewAndGetPageName(id);
12
13       if (vc.containsChildActivityId()) {
14         URI target = new URI("/Activity/" + vc.getChildActivityContextId());
15         return Response.seeOther(target).contentLocation(target).build();
16       }
17       return Response.ok(new Viewable("/" + vc.getNextPage() + ".jsp", vc.getPageEntity()↩
             )).build();
18     }
19     return null;
20   }
21
22   @POST
23   @Path("{id}")
24   public Response submit(MultivaluedMap<String, String> form, @PathParam("id") String id)↩
         {
25     // Initialisieren des ActivityExecutors ae und Setzen der Parameter
26     if (context == null) {
27       return Response.ok(new Viewable("/InvalidState.jsp", null)).build();
28     }
29
```

```
30      if (IsUserInRolesTag.isUserInRoles(request, ae.getAllowedRolesString())) {
31        // Setzen von Parametern wie Nutzername etc.
32        try {
33          ae.initPageEntityAndAction(id, form);
34        } catch (ValidatorException e) {
35          return Response.status(400).entity(new Viewable("/" + ae.getPageNameForActionId(↩
                id) + ".jsp", ae.getPageEntityForActionId(id))).build();
36        }
37
38        // Ausführen von Code nach Absenden und Ermitteln der nächsten Aktion
39        if (nextActionId == null) {
40          return Response.ok(new Viewable("/ActivityFinished.jsp", null)).build();
41        }
42        URI target = new URI("/Action/"+nextActionId);
43        return Response.seeOther(target).contentLocation(target).build();
44      }
45      return null;
46    }
47  }
```

Abbildung 8.46.: Ressourcenklasse zum Aufruf und Absenden von Aktionen

Die Funktion `getAction` besitzt zwei Parameter vom Typ `String`, wobei `id` den eindeutigen Bezeichner der Aktion repräsentiert, `menuid` erneut lediglich zur Aktivierung des korrekten, geöffneten Menüeintrags dient. Im ersten Teil der Funktion wird der zur Aktion gehörende `ActivityExecutor` initialisiert, wobei der aktuelle Zustand aus dem `GlobalWorkflowContext` abgerufen wird (symbolisiert durch den Kommentar in Zeile 8). Nachfolgend wird überprüft, ob der aktuell aufrufende Nutzer hinsichtlich seiner Rollen im System berechtigt ist, die angeforderte Aktion auszuführen. Dies ist der Fall, wenn der aktuelle Nutzer Zugriff auf die übergeordnete Aktivität besitzt (Zeile 9). Sollte dies der Fall sein, werden weitere Parameter wie beispielsweise der aktuelle Nutzer und die `menuid` aus dem Funktionsaufruf gesetzt und der Teil der Aktion ausgeführt, der *vor* der Anzeige der Seite deklariert wurde (Zeilen 10-11). Ergebnis dieses Aufrufs ist ein `ViewContext`-Objekt, welches die notwendigen Informationen für die Darstellung der zur Aktion gehörigen Seite enthält. Dies sind beispielsweise die zugehörige `PageEntity` sowie der Name der darzustellenden JSP-Seite. Ferner kann dieses Objekt den Bezeichner einer potenziellen Kindaktivität enthalten, wohin in diesem Fall umgeleitet wird (Zeilen 13-16). Dies führt letztlich zum Aufruf der Methode `continueActivity`, wie sie in Abbildung 8.44 dargestellt ist. Im Fall, dass die aufgerufene Aktion keine Kindaktivität aufruft, wird im Regelfall die zur Aktion zugehörenden JSP-Seite mit der entsprechenden `PageEntity` zurückgegeben (Zeile 17). Die soeben beschriebenen Abläufe sind die in Abbildung 8.45 unter (1) bzw. (3) dargestellten.

Bei der Anzeige der Seite wird ein einziger Absenden-Knopf zu den Inhalten hinzu generiert, wenn der nachfolgende Ablauf eindeutig determiniert ist bzw. erst durch Bearbeitung der übermittelten Parameter ermittelt werden kann. Anders verhält es sich in dem Fall, dass nach der aktuell dargestellten Aktion ein Entscheidungsknoten folgt, der über keine Wächter-

bedingungen verfügt. Dies ist auch im dargestellten Beispiel der Fall (2). Aufgrund dessen wird für jede der am Entscheidungsknoten möglichen Alternativen ein Knopf in der Seite angezeigt, der die Auswahl einer der Aktionen repräsentiert. Im vorliegenden Beispiel wären dies je ein Knopf für die Aktionen `SaveAndNotify` und `Save`. Diese Information wird als Teil der Formulardaten mit übermittelt, was in Abbildung 8.5.2 über den Parameter `chosenId` dargestellt wird. Wie üblich bei Formulardaten werden diese per HTTP `POST` übermittelt, was zum Aufruf der Funktion `submit` in der `ActionResource` in Abbildung 8.46 führt. Dieser Funktion besitzt als Parameter eine `MulitvaluedMap`, die die Formularwerte enthält, sowie einen `String id`, der den eindeutigen Bezeichner der Aktion repräsentiert, an die die Daten zur Bearbeitung gesandt werden. Auch in dieser Funktion werden zu Beginn die `Executor`-Klassen erzeugt und mit den Kontexten aus dem `GlobalWorkflowContext` initialisiert (Kommentar in Zeile 25). Sollte der zur Aktion gehörige `context null` sein, so wird zur Seite `InvalidState.jsp` umgeleitet. Dies kann beispielsweise in dem Fall auftreten, dass ein Formular doppelt abgesandt wurde. In diesem Fall existieren zur übermittelten `id` keine gültigen Kontextinformationen mehr, da der Ablauf bereits einen oder mehrere Schritte weiter ausgeführt wurde (Zeilen 26-28).

Auch in dieser Funktion wird die Zulässigkeit des Aufrufs der Aktion überprüft (Zeile 30), was jedoch an dieser Stelle die Regel ist, da sonst die Seite nicht angezeigt worden wäre. Im nächsten Schritt werden weitere Parameter wie beispielsweise Nutzernamen in der `Executor`-Klasse gesetzt (Zeile 31) und die Werte aus dem übermittelten Formular werden in das `PageEntity`-Objekt der zur Aktion gehörenden Seite übertragen. Sollten hier syntaktische Fehler auftreten (wie invalide E-Mail-Adressen oder beliebige Zeichen wo Zahlen erwartet wurden), wird dies per `ValidatorExeception` abgefangen und die zuvor angezeigte Seite wird erneut mit den übermittelten Werten und entsprechenden Fehlermeldungen angezeigt (Zeilen 32-36).

In den Schritten, von denen über den Kommentar in Zeile 38 abstrahiert wurde, wird zunächst der Code ausgeführt, der in der aktuellen Aktion *nach* der Anzeige der Seite spezifiziert wurde. Anschließend wird von Seiten des `ActionExecutors`, der zur aktuellen Aktion gehört, die übergeordnete Aktivität über Verwendung des Observer-Musters benachrichtigt, so dass dieser mit der Ausführung nachgelagerter Knoten fortfahren kann. Im aktuell betrachteten Beispiel wird die als Nächstes auszuführende Aktion per übermitteltem Parameter `chosenId` ermittelt. In anderen Fällen können an dieser Stelle jedoch komplexe Abfolgen von Knoten oder auch Aktionen ohne darzustellende Seiten ausgeführt werden, wobei stets per Observer-Muster die Ausführung vom zugehörigen `ActivityExecutor` gesteuert wird. Rückgabewert dieses Prozesses ist der eindeutige Bezeichner der als Nächstes anzuzeigenden Aktion. Sollte die Aktivität bereits beendet sein, wird dies durch die Rückgabe eines `null`-Wertes und der Anzeige der `ActivityFinished.jsp`-Seite behandelt (Zeilen 39-41). Im Erfolgsfall wird jedoch per Weiterleitung an die nächste Aktion verwiesen (Zeilen 42-43). Dies führt dazu, dass der in diesem Abschnitt beschriebene Ablauf erneut mit der `GET`-Anfrage startet, wie dies in (3) in Abbildung 8.5.2 dargestellt ist. In (4) wird durch erneutes Absenden eines Formulars der Ablauf der Aktivität beendet.

8.5.3. Weitere Laufzeitkomponenten

Neben den im vorherigen Abschnitt beschriebenen Klassen enthält das Projekt `montiwis-target` einige weitere Klassen, die unterstützende Funktionalität für das generierte System bereitstellen. Auf diese wird im Folgenden kurz eingegangen.

Taglibrary zur Überprüfung von Nutzerrollen

Die Klasse `de.montiwis.taglib.IsUserInRoleTag` stellt Funktionalität bereit, über die geprüft werden kann, ob ein bestimmter Nutzer einer der übergebenen Rollen zugeordnet ist. Hierfür kann dank der Verwendung des *FlexibleJdbcRealms* (vgl. Abschnitt 9.4.2) über die Funktion `isUserInRole` der Klasse `HttpServletRequest` die Zugehörigkeit zu einer Rolle im Java EE Server abgefragt werden. Diese Klasse mit ihren Methoden wird einerseits aus dem Java-Code der generierten Applikation direkt verwendet, steht andererseits aber auch als Taglibrary zur Verfügung, so dass sie in den JSP-Seiten zum Einsatz kommen kann.

Servletfilter

Eine weitere Klasse, die als Teil des `montiwis-target`-Projekts mit ausgeliefert wird, ist `de.montiwis.filter.PostWrapperFilter`. Diese dient dazu, HTTP `POST` Anfragen, denen über den Parameter `_method` einer der Werte `put` oder `delete` hinzugefügt wurde, in eine entsprechende HTTP-Anfrage zu überführen. Diese Klasse wird als Servletfilter in die Bearbeitung von HTTP-Anfragen eingehängt, so dass in den nachgelagert aufgerufenen Ressourcenklassen die Anfragen transparent für diese verarbeitet werden können. Der Hintergrund für dieses Verfahren ist in Abschnitt 5.3.1 erläutert.

8.6. Andere Ansätze

Nach der Spezifikation der Anforderungen, die für MontiWIS an die Ablaufbeschreibungssprache gestellt wurden, wird im folgenden Abschnitt diskutiert, welche Möglichkeiten für die Umsetzung der Anforderungen betrachtet wurden, welche Vor- und Nachteile diese Ansätze bieten und welcher Ansatz letztlich aus welchen Gründen gewählt wurde.

8.6.1. Workflowbeschreibungssprachen

Durch die Anforderungen, die in Abschnitt 8.1.2 aufgestellt wurden, wird unmittelbar die Nähe zu Workflow- oder Prozessbeschreibungssprachen deutlich. Ein Workflow ist nach der Definition der Workflow Management Coalition *„Workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal.“* [Hol95]. Dies trifft auch für die Abläufe in MontiWIS zu. Seiten oder auch Komponenten, die über Kanten miteinander verbunden sind und zwischen denen alternative Abhängigkeiten bestehen, werden auch hier zwischen beteiligten Akteuren (menschlich oder auch automatisch) ausgetauscht, wobei der Gesamtablauf sich aus den einzelnen Schritten zusammensetzt.

Eine für die Spezifikation von klassischen Geschäftsprozessen und Abläufen häufig eingesetzte Notation ist der BPMN Standard [Obj11a], welcher oftmals im Kontext professioneller Workflowengines zum Einsatz kommt. Obwohl die Anforderungen recht ähnlich zu denen von Workflows sind, die in *ausgewachsenen* Workflowengines ausgeführt werden, wurde in MontiWIS davon Abstand genommen, eine solche einzusetzen. Mit einem Web-Informationssystem als Zieldomäne ist die Integration einer Workflowengine nicht vollständig zielführend, da das Zusammenspiel mit derartigen Engines in der Regel relativ aufwendig bzw. nur nach umfangreichen Anpassungen möglich wäre. So werden in MontiWIS die einzelnen Schritte eines Ablaufs entweder zur Anzeige von Seiten, zur Ausführung beliebiger Geschäftslogik oder beidem genutzt. Diese Anforderungen werden in der Regel von Workflowengines nicht optimal unterstützt, obwohl einige Ansätze existieren, die hier interessante und durchaus praktikable Lösungen bieten, jedoch weiterhin Unzulänglichkeiten besitzen, die von einem Einsatz abhielten.

Activiti [Act] und *jBPM* [JBP] sind beides fortschrittliche Workflowengines, deren Prozesse über BPMN-Modelle [Obj11a] spezifiziert werden. Zwar lassen sich damit sämtliche ablaufspezifischen Konstrukte abbilden, die für MontiWIS gefordert wurden, aber die Einbindung von beliebig gestalteten Seiten wird nicht ohne Weiteres unterstützt. Dies trifft ebenfalls auf weitere untersuchte Workflowengines zu, deren Abläufe in der Regel über BPMN definiert werden. Diese bieten überwiegend eigenständige Oberflächen an, um beispielsweise für einen Nutzer sämtliche vorliegenden Aufgaben anzuzeigen oder bestimmte Eingaben zu tätigen. Sie sind jedoch oftmals so spezifisch, dass sie sich schwerlich in ein allgemeines Web-Informationssystem integrieren lassen. Ein Aufruf einer externen Workflowengine wäre zugegebenermaßen möglich, um den Zustand eines bestimmten Ablaufs zu verfolgen, potenzielle nächste Seiten zu bestimmen und nachfolgende Aktionen auszuführen, jedoch würde auch dies erhebliche Anpassungen an weiteren eingesetzten Technologien und der Workflowengine selbst erfordern, wovon auch hier Abstand genommen wurde.

Spring Web Flow [Webb] bietet eine proprietäre Syntax und ist an keinen Standard angelehnt. Es lässt sich hervorragend in die Spring Infrastruktur [Spr] einbinden inklusive der Möglichkeit, diese in JSF oder JSP-Seiten zu integrieren, allerdings existiert keine Unterstützung für gewünschte Funktionalität wie beispielsweise verteilte Workflows.

Weiterhin existieren eine Reihe weiterer Sprachen, die bei der Spezifikation von Geschäftsprozessen zum Einsatz kommen, aber hier nicht näher betrachtet werden sollen, da für sie hinsichtlich der Eignung für den Einsatz in einem Web-Informationssystem Ähnliches gilt wie für die bereits erwähnten Sprachen. Ein sehr ausführlicher Überblick über Business Process Management (BPM) Ansätze und Notationen wird in [KLL09, Ko09] gegeben.

Die Generierung von Code aus Aktivitätsdiagrammen, die sich möglichst nah an den UML-Standard und die Semantik der UML hält, wird in [GR11] beschrieben. Die dort beschriebenen Codestücke beziehen sich jedoch nicht auf eine bestimmte Anwendungsdomäne, sondern die Ausführungssemantik von Aktivitäten generell, weswegen sie sich auch von dem in MontiWIS verwendeten Code unterscheiden und keine Möglichkeit von Persistierung des Zustands bieten. Werkzeugunterstützung für einen ähnlichen Ansatz, wobei Aktivitätsdiagramme ausgeführt werden können, stellt [SKRG07] dar.

Zusammenfassend lässt sich feststellen, dass vollständige Workflowengines inklusive der

dafür eingesetzten Sprachen eine mächtige Möglichkeit darstellen, um diejenigen Anforderungen, die auch für MontiWIS für die Ablaufsteuerung von Seiten und Geschäftslogik bestehen, umzusetzen. Allerdings sind die angebotenen Schnittstellen (in der Regel per API oder auch per Webservice) nur bedingt geeignet, zur Steuerung eines Web-Informationssystems eingesetzt zu werden.

8.6.2. Web-Modellierungsansätze

Neben den Sprachen und Applikationen, die primär ihren Fokus auf Ablauf- und Workflow-Modellierung legen, existieren weiterhin Ansätze zur holistischen Beschreibung von Web-Informationssystemen, was auch die Spezifikation von Seitenablauf und Geschäftslogik auf die eine oder andere Weise einschließt.

WebML [CFB+03, BCFM08] beispielsweise erlaubt die Spezifikation von Abläufen über explizite Verknüpfungen zwischen Seiten, die auch als solche dargestellt werden. Diese spannen damit Hyperlink-Strukturen auf, die per se zunächst keine weitere Logik unterstützen, aber das Einbinden von Operationsblöcken erlauben, über die Modifikationen an Datenobjekten vorgenommen werden können. Ferner können funktionale Konstrukte zur Abbildung von Prozess-Strukturen eingebunden werden [BCF06], die eine ähnliche Ausdrucksmächtigkeit wie bereits besprochene Workflowbeschreibungssprachen besitzen. Die Verwendung vieler einzelner Konstrukte, die in einem Modell gemeinsam mit den Seitenbeschreibungen und potenziellen Geschäftslogik- oder Webserviceaufrufen codiert werden, wurde im Kontext von MontiWIS als nicht vorteilhaft erachtet. Für eine grafische Sprache, wie sie WebML verwendet, mag dies praktikabel sein, eine textuelle Sprache würde aufgrund einer solchen Vorgehensweise sehr unübersichtlich werden und ohne extensive Editorunterstützung auch schwer erlern- und handhabbar sein. Das proprietäre Werkzeug *WebRatio*, das zur Modellierung von WebML eingesetzt wird, unterstützt in der aktuellen Version auch die Spezifikation von BPMN-Modellen. Diese lassen sich mit WebML-Modellen verknüpfen, um so auf Seiten bzw. Funktionalität oder Datenmodell während der Ausführung der generierten Prozessapplikation zugreifen zu können.

Auch in *WebDSL* [Vis08b] wird ebenfalls ein Ansatz verwendet, der explizite Hyperlinks auf andere Seiten erlaubt. Diese werden in textueller Form als Verweise auf eine andere Seite angegeben. Ferner existiert auch für WebDSL eine Erweiterung, die Prozessunterstützung bereitstellt [HVV08]. Diese wird jedoch in der aktuellen Version offenbar nicht (mehr) unterstützt (siehe hierzu die Dokumentation auf [Webc]). Geschäftslogik in WebDSL kann entweder per domänenspezifischer Aktionssprache spezifiziert werden, oder aber auch per Aufruf von externem Java-Programmcode. Damit verfolgt WebDSL insofern einen ähnlichen Ansatz wie MontiWIS, als auch dort beliebige Geschäftslogik eingebunden werden kann. Für eher simple Logik wurde in MontiWIS jedoch statt einer übersichtlichen, aber dennoch eigenständigen Aktionssprache direkt Java verwendet, was die Lernkurve für die Benutzung nochmals etwas abflachen lässt.

In *WSDM* [DCP08] wird Geschäftslogik der Anwendung und Nutzerinteraktion über sog. *tasks* abgebildet, die ähnlich zu Workflowbeschreibungssprachen miteinander verknüpft werden können. Diese werden schrittweise in Komponenten aufgeteilt, zu Seiten zusam-

mengefasst und über Templates letztlich in einem Transformations- und Generierungsschritt verarbeitet. Die Spezifikation erfolgt dabei durch die Verwendung grafischer Notationselemente, wobei die Speicherung und Weiterverarbeitung über RDF bzw. einen eigenen XML-Dialekt geschieht.

Auch in *OOHDM* [RS08] lassen sich Seiten über sog. *navigational schemes* spezifizieren, die Abhängigkeiten und Navigationsmöglichkeiten definieren. Dazu kommen abgewandelte Klassendiagramme zum Einsatz.

UWE [KKZB08, SK06] verwendet einen weitestgehend UML-konformen Ansatz für die Modellierung von Abläufen und Navigationselementen. Klassendiagramme kommen dabei im *navigation model* zum Einsatz. Klassen besitzen spezielle (grafische) Stereotypen für die Spezifikation von Seitenarten, Assoziationen werden für die Verknüpfung von Seiten verwendet. Das Verhalten bestimmter Klassen kann per Aktivitätsdiagramm näher spezifiziert werden.

In *RMM* [ISB95] orientieren sich die Verweise stark am zugrundeliegenden Datenmodell, so dass überwiegend von Datenobjekten zu assoziierten Objekten navigiert werden kann, wobei Mengen von Objekten beispielsweise über Listen und Indizes dargestellt werden können.

Für *Hera* wurden in [BFH06] Aktivitätsdiagramme vorgeschlagen, die dann in die Navigationsstrukturen von Hera übersetzt und von dort weiterverarbeitet werden können.

In *OO-H* [GCP01] werden sogenannte Navigational Access Diagrams verwendet, um die Hyperlinkstruktur der Applikation für bestimmte Nutzergruppen festzulegen. Diese können aus Aktivitätsdiagrammen abgeleitet werden [KKCM04]. Eine bedingte Einbindung von Geschäftslogik ist über eine proprietäre Sprache möglich, womit bestimmte Aktionen an das Vorkommen bestimmter Ereignisse geknüpft werden können.

OOWS [FPP+08] lässt über *navigational maps* die Hyperlinkstruktur der Anwendung spezifizieren. Ferner erlauben Aktivitätsdiagramme bereits in der Anforderungsphase die Modellierung von Prozessen, über die Prozessstrukturen aufgebaut und ausgeführt werden können. Eine andere Möglichkeit ist die Spezifikation von BPMN-Modellen, die sowohl in WS-BPEL Code als auch verschiedene *abstract information units* transformiert und ausgeführt werden können [TP06].

Kapitel 9.

Applikationssprache

9.1. Grundlagen und Anforderungen

9.1.1. Grundlagen

Über die bisher vorgestellten sprachlichen Mittel lassen sich bereits große Teile eines Web-Informationssystems beschreiben. Über die Klassendiagramme aus Kapitel 6 wird das Domänendatenmodell beschrieben, das die Struktur der Daten beschreibt, die in der generierten Applikation verwaltet werden sollen. Webseiten, auf denen die besagten Datenobjekte unter anderem angezeigt und mit Werten belegt werden können, lassen sich über die Seitenbeschreibungssprache aus Kapitel 7 spezifizieren. Die Abläufe, die die Geschäftslogik und die Verschaltung der Seiten in einem Web-Informationssystem repräsentieren, lassen sich über die Aktivitätsdiagramme aus Kapitel 8 beschreiben.

Bislang wurden jedoch noch keinerlei Konzepte vorgestellt, über die sich der gesamte Aufbau der Applikation beschreiben lässt. Weder kann über die bis dato erläuterten Sprachmittel festgelegt werden, welche Aktivitäten und Seiten in welcher Art und Weise einem Nutzer zur Verfügung stehen, noch welche Nutzergruppe diese aufrufen darf und welcher Gruppe der Aufruf verwehrt bleibt. Die Umsetzung dieser Anforderungen erfolgt in der sogenannten *Applikationssprache*. Im folgenden Kapitel wird diese Sprache und deren Umsetzung in MontiWIS erläutert.

Der Aufbau dieses Kapitels orientiert sich an den vorherigen Kapiteln, weswegen in Abschnitt 9.1.2 zunächst die Anforderungen spezifiziert werden, die an Sprache konkret gestellt werden. Anschließend wird in Abschnitt 9.2 sowohl auf den Aufbau der Sprache als auch auf ihre Verwendung eingegangen. Nachfolgend erläutert Abschnitt 9.3, wie die Modelle verarbeitet werden, wobei die Modellverarbeitung, die definierten Kontextbedingungen, die Zwischenstruktur und ihr Einsatz sowie der Generator selbst erläutert wird. Schließlich beschreiben Abschnitt 9.4 die Umsetzung der betrachteten Aspekte im generierten System und Abschnitt 9.5 erläutert alternative Möglichkeiten zur Spezifikation und Umsetzung ebendieser.

9.1.2. Anforderungen an die Applikationssprache

Die grundlegenden Anforderungen an eine Applikationssprache bzw. die Sprachmittel zur Beschreibung der genannten Funktionalitäten wurden bereits in Abschnitt 2.2 angerissen. Nachfolgend werden diese Punkte konkretisiert und detaillierter beschrieben.

Definition der Menüstruktur der Applikation

In Web-Informationssystemen, wie in MontiWIS eines generiert wird, dienen in der Regel Menüs zur Navigation innerhalb der Anwendung und zum Aufruf der angebotenen Funktionalität. Dies beinhaltet einerseits der Aufruf von Seiten, die entweder rein statische Inhalte präsentieren oder aber standardmäßigen Zugriff auf Datenobjekte ermöglichen, sowie andererseits die Ausführung und das Starten von spezialisierten Abläufen in der Applikation.

Das spezifizierte Menü sollte verschiedene hierarchische Ebenen bieten, beispielsweise um das Anlegen von Objekten und das Auflisten von Objekten eines bestimmten Typs unter einem überschriftartigen Menüpunkt zu bündeln. Die Tiefe der Hierarchiestufen sollte zumindest *drei* betragen, so dass neben übergeordneten Abschnitten zwei weitere (Unter-)Ebenen möglich sind. Auf der obersten Ebene des Menüs können keine Verweise auf Funktionalität angelegt, sondern lediglich Gliederungsüberschriften spezifiziert werden.

Beschriftung von Überschriften und Verweisen

Sowohl Überschriften als auch Verweise auf Seiten oder Aktivitäten sollten beschriftet werden können, was pragmatisch über eine einfache Zeichenkette aber auch durch Verweis auf ausgelagerte Bezeichnungen geschehen kann. Ersteres erlaubt schnelle Ergebnisse, da die Spezifikation einer einzigen Beschriftung genügt, Letzteres erlaubt nachhaltige Inhalte, potenziell in mehreren Sprachen.

Referenzierbarkeit von Aktivitäten, Seiten und Standardseiten im Menü

Neben seiner Beschriftung enthält ein Menüeintrag selbstverständlich auch eine Referenz auf das Ziel, das bei Anklicken des Menüpunkts aufgerufen wird. Hierbei kann es sich in MontiWIS um statische Seiten handeln, die keinerlei Parameter und somit auch keine dynamischen Anteile besitzen. Eine Referenz hierauf sollte über den Seitennamen möglich sein. Weiterhin sollten Standardseiten verlinkt werden können, die beispielsweise zum Neuanlegen von Objekten oder Auflisten sämtlicher vorhandenen Objekte eines Typs dienen (vergleiche hierzu Abschnitt 6.4.2). Wie solche Standardseiten aufgerufen werden, wird nachfolgend in Abschnitt 9.2.2 erläutert. Schließlich sollte es selbstverständlich auch möglich sein, Aktivitäten zu starten. Diese sollten über ihren Namen im Menü referenzierbar sein.

Neben den beschriebenen Elementen sollten ebenfalls bestimmte vordefinierte Seiten referenziert werden können. Dies sind beispielsweise solche, die das Fortsetzen von bereits gestarteten Aktivitäten erlauben, aber auch Login- bzw. Logoutseiten. Derartige Seiten werden ebenfalls über einen festgelegten Namen im Menü verankert.

Definition von Nutzergruppen

Um ein Web-Informationssystem für unterschiedliche Nutzergruppen zugänglich zu machen und jeder Gruppe eine unterschiedliche Sicht auf das System zu ermöglichen, sollten sich verschiedene Gruppen definieren lassen. Darüber werden Zugriffsrechte auf die im vorherigen Abschnitt beschriebenen Komponenten der Applikation festgelegt, so dass die Zugehörigkeit zu einer Gruppe auch die Verfügbarkeit bestimmter Funktionalität einschränkt bzw. ermöglicht.

Nutzergruppen sollten dynamisch zugewiesen werden können, so dass ein Nutzer nicht zwangsläufig zu jedem Zeitpunkt den gleichen Gruppen zugeordnet ist, sondern diese sich durch die Geschäftslogik der Anwendung zuweisen oder entziehen lassen.

Zugriffsrechte auf Aktivitäten und Seiten

Über die soeben beschriebenen Nutzergruppen sollten Zugriffsrechte sowohl auf Seiten als auch auf Aktivitäten zugeordnet werden können. Hierbei handelt es sich um binäre Beziehungen, die entweder die Erlaubnis oder die Verweigerung des Zugriffs zur Folge haben.

Ferner ist es wünschenswert, wenn für jede Klasse festgelegt werden kann, welche Arten der Interaktion mit Objekten dieser Klasse möglich sind. Hierbei sollte zwischen der Berechtigung, neue Objekte anzulegen, dem Bearbeiten von Objekten (was auch das Löschen mit einschließt) sowie dem Anzeigen aller Objekte einer Klasse unterschieden werden. Diese Berechtigungen sollten sich sowohl auf potenzielle Verweise im Menü als auch auf Verweise innerhalb der Applikation auswirken. Anzumerken ist hierbei, dass sich diese Berechtigungen lediglich auf die standardmäßig mitgenerierten CRUD-Seiten beziehen und nicht applikationsweit auf alle Objekte einer bestimmten Klasse. Veränderungen, die im Rahmen von Aktivitäten daran vorgenommen werden, sind von diesen Einschränkungen nicht betroffen.

Kombinierte Definition dieser Aspekte

Während der Entwicklung der Sprache, die die oben genannten Aspekte beschreiben, wurden verschiedene Möglichkeiten hinsichtlich Granularität und Aufteilung betrachtet. Zunächst bestand die Option, sowohl für die Menüstruktur als auch für die Rechte jeweils eigene Sprachen zu definieren, was im einfachsten Fall auch zu mehreren Modelldateien geführt hätte. Aus pragmatischen Gründen und insbesondere aufgrund der Tatsache, dass diese Aspekte in der Regel zusammen definiert werden, sollte die Spezifikation dieser Sachverhalte in einer Datei geschehen. Die Struktur der Menüs hängt wesentlich davon ab, welche Nutzer auf welche Funktionalitäten Zugriff haben, so dass eine gemeinsame Definition sinnvoll erscheint.

9.2. Sprache

Die Beschreibung von Menüstruktur und Zugriffsberechtigung wird, wie in Abschnitt 9.1.2 gefordert, in einer einzigen Sprache realisiert. In diesem Abschnitt wird zunächst in Abschnitt 9.2.1 der Aufbau der Sprache anhand einer kompakten Version der Grammatik erläutert. In Abschnitt 9.2.2 wird die Beschreibung der Menüstrukturen erklärt sowie wie verschiedene Rechtegruppen definiert und ihnen die einzelnen Funktionalitäten der Applikation zugeordnet werden.

9.2.1. Aufbau der Applikationssprache

Der vereinfachte Aufbau der Grammatik der Applikationssprache ist in Abbildung 9.1 dargestellt.

MCSimp

```
grammar Application extends mc.umlp.common.Common {

interface ApplicationElement;

ApplicationDefinition =
  "application" name:Name
  "{"
    ApplicationElement*
  "}";

Menu implements ApplicationElement =
  "menu" name:Name
  "{"
    MenuEntry+
  "}";

MenuEntry =
  LabelEntry | ActivityOrPageReferenceEntry | ImplicitViewEntry;

LabelEntry =
  LevelBullet (label:StringLiteral | labelRef:QualifiedName) ";";

ActivityOrPageReferenceEntry =
  LevelBullet (label:StringLiteral | labelRef:QualifiedName) ":" referencedActivityOrPage↩
      :QualifiedName ";";

ImplicitViewEntry =
  LevelBullet (label:StringLiteral | labelRef:QualifiedName) ":" referencedClass:↩
      QualifiedName "." viewName:Name "()" ";";

LevelBullet =
  LevelOne:["*"] | LevelTwo:["**"] | LevelThree:["***"] ;

Right implements ApplicationElement =
  "right" rightNames:Name ("," rightNames:Name)*
  "{"
     "runs" Runnable ("," Runnable)* ";"
  "}";

Runnable =
  ImplicitView | ActivityOrPageName ;

ActivityOrPageName =
  activityOrPageName:QualifiedName;

ImplicitView =
  referencedClass:QualifiedName "." viewName:Name "()";
}
```

Abbildung 9.1.: Vereinfachte Darstellung der Grammatik der Applikationssprache

Wie in Zeile 1 zu sehen, wird auch diese Grammatik von `mc.umlp.common.Common` abgeleitet, weswegen direkt die benötigten Datentypen und Literale zur Verfügung stehen. Stellvertretend für die verschiedenen Inhalte der Applikationssprache wurde die Schnittstelle `ApplicationElement` (Zeile 3) definiert, die in der `ApplicationDefinition` (Zeilen 5-9) beliebig oft Verwendung findet. Dieses Nichtterminal ist gleichzeitig auch das Startelement bei der Bearbeitung der Sprache.

Das Nichtterminal `Menu` stellt das erste Element dar, das den eigentlichen Inhalt der Applikationssprache repräsentiert. Darüber wird ein *Menü* definiert, welches seinerseits aus beliebig vielen `MenuEntry`-Nichtterminalen besteht. Ein Menüeintrag selbst, definiert in Zeilen 17-18, kann von dreierlei Art sein.

Zum einen kann er als `LabelEntry` eine Überschrift repräsentieren, wobei hinter dem Aufzählungszeichen (`LevelBullet`) lediglich eine Zeichenkette (`StringLiteral`) oder ein qualifizierter Name (`QualifiedName`) auftreten kann. Die Zeichenkette dient zur direkten Angabe der Überschrift, der Name wird als Platzhalter für ausgelagerte Zeichenketten eingesetzt, so dass potenziell mehrere Sprachen unterstützt werden können. Das Aufzählungszeichen (Zeilen 29-30) kann aus einem, zwei oder drei Asterisk-Symbolen (∗) bestehen, wobei diese analog zur Wiki-Syntax [Wik] die Hierarchiestufen innerhalb des Menüs repräsentieren.

Zum anderen erlaubt ein `ActivityOrPageReferenceEntry` die Deklaration eines Verweises auf eine Seite oder eine Aktivität. Ein solcher beginnt ebenfalls mit einem `LevelBullet` und entweder einem `StringLiteral` oder `QualifiedName` zur Bezeichnung der Beschriftung des Links. Schließlich wird auf die Seite oder die Aktivität über einen `QualifiedName` verwiesen.

Zu guter Letzt lassen sich Standardseiten (vgl. Abschnitt 6.4.2) ebenfalls im Menü verankern, so dass darauf zugegriffen werden kann. Dieses Konstrukt wird über das Nichtterminal `ImplicitViewEntry` (Zeilen 26-27) abgebildet. Auch dieser beginnt mit der Deklaration von Hierarchiestufen und Bezeichnung, endet jedoch mit der Möglichkeit, einen qualifizierten Namen und die Art der Standardseite (`viewName`) zur Definition der Seite anzugeben.

Zusätzlich Inhalt der Applikationssprache ist die Definition von *Zugriffsrechten*. Jedes Recht wird spezifiziert über das Nichtterminal `Right` (Zeilen 32-36). Dieses implementiert die Schnittstelle `ApplicationElement`, weswegen es aufgrund des Inhalts von `ApplicationDefinition` mehrfach vorkommen kann. Innerhalb eines einzelnen Rechts können mehrere `Runnable` (Zeilen 38-39) Nichtterminale vorkommen, welche ihrerseits entweder ein `ActivityOrPageName` (Zeilen 41-42) oder ein `ImplicitView` (Zeilen 44-45) Nichtterminal sein können. Ersteres repräsentiert eine Referenz auf eine Aktivität oder eine Seite und besteht lediglich aus dem qualifizierten Namen, über Letzteres lassen sich die verschiedenen Standardseiten der CRUD-Funktionalität referenzieren, die sich wiederum aus dem Namen des Objekts und der Art der Seite zusammensetzen.

9.2.2. Verwendung der Applikationssprache

Nachdem im vorherigen Abschnitt die Implementierungsdetails der Applikationssprache auf Grammatikebene beschrieben wurden, wird in den folgenden zwei Abschnitten die Verwendung von Menü- und Zugriffsrechtebeschreibung erläutert, was Abbildung 9.2 zeigt.

App

```
1  package tm;
2
3  import tm.*;
4
5  application ThesisManamagement {
6
7    menu FirstMenu {
8      * "Pages";
9      ** "Index" : Index;
10     ** "Login" : LoginPage;
11     ** "Logout" : Logout;
12     ** tm.label.contact : Contact;
13
14     * "Student";
15     ** "List" : Student.list();
16     ** "New" : Student.new();
17
18     // weitere Punkte
19
20     * "Activities";
21     ** "Register" : Register;
22     ** "Grade thesis" : GradeThesis;
23     ** "Next actions" : RunningActivities;
24   }
25
26   right guest {
27    runs LoginPage, Index, Register, Contact, TestModell, HierarchyDemo;
28   }
29
30   right user {
31     runs Index, Logout, GradeThesis // ...
32   }
33
34   right admin {
35     runs Index, Logout, Student.edit(), Student.list(), Student.new(), RunningActivities ↩
           // ...
36   }
37 }
```

Abbildung 9.2.: Beispiel für die Applikationssprache

Wie dort zu sehen ist, werden Applikationsbeschreibungen ebenfalls in einem `package` abgelegt (Zeile 1), wobei sie die referenzierten Seiten und Aktivitäten importieren sollten (Zeile 3). Die eigentliche Applikationsbeschreibung beginnt mit dem Schlüsselwort `application`, gefolgt vom Namen der Applikation (hier `ThesisManagement`) (Zeile 5).

Die Menübeschreibung und die Rechtedefinition werden in geschweiften Klammern eingeschlossen und sind in den folgenden Abschnitten beschrieben.

Damit in MontiWIS eine sinnvolle Anwendung generiert werden kann, muss ein Applikationsmodell vorhanden sein, weswegen ein rudimentäres Standardmodell bereits in einem leeren MontiWIS-Anwendungsprojekt vorhanden ist.

Menübeschreibung

Die Beschreibung des Menüs wird eingeleitet durch das Schlüsselwort `menu` (Zeile 7) gefolgt vom Namen des Menüs. Aktuell wird ein einziges Menü in der Applikation unterstützt, jedoch wären bei geringfügigen Anpassungen des Generators auch mehrere Menüs an unterschiedlichen Stellen der Anwendung denkbar, weswegen hier eine Unterscheidung per Namen ermöglicht wird. Die Menüspezifikation selbst findet, eingeschlossen in geschweifte Klammern, im Anschluss daran statt.

Jeder Menüpunkt wird eingeleitet durch ein bis drei Asterisk-Symbole (`*`), die die hierarchische Einordnung des einzelnen Eintrags repräsentieren. Diese Notation ist angelehnt an die in Wikis verwendeten Listendefinitionen [Wik]. Hierbei repräsentiert ein Stern die oberste Ebene des Menüs, zwei Sterne stehen für die zweite Ebene und drei Sterne entsprechend für die dritte Hierarchiestufe. Weitere Ebenen sind aktuell aufgrund der leichteren Benutzbarkeit und sinnvollen Darstellung innerhalb des generierten Systems nicht vorgesehen. Nach den Sternen wird die Bezeichnung des Menüpunkts im generierten System spezifiziert. Dies kann entweder ein vollqualifizierter Name sein, der in der `langstring.properties` Datei abgelegt wird (Zeile 12), oder aber, wie in den übrigen Einträgen zu sehen, eine einfache Zeichenkette, eingeschlossen in Anführungszeichen. Erstere Möglichkeit erlaubt die mehrsprachige Angabe und spätere Änderung der Bezeichnung ohne eine Neugenerierung des Systems, letztere lässt sehr einfach und übersichtlich eine pragmatische Definition des Menüs zu, falls die eben beschriebene Flexibilität später nicht benötigt wird. Nach der Bezeichnung wird nach einem Doppelpunkt der Verweis auf eine Seite (z. B. `Contact` in Zeile 12), eine Aktivität (z. B. `Register` in Zeile 21) oder aber die Referenz auf eine Standardseite (z. B. `Student.list()` und `Student.new()` in Zeilen 15-16) angegeben. Von den vorhandenen Standardseiten bieten auch nur diese beiden Arten einen sinnvollen Einstiegspunkt in die Bearbeitungskette, weswegen beispielsweise die Seite zum Ändern von Objekten (prinzipiell abgebildet durch das Suffix `edit`) nicht in einem Menü auftreten kann. Diese Funktionalität ist über die `list`-Seite erreichbar. Neben den Seiten, die über das Modell definiert wurden, lassen sich einige wenige weitere Seiten in das Menü einbinden, die spezielle Funktionalität innerhalb des Web-Informationssystems erfüllen. Ein Überblick darüber ist in Tabelle 9.3 dargestellt.

Über `LoginPage` wird auf die Seite verwiesen, auf der dem Nutzer die Möglichkeit zum Einloggen in die Applikation angeboten wird. `Logout` verweist auf die Seite, die das Ausloggen aus der Applikation anstößt und `RunningActivities` schließlich sendet den Nutzer zur Seite, die die für ihn aktuell laufende Aktivitäten auflistet und die Möglichkeit zum Fortsetzen einer Aktivität bietet.

Bei der Definition von hierarchischen Menüs ist zu beachten, dass beim Hinzufügen einer weiteren Hierarchiestufe, wie dies beispielsweise in den Zeilen 8-9, 14-15 oder 20-21 der Fall ist, stets der letzte Punkt vor der neuen Hierarchiestufe selbst keinen Verweis auf Funktionalität

Seitenreferenz	Erläuterung
`LoginPage`	Verweist auf die Login-Seite der Applikation
`Logout`	Verweist auf die Logout-Funktionalität der Applikation
`RunningActivities`	Verweist auf die Seite zur Anzeige von aktuell laufenden Aktivitäten

Tabelle 9.3.: Mögliche statische Verweise in der Menüdefinition

besitzen, sondern lediglich eine Bezeichnung für das untergeordnete Menü beinhalten darf.

Ein Beispiel für ein generiertes Menü ist in Abbildung 9.4 dargestellt.

Abbildung 9.4.: Generiertes Menü aus dem Beispiel in Abbildung 9.2 als `guest` Nutzer

Dieses Menü entspricht dem Beispiel aus Abbildung 9.2 mit den Nutzerberechtigungen eines `guest`, wobei hier aufgrund von Zugriffsbeschränkungen das Menü `Student` mit seinen Unterpunkten nicht mit dargestellt ist. Wie dort zu sehen, werden die Bezeichnungen `Pages` und `Activities` als aufklappbarer Menüpunkt umgesetzt, die darunter angeordneten Verweise auf die Funktionalität als einfacher Link.

Beschreibung der Rollen und Rechte

Neben der Spezifikation der Menüstruktur der Anwendung werden zusätzlich die verschiedenen Nutzergruppen innerhalb des Web-Informationssystems in der Applikationssprache definiert. Dies ist ebenfalls im Beispiel in Abbildung 9.2 dargestellt. Eine Berechtigung wird definiert durch das Schlüsselwort `right` (Zeilen 26, 30 und 34) gefolgt von einem Namen, der idealerweise die Art der Berechtigung widerspiegeln sollte. Nach dem Namen werden in geschweiften Klammern die Funktionalitäten definiert, auf die diese Nutzergruppe Zugriff hat, wobei die Liste der Artefakte nach dem Schlüsselwort `runs` angegeben wird. Dies können sowohl Namen von Seiten aus den Modellen, die in Tabelle 9.3 gelisteten vordefinierten Seiten, die in den Modellen definierten Aktivitäten sowie Berechtigungen auf Klassen des Domänendatenmodells sein. Die verschiedenen Berechtigungen, die für Objekte einer bestimmten Klasse definiert werden können, sind in Tabelle 9.5 zusammengefasst. Die dargestellten Berechtigungen werden an den Klassennamen angehängt und davon durch einen Punkt getrennt, wie beispielsweise in Zeile 35 des Beispiels in Abbildung 9.2 dargestellt.

Berechtigung	Erläuterung
`new()`	Berechtigung zum Anlegen neuer Objekte und Anzeigen der entsprechenden Seite und des Menüpunkts
`list()`	Berechtigung zum Auflisten aller Objekte eines Typs und Anzeige der entsprechenden Seite und des Menüpunkts
`edit()`	Berechtigung zum Bearbeiten von Objekten einer Klasse sowie zum Löschen derartiger Objekten

Tabelle 9.5.: Bedeutung von Berechtigungen auf Klassen

Die `new()` Berechtigung erlaubt einem Nutzer die Anzeige der Seite zum Anlegen eines neuen Objekts und den Zugriff auf den entsprechenden Menüpunkt. Nutzer mit `list()` Berechtigung sind in der Lage, sich eine Liste sämtlicher Objekte eines Typs anzeigen zu lassen. Auch hier wird dadurch der Verweis im Menü angezeigt. Ein Nutzer mit der `edit()`-Berechtigung darf generell die Werte sämtlicher Objekte eines Typs bearbeiten sowie Objekte des Typs löschen. Diese Berechtigung manifestiert sich nicht durch die Anzeige eines Links im Menü, sondern erst auf den generierten Seiten. So werden beispielsweise in der Listenansicht entsprechende Verweise zum Bearbeiten des Objekts oder zu dessen Entfernung aus der Datenbank angezeigt, sofern dem Nutzer dieses Recht erteilt wurde. Generell ist zu bemerken, dass die in der Applikationssprache festgelegten Berechtigungen auch Einfluss auf die generierten Verweise innerhalb der Seiten und demnach auch auf die Transitionen im Zustandsdiagramm in Abbildung 6.22 haben.

Die hier beschriebenen Berechtigungen beziehen sich im Wesentlichen auf die Standardseiten. Eine Prüfung der Zugriffsrechte auf Objekte, die innerhalb von Aktivitäten verarbeitet werden, findet nicht statt.

9.3. Generierungsprozess

Nach der Erläuterung, wie die Applikationssprache auf Modellebene zu verwenden ist, wird nun auf die Verarbeitung der Modelle eingegangen. Zunächst werden in Abschnitt 9.3.1 die Kontextbedingungen erläutert, die Hinweise auf semantische Inkonsistenzen in den Modellen aufdecken und an den Entwickler der Modelle weitergeben. Anschließend gibt Abschnitt 9.3.2 einen Überblick über die Zwischenstruktur, die im weiteren Verlauf der Modellverarbeitung und Generierung zum Einsatz kommt. Der Prozess der Erzeugung der Zwischenstruktur aus dem AST der Modellinstanzen durch einen Visitor wird in Abschnitt 9.3.3 ausführlich erklärt. Auf der Zwischenstruktur werden weiterhin Rekonstruktionen ausgeführt, die in Abschnitt 9.3.4 erläutert werden. Schließlich werden in Abschnitt 9.3.5 die Schritte der Codegenerierung erläutert, an denen die Artefakte der Applikationssprache beteiligt sind.

9.3.1. Kontextbedingungen

Wie auch bei den bisher besprochenen Sprachteilen ist es bei der Applikationssprache sinnvoll, bestimmte Sachverhalte vor der finalen Generierung der Applikation zu prüfen, um die Sinnhaftigkeit und Korrektheit des Modells sicherzustellen bzw. Hinweise auf etwaige Ungereimtheiten zu geben. Diese Mechanismen werden im folgenden Abschnitt erläutert.

Verwendete Symboltabelleneinträge

In einem ersten Schritt werden bei der Prüfung der Kontextbedingungen die Symboltabelleneinträge aus der AST-Struktur des eingelesenen und verarbeiteten Modells erstellt. Die Klassen, die hierbei beteiligt sind, zeigt Abbildung 9.6.

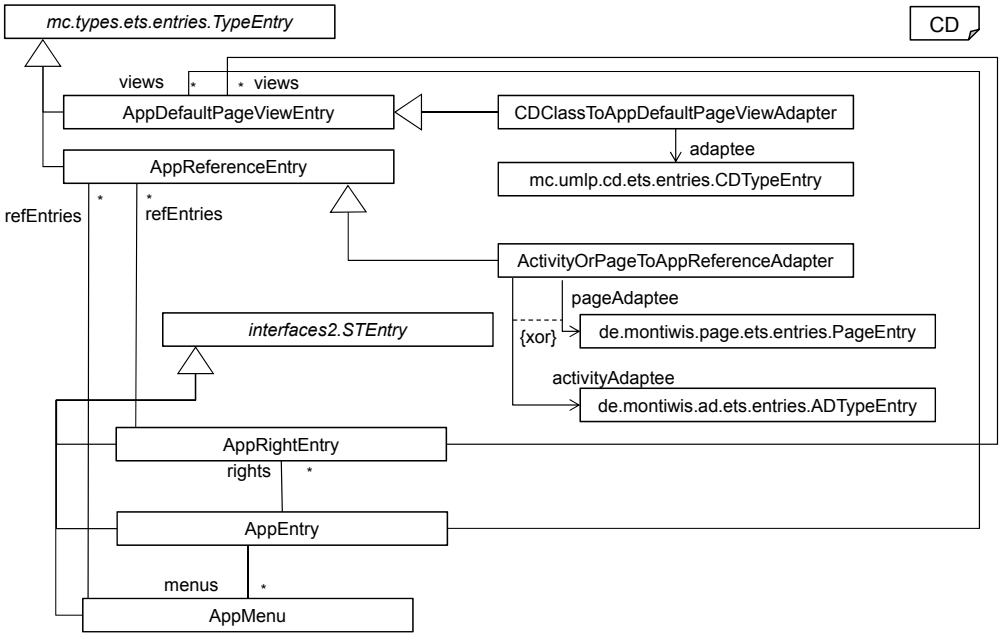


Abbildung 9.6.: Klassen der Symboltabellenstruktur für die Applikationssprache

Die Oberklassen der einzelnen Einträge sind von den Klassen `TypeEntry` bzw. `STEntry` abgeleitet, die beide Teil der Symboltabelleninfrastruktur sind (vgl. Abschnitt 3.3.2). Die Klasse `AppEntry` repräsentiert das Wurzelelement und dient damit als Äquivalent zur Applikationssprache selbst. Es beinhaltet eine Menge von `AppRightEntry`- und `AppMenu`-Objekten. Jedes dieser Objekte repräsentiert entweder ein Menü oder eine Rechtedefinition. Beiden ist gemein, dass sie eine Menge von `AppReferenceEntry`- und `AppDefaultPageViewEntry`-Objekten enthalten. Erstere repräsentieren sowohl Seiten als auch Aktivitäten, auf die im Menü bzw. in der Rechtedefinition verwiesen wird, Letztere hingegen stehen für die Standardseiten zur Bearbeitung und Auflistung von Objekten eines Typs. Da sowohl in einem Menü als auch in den Rechtedeklarationen lediglich Verweise auf andere Artefakte des Web-Informationssystems spezifiziert werden, sind auf Ebene der Symboltabelle *Ad-*

apter nötig, die die entsprechenden Verknüpfungen herstellen. Im Fall der Standardseiten zur Bearbeitung von Objekten wird von der Klasse `AppDefaultPageViewEntry` die Klasse `CDClassToAppDefaultPageViewAdapter` abgeleitet. Diese besitzt eine Referenz zum zugehörigen `CDTypeEntry`-Objekt, sofern die Qualifizierung der Einträge erfolgreich war. Für den Fall von gewöhnlichen Seite oder Aktivitäten wird von der Klasse `AppReferenceEntry` die Klasse `ActivityOrPageToAppReferenceAdapter` abgeleitet. Da dieser Eintrag sowohl auf eine Seite als auch eine Aktivität verweisen kann, existieren darin je ein Eintrag auf einen `PageEntry` sowie einen `ADTypeEntry`, die je nach Art der tatsächlichen Referenz entsprechend aufgelöst und belegt werden.

Korrekte Referenz von Aktivitäten, Seiten und Standardseiten

Die einzige Konsistenzbedingung innerhalb der Applikationssprache ist die korrekte Referenz von Aktivitäten, Seiten und Standardseiten. Hierbei wird sichergestellt, dass diese Artefakte in den Modellen definiert wurden. Die Überprüfung selbst findet direkt bei der Vollqualifizierung der Einträge aus dem vorherigen Abschnitt statt. Aktivitäten und Seiten werden direkt aufgelöst und so deren Existenz überprüft. Bei den Standardseiten werden die Einträge der Klassen im Domänendatenmodell aufgelöst und die entsprechenden Einträge in den Adaptern gesetzt.

9.3.2. Aufbau der Zwischenstruktur

Wie in Abschnitt 4.4 erläutert, wird in MontiWIS eine Zwischenstruktur zur Weiterverarbeitung der Modelle verwendet. Auch für die Applikationssprache wird eine solche benutzt. Die Klassenstruktur ist in Abbildung 9.7 dargestellt.

Speicherung der Menüstruktur

Im oberen Teil der Abbildung ist die Zwischenstruktur dargestellt, die für die Menübeschreibung verwendet wird. Die Struktur folgt dem *Composite*-Muster [GHJV95], über das die hierarchische Menüstruktur abgebildet wird. Von der abstrakten Klasse `MenuComponent` werden die beiden konkreten Klassen `Menu` und `MenuItem` abgeleitet. Erstere stellt analog zum Muster das Kompositum dar, welches seinerseits wiederum eine Liste von `MenuCompontents` enhalten kann, weswegen darin etwa auch der `Iterator` implementiert ist. `MenuItem` hingegen ist ein Blatt im Baum, welches den eigentlichen Verweis auf Aktivitäten, Seiten oder Standardseiten sowie Informationen über die Hierarchiestufe, auf der sich das Element befindet, enthält. Ferner besitzen all diese Klassen die Möglichkeit, die Bezeichnung des Verweises bzw. des Menüs zu speichern.

Speicherung der Rechte

Im unteren Teil der Abbildung ist die Zwischenstruktur zur Repräsentation von Artefakten und den festgelegten Rechten darauf dargestellt. Die Schnittstelle `RightAssociation` definiert boolesche Funktionen zur Unterscheidung der Inhalte (z. B. `isActivityRightAssociation`)

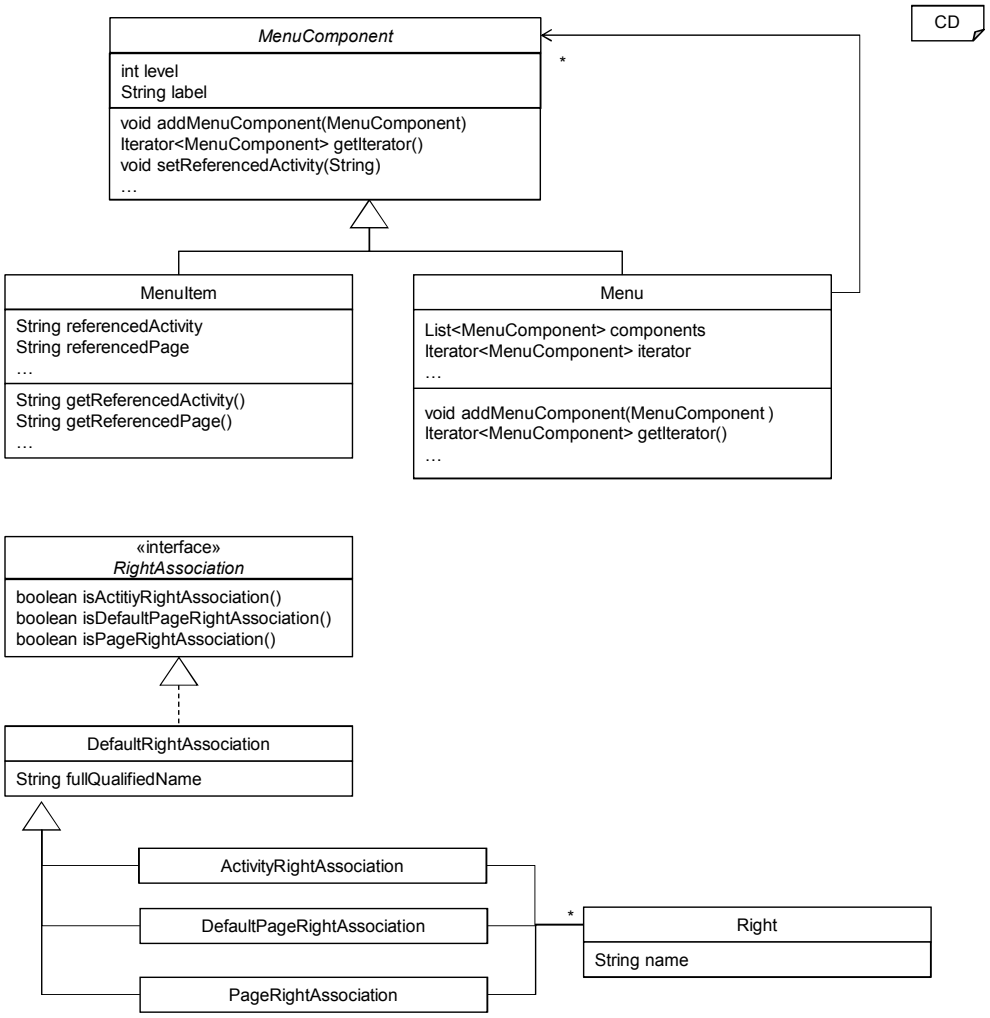


Abbildung 9.7.: Klassen der Zwischenstruktur für die Applikationssprache

sowie zum Abrufen von u. a. vollqualifizierten Namen (`fullQualifiedName`) des jeweiligen Artefakts. Davon abgeleitet ist die Klasse `DefaultRightAssociation`, die für sämtliche booleschen Funktionen zunächst `false` zurückgibt. Jede davon abgeleitete Klasse dient zur Abbildung eines Typs von Artefakten, für die jeweils Rechte zugeordnet werden können. Die Klasse `ActivityRightAssociation` dient hierbei zur Repräsentation von Aktivitäten, über `DefaultPageRightAssociation` werden Rechte auf Standardseiten gespeichert und die Klasse `PageRightAssociation` schließlich dient zur Abbildung von Rechten auf gewöhnliche Seiten. Ein Recht selbst wird über die Klasse `Right` spezifiziert, wovon einem Artefakt eine beliebige Anzahl zugeordnet werden können.

Speicherung von Menüs und Rechten im ApplicationGlobalStore

Die Objekte der Zwischenstruktur werden auch bei der Applikationssprache in der zentralen Klasse `DefaultApplicationGlobalStore` gespeichert, die die Schnittstelle `ApplicationGlobalStore` implementiert und, da durch die Annotation `@Singleton` ausgezeichnet, als zentraler Speicher lediglich ein einziges Mal in der Applikation existiert. Einen Überblick darüber gibt Abbildung 9.8.

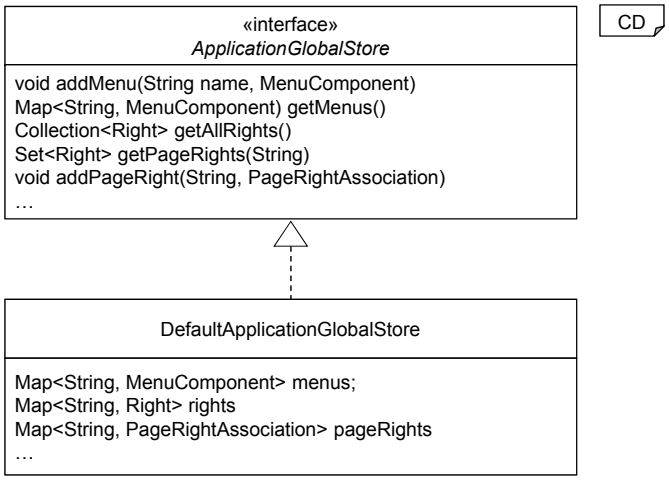


Abbildung 9.8.: Zentraler Speicher für die Applikationssprache

Diese Klasse besitzt Methoden zum Hinzufügen von Menüs (`addMenu`), die unter ihrem Namen abgespeichert werden. Ferner lassen sich Rechte sowohl anhand des vollqualifizierten Namens des Artefakts (z. B. `addPageRight`) als auch unterschieden nach Namen des Rechts abspeichern und abrufen, wobei jeweils hinsichtlich des Artefakttyps unterschieden wird (z. B. `getPageRights`). Schließlich existieren Funktionen zum Abrufen von sämtlichen Rechten eines bestimmten Typs, um einfaches Iterieren darüber zu ermöglichen. Die Speicherung der Objekte findet in der Klasse `DefaultApplicationGlobalStore` über `Maps` statt, wie ebenfalls in Abbildung 9.8 dargestellt.

9.3.3. Modellverarbeitung durch einen Visitor

Die Objekte der Zwischenstruktur werden bei der Bearbeitung des AST der Modelle durch die Klasse `ApplicationTransformatorConcreteVisitor` erstellt und der globalen Speicherstruktur hinzugefügt.

Mit der Bearbeitung der `ASTApplicationDefinition` beginnt wie bei allen Durchläufen des Visitors die Verarbeitung der AST-Strukturen. Hierbei wird über sämtliche `ASTApplicationElement`-Objekte iteriert und für jedes dieser Elemente die `ownVisit`-Methode aufgerufen.

Für den Aufbau der Menüstruktur werden zunächst die `ASTMenu`-Objekte bearbeitet. Hierfür wird ein `Menu`-Objekt erzeugt und mit den notwendigen Attributwerten wie dem Namen und

der Hierarchiestufe 1 belegt. Nachfolgend werden, wie bereits erwähnt, die Kindelemente bearbeitet. Im Falle eines ASTLabelEntrys wird ebenfalls ein Menu-Objekt angelegt, da auf ein Label stets Kindelemente des Menüeintrags folgen. Auch hier werden Attribute, Bezeichnung und Hierarchiestufe gesetzt. Da bei den ASTActivityOrPageReferenceEntrys nicht klar ist, welche Art von Artefakt zu dem angegebenen Namen gehört, wird hier per Symboltabelle der entsprechende Symboltabelleneintrag ermittelt. Darauf basierend werden die Werte des neu angelegten MenuItem-Objekts gesetzt, was sowohl Namen, Typ der Referenz als auch Hierarchiestufe umfasst. Auch bei der Bearbeitung der ASTImplicitViewEntrys wird per Symboltabelle der zugehörige Eintrag der referenzierten Klasse ermittelt und darüber Informationen wie vollqualifizierte Namen abgerufen und die entsprechenden Werte im MenuItem-Objekt gesetzt. All diese MenuItems werden dem übergeordneten Menu hinzugefügt.

Bei der Bearbeitung der ASTRight-Objekte wird zunächst ein Right-Objekt angelegt, welches das aktuell bearbeitete Recht repräsentiert. Dies sind beispielsweise Rechte wie guest, admin oder dergleichen. Bei der Iteration über die ASTRunnable-Objekte, die entweder gewöhnliche Seiten, Standardseiten oder Aktivitäten repräsentieren, wird ebenfalls die Symboltabelle bemüht, um das zugehörige Artefakt zu ermitteln. Auf Basis dieses Ergebnisses wird entweder ein ActivityRightAssociation- oder ein PageRightAssociation-Objekt erstellt, bei dem die entsprechenden Attribute gesetzt werden. Zusätzlich wird das Right-Objekt hinzugefügt. Falls es sich bei dem zugehörigen Recht um eine Standardseite handelt, werden über den Symboltabelleneintrag der Klasse die notwendigen Informationen ermittelt und im erstellten DefaultPageRightAssociation-Objekt gesetzt. Schließlich wird auch hier das zugehörige Right-Objekt assoziiert.

9.3.4. Rekonstruktion

Der Rekonstruktionsprozess gestaltet sich bei der Applikationssprache relativ einfach. In der Klasse ApplicationReconstructor wird lediglich die Methode replaceMenuItemNamesWithShorterOnes aufgerufen. Darin wird über sämtliche MenuComponent-Objekte iteriert und in den MenuItem-Objekten die Namen der Artefakte durch die tatsächlichen Namen aus den jeweiligen Zwischenstrukturelementen ersetzt. Dies geschieht aufgrund der Tatsache, dass beim Durchlauf des Visitors zwar die Symboltabellenstruktur bereits aufgebaut ist, dort jedoch primär die vollqualifizierten Namen der Artefakte verfügbar sind. Diese müssen nicht zwangsläufig mit denen übereinstimmen, über die die Artefakte im generierten System aufgerufen werden. Diese Information ist zuverlässig erst nach Verarbeitung der übrigen Elemente durch ihre Rekonstruktoren verfügbar, weswegen diese Informationen auch erst in diesem (recht späten) Schritt ersetzt werden.

9.3.5. Generator

Während des Generierungsprozesses, der aus der Zwischenstruktur die entsprechende Funktionalität im Web-Informationssystemen umsetzt, werden zum einen separate Dateien erzeugt, in denen die Informationen aus der Applikationssprache verarbeitet werden, zum anderen jedoch auch Informationen in Artefakten wie beispielsweise Ressourcenklassen verankert,

die die modellierte Funktionalität umsetzen. Die Steuerung des Generationsprozesses übernimmt die Klasse `de.montiwis.combined.generator.CombinedGenerator`. Die Templates, die beim Generierungsprozess verwendet werden um explizit Seiten(teile) bzw. weitere zugehörige Dateien zu generieren, sind inklusive einer kurzen Beschreibung in Tabelle 9.9 aufgelistet.

Template-Datei	Erläuterung
`jsp/LoginPage.jsp.ftl`	Erzeugt die JSP-Loginseite
`jsp/Logout.jsp.ftl`	Erzeugt die JSP-Logoutseite
`jsp/LoginError.jsp.ftl`	Erzeugt die Seite für fehlgeschlagenen Login
`conf/langstring_resources.properties.ftl`	Erzeugt die Ressourcendatei für mehrsprachige Zeichenketten
`conf/sun-web.xml.ftl`	Erzeugt die Konfigurationsdatei inkl. Benutzerrollen
`conf/web.xml.ftl`	Erzeugt die Konfigurationsdatei inkl. Benutzerrollen, Authentifizierungsmechanismen und Webcontainerkonfiguration
`jsp/Menu.jsp.ftl`	Erzeugt das Menü der Applikation
`jsp/ResourceNotAllowed.jsp.ftl`	Erzeugt die Seite für nicht erlaubten Zugriff auf eine Seite
`jsp/RunningActivities.jsp.ftl`	Erzeugt die Seite zur Anzeige von aktuell laufenden Aktivitäten

Tabelle 9.9.: Die Template-Dateien der Applikationssprache und ihre Bedeutung

Nicht alle in Tabelle 9.9 gelisteten Templates werden direkt Informationen aus den Modellen der Applikationssprache verarbeiten, sondern betreffen zum Teil auch Generate, die inhaltlich den Berechtigungen zuzuordnen sind. Die Templates und die Generierung werden in den nachfolgenden Abschnitten erläutert.

Generierung des Menüs

Die Menüstruktur, die im Modell der Applikationssprache spezifiziert wurde, wird durch das Template `Menu.jsp.ftl` verarbeitet. Das Ergebnis der Generierung ist JSP-Code, der in jede generierte Seite eingebunden wird. Diesem Template wird lediglich ein `ApplicationGlobalStore`-Objekt übergeben, was die Zwischenstruktur sowohl für das Menü als auch

die Rechtedefinition enthält. Im Template selbst wird zur Generierung des Menüs über die dazugehörige Zwischenstruktur iteriert. Dabei wird jeweils unterschieden, zu welchem Typ das aktuell bearbeitete `MenuComponent`-Objekt gehört und entsprechend darauf reagiert.

Bei `Menu`-Objekten werden zunächst Abschnitte erzeugt, die die Zugriffsrechte sowohl von direkten Kindelementen als auch tiefer angesiedelten Kindelementen überprüfen. Darüber wird im generierten Code festgestellt, ob ein bestimmtes Menü angezeigt werden soll, was nur dann der Fall ist, wenn der Nutzer auf zumindest auf eines der enthaltenen Kindelemente Zugriff hat. Die Überschrift selbst enthält entweder die direkte Bezeichnung des Menüs oder aber eine Referenz auf die Ressourcendatei mit mehrsprachigen Ausdrücken. Dieser Ablauf wird analog für Menüs auf jeder Hierarchiestufe durchlaufen.

Ebenfalls auf jeder Hierarchiestufe werden die Menüeinträge generiert, die die Referenzen auf Seiten, Standardseiten oder Aktivitäten enthalten. Auch hier wird um jedes dieser Elemente ein Block mit Anweisungen generiert, der bei der Ausführung des JSP-Abschnitts überprüft, ob der aktuelle Nutzer Zugriff auf das jeweilige Element hat. Ferner werden die Bezeichnungen (ebenfalls in zweierlei Art, wie soeben beschrieben) und Verweise auf die Artefakte selbst generiert.

Generierung der Berechtigungsmechanismen

Die Generierung der Berechtigungsmechanismen findet an ganz unterschiedlichen Stellen in MontiWIS statt. Aufgrund der Vielzahl von Templates, die davon betroffen sind, wurden sie nicht explizit in der Tabelle 9.9 mit aufgeführt, sondern werden im Rahmen der Beschreibung dieses Abschnitts genannt und erläutert. Da die hier beschriebenen Templates in der Regel bereits an anderer Stelle beschrieben wurden, sei hier nur kurz auf die jeweils relevante Stelle eingegangen. Die Art und Weise, wie die Überprüfung der Zugriffsrechte letztlich stattfindet, wird in Abschnitt 9.4.2 erläutert.

Im Template `activity.ftl`, das für jede Aktivität in der Zwischenstruktur verwendet wird, um eine Aktivität im generierten System zu erzeugen, werden die Rechte in der Funktion `getAllowedRolesString` in der Art zusammengesetzt, dass sie dort von anderen Klassen abgefragt werden können. Das Template selbst wurde bereits in Abschnitt 8.4.5 erläutert.

Da für jede Seite in MontiWIS eine eigene Ressourcenklasse erzeugt wird, werden auch im Rahmen ihrer Generierung die Berechtigungen berücksichtigt. Hierfür wird im Template `ResourceFileForPage.ftl` die Liste der zugriffsberechtigten Rollen in den Aufruf zur Funktion, die die Überprüfung durchführt, übernommen. Ähnliches geschieht im Template `ResourceFileForEntity.ftl`, welches zur Generierung der Ressourcenklassen für die Standardseiten verwendet wird. Diese Templates wurden bereits in den Abschnitten 7.3.5 bzw. 6.3.6 erläutert.

Schließlich findet ebenfalls in den Seiten selbst eine Überprüfung statt, ob bestimmte Verweise bzw. bestimmte Funktionalität zur Verfügung stehen soll. Daher werden in den Templates `listEntity.ftl` und `table.ftl` Abfragen mit den entsprechenden Rechten generiert, die beispielsweise überprüfen, ob ein Nutzer Bearbeitungsrechte auf eine bestimmte Klasse besitzt, so dass selektiv Verweise zum Bearbeiten oder Löschen angezeigt oder ausgeblendet werden.

Generierung von statischen Seiten

Neben der Menüstruktur werden im Rahmen des `CombinedGenerators` ebenfalls einige statische Seiten generiert, die jedoch inhaltlich zu der Funktionalität der Applikationssprache gehören. Für die Erzeugung von Login- und Logoutseiten werden die Templates `LoginPage.jsp.ftl` sowie `Logout.jsp.ftl` verwendet. In beiden Seiten werden sowohl die üblichen `header.ftl` und `footer.ftl` Templates eingebunden, die die gemeinsamen Teile einer JSP-Seite erzeugen (vgl. hierzu Abschnitt 7.3.5). Auf der Loginseite werden lediglich die benötigten Formularelemente erzeugt, die an die zugehörige Serverseite gesendet werden, auf der Logoutseite wird der Logout vollzogen und auf die Hauptseite der Applikation weitergeleitet. Die durch das Template `LoginError.jsp.ftl` erzeugte Fehlerseite enthält lediglich die Ausgabe einer Fehlermeldung und einen Verweis zur Loginseite. Die Seite, die auf nicht ausreichende Rechte hinweist, wird durch das Template `ResourceNotAllowed.jsp.ftl` generiert und beinhaltet besagte Meldung und einen Link zur Hauptseite. Schließlich enthält die Seite, die über das Template `RunningActivities.jsp.ftl` erzeugt wird, einen Hinweis zur Auswahl einer Aktivität sowie eine Tabelle, die beim Aufruf der Seite dynamisch über JavaScript-Anweisungen mit Inhalt befüllt wird.

Generierung von Konfigurationsdateien

Ähnlich zu den statischen Seiten, die im vorherigen Abschnitt beschrieben wurden, lassen sich auch die Konfigurationsdateien für den Server, auf dem die Applikation ausgeführt wird, inhaltlich zur Applikationssprache rechnen.

Über das Template `langstring_resources.properties.ftl` werden sämtliche Schlüssel-Wert-Paare, die während der Modellverarbeitung oder aber auch während der Rekonstruktion gesammelt wurden, in einer Datei zusammengefasst. Bei der Generierung wird dem Template ein Objekt des `CombinedGlobalStores` übergeben, da dort die benötigten Informationen abgelegt wurden. Die Generierung unterscheidet sich insofern von anderen Dateien, als hier bei der ersten Ausführung zwei Dateien erzeugt werden, die den gleichen Inhalt besitzen. Die Datei `langstring.properties` wird lediglich beim ersten Durchlauf der Generierung erzeugt. In ihr können potenziell Änderungen an den Bezeichnungen vorgenommen werden, die bei weiteren Durchläufen des Generators nicht überschrieben werden dürfen. Damit dennoch potenziell weitere Schlüssel-Wert-Paare aufgenommen werden können, wird in jedem Fall die zweite Datei mit dem Namen `langstring.properties_will_be_overwritten` generiert. Darin sind sämtliche (alten und) neuen Werte enthalten, die nachfolgend manuell in die neue Datei übernommen werden können.[29]

Das Template `sub-web.xml.ftl` wird zur Generierung der `sun-web.xml` Datei verwendet, die zur Konfiguration des Servers nötig ist. Diesem Template werden neben Informationen wie dem Projektnamen die Menge der `Right`-Objekte übergeben. Diese werden innerhalb des Templates verwendet, um die Rechte in der Konfigurationsdatei als Loginrolle anzulegen.

Ähnlich verhält es sich mit dem Template `web.xml.ftl`. Auch ihm wird die Menge von `Right`-Objekten übergeben, damit darüber im Template die nötigen Einträge generiert werden

[29] Hierzu ist beipsielsweise ein Texteditor hilfreich, der Unterschiede zwischen Dateien deutlich machen kann.

können. Bis auf den Projektnamen, der zur Erzeugung eines eindeutigen Namens für die Datenbankverbindung benötigt wird, enthält das Template darüber hinaus lediglich statische Anteile zur Serverkonfiguration.

Generierung der Initalisierungsdatei für Berechtigungen und Standardnutzer

In MontiWIS werden für eine korrekte Funktionsweise des Berechtigungsmechanismus in der Datenbank Einträge benötigt, die bei erster Initialisierung des Web-Informationssystems angelegt werden müssen. Diese werden ebenfalls als Teil der Generierung im Rahmen der Applikationssprachbearbeitung generiert, was über das Template `UserRoleInitializer.java.ftl` geschieht. Darin werden für jedes `Right`-Objekt zugehörige Objekte in der Sprache der Zielplattform erzeugt und ebenso Standardnutzer angelegt, denen die entsprechenden Rechte zugeordnet werden.

Bei der Generierung dieses Codes werden zwei Szenarien unterschieden. Abhängig vom Schalter `local_deploy`, der in der Konfigurationsdatei gesetzt wird, werden unterschiedliche Arten der Speicherung in der Zielstruktur generiert. Falls dieser Parameter `true` ist, wird Code generiert, der die Speicherung über ein `DAO` vornimmt. Anderenfalls erfolgt die Speicherung direkt per `EntityManager`, der auf eine potenziell entfernt laufende Datenbank zugreift. Letzteres ist notwendig, weil die `DAO`s nicht von entfernten Rechnern aus zugreifbar sind, direkte Datenbankverbindungen hingegen in der Regel schon.

9.4. Umsetzung im generierten System

Nach der Beschreibung, wie die Generierung im Einzelnen abläuft, wird im folgenden Abschnitt beschrieben, wie die einzelnen Aspekte in der generierten Webapplikation umgesetzt wurden. Zunächst wird in Abschnitt 9.4.1 erläutert, wie das Menü im generierten Code implementiert wurde. Anschließend wird die Umsetzung der Berechtigungen an verschiedenen Stellen innerhalb der Applikation in Abschnitt 9.4.2 erläutert.

9.4.1. Umsetzung des Menüs

Wie bereits in Abschnitt 9.3.5 erläutert, wird das Menü des Web-Informationssystems als JSP-Ausschnitt umgesetzt. Ein Teil des Codes, der aus dem Modell in Abbildung 9.2 generiert wurde, ist in Abbildung 9.10 dargestellt.

JSP

```
1 <div id="navigation">
2   <div class="accordion">
3     <c:if test="${m:isUserInRoles(pageContext, 'guest,admin,user,guest,')}">
4       <h3 id="menu1"><a href="#"> Pages </a></h3>
5       <ul>
6         <c:if test="${m:isUserInRoles(pageContext, 'guest,admin,user,')}">
7           <li><a href="${pageContext.request.contextPath}/resources/page/Index?menuid=↩
                menu1">
```

```
8               Index </a></li>
9         </c:if>
10
11        <c:if test="${m:isUserInRoles(pageContext, 'admin,user,')}">
12          <li><a href="${pageContext.request.contextPath}/resources/page/Logout?menuid=↩
                menu1">
13              Logout </a></li>
14        </c:if>
15      </ul>
16    </c:if>
17
18    <c:if test="${m:isUserInRoles(pageContext, 'admin,user,admin,user,')}">
19      <h3 id="menu2"><a href="#"> Student </a></h3>
20      <ul>
21        <c:if test="${m:isUserInRoles(pageContext, 'admin,user,')}">
22          <li><a href="${pageContext.request.contextPath}/resources/Student/list?menuid=↩
                menu2">
23              List </a></li>
24        </c:if>
25
26        <c:if test="${m:isUserInRoles(pageContext, 'admin,user,')}">
27          <li><a href="${pageContext.request.contextPath}/resources/Student/new?menuid=↩
                menu2">
28              New </a></li>
29        </c:if>
30      </ul>
31    </c:if>
32
33    <c:if test="${m:isUserInRoles(pageContext, 'guest,user,admin,')}">
34      <h3 id="menu6"><a href="#"> Activities </a></h3>
35      <ul>
36        <c:if test="${m:isUserInRoles(pageContext, 'guest,')}">
37          <li><a href="javascript:void(0)" onclick="javascript:sendPost('${pageContext.↩
                request.contextPath}/resources/Activity/register?menuid=menu6')">
38              Register </a></li>
39        </c:if>
40      </ul>
41    </c:if>
42  </div>
43 </div>
```

Abbildung 9.10.: Umsetzung des Menüs

Wie dort zu sehen ist, wird das Menü in zwei `div`-Elementen definiert (Zeilen 1-2). Das erste Element (mit der `id` `navigation`) dient zur Positionierung des Menüs, das zweite Element (mit der `id` `accordion`) wird verwendet, um beim Start der Seite per JavaScript die Funktionalität für das Menü hinzuzufügen. Dazu wird die *Accordion*-Funktionalität verwendet, die ein Teil der JQueryUI [JQuc] ist. Daraus resultiert auch der im folgenden beschriebene Aufbau, der für die korrekte Funktionsweise dieses Konstrukts obligatorisch ist. Im nächsten Schritt wird überprüft, welche Nutzergruppen Zugriff auf das folgende Untermenü haben sollen. Dazu

wird die für MontiWIS definierte Expression Language (EL)-Funktion `isUserInRole` (Zeile 3), das dem Namespace mit dem Prefix `m` und der URL `http://montiwis.de/taglib` (nicht im Beispiel dargestellt) zugeordnet ist. In diesem Funktionsaufruf sind die Namen sämtlicher Rechte als Parameter enthalten, die in den untergeordneten Menüs und Menüeinträgen zum Zugriff berechtigen. Die genaue Funktionalität wird in Abschnitt 9.4.2 erläutert.

Ein Menü bzw. Untermenü wird spezifiziert über eine Überschrift (abgebildet über ein `h3` Element), gefolgt von einer ungeordneten Liste (über das Tag `ul`), das die einzelnen Menüeinträge (über das Tag `li`, eingeschlossen in einen Link über das Tag `a`) enthält (Zeilen 7-8). Für jeden dieser Einträge wird wiederum über ein `isUserInRole`-Tag geprüft, ob es für den aktuell eingeloggten Nutzer sichtbar sein soll, oder ob es im Menü ausgeblendet wird. Ein analoger Aufbau erfolgt für die restlichen gezeigten Menüs und Einträge (Zeilen 18-31 und 33-41).

Die Verweise zu gewöhnlichen Seiten sind in den Zeilen 7-8 bzw. 12-13 zu sehen. Hier werden einfache Links verwendet, um die Seiten aufzurufen. Ebenfalls normale Verweise werden beim Aufruf der Standardseiten für Objekte einer bestimmten Klasse verwendet. Diese sind in den Zeilen 22-23 bzw. 27-28 dargestellt. Anders verhält es sich mit Verweisen, die eine neue Aktivität starten, wie dies in den Zeilen 37-38 zu sehen ist. Hier wird, analog zum REST-Paradigma, eine HTTP `POST`-Abfrage abgesandt, was durch den Aufruf der JavaScript-Funktion `sendPost` geschieht.

In all diesen Verweisen wird zusätzlich zur eigentlichen Adresse der Parameter `menuid` übertragen, der angibt, in welchem Menü der abgesandte Eintrag zu finden war. Dessen Wert wird bei Versand der Antwortseite eingebunden, so dass über eine JavaScript-Funktion das entsprechende Menü wieder geöffnet werden kann.

9.4.2. Umsetzung der Rechte

Die Implementierung der Berechtigungen in MontiWIS basiert grundsätzlich auf den Mechanismen, die der Applikationsserver bietet. Dieses grundlegende Konzept wird ebenso in den folgenden Abschnitten erläutert wie der Mechanismus, wie die Gruppenzugehörigkeit abgefragt wird sowie die Stellen in der Applikation, in denen diese zum Tragen kommt.

Gruppenzuordnung im Applikationsserver

Die Berechtigungen in MontiWIS werden generell über den Mechanismus realisiert, den der Glassfish Applikationsserver bzw. das Java EE Framework über JDBC Realms bereitstellt [Gla, Hef10]. Bei den gewöhnlichen Möglichkeiten, die eine JDBC Ressource bietet, sind jedoch einige Defizite zu erkennen, die die Einsatzmöglichkeiten sehr stark einschränken würden. Aus diesem Grund wird auf die Erweiterung *Flexible JDBC Realm* [Fle] zurückgegriffen, die hier wesentlich mehr Möglichkeiten bietet. Einerseits lassen sich darüber quasi beliebige Datenstrukturen zur Abbildung von Gruppenzugehörigkeiten verwenden, die hinsichtlich der möglichen Flexibilität weit über das hinausgehen, was der gewöhnliche JDBC Realm bietet. Nutzern können so im laufenden Betrieb Gruppenberechtigungen zugeordnet und entzogen werden. Ferner ist die Struktur von Datenbank und Tabellen wesentlich flexibler, was für

die freie Verknüpfung von Nutzerklassen zu weiteren Klassen im Datenmodell eines Web-Informationssystems obligatorisch ist.

Wie in Abschnitt 6.2.4 erläutert, werden die Gruppen, denen ein Nutzer zugeordnet werden kann, bereits im Klassendiagramm über die Klasse `GroupInternal` repräsentiert. Ebenso wird in Abschnitt 8.5.1 erläutert, dass die Klasse `WorkflowUser` verwendet wird, um Nutzername und Passwort zu speichern. Diese beiden Klassen in Kombination werden auch für die Konfiguration des Flexible JDBC Realms herangezogen, dessen Parameter in Tabelle 9.11 gelistet bzw. erläutert werden.

Parameter	Wert bzw. Erläuterung
`jaas.context`	Name des Realms, der in der `login.conf` Datei des Glassfish Servers festgelegt wurde
`datasource.jndi`	Name der JDBC Resource, über die die Verbindung zur Datenbank hergestellt wird
`sql.password`	`SELECT PASSWORD FROM WORKFLOWUSER WHERE (USERNAME = ?)`
`password.digest`	`MD5`
`sql.seed`	`SELECT SALT FROM WORKFLOWUSER WHERE (USERNAME = ?)`
`sql.groups`	`SELECT GROUPNAME FROM GROUPINTERNAL, GROUPIN-TERNAL_GROUPS, WORKFLOWUSER WHERE GROUPINTERNAL.ID = GROUPINTERNAL_GROUPS.GROUPS_ID AND WORKFLOWUSER.ID = GROUPINTERNAL_GROUPS.USERS_ID AND (WORKFLOWUSER.USERNAME =?)`

Tabelle 9.11.: Konfigurationsparameter des Flexible JDBC Realms

Die in der Tabelle angegebenen Parameter sind bei der Konfiguration des Realms anzugeben, was entweder direkt über die Web-Konfigurationsoberfläche des Glassfish Servers, das Kommandozeilenwerkzeug `asadmin` oder über die in Abschnitt 10.2 beschriebene Builddatei geschehen kann, in der ebenfalls das `asadmin`-Werkzeug zum Einsatz kommt.

Der Parameter `jaas.context` korrespondiert mit dem Namen des Realms, der in der Konfigurationsdatei `login.conf` im Glassfish Server eingetragen wurde. In dieser Datei wird festgelegt, welche Klasse für die Funktionsfähigkeit des Realms mit dem entsprechenden Namen verantwortlich ist, womit im Wesentlichen die Art des Realms spezifiziert wird. Über den Konfigurationsparameter `datasource.jndi` wird definiert, über welche im Glassfish Server konfigurierte Datenbankverbindung die Abfragen gesandt werden, die zur Ermittlung von Nutzerinformationen und Gruppenzugehörigkeit benötigt werden. Die Abfragen, wie diese Informationen abzurufen sind, werden in den nachfolgenden Parametern konfiguriert. Im Parameter `sql.password` wird festgelegt, wie das Passwort zu einem gegebenen Nutzernamen abgefragt werden kann. Dies geschieht über den Zugriff auf die Tabelle `WORKFLOWUSER`, in der die Objekte der Klasse `WorkflowUser` gespeichert werden. Als Zusatz zu dieser Abfrage wird

im Parameter `password.digest` spezifiziert, welcher Hashalgorithmus zur Verschleierung des Passworts verwendet wird und im Parameter `sql.seed` wird zudem definiert, wie das sog. *Salt* abgefragt werden kann, welches zusätzlich zum Password in der Hashfunktion berücksichtigt wird. Letzteres ist ebenfalls in der `WORKFLOWUSER`-Tabelle gespeichert. Eine Erläuterung der Sicherheitsmechanismen in MontiWIS ist in Abschnitt 7.4.5 zu finden. Der letzte notwendige Parameter dient zur Ermittlung der Gruppen, denen ein Nutzer mit einem bestimmten Nutzernamen zugeordnet ist. Die hier verwendete Abfrage ruft über die Verknüpfung der Tabellen für Nutzer (in diesem Fall `WORKFLOWUSER`) und Gruppen (hier `GROUPINTERNAL`) über die Tabelle `GROUPINTERNAL_GROUPS` die Namen der Gruppen ab, denen der Nutzer mit dem gegebenen Nutzernamen zugeordnet ist.

Die hier beschriebenen Abfragen werden innerhalb des *Flexible JDBC Realms* dazu verwendet, die in der Datenbank gespeicherten Daten auf die Zugriffsmechanismen des Glassfish Servers abzubilden.

Abfrage der Gruppenzugehörigkeit

Der Zugriff auf die Funktionalität, die in Java EE Applikationen zur Verfügung steht um festzustellen, ob ein Nutzer Mitglied einer Nutzergruppe ist, ist in MontiWIS gekapselt in eine JSP EL Funktion in der Klasse `de.montiwis.taglib.IsUserInRolesTag`, die Teil des `montiwis-target` Projekts ist. Darin existieren zwei überladene Methoden. Beide benötigen einen `String`, der eine Liste mit den zusammengesetzten Rollen, getrennt durch Kommata, enthält. Als zweiten Parameter wird entweder ein `javax.servlet.jsp.PageContext` Objekt oder ein `javax.servlet.http.HttpServletRequest` Objekt benötigt. Ersteres resultiert aus der Verwendung der Methode als EL Funktion, zweiteres kommt etwa im Kontext von Ressourcenklassen zum Einsatz. In beiden Funktionen wird über die übergebenen Gruppennamen iteriert und per Funktion `isUserInRole` des übergebenen bzw. aus dem `PageContext` abgefragten `HttpServletRequest` geprüft, ob der aktuelle Nutzer einer der Gruppen zugeordnet ist. Die Informationen, ob eine entsprechende Rollenzuordnung existiert, werden intern über die Abfragen aus Tabelle 9.11 realisiert.

Verwendung der Gruppenabfrage in der Webanwendung

Wie im vorherigen Abschnitt erläutert, existieren zwei Methoden, die zur Abfrage der Gruppenzugehörigkeit herangezogen werden. Deren konkrete Verwendung wird nachfolgend kurz erläutert.

Zunächst erfolgt die Abfrage, wie in Abbildung 9.10 mehrfach dargestellt, über die EL Funktion `isUserInRoles`. Darin wird zum einen der `PageContext`, der per Variable in der JSP-Seite verfügbar ist, zum anderen die Liste mit erlaubten Nutzergruppen übergeben. Der Aufruf dieser Funktion erfolgt in einem JSTL `if` Tag, worüber ermittelt wird, ob der Inhalt des Abschnitts (in diesem Fall ein bestimmtes (Unter-)Menü) angezeigt werden soll. Ähnliche Konstrukte kommen an Stellen zum Einsatz, wo Verweise zu den Standardseiten eingeblendet werden, sofern ein Nutzer das entsprechende Recht dazu besitzt (vgl. Tabelle 9.5 sowie die Erläuterungen dazu).

Die zweite Stelle, wo entsprechende Abfragen hinsichtlich Nutzerberechtigungen verwendet werden, ist beispielsweise in den Ressourcenklassen (siehe Abbildungen 8.46, 7.44 sowie 8.44). Der Funktion `isUserInRoles` der Klasse `IsUserInRolesTag` wird das `HttpServletRequest`-Objekt übergeben, das vorher per `@Context`-Annotation in der Ressourcenklasse injiziert wurde (vgl. Abbildung 7.44, Zeilen 4-5). Die Zeichenkette mit den Gruppen, denen an einer bestimmten Stelle der Zugriff erlaubt ist, wurde im Falle der `ContactResource` in Abbildung 7.44 direkt in die Klasse generiert, da diese Klasse explizit für eine bestimmte Seite erzeugt wurde und damit bereits zum Zeitpunkt der Generierung feststand, welche Zugriffsbeschränkungen gelten. Im Fall von `ActionResource` bzw. `ActivityResource` Klassen (siehe Abbildungen 8.46 bzw. 8.44) wird die Zeichenkette per Delegation über den `ActionExecutor` bzw. `ActivityExecutor` letztlich aus den generierten `Activity` Klassen abgerufen, da innerhalb der Applikationssprache für Aktivitäten die Berechtigungen festgelegt wurden und dementsprechend bei der Generierung mit berücksichtigt werden können.

9.5. Andere Ansätze

Die in diesem Abschnitt beschriebenen Aspekte werden in vergleichbaren Ansätzen auf unterschiedliche Art und Weise spezifiziert und umgesetzt. Nachfolgend sollen ähnliche und abweichende Konzepte sowie deren Vor- und Nachteile gegenüber der in MontiWIS gewählten Lösung diskutiert.

9.5.1. Möglichkeiten der Zugriffssteuerung

Einen extensiven Überblick über verschiedene Arten der Zugriff- und Nutzersteuerung bietet [SdV01] sowie [SS94]. Hier werden verschiedene Konzepte vorgestellt, die Restriktionen unterschiedlicher Komplexität ermöglichen und auf ein formales Fundament gestellt. [JAGS01] beleuchtet die Eignung der beschriebenen Ansätze für den Einsatz in Webanwendungen. Den Fokus auf den Ansatz Role Based Access Control (RBAC), der auch in MontiWIS verwendet wird, legt [SCF+96]. Dort werden die Grundlagen dieses Konzepts erklärt, welches, im Gegensatz zur Umsetzung in MontiWIS, weitere Möglichkeiten wie beispielsweise hierarchische Rollen oder die Spezifikation von logischen Einschränkungen bei der Rollenzuweisung, bietet. Sehr komplexe formale Grundlagen für die Modellierung von Sicherheitsmechanismen mit Fokus auf die UML liefert UMLSec [Jür05]. SecureUML [LBD02] definiert ein UML-Profil für ein RBAC, das neben den Möglichkeiten, die MontiWIS bietet, auch die Definition von feingranularen Zugriffsrechten auf Basis von OCL erlaubt. Derartige Möglichkeiten liegen nach den Anforderungen außerhalb des Fokus von MontiWIS.

Hinsichtlich der Implementierung soll sich aufgrund der Vielzahl der möglichen Umsetzungen auf populäre Ansätze auf Basis von Java beschränkt werden. In der Java EE [Hef10, Sun09a] werden über die Security Realms im Wesentlichen die Mechanismen unterstützt, die (in erweiterter, flexiblerer Form) auch in MontiWIS zum Einsatz kommen. Alternativ dazu ließe sich ein Framework wie beispielsweise *Apache Shiro* [Shi] nutzen, was ebenfalls eine sehr flexible Möglichkeit zur Spezifikation von Rechten und Rollen in beliebigen Anwendungen erlaubt. Auch Spring [Spr] bietet Sicherheitsmechanismen, die jedoch eher

in das zugehörige Ökosystem integriert sind und sich relativ umständlich in Java EE nutzen lassen. Insgesamt wurde von der Verwendung anderer als der bereits in Java EE vorhandenen Mechanismen abgesehen, da die generative Entwicklung die Integration entsprechender Abfragen in konsistenter Weise über die Applikation verteilt ermöglicht.

OOWS [FPP+08] bietet über die user diagram models die Möglichkeit, verschiedene Nutzer auch hierarchisch unter der Verwendung von Vererbungsbeziehungen zu spezifizieren.

Ebenso ist in *WSDM* [DCP08] die Modellierung von (potenziell hierarchischen) Nutzerklassen und deren Zugriffsrechten explizit Teil des vorgeschlagenen Entwicklungsprozesses. Durch die mögliche Vererbung von Nutzerklassen bieten sich im Vergleich zu MontiWIS mehr Möglichkeiten, die für diesen Ansatz aber nicht für nötig erachtet wurden.

Die Mechanismen zur Zugriffsbeschränkung in *WebDSL* wurden in [GV08] beschrieben. Dabei lassen sich verschiedene Arten von Zugriffsrechten definieren, die sich sehr flexibel über grundlegende Konstrukte von WebDSL wie Datendefinitionen spezifizieren lassen. Dadurch wird eine erhebliche Ausdrucksmächtigkeit, aber einhergehend auch eine gewisse Komplexität erreicht, die zur Umsetzung von Zugriffsrechten nötig sind.

In *UWE* [KKZB08, Kro08] werden Nutzer in einem separaten User Model abgelegt, welches, ebenso wie das Domänendatenmodell, über ein Klassendiagramm codiert wird. Diese können einerseits ähnlich dem Domänendatenmodell, d. h. ohne separat dafür vorgesehene Konstrukte, abgefragt werden, andererseits aber auch aspektorientiert an dezidierten Stellen zur Adaption des Verhaltens mit einbezogen werden. Die Nutzerberechtigungen in UWE basieren im Wesentlichen auf einem RBAC-Modell, wobei die Rollen über Objektdiagramme definiert werden und Bedingungen sowohl an den Navigationszuständen als auch den Zustandsübergängen per Stereotyp angegeben werden [BKK11]. Darauf aufbauend lässt sich das Verhalten in bestimmten Angriffssituationen und explizite Sicherheitsmechanismen wie verschlüsselte Verbindungen spezifizieren [BKS14]. Derartige Konzepte sind in MontiWIS nicht vorhanden bzw. werden im Fall von verschlüsselten Verbindungen in die Hände des Anwendungsadministrators gelegt.

WebML bietet Usermodelle, die ähnlich MontiWIS die Zuordnung von Nutzern zu Nutzergruppen erlauben, um darüber den Zugriff auf bestimmte Aktionen und Seiten zur regeln [BCF06, BCFM08].

Für die Webanwendung in [OVD08] schlagen die Autoren explizit die Anwendung von RBAC auf die Beschränkung von Nutzerrechten bei den Übergängen zwischen verschiedenen Seiten einer Anwendung vor. Hierbei lassen sich Bedingungen festlegen, unter denen eine Nutzergruppe Berechtigungen zu bestimmten Seiten besitzt. Im Gegensatz zu MontiWIS lassen sich jedoch lediglich Seiten innerhalb einer Webanwendung sichern und keine beliebigen Aktivitäten bzw. komplexere Sachverhalte innerhalb eines Ablaufs, wie sie über Aktivitäten abgebildet werden können.

9.5.2. Methoden zur Spezifikation von Menüs in Webanwendungen

Die Spezifikation von Menüs wird von einem Großteil der Web-Modellierungsansätze unterstützt, wobei sich die Art der Spezifikation in die zur Verfügung stehenden Modelle eingliedert. Sie werden explizit modelliert oder aber aus der allgemeinen Navigationsstruktur abgeleitet.

In *OOWS* [FPP+08] beispielsweise werden die Menüeinträge aus den Navigationslinks erzeugt, wobei ebenfalls die Erreichbarkeit für bestimmte Nutzergruppen berücksichtigt wird.

Anders wird bei *OO-H* [KKCM04] verfahren. Hierbei werden über die Navigation Access Diagrams explizit Menüs und Verlinkungen spezifiziert, die die Navigationsmöglichkeiten bestimmen.

In *WebML* werden die Verlinkungen explizit auf einer Seite modelliert und ausgehend von der Menge möglicher Aktionen die Navigation erzeugt.

In *OOHDM* dienen die Navigational Context Diagrams mit ihrer proprietären Notation zur Festlegung, welche Sichten global erreichbar sind und wie von dort aus navigiert werden kann.

In *UWE* [Kra07] lassen sich Menüs über spezielle Konstrukte im Navigationsmodell abbilden. Diese erlauben es, Verweise in ähnlicher Form wie in Menüs zu modellieren. Die Anwendung von nutzerspezifischen Anpassungen hat an dieser Stelle, wie bereits in Abschnitt 9.5.1 erwähnt, über aspektorientierte Mechanismen zu erfolgen.

In *WebDSL* lassen sich ebenfalls Menüstrukturen definieren, die selbst Verweise auf andere Seiten enthalten können [Vis08b, Vis08a]. Diese werden explizit als Teil von Seiten definiert und können, analog zu MontiWIS, ebenfalls hierarchisch aufgebaut sein.

Kapitel 10.

Anwendungsmethodik

Nach der Beschreibung sämtlicher Sprachen, des Generierungsprozesses und der Implementierung im generierten Web-Informationssystem widmet sich dieser Abschnitt der Anwendung und Erweiterung eines MontiWIS-Anwendungsprojekts. Zunächst wird im Abschnitt 10.1 der Aufbau eines solchen Projektes erläutert, inklusive der vorhandenen Verzeichnisse und Konfigurationsdateien. Abschnitt 10.2 erläutert anschließend, welche Schritte vor und nach der Generierung notwendig sind, um das Web-Informationssystem auf einen Server zu deployen. In Abschnitt 10.3 wird erläutert, in welchen Schritten die Modelle bearbeitet werden können und welche Besonderheiten bei einem iterativen Vorgehen zu beachten sind. Abschließend enthält Abschnitt 10.4 Hinweise zu Anpassungen, die in MontiWIS vorgesehen sind und wie sie umgesetzt werden.

10.1. Überblick über MontiWIS-Anwendungsprojekte

Dieser Abschnitt bietet einen Überblick über die Struktur von MontiWIS-Anwendungsprojekten, wobei zunächst das Anlegen eines neuen Projekts in Abschnitt 10.1.1 und nachfolgend in Abschnitt 10.1.2 sowohl die Verzeichnisstruktur als auch die Konfigurationsdateien erläutert werden.

10.1.1. Anlegen eines neuen Projekts

Das Anlegen eines neuen Projekts ist nicht in vollem Maße automatisiert, lässt sich jedoch sehr einfach bewerkstelligen. Im `MontiWIS`-Projekt existieren in der `build.xml`-Datei die ausführbaren Ziele (engl. *Targets*) `create-dev-template-project` sowie `create-dist-template-project`. Ersteres richtet sich an Nutzer, die selbst an der Entwicklung von MontiWIS beteiligt sind und damit eine Referenz auf das MontiWIS Projekt nutzen wollen, zweiteres wird in der Form ausgeliefert, dass die Generierung über einen Online Software Transformation Platform (OSTP) Dienst [Her06, Her14] ausgeführt wird. Beide Targets führen jedoch dazu, dass im `dist`-Ordner des `MontiWIS`-Projekts eine Archivdatei erzeugt wird, die sämtliche notwendigen Dateien zum Beginn eines neuen Projekts enthält. Ein solches Archiv kann einfach als neues Projekt importiert werden. Dazu wird in einem Eclipse `File > Import ...` und darin `General > Existing Projects into Workspace` gewählt. Im folgenden Auswahldialog wird das gewünschte Archiv ausgewählt und die Auswahl bestätigt. Daraufhin wird der Inhalt als neues Projekt im Eclipse Workspace angelegt.

Das angelegte Projekt selbst sollte zunächst aussagekräftig und sinnvoll umbenannt werden. Insbesondere sollte der Name `exampleproject`, der im `model`-Ordner als Paketname sowohl als Verzeichnis existiert als auch in den vorhandenen Standardmodellen als Paket vorliegt, durch einen aussagekräftigen Namen ersetzt werden. Dieser sollte ebenfalls in der Konfigurationsdatei `model/application.properties` unter `project_name` eingetragen werden. Näheres dazu wird in Kürze in Abschnitt 10.1.2 erläutert.

In einem nächsten Schritt sollten die fehlenden bzw. inkorrekten Pfadangaben korrigiert werden, was bei Betrachtung der Einträge relativ selbsterklärend sein sollte.

10.1.2. Aufbau eines MontiWIS-Anwendungsprojektes

Ein MontiWIS-Anwendungsprojekt ist im Wesentlichen ein gewöhnliches Eclipse-Projekt. Die darin enthaltenen Verzeichnisse sind überblicksartig in Tabelle 10.1 dargestellt.

Verzeichnis	Erläuterung
`model`	Enthält die Modelle, die das Web-Informationssystem beschreiben
`src`	Enthält die Generatorklasse sowie potenziell manuell geschriebenen Code, der in der generierten Applikation verwendet werden soll
`gen`	Enthält die generierten Klassen und Konfigurationsdateien
`gen/resource_bundles`	Enthält Dateien mit ausgelagerten Bezeichnungen
`res`	Enthält statische Dateien, die der Webanwendung zur Verfügung gestellt werden müssen (z. B. CSS, JavaScript)
`lib`	Enthält die benötigten Bibliotheken, die per Abhängigkeitswerkzeug Ivy aufgelöst und abgerufen wurden
`symtab`	Enthält die generierten Dateien, die für die Symboltabelle notwendig sind
`webapp`	Enthält die generierten JSP-Dateien sowie kopierte Dateien aus den `res` und `lib` Verzeichnissen
`dist`	Enthält die erzeugte Webapplikation in Paketform

Tabelle 10.1.: Verzeichnisse in MontiWIS-Anwendungsprojekten

Das zentrale Verzeichnis in einem MontiWIS-Anwendungsprojekt ist das `model` Verzeichnis. Darin sind sämtliche Modelle zu finden, die das Web-Informationssystem beschreiben und aus denen die Webapplikation generiert wird. Darüber hinaus liegt in diesem Verzeichnis eine Konfigurationsdatei, die in Kürze beschrieben wird.

Das Verzeichnis `src` beinhaltet zunächst die Klasse `Generate`, die zum Start der Modellver-

arbeitung und Generierung ausgeführt wird. Ferner sind in diesem Verzeichnis die Klassen enthalten, die einerseits innerhalb der Aktionen angesprochen oder aber auch außerhalb der Webapplikation ausgeführt werden sollen, aber dennoch zum Kontext des Projektes gehören.

Das Verzeichnis `gen` enthält (nach dem Ablauf der Generierung) sämtliche Klassen, die in der Webanwendung generiert wurden. Darin enthalten ist weiterhin das Verzeichnis `resource_bundles`, in dem die Standardbezeichnungen für Knöpfe und Attribute enthalten sind.

Im Verzeichnis `res` sind sämtliche statischen Inhalte für die Webanwendung enthalten. Dies umfasst JavaScript-Code, CSS-Dateien sowie Bilder, die beispielsweise auf Bedienelementen verwendet werden. Die dort enthaltenen Dateien werden bei der Komposition der Applikation in das `webapp` Verzeichnis kopiert und damit dem System zugänglich gemacht.

Im `lib`-Verzeichnis finden sich sämtliche Bibliotheken, die per Ivy-Datei aufgelöst und heruntergeladen wurden. Dort sind alle Bibliotheken abgelegt, die für die Ausführung der Applikation notwendig und nicht bereits Teil des Servers sind.

Das `symtab`-Verzeichnis beinhaltet serialisierte Symboltabelleneinträge, die während des Aufbaus der Symboltabelle erstellt und zur späteren Verwendung abgelegt wurden. Diese sind in der Regel nicht von Belang für einen Nutzer und können daher ignoriert werden.

Das `webapp`-Verzeichnis schließlich beinhaltet einerseits die generierten JSP-Dateien, aber auch kopierte `class`-Dateien sowie die statischen Inhalte aus dem bereits erläuterten `res`-Verzeichnis. Im Wesentlichen wird aus diesem Verzeichnis das `war`-Archiv erzeugt, das im `dist`-Verzeichnis abgelegt wird und während des Hochspielens auf den Server verwendet wird.

Im Hauptverzeichnis eines MontiWIS-Anwendungsprojektes befindet sich das Ant-Buildskript `build.xml`, das sämtliche Logik zum Hochspielen der generierten Webapplikation auf den Server enthält. Welche Targets darin in welcher Situation und Reihenfolge ausgeführt werden müssen, wird in Abschnitt 10.2 erläutert. Ferner existieren die `ivy.xml` und `ivysettings.xml` Dateien, die zum Auflösen und Herunterladen benötigter Bibliotheken dienen. Erstere Datei beinhaltet eine Liste benötigter Bibliotheken und letztere enthält unter anderem Angaben zu Servern, von denen die Dateien abzurufen sind.

Für die erfolgreiche Generierung des Web-Informationssystems müssen in zwei Konfigurationsdateien noch einige Werte angepasst werden, so dass sowohl der Generator als auch das Ant-Skript korrekt funktionieren können. Die Dateien sind in Tabelle 10.2 aufgelistet.

Datei	Erläuterung
`model/application.properties`	Enthält Parameter, die während der Generierung benötigt werden
`build.properties`	Enthält Parameter, die während des Ausführens der Ant-Datei benötigt werden

Tabelle 10.2.: Konfigurationsdateien

Die Datei `application.properties`, die sich im `model`-Verzeichnis befindet, beinhaltet

Parameter, die zur Zeit der Generierung bereits benötigt werden. Tabelle 10.3 beschreibt diese.

Parameter	Erläuterung
`project_name`	Projektname, der auch durchgängig in der Applikation verwendet wird
`package`	Paketname, unter dem die Klassen generiert werden
`target_web_app_dir`	Verzeichnis, in das die JSP-Seiten generiert werden sollen
`feedback_client`	Gibt an, ob der Feedback Client in die Applikation integriert werden soll
`database_platform`	Art der Datenbankverbindung, einer der Werte `mysql` oder `derby`
`only_crud`	Gibt an, ob die CRUD-Funktionalität für jede Klasse in das Menü aufgenommen werden soll. Muss entweder `true` oder `false`.
`glassfish_host`	Domänenname, unter dem der Glassfish Server erreichbar ist
`local_deploy`	Gibt an, ob die Webapplikation auf dem gleichen Rechner oder auf einem entfernten Rechner laufen soll. Muss entweder `true` oder `false` sein.

Tabelle 10.3.: Einträge der Datei `model/application.properties`

Der Parameter `project_name` ist insofern von zentraler Bedeutung, als daraus sowohl der Kontext auf dem Server und damit auch der URL-Teil, unter dem die Applikation zu erreichen ist als auch Fakten wie Datenbanknamen abgeleitet werden. Dementsprechend sollte der Projektname an dieser Stelle eindeutig sein und mit Bedacht gewählt werden, da er in der Regel nach außen sichtbar ist.

Der Parameter `package` gibt an, welches Paket als Wurzelpaket fungieren soll. Sämtliche anderen Pakete sind darunter angeordnet. Hier bietet es sich an, das Paket bei `de.montiwis` zu belassen, da daraus keine Nachteile entstehen. Der Parameter `target_web_app_dir` gibt an, in welches Verzeichnis die JSP-Seiten generiert und wohin sämtliche Klassendateien und statische Ressourcen kopiert werden sollen. Auch dieser Parameter sollte in der Regel auf der Voreinstellung `./webapp/WEB-INF` gelassen werden.

Der Parameter `feedback_client` bestimmt, ob in die generierte Webanwendung der Feedback Client, der innerhalb der Dissertation von Thomas Kurpick entsteht, eingebunden werden soll [LHK+12]. Dies bewirkt, dass an dezidierten Stellen entsprechender Code eingebunden wird. Voraussetzung ist, dass im SSELab [Her14] der Feedback-Dienst für ein Projekt konfiguriert ist und die entsprechenden JavaScript-Dateien im `res/script`-Verzeichnis vorhanden sind, da darin sowohl Funktionalität als auch Zugangsdaten zu den Diensten enthalten sind. Zulässige Werte für diesen Parameter sind `true` und `false`.

Über den Parameter `database_platform` wird konfiguriert, welches Datenbankmanage-

mentsystem in der Zielplattform verwendet wird. Wie bereits in Abschnitt 6.3.6 bei der Beschreibung der Datenbankkonfiguration erwähnt, werden bestimmte Teile im generierten System abhängig vom verwendeten Datenbankmanagementsystem generiert. An dieser Stelle werden aktuell die Werte `mysql` oder `derby` unterstützt.

Der Parameter `only_crud` bestimmt, ob im generierten System für jede Klasse des Domänendatenmodells ein Menü mit entsprechenden Untereinträgen zur kompletten Bearbeitung des jeweiligen Datentyps erzeugt werden soll. Diese Funktionalität wird dem Gastnutzer zur Verfügung gestellt, so dass die vollständige CRUD-Anwendung ohne Authentifizierung am System möglich ist. Unterstützte Werte für diesen Parameter sind `true` und `false`.

Über den Parameter `glassfish_host` wird der Domänenname spezifiziert, auf dem der Glassfish Server läuft, auf den die generierte Applikation installiert werden soll. Dies hat lediglich Bedeutung bei der Konfiguration externer Datenbankverbindungen in der Datenbankkonfigurationsdatei `persistence.xml` im Fall der Verwendung von `derby`, was es leider nötig macht, bei der Konfiguration des Generators bereits Aussagen zum späteren Einsatzort zu treffen.

In diesem Kontext ist auch der Parameter `local_deploy` angesiedelt. Darüber wird festgelegt, ob die Webanwendung auf dem gleichen Rechner installiert wird, auf dem sie generiert und von dem aus sie installiert wird. Sollte dies der der Fall sein, so ist bei diesem Parameter `true` zu wählen, was zur Folge hat, dass die Initialisierungsklasse auf die entsprechenden DAO-Objekte zugreifen kann. Anderenfalls werden die Daten direkt per `EntityManager` in die zugehörige Datenbank geschrieben.

Die zweite, bereits in Tabelle 10.2 erwähnte Konfigurationsdatei befindet sich im Wurzelverzeichnis eines MontiWIS-Anwendungsprojekts und besitzt den Namen `build.properties`. Analog zum Namen befinden sich darin Parameter, die den Buildprozess steuern bzw. konfigurieren. Die einzelnen Parameter sind inklusive einer kurzen Erläuterung in Tabelle 10.4 gelistet.

Der erste dargestellte Parameter gibt an, in welchem Verzeichnis der Glassfish Server installiert ist. Der Pfad, der dort angegeben und gefordert ist, endet in einer Standardinstallation mit den Verzeichnissen `glassfish3/glassfish` (ohne abschließendes "/"). Darin sind in der Regel beispielsweise die Verzeichnisse `bin`, `lib` sowie `domains` zu finden. Die korrekte Angabe dieses Parameters wird zum Ansprechen der Glassfish Administrationsprogramme benötigt, um beispielsweise die Applikation auf den Server zu spielen oder auch den Server konfigurieren zu können. Weitere Parameter, die zur korrekten Konfiguration des Servers zu setzen sind, umfassen `glassfish_domain`, dem in der Regel `domains` zugeordnet bleiben kann, da dies das Standardverzeichnis einer Glassfish-Installation ist. Ebenfalls kann, sofern nicht geändert, `4848` als `glassfish.admin.port` belassen werden. Der Parameter `glassfish.host` gibt an, unter welcher Adresse der Glassfish Server installiert ist. Dies wird unter anderem zur Konfiguration und zum Installieren der generierten Applikation benötigt. Ferner geben `glassfish.user` den Nutzer und `glassfish.passwordfile` sowohl Nutzernamen als auch Konfigurationsdatei für das Glassfish Administrationskonto an. In der Konfigurationsdatei muss hinter dem Parameternamen `AS_ADMIN_PASSWORD` das entsprechende Passwort spezifiziert werden.

Neben Parametern zur Konfiguration des Glassfish Servers und zur Installation des Web-

Parameter	Erläuterung
`glassfish.home`	Verzeichnis, wo der Glassfish Server installiert ist
`glassfish.domain`	`domains` innerhalb des Glassfish Verzeichnisses
`glassfish.admin.port`	HTTP Port, über den der Adminstrationsdienst erreichbar ist
`glassfish.host`	Domänenname, unter dem der Glassfish Server erreichbar ist
`glassfish.user`	Nutzername des Glassfish `admin` Nutzers
`glassfish.passwordfile`	Name der Datei, in der das Glassfish Administratorkennwort abgelegt ist
`database.admin.username`	Nutzername für den Administrator des Datenbankservers
`database.admin.password`	Passwort für den Administrator des Datenbankservers
`montiWISTargetDir`	Pfad zur Bibliothek `mw-target.jar`

Tabelle 10.4.: Einträge der Datei `build.properties`

Informationssystems darauf, werden mit den beiden Parametern `database.admin.username` und `database.admin.password` Daten für den Zugriff auf den Datenbankserver abgefragt. Die hier angegebenen Daten müssen zum Anlegen einer Datenbank und den Zugriff darauf berechtigen.

Schließlich wird über den Parameter `montiWISTargetDir` festgelegt, in welchem Verzeichnis die Bibliothek `mw-target.jar` zu finden ist, die zusammen mit den notwendigen anderen Bibliotheken der Applikation zur Verfügung gestellt und dazu in das `WEB-INF/lib`-Verzeichnis kopiert wird.

10.2. Durchführung der Generierung

Nach dem generellen Aufbau eines MontiWIS-Anwendungsprojekts wird in diesem Abschnitt Augenmerk auf die eigentliche Durchführung der Generierung aus Sicht eines Nutzers gelegt. Zunächst werden in Abschnitt 10.2.1 die Bedingungen erläutert, die zur erfolgreichen Generierung und insbesondere zur Installation des Web-Informationssystems auf einem Server notwendig sind. Anschließend wird zunächst die Generierung über eine lokale Installation des Generators in Abschnitt 10.2.2 und darauf folgend die Ausführung des Generators über einen OSTP-Service in Abschnitt 10.2.3 erläutert. Unabhängig von der Art der Generierung sind in jedem Fall zusätzlich die Schritte notwendig, die in Abschnitt 10.2.4 erläutert werden.

10.2.1. Lokale Vorbedingungen

Die in diesem Abschnitt beschriebenen Bedingungen und Schritte sind essenziell für die korrekte Generierung und insbesondere eine erfolgreiche Installation des generierten Systems.

Projektsetup und Konfiguration

Wie bereits in Abschnitt 10.1.1 erläutert, lassen sich MontiWIS-Anwendungsprojekte per Import als neues Eclipse-Projekt anlegen. Dies ist in jedem Fall notwendig. Daran anschließend sollten sinnvolle Umbenennungen des Projekts selbst aber auch der im `model` vorhandenen Pakete vorgenommen werden. Abschließend ist es obligatorisch, dass die Parameter angepasst werden, die in Abschnitt 10.1.2 erläutert wurden.

Installation des Glassfish Servers

Für die spätere korrekte Funktionsweise der Applikationsinstallation durch das Buildskript muss selbstverständlich ein Glassfish Server auf dem System installiert sein. Getestet und für funktionsfähig befunden wurde das ausgelieferte System mit einem *Full Platform Server* der Version 3.1.2.2. Für die Installation bieten sich mehrere Möglichkeiten, wobei an dieser Stelle auf [Gla] verwiesen sei. In jedem Fall sollte der Ort der Installation mit den Eintragungen in den Konfigurationsdateien übereinstimmen.

Auswahl und Installation eines Datenbankservers

Neben einem Glassfish Server ist es obligatorisch, dass ein Datenbankserver auf dem System installiert ist. Unterstützt werden aktuell sowohl der Derby Server [Der], der u. a. als Teil des Glassfish Servers mitgeliefert wird, als auch MySQL [Mys].

Der Derby Server sollte in der Regel dann gewählt werden, wenn es lediglich um eine kurzfristige Demonstration der generierten Applikation geht. Für die Verwendung dieses Systems ist keine separate Installation notwendig, da ein Derby Server bereits Teil der Glassfish Installation ist und auch über Befehle gegen diesen Server gestartet und gestoppt werden kann. Nachteilig dabei ist jedoch, dass Administrationsaufgaben wie beispielsweise Backups und Restores der Datenbank eher schwierig zu handhaben sind.

Demgegenüber ist der MySQL Server ein robustes Datenbankmanagementsystem, das sämtliche Vorzüge einer Datenbank für den Produktivbetrieb bietet. Ein solcher Server muss auf dem System installiert sein, damit die Daten entsprechend abgelegt werden können.

Für beide Varianten existieren in der vorhandenen Ant-Builddatei entsprechende Targets, so dass mit den unterschiedlichen Vorgehensweisen z. B. bei Löschen einer Datenbank entsprechend umgegangen werden kann.

10.2.2. Lokale Generierung

Für die tatsächliche Durchführung der Generierung existieren im Wesentlichen zwei Möglichkeiten. In der ersten Möglichkeit findet die Generierung vollständig lokal im MontiWIS-Anwendungsprojekt statt. Dazu muss im Pfad zunächst das `MontiWIS`-Projekt verlinkt sein.

Dies kann entweder durch ein parallel installiertes und vollständiges `MontiWIS`-Projekt erfüllt werden, welches in den Eclipse Pfad des MontiWIS-Anwendungsprojekts aufgenommen wird, oder aber durch ein aus dem `MontiWIS`-Projekt erzeugtes jar-Archiv, das in den Pfad aufgenommen wird.

Die Generierung selbst erfolgt durch das Ausführen der `Generate`-Klasse aus dem Defaultpaket bzw. Wurzelverzeichnis des `src`-Verzeichnisses eines jeden MontiWIS-Anwendungsprojekts. Dadurch wird der in Abbildung 4.2 beschriebene Generierungsprozess angestoßen.

10.2.3. Generierung über OSTP

Eine weitere Möglichkeit ist prinzipiell die Verwendung eines OSTP-Dienstes [Her06, Her14], über den die Generierung ausgeführt wird. Dazu ist es notwendig, dass das entsprechende Eclipse-Plugin installiert ist und der Server die aktuelle und funktionsfähige Version des MontiWIS-Generators bereithält.

Um die Generierung durchzuführen, muss in der zum OSTP-Plugin gehörenden Eclipse Einstellungsseite Nutzername und Passwort eingetragen werden, so dass eine Authentifizierung mit dem *SSELab-Server* durchgeführt werden kann. Nach erfolgreicher Anmeldung lässt sich das passende Projekt und der passende Dienst (in diesem Fall MontiWIS) auswählen.

Die Generierung selbst wird durch einen Rechtsklick auf das MontiWIS-Anwendungsprojekt und der Auswahl des Punkts `OSTP > MontiWIS` ausgeführt. Das Projekt wird daraufhin zum SSELab-Server gesendet, wo der Generierungsprozess ausgeführt wird und die neuen Dateien zurück zum aufrufenden Client gesandt und in das aufrufende Projekt integriert werden. Damit ist das Ergebnis im Wesentlichen das gleiche wie bei der lokalen Ausführung des Generators.

Nichtsdestotrotz wird in der Regel die lokale Ausführung empfohlen, da sowohl Warnungs- als auch potenzielle Fehlermeldungen in diesem Fall ausführlicher sind als dies bei der OSTP-Ausführung der Fall ist.

10.2.4. Schritte nach der Generierung

Nach der Generierung, wie auch immer sie durchgeführt wurde, sind einige Schritte vorzunehmen, um die Applikation auf dem vorhandenen Glassfish Server zu installieren. All diese Schritte sind in der `build.xml` Datei definiert und sollten darin, z. B. per Eclipse Ant-View, ausgeführt werden.

Vollständige, erstmalige Installation

Für die erstmalige Installation ist nach dem Generieren das Ausführen des Targets `XXX_1_Setup_Infrastructure`, wobei `XXX` an dieser Stelle entweder für `DERBY` oder für `SQL` steht. Diese Konvention gilt auch im folgenden Verlauf dieses und der nächsten Abschnitte. Die Targets sind jeweils angepasst für die beiden Datenbankmanagementsysteme Derby im ersten Fall und MySQL im zweiten und sollten je nach verwendetem System aufgerufen werden.

Mit dem Ausführen des Targets `1_Setup_Infrastructure` werden im Glassfish Server zunächst einige Einstellungen vorgenommen. So wird beispielsweise die Datenbankverbindung

konfiguriert, die Authentifizierungsrealms werden eingerichtet sowie einige weitere Einstellungen vorgenommen, die das reibungslose Funktionieren der Installation gewährleisten. Diese Schritte sind in der Regel nur beim erstmaligen Installieren einer Applikation (wohlgemerkt: einer bestimmten Applikation, da die hier durchgeführten Schritte spezifisch für eine einzelne Applikation sind und für jedes eigenständige Web-Informationssystem durchgeführt werden müssen) notwendig.

In einem zweiten Schritt wird die Applikation deployed, was der Installation auf dem Server entspricht. Ferner werden im gleichen Schritt die Initialisierungen wie Anlegen von Nutzern und Nutzergruppen durchgeführt. Dies geschieht über das Target `XXX-2_Deploy_and_Initialize`. Wichtig ist bei diesem Target jedoch, dass es nur einmalig nach Anlegen der Datenbank ausgeführt werden darf, da anderenfalls Nutzer und Nutzergruppen mehrfach angelegt werden, was eine korrekte Funktionsweise des Systems verhindert.

Entfernen der Applikation vom Server

Für den Fall, dass Änderungen an den Modellen und damit auch dem neu generierten System vorgenommen wurden, die eine Neuinstallation der Applikation erfordern, sind zunächst die bereits installierte Anwendung zu entfernen (auch: undeployen) und die Datenbank zu löschen. Dies geschieht über das Target `XXX-4_Teardown`. Für die vollständige Installation danach muss erneut das bereits erwähnte Target `XXX-2_Deploy_and_Initialize` ausgeführt werden, da auch die Datenbank und die darin zu erstellenden Objekte neu erzeugt und initialisiert werden müssen.

Erneute Installation unter Beibehaltung der Datenbank

Im Fall, dass an der Datenstruktur keine Änderungen vorgenommen wurden und auch aus anderen Gründen keine Neuinitialisierung der Datenbank gewünscht ist, genügt das Ausführen des Targets `XXX-3_Deploy-only`. Dabei wird zwar die neu generierte Anwendung erneut deployed, jedoch bleibt die bislang existierende Datenbank und somit auch deren Inhalte erhalten.

10.3. Iterative Entwicklung des Web-Informationssystems

Nach der Erläuterung, wie ein MontiWIS-Anwendungsprojekts aufgebaut ist und welche Schritte für die Generierung des Web-Informationssystems auszuführen sind, wird in diesem Abschnitt beleuchtet, wie die iterative Entwicklung eines Systems vonstattengehen kann. Die beschriebenen Schritte können in der Reihenfolge ausgeführt werden, in der sie beschrieben sind, sie können jedoch diesbezüglich auch variieren. Die Abhängigkeiten der Modelle ergeben sich direkt aus Abbildung 3.3 und den Beschreibungen dieser Grafik. Damit ist offensichtlich, dass Verweise von einem Modell auf ein anderes zumindest hinsichtlich der Existenz des referenzierten Modells korrekt sein müssen.

Im folgenden Abschnitt werden die einzelnen Entwicklungsschritte ausgehend von einem neu angelegten Projekt (siehe Abschnitt 10.1.1) mit seinen standardmäßig vorhandenen

Modellen (Klassendiagramme, vordefinierte Seiten, Applikationsmodell und erstes Aktivitätsdiagramm) erläutert.

Allein durch diese Modelle ist ohne weitere Anpassungen ein vollständig lauffähiges System inklusive Berechtigungen generierbar. Am einfachsten zu generieren ist eine CRUD-Applikation für den Fall, dass keine Nutzeranmeldung erforderlich ist. In diesem Fall genügt zusätzlich zum erweiterten Klassendiagramm die Nutzung des `only_crud` Parameters (vgl. hierzu Abschnitt 10.1.2). Durch Erweiterung des Klassendiagramms sowie der Verlinkung der Standardfunktionalität in der Applikationssprache ist eine angepasste CRUD-Anwendung erstellbar. Der Aufwand dafür ist nur geringfügig höher als die Alternative ohne Nutzeranmeldung, jedoch ist dadurch die Übernahme der Standardfunktionalität für jede Klasse als vollwertigen Applikationsteil in das Web-Informationssystem möglich.

10.3.1. Festlegen von Anwendungsfällen und Akteuren

Auch wenn MontiWIS keine explizite Möglichkeit vorsieht, Anwendungsfälle und Akteure zu definieren, lassen sich derartige Anforderungsspezifikationen zumindest begrenzt durch die angebotenen Modelle abdecken. Ein Akteur eines Anwendungsfalldiagramms entspricht im Wesentlichen einer Rolle in der Applikationssprache. Damit lassen sich in einem ersten Schritt diejenigen Rollen definieren, die im Web-Informationssystem vorhanden sein müssen. Die Anwendungsfälle selbst sind dahingehend zu untersuchen, ob sie aus einem Workflow bestehen, der über ein Aktivitätsdiagramm abgebildet werden muss (etwa im Fall von *Nutzer registrieren*), ob er durch die Verwendung einer oder mehrerer Standardseiten gelöst werden kann (etwa bei einem Anwendungsfall *Personen verwalten*) oder ob die Anzeige einer Seite ohne weitere Interaktion den Anwendungsfall abdeckt. Ein jeder Anwendungsfall muss anschließend den passenden Rollen zugeordnet werden. Damit die hinzugefügten Artefakte auch erreichbar sind, müssen sie weiterhin dem Menü hinzugefügt werden. Ferner ist dafür Sorge zu tragen, dass die referenzierten Modelle (Klassen, Aktivitäten, Seiten) existieren. Nähere Erläuterungen dazu folgen in nächsten Abschnitten.

10.3.2. Erweiterung der Datenstruktur

Die Erweiterung der Datenstruktur kann, wie vorgesehen, ebenfalls iterativ erfolgen. Neue Klassen können hinzugefügt werden und mit vorhandenen Klassen verknüpft werden. Sollte die hinzu generierte Standardfunktionalität benötigt werden, so ist sie über die Applikationssprache einer oder mehreren Rollen im System verfügbar zu machen. Zu Bedenken ist bei der Erweiterung der Datenstruktur jedoch, dass in der Regel eine bereits vorhandene Datenbank nicht erhalten bleiben kann. Derartige Schemaevolution ist als weiterführendes Forschungsthema aktuell nicht Teil von MontiWIS und muss durch spätere Arbeiten in Verbindung mit Modellevolution und -versionierung angegangen werden.

10.3.3. Erstellen von Aktivitäten

Die Erweiterung des Web-Informationssystems um weitere Workflows kann zunächst durch das einfache Anlegen von neuen Aktivitäten bzw. das Kopieren der initial vorhandenen

Aktivität erfolgen. Darin muss einzig eine Aktion definiert sein, auf die vom Startknoten heraus referenziert wird. Alle weiteren Aktionen lassen sich iterativ hinzufügen. Dabei hat sich bewährt, dass zunächst das Gerüst der Aktionen und die dazwischen existierenden Kanten modelliert werden. Der Inhalt von Aktionen kann zu Beginn aus Kommentaren bestehen, die als rudimentäre Dokumentation beschreiben, was Aufgabe der jeweiligen Aktion sein soll. Dies kann nach und nach durch konkreten Inhalt und eine explizite Implementierung ersetzt werden.

Sollten bereits andere Modelle referenziert werden (Klassen des Domänendatenmodells als Teil von Java oder als Parameter von Aktionen oder auch Seiten), so müssen diese zumindest existieren, damit die Modelle korrekt verarbeitet werden können.

Eine Neugenerierung von Aktivitäten erfordert keine Neuinitialisierung der Datenbank, einzig begonnene Abläufe der gleichen Art lassen sich aufgrund des veränderten Aufbaus nicht ohne Weiteres fortsetzen.

10.3.4. Spezifikation von Seiten

Auch bei der Definition von Seiten genügt zu Beginn ein sehr rudimentärer Aufbau. Neben dem Namen und der Seitendefinition kann der Inhalt der Seite aus Kommentaren oder einfachen Texten bestehen, die den späteren Inhalt der Seite beschreiben. Sollten bereits Seitenparameter angegeben werden, so ist auf die Existenz der Klassen und den korrekten Aufruf aus Aktionen heraus zu achten. Abgesehen davon gelten die gleichen Regeln wie bei der der Definition aller anderen Artefakte.

Auch die Neugenerierung von Seiten erfordert kein Neuanlegen der Datenbank und ist diesbezüglich unkritisch.

10.4. Anpassung / Erweiterung der Implementierung

Nach der Beschreibung, wie ein MontiWIS Projekt aufgebaut ist und über welche Schritte die Applikation erzeugt und installiert wird und unter welchen Voraussetzungen die Modellierung in der Regel vonstattengeht, befasst sich dieser Abschnitt mit den Modifikationen, die einfach in MontiWIS-Projekten durchgeführt werden können und wobei die Architektur von MontiWIS bzw. der generierten Applikation hilfreich ist.

10.4.1. Umbenennen von Buttons und Standardbezeichnungen

Häufige werden bei einem generierten System die Standardbezeichnungen nicht dem entsprechen, was im Kontext einer spezifischen Anwendung gewünscht ist. Dies betrifft sowohl die Bezeichnungen der Attribute in den Standardseiten als auch die Beschriftungen der Knöpfe zur Auswahl von Alternativen oder Absenden der eingegebenen Daten. Die Datei, in der diese Bezeichnungen gespeichert sind, befindet sich im Ordner `resource_bundles`, welcher sich wiederum im `gen`-Ordner findet. Der Name der Datei, die letztlich in die Applikation eingebunden wird, lautet `langstring.properties`. Neben dieser findet sich eine Datei `langstring.properties_will_be_overwritten`, die mit jedem Generationsdurchlauf neu

erzeugt wird und damit sämtliche in der aktuellen Applikation vorhandenen Schlüssel-Wert-Paare enthält. Sollten also in der `langstring.properties`-Datei Einträge fehlen, so müssen diese aus der neu generierten Datei übernommen werden. Hilfreich für derartige Dateivergleiche ist prinzipiell jedes Werkzeug, das komfortable Textvergleiche erlaubt.

Der Aufbau und Inhalt der `langstring.properties`-Datei ist beispielhaft in Abbildung 10.5 dargestellt.

Config

```
1 # Attribute von Klassen
2 Person_birthDate=birthDate
3
4 # Aktivitynamen
5 activityname.Child=Child
6
7 # Beschriftung eines Knopfes
8 de.montiwis.GradeThesis.g_G_Decision1.T0.g_Save=Save
9
10 # Beschriftung von Standardknoepfen
11 default.button.label.finish=Finish
12
13 # Text auf Standardseiten
14 default.text.activity_finished=This activity has finished so far. More activities might ↩
     appear on the page of your running activity list.
```

Abbildung 10.5.: Beispielhafter Inhalt der `langstring.properties` Datei

Einerseits werden in der Datei für jedes Attribut einer jeden Klasse Bezeichnungen festgelegt, die beispielsweise in den Standardseiten verwendet werden, die für jede Klasse generiert werden. Deren Bezeichner setzt sich aus Klassennamen und Attributnamen zusammen, getrennt von einem Unterstrich, wie anhand des Beispiels `Person_birthDate` (Zeile 2) dargestellt ist.

Des Weiteren werden Aktivitätsnamen definiert wie in Zeile 5 zu sehen. Das Prefix `activityname` gefolgt von einem Punkt und dem Aktivitätsnamen dienen hier als Schlüssel. Der zugeordnete Wert wird beispielsweise in der Liste aktuell laufender Aktivitäten angezeigt (vgl. hierzu Tabelle 9.3 und entsprechende Erklärungen dazu).

Die Beschriftung eines Knopfes, der in einer Seite bei freier Auswahl der nächsten Aktion angezeigt wird, ist wie in Zeile 8 zu sehen definiert. Hierbei werden explizit die Knotennamen per vollqualifiziertem Namen adressiert. `de.montiwis.GradeThesis` ist hierbei der Aktivitätsname und `g_Decision1` sowie `g_Save` sind die an einer Kante beteiligten Knoten, die durch Punkte und das Schlüsselwort `T0` getrennt werden. Damit ist klar, dass hier die Kante vom Knoten `g_Decision1` zum Knoten `Save` bezeichnet wird und der entsprechende Knopf bei freier Auswahl (d. h. der Verknüpfung per Verzweigungsknoten ohne Bedingungen, vgl. Abbildung 8.12 sowie entsprechende Erläuterungen) in der Seite angezeigt wird, die dem Verzweigungsknoten vorgelagert ist. Der Verzweigungsknoten kann potenziell auch durch Transformation aus einer Kante in Kurzschreibweise entstanden sein und muss nicht zwangsläufig explizit Teil der Aktivität gewesen sein. Dennoch sollte relativ einleuchtend sein,

dass die Bezeichnung aus dem Namen des Zielknotens abgeleitet wird und dementsprechend auch hier angepasst werden kann.

Etwas einfacher gestalten sich die Schlüssel für Standardknöpfe wie beispielsweise dem Knopf, der zu einem Endknoten führt. Dessen Schlüssel ist in Zeile 11 dargestellt und setzt sich zusammen aus dem Schlüsselworten `default.button.label` sowie dem Namen des Knotens, in diesem Fall `Finish`.

Ähnlich einfach lassen sich Texte ändern, die standardmäßig auf einigen Seiten angezeigt werden. Beispielhaft ist in Zeile 14 dargestellt, welcher Text auf der Seite angezeigt wird, die nach Beendigung eines Ablaufs angezeigt wird. Deren Bezeichner resultiert aus den Schlüsselworten `default.text` sowie dem Namen der Seite, auf der er angezeigt wird, was in diesem Fall `activity_finished` ist. Sowohl für Standardknoten als auch für derartige Standardseiten existieren weitere Einträge, die jedoch selbsterklärend sind.

10.4.2. CSS-Anpassungen

Neben den Anpassungen an generierten Beschriftungen lässt sich relativ einfach das Aussehen der Bedienelemente wie Knöpfe, dem Menü oder auch den Tabellen im Web-Informationssystem anpassen. Dazu finden sich im Verzeichnis `res/css` die CSS-Stylesheets, die dazu benötigt werden. Wie bereits in Abschnitt 5.3.2 erwähnt, kommt bei der Darstellung der genannten Elemente jQueryUi [JQuc] zum Einsatz. Darüber lassen sich auf der entsprechenden Homepage über eine Generatorapplikation passende CSS-Dateien erzeugen, die in einem MontiWIS Projekt in `res/css/custom` mit dem Namen `jquery-ui.custom.css` abgelegt werden müssen, damit sie nach Kopieren an die entsprechende Stelle eingebunden werden können.

Weiterhin lässt sich das Aussehen von beispielsweise Schriften und Hintergrund der Seite in der Datei `res/css/style.css` anpassen.

Kapitel 11.

Fallstudien und Bewertung

Nach der vollständigen Erläuterung der Konzepte des MontiWIS-Ansatzes soll dieser Abschnitt dazu dienen, umgesetzte Systeme zu beschreiben und damit die Sinnhaftigkeit und vielseitige Einsetzbarkeit des entwickelten Ansatzes zu verifizieren. Dabei soll nicht auf jedes Detail der Implementierung eingegangen werden, sondern lediglich ein Überblick über Besonderheiten und generelle Funktionsweise der entwickelten Web-Informationssysteme dargestellt werden.

11.1. Bibliotheksverwaltung

Ein System, das mit MontiWIS entwickelt wurde, ist das interne Bibliothekssystem des Instituts für Software Engineering der RWTH Aachen.

Die Anforderung dabei war, dass sowohl Bücher als auch Personen in einem webbasierten System verwaltet werden können. Personen sollten sich registrieren können, Bücher suchen und ausleihen. Eigene ausgeliehene Bücher sollten anderen Personen zugeordnet werden können. Von anderen Personen sollen Bücher ebenfalls übernommen werden können. Ferner soll zu jedem Buch die Ausleihhistorie gespeichert und der Zugriff auf eine externe Seite ermöglicht werden, die eine Suche über einem externen Bücherbestand realisiert.

11.1.1. Datenmodell

Das Datenmodell in der Bibliotheksapplikation ist recht übersichtlich. Es existiert eine Klasse `Person`, die als Nutzer im Web-Informationssystem gekennzeichnet ist. Diese Klasse besitzt Attribute wie Vor- und Zuname, E-Mail-Adresse und Raumbezeichnung. Ein `Book` besitzt einen Titel, einen Freitext mit diversen Autoren, einen Freitext, in dem die bestellende Person eingetragen werden kann, einen Preis sowie einige weitere Attribute. Ferner existiert eine Klasse `Lending`, die neben einem Ausleihdatum Referenzen zu einer Person und einem Buch besitzt. Diese Assoziation wird zum Speichern der Ausleihhistorie verwendet. Zwischen `Person` und `Book` besteht eine Ausleihbeziehung, die den aktuellen Besitz eines Buches repräsentiert.

11.1.2. Aktivitäten

Die überwiegende Funktionalität des Systems findet sich in Aktivitäten wieder, die im Folgenden kurz beschrieben werden.

Registrierung

Wie gefordert existiert in der Bibliotheksapplikation die Möglichkeit für einen Nutzer, sich neu zu registrieren. Die Maske für die Eingabe von Nutzerdaten ist in Abbildung 11.1 dargestellt.

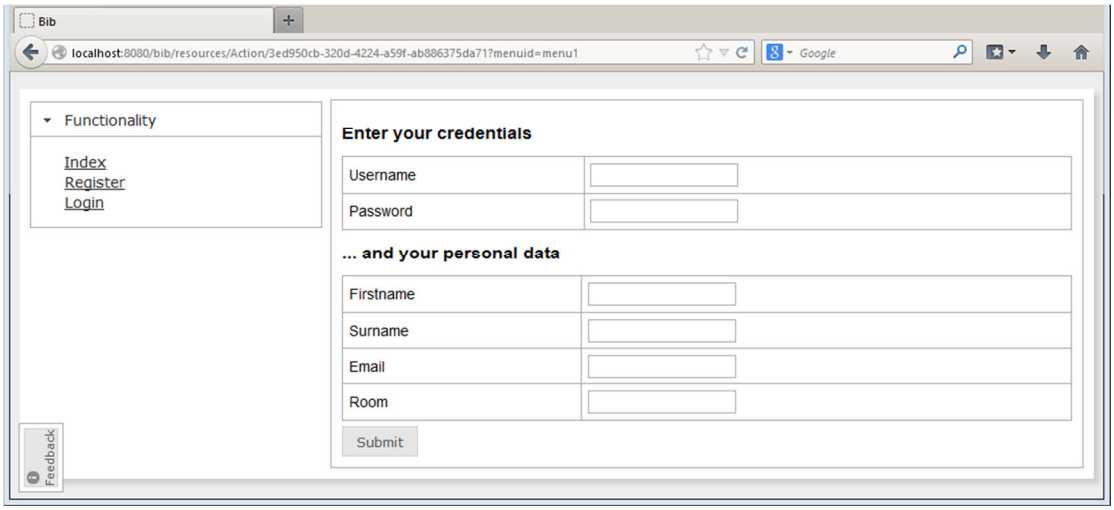


Abbildung 11.1.: Registrierungsmaske des Bibliothekssystems

Dabei werden zunächst einige wenige Daten des Nutzers abgefragt, die in einem nächsten Schritt hinsichtlich Vollständigkeit überprüft werden. Bei fehlenden Angaben werden entsprechende Warnungen angezeigt und der Nutzer erneut auf die Eingabeseite geleitet. Bei vollständiger Eingabe wird dem Nutzer weiterhin die Gruppe eines registrierten Nutzers (`user`) zugeordnet und in der Datenbank gespeichert.

Anzeigen eigener ausgeliehener Bücher und Neuzuordnung

In einer weiteren Aktivität lassen sich die Bücher anzeigen, die vom aktuellen Nutzer ausgeliehen wurden. Aus dieser Liste lassen sich beliebig viele Bücher auswählen, um sie dann per Auswahlknopf entweder zurückzugeben (womit das Leihverhältnis aufgelöst wird), oder aber einem anderen Nutzer zuzuordnen. Damit wird diese Person als aktuell Ausleihender gespeichert.

Suchen von Büchern und Ausleihen

Nach Büchern kann ein Nutzer im generierten Web-Informationssystem auf zwei Arten suchen. Dabei ist jeweils ausschlaggebend, nach welchem Kriterium die Suche erfolgen soll.

Zunächst ist es möglich, aus einer Liste von allen vorhandenen Büchern das gewünschte auszuwählen, um es entweder direkt auszuleihen oder auf einer nächsten Seite Informationen über das gewählte Buch anzuzeigen. Auf dieser Informationsseite kann das Buch geliehen oder zur vorherigen Seite mit der Auswahlliste aller Bücher zurückgekehrt werden. Sollte das Buch bereits ausgeliehen sein, was auf der Übersichtsseite ersichtlich wäre, wird der

vorherige Ausleiher beim erneuten Ausleihen per E-Mail benachrichtigt. Diese Funktionalität wurde per externem Methodenaufruf in einer Aktion implementiert.

Ein anderes Szenario für die Ausleihe zeichnet sich dadurch aus, dass der aktuelle Nutzer weiß, welcher andere Nutzer ein bestimmtes Buch besitzt bzw. besaß. In diesem Fall lässt sich eine andere Aktivität verwenden, in der zunächst eine Liste sämtlicher Nutzer angezeigt wird. Nach Auswahl des gewünschten Nutzers werden sämtliche seiner Bücher angezeigt. Aus dieser Liste können die entsprechenden Bücher gewählt und übernommen werden.

Zugriff auf externe Bibliothek

Bei der Anforderungsanalyse existierte die Anforderung, aus dem Bibliothekssystem heraus Zugriff auf die externe Bibliothek des Fachbereichs Informatik zu erhalten. Diese Anforderung wurde pragmatisch über die Einbindung der externen Suchmaske über einen `iframe` gelöst, was den Anforderungen genügte. Hierzu kam das `html`-Element der Seitenbeschreibungssprache zum Einsatz.

Datenimport

Da es bei der Bibliotheksanwendung um den Nachfolger einer bereits existierenden, allerdings den Anforderungen nicht mehr genügenden Applikation handelt, war die Datenübernahme aus dem Altsystem obligatorisch. Hierzu wurde eine Aktivität angelegt, in der diese Übernahme angestoßen werden kann. Dazu wird in einem ersten Schritt der Name der Datei abgefragt, der die Daten aus der vorliegenden Datenbank enthält. In einem weiteren Schritt werden intern diese Daten geparsed und in neue Objekte des generierten Systems überführt. Diese werden anschließend per Zugriff auf die generierten `DAO`-Klassen in der neuen Applikation gespeichert. Diese Aktivität ist selbstverständlich lediglich einem Administratornutzer zugänglich.

11.1.3. Applikationsmodell

In der Bibliotheksapplikation existieren drei verschiedene Nutzergruppen, deren Kompetenzen und Möglichkeiten nachfolgend kurz erläutert werden.

Der *Gastnutzer* hat in diesem System keine umfangreichen Handlungsalternativen. Unregistriert ist es lediglich möglich, sich über den Ablauf der Registrierung am Web-Informationssystem anzumelden.

Bei der Registrierung erhält ein Nutzer die Berechtigungen eines *Nutzers* (`user` in der Applikation). Damit sind sämtliche wichtigen, in Abschnitt 11.1.2 vorgestellten Abläufe auch für einen Nutzer verfügbar.

Neben einem normalen Nutzer existiert ein *studentischer Nutzer*, der sehr eingeschränkte Rechte besitzt. Ihm ist es lediglich möglich, seine eigenen Ausleihen zu betrachten sowie nach Büchern des Institutsbestands und der Fachbereichsbibliothek zu suchen. Ausleihen ist für ihn nicht ohne Weiteres möglich, dazu muss ein gewöhnlicher Nutzer dem Studenten das entsprechende Buch zuordnen.

Schließlich existiert ein *Administrator*. Nutzer dieser Nutzergruppe haben keinen Zugriff auf die Abläufe eines gewöhnlichen Nutzers, jedoch ist der Zugriff auf die generierten Standardfunktionalitäten möglich, die für jede Klasse generiert wurden. Darüber lassen sich die Objekte in der Datenbank direkt bearbeiten, sollte dies nötig sein. Ferner dürfen sie den Datenimport durchführen, wie er in Abschnitt 11.1.2 beschrieben ist.

11.2. Modulhandbuch

Als eine weitere Fallstudie wurden die Essenzen des *Modulhandbuch*-Systems ausgewählt. Hierfür existiert unter [MHB] das manuell implementierte Pendant, das zunächst über Apache Struts [Str] unter Zuhilfenahme der Velocity Template Engine [Vel] implementiert wurde. Erfahrungen aus der Entwicklung wurden in [RRSQS08] veröffentlicht.

Bei dem Modulhandbuch handelt es sich um eine Webanwendung, in der sowohl Institute, Studiengänge, Lehrmodule als auch zugehörige Lehrveranstaltungen verwaltet werden sollen. Module sind dabei Studiengängen zugeordnet und werden mit Lehrveranstaltungen verknüpft. Exklusivität der Zuordnung besteht dabei an keiner Stelle. Sowohl Module als auch Studiengänge besitzen jeweils Verantwortliche, die über die Zuordnung von Modulen zu Studiengängen zu entscheiden haben. Die Zuordnung kann dabei von beiden Seiten erfolgen, wobei der jeweils andere Verantwortliche die Verknüpfung bestätigen und letztlich speichern muss, damit sie zustande kommt.

Eine weitere Einschränkung für die Bearbeitung von sowohl Veranstaltungen, Modulen, Studiengängen und Instituten ist die Zugehörigkeit zu bzw. Verantwortlichkeit für diese. So lassen sich Institute nur von Personen bearbeiten, die ihnen angehören und Module nur von den Angehörigen des Instituts verändern, das für das Modul zuständig ist. Auch diese Anforderung sollte im System umgesetzt werden können.

11.2.1. Datenmodell

Das Datenmodell leitet sich direkt aus den Anforderungen ab. Es existiert eine Klasse `Person` zur Abbildung von Personen, die auch als Nutzerklasse deklariert ist. Davon abgeleitet ist eine Klasse `Staff`, die Bedienstete der Universität abbildet und entsprechend noch weitere Attribute besitzt. Eine Klasse `Institution` repräsentiert eine Institution innerhalb der Universität. Ihr zugeordnet können sowohl `Module`-Objekte, `Lecture`-Objekte als auch Personen in Form von `Staff`-Objekten sein, wobei jede dieser Verbindungen bidirektional ist. Zusätzlich existiert mit `CourseOfStudy` eine Klasse, die Studiengänge abbildet. Diese besitzen, ebenso wie Module und Lehrveranstaltungen auch, beliebig viele Verantwortliche bzw. Lehrende, die diesen zugeordnet werden können.

11.2.2. Applikationsmodell

Aufgrund der Bedeutung, die es auch bei der Ausführung von Aktivitäten hat, wird das Applikationsmodells an dieser Stelle bereits erläutert.

In der Modulhandbuch-Anwendung existieren insgesamt vier Nutzergruppen, die jeweils unterschiedliche Berechtigungen und Funktionen besitzen.

Der *Gastnutzer* kann auch in dieser Anwendung lediglich seine Registrierung vornehmen.

Ein Nutzer, der die Rolle des *Bearbeiters* (`editor`) übernimmt, agiert als ein Mitglied eines oder mehrerer Institute und kann damit sämtliche Module und Veranstaltungen bearbeiten, die dem Institut zugeordnet sind. Zudem kann er neue Objekte dieser Typen anlegen. Dementsprechend sind auch die Rechte zum Anlegen dieser Objekte gesetzt.

Ein *Studiengangsverwalter* (`manager`) ist für einen oder mehrere Studiengänge verantwortlich und kann daher sowohl neue Studiengänge anlegen als auch über die Module entscheiden, die seinen Studiengängen zugeordnet wurden bzw. werden sollen.

Schließlich existiert ein *Administrator*, der für sämtliche Klassen die Administrationsrechte und damit Zugriff auf die Standardseiten mit der zugehörigen CRUD-Funktionalität besitzt. Darüber ist auch die Zuordnung von Gruppen zu bestimmten Nutzern möglich, so dass die Nutzer in den entsprechenden Rollen agieren können.

Neben den nutzergruppenspezifischen Rechten ist es prinzipiell jedem Nutzer erlaubt, zumindest lesenden Zugriff auf Nutzerdaten, Institutionen, Lehrveranstaltungen, Module und Studiengänge zu haben. Anzumerken ist weiterhin, dass die Nutzerrechte sich nicht ausschließen. Sollte ein Nutzer sowohl Bearbeiter- als auch Managerrechte für Studiengänge besitzen, so werden sämtliche Berechtigungen dieser beiden Rollen gewährt.

11.2.3. Aktivitäten

Auch in der Modulhandbuch-Applikation wird die Funktionalität über Aktivitäten abgebildet. Kurze Erläuterungen dazu folgen in diesem Abschnitt.

Registrierung von Nutzern

Auch für das Modulhandbuch sollten sich Nutzer registrieren können. Dies geschieht ebenfalls durch einen ähnlichen Ablauf wie bei der Registrierung in der Bibliotheksanwendung (vgl. Abschnitt 11.1). Neben dem Unterschied, dass andere Attribute anzugeben sind, findet auch eine Gruppenzuordnung erst durch den Administrator statt. Dies ist durch die Auswirkungen bedingt, die eine automatische Zuordnung hätte. Ein Nutzer könnte direkt sämtliche Datenobjekte seines Instituts bearbeiten, was ohne vorherige Kontrolle bzw. korrekte Freigabe seitens eines Instituts nicht gewünscht ist.

Anlegen neuer Objekte verschiedenen Typs

Zum Anlegen neuer Objekte existieren jeweils eigene Aktivitäten. Sowohl Veranstaltungen als auch Module lassen sich einfach durch Ausführung des entsprechenden Ablaufs anlegen. Dabei werden diese Abläufe lediglich denjenigen Nutzern angezeigt, die zur Nutzergruppe der Bearbeiter gehören.

Auch Studiengänge lassen sich durch einen einfachen Ablauf erzeugen. Dieser ist jedoch den Studiengangsmanagern vorbehalten.

Neue Nutzer im Modulhandbuchsystem werden durch Registrierung angelegt. Neue Institutionen werden generell durch den Administrator erzeugt, da diese organisatorische Aufgabe die Universitätsstruktur abbildet und damit nicht von jedem beliebigen Nutzer übernommen werden kann.

Bearbeiten von Objekten verschiedenen Typs

Ebenso wie das Anlegen von Objekten ist das Bearbeiten den jeweiligen Nutzergruppen vorbehalten. Bearbeiter sind für Module und Veranstaltungen zuständig und Studiengangsverantwortliche können Studiengangsdaten aktualisieren. Der Zugriff auf die jeweiligen Daten ist jedoch stets abhängig von der Zugehörigkeit des Bearbeiters. So lassen sich lediglich Module und Veranstaltungen bearbeiten, die zu den gleichen Institutionen wie der Bearbeiter gehören. Das Bearbeiten eines Moduls beispielsweise ist in Abbildung 11.2 dargestellt.

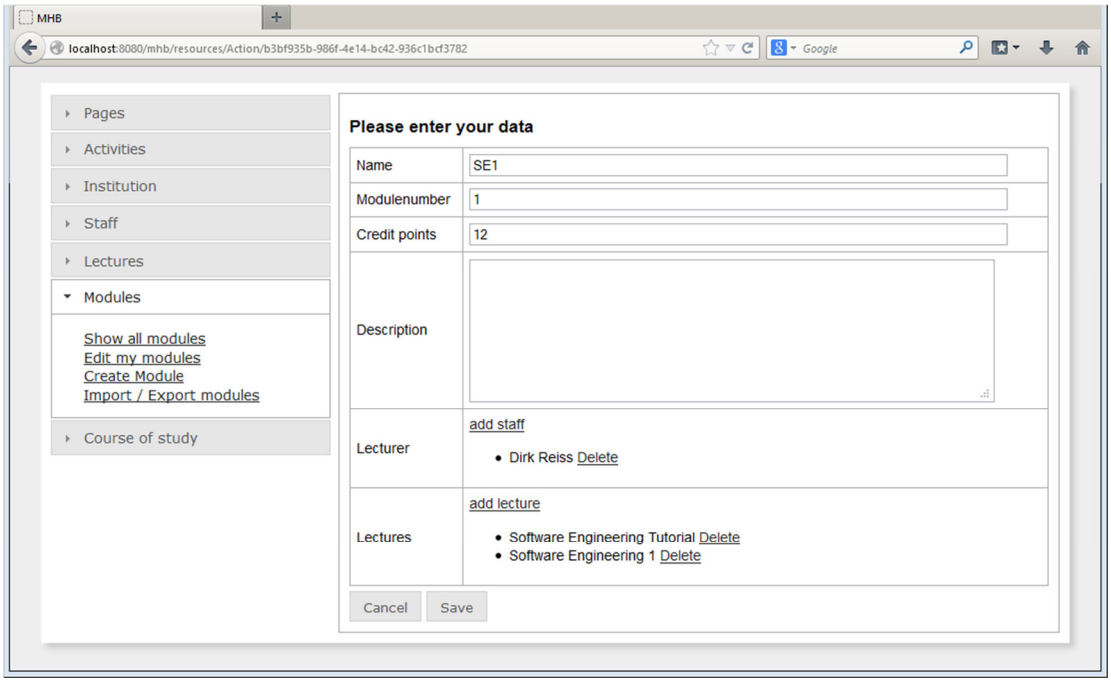


Abbildung 11.2.: Bearbeitungsmaske für Module im Modulhandbuchsystem

Gleiches gilt für Studiengänge. Hier müssen Nutzer mit Rolle Studiengangsverantwortliche auch als Verantwortliche zugeordnet sein um Änderungen vornehmen zu können.

Die Bearbeitung von Institutionen ist ebenfalls den zugehörigen Bearbeitern vorbehalten. Wie bereits erwähnt, lassen sich diese Objekte jedoch nicht beliebig von jedem beliebigen Bearbeiter ändern. So sind zwar Angaben wie die Adresse volatil, der Name beispielsweise jedoch nicht. Dementsprechend ist auch die Bearbeitungsmaske gestaltet, wie Abbildung 11.3 zeigt.

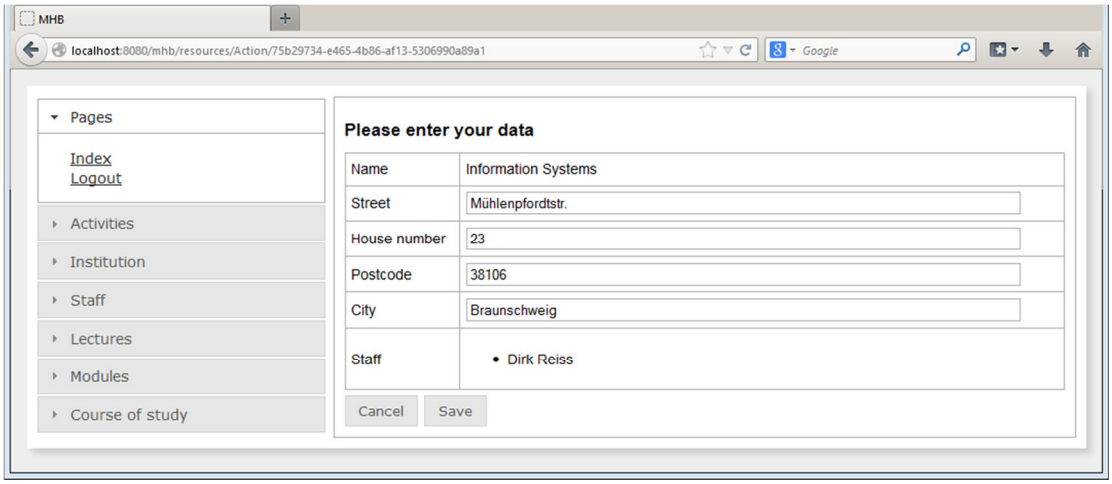


Abbildung 11.3.: Editormaske für eine Institution im Modulhandbuchsystem

Verknüpfung von Modulen und Studiengängen

Das *Exportieren* von Modulen *an* Studiengänge bzw. das *Importieren* von Modulen *in* Studiengänge ist ein relativ komplexer Prozess in der Modulhandbuchapplikation. Diese Aktivität kann von beiden Nutzergruppen aufgerufen werden.

In einem ersten Schritt wird überprüft, welcher Nutzergruppe der aktuelle Nutzer angehört. Sollte er beiden Gruppen angehören, so wird er auf einer Seite um eine Entscheidung gebeten, welche Zuordnung er durchführen möchte. Anderenfalls wird er direkt zu der Seite geleitet, die seiner Rolle entspricht.

Im Fall, dass ein Bearbeiter den Ablauf startet (oder der Nutzer diese Rolle ausgewählt hat), werden ihm seine Module präsentiert, von denen er eines auswählen kann. In einem nächsten Schritt erfolgt die Auswahl des Studiengangs, in dem das Modul eingebunden werden soll, wonach die Arbeit für den Bearbeiter abgeschlossen ist. Der Ablauf kann nun durch den zugehörigen Studiengangsbearbeiter fortgesetzt werden. In diesem Fall wird das Modul präsentiert, in dem die vorher gesetzte Zuordnung dargestellt wird. Dieses kann nun akzeptiert und damit gespeichert oder aber abgelehnt und verworfen werden.

Der umgekehrte Fall tritt beim Import eines Moduls durch den Studiengangsmanager ein. Hierbei wird zunächst ein Studiengang und in einem nächsten Schritt das zu importierende Modul ausgewählt. Die Bestätigung wird von einem berechtigten Bearbeiter des zugehörigen Instituts verlangt.

11.3. Robotersteuerung

In der Arbeit von Dominik Merkelbach [Mer14] wurde MontiWIS verwendet, um die Steuerung von Robotern in einem Krankenhaus zu initiieren. Dabei geht es im Wesentlichen darum, eine bestimmte Aufgabe für einen Roboter auszuwählen, diese Aufgabe zu parametrisieren und letztlich an den Roboter zu schicken.

11.3.1. Datenmodell

Das Datenmodell der Anwendung besteht einerseits aus der Nutzerdatenbasis, wobei hier zwischen einer Krankenschwester und einem Administrator unterschieden wird, was sich in unterschiedlichen Attributen ausdrückt, die eine Person neben den ihr gemeinen Attributen besitzt. Die Spezialisierung wird hierbei über Vererbung realisiert.

Ferner existieren im Domänendatenmodell Klassen für Aufgaben, deren Parameter und Werte, die diese Parameter besitzen können. Eine Aufgabe kann dabei etwas wie das Fahren des Roboters zu einem bestimmten Platz zu einer bestimmten Zeit sein, wobei sowohl Ort als auch Zeit als Parameter abgebildet werden.

11.3.2. Aktivitäten

Die manuell zu implementierende Geschäftslogik besteht bei dieser Applikation im Wesentlichen aus einer Aktivität, die zur Auswahl einer Aufgabe, der Belegung ihrer Parameter mit konkreten Werten und schließlich das Absenden dieser Aufgabe an einen Roboter dient. Die Auswahl und Belegung erfolgt über originäre MontiWIS Funktionalität, das Senden der Aufgaben an den Roboter geschieht über Remote Method Invocation (RMI), was eine direkte Ansteuerung der externen Roboterinfrastruktur erlaubt.

Weitere Funktionalitäten innerhalb dieser Webanwendung, etwa zum Anlegen und Bearbeiten der Datenobjekte, werden über die standardmäßig generierten Seiten zur Bearbeitung der Objekte der Domänendatenstruktur realisiert, weswegen hierzu kein weiterer Aufwand notwendig war.

11.3.3. Applikationsmodell

Für die Robotersteuerung existieren lediglich zwei Arten von Nutzern. Einerseits soll die Steuerung direkt von nicht-eingeloggten Nutzern vorgenommen werden können, weswegen die Aktivität dazu von einer *Gastnutzerin* ausgeführt werden darf. Andererseits werden die administrativen Aufgaben wie Aufgaben- und Parameterverwaltung von einem *Administrator* wahrgenommen, weswegen sie auch einem Admin-Nutzer zugeordnet werden.

Dir Darstellung erfolgt wie in MontiWIS üblich über ein Menü, das diese Berechtigungen entsprechend umsetzt.

11.4. Weitere Verwendung

MontiWIS wurde im Rahmen zweier studentischer Arbeiten eingesetzt, um Feedback in der generierten Applikation zu sammeln. Diese beiden Arbeiten sowie ein Prototyp eines Online-Demonstrators werden im Folgenden kurz vorgestellt.

11.4.1. Feedback bezüglich statischen Komponenten von MontiWIS

Im Rahmen der Masterarbeit von Ludger Steens [Ste12] wurde MontiWIS eingesetzt, um Nutzerfeedback zu sammeln, welches primär die statischen Komponenten von MontiWIS betrifft.

Dabei war es einem Nutzer möglich, auf Seiten zu spezifizieren, welche Attribute in welcher Art und Weise geändert werden sollen. Dazu lassen sich auf der Seite bestimmte Elemente markieren und über das bereitgestellte Feedback-System Anmerkungen dazu spezifizieren. Diese Meldungen werden in einem Ticket im Feedback-System derartig gespeichert, dass ein Rückschluss auf die Modellelemente möglich ist, aus denen die betreffenden Artefakte der Seite generiert wurden.

Für die Implementierung dieser Erweiterung wurden an dezidierten Stellen im MontiWIS-Generator Erweiterungen vorgenommen, die eine Kommunikation mit Serverkomponenten ermöglichen, die die Zuordnung von Nutzerfeedback zu Modellelement vornehmen.

11.4.2. Feedback bezüglich dynamischer Komponenten von MontiWIS

Auf der in [Ste12] geschaffenen Infrastruktur setzt die Masterarbeit von Robert Eikermann [Eik13] auf. Die dort entwickelten Konzepte wurden jedoch derartig erweitert, dass sie Feedback sowohl auf Seitenelemente als auch auf einzelne Teile einer Aktivität zulassen. Auch hierbei lassen sich Teile der Seite markieren, die durch entsprechend hinzu generierte Informationen den ursprünglichen Modellelementen zugeordnet werden können.

Auch für die Bereitstellung dieser Funktionalität wurden lediglich marginale Änderungen am MontiWIS-Generator vorgenommen um eine Einbindung in die Feedback-System-Infrastruktur zu erlauben. Näheres zur Implementierung ist in [Eik13] zu finden.

11.4.3. Online-Demonstrator

Eine weitere Anwendung von MontiWIS ist ein (intern am Lehrstuhl für Software Engineering) in der Entwicklung befindlicher Online-Demonstrator. Darin ist es aktuell möglich, aus einem Klassendiagramm über die im MontiWIS-Generator implementierte Standardfunktionalität eine CRUD-Anwendung zu generieren und direkt zu deployen. Die Unterstützung weiterer Modelle ist in der aktuellen Version nicht gegeben, ließe sich aber bei entsprechendem Ressourceneinsatz nachrüsten.

11.5. Bewertung der Ergebnisse

Abschließend gilt es, die Ergebnisse dieser Arbeit anhand der umgesetzten Anwendungen sowie die in den vorherigen Abschnitten aufgestellten Kriterien zu bewerten und zu evaluieren. Dazu werden in Abschnitt 11.5.1 die Mengengerüste geschriebenen und generierten Codes betrachtet und anschließend in Abschnitt 11.5.2 die Ergebnisse hinsichtlich der aufgestellten Kriterien evaluiert. Abschließend werden in Abschnitt 11.5.3 die Anforderungen an Modelle und die Unterstützung bei der Verarbeitung und in Abschnitt 11.5.4 schließlich die Designentscheidungen bei der Auswahl von Technologien und der Implementierung reflektiert und evaluiert.

11.5.1. Quantitative Bewertung

Für die Abschätzung, welcher Produktivitätsgewinn durch den generativen Ansatz im Fall von MontiWIS zu verzeichnen ist, werden in diesem Abschnitt die Anzahl der Codezeilen gegenübergestellt, die geschrieben und als Ergebnis generiert wurden. Zunächst werden jedoch die Codezeilen (Lines of Code (LOC)) betrachtet, die für die Entwicklung von MontiWIS geschrieben bzw. generiert wurden. Einen Überblick darüber ist in Tabelle 11.4 dargestellt.

Projekt	LOC Model	LOC generiert	LOC handgeschrieben
`MontiWIS`	1026	176976	30296
`MontiWIS-Target`			4251

Tabelle 11.4.: Lines of Code für MontiWIS-Projekte

Anzumerken ist hierbei, dass die Zeilenangaben exklusive der Klassendiagramme, der UML/P Common und Java-Grammatiken berechnet wurden, da diese unverändert in MontiWIS übernommen wurden.

Die Mengengerüste für Modellelemente in Projekten, in denen MontiWIS-Modelle zum Einsatz kommen und Web-Informationssysteme generiert werden, ist in 11.5 dargestellt.

Projekt	LOC Model	LOC Java (gen.)	LOC JSP (gen.)	LOC gen. gesamt
Bibliothekssystem	838	12822	5082	17904
Modulhandbuch	933	17581	9232	26813

Tabelle 11.5.: Lines of Code für MontiWIS Anwendungen

Ein Vergleich zwischen der Anzahl Zeilen Modell und der Anzahl Zeilen generierter Code (sowohl Java als auch JSP) ist in Abbildung 11.6 dargestellt.

Ein Vergleich der hier erzeugten Codemenge mit vollständig handgeschriebenen Applikationen ist generell schwierig (vgl. z. B. [FRS13]), insbesondere da für handgeschriebene Anwendungen in der Regel Frameworks zum Einsatz kommen, bei denen bestimmte Funktionalität als Teil des Frameworks geliefert wird. Dies ist erfahrungsgemäß beispielsweise bei Apache Tapestry [Tap] der Fall, wo Teile der Mächtigkeit des Frameworks in den Classloader integriert sind. Aus diesem Grund lässt sich die Zeilenanzahl in MontiWIS nicht ohne Weiteres mit denen anderer Applikationen vergleichen.

Dennoch lassen sich, wie in den Übersichten mit den Codezeilen ersichtlich, erhebliche Produktivitätsgewinne erzielen. Zwar sind in der Regel Zeilen von Modellcode nicht gleichwertig zu Codezeilen in einer generischen Programmiersprache (wie z. B. Java), sondern i. d. R. schwergewichtiger als diese, jedoch sind die Sprachelemente an die Syntax gewöhnlicher Programmiersprachen angelehnt bzw., im Fall von Java-Programmcode in Aktionen, direkt davon übernommen.

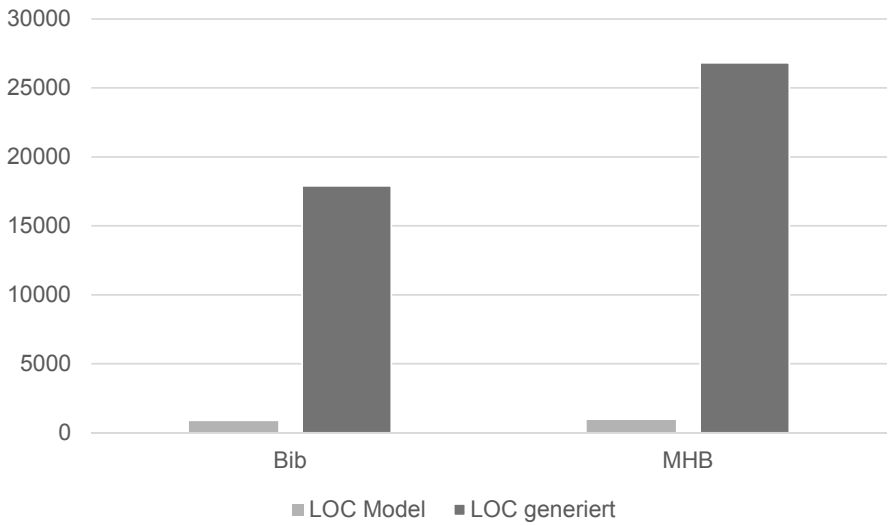


Abbildung 11.6.: Vergleich Modell- und generierter Code

11.5.2. Qualitative Bewertung

In diesem Abschnitt werden, ausgehend von den Kriterien aus Abschnitt 2.2, die Anforderungen an ein Web-Informationssystem evaluiert. Auf die Anforderungen an die einzelnen Sprachen wird in den entsprechenden Unterabschnitten kurz eingegangen.

Eingabe und Speicherung von Daten

Die zentrale Aufgabe eines Web-Informationssystems, nämlich die Speicherung, Verarbeitung und Verwaltung von Daten in einem webbasierten System, wird in jeder der besprochenen Applikationen erfüllt. Die Modellierung erfolgt über eine Sprache, die objektorientierte Konzepte wie Vererbung, Assoziationen und Kompositionen unterstützt. Die geforderten und in Klassen verwendeten Datentypen wie beispielsweise E-Mail-Adresse oder Datum werden unterstützt und entsprechend in der Oberfläche berücksichtigt, wobei dort eine flexible Kombination von verschiedenen Datentypen erlaubt ist. Auch die geforderten Standardansichten zur Bearbeitung einer jeden Domänenklasse lassen sich erzeugen und über minimalen Aufwand in den Applikationen verwenden. Diese werden in den betrachteten Anwendungen primär in Administrationsansichten genutzt, wofür sie prädestiniert sind.

In MontiWIS ist es relativ einfach möglich, innerhalb der eigentlichen Applikationslogik durch Kommandos auf persistente Daten zuzugreifen. Ferner ist aber auch der Zugriff über die generierten `DAO`-Klassen aus externem Code heraus möglich, was in den betrachteten Anwendungen insbesondere beim Import von Daten zum Einsatz kommt.

Darstellung von statischen und dynamischen Seiten

Über die Seitenbeschreibungssprache ist es möglich, sowohl statische Anteile von Seiten als auch dynamische, d. h. solche, die aus den Objekten des Domänendatenmodells zusammengestellt werden, zu spezifizieren. Dieser Zugriff erfolgt sowohl lesend als auch schreibend, wobei die Daten zur Weiterverarbeitung auf Serverseite übermittelt werden. Über die Seitenbeschreibungssprache lassen sich ebenfalls Teilsichten auf Attribute aus unterschiedlichen Objekten beschreiben, wie dies in den Anforderungen an die Sprache gefordert wurde. Hinsichtlich des Sprachdesigns wurde auf Abstraktion von Implementierungsdetails und kompakte Syntax geachtet. Dennoch wurde größtmögliche Flexibilität durch die Möglichkeit einer Kombination von dynamischen Inhalten mit HTML erreicht, was an einzelnen Stellen von den bisherigen Anwendern als positiv empfunden und intuitiv verstanden wurde.

Abläufe über mehrere Webseiten hinweg

Die Anforderung, Abfolgen zwischen Seiten in Form von Workflows als ein vorrangiges Mittel zur Beschreibung der Abläufe in einem Web-Informationssystem zu sehen, erwies sich als sinnvolle Entscheidung. In den umgesetzten Beispielanwendungen ließen sich darüber im Wesentlichen sämtliche benötigten Funktionalitäten der zu entwickelten Systeme abbilden. Durch die Verwendung von Aktivitätsdiagrammen und die darin unterstützten Verzweigungen mit und ohne Bedingungen lassen sich die Anforderungen der manuellen und automatischen (d. h. durch Auswertung von automatischen Berechnungen oder eingegebenen Daten ermittelten) Ablaufentscheidungen sehr gut abbilden. Auch zwischen Nutzern verteilte Abläufe lassen sich mit den Modellelementen der Aktivitätsdiagramme problemlos umsetzen. Die Anforderungen hinsichtlich möglicher Geschäftslogik im weitesten Sinne (also den Aufruf von Seiten, den Aufruf von Metafunktionalität sowie die im nächsten Abschnitt betrachtete Ausführung beliebigen Codes) sind durch die verschiedenen Möglichkeiten, Aktionen mit Logik zu versehen, vollständig erfüllt. Das Konzept wurde bei der Umsetzung der beschriebenen Web-Informationssysteme als sinnvoll erachtet und nach kurzer Einarbeitung selbstständig zur Implementierung der Systeme verwendet.

Ausführung beliebigen Codes

Eine zentrale Anforderung an die Web-Informationssysteme, die in MontiWIS entwickelt werden sollten, ist die Ausführung beliebigen Codes als Teil der Geschäftslogik der Anwendung. Dies wurde durch die Verwendung von Java als Aktionssprache innerhalb von Aktionen in Aktivitätsdiagrammen erreicht. Einerseits lässt sich darüber beliebige Geschäftslogik direkt vor und nach dem Aufruf von Seiten ausführen, andererseits aber auch komplexere Logik, die extern definiert wurde, in die Abläufe einbinden.

Extensiv eingesetzt wurde diese Möglichkeit insbesondere beim Import von Daten, wo aus verschiedenen Datenquellen (Datei, Datenbank) Daten gelesen und in die modellierte und generierte Domänendatenstruktur übertragen und gespeichert wurden.

Nutzerspezifische Sichten auf die Applikation

Die Umsetzung von nutzerspezifischen Sichten auf die Applikation wurde durch die Applikationssprache realisiert. Darin lassen sich höchst einfach und flexibel Nutzergruppen und deren Berechtigungen definieren, was für Seiten, Aktivitäten und die Standardansichten möglich ist. In den entwickelten Anwendungen werden diese Möglichkeiten genutzt, so dass beispielsweise Administrationsansichten oder unterschiedliche Sichten für Studierende und Angestellte umgesetzt werden konnten.

Einheitliche Darstellung und Bedienung

Bei der Forderung nach einheitlicher Darstellung und guter Bedienbarkeit wurde auf allgemeine Best Practices und weit verbreitete Frameworks gesetzt, über die das globale Aussehen der Applikation einfach angepasst werden kann. Ferner wurde darauf geachtet, dass relativ einfach Beschriftungen von Knöpfen und Bezeichnungen von Feldern angepasst werden können. Diese Anpassungen wurden in den umgesetzten Applikationen genutzt und als sehr einfach und hilfreich empfunden.

11.5.3. Modell- und Infrastrukturanforderungen

Die in Abschnitt 3.4 spezifizierten Anforderungen an die Sprachen können, wie soeben diskutiert, als erfüllt angesehen werden. Zu betrachten bleiben schließlich noch die Anforderungen an die Modellverarbeitung und die Unterstützung der Nutzer.

Die Interdependenzen zwischen den einzelnen Modellen wurden durch die Implementierung der Symboltabelleninfrastruktur [Völ11] realisiert. Auch die geforderten Kontextbedingungen wurden unter Verwendung der dort angebotenen Mechanismen implementiert. Zwar ließen sich eine Reihe weiterer Kontextbedingungen erdenken, die ebenfalls bei der Entwicklung geprüft werden könnten, jedoch bieten die bereits in den einzelnen vorherigen Kapiteln beschriebenen und implementierten Überprüfungen einen soliden Grundstock für ein funktionierendes und sinnvolles System.

Die ebenfalls angestrebte Integration von nutzerunterstützenden Funktionalitäten in den Modelleditor wurde ebenfalls realisiert. Hierzu wurden die Mechanismen verwendet, die in [KRV07a] konzipiert und in [Kra10, Völ11] implementiert und beschrieben wurden. Dazu waren lediglich einige wenige Modellzeilen notwendig, aus denen ein entsprechendes Plugin für die Eclipse-Entwicklungsumgebung generiert wurde. Das Ergebnis ist in Abbildung 11.7 dargestellt.

Im Hauptfenster ist zu sehen, dass die Schlüsselwörter der Aktivitätsdiagrammsprache entsprechend hervorgehoben wurden. Ebenfalls dargestellt ist ein Fehlermerker, der aus der inkorrekten Schreibweise des Schlüsselworts `activity` resultiert, was zu einem syntaktisch inkorrekten Modell führt. Schließlich sind auf der rechten Seite die einzelnen Aktionen in einer sogenannten *Outline* dargestellt, die einen Überblick über die Struktur des Modells liefert.

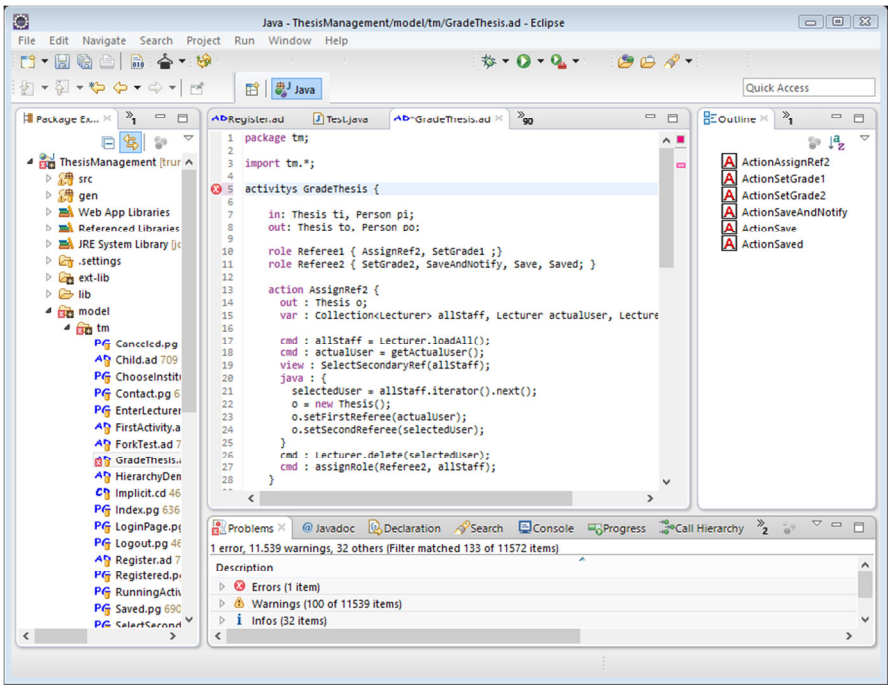


Abbildung 11.7.: Editoransicht für Aktivitäten in Eclipse

11.5.4. Bewertung der Designentscheidungen

Neben der Bewertung der Kriterien, die für die generierte Anwendung, die Modelle und die Methodik aufgestellt wurden, sind auch die Entscheidungen bei der Wahl der Werkzeuge zu betrachten und zu bewerten, was im folgenden Abschnitt geschieht.

Verwendung von MontiCore

Die Entscheidung, bei der Definition von Sprache und deren Verarbeitung auf MontiCore zu setzen, stellte sich letztlich aus richtig heraus. Da als Zielgruppe von MontiWIS Softwareentwickler identifiziert wurden, ist die Wahl, textuelle Sprachen zu nutzen, sehr passend und zielführend. Das Erstellen und die Verwaltung von Modellen lässt sich darüber sehr effizient bewerkstelligen, wie beispielsweise in [KRV08] diskutiert wurde. Da bei der Beschreibung des Web-Informationssystems mehrere Sprachen zum Einsatz kommen, stellte sich die Modularität der Sprachen und die bestehende Symboltabelleninfrastruktur als sehr hilfreich heraus. Einerseits konnten über Sprachvererbung bereits definierte Konzepte (z. B. UML/P Common, siehe C.3) wiederverwendet werden. Andererseits ließ sich durch die Einbettung von Sprachen (z. B. Java in Aktionen der Aktivitätsdiagramme) die bereits definierte Grammatik von Java perfekt als Aktionssprache nutzen, was eine Integration von externem Code ebenfalls sehr erleichtert.

Verwendung einer Zwischenstruktur

Trotz der Verwendung von MontiCore wurde darauf verzichtet, die AST-Struktur, die beim Parsen der Modelle entsteht, zu modifizieren und schließlich als Eingabe zur Codegenerierung zu verwenden, wie dies beispielsweise in der Arbeit von Martin Schindler geschieht [Sch12]. Ein Grund dafür war die relativ tief verschachtelte AST-Struktur, die ein Navigieren entlang der Hierarchien darin sehr beschwerlich und umständlich machte. Durch die Zwischenstruktur ist es von vornherein möglich, relativ implementierungsnahe Zugriffsmethoden anzubieten, die sich gut in den Templates ansprechen lassen, so dass diese übersichtlich und kompakt gehalten werden können. Ein weiterer Grund für die Verwendung der Zwischenstruktur sind die Umbauten und das hinzu generierte Standardverhalten, das letztlich ebenfalls in den Templates verarbeitet und in der generierten Applikation umgesetzt wird. So werden aus Klassen beispielsweise diverse Seiten inklusive zugehöriger `PageEntity`-Klassen, Ressourcenklassen und JSP-Seiten generiert. Dieses Verhalten ließe sich zwar ebenfalls in AST-Form umsetzen, was jedoch aus genannten Gründen ebenfalls relativ umständlich wäre. Durch die Verwendung der Zwischenstruktur ließ sich ebenfalls eine Trennung erreichen zwischen Informationen, die zur Modellstruktur gehören, und solchen, die die Struktur der generierten Applikation abbilden, was sich im Aufbau und den angebotenen Methoden der Datenstruktur widerspiegelt.

Wahl der Zielplattform

Auch die Verwendung der Kombination von Java EE, JPA, JSP sowie der im Rahmen von MontiWIS entwickelten Ablaufsteuerung erwies sich letztlich als gute Entscheidung.

Die Entscheidung zur Verwendung dieser Technologien wurde primär aufgrund der hohen Freiheitsgrade getroffen, die diese bieten. Technologien wie beispielsweise JSF (oder Apache Tapestry wie im Fall des Vorgängerprojekts MontiWeb) bieten zwar einige vorgefertigte Komponenten an, die jedoch nicht vollständig dem entsprechen, was in MontiWIS anvisiert wurde. So lassen sich etwa Seitenabfolgen und Navigation nicht ohne Weiteres mit dem Paradigma der Aktivitätsdiagramme mit seinen verteilten Abläufen, der Möglichkeit der späteren Fortführung von Abläufen und der Seitenintegration vereinbaren, wie es in MontiWIS angestrebt wurde. Änderungen an den Frameworks wären zwar möglich, aber sehr aufwendig gewesen. Ferner lässt sich die weitere Entwicklung dieser Frameworks relativ schwer vorhersehen, so dass potenzielle Änderungen stets in späteren Versionen nachgezogen werden müssten.

Auch durch die Wahl von JSP zur Darstellung von Nutzerschnittstellen war es weitestgehend möglich, die Darstellung und die Funktionalität auf Clientseite entsprechend der Anforderungen und Vorstellungen in MontiWIS umzusetzen. Auch wenn JSP und die Funktionalität, die durch die `PageEntities` in MontiWIS realisiert wurde, potenziell mehr Code erfordert als bei Verwendung von Tapestry oder JSF, wo dies Teil des Frameworks ist, lässt sich dieser Nachteil durch die Verwendung des Codegenerators ausgleichen. Auch die clientseitigen Komponenten dynamische Tabellen lassen sich durch die Verwendung von JQuery(UI) ebenfalls sehr einfach realisieren.

Die Verwendung von JPA als Framework zum objektrelationalen Mapping hat sich insofern als positiv herausgestellt, als das Laden und Speichern dadurch in der Regel sehr vereinfacht

wurde im Gegensatz zur Verwendung von purem SQL zur Ansteuerung der Datenbank. Zwar sind durch diese eingezogene Abstraktionsschicht nicht sämtliche Optimierungspotenziale ausgeschöpft, die pures SQL und sehr angepasste Datenbankstrukturen böten, jedoch würde eine solche Optimierung auch wesentlich mehr Informationen auf Modellebene erfordern, die eben auf solche Potenziale schließen lassen. In der aktuellen Implementierung lassen sich keine Defizite oder Bottlenecks erkennen, die eine solche Optimierung nötig machen würden.

Kapitel 12.

Zusammenfassung und Ausblick

Im Rahmen der vorliegenden Arbeit wurde mit MontiWIS ein Werkzeug entwickelt, über das sich Web-Informationssysteme spezifizieren und generieren lassen. Dafür wurden sowohl die Sprachen zur Beschreibung des Systems, die zugehörigen Kontextbedingungen zur weiterführenden Überprüfung der Sinnhaftigkeit der Modelle, sowie die Generatoren zur Erzeugung des Web-Informationssystems implementiert. Auch die Konzepte für die generierte Applikation wurden entworfen und umgesetzt.

In diesem Abschnitt sollen die erreichten Ziele rekapituliert werden, was in Abschnitt 12.1 geschieht. Ferner werden in Abschnitt 12.2 Hinweise auf weitere Forschungsrichtungen gegeben, die nach Abschluss dieser Arbeit eingeschlagen werden können.

12.1. Zusammenfassung

In der Arbeit wurde in Kapitel 1 zunächst die Arbeit motiviert und einige Ziele definiert, die im Rahmen der Arbeit erfüllt werden sollten. Zusätzlich wurde ein Überblick über den weiteren Aufbau der Ausarbeitung gegeben. Inwieweit die dort gestellten Anforderungen als erfüllt betrachtet werden können, wurde bereits in Abschnitt 11.5.2 untersucht.

In Kapitel 2 wurde detaillierter auf die Domäne eingegangen, für die in der vorliegenden Arbeit Abstraktionen entwickelt wurden. Dazu wurden dort Anforderungen festgelegt, die das zu generierende System charakterisieren. Hinsichtlich möglicher Implementierungsarten und -methoden wurden verschiedene Ansätze vorgestellt, die aktuell populär und verbreitet sind. Die Ausführungen dort grenzen die Systeme ein, die mit MontiWIS entwickelt werden sollen und spezifizieren, auf welche Aspekte Wert gelegt wird und wo die Grenzen der Systeme liegen.

Nach der Definition der Zieldomäne wurde in Kapitel 3 die modellgetriebene generative Softwareentwicklung erläutert. Hier wurden verschiedene Ansätze vorgestellt, die sich insbesondere hinsichtlich ihrer Modellarten (grafisch, textuell) unterscheiden. Im Fall von MontiWIS wurde MontiCore als Framework zur Definition und Verarbeitung der Modelle gewählt. Mit dieser Entscheidung fiel auch die Wahl auf textuelle Modelle, was einher geht mit der Festlegung von Softwareentwicklern als Zielgruppe der Anwendung. Die grundlegende Funktionsweise von MontiCore wurde eingeführt, so dass die in MontiWIS verwendeten Konzepte nachvollziehbar sind. Ferner wurden Anforderungen an die Modelle festgelegt, die bei der Definition der Sprachen berücksichtigt werden sollten. Schließlich wurden dort andere Ansätze vorgestellt, die ebenfalls modellbasierte Techniken verwenden, um Webanwendungen

zu beschreiben und partiell auch zu generieren. Im Vergleich zu einem Großteil der dort beschriebenen Ansätze wird bei MontiWIS explizit Wert auf die vollständige Generierung einer Anwendung mit geringst möglichem Aufwand und größtmöglichem Standardverhalten gelegt.

In Kapitel 4 wurde konkret auf den Aufbau des MontiWIS-Projekts und die verschiedenen Komponenten eingegangen. Dazu wurden einerseits die Komponenten und deren Aufgaben und Funktionalitäten erläutert, andererseits aber auch die Abläufe und Zusammenhänge beschrieben, die im Zuge der Modellverarbeitung und Generierung ausgeführt werden. Dort wurde ebenfalls motiviert, warum abweichend von der Architektur, wie sie in [Sch12] vorgeschlagen wurde, eine explizit auf die Generierung abgestimmte Zwischenstruktur eingesetzt wird.

Anschließend wurde in Kapitel 5 überblicksartig die Architektur des generierten Systems beschrieben. Dazu wurde insbesondere ein Augenmerk auf die Technologien gelegt, die in der Webanwendung zum Einsatz kommen. Ferner wurden die Entscheidungen für den Einsatz dieser Techniken erläutert. Dieser Überblick diente primär zur Einordnung der detaillierteren Erläuterungen in den nachfolgenden Kapiteln, die zum Teil auf Zusammenhänge aufsetzen, die in diesem vorgelagerten Kapitel beschrieben wurden.

Die Kapitel 6-9 sind im Wesentlichen ähnlich aufgebaut. Dort wurden zunächst die Anforderungen an die jeweils betrachtete Sprache definiert. Im Anschluss daran wurde die Sprache erläutert, wobei zunächst kurz auf den Aufbau der Sprache anhand der spezifizierten Grammatik eingegangen wurde, um nachfolgend die Verwendung der Sprache an konkreten Beispielen zu erläutern. Der nächste Abschnitt befasste sich stets mit dem Generierungsprozess. Dabei wurden in einem ersten Schritt die Kontextbedingungen erläutert, die bei der Modellverarbeitung geprüft wurden. Anschließend wurde ein Überblick über die Zwischenstruktur gegeben, die bei der Modellverarbeitung aufgebaut wurde. Darauf aufbauend wurden die einzelnen Schritte der Modellverarbeitung beschrieben. Sofern vorhanden, folgte die Erklärung der Umstrukturierungen und Transformationen, die nach dem Aufbau der kompletten Zwischenstruktur darauf ausgeführt wurden. Der letzte Schritt, die eigentliche Generierung über Templates, wurde im letzten Teil des jeweiligen Abschnitts erläutert. Abschließend wurden die einzelnen Komponenten im generierten System und die konkrete Funktionsweise dieser Generate beschrieben. Als letzter Teil eines jeden Abschnittes erfolgte eine Diskussion ähnlicher Ansätze, was sowohl die Konzepte der Modelle als auch die finale Umsetzung betraf.

Um die Anwendung des MontiWIS Ansatzes zu erleichtern, wurde auf die Anwendungsmethodik gesondert in Kapitel 10 eingegangen. Dabei wurde der Aufbau eines MontiWIS-Anwendungsprojekts erläutert, aber auch die einzelnen Schritte, die zur Generierung und Installation der Applikation notwendig sind. Auf die iterative Evolution der Anwendung und die Implikationen, die die Abhängigkeiten zwischen den Modellen darauf haben, wurde separat eingegangen. Ferner wurden die Elemente beschrieben, die zur Anpassung der generierten Anwendung vorgesehen sind.

Kapitel 11 schließlich beschrieb einige Fallstudien, die zur Verifizierung des Ansatzes und Demonstration seiner Möglichkeiten entwickelt wurden. Die Unterschiedlichkeit der generierten Systeme und die Nutzung des MontiWIS-Ansatzes einzig aufgrund vorhandener

Projekte und Erläuterungen durch den Autoren dieser Arbeit zeigte, dass sowohl die Flexibilität der Konzepte ausreicht, um verschiedenartige Projekte umzusetzen, als auch die Nutzung des Ansatzes selbst mit wenig Erläuterung möglich und erfolgreich durchführbar ist.

Mit MontiWIS ist ein Ansatz entstanden, mit dem sich effizient und flexibel Web-Informationssysteme entwickeln lassen. Allein die Definition eines Klassendiagramms in der an Java angelehnten Syntax der UML/P genügt, um ein vollständiges Web-Informationssystem zu erzeugen, mit dem diese Daten verwaltet werden können. Damit findet sich MontiWIS in einer Reihe mit einer Anzahl von Ansätzen, deren primärer Fokus die Erzeugung von CRUD-Anwendungen über sogenanntes Scaffolding ist. Die so erzeugte Funktionalität lässt sich auf einfache Art auch in späteren Schritten als Teil einer Administrationsansicht verwenden, so dass in dieser Hinsicht kein weiterer Aufwand zu treiben ist. Die Modellierung weiterer Teile der Anwendung kann ausgehend von der Festlegung unterschiedlicher Nutzergruppen und deren Anwendungsfälle im System als Teil des Applikationsmodells erfolgen, wobei die Grundstruktur des Systems bei Beachtung einiger weniger, in Abschnitt 10.3 erläuterter, Restriktionen iterativ wächst. Die Funktionen des Systems spiegeln sich in der Regel in Aktivitäten wider, die in Aktivitätsdiagrammen modelliert werden. Die Ausdrucksmittel dieser Sprache genügten in den modellierten Anwendungen, um sämtliche Sachverhalte abzubilden. Durch fortgeschrittene Konzepte wie verteilte Abläufe lassen sich auch komplexe Logiken modellieren, wie sie auch in ausgewachsenen Workflowmanagementsystemen ausgeführt werden könnten. Die in den Aktionen einer Aktivität verwendeten Java-Inhalte bieten die Flexibilität, um sowohl einfache Initialisierungen und Zuweisungen direkt lokal in den Aktionen auszuführen, aber auch extern definierte Geschäftslogik als Teil des Ablaufs anzubinden. Der Zugriff auf Serverfunktionalität durch eine Kommandosprache und der Aufruf von Seiten als Schnittstelle zum Nutzer der Anwendung werden ebenfalls als Teil von Aktionen spezifiziert. Die Seitenbeschreibung selbst war in den bisherigen Fällen umfassend genug, um die gewünschten Nutzerschnittstellen zu spezifizieren und auch der Zugriff darin auf die darzustellenden oder zu modifizierenden Daten erfolgt kompakt auf Basis ihres Datentyps. Die Komposition der Anwendung und die Konzepte der Nutzergruppen, die über die Applikationssprache spezifiziert werden, stellten sich als einfach zu nutzen und von ihrer Ausdrucksmächtigkeit her als hinreichend heraus.

Bei der Erstellung der Applikation helfen die definierten Kontextbedingungen, offensichtliche Fehler in den Modellen bzw. zwischen verschiedenen Modellen zu erkennen. Rein syntaktische Probleme werden direkt bei Eingabe im Editor erkannt und gemeldet.

Das generierte System ist von seiner Optik neutral gehalten, so dass es in unterschiedlichen Kontexten eingesetzt werden kann. Das Aussehen von Farben und Bedienelementen lässt sich einfach anpassen, so dass zumindest teilweise eine einheitliche Corporate Identity erzielt werden kann.

12.2. Ausblick

Während der Anwendung des MontiWIS-Ansatzes und insbesondere auch bei der Implementierung sowohl der Fallstudien als auch weiterer Systeme wurde deutlich, dass über die im Rahmen dieser Arbeit entwickelten Sprachen und Generatoren relativ einfach vollständige

Web-Informationssysteme beschrieben und generiert werden können. Dennoch wurde offenbar, dass sowohl bei der Anwendung direkt als auch hinsichtlich weiterführendem Einsatz einige Konzepte hilfreich wären.

Ein offensichtlich wünschenswerte Erweiterung ist weiterführende Nutzerunterstützung bei der Erstellung und Bearbeitung von Modellen. Kontextsensitive Hilfen innerhalb des Editors und direkte Rückmeldung bei Fehlern zwischen den Modellen wären sicherlich hilfreich für weitere Nutzerakzeptanz. Derartige Konzepte sollten jedoch direkt in die Generierung von Editoren einfließen und nicht explizit für jede Sprache und Sprachkombination erstellt werden müssen.

Eine hilfreiche Erweiterung betrifft die Möglichkeit, Komponenten von Modellen zu erstellen und diese in anderem Kontext wiederzuverwenden. Dazu wäre es nötig, Aktivitäten zusammen mit Seiten kombinieren zu können und ferner Anforderungen an die Datenstruktur zu stellen oder ableiten zu können, um die Funktionsfähigkeit derartiger Komponenten zu gewährleisten. Ein Anwendungsfall, der diskutiert wurde, ist beispielsweise die Implementierung und Weitergabe des Registrierungsablaufs. Da jedoch in jedem System unterschiedliche Datenstrukturen hinterlegt sind, unterschiedliche Gruppen initial zugeordnet werden und unterschiedliche Attribute auf den Seiten abgefragt werden, gestaltet sich die Übertragbarkeit eines solchen Ablaufs schwierig.

Ebenfalls wäre Unterstützung für Modellevolution wünschenswert, wie sie z. B. in [CDIP11] für WebML angedacht ist. Weiterentwicklung von Modellen kann in der Regel auf mehreren Ebenen passieren. Einerseits kann das Domänendatenmodell sich ändern, Klassen oder Attribute können hinzukommen oder umbenannt werden. Da in MontiWIS JPA zum Einsatz kommt, erfordert die Veränderung des Domänendatenmodells in der Regel die Neugenerierung der Datenbankstruktur, sollten keine fortgeschrittenen Konzepte zum Einsatz kommen. Stillschweigende Änderungen am Datenmodell führen daher zu Inkompatibilitäten, die eine einfache Datenübernahme unmöglich machen. Auch die Darstellung von Seiten kann sich ändern, es können mehr, weniger oder einfach andere Attribute und Klassen dargestellt werden, was sich auch auf Abläufe auswirken kann. Auch Abläufe in Form von Aktivitäten selbst können geändert werden, indem Aktionen hinzukommen, gelöscht werden oder sich die Geschäftslogik innerhalb von Aktionen ändert. Generell wäre es an dieser Stelle sinnvoll, die Änderungen je Modell per Delta der Modelle anzugeben und eine Transformation auf verschiedenen Ebenen durchführen zu können. Einerseits könnten darüber Datenbankstrukturen angepasst und migriert werden, andererseits ließen sich auch Abläufe aktualisieren und potenziell alte Abläufe sogar weiterhin fortführen.

Literaturverzeichnis

[ACM03] Alur, D.; Crupi, J.; Malks, D.: *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2.. Auflage, 2003.

[Act] Activiti Homepage. `http://activiti.org`, (Abruf: Dez 2015).

[ADH+09] Aschenbrenner, N.; Dreyer, J.; Hahn, M.; Jubeh, R.; Schneider, C.; Zündorf, A.: Building Distributed Web Applications based on Model Versioning with CoObRa: an Experience Report. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, S. 19–24. IEEE, 2009.

[AELH13] Aragón, G.; Escalona, M.-J.; Lang, M.; Hilera, J. R.: An Analysis of Model-Driven Web Engineering Methodologies. *International Journal of Innovative Computing, Information and Control*, Band 9, Nr. 1, S. 413–436, 2013.

[Ant] Ant Homepage. `http://ant.apache.org`, (Abruf: Dez 2015).

[Apa] Apache httpd Homepage. `https://httpd.apache.org/`, (Abruf: Dez 2015).

[ASU86] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers - Principles, Techniques, Tools*. Addison Wesley, 1986.

[Bau96] Baumgarten, B.: *Petri-Netze: Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, 2.. Auflage, 1996.

[Bay03] Bayern, S.: *JSTL in Action*. Manning, 2003.

[BBvB+] Beck, K.; Beedle, M.; Bennekum, A. v.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R. C.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D.: Agile Manifesto. `http://agilemanifesto.org`, (Abruf: Dez 2015).

[BC06] Bozzon, A.; Comai, S.: Conceptual Modeling and Code Generation for Rich Internet Applications. In Wolber, D.; Calder, N.; Brooks, C.; Ginige, A. (Hrsg.): *Proceedings of the 6th International Conference on Web Engineering - ICWE 2006*. ACM, 2006.

[BCC+02] Brambilla, M.; Ceri, S.; Comai, S.; Fraternali, P.; Manolescu, I.: Specification and design of workflow-driven hypertexts. *Journal of Web Engineering*, Band 1, Nr. 2, S. 163–182, April 2002.

[BCF06] Brambilla, M.; Ceri, S.; Fraternali, P.: Process Modeling in Web Applications. *ACM Transactions on Software Engineering and Methodology*, Band 15, Nr. 4, S. 360–409, 2006.

[BCFM08] Brambilla, M.; Comai, S.; Fraternali, P.; Matera, M.: Designing Web Applications with WebML and WebRatio. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, S. 221–262. Springer, 2008.

[BCMM06] Baresi, L.; Colazzo, S.; Mainetti, L.; Morasca, S.: W2000: A Modelling Notation for Complex Web Applications. In Mendes, E.; Mosley, N. (Hrsg.): *Web Engineering*, S. 335–364. Springer, 2006.

[Ber03] Bergsten, H.: *JavaServer pages*. O'Reilly Media, 2003.

[Bet13] Bettini, L.: *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, Birmingham, Mumbai, 2013.

[Béz04] Bézivin, J.: In Search of a Basic Principle for Model Driven Engineering. *Novatica Journal, Special Issue*, Band 5, Nr. 2, S. 21–24, 2004.

[BFH06] Barna, P.; Frasincar, F.; Houben, G.-J.: A Workflow-driven Design of Web Information Systems. In *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06, S. 321–328. ACM, New York, NY, USA, 2006.

[BKK11] Busch, M.; Knapp, A.; Koch, N.: Modeling Secure Navigation in Web Information Systems. In Grabis, J.; Kirikova, M. (Hrsg.): *10th International Conference on Business Perspectives in Informatics Research*, Nr. 90 der Reihe Lecture Notes in Business Information Processing, S. 239–253. Springer, 2011.

[BKS14] Busch, M.; Koch, N.; Suppan, S.: Modeling Security Features of Web Applications. In Heisel, M.; Joosen, W.; Lopez, J.; Martinelli, F. (Hrsg.): *Engineering Secure Future Internet Services and Systems*, Nr. 8431 der Reihe Lecture Notes in Computer Science, S. 119–139. Springer, 2014.

[BKVV08] Bravenboer, M.; Kalleberg, K. T.; Vermaas, R.; Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, Band 72, S. 52–70, 2008.

[Boc03a] Bock, C.: UML 2 Activity and Action Models. *Journal of Object Technology*, Band 2, Nr. 4, S. 43–53, 2003.

[Boc03b] Bock, C.: UML 2 Activity and Action Models Part 2: Actions. *Journal of Object Technology*, Band 2, Nr. 4, S. 27–41, 2003.

[Boc03c] Bock, C.: UML 2 Activity and Action Models Part 3: Control Nodes. *Journal of Object Technology*, Band 2, Nr. 6, S. 7–23, 2003.

[Boc04a] Bock, C.: UML 2 Activity and Action Models Part 4: Object Nodes. *Journal of Object Technology*, Band 3, Nr. 1, S. 27–41, 2004.

[Boc04b] Bock, C.: UML 2 Activity and Action Models Part 5: Partitions. *Journal of Object Technology*, Band 3, Nr. 7, S. 37–56, 2004.

[Bur10] Burke, B.: *RESTful Java with JAX-RS*. O'Reilly Media, 2010.

[CDDM09] Casteleyn, S.; Daniel, F.; Dolog, P.; Matera, M.: *Engineering Web Applications*. Springer, 2009.

[CDIP11] Cicchetti, A.; Di Ruscio, D.; Iovino, L.; Pierantonio, A.: Managing the evolution of data-intensive Web applications by model-driven techniques. *Software and System Modeling (SoSyM)*, Band 12, Nr. 1, S. 53–83, Februar 2011.

[CDMF07] Ceri, S.; Daniel, F.; Matera, M.; Facca, F. M.: Model-Driven Development of Context-Aware Web Applications. *ACM Transactions on Internet Technology*, Band 7, Nr. 1, S. 1–33, 2007.

[CFB+03] Ceri, S.; Fraternali, P.; Bongio, A.; Brambilla, M.; Comai, S.; Matera, M.: *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2003.

[CG04] Cekvenich, V.; Gehner, W.: *Struts - best practices : die Erstellung kommerzieller Web-Anwendungen*. dpunkt.verlag, Heidelberg, 2004.

[Che76] Chen, P. P.-S.: The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, Band 1, Nr. 1, S. 9–36, 1976.

[Con03] Conallen, J.: *Bulding Web Applications with UML - Second Edition*. Addison Wesley, 2.. Auflage, 2003.

[CSS] Cascading Style Sheets Homepage. `http://www.w3.org/Style/CSS/`, (Abruf: Dez 2015).

[CZP01] Christodoulou, S. P.; Zafiris, P. A.; Papatheodorou, T. S.: Web Engineering: the Developers' View and a Practitioner's Approach. In Murugesan, S.; Deshpande, Y. (Hrsg.): *Web Engineering*, Lecture Notes in Computer Science, S. 170–187. Springer Berlin Heidelberg, 2001.

[Dat] jQuery Datatables Homepage. `http://datatables.net/`, (Abruf: Dez 2015).

[DCP08] De Troyer, O.; Casteleyn, S.; Plessers, P.: WSDM : Web Semantics Design Method. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series. Springer, 2008.

[Der] Apache Derby Homepage. `https://db.apache.org/derby`, (Abruf: Dez 2015).

[Dja] Django Homepage. `https://www.djangoproject.com`, (Abruf: Dez 2015).

[DRRS09] Dukaczewski, M.; Reiss, D.; Rumpe, B.; Stein, M.: MontiWeb - Modular Development of Web Information Systems. In Rossi, M.; Sprinkle, J.; Gray, J.; Tolvanen, J.-P. (Hrsg.): *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, 2009.

[dS01] Silva, P. P. d.: User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of the 7th international conference on Design, specification, and verification of interactive systems*, DSV-IS'00, S. 207–226. Springer-Verlag, Berlin, Heidelberg, 2001.

[dS02] Silva, P. P. d.: *Object Modeling of Interactive Systems: The UMLi Approach*. Dissertation, University of Manchester, 2002.

[dSP03] Silva, P. P. d.; Paton, N. W.: User Interface Modeling in UMLi. *IEEE Software*, S. 62–69, 2003.

[DW05] Draheim, D.; Weber, G.: Modelling form-based interfaces with bipartite state machines. *Interacting with Computers*, Band 17, S. 207–228, 2005.

[Ecl] Eclipse Homepage. `http://eclipse.org`, (Abruf: Dez 2015).

[EGHZ12] Eickhoff, C.; Geiger, N.; Hahn, M.; Zündorf, A.: Developing Enterprise Web Applications Using the Story Driven Modeling Approach. In Harth, A.; Koch, N. (Hrsg.): *Current Trends in Web Engineering SE - 21*, Lecture Notes in Computer Science, Band 7059, S. 196–210. Springer, Berlin Heidelberg, 2012.

[Eik13] Eikermann, R.: *Werkzeuggestützte Zuordnung von Benutzerkommentaren bei dynamischen Modellen*. Masterarbeit, RWTH Aachen, 2013.

[EMFa] Eclipse Modefling Framework (EMF) Homepage. `http://www.eclipse.org/modeling/emf/`, (Abruf: Dez 2015).

[EMFb] EMFText Homepage. `http://www.emftext.org`, (Abruf: Dez 2015).

[FBNG07] Freudenstein, P.; Buck, J.; Nussbaumer, M.; Gaedke, M.: Model-driven Construction of Workflow-based Web Applications with Domain-specific Languages. In *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007)*, CEUR Workshop Proceedings, 2007.

[FCBC10] Fraternali, P.; Comai, S.; Bozzon, A.; Carughi, G. T.: Engineering Rich Internet Applications with a Model-Driven Approach. *ACM Transactions on the Web*, Band 4, Nr. 2, S. 1–47, April 2010.

[FGM$^+$99] Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T.: RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, 1999.

[Fie00] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine, 2000.

[Fle] Flexible JDBC Realm Homepage. `http://flexiblejdbcrealm.wamblee.org`, (Abruf: Dez 2015).

[FM07] Ficek, C.; May, F.: *Umsetzung der Java 5 Grammatik für MontiCore*. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[Fow03] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[Fow10] Fowler, M.: *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[FP00] Fraternali, P.; Paolini, P.: Model-Driven Development of Web Applications: The Autoweb System. *ACM Transactions on Information Systems*, Band 28, Nr. 4, S. 323–382, 2000.

[FPP+08] Fons, J.; Pelechano, V.; Pastor, O.; Valderas, P.; Torres, V.: Applying the OOWS Model-Driven Approach for Developing Web Applications. The Internet Movie Database Case Study. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, S. 65–108. Springer, 2008.

[FR07] France, R.; Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07)*, S. 37–54. IEEE, 2007.

[Fra99] Fraternali, P.: Tools and Approaches for Developing Data-Intensive Web Applications: A Survey. *ACM Computing Surveys*, Band 31, Nr. 3, S. 227–263, 1999.

[Fra03] Frankel, D. S.: *Model Driven Architecture - Applying MDA to Enterprise Computing*. Wiley, 2003.

[Fre] Freemarker Homepage. `http://freemarker.org/`, (Abruf: Dez 2015).

[Fre09] Freudenstein, P.: *Web Engineering for Workflow-based Applications - Models, Systems and Methodologies*. Dissertation, Universität Karlsruhe, 2009.

[FRS13] France, R.; Rumpe, B.; Schindler, M.: Why it is so hard to use models in software development: observations. *Software and Systems Modeling*, Band 12, Nr. 4, S. 665–668, Oktober 2013.

[FT02] Fielding, R. T.; Taylor, R. N.: Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, Band 2, Nr. 2, S. 115–150, Mai 2002.

[Gar05] Garret, J. J.: Ajax: A New Approach to Web Applications. `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/`, (Abruf: Dez 2015), 2005.

[Gar11] Garrett, J. J.: *Die Elemente der User Experience*. Addison Wesley Verlag, 2011.

[GCP01] Goméz, J.; Cachero, C.; Pastor, O.: Conceptual Modeling of Device-Independent Web Applications. *IEEE Multimedia*, Band 8, Nr. 2, S. 26–39, 2001.

[GGH+10] Geiger, N.; George, T.; Hahn, M.; Jubeh, R.; Zündorf, A.: Using Actions Charts for Reactive Web Application Modeling. In *ICWE 2010 Workshops*, Lecture Notes in Computer Sciences, S. 49–60, 2010.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHK+15] Greifenberg, T.; Hoelldobler, K.; Kolassa, C.; Look, M.; Mir Seyed Nazari, P.; Mueller, K.; Navarro Perez, A.; Plotnikov, D.; Reiss, D.; Roth, A.; Rumpe, B.; Schindler, M.; Wortmann, A.: A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *MODELSWARD 2015: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, S. 74–85. SciTePress, Angers, France, 2015.

[GHKV08] Groenewegen, D. M.; Hemel, Z.; Kats, L. C. L.; Visser, E.: When Frameworks Let You Down. Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages. In Gray, J.; Sprinkle, J.; Tolvanen, J.-P.; Rossi, M. (Hrsg.): *Proceedings of the 8th OOPSLA Workshop on Domain Specific Modelling (DSM'08)*. Nashville, Tennessee, USA, 2008.

[Gin98] Ginige, A.: Web Engineering: Methodologies for Developing Large and Maintainable Web-Based information Systems. In *Proceedings of the IEEE International Conference on Networking India and the World*, 1998.

[Gla] Glassfish Homepage. `http://glassfish.java.net/`, (Abruf: Dez 2015).

[GM08] Gaedke, M.; Meinecke, J.: The Web as an Application Platform. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering - Modeling and Implementing Web Applications*, Human-Computer Interaction Series, S. 33–45. Springer, 2008.

[GPS91] Garzotto, F.; Paolini, P.; Schwabe, D.: HDM - A Model for the Design of Hypertext Applications. In *Proceedings of the third annual ACM conference on Hypertext*, HYPERTEXT '91, S. 313–328. ACM, New York, NY, USA, 1991.

[GPS93] Garzotto, F.; Paolini, P.; Schwabe, D.: HDM—A Model-Based Approach to Hypertext Application Design. *ACM Transactions on Information Systems*, Band 11, Nr. 1, S. 1–26, 1993.

[GR11] Gessenharter, D.; Rauscher, M.: Code Generation for UML 2 Activity Diagrams. In *Proceedings of the 7th European Conference on Modelling Foundations and Applicationsapplications*, S. 205–220, 2011.

[Gra] Grails Homepage. `http://grails.org`, (Abruf: Dez 2015).

[Gro] Groovy Homepage. `http://groovy.codehaus.org`, (Abruf: Dez 2015).

[GRR10] Grönniger, H.; Reiß, D.; Rumpe, B.: Towards a Semantics of Activity Diagrams with Semantic Variation Points. In Petriu, D.; Rouquette, N.; Haugen, Ø. (Hrsg.): *Model Driven Engineering Languages and Systems, Proceedings of MODELS 2010*, Nr. 6394 der Reihe Lecture Notes in Computer Science, S. 331–345. Springer, 2010.

[Gui] Guice Homepage. `https://code.google.com/p/google-guice/`, (Abruf: Dez 2015).

[GV08] Groenewegen, D.; Visser, E.: Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns. In *2008 Eighth International Conference on Web Engineering*, S. 175–188. IEEE, Juli 2008.

[GV10] Groenewegen, D. M.; Visser, E.: Integration of data validation and user interface concerns in a DSL for web applications. *Software and Systems Modeling*, Band 12, Nr. 1, S. 35–52, September 2010.

[GWT] Google Web Toolkit Homepage. `http://www.gwtproject.org`, (Abruf: Dez 2015).

[HB04] Hall, M.; Brown, L.: *Core Servlets und JavaServer Pages*. Markt-+-Technik-Verl., München, 2004.

[Hec15] Hecht, R.: *Konzeptuelle und Methodische Aufarbeitung von NoSQL-Datenbanksystemen*. Dissertation, Universität Bayreuth, Bayreuth, Dezember 2015.

[Hef10] Heffelfinger, D.: *Java EE 6 with GlassFish 3 Application Server*. Packt Publishing, 2010.

[Her06] Herrmann, C.: *Online Software Transformation Platform*. Diplomarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2006.

[Her14] Herrmann, C.: *Integrierte Software Engineering Services zur effizienten Unterstützung von Entwicklungsprojekten*. Doktorarbeit, RWTH Aachen, 2014.

[Hib] Hibernate Homepage. `http://www.hibernate.org`, (Abruf: Dez 2015).

[HKGV09] Hemel, Z.; Kats, L. C. L.; Groenewegen, D. M.; Visser, E.: Code generation by model transformation: a case study in transformation modularity. *Software and System Modeling*, Band 9, Nr. 3, S. 375–402, November 2009.

[HKRV11] Houben, G.-J.; Koch, N.; Rossi, G.; Vallecillo, A.: Guest Editorial to the Theme Section on Model-Driven Web Engineering. *Software and System Modeling (SoSyM)*, Band 12, Nr. 1, S. 11–13, Februar 2011.

[Hoa73] Hoare, C. A. R.: Hints on programming language design. Technischer Bericht, Stanford University, Stanford, CA, USA, 1973.

[Hol95] Hollingsworth, D.: Workflow Management Coalition - The Workflow Reference Model. `http://www.wfmc.org/standards/docs/tc003v11.pdf`, (Abruf: Dez 2015), 1995.

[HP05] Haustein, S.; Pleumann, J.: A model-driven runtime environment for Web applications. *Software and System Modeling (SoSyM)*, Band 4, Nr. 4, S. 443–458, Juni 2005.

[HTM] HTML Specification. `http://www.w3.org/TR/html5`,(Abruf: Dez 2015).

[HvdSB+08] Houben, G.-J.; Sluijs, K. v. d.; Barna, P.; Broekstra, J.; Casteleyn, S.; Fiala, Z.; Frasincar, F.: HERA. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering - Modeling and Implementing Web Applications*, Human-Computer Interaction Series, S. 263–302. Springer, 2008.

[HVV08] Hemel, Z.; Verhaaf, R.; Visser, E.: WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications. In Czarnecki, K.; Ober, I.; Bruel, J.-M.; Uhl, A.; Völter, M. (Hrsg.): *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, Lecture Notes in Computer Science, Band 5301, S. 113–127. Springer, Heidelberg, September 2008.

[IBV98] Isakowitz, T.; Bieber, M.; Vitali, F.: Web Information Systems. *Communications of the ACM*, Band 41, Nr. 7, S. 78–80, Juli 1998.

[IFM] IFML Homepage. `http://www.ifml.org`, (Abruf: Dez 2015).

[ISB95] Isakowitz, T.; Stohr, E. A.; Balasubramanian, P.: RMM: a methodology for structured hypermedia design. *Communications of the ACM*, Band 38, Nr. 8, S. 34–44, August 1995.

[Isi] Apache Isis Homepage. `http://incubator.apache.org/isis/`, (Abruf: Dez 2015).

[JAGS01] Joshi, J. B. D.; Aref, W. G.; Ghafoor, A.; Spafford, E. H.: Security models for web-based applications. *Communications of the ACM*, Band 44, Nr. 2, S. 38–44, Februar 2001.

[Jak11] Jakob, M.: *Model-based Engineering of Web Applications: The flashWeb Method*. Doktorarbeit, Universität Stuttgart, 2011.

[Jaz07] Jazayeri, M.: Some Trends in Web Application Development. In *FOSE '07: 2007 Future of Software Engineering*, S. 199–213. IEEE Computer Society, Washington, DC, USA, 2007.

[JBP] JBPM Homepage. `http://www.jboss.org/jbpm`, (Abruf: Dez 2015).

[Jer] Jersey Homepage. `https://jersey.java.net`, (Abruf: Dez 2015).

[JPV09] Jeary, S.; Phalp, K.; Vincent, J.: An evaluation of the utility of web development methods. *Software Quality Journal*, Band 17, Nr. 2, S. 125–150, 2009.

[jQua] jQuery Homepage. `http://jquery.com/`, (Abruf: Dez 2015).

[jQub] jQuery Validation Plugin Homepage. `http://bassistance.de/jquery-plugins/jquery-plugin-validation/`, (Abruf: Dez 2015).

[JQuc] jQueryUI Homepage. `http://jqueryui.com/`, (Abruf: Dez 2015).

[JSO] JSON Homepage. `http://json.org`, (Abruf: Dez 2015).

[Jür05] Jürjens, J.: *Secure Systems Development with UML*. Springer, 2005.

[KC13] Kelly Reiner, R.; Cegielski, C. G.: *Introduction to Information Systems*. John Wiley & Sons, 4.. Auflage, 2013.

[KE11] Kemper, A.; Eickler, A.: *Datenbanksysteme - Eine Einführung*. Oldenbourg, 8.. Auflage, 2011.

[Ker12] Kersken, S.: *Apache 2.4 : das umfassende Handbuch*. Galileo-Press, Bonn, 2012.

[KK03] Koch, N.; Kraus, A.: Towards a Common Metamodel for the Development of Web Applications. In Cueva Lovelle, J. M.; Gonzales, B. M.; Rodrigue, L.; Joyanes Aguilar, L.; Labra Gayo, J. E.; Paule Ruiz, M. P. (Hrsg.): *Proceedings of the 3rd International Conference on Web Engineering (ICWE)*, Lecture Notes in Computer Science. Springer, 2003.

[KKCM04] Koch, N.; Kraus, A.; Cacharo, C.; Meliá, S.: Integration of business processes in Web application models. *Journal of Web Engineering*, Band 3, Nr. 1, S. 22–49, 2004.

[KKK09] Kroiß, C.; Koch, N.; Knapp, A.: UWE4JSF: A Model-Driven Generation Approach for Web Applications. In Gaedke, M.; Grossniklaus, M.; Díaz, O. (Hrsg.): *ICWE 2009*, Nr. 5648 der Reihe Lecture Notes in Computer Science. Springer, 2009.

[KKP+09] Karsai, G.; Krahn, H.; Pinkernell, C.; Rumpe, B.; Schindler, M.; Völkel, S.: Design Guidelines for Domain Specific Languages. In Sprinkle, J.; Gray, J.; Rossi, M.; Tolvanen, J.-P. (Hrsg.): *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, 2009.

[KKZB08] Koch, N.; Knapp, A.; Zhang, G.; Baumeister, H.: UML-based Web Engineering: An Approach based on Standards. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering - Modeling and Implementing Web Applications*, Human-Computer Interaction Series. Springer, 2008.

[KKZH04] Knapp, A.; Koch, N.; Zhang, G.; Hassler, H.-M.: Modeling Business Processes in Web Applications with ArgoUWE. In *7th International Conference on the Unified Modeling Language (UML2004)*, LNCS 3273, 2004.

[KLL09] Ko, R. K. L.; Lee, S. S. G.; Lee, E. W.: Business process management (BPM) standards: a survey. *Business Process Management Journal*, Band 15, Nr. 5, S. 744–791, 2009.

[Kni11] Knieke, C.: *Modellierung und Validierung von ausführbaren Anforderungsspezifikationen mit erweiterten UML Aktivitätsdiagrammen.* Dissertation, Technische Universität Braunschweig, 2011.

[Ko09] Ko, R. K. L.: A computer scientist's introductory guide to business process management (BPM). *Crossroads*, Band 15, Nr. 4, S. 11–18, 2009.

[Koc07] Koch, N.: Classification of model transformation techniques used in UML-based Web engineering. *IET Software*, Band 1, Nr. 3, S. 98–111, 2007.

[KPRR06a] Kappel, G.; Pröll, B.; Reich, S.; Retschitzegger, W.: An Introduction to Web Engineering. In Kappel, G.; Pröll, B.; Reich, S.; Retschitzegger, W. (Hrsg.): *Web Engineering*, S. 1–21. John Wiley & Sons, 2006.

[KPRR06b] Kappel, G.; Pröll, B.; Reich, S.; Retschitzegger, W. (Hrsg.): *Web Engineering*. John Wiley & Sons, 2006.

[Kra07] Kraus, A.: *Model Driven Software Engineering for Web Applications*. Dissertation, Ludwig-Maximilians-Universität München, 2007.

[Kra10] Krahn, H.: *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Dissertation, RWTH Aachen University, 2010.

[Kro08] Kroiß, C.: *Modellbasierte Generierung von Web-Anwendungen mit UWE (UML-based Web Engineering)*. Dissertation, Ludwig-Maximilians-Universität, München, 2008.

[Krö11] Kröner, P.: *HTML 5*. Open Source Press, 2.. Auflage, 2011.

[KRV06] Krahn, H.; Rumpe, B.; Völkel, S.: Roles in Software Development using Domain Specific Modelling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006*, S. 150–158. University of Jyväskylä, Finland, 2006.

[KRV07a] Krahn, H.; Rumpe, B.; Völkel, S.: Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling 2007*, 2007.

[KRV07b] Krahn, H.; Rumpe, B.; Völkel, S.: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, S. 286–300, 2007.

[KRV08] Krahn, H.; Rumpe, B.; Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, Lecture Notes in Business Information Processing, Band 11. Springer-Verlag Berlin-Heidelberg, 2008.

[KRV10] Krahn, H.; Rumpe, B.; Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. *STTT*, Band 12, Nr. 5, S. 353–372, 2010.

[KS09] Keith, M.; Schincariol, M.: *Pro JPA 2 : Mastering the Java Persistence API*. Apress, 2009.

[KT08] Kelly, S.; Tolvanen, J.-P.: *Domain Specific Modeling - Enabling Full Code Generation*. John Wiley & Sons, 2008.

[Küh05] Kühne, T.: What is a Model? In Bézivin, J.; Heckel, R. (Hrsg.): *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings, Band 04101. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[KV10] Kats, L. C. L.; Visser, E.: The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In Rinard, M. (Hrsg.): *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, 2010.

[KWB03] Kleppe, A. G.; Warmer, J.; Bast, W.: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[LBD02] Lodderstedt, T.; Basin, D.; Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jézéquel, J.-M.; Hussmann, H.; Cook, S. (Hrsg.): *UML 2002 — The Unified Modeling Language*, Nr. 2460 der Reihe Lecture Notes in Computer Science, S. 426–441. Springer, 2002.

[LF05] Lang, M.; Fitzgerald, B.: Hypermedia Systems Development Practices: A Survey. *IEEE Software*, Band 22, Nr. 2, S. 68–75, März 2005.

[LHK+12] Liskin, O.; Herrmann, C.; Knauss, E.; Kurpick, T.; Rumpe, B.; Schneider, K.: Supporting Acceptance Testing in Distributed Software Projects with Integrated Feedback Systems: Experiences and Requirements. In *Proceedings of 7th International Conference on Global Software Engineering (ICGSE'12)*, S. 84–93. Puerto Alegre, Brazil, 2012.

[Mar83] Martin, J.: *Managing the Data-base Environment*. Prentice Hall, 1983.

[Mau11] Maurice, F.: *CSS3*. Addison Wesley Verlag, 2011.

[McD10] McDaniel, A.: *Perl and Apache : your visual blueprint for developing dynamic web content*. Wiley, Indianapolis, Ind., 2010.

[MCF03] Mellor, S. J.; Clark, A. N.; Futagami, T.: Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, Band 20, S. 14–18, 2003.

[MDHG01] Murugesan, S.; Deshpande, Y.; Hansen, S.; Ginige, A.: Web Engineering: A New Discipline for Development of Web-Based Systems. In *Web Engineering, Software Engineering and Web Application Development*, S. 3–13. Springer-Verlag, London, UK, UK, 2001.

[Mer10] Merkle, B.: Textual modeling tools. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, S. 139–148. ACM Press, New York, New York, USA, Oktober 2010.

[Mer14] Merkelbach, D.: *Entwicklung eines webbasierten Editors für Roboteraufgaben*. Bachelorarbeit, RWTH Aachen, 2014.

[Met] MetaEdit+ Homepage. `http://www.metacase.com/products.html`, (Abruf: Dez 2015).

[Mey97] Meyer, B.: *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[MHB] Modulhandbuch der Technischen Universität Braunschweig. `https://mhb.tu-bs.de`, (Abruf: Dez 2015).

[MHS05] Mernik, M.; Heering, J.; Sloane, A. M.: When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, Band 37, Nr. 4, S. 316–344, 2005.

[MM06] Mendes, E.; Mosley, N. (Hrsg.): *Web Engineering*. Springer, 2006.

[MRV08] Moreno, N.; Romero, J. R.; Vallecillo, A.: An Overview of Model-Driven Web Engineering and the MDA. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering - Modeling and Implementing Web Applications*, S. 353–382. Springer, 2008.

[MSFB05] Muller, P.-A.; Studer, P.; Fondement, F.; Bezivin, J.: Platform independent Web application modeling and development with Netsilon. *Software and System Modeling (SoSyM)*, Band 4, Nr. 4, S. 424–442, Juni 2005.

[Mül10] Müller, B.: *JavaServer Faces 2.0: ein Arbeitsbuch für die Praxis*. Hanser Fachbuchverlag, 2010.

[Mys] MySQL Homepage. `https://www.mysql.com`, (Abruf: Dez 2015).

[NDMN14] Norrie, M. C.; Di Geronimo, L.; Murolo, A.; Nebeling, M.: The Forgotten Many? A Survey of Modern Web Development Practices. In Casteleyn, S.; Rossi, G.; Winckler, M. (Hrsg.): *Proceedings of ICWE 2014*, Nr. 8541 der Reihe Lecture Notes in Computer Science, S. 209–307. Springer International Publishing Switzerland, 2014.

[NFG06a] Nussbaumer, M.; Freudenstein, P.; Gaedke, M.: Towards DSL-based Web Engineering. In *Proceedings of the 15th International Conference on World Wide Web (WWW '06)*, S. 893–894. ACM Press, New York, NY, 2006.

[NFG06b] Nussbaumer, M.; Freudenstein, P.; Gaedke, M.: Web Application Development Employing Domain-Specific Languages. In *Proceedings of the IASTED International Conference on Software Engineering (SE2006)*, S. 13–18. Innsbruck, Austria, 2006.

[Nok] Nokia Case Study. `https://www.metacase.com/papers/MetaEdit_in_Nokia.pdf`, (Abruf: Dez 2015).

[Obj03] Object Management Group: MDA Guide Version 1.0.1 (2003-06-01). `http://www.omg.org/cgi-bin/doc?omg/03-06-01`, (Abruf: Dez 2015), 2003.

[Obj11a] Object Management Group: Business Process Model and Notation (BPMN), Version 2.0. `http://www.omg.org/spec/BPMN/2.0/`, (Abruf: Dez 2015), 2011.

[Obj11b] Object Management Group: Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification, Version 1.1. `http://www.omg.org/spec/QVT/1.1`, (Abruf: Dez 2015), 2011.

[Obj11c] Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1 (11-08-06), August 2011.

[Obj13a] Object Management Group: Action Language for Foundational UML (Alf) - Concrete Syntax for a UML Action Language - Version 1.0.1. `http://www.omg.org/spec/ALF/1.0.1`, 2013.

[Obj13b] Object Management Group: Interaction Flow Modeling Language (IFML) - 1.0 Beta 1. `http://www.omg.org/spec/IFML/1.0`, 2013.

[Obj13c] Object Management Group: OMG Meta Object Facility (MOF) Core Specification, Version 2.1.4. `http://www.omg.org/spec/MOF/2.4.1`, (Abruf: Dez 2015), 2013.

[Oec03] Oechslin, P.: Making a Faster Cryptanalytic Time-Memory Trade-Off. In *The 23rd Annual International Cryptology Conference, CRYPTO '03*, Nr. 2729 der Reihe Lecture Notes on Computer Science, S. 617–630, 2003.

[Oli] OlivaNova Homepage. `http://www.care-t.com`, (Abruf: Dez 2015).

[Ora11] Oracle Corporation: JSR 211: JDBC 4.1 Specification, 2011.

[Ora13a] Oracle Corporation: JSR 338: Java Persistence API, Version 2.1, 2013.

[Ora13b] Oracle Corporation: JSR-339: JAX-RS: Java API for RESTful Web Services, 2013.

[Ora13c] Oracle Corporation: JSR 342: Java Platform Enterprise Edition (Java EE) Specification, v7, 2013.

[Ora13d] Oracle Corporation: JSR 344 - JavaServer Faces Specification, 2013.

[OVD08] Oberortner, E.; Vasko, M.; Dustdar, S.: Towards Modeling Role-Based Pageflow Definitions within Web Applications. In *Proc. of the 4th International Workshop on Model-Driven Web Engineering (MDWE 2008)*, CEUR Workshop Proceedings, Band 389, S. 1–15. CEUR-WS.org, Toulouse, France, September 2008.

[OWA] OWASP - The Open Web Application Security Project: Top 10 Security Risks 2013. `https://owasp.org/index.php/Top_10_2013`, (Abruf: Dez 2015).

[Par07] Parr, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, 1.. Auflage, 2007.

[Paw04] Pawson, R.: *Naked objects*. Dissertation, University of Dublin, Trinity College, 2004.

[PCY+05] Plessers, P.; Casteleyn, S.; Yesilada, Y.; De Troyer, O.; Stevens, R.; Harper, S.; Goble, C.: Accessibility: a web engineering approach. In *Proceedings of the 14th international conference on World Wide Web*, S. 353–362. ACM, 2005.

[PFPA06] Pastor, O.; Fons, J.; Pelechano, V.; Abrahão, S.: Conceptual Modelling of Web Applications: The OOWS Approach. In Mendes, E.; Mosley, N. (Hrsg.): *Web Engineering*, S. 277–302. Springer, 2006.

[PGIP01] Pastor, O.; Goméz, J.; Isfrán, E.; Pelechano, V.: The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Information Systems*, Band 26, S. 507–534, 2001.

[PHP] PHP Homepage. `http://php.net`, (Abruf: Dez 2015).

[PLMC+08] Preciado, J. C.; Linaje, M.; Morales-Chaparro, R.; Sanchez-Figueroa, F.; Zhang, G.; Kroiß, C.; Koch, N.: Designing Rich Internet Applications Combining UWE and RUX-Method. In *2008 Eighth International Conference on Web Engineering*, S. 148–154. IEEE, Juli 2008.

[PM01] Pawson, R.; Matthews, R.: Naked Objects: a technique for designing more expressive systems. *SIGPLAN Notices*, Band 36, Nr. 12, S. 61–67, 2001.

[PM02] Pawson, R.; Matthews, R.: *Naked Objects*. John Wiley & Sons, 2002.

[Pok13] Pokorny, J.: NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, Band 9, Nr. 1, S. 69–82, 2013.

[Pow10] Powers, D.: *PHP solutions : dynamic web design made easy*. Apress, New York, NY, 2010.

[PP08] Pfeiffer, M.; Pichler, J.: A Comparison of Tool Support for Textual Domain-Specific Languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, S. 1–7, 2008.

[PYP] PYPL PopularitY of Programming Language. `https://pypl.github.io/PYPL.html`, (Abruf: Dez 2015).

[RC04] Robinson, D.; Coar, K.: RFC 3875: The Common Gateway Interface (CGI) Version 1.1, 2004.

[Ree79] Reenskaug, T.: Models - Views - Controllers. Technical Report, Technical Note, Xerox Parc, 1979.

[RoR] Ruby on Rails Homepage. `http://rubyonrails.org`, (Abruf: Dez 2015).

[RQZ07] Rupp, C.; Quens, S.; Zengler, B.: *UML 2 glasklar*. Hanser Fachbuchverlag, 2007.

[RR13] Reiß, D.; Rumpe, B.: Using Lightweight Activity Diagrams for Modeling and Generation of Web Information Systems. In Mayr, H.; Kop, C.; Liddle, S.; Ginige, A. (Hrsg.): *Proceedings of the 4th International United Information Systems Conference, UNISCON 2012*, Lecture Notes in Business Information Processing, S. 61–73. Springer, Yalta, Ukraine, 2013.

[RRSQS08] Reiss, D.; Rumpe, B.; Schulze-Quester, M.; Stein, M.: Evolving and Implanting Web-Based E-Government-Systems in Universities. In *UNISCON*, S. 282–295, 2008.

[RS06] Rossi, G.; Schwabe, D.: Model-Based Web Application Development. In Mendes, E.; Mosley, N. (Hrsg.): *Web Engineering*, S. 303–333. Springer, 2006.

[RS08] Rossi, G.; Schwabe, D.: Modeling and Implementing Web Applications with OOHDM. In Rossi, G.; Pastor, O.; Schwabe, D.; Olsina, L. (Hrsg.): *Web Engineering - Modeling and Implementing Web Applications*, Human-Computer Interaction Series, S. 109–156. Springer, 2008.

[Rum11] Rumpe, B.: *Modellierung mit UML*. Springer, Berlin, 2.. Auflage, 2011.

[Rum12] Rumpe, B.: *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, Berlin, 2.. Auflage, 2012.

[RvdATW06] Russell, N.; Aalst, W. M. P. v. d.; Ter Hofstede, A. H. M.; Wohed, P.: On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In Stumptner, M.; Hartmann, S.; Kiyoki, Y. (Hrsg.): *Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling*, S. 95–104. Australian Computer Society, Inc., 2006.

[SCF+96] Sandhu, R. S. R. S.; Coyne, E. J.; Feinstein, H. L.; Youman, C. E.; Coynek, E. J.; Feinsteink, H. L.; Youmank, C. E.: Role-Based Access Control Models. *IEEE Computer*, Band 29, Nr. 2, S. 38–47, 1996.

[Sch06] Schmidt, D.: Model-Driven Engineering. *IEEE Computer*, Band 39, Nr. 2, S. 25–31, Februar 2006.

[Sch12] Schindler, M.: *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, 2012.

[SCR01] Shachor, G.; Chace, A.; Rydin, M.: *JSP Tag Libraries*. Manning, 2001.

[SdV01] Samarati, P.; Vimercati, S. C. d.: Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, S. 137–196. Springer, 2001.

[Sel03] Selic, B.: The Pragmatics of Model-Driven Development. *IEEE Software*, Band 20, Nr. 5, S. 19–25, 2003.

[SH06] Shan, T. C.; Hua, W. W.: Taxonomy of Java Web Application Frameworks. In *IEEE International Conference on E-Business Engineering*, S. 378–385, 2006.

[Shi] Apache Shiro Homepage. `https://shiro.apache.org`, (Abruf: Dez 2015).

[SK06] Schwinger, W.; Koch, N.: Modeling Web Applications. In Kappel, G.; Pröll, B.; Reich, S.; Retschitzegger, W. (Hrsg.): *Web Engineering*. John Wiley & Sons, 2006.

[SKRG07] Sarstedt, S.; Kohlmeyer, J.; Raschke, A.; Gessenharter, D.: ActiveCharts: Eine pragmatische Umsetzung der MDA mit UML2-Aktivitätsdiagrammen. *OBJEKTspektrum*, Band 6, 2007.

[Spo] The Spoofax Language Workbench Homepage. `http://strategoxt.org/Spoofax`, (Abruf: Dez 2015).

[Spr] Spring Homepage. `http://www.springsource.org`, (Abruf: Dez 2015).

[SRS+08] Schwinger, W.; Retschitzegger, W.; Schauerhuber, A.; Kappel, G.; Wimmer, M.; Pröll, B.; Cachero Castro, C.; Casteleyn, S. et al.: A survey on web modeling approaches for ubiquitous web applications. *International Journal of Web Information Systems*, Band 4, Nr. 3, S. 234–305, 2008.

[SS94] Sandhu, R.; Samarati, P.: Access control: principle and practice. *IEEE Communications Magazine*, Band 32, Nr. 9, S. 40–48, September 1994.

[Sta73] Stachowiak, H.: *Allgemeine Modelltheorie*. Springer, 1973.

[Ste12] Steens, L.: *Zuordnung von Benutzerkommentaren aus dem generierten System zu den Quellmodellen*. Masterarbeit, RWTH Aachen, 2012.

[Ste14] Steyer, R.: *jQuery : das universelle JavaScript-Framework für das interaktive Web und mobile Anwendungen*. Hanser Fachbuchverlag, München, 2014.

[Stö04] Störrle, H.: Towards a Petri-net Semantics of Data Flow in UML 2.0 Activities. Technischer Bericht Nr. TR 0504, University of Munich, 2004.

[Str] Apache Struts Homepage. `http://struts.apache.org/`, (Abruf: Dez 2015).

[Stu] StuBS Homepage. `http://stubs.tu-bs.de`, (Abruf: Dez 2015).

[Sun02] Sun Microsystems: JSR 52: A Standard Tag Library for JavaServer Pages, 2002.

[Sun06] Sun Microsystems: Java Architecture for XML Binding (JAXB) 2.0, 2006.

[Sun09a] Sun Microsystems: JSR 316: Java Platform Enterprise Edition (Java EE) Specification, v6, 2009.

[Sun09b] Sun Microsystems: JSR 317: Java Persistence API, Version 2.0, 2009.

[Sun10] Sun Microsystems: JSR 315: JavaTM Servlet 3.0 Specification, 2010.

[SVEH07] Stahl, T.; Völter, M.; Efftinge, S.; Haase, A.: *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. dpunkt.verlag, Heidelberg, 2.. Auflage, 2007.

[Tap] Apache Tapestry Homepage. `http://tapestry.apache.org/`, (Abruf: Dez 2015).

[TGP12] Torres, V.; Giner, P.; Pelechano, V.: Developing BP-driven web applications through the use of MDE techniques. *Software and System Modeling (SoSyM)*, Band 11, Nr. 4, S. 609–631, Oktober 2012.

[THET13] Tacy, A.; Hanson, R.; Essington, J.; Tökke, A.: *GWT in Action*. Manning, 2.. Auflage, 2013.

[TK09] Tolvanen, J.-P.; Kelly, S.: MetaEdit+: defining and using integrated domain-specific modeling languages. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, S. 819. ACM Press, New York, New York, USA, Oktober 2009.

[Tom] Apache Tomcat Homepage. `https://tomcat.apache.org/`,(Abruf: Dez 2015).

[TP06] Torres, V.; Pelechano, V.: Building Business Process Driven Web Applications. In Dust-dar, S.; Fiadeiro, J. L.; Sheth, A. (Hrsg.): *Business Process Management, 4th International Conference, BPM 2006*, Lecture Notes in Computer Science, Band 4102, S. 322–337. Springer, Berlin/Heidelberg, 2006.

[TPRV05] Torres, V.; Pelechano, V.; Ruiz, M.; Valderas, P.: A Model Driven Approach for the Integration of External Functionality in Web Applications. The Travel Agency System. In *Proceedings of 1st Workshop on Model-Driven web Engineering, in conjunction with ICWE 2005*. Sydney, Australia, 2005.

[TYF86] Teorey, T. J.; Yang, D.; Fry, J. P.: A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys*, Band 18, Nr. 2, S. 197–222, 1986.

[URGS07] Urbieta, M.; Rossi, G.; Ginzburg, J.; Schwabe, D.: Designing the Interface of Rich Internet Applications. In *Proceedings of the 2007 Latin American Web Conference*, S. 144–153. IEEE Computer Society, Oktober 2007.

[vdATKB03] Aalst, W. M. P. v. d.; Ter Hofstede, A. H. M.; Kiepuszewski, B.; Barros, A. P.: Workflow Patterns. *Distrib. Parallel Databases*, Band 14, Nr. 1, S. 5–51, 2003.

[vDKV00] Deursen, A. v.; Klint, P.; Visser, J.: Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, Band 35, Nr. 6, S. 26–36, Juni 2000.

[Vel] Velocity Homepage. `http://velocity.apache.org/`, (Abruf: Dez 2015).

[VG11] Vukotic, A.; Goodwill, J.: *Apache Tomcat 7*. Apress, 2011.

[Vis08a] Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. Technischer Bericht Nr. TUD-SERG-2008-023, Delft University of Technology, Software Engineering Research Group, 2008.

[Vis08b] Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In Lämmel, R.; Visser, J.; Saraiva, J. (Hrsg.): *Generative and Transformational Techniques in Software Engineering II*, S. 291–373. Springer Verlag, Berlin, Heidelberg, 2008.

[VK05] Vitolins, V.; Kalnins, A.: Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, S. 181–194. IEEE Computer Society, Washington, DC, USA, 2005.

[Voe13] Voelter, M.: *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. Online-Publikation (`http://dslbook.org`), 2013.

[Völ11] Völkel, S.: *Kompositionale Entwicklung domänenspezifischer Sprachen*. Dissertation, TU Braunschweig, 2011.

[WD08] Wright, J.; Dietrich, J.: Survey of Existing Languages to Model Interactive Web Applications. In *APCCM '08: Proceedings of the fifth on Asia-Pacific conference on conceptual modelling*, S. 113–123. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2008.

[Weba] November 2013 Web Server Survey. `http://news.netcraft.com/archives/category/web-server-survey/`, (Abruf: Dez 2015).

[Webb] Spring Web Flow Homepage. `http://www.springsource.org/spring-web-flow`, (Abruf: Dez 2015).

[Webc] WebDSL Homepage. `http://www.webdsl.org`, (Abruf: Dez 2015).

[Webd] WebRatio Homepage. `http://webratio.com`, (Abruf: Dez 2015).

[Wen10] Wenger, R.: *C# und .NET-Grundlagen*. Microsoft Press, Unterschleißheim, 2010.

[Wen11] Wenger, R.: *Benutzeroberflächen mit WPF, Windows Forms, ASP.NET und Silverlight*. Microsoft Press, Unterschleißheim, 2011.

[WG08] Wilde, E.; Gaedke, M.: Web Engineering Revisited. In *The British Computer Society (BCS) Conference on Visions of Computer Science*, S. 41–49, 2008.

[Wik] Wikipedia: Wikipedia Dokumentation - Einfache Listen. `https://de.wikipedia.org/wiki/Hilfe:Listen#Einfache_Listen`, (Abruf: Dez 2015).

[Win13] Wintermeyer, S.: *Ruby on Rails 3.2 : für Ein-, Um- und Quereinsteiger*. Addison-Wesley, München [u.a.], 2013.

[Wir74] Wirth, N.: On the Design of Programming Languages. In *IFIP Congress*, S. 386–393, 1974.

[XFo] XForms 1.0 Recommendation. `http://www.w3.org/TR/2007/REC-xforms-20071029/`, (Abruf: Dez 2015).

[Xte] Xtext Homepage. `http://www.eclipse.org/Xtext`, (Abruf: Dez 2015).

[ZW06] Zeppenfeld, K.; Wolters, R.: *Generative Software-Entwicklung mit der MDA*. Elsevier, Spektrum Akad. Verl., München, 2006.

Abbildungsverzeichnis

2.1 Generierte Applikation aus dem durchgängigen Beispiel 17

3.1 Schnittstelle zur Symboltabelle (in Anlehnung an [Sch12]) 24
3.2 Beispiel für Typadaptierung 25
3.3 Abhängigkeiten zwischen den verschiedenen Sprachen 29

4.1 Komponenten von MontiWIS und Rollen bei der Entwicklung 40
4.2 Zweistufiger Generierungsprozess in MontiWIS 41

5.1 Komponenten des generierten Systems 50

6.1 Vereinfachte Darstellung der Klassendiagramm-Grammatik aus [Sch12] . . . 62
6.2 Grundaufbau eines UML/P Klassendiagramms 64
6.3 Beispiel für ein Klassendiagramm mit Vererbung 64
6.4 Klassendiagramm aus Abbildung 6.3 in textueller Form 65
6.5 Beispiel für ein Klassendiagramm mit Assoziation und Komposition 66
6.6 Klassendiagramm aus Abbildung 6.5 in textueller Form 67
6.9 Spezifikation von vordefinierten Klassen über den Stereotyp `implicit` 73
6.10 Grundaufbau einer Klasse für Nutzerkonten 74
6.11 Struktur der Klasse `GroupInternal` zur Gruppendefinition 76
6.12 Assoziation zwischen Nutzerklasse und -gruppe 77
6.13 Zwischenstruktur für Klassendiagramme 79
6.14 Zentraler Speicher für Datenstrukturinformationen 82
6.17 Schematischer Aufbau einer generierten JPA-Klasse 95
6.18 Remoteschnittstelle einer generierten DAO-Klasse 98
6.19 Schematischer Aufbau einer generierten DAO-Klasse 99
6.20 Aufruf einer DAO-Klasse 101
6.21 Beispiel für Standardseiten 102
6.22 Navigationsmöglichkeiten zwischen den Seiten 103

7.1 Workflow als Beispiel für Seitenbeschreibung 113
7.2 Vereinfachte Darstellungen der Seitenbeschreibungs-Grammatik 115
7.3 Beispielhafter Aufbau einer Page-Sprachdatei 116
7.4 Generierte Seite der Sprachdatei aus Abbildung 7.3 117
7.5 Genereller Aufbau einer Seitenbeschreibung 118
7.6 Verwendung einer Überschrift 119

7.7 Verwendung eines Textabsatzes 120
7.8 Verwendung einer Tabelle 121
7.10 Verwendung des Elements zur Darstellung aller Attribute 123
7.11 Verwendung der Listendarstellung zur Auswahl eines Objekts 124
7.12 Verwendung der Listendarstellung zur Auswahl mehrerer Objekte 125
7.13 Verwendung der Listendarstellung zur Ansicht mehrerer Objekte 126
7.14 Verwendung der Einbindung von beliebigem HTML-Code 127
7.15 Generierte Darstellung für Überschriften (`head`) 128
7.16 Generierte Darstellung für Freitext (`text`) 128
7.17 Generierte Darstellung für Tabellenkopfzeilen 129
7.18 Generierte Eingabe- und Ausgabedarstellung für `String` 129
7.19 Generierte Eingabe- und Ausgabedarstellung für `Text` 129
7.20 Generierte Eingabe- und Ausgabedarstellung für `Date` 130
7.21 Generierte Eingabe- und Ausgabedarstellung für `Integer` 130
7.22 Generierte Eingabe- und Ausgabedarstellung für `Float` 131
7.23 Generierte Eingabe- und Ausgabedarstellung für `Boolean` 131
7.24 Generierte Eingabe- und Ausgabedarstellung für `Email` 132
7.25 Generierte Eingabe- und Ausgabedarstellung für `Password` 132
7.26 Generierte Eingabe- und Ausgabedarstellung von Assoziationen 133
7.27 Generierte Eingabe- und Ausgabedarstellung Kompositionen 134
7.28 Generierte Listendarstellung für die Auswahl eines Objekts 135
7.29 Generierte Listendarstellung für die Auswahl mehrerer Objekte 135
7.30 Generierte Listendarstellung im Anzeigemodus 136
7.31 Generierte Listendarstellung im Editiermodus 136
7.32 Übersicht über Symboltabelleneinträge für Seiten 138
7.33 Übersicht über die Zwischenstruktur für die Seitenbeschreibungssprache . . 140
7.34 Zentraler Speicher für Seiteninformationen 143
7.35 Übersicht über die an der Rekonstruktion von Seiten beteiligten Klassen . . . 145
7.37 Zusammenhang zwischen Templates, Seiten- und ihren Anteilen 152
7.38 Grundaufbau der generierten JSP-Seiten 154
7.39 Grundaufbau von Eingabefeldern 158
7.40 Umsetzung von Assoziationen 161
7.41 Umsetzung von Kompositionen 162
7.42 Umsetzung von `PageEntity`-Klassen 164
7.43 Umsetzung von `PageEntityInitializer`-Klassen 165
7.44 Umsetzung von Ressourcenklassen für Seiten 167
7.45 Beispielhafter Aufbau eines Freemarker-Templates 171
7.46 Beispiel einer deklarativen Seitenbeschreibung in UWE (vgl. [KKZB08]) . . . 172

8.1 Vereinfachte Darstellung der Aktivitätsdiagramm-Grammatik 179
8.2 Vereinfachte Darstellung des `java` Aktionsinhalts 180
8.3 Vereinfachte Darstellung des `view` Aktionsinhalts 180
8.4 Vereinfachte Darstellung des `cmd` Aktionsinhalts 181

8.5 Grafisches Beispiel einer Aktivität 182
8.6 Textuelle Repräsentation des Beispiels aus Abbildung 8.5 183
8.7 Syntax einer Aktivität 184
8.8 Syntax von Aktivitätsparametern 185
8.9 Syntax von Aktionen 186
8.10 Syntax von Aktivitätsbereichen 187
8.11 Syntax von Kanten 187
8.12 Syntax von Verzweigungsknoten 188
8.13 Syntax von Verzweigungsknoten mit Bedingungen 189
8.14 Syntax von Verbindungsknoten 190
8.15 Syntax von Parallelisierungsknoten 191
8.16 Syntax von Synchronisationsknoten 193
8.17 Syntax von Startknoten 194
8.18 Syntax von Endknoten 195
8.19 Syntax von Ablaufendknoten 195
8.20 Parameter von Aktionen 196
8.21 Variablen in Aktionen 197
8.22 Seitenaufruf in Aktionen 198
8.23 Einbindung von Java in Aktionen 199
8.24 Kommando zum Laden von allen Objekten eines Typs 201
8.25 Kommando zum Laden eines Objekts per Id 202
8.26 Kommando zum Suchen und Laden von Objekten 203
8.27 Kommando zum Speichern eines Objekts 203
8.28 Kommando zum Speichern einer Menge von Objekten 204
8.29 Kommando zum Löschen eines Objekts 204
8.30 Kommando zum Abfragen des aktuellen Nutzers 205
8.31 Kommando zum Zuweisen von Nutzern zu Rollen 206
8.32 Kommando zum Darstellen von Fehlermeldungen 207
8.33 Aufruf einer Aktivität (rechts) aus einer Aktion heraus (links) 208
8.34 Symboltabelleneinträge für Aktivitätsdiagramme 210
8.35 Zwischenstruktur für Aktivitätsdiagramme 214
8.36 Zentraler Speicher für Aktivitätsinformationen 216
8.39 Komponentenstruktur der Aktivitätsausführungsschicht 225
8.40 Klassen zur Repräsentation von Aktivitäten in der Laufzeitumgebung 227
8.41 Setzen und Abrufen von Parametern in Knoten 228
8.42 Klassen zur Repräsentation von Ausführungskontext 230
8.43 Überblick über Klassen von Aktivitätsstruktur, ihrem Kontext und ihren Executor-Klassen 233
8.44 Ressourcenklasse zum Starten und Fortführen von Aktivitäten 235
8.45 Ablauf bei Aufruf und Absenden von Aktionen 237
8.46 Ressourcenklasse zum Aufruf und Absenden von Aktionen 239

9.1 Vereinfachte Darstellung der Grammatik der Applikationssprache 248

9.2 Beispiel für die Applikationssprache 250
9.4 Generiertes Menü aus dem Beispiel in Abbildung 9.2 als `guest` Nutzer 252
9.6 Klassen der Symboltabellenstruktur für die Applikationssprache 254
9.7 Klassen der Zwischenstruktur für die Applikationssprache 256
9.8 Zentraler Speicher für die Applikationssprache 257
9.10 Umsetzung des Menüs 263

10.5 Beispielhafter Inhalt der `langstring.properties` Datei 282

11.1 Registrierungsmaske des Bibliothekssystems 286
11.2 Bearbeitungsmaske für Module im Modulhandbuchsystem 290
11.3 Editormaske für eine Institution im Modulhandbuchsystem 291
11.6 Vergleich Modell- und generierter Code 295
11.7 Editoransicht für Aktivitäten in Eclipse 298

C.1 Grammatik der MontiCore Literals Version 1.3 [Sch12] 345
C.2 Grammatik der MontiCore Types Version 1.3 [Sch12] 352
C.3 Grammatik der MontiCore UML/P Common Version 1.3 [Sch12] 356
C.4 Grammatik der Java/P Version 1.1.5 [FM07] 379
C.5 Grammatik der UML/P Klassendiagramme Version 1.6 [Sch12] 386
C.6 Grammatik der Seitenbeschreibungssprache Version 1.0 391
C.7 Grammatik der Aktivitätsdiagramme Version 1.0 398
C.8 Grammatik des Aktionsinhalts für Seitenaufrufe Version 1.0 399
C.9 Grammatik des Aktionsinhalts für Kommandos Version 1.0 401
C.10 Grammatik des Aktionsinhalts für Java Code Version 1.0 402
C.11 Grammatik der Applikationssprache Version 1.0 405

Tabellenverzeichnis

4.3 Sprachpakete im `MontiWIS`-Projekt und zugehörige Modelle 43

5.2 HTTP-Methoden und Bedingungen hinsichtlich Seiteneffekten 53

6.7 Vordefinierte Datentypen in MontiWIS 70
6.8 Stereotypen in MontiWIS 73
6.15 Erzeugung von Seitennamen für Standardseiten 88
6.16 Die Template-Dateien für Klassendiagramme und ihre Bedeutung 91
6.23 Zugriff auf Objekte über REST-Schnittstelle 104

7.9 Möglicher Inhalt von Tabellenzellen 122
7.36 Die Template-Dateien für Seiten und ihre Verwendung 147

8.37 Umformungen bei der Bearbeitung von Kantendeklarationen 219
8.38 Die Template-Dateien für Aktivitäten und ihre Bedeutung beim Generierungsprozess 222

9.3 Mögliche statische Verweise in der Menüdefinition 252
9.5 Bedeutung von Berechtigungen auf Klassen 253
9.9 Die Template-Dateien der Applikationssprache und ihre Bedeutung 259
9.11 Konfigurationsparameter des Flexible JDBC Realms 265

10.1 Verzeichnisse in MontiWIS-Anwendungsprojekten 272
10.2 Konfigurationsdateien 273
10.3 Einträge der Datei `model/application.properties` 274
10.4 Einträge der Datei `build.properties` 276

11.4 Lines of Code für MontiWIS-Projekte 294
11.5 Lines of Code für MontiWIS Anwendungen 294

Anhang A.

Glossar

Aktivität

Nach [Obj11c] ist eine Aktivität *the specification of parameterized behavior as the coordinated sequencing of subordinate units whose individual elements are actions.*. Eine Aktivität besteht ihrerseits aus Aktionen, deren Verhalten wiederum in einer Aktivität spezifiziert sein kann. In MontiWIS handelt es sich bei einer Aktivität um den Inhalt eines → Aktivitätsdiagramms. Hierbei wird eine Aktivität pro Datei gespeichert.

Aktivitätsdiagramm

Ein Aktivitätsdiagramm allgemein stellt eine oder mehrere → Aktivitäten dar. In MontiWIS bezeichnet ein Aktivitätsdiagramm eine Datei, in der exakt eine Aktivität spezifiziert ist.

Domänendatenmodell

Unter dem Domänendatenmodell wird dasjenige verstanden, das als Grundlage für die zu generierende Applikation dient. Es wird in einem → Klassendiagramm codiert .

Generator

Ein Generator in MontiWIS operiert auf der Zwischenstruktur und überführt diese Objektstruktur durch die Vearbeitung durch Templates in wie auch immer gearteten Code. Dies können beispielsweise Java-Code oder auch JSP-Seiten zur Darstellung sein.

Hypermediaapplikation

Unter einer Hypermediaapplikation wird eine Anwendung verstanden, bei der primär von Seite zu Seite bzw. von Datenobjekt zu Datenobjekt navigiert wird. Seiten bzw. Datenobjekte lassen sich dabei speichern. Im Gegensatz zu → Web-Informationssystemen kommt der Ausführung von Geschäftslogik dabei eine sekundäre Bedeutung zu.

Informationssystem

Nach [KC13] gilt: *An information system collects, processes, stores, analyzes and disseminates information for a specific purpose.* Damit dient es primär zur Verarbeitung von Informationen, die eingegeben und durch verschiedenartige Arten von Logik verarbeitet werden.

Klassendiagramm

Wenn in dieser Arbeit von einem Klassendiagramm die Rede ist, handelt es sich in aller Regel um ein UML/P Klassendiagramm, welches in textueller Syntax codiert ist. Dementsprechend orientiert sich auch die Darstellung von Attributen an der Java-Schreibweise und nicht an der in der UML empfohlenen Schreibweise.

MontiWIS-Anwendungsprojekt

Ein MontiWIS-Anwendungsprojekt bezeichnet ein (Eclipse-)Projekt, bei dem der MontiWIS-Generator zur Generierung eines Web-Informationssystem zum Einsatz kommt. Es muss einem bestimmten Aufbau folgen und bestimmte Dateien enthalten, damit der Generator erfolgreich die Modelle verarbeiten kann.

PageEntity

Unter einer `PageEntity`-Klasse wird diejenige Klasse verstanden, die zu jeder spezifizierten Seite generiert wird. In ihr sind werden die Datenobjekte, die auf der Seite dargestellt oder verändert werden, bereitgestellt und über sogenannte `PageEntityInitializer`-Klassen mit Objekten belegt.

Rekonstruktor

Ein Rekonstruktor in MontiWIS operiert auf der Zwischenstruktur, die durch den → Transformator aufgebaut wurde und fügt beispielsweise Standardverhalten den dortigen Klassen hinzu oder baut die Objekte derartig um, dass Kurzschreibweisen in entsprechende verbose Objektstrukturen überführt werden.

Ressource

Ressourcen sind als Artefakte zu verstehen, die im Sinn von REST adressierbar und manipulierbar sind. In MontiWIS sind dies Seiten, Objekte der Domänendatenstruktur aber auch Aktivitäten und Aktionen als Konstrukte der Ablauflogik des Web-Informationssystems.

Ressourcenklasse

Die Aufgabe einer Ressourcenklasse in MontiWIS ist die Behandlung von HTTP-Anfragen. Über sie werden → Ressourcen adressierbar. Sie reagieren auf (im Kontext der Ressource sinnvolle) HTTP-Anfragen, führen zur Ausführung der korrekten Logik und liefern das spezifizierte Ergebnis per HTTP zurück.

Seitenparameter

Unter einem Seitenparameter werden die Parameter verstanden, die einer Seite übergeben werden und auf die darin zugegriffen werden kann. Der Datentyp kann entweder Teil des → Domänendatenmodells sein oder aber auch ein → vordefinierter Datentyp .

Sprachpaket

Ein Sprachpaket ist ein Paket im Sinn eines Java-Pakets, das sämtliche zur Sprachverarbeitung notwendigen Klassen enthält. Es exisitert im MontiWIS-Projekt für jede Sprache und enthält beispielsweise PrettyPrinter, Symboltabelleninfrastruktur, → Transformatorklassen und Klassen für die Codegenerierung.

Standardseite

Unter einer Standardseite wird eine Webseite verstanden, die das → Standardverhalten implementiert..

Standardverhalten

In MontiWIS wird für jede Klasse eines Klassendiagramms sogenanntes Standardverhalten erzeugt. Damit ist Funktionalität in Form von → Standardseiten gemeint, die es ermöglicht, Objekte der jeweiligen Klasse anzulegen, zu bearbeiten, zu löschen und zu verwalten. Im Allgemeinen wird derartiges Verhalten unter CRUD zusammengefasst..

Transformator

Eine Transformatorklasse verarbeitet einen AST und erstellt aus der darüber repräsentierten Modellstruktur eine → Zwischenstruktur, die potenziell über einen → Rekonstruktor weiter bearbeitet und letztlich durch einen → Generator in Code wie beispielsweise Java-Code oder auch JSP-Seiten überführt wird.

Vordefinierter Datentyp

Ein vordefinierter Datentyp ist beispielsweise ein `String`, eine `Email` oder ein `Date`. Diese Datentypen werden in einem → Klassendiagramm als Attributtyp verwendet und im generierten System entsprechend ihres Typs behandelt.

Web-Engineering

Laut [MDHG01] wird Web Engineering als *use of sound scientific, engineering and management principles, and disciplined and systematic approaches to development, deployment and maintenance of Web-based systems* beschrieben, also den systematischen Entwicklungsmethodiken von → Webanwendungen.

Web-Informationssystem

Als Web-Informationssystem wird ein → Informationssystem verstanden, das über einen Webbrowser bedient wird.

Webanwendung

Eine Webanwendung ist der Oberbegriff für jedwede Art von Anwendung, auf die über einen Webbrowser zugegriffen wird. Dies umschließt neben → Web-Informationssystemen auch Anwendungen mit rein statischen Inhalten.

Workflow

Ein Workflow ist nach der Definition der Workflow Management Coalition "'*Workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal.*"' [Hol95].

Zwischenstruktur

Die Zwischenstruktur in MontiWIS ist die Klassenstruktur, die zum einfacheren Zugriff durch → Rekonstruktor und → Generator erstellt und von diesen verarbeitet wird. Darin sind sämtliche Informationen aus den Modellen gespeichert, die für die letztliche Generierung der Applikation notwendig sind.

Anhang B.

Abkürzungen

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

AST Abstract Syntax Tree

ATL Atlas Transformation Language

BPM Business Process Management

BPMN Business Process Modeling Notation

CGI Common Gateway Interface

CIM Computation Independent Model

CRUD Create / Read / Update / Delete

CSS Cascading Style Sheets

DAO Data Access Object

DSL Domain Specific Language

DSM Domain Specific Modeling

EBNF Extended Backus Naur Form

EJB Enterprise Java Bean

EL Expression Language

EMF Eclipse Modeling Framework

ER Entity Relationship

GWT Google Web Toolkit

HDM Hypermedia Design Method

HTTP Hypertext Transfer Protocol

HTML Hypertext Markup Language

HTML5 Hypertext Markup Language 5

IFML Interaction Flow Modeling Language

IoC Inversion of Control

Java EE Java Platform Enterprise Edition

JAXB Java Architecture for XML Binding

JDK Java Development Kit

JDBC Java Database Connectivity

JPA Java Persistence API

JSF JavaServer Faces

JSON JavaScript Object Notation

JSP JavaServer Pages

JSTL JavaServer Pages Standard Tag Library

JTA Java Transaction API

JVM Java Virtual Machine

LOC Lines of Code

MD5 Message-Digest Algorithm 5

MDA Model-driven Architecture

MOF Meta-Object Facility

MVC Model View Controller

oAW openArchitectureWare

OCL Object Constraint Language

OMG Object Management Group

OO-H Object-Oriented-Hypermedia

OOHDM Object-Oriented Hypermedia Design Method

OOWS Object-Oriented Web Solution

OSTP Online Software Transformation Platform

OWL Web Ontology Language

NoSQL Not only SQL

PHP PHP: Hypertext Processor bzw. Personal Home Page Tools (veraltet)

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query View Transformation

RBAC Role Based Access Control

RDF Resource Description Framework

REST Representational state transfer

RIA Rich Internet Application

RMM Relationship Management Methodology

RMI Remote Method Invocation

SOAP Simple Object Access Protocol

SQL Structured Query Language

UML Unified Modeling Language

UML/P Unified Modeling Language / Programmier-geeignet

URL Uniform Resource Locator

UUID Universally Unique Identifier

UWE UML-based Web Engineering

WebML Web Modeling Language

WIS Web-Informationssystem

WSDM Web Semantics Design Method

XMI XML Metadata Interchange

XML Extensible Markup Language

XSS Cross Site Scripting

Anhang C.

Grammatiken

C.1. MontiCore Literals

MC

```
package mc.literals;

version "1.3";

/**
 * This grammar defines Java compliant literals. The scope of this grammar is to
 * ease the reuse of literals structures in Java-like sublanguages, e.g., by
 * grammar inheritance or grammar embedment.
 * The grammar contains literals from Java, e.g., Boolean, Char, String, ....
 *
 * @author Martin Schindler
 */
grammar Literals {

    /*=================================================================*/
    /*============================== OPTIONS ==========================*/
    /*=================================================================*/

    options {
        parser lookahead=3
        lexer lookahead=3
        nostring noident nows
    }

    /*=================================================================*/
    /*====================== INTERFACE DEFINITIONS ====================*/
    /*=================================================================*/

    ast BooleanLiteral =
        method public boolean getValue(){
          return this.source == ASTConstantsLiterals.TRUE;
        };

    ast CharLiteral =
        method public char getValue() {
          try {
            return
                mc.literals.LiteralsHelper.getInstance().decodeChar(getSource());
          }
          catch (java.io.CharConversionException e) {
            return ' ';
          }
```

```
43          };
44
45      ast StringLiteral =
46          method public String getValue() {
47            try {
48              return
49                  mc.literals.LiteralsHelper.getInstance().decodeString(getSource());
50            }
51            catch (Exception e) {
52              return "";
53            }
54          };
55
56      ast IntLiteral =
57          method public int getValue() {
58            try {
59              return
60                  mc.literals.LiteralsHelper.getInstance().decodeInt(getSource());
61            }
62            catch (NumberFormatException e) {
63              return 0;
64            }
65          };
66
67      ast SignedIntLiteral astextends IntLiteral =
68          method public int getValue() {
69            if (negative) {
70              return - super.getValue();
71            }
72            return super.getValue();
73          };
74
75      ast LongLiteral =
76          method public long getValue() {
77            try {
78              return
79                  mc.literals.LiteralsHelper.getInstance().decodeLong(getSource());
80            }
81            catch (NumberFormatException e) {
82              return 0;
83            }
84          };
85
86      ast SignedLongLiteral astextends LongLiteral =
87          method public long getValue() {
88            if (negative) {
89              return - super.getValue();
90            }
91            return super.getValue();
92          };
93
94      ast FloatLiteral =
95          method public float getValue() {
96            try {
97              return
98                  mc.literals.LiteralsHelper.getInstance().decodeFloat(getSource());
99            }
100           catch (NumberFormatException e) {
101             return 0f;
102           }
103         };
104
```

```
105     ast SignedFloatLiteral astextends FloatLiteral =
106         method public float getValue() {
107           if (negative) {
108             return - super.getValue();
109           }
110           return super.getValue();
111         };
112
113     ast DoubleLiteral =
114         method public double getValue() {
115           try {
116             return
117                 mc.literals.LiteralsHelper.getInstance().decodeDouble(getSource());
118           }
119           catch (NumberFormatException e) {
120             return 0d;
121           }
122         };
123
124     ast SignedDoubleLiteral astextends DoubleLiteral =
125         method public double getValue() {
126           if (negative) {
127             return - super.getValue();
128           }
129           return super.getValue();
130         };
131
132     /** ASTLiteral is the interface for all literals (NullLiteral,
133         BooleanLiteral, CharLiteral, StringLiteral and all NumericLiterals)
134     */
135     interface Literal;
136
137     /** ASTSignedLiteral is the interface for all literals (NullLiteral,
138         BooleanLiteral, CharLiteral, StringLiteral and all NumericLiterals).
139         Compared to Literal it also includes negative NumericLiterals
140     */
141     interface SignedLiteral;
142
143     /** The interface ASTNumericLiteral combines the numeric literal types for
144         Integer, Long, Float and Double
145     */
146     interface NumericLiteral extends Literal;
147
148     /** The interface ASTNumericLiteral combines the numeric literal types for
149         Integer, Long, Float and Double.
150         Compared to NumericLiteral it also includes negative numbers.
151     */
152     interface SignedNumericLiteral extends SignedLiteral;
153
154
155     /*=================================================================*/
156     /*============================ PARSER RULES ===========================*/
157     /*=================================================================*/
158
159     /** ASTNullLiteral represents 'null'
160     */
161     NullLiteral implements Literal, SignedLiteral =
162         "null";
163
164     /** ASTBooleanLiteral represents "true" or "false"
165         @attribute source String-representation (including '"').
166     */
```

```
167     BooleanLiteral implements Literal, SignedLiteral =
168         source:["true" | "false"];
169
170     /** ASTCharLiteral represents any valid character parenthesized with "'".
171         @attribute source String-representation (including "'").
172     */
173     CharLiteral implements Literal, SignedLiteral =
174         source:Char;
175
176     /** ASTStringLiteral represents any valid character sequence parenthesized
177         with '"'.
178         @attribute source String-representation (including '"').
179     */
180     StringLiteral implements Literal, SignedLiteral =
181         source:String;
182
183     /** ASTIntLiteral represents a positive Integer number.
184         @attribute source String-representation (including '"').
185     */
186     IntLiteral implements NumericLiteral =
187         source:Num_Int;
188
189     /** ASTSignedIntLiteral represents a positive or negative Integer number.
190         @attribute source String-representation (including '"').
191     */
192     SignedIntLiteral implements (("-")? Num_Int)=> SignedNumericLiteral =
193         (negative:["-"])? source:Num_Int;
194
195     /** ASTLongLiteral represents a positive Long number.
196         @attribute source String-representation (including '"').
197     */
198     LongLiteral implements NumericLiteral =
199         source:Num_Long;
200
201     /** ASTSignedLongLiteral represents a positive or negative Long number.
202         @attribute source String-representation (including '"').
203     */
204     SignedLongLiteral implements (("-")? Num_Long)=> SignedNumericLiteral =
205         (negative:["-"])? source:Num_Long;
206
207     /** ASTFloatLiteral represents a positive Float number.
208         @attribute source String-representation (including '"').
209     */
210     FloatLiteral implements NumericLiteral =
211         source:Num_Float;
212
213     /** ASTSignedFloatLiteral represents a positive or negative Float number.
214         @attribute source String-representation (including '"').
215     */
216     SignedFloatLiteral implements (("-")? Num_Float)=> SignedNumericLiteral =
217         (negative:["-"])? source:Num_Float;
218
219     /** ASTDoubleLiteral represents a positive Double number.
220         @attribute source String-representation (including '"').
221     */
222     DoubleLiteral implements NumericLiteral =
223        source:Num_Double;
224
225     /** ASTSignedDoubleLiteral represents a positive or negative Double number.
226         @attribute source String-representation (including '"').
227     */
228     SignedDoubleLiteral implements (("-")? Num_Double)=> SignedNumericLiteral =
```

```
229         (negative:["-"])? source:Num_Double;
230
231
232
233     /*======================================================================*/
234     /*========================== LEXER RULES ===============================*/
235     /*======================================================================*/
236
237     // Definition of IDENTs (Java-style)
238     token Name
239         options{testLiterals=true;} = // check Literals first
240         ('a'..'z' | 'A'..'Z' | '_' | '$')
241         ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$')*;
242
243     // Character literals
244     token Char =
245         '\'' ( ESC | ~('\'' | '\n' | '\r' | '\\') ) '\'';
246
247     // String literals
248     token String =
249         '"' (ESC | ~('"' | '\\' | '\n' | '\r'))* '"';
250
251     // Hexadecimal digit (used inside Numbers)
252     protected token Hex_Digit =
253         ('0'..'9'|'A'..'F'|'a'..'f');
254
255     // Suffix of float numbers (used inside Numbers)
256     protected token Float_Suffix =
257         'f'|'F'|'d'|'D';
258
259     // Exponent for decimal numbers, used inside Numbers
260     protected token Decimal_Exponent =
261         ('e'|'E') ('+'|'-')? ('0'..'9')+;
262
263     // Exponent for hexadecimal numbers, used inside Numbers
264     protected token Hex_Exponent =
265         ('p'|'P') ('+'|'-')? ('0'..'9')+;
266
267     // Numbers
268     token Numbers
269         options{testLiterals=true;}
270         {boolean isDecimal=false, isHex=false, isDoubleDot=false;} =
271         ('.'
272           (
273             ('.')
274             |
275             ('.' '.')
276             |
277             (
278               ('0'..'9')+ (Decimal_Exponent)? (t=Float_Suffix)?
279               {
280                 if (t != null && t.getText().toUpperCase().indexOf('F')>=0){
281                   _ttype = Num_Float;
282                 }else{
283                   _ttype = Num_Double; // assume double
284                 }
285               }
286             )?
287           )
288         )
289
290         |
```

```
291
292         (
293           // TODO: check parsing of this comment
294           // special case for '0'
295           ('0' {isDecimal = true; _ttype = Num_Int;}
296             (
297               // HEX
298               (('x'|'X') ((Hex_Digit)+ | ('.' (Hex_Digit)+) | Hex_Exponent)) =>
299               (
300                 ('x'|'X')
301                 // The decimal exponent and the float suffix look like
302                 // hex digits, hence the (...)* doesn't know when to stop
303                 // (-> ambig). ANTLR resolves it correctly by matching
304                 // immediately. It is therefore ok to hush warning.
305                 (options{warnWhenFollowAmbig=false;}: Hex_Digit)*
306                 {isHex = true;}
307               )
308
309               |
310               // FLOAT or DOUBLE with leading zero
311               (('0'..'9')+ ('.'|Decimal_Exponent|Float_Suffix)) => ('0'..'9')+
312
313               |
314               // OCTAL
315               ('0'..'7')+
316             )?
317           )
318
319           |
320           // NON-ZERO DECIMAL
321           (('1'..'9') ('0'..'9')*  {isDecimal=true; _ttype = Num_Int;})
322         )
323         {
324           if (LA(1)=='.' && LA(2)=='.'){
325             isDoubleDot = true;
326           }else{
327             isDoubleDot = false;
328           }
329         }
330         (
331           ('l'|'L') {_ttype = Num_Long;}
332
333           // only check to see if it's a float if looks like decimal so far
334           |
335           {isDecimal && !isDoubleDot}?
336           (
337             {isHex}?
338             ( // exponent is mandatory for floating point hex digits
339               '.' (Hex_Digit)* Hex_Exponent (t=Float_Suffix)?
340               |
341               Hex_Exponent (t=Float_Suffix)?
342               |
343               t=Float_Suffix
344             )
345             |
346             (
347               '.' ('0'..'9')* (Decimal_Exponent)? (t=Float_Suffix)?
348               |
349               Decimal_Exponent (t=Float_Suffix)?
350               |
351               t=Float_Suffix
352             )
```

```
353           )
354           {
355             if (t != null && t.getText().toUpperCase().indexOf('F') >= 0){
356               _ttype = Num_Float;
357             }else{
358               _ttype = Num_Double; // assume double
359             }
360           }
361         )?;
362 
363     // Escape sequence -- note that this is protected; it can only be called
364     // from another lexer rule -- it will not ever directly return a token to
365     // the parser.
366     // There are various ambiguities hushed in this rule. The optional
367     // '0'...'9' digit matches should be matched here rather than letting them
368     // go back to String to be matched. ANTLR does the right thing by matching
369     // immediately; hence, it's ok to shut off the FOLLOW ambig warnings.
370     protected token ESC =
371         '\\'
372         (
373           'n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\' | ('u')+
374           Hex_Digit Hex_Digit Hex_Digit Hex_Digit | '0'..'3'
375           (
376             options{warnWhenFollowAmbig=false;} : '0'..'7'
377             (options{warnWhenFollowAmbig=false;} : '0'..'7')?
378           )?
379           |
380           '4'..'7' (options{warnWhenFollowAmbig=false;} : '0'..'7')?
381         );
382 
383     // Whitespace -- ignored
384     token WS =
385         (
386           ' ' | '\t' | '\f'
387           |
388           (
389             options{generateAmbigWarnings=false;}:
390             "\r\n" | '\r' | '\n'
391           )
392           {newline();}
393         )+
394         {_ttype = Token.SKIP;};
395 
396 }
```

Abbildung C.1.: Grammatik der MontiCore Literals Version 1.3 [Sch12]

C.2. MontiCore Types

MC

```
1  package mc.types;
2
3  version "1.3";
4
5  /**
6   * This grammar defines Java compliant types. The scope of this grammar is to
7   * ease the reuse of type structures in Java-like sublanguages, e.g., by grammar
8   * inheritance or grammar embedment.
9   * The grammar contains types from Java, e.g., primitives, void, types with
10  * dimensions, reference types, generics, and type parameters.
11  *
12  * @author Martin Schindler
13  */
14 grammar Types extends mc.literals.Literals {
15
16     /*=================================================================*/
17     /*============================== OPTIONS ==========================*/
18     /*=================================================================*/
19
20     concept antlr {
21         parser java {
22             /**
23              * Counts the number of LT of type parameters and type arguments.
24              * It is used in semantic predicates to ensure the right number
25              * of closing '>' characters; which actually may have been
26              * either GT, SR (GTGT), or BSR (GTGTGT) tokens.
27              */
28             public int ltCounter = 0;
29         }
30     }
31
32
33     /*=================================================================*/
34     /*====================== INTERFACE DEFINITIONS ====================*/
35     /*=================================================================*/
36
37     ast QualifiedName =
38         method public String toString(){
39           return mc.helper.NameHelper.dotSeparatedStringFromList(
40               this.getParts());
41         };
42
43     ast PrimitiveType =
44         method public String toString(){
45           if (isBoolean()){
46             return "boolean";
47           }
48           if (isByte()){
49             return "byte";
50           }
51           if (isChar()){
52             return "char";
53           }
54           if (isShort()){
55             return "short";
56           }
```

```
57          if (isInt()){
58            return "int";
59          }
60          if (isFloat()){
61            return "float";
62          }
63          if (isLong()){
64            return "long";
65          }
66          if (isDouble()){
67            return "double";
68          }
69          return "";
70        }
71        method public boolean isBoolean(){
72          return this.getPrimitive()==ASTConstantsTypes.BOOLEAN;
73        }
74        method public boolean isByte(){
75          return this.getPrimitive()==ASTConstantsTypes.BYTE;
76        }
77        method public boolean isChar(){
78          return this.getPrimitive()==ASTConstantsTypes.CHAR;
79        }
80        method public boolean isShort(){
81          return this.getPrimitive()==ASTConstantsTypes.SHORT;
82        }
83        method public boolean isInt(){
84          return this.getPrimitive()==ASTConstantsTypes.INT;
85        }
86        method public boolean isFloat(){
87          return this.getPrimitive()==ASTConstantsTypes.FLOAT;
88        }
89        method public boolean isLong(){
90          return this.getPrimitive()==ASTConstantsTypes.LONG;
91        }
92        method public boolean isDouble(){
93          return this.getPrimitive()==ASTConstantsTypes.DOUBLE;
94        };
95
96    ast ReferenceType astextends Type;
97
98    ast ArrayType astimplements ReferenceType =
99        dimensions:/int
100       componentType:Type;
101
102   ast QualifiedType astimplements ReferenceType;
103
104   /** ASTType defines types like primitives, Set, List, Collection, or
105       class types. It might also be an array or generic type.
106   */
107   interface Type extends TypeArgument, ReturnType;
108
109   /** ASTReferenceType defines a reference type like arrays or complex types.
110   */
111   interface ReferenceType;
112
113   /** ASTTypeArgument represents a type argument (generics).
114   */
115   interface TypeArgument;
116
117   /** ASTReturnType represents return types.
118   */
```

```
119 interface ReturnType;
120
121
122 /*========================================================================*/
123 /*============================ PARSER RULES ==============================*/
124 /*========================================================================*/
125
126 /** The ASTQualifiedName represents a possibly qualified name in the AST.
127     The different parts of a qualified name are separated by '.'; they are
128     stored in an ASTStringList.
129     @attribute parts A List of ASTStringList concludes all name parts
130 */
131 QualifiedName =
132     parts:Name (options{greedy=true;}: "." parts:Name)*;
133
134 /** The ASTArrayType represents an array of any type. The rule
135     ComplexArrayType itself treats all arrays except the primitive
136     ones. Especially it treats generic types.
137     @attribute componentType The kind of type which is used for the array.
138                              Could be every complex type.
139     @attribute dimensions    Counts the number of '[]'
140 */
141 ComplexArrayType: ArrayType implements Type returns Type
142 {int ltLevel = ltCounter;} =
143     ret=ComplexType
144     (
145       {ltCounter==ltLevel}?
146       (
147         astscript{!(ComponentType=ret;);}
148         (
149           options{greedy=true;}: "[" "]"
150           {a.setDimensions(a.getDimensions()+1);}
151         )+
152       )
153       | /* let the dims for the enclosing type reference */
154     );
155
156 /** The ASTArrayType represents an array of any type. The rule
157     PrimitiveArrayType itself treats arrays of primitive types, such as
158     'int[]'.
159     @attribute componentType The kind of which is used for the array.
160                              Could be every primitive type.
161     @attribute dimensions    Counts the number of '[]'
162 */
163 PrimitiveArrayType: ArrayType implements (PrimitiveType "[")=>Type
164 returns Type =
165     ret=PrimitiveType
166     (
167       options{greedy=true;}:
168       astscript{!(ComponentType=ret;);}
169       (
170         options{greedy=true;}: "[" "]"
171         {a.setDimensions(a.getDimensions()+1);}
172       )+
173     )?;
174
175 /** ASTVoidType represents the return type "void".
176 */
177 VoidType implements ReturnType =
178     "void";
179
180 /** The BooleanType rule represents boolean primitive types (BOOLEAN).
```

```
181         An instance will be of type ASTPrimitiveType.
182     */
183     BooleanType:PrimitiveType =
184         primitive: ["boolean"];
185
186     /** The IntegralType rule represents integral primitive types
187         (BYTE, SHORT, INT, LONG, or CHAR).
188         An instance will be of type ASTPrimitiveType.
189     */
190     IntegralType:PrimitiveType =
191         primitive: ["byte" | "short" | "int" | "long" | "char"];
192
193     /** The FloatingPointType rule represents floating point primitive types
194         (LONG or DOUBLE).
195         An instance will be of type ASTPrimitiveType.
196     */
197     FloatingPointType:PrimitiveType =
198         primitive: ["float" | "double"];
199
200     /** The NumericType rule represents numeric types
201         (BOOLEAN, BYTE, CHAR, SHORT, INT, FLOAT, LONG, or DOUBLE).
202         An instance will be of type ASTPrimitiveType.
203     */
204     NumericType:PrimitiveType =
205         ret=FloatingPointType | ret=IntegralType;
206
207     /** ASTPrimitiveType represents every primitive type supported by Java.
208         @attribute primitive BOOLEAN, BYTE, CHAR, SHORT, INT, FLOAT, LONG,
209                              or DOUBLE
210     */
211     PrimitiveType implements Type =
212         ret=FloatingPointType | ret=IntegralType | ret=BooleanType;
213
214     /** The ComplexType rule represents a complex type (in contrast to
215         PrimitiveTypes; e.g. class or interface types). It handels
216         SimpleReferenceTypes and QualifiedTypes. An instance will be of type
217         ASTType.
218     */
219     ComplexType returns Type =
220         ret=SimpleReferenceType
221         (options{greedy=true;}: "." ret=QualifiedType->[ret])*;
222
223     /** ASTSimpleReferenceType represents types like class or interface types
224         which could have a qualified name like this: a.b.c<Arg>. The
225         qualification stored in the name (a.b) could be package or a type name.
226         The qualified name could contain type arguments only at the end.
227         a.b.c<Arg>.d would be one ASTSimpleReferenceType (a.b.c<Arg>) and one
228         ASTQualifiedType (d).
229         @attribute name          Name of the type
230                                  Note: Allthough the class name contains the
231                                  word 'simple', the name could be a qualified
232                                  one. So it is saved in an ASTStringList
233         @attribute typeArguments The types between '<...>'
234     */
235     SimpleReferenceType implements ReferenceType =
236         Name
237         (options{greedy=true;}: "." Name)*
238         (options{greedy=true;}: TypeArguments)?;
239
240     /** ASTQualifiedType represents types like class or interface types which
241         always have another ASTQualifiedType or ASTSimpleReferenceType as
242         qualification. So the qualification is in every case a type.
```

```
243         For example:
244         a.b.c<Arg>.d.e<Arg> would be one ASTSimpleReferenceType (a.b.c<Arg>)
245         and two ASTQualifiedType (d; e<Arg>)
246         @attribute name          Name of the type
247         @attribute typeArguments The types between '<...>'
248         @attribute qualification Another ASTQualifiedType or
249                                  ASTSimpleReferenceType.
250     */
251     QualifiedType [qualification:Type] =
252         Name
253         (options{greedy=true;}: TypeArguments)?;
254
255     /** ASTTypeArguments represents a list of generic arguments parenthesized
256         by '<...>'. It is also possible to nest type arguments in each other
257         like this <A<B<C>>>.
258         @attribute typeArguments List of arguments
259     */
260     TypeArguments {int currentLtLevel = 0;} =
261         {currentLtLevel = ltCounter;}
262         (
263           "<" {ltCounter++;}
264           typeArguments:TypeArgument
265           (options{greedy=true;}:
266             /*
267              * The following semantic predicates are needed to construct
268              * trees properly in case of ">>" or ">>>" tokens within nested
269              * TypeArguments (e.g., "var<O1<I1<M1>>, O2<I2>> a;").
270              */
271             {inputState.guessing !=0 || ltCounter == currentLtLevel + 1}?
272                 "," typeArguments:TypeArgument
273           )*
274           (
275             /*
276              * The token stream contains GT, GTGT, or GTGTGT tokens,
277              * so the parser has to expect one of these three possibilities.
278              * Furthermore the angle brackets are counted to check if there
279              * are some left open. After parsing the ltCounter has to be 0,
280              * what is checked by the predicate below.
281              */
282             options{generateAmbigWarnings=false;}:
283             ">" {ltCounter-=1;}
284             | ">>" {ltCounter-=2;}
285             | ">>>" {ltCounter-=3;}
286           )?
287         )
288         // This predicate checks if we have a valid angle bracket structure.
289         {(currentLtLevel != 0) || ltCounter == currentLtLevel}?;
290
291     /** ASTWildcardType represents a wildcard type in a type argument (generics)
292         It could also contain an upper- or lower bound.
293         @attribute upperBound Supertye of the type argument
294         @attribute lowerBound Subtype of the type argument
295     */
296     WildcardType implements TypeArgument =
297         "?" (
298               options{greedy=true;}:
299               ("extends" upperBound:Type) | ("super" lowerBound:Type)
300             )?;
301
302     /** ASTTypeParameters represents a list of generic parameter parenthesized
303         by '<...>' in type declarations (e.g., class-, interface-, method-, or
304         constructor declarations).
```

```
305         @attribute typeVariableDeclarations List of parameters
306     */
307     TypeParameters {int currentLtLevel = 0;} =
308         {currentLtLevel = ltCounter;}
309         (
310           options{greedy=true;}:
311           "<" {ltCounter++;}
312           typeVariableDeclarations:TypeVariableDeclaration
313           (
314             options{greedy=true;}:
315             "," typeVariableDeclarations:TypeVariableDeclaration
316           )*
317           ( // inlined typeArgumentsEnd
318             options{generateAmbigWarnings=false;}:
319             ">" {ltCounter-=1;}
320             | ">>" {ltCounter-=2;}
321             | ">>>" {ltCounter-=3;}
322           )?
323         )
324         // This predicate checks if we have a valid angle bracket structure
325         // (if we are at the "top level" of nested TypeArgument productions).
326         {(currentLtLevel != 0) || ltCounter == currentLtLevel}?
327         | /* nothing! (instead of optionality) */;
328
329     /** ASTTypeVariableDeclaration represents the generic variable declaration
330         in '<'...'>' (e.g., in front of method or constructor declarations or
331         behind the class or interface name).
332         E.g.: public <T extends SuperClass> void test(T t)
333         @attribute name        Name of the type variable
334         @attribute upperBounds Optional list of required super classes
335     */
336     TypeVariableDeclaration =
337         Name
338         (
339           options{generateAmbigWarnings=false;}:
340           "extends" upperBounds:ComplexType
341           ("&" upperBounds:ComplexType)*
342         )?;
343
344 }
```

Abbildung C.2.: Grammatik der MontiCore Types Version 1.3 [Sch12]

C.3. UML/P Common

MC

```
1  package mc.umlp.common;
2
3  version "1.6";
4
5  /**
6   * Grammar for common elements of the UML/P-Language-Family
7   *
8   * @author Martin Schindler
9   */
10 grammar Common extends mc.types.Types {
11
12     options {
13         parser lookahead=3
14         lexer lookahead=7
15     }
16
17     /*=========================================================================*/
18     /*====================== INTERFACE DEFINITIONS ============================*/
19     /*=========================================================================*/
20
21     ast Stereotype =
22         method public boolean containsStereoValue(String name) {
23           return (getStereoValue(name) != null);
24         }
25         method public boolean containsStereoValue(String name, String value) {
26           for (ASTStereoValue sv : values) {
27             if (sv.getName().equals(name)
28                 && sv.getValue() != null && sv.getValue().equals(value)) {
29               return true;
30             }
31           }
32           return false;
33         }
34         method public ASTStereoValue getStereoValue(String name) {
35           for (ASTStereoValue sv : values) {
36             if (sv.getName().equals(name)) {
37               return sv;
38             }
39           }
40           return null;
41         };
42
43     ast StereoValue =
44         method public String getValue() {
45           try {
46             return
47                 mc.literals.LiteralsHelper.getInstance().decodeString(source);
48           }
49           catch (Exception e) {
50             return "";
51           }
52         }
53         method public void setValue(String value) {
54           this.source = '"' + value + '"';
55         };
56
```

```
57      ast Cardinality =
58          lowerBound:/int
59          upperBound:/int;
60
61      external InvariantContent;
62
63
64      /*==========================================================================*/
65      /*============================== GRAMMAR ===================================*/
66      /*==========================================================================*/
67
68      /** ASTStereotype represents Stereotypes in the UML/P
69          @attribute values List of Values of this Stereotype
70      */
71      Stereotype astextends /mc.umlp.common._ast.UMLPNode =
72          "<<" values:StereoValue ("," values:StereoValue)* ">>";
73
74      /** ASTStereoValue represents a Value of a Steretype in the UML/P
75          @attribute name  Name of the Stereotype-Value
76          @attribute source Source of the Stereotype (String including '"';
77                            use getValue() for decoded String)
78      */
79      StereoValue astextends /mc.umlp.common._ast.UMLPNode =
80          Name ("=" source:String)?;
81
82      /** ASTCardinality represents a Cardinality in the UML/P
83          @attribute many          True if "*" is set as Cardinality
84          @attribute lowerBound    Minimum number of associated Classes/Ojects
85          @attribute upperBound    Maximum number of associated Classes/Ojects
86          @attribute lowerBoundLit Lower bound as Literal
87          @attribute upperBoundLit Upper bound as Literal
88          @attribute noUpperLimit  True if no upper bound exists
89      */
90      Cardinality astextends /mc.umlp.common._ast.UMLPNode =
91          "["
92          (
93            many:["*"] {a.setLowerBound(0);a.setUpperBound(0);}
94            |
95            lowerBoundLit:IntLiteral
96            {
97              a.setLowerBound(a.getLowerBoundLit().getValue());
98              a.setUpperBound(a.getLowerBound());
99            }
100           (
101             ".."
102             (
103               upperBoundLit:IntLiteral
104               ({a.setUpperBound(a.getUpperBoundLit().getValue());})
105               |
106               noUpperLimit:["*"] {a.setUpperBound(0);}
107             )
108           )?
109         )
110         "]";
111
112     /** ASTCompleteness represents the completeness in the UML/P
113         Syntax: (left-completeness, right-completeness)
114         Interpretation:
115           CD: Diagramm: left: Types,      right: Assoziations
116               Types:    left: Attributes, right: Methods
117           OD: Diagramm: left: Objects,    right: Links
118               Objects:  left=right: Attributes
```

```
119           SC: Diagramm: left: States (state space coverage),
120                         right: Transitions
121               States:   left: Substates,
122                         right: Transitions of Substates
123           SD: Diagramm: left=right: Interactions:
124                                     (c) = <<match:complete>>
125                                     (...) = <<match:free>>
126               Objects:  left=right: Interactions of this Object
127         @attribute incomplete    true if left and right side are
128                                  incomplete (...)
129         @attribute complete      true if left and right side are
130                                  complete (c)
131         @attribute rightComplete true if only right side is complete (...,c)
132         @attribute leftComplete  true if only left is complete (c,...)
133     */
134     Completeness astextends /mc.umlp.common._ast.UMLPNode =
135         //the following complex lexer symbol construction is to prevent clash
136         //with ident "c"
137         complete:[COMPLETE:"(c)"]
138           | ("(" incomplete:[INCOMPLETE:"..."] ")") //separate brackets to avoid
139                                                    //lexer-symbol clash with SD/SC
140           | incomplete:[INCOMPLETE:"(...,...)"] | complete:[COMPLETE:"(c,c)"]
141           | rightComplete:[RIGHTCOMPLETE:"(...,c)"]
142           | leftComplete:[LEFTCOMPLETE:"(c,...)"];
143
144     /** ASTModifier represents a Modifier for Classes, Interfaces, Methods,
145         Constructors and Attributes in the UML/P
146         @attribute stereotype Optional Stereotype
147         @attribute public     true if Modifier is public
148                               (i.e. Modifier written as "public" or "+")
149         @attribute private    true if Modifier is private
150                               (i.e. Modifier written as "private" or "-")
151         @attribute protected  true if Modifier is protected
152                               (i.e. Modifier written as "protected" or "#")
153         @attribute final      true if Modifier is final
154                               (i.e. Modifier written as "final")
155         @attribute abstract   true if Modifier is abstract
156                               (i.e. Modifier written as "abstract")
157         @attribute local      true if Modifier is local
158                               (i.e. Modifier written as "local")
159         @attribute derived    true if Modifier is derived
160                               (i.e. Modifier written as "derived" or "/")
161         @attribute readonly   true if Modifier is readonly
162                               (i.e. Modifier written as "readonly" or "?")
163         @attribute static     true if Modifier is static
164                               (i.e. Modifier written as "static")
165     */
166     Modifier astextends /mc.umlp.common._ast.UMLPNode =
167         Stereotype?
168         (   Public:["public"]       | Public:[PUBLIC:"+"]   //type modifier
169           | Private:["private"]     | Private:[PRIVATE:"-"]
170           | Protected:["protected"] | Protected:[PROTECTED:"#"]
171           | Final:["final"]
172           | Abstract:["abstract"]
173           | Local:["local"]
174           | Derived:["derived"]     | Derived:[DERIVED:"/"] //attribute modifier
175           | Readonly:["readonly"]   | Readonly:[READONLY:"?"]
176           | Static:["static"]
177         )*;
178
179     /** ASTInvariant represents an Invariant in the UML/P
180         @attribute kind     Kind/Language of the Invariant
```

```
181         @attribute content  Content/condition of the Invariant
182     */
183     / Invariant astextends /mc.umlp.common._ast.UMLPNode =
184         (kind:Name ":")?
185         "[" content:InvariantContent(parameter kind) "]";
186 
187 }
```

Abbildung C.3.: Grammatik der MontiCore UML/P Common Version 1.3 [Sch12]

C.4. Java

MC

```
package mc.javadsl;

version "1.1.5";

grammar JavaDSL extends mc.types.Types {

    options {
        lexer lookahead=4
        keywords {"."}
    }

    /** The CompilationUnit represents a whole JAVA file
        @attribute PackageDeclaration Represnts the package, the file belonging to
        @attribute ImportDeclarations A list of the ImportDeclaration
        @attribute TypeDeclarations a list of declarated types in the file
    */
    CompilationUnit =
        (((Annotation)* "package")=> PackageDeclaration:PackageDeclaration)?
        (ImportDeclarations:ImportDeclaration)*
        (TypeDeclarations:TypeDeclaration | ";" )*
    ;

    /** ASTPackageDeclaration represents a package declaration.
        @attribute Modifiers Annotations in front of the declaration
        @attribute Name Qualified package name
    */
    PackageDeclaration =
        // Semantic: only one package name component is allowed to be annotated
       (Modifiers:Annotation)*
       "package"
       Name ( "." Name )*
       ";"
    ;

    /** ASTImportDeclaration represents an import declaration
        @attribute Name Qualified name of the class or the package, which is
                        imported
        @attribute IsStatic Is true if keyword static is found in the declaration
        @attribute IsOnDemand Is true if declaration ends with a '*', so all
                              classes of a package are imported
    */
    ImportDeclaration =
        "import" (IsStatic:["static"])?
        Name (options{greedy=true;}: "." Name)*
        ("." IsOnDemand:["*"])?
        ";"
    ;

    /** A TypeDeclaration represents the possible top level types in JAVA, which
        can be a class declaration, enum declaration, interface declaration
        or annotation type declaration. The rule redirects "only" to the passing
        rules.
    */
    interface TypeDeclaration extends
        (TypeDeclaration)=>AnnotationMemberDeclaration,
        (TypeDeclaration)=>MemberDeclaration;
```

```
57
58      ast TypeDeclaration =
59          Name
60          Modifiers:Modifier*
61      ;
62
63      /** ASTVariableDeclaration represents a whole variable declaration. It consist
64          of modifiers, a type and one or more VariableDeclarators seperated by ';'
65          @attribute Modifiers A list of the modifier
66          @attribute Type The variable type
67          @attribute Declarators List of VariableDeclarators
68      */
69      VariableDeclaration =
70          (Modifiers:Modifier)*
71          Type:Type
72          Declarators:VariableDeclarator
73          (options{warnWhenFollowAmbig=false;}: "," Declarators:VariableDeclarator)*
74      ;
75
76      /** ASTVariableDeclarationStatement represents just an VariableDeclaration
77          followed by an ";". Only a redirect to VariableDeclaration, but needed
78          to make an Statement of it
79      */
80      VariableDeclarationStatement implements (VariableDeclaration ";")=>Statement =
81          Declaration:VariableDeclaration ";"
82      ;
83
84      /** ASTVariableDeclarationExpression represents just an
85          VariableDeclarationExpression. Only a redirect to VariableDeclaration,
86          but needed to make an Expression of it
87      */
88      VariableDeclarationExpression implements (VariableDeclaration)=>Expression =
89          Declaration:VariableDeclaration
90      ;
91
92      /** The interface Modifer redirect to an Annotation or a PrimitveModifier
93      */
94      interface Modifier;
95
96      /** ASTPrimitiveModifier represents the modifier keywords.
97          @attribute Modifier Modifier represents one of the Modifer Keywords:
98                              "private", "public", "protected", "static",
99                              "transient", "final", "abstract", "native",
100                             "threadsafe", "synchronized", "const",
101                             "volatile", "strictfp"
102     */
103     PrimitiveModifier implements Modifier =
104         Modifier:["private" | "public" | "protected" | "static" | "transient"
105         | "final" | "abstract" | "native" | "threadsafe" | "synchronized"
106         | "const" | "volatile" | "strictfp"]
107     ;
108
109     /** An EnumDeclaration is the declaration for an enum type.
110         @attribute Modifiers A list of Modifiers
111         @attribute ImplementedTypes A list of implemented interfaces
112         @attribute EnumConstantDeclarations A list of EnumConstandDeclarations
113         @attribute MemberDeclarations A list of MemberDclarations (a class body)
114     */
115     EnumDeclaration implements (Modifier* "enum")=>TypeDeclaration =
116         (Modifiers:Modifier)* "enum" Name
117         ("implements" ImplementedTypes:Type ("," ImplementedTypes:Type)* )?
118         "{"
```

```
119           (EnumConstantDeclarations:EnumConstantDeclaration
120           (options{greedy=true;}:
121             "," EnumConstantDeclarations:EnumConstantDeclaration)*)?
122           (",")? // Note: This comma will not be stored in AST,
123                  //       so the Pretty Printer can't print it
124           (";" (MemberDeclarations:MemberDeclaration | ";")* )?
125         "}"
126     ;
127
128     /** An EnumConstantDeclaration is in fact a class instance and can be followed
129         by the usual class declarations
130         @attribute Annotations A list of Annotations
131         @attribute Arguments A list of Expressions
132         @attribute MemberDeclarations A list of MemberDeclaration
133     */
134     EnumConstantDeclaration =
135         (Annotations:Annotation)* Name
136         (
137           "("
138             (Arguments: Expression ("," Arguments: Expression)* )?
139            ")"
140         )?
141         (
142           "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"
143           {a.getMemberDeclarations().set_Existent(true);}
144         )?
145     ;
146
147     /** An AnnotationTypeDeclaration is a special kind of interface declaration for
148         annotations marked with an "@"
149         @attribute Modifers A list of (Interface-) Modifers
150         @attribute Name The name of the annotation
151         @attribute MemberDeclarations A list of MemberDeclaration, similar to
152                                       class body
153     */
154     AnnotationTypeDeclaration implements (Modifier* "@")=>TypeDeclaration =
155         //semantic: this is an interface, not all Modifers allowed
156         (Modifiers:Modifier)* "@" "interface" Name
157         ("extends" extendedAnnotation:Type)?
158         "{"
159           (MemberDeclarations:AnnotationMemberDeclaration | ";")*
160         "}"
161     ;
162
163     /** A AnnotationMemberDeclaration represents the possible member declarations
164         which can be used in an annotation type declaration. This rule only
165         redirects to the passing rules
166     */
167     interface AnnotationMemberDeclaration;
168
169     ast AnnotationMemberDeclaration astextends MemberDeclaration;
170
171     /** An annotation method declaration are special kind of mthod declarations
172         which are only used in an annotation member declaration (and so in an
173         annotation type declaration)
174         @attribute Modifiers A list of Modifier
175         @attribute Type The type of the method
176         @attribute Name The name of the declarated method
177         @attribute DefaultValue An AnnotationMemberValue which is an optional
178                                 default value in the method declaration
179     */
180     AnnotationMethodDeclaration
```

```
181     implements ((Modifier)* ReturnType Name "(" ")")=>AnnotationMemberDeclaration =
182         //semantic check: modifers are only allowed to be:
183         //                public, abstract, annotation
184         (Modifiers:Modifier)* ReturnType Name "(" ")"
185         ("default" DefaultValue:AnnotationMemberValue )? ";"
186     ;
187
188     /** There are three kinds of Annotation-types: The MapeddMemberAnnoation,
189         SingleMemberAnnotation and the MarkerAnntoation. The Annotation rule only
190         redirects to the under rules.
191     */
192     interface Annotation extends Modifier;
193
194     ast Annotation astextends Expression, Modifier;
195
196     /** The MappedMemberAnnotation is the normal way to construct an annotation. It
197         exists of a TypeName, a StringList to construct qualified names, followed by
198         an list of MemberValuePairs,
199         e.g. @javax.persistence.JoinTable(name="a_name", schema="a_schema")
200         @attribute Name Name is the TypeName of an annotation
201         @attribute MemberValue MemberValue is a AnnotationMemberValuePair
202     */
203     MappedMemberAnnotation
204     implements ("@" Name ("." Name)* "(" AnnotationMemberValuePair)=>Annotation =
205         "@" Name (options{greedy=true;}:  "." Name)*
206         "("
207           MemberValues:AnnotationMemberValuePair
208           ("," MemberValues:AnnotationMemberValuePair)*
209         ")"
210     ;
211
212     /** The SingleMemberAnnotation is a short hand design for the normal
213         MappedMemberAnnotation for use with single element annotation. It exists
214         of a TypeName, a StringList to construct qualified names, followed by a
215         single AnnotationMemberValue, e.g. @SuppressWarnings("unchecked")
216         @attribute Name Name is the TypeName of an annotation
217         @attribute MemberValue MemberValue an AnnotationMemberValue
218     */
219     SingleMemberAnnotation
220     implements ("@" Name ("." Name)* "(" AnnotationMemberValue)=>Annotation =
221         "@" Name (options{greedy=true;}: "." Name)*
222         "(" MemberValue:AnnotationMemberValue ")"
223     ;
224
225     /** The MarkerAnnotation is a very short hand design for the normal
226         MappedMemberAnnotation for use with only the Name, the marker. It only
227         exists of a the TypeName followed by an optional "()", e.g. @WebService
228         @attribute Name Name is the TypeName of an annotation
229     */
230     MarkerAnnotation implements Annotation =
231         "@" Name (options{greedy=true;}:  "." Name)*
232         ("(" ")" )?
233     ;
234
235     /** AnnotationMemberValuePair is a single identifier followed by an
236         AnnotationMemberValue.
237         @attribute MemberName The member name is an identifier
238         @attribute Value The value is by the type of an AnnotationMemberValue
239     */
240     AnnotationMemberValuePair =
241         MemberName:Name "=" Value:AnnotationMemberValue
242     ;
```

```
243
244     /** AnnotationMemberValue only redirect to three other rules: to an annotation,
245         a conditional expression or to a annotation member array initializer.
246         There never will be instances of this ASTNode.
247     */
248     AnnotationMemberValue returns Expression =
249         ret = Annotation
250         | ret = ConditionalExpression
251         | ret = AnnotationMemberArrayInitializer
252     ;
253
254     /** AnnotationMemberArrayInitializer is an array which only content
255         AnnoationMemberValues
256         @attribute Initializers List of AnnotationMemberValues
257     */
258     //Need to be an Expression because of the rule AnnotationMemberValue
259     AnnotationMemberArrayInitializer
260     implements ("{" (AnnotationMemberValue ("," AnnotationMemberValue)*)?
261                (",")? "}")=>Expression =
262         "{"
263           (
264             Initializers:AnnotationMemberValue
265             (
266               options{warnWhenFollowAmbig=false;}:
267               "," Initializers:AnnotationMemberValue
268             )*
269           )?
270           (",")? // Note: This comma will not be stored in AST,
271                  //       so the Pretty Printer can't print it
272         "}"
273     ;
274
275     /** A class declaration represents a whole class with its block identified by
276         the keyword "class".
277         @attribute Modifers A Modifierlist with all possible Modifer
278         @attribute Name The class name
279         @attribute TypeParameters A list of type paramters
280         @attribute ExtendedType A Type which is extended by the class.
281                                 Only one allowed!
282         @attribute ImplementedInterfaces A list of implemented interfaces.
283         @attribute MemberDeclarations The class body is represented by the
284                                       "{ classBody }", which exists of multiple
285                                       MemeberDeclaration or a ";"
286     */
287     ClassDeclaration implements ((Modifiers:Modifier)* "class")=>TypeDeclaration =
288         /* accords to classBody; rule couldn't be converted*/
289         //semantic check: not all modifers allowed
290         (Modifiers:Modifier)* "class" Name
291         TypeParameters
292         ("extends" ExtendedClass:Type)?
293         ("implements" ImplementedInterfaces:Type ("," ImplementedInterfaces:Type)*)?
294         "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"
295     ;
296
297     /** An interface declaration represents a whole interface, which is very
298         similar to a class.
299         Identified by the keyword interface
300         @attribute Modifers A Modifierlist with all possible Modifer
301         @attribute Name The interface name
302         @attribute TypeParameters A list of type paramters
303         @attribute ExtendedTypes A list of extended types. Multiple allowed
304         @attribute MemberDeclarations The class body is represented by the
```

```
305                                     "{ classBody }", which exists of multiple
306                                     MemeberDeclaration or a ";". In comparing
307                                     with the class declaration, here not all
308                                     possible member declaration allowed.
309     */
310     InterfaceDeclaration implements (Modifier* "interface")=>TypeDeclaration =
311         //semantic check: not all modifers allowed
312         (Modifiers:Modifier)* "interface" Name
313         TypeParameters
314         ("extends" ExtendedInterfaces:Type ("," ExtendedInterfaces:Type)* )?
315         "{"( MemberDeclarations:MemberDeclaration  | ";" )* "}"
316         //accords to classBody; rule couldn't be converted
317         //semantic check for allowed member declarations
318     ;
319
320     /** ASTMemberDeclaration is an interface for ASTTypeDeclaration,
321         ASTMethodDeclarathion, ASTConstructorDeclaration and ASTFieldDeclaration.
322     */
323     interface MemberDeclaration;
324
325     MemberDeclarations = MemberDeclaration*;
326
327     /** ASTTypeInitializer represents a type initializer, which is just a
328         ASTBlockStatement headed by an optional 'static' modifier
329         @attribute IsStatic Is true if 'static' keyword is present
330     */
331     TypeInitializer implements (("static")? BlockStatement)=>MemberDeclaration =
332         (IsStatic:["static"])?
333         Body:BlockStatement
334     ;
335
336     /** ASTMethodOrConstructorDeclaration is an shared interface of the two classes
337     */
338     interface MethodOrConstructorDeclaration; //This rule is never invocated
339
340     ast MethodOrConstructorDeclaration astextends MemberDeclaration =
341         Name
342         Throwables:QualifiedName*
343         Modifiers:Modifier*
344         TypeParameters
345         Parameters:ParameterDeclaration*
346         Block:BlockStatement
347     ;
348
349     /** The ASTConstructorDeclaration represents the declaration of a constructor
350         @attribute Modifiers List of modifier or annotations
351         @attribute TypeParameter Contains a list of generic TypeParameter (optional)
352         @attribute Name Single name of the constructor
353         @attribute Parameters List of parameters
354         @attribute Throwables List of the qualified names of exceptions, which could
355                                 be thrown by the constructor
356         @attribute Block the actual block statement
357     */
358     ConstructorDeclaration
359     implements ((Modifier)* TypeParameters Name "(")=>MemberDeclaration =
360         //Modifiers and TypeParameters are moved from rule memberDeclaration
361         (Modifiers:Modifier)*
362         TypeParameters
363         Name
364         "("
365           (Parameters:ParameterDeclaration ("," Parameters:ParameterDeclaration)*)?
366         ")"
```

```
367         ( //this is the inlined rule thowsClause
368           "throws" Throwables:QualifiedName ("," Throwables:QualifiedName)*
369         )?
370         Block:BlockStatement
371     ;
372
373     ast ConstructorDeclaration astimplements MethodOrConstructorDeclaration;
374
375     /** The ASTMethodDeclaration represents the declaration of a method
376         @attribute Modifiers List of modifier or annotations
377         @attribute TypeParameter Contains a list of generic TypeParameter(optional)
378         @attribute ReturnType The type of the return value of the method
379         @attribute Name Single name of the constructor
380         @attribute Parameters List of parameters
381         @attribute Throwables List of the qualified names of exceptions, which could
382                                 be thrown by the constructor
383         @attribute Block the actual block statement
384     */
385     MethodDeclaration
386     implements ((Modifier)* TypeParameters ReturnType Name "("=>MemberDeclaration =
387         //Modifiers and TypeParameters are moved from rule memberDeclaration
388         (Modifiers:Modifier)*
389         TypeParameters
390         ReturnType Name
391         "("
392           (Parameters:ParameterDeclaration ("," Parameters:ParameterDeclaration)*)?
393         ")"
394         //this is the inlined rule optionalDims
395         (
396           options{greedy=true;}: "[" "]"
397           {a.setAdditionReturnTypeDimensions(
398               a.getAdditionReturnTypeDimensions()+1);}
399         )*
400         ( //this is the inlined rule thowsClause with the inlined rule greedyName
401           "throws" Throwables:QualifiedName ("," Throwables:QualifiedName)*
402         )?
403         ( ";" | Block:BlockStatement )
404     ;
405
406     Methods =
407         MethodDeclaration+;
408
409     ast MethodDeclaration astimplements MethodOrConstructorDeclaration =
410         AdditionReturnTypeDimensions : /int
411     ;
412
413     /** The ASTFieldDeclaration represents the declaration of a field. This is just
414         a VariableDeclaration followed by a ';'.
415         @attribute Declaration One VariableDeclaration
416     */
417     FieldDeclaration
418     implements ((Modifier)* Type VariableDeclarator)=>MemberDeclaration,
419                ((Modifier)* Type VariableDeclarator)=>AnnotationMemberDeclaration =
420         //Modifiers are moved from rule memberDeclaration to VariableDeclaration
421         Declaration:VariableDeclaration ";"
422     ;
423
424     /** The ASTVariableDeclarator represents the part of a variable declaration
425         beginning at the name, e.g. 'varName = "someInitializer" '
426         It is also part of an field declaration
427         @attribute Name Variable name
428         @attribute AdditionalArrayDimensions Counts the dimension if variable is
```

```
429                                         an array
430         @attribute Represents the initial assignment
431     */
432     VariableDeclarator =
433         Name
434         (
435           options{greedy=true;}: "[" "]"
436           {a.setAdditionalArrayDimensions(a.getAdditionalArrayDimensions()+1);}
437         )*
438         ("=" Initializer:Initializer)?
439     ;
440
441     ast VariableDeclarator =
442         AdditionalArrayDimensions:/int
443     ;
444
445     /** The ASTArrayInitializer represents the statement
446         in curly braces (incl. this braces!), which is used to initialize
447         an array.
448         @attribute Initializers A list of expressions or other arrayinitializers
449     */
450     ArrayInitializer implements Expression =
451         "{"
452         (
453           Initializers:Initializer
454           (options{warnWhenFollowAmbig=false;}: "," Initializers:Initializer)*
455         )?
456         (",")?  // Note: This comma will not be stored in AST,
457                 //       so the Pretty Printer can't print it
458         "}"
459     ;
460
461     /** The rule Initializer is necessary to combine Expression an ArrayInitializer.
462         It represents the X from this example: int[] a = { X , X }
463         Never returns an ASTInitializer object
464     */
465     Initializer returns Expression =
466         ret=Expression
467     ;
468
469     /** ASTParameterDeclaration represents the declaration of a parameter in
470         a method or constructor declaration
471         @attribute Modifiers List of modifier
472         @attribute Type Parameter type
473         @attribute Ellipsis True if type is declared with '...'. So parameter could
474                             accept more than one argument of this type. Note: The
475                             grammar allows things like this in the moment:
476                             method(int... i, int... j, int k). Java 1.5 allows just
477                             one of this parameter in the end of the parameter list
478         @attribute Name Parameter name
479         @attribute AdditionalArrayDimensions Counts dimensions if parameter is an
480                                             array
481     */
482     ParameterDeclaration =
483         (Modifiers:Modifier)* Type:Type
484         (Ellipsis:["..."])?
485         Name
486         (
487           options{greedy=true;}: "[" "]"
488           {a.setAdditionalArrayDimensions(a.getAdditionalArrayDimensions()+1);}
489         )*
490     ;
```

```
491
492     ast ParameterDeclaration =
493         AdditionalArrayDimensions:/int
494     ;
495
496     /** A BlockStatement represents a block which is rounded by: "{...}". In the
497         Block could stand any Statement. This is used in many contexts:
498         Inside a class definition prefixed with "static": it is a class initializer.
499         Inside a class definition without "static": it is an instance initializer.
500         As the body of a method. As a completely indepdent braced block of code
501         inside a method:
502         it starts a new scope for variable definitions
503         @attribute Statement Represents any Statements within the Block
504     */
505     BlockStatement implements ("{")=>Statement =
506         "{" (Statements:Statement)* "}"
507     ;
508
509     /** The interface Statement contains all possible Statements and redirect to
510         the passing rules
511     */
512     interface Statement;
513
514     Statements = Statement*;
515
516     /** An ExpressionStatement is a lonely Expression followed by a ";"
517         @attribute Expression The  for a ";"
518     */
519     ExpressionStatement implements (Expression ";")=>Statement =
520         Expression:Expression ";"
521     ;
522
523     /** A TypeDeclarationStatement is the capsule of a TypeDeclaration in a
524         Statement
525         @attribute TypeDeclaration Represents a TypeDeclaration
526     */
527     TypeDeclarationStatement implements Statement =
528         TypeDeclaration:TypeDeclaration
529     ;
530
531     /** An ASTEmptyStatement represents a single semicolon
532     */
533     EmptyStatement implements Statement =
534         ";"
535     ;
536
537     /** An ASTContinueStatement represents the continue keyword
538         eventually followed by a jump label.
539         @attribute Label Name of the jump label.
540     */
541     ContinueStatement implements Statement =
542         "continue" (Label:Name)? ";"
543     ;
544
545     /** An ASTSwitchDefaultStatement represents the default keyword
546         in a switch statement.
547         */
548     SwitchDefaultStatement astimplements Statement =
549         "default"
550     ;
551
552     /** An ASTBreakStatement represents the break keyword
```

```
553         eventually followed by a jump label.
554         @attribute Label Name of the jump label.
555     */
556     BreakStatement implements Statement =
557         "break" (Label:Name)? ";"
558     ;
559
560     /** A ReturnStatement: "return a ;". The Expression a is optional
561         @attribute Expression The Expression which is returns is optional
562     */
563     ReturnStatement implements Statement =
564         "return" (Expression:Expression)? ";"
565     ;
566
567     SwitchStatement implements Statement  =
568         "switch" "(" Expression:Expression ")"
569         "{"
570         (
571           (
572             options{greedy=true;}:
573             (Statements:CaseStatement | Statements:SwitchDefaultStatement) ":"
574           )+
575           (Statements:Statement)*
576         )*
577         "}"
578     ;
579
580     ast SwitchStatement =
581         Statements:Statement*
582     ;
583
584     /** A SwitchCaseStatement is only the "case a" construct of a switch construct
585         @attribute Expression The Expression which is proofed for a special case
586     */
587     CaseStatement astimplements Statement =
588         "case" Expression:Expression
589     ;
590
591     /** A ThrowStatement represents a construct like this: "throw Expression ;"
592         @attribute Expression the "target" in a throwStatement
593     */
594     ThrowStatement implements Statement =
595         "throw" Expression:Expression ";"
596     ;
597
598     /** An AssertStatement is a checking for a boolean Expression, the Asserting,
599         if the value of Asserting is false the Message-Expression gets evualated
600         @attribute Asserting The Asserting is an Expression which gets evualated
601                              for a boolean value. A false-value throws an
602                              "AssertionError".
603         @attribute Message   If the value of the Asserting is false, the Message
604                              get evualated and his value is converted to a String
605                              using string conversion. The "detail message"
606                              of the "AssertionError" is then the value of Message
607     */
608     AssertStatement implements Statement =
609         "assert" Assertion:Expression (":" Message:Expression)? ";"
610     ;
611
612     /** A WhileStatement represents a loop. The Condition gets evualated and if the
613         value is a boolean "true" the Statement gets evualated.
614         @attribute Condition The Condition gets evualated for a boolean value
```

```
615         @attribute Statement The Statement gets evualated if the Condition-value is
616                              true
617     */
618     WhileStatement implements Statement =
619         "while" "(" Condition:Expression ")" Statement:Statement
620     ;
621
622     /** A DoWhileStatement represents a loop. The Statement is in any case executed
623         for once, then the Condition gets evualated and if the value is a boolean
624         "true" the Statement the loop begins for another time.
625         @attribute Statement The Statement gets evualated
626         @attribute Condition The Condition gets evualated for a boolean value
627     */
628     DoWhileStatement implements Statement =
629         "do" Statement:Statement "while" "(" Condition:Expression ")" ";"
630     ;
631
632     /** A for-each-statement is the enhanced form of a for-loop. The loop gets
633         executed for every element the iterable has. The actual value get "saved"
634         in the VariableDeclaration.
635         @attribute VariableDeclaration is variable that gets a value from th
636                                        iterable expresion for each loop execution
637         @attribute Iterable The Iterable is an Expression by type of an iterable
638         @attribute Statement Statement gets executed every time the loop gets
639                              executed
640     */
641     ForEachStatement implements ("for" "(" ParameterDeclaration ":")=>Statement =
642         "for" "("
643           VariableDeclaration:ParameterDeclaration ":" Iterable:Expression
644           //semantic check: don't allow '...' in VariablenDeclaration
645         ")" Statement:Statement
646     ;
647
648     /** A for-basic-statement is the normal form of a for-loop with an initializer,
649         an boolean-condition, an update expression and a executive statement
650         @attribute Initialization A Initialization is a ForInitializer
651         @attribute Updates The Update is a List of Expressions which gets evualated
652                            for every time the loop gets executed.
653         @attribute Statement The Statement gets executed if the Conditon is true,
654                              absent or the value of it is not a boolean.
655     */
656     ForStatement implements Statement =
657         "for" "("
658           (
659             (VariableDeclaration)=> Initializations:VariableDeclarationExpression
660             |
661             Initializations:Expression ("," Initializations:Expression)*
662           )?
663           ";" (Condition:Expression)?
664           ";" (Updates:Expression ("," Updates:Expression)*)?
665         ")" Statement:Statement
666     ;
667
668     ast ForStatement =
669         Initializations:Expression*
670     ;
671
672     /** A IfStatement evualated the Expression for a boelean value. For a true the
673         Statement get executed, for a false the ElseStatement.
674         @attribute Condition The Condition gets evualated for a boolean value
675         @attribute Statement The Statement gets executed if the Expression value is
676                              a true
```

```
677         @attribute ElseStatement The ElseStatement gets evualated if the Expression
678                                  vaule is false
679     */
680     IfStatement implements Statement =
681         "if" "(" Condition:Expression ")" SuccessStatement:Statement
682         (options{greedy=true;}: "else" OptionalElseStatement:Statement)?
683     ;
684
685     /** A Statement could be labeled, so it is possible to "jump" to the Statement
686         using the label.
687         @attribute Label The Label is only the "name" of the label
688         @attribute Statement The Statement followed the Label
689     */
690     LabeledStatement implements Statement =
691         Label:Name ":" Statement:Statement
692     ;
693
694     /** A SynchronizedStatement locks the Expression and evualted the Block.
695         @attribute Expression The Expression which gets lock (or get synchronizes
696                               with the Block)
697         @attribute Block The Block gets executed, when the Expression is locked
698     */
699     SynchronizedStatement implements Statement =
700         "synchronized" "(" Expression:Expression ")" Block:BlockStatement
701     ;
702
703     /** A TryStatement is a capsuled Block who is tried to execute, possible errors
704         can be "catched" by the catch clauses and the right clause gets executed.
705         The finally clause gets executed in every case independet if an error
706         occured or not an also if an error get catched by the catch clauses.
707         @attribute Block The Block gets executed and can of course occure an error
708         @attribute CatchClause The CatchClause can catch a specific error and gets
709                                then executed
710         @attribute FinallyClause The FinallyClause, if existents, gets executed in
711                                  every case
712     */
713     TryStatement implements Statement =
714         "try" Block:BlockStatement
715         (CatchClause:CatchClause)*
716         (FinallyClause:FinallyClause)?
717     ;
718
719     /** A FinallyClause belongs to a try-catch-finally-block (see TryStatement).
720         @attribute Block see TryStatement
721     */
722     FinallyClause =
723         "finally" Block:BlockStatement
724     ;
725
726     /** A CatchClause belongs to a try-catch-block (see TryStatement).
727         @attribute Block see TryStatement
728     */
729     CatchClause =
730         "catch" "(" ExceptionVariable:ParameterDeclaration ")" Block:BlockStatement
731     ;
732
733     // Expressions
734     // Note that most of these expressions follow the pattern
735     //   thisLevelExpression :
736     //       nextHigherPrecedenceExpression
737     //           (OPERATOR nextHigherPrecedenceExpression)*
738     // which is a standard recursive definition for a parsing an expression.
```

```
739     // The operators in java have the following precedences:
740     //    lowest  (13)  = *= /= %= += -= <<= >>= >>>= &= = |=
741     //            (12)  ?:
742     //            (11)  ||
743     //            (10)  &&
744     //            ( 9)  |
745     //            ( 8)
746     //            ( 7)  &
747     //            ( 6)  == !=
748     //            ( 5)  < <= > >=
749     //            ( 4)  << >>
750     //            ( 3)  +(binary) -(binary)
751     //            ( 2)  * / %
752     //            ( 1)  ++ -- +(unary) -(unary)  ~  !  (type)
753     //                  []   () (method call)  . (POINT -- identifier qualification)
754     //                  new   ()  (explicit parenthesis)
755     //
756     // The last two are not usually on a precedence chart; I put them in
757     // to point out that new has a higher precedence than '.', so you
758     // can validy use
759     //     new Frame().show()
760     //
761     // Note that the above precedence levels map to the rules below...
762     // Once you have a precedence chart, writing the appropriate rules as below
763     // is usually very straightfoward.
764
765     /** The interface expression represents all java expressions.
766     */
767     interface Expression;
768
769     /** A AssignmentExpression represents a "x = y"
770         with a =, +=, -=, *=, /=, %=, >>=, >>>=, <<=, &=,^=, |= as the operator
771         @attribute LeftHand The left hand is a ConditionalExpression, the "x"
772         @attribute RightHand The right hand is a AssignmentExpression, the "y"
773         @attribute AssignmentOperator = (EQUALS), += (PLUSEQUALS), -= (MINUSEQUALS),
774                                         *= (STAREQUALS), /= (SLASHEQUALS),
775                                         %= (PERCENTEQUALS), >>= (GTGTEQUALS),
776                                         >>>= (GTGTGTEQUALS), <<= (LTLTEQUALS),
777                                         &= (ANDEQUALS),^= (ROOFEQUALS),
778                                         |= (PIPEEQUALS)
779     */
780     AssignmentExpression
781     implements (ConditionalExpression
782                 (("="|"+="|"-="|"*="|"/="|"%="|">>="|">>>="|"<<="|"&="|"^="|"|=")
783                 AssignmentExpression)?)=>Expression
784     returns Expression =
785         ret=ConditionalExpression
786         (
787           astscript{!(LeftHand=ret;);}
788           AssignmentOperator:["=" | "+=" | "-=" | "*=" | "/="  | "%="
789                               | ">>=" | ">>>=" | "<<=" | "&=" | "^=" | "|="]
790           RightHand:AssignmentExpression
791         )?
792     ;
793
794     ast AssignmentExpression =
795         LeftHand:Expression
796     ;
797
798     /*  This classgen is needed to produce a attribut LeftOperand with the
799         type Expression and not ExpressionList like it would be created from
800         the expression rules
```

```
801     */
802     ast InfixExpression =
803         LeftOperand:Expression
804         RightOperand:Expression
805         Operator:/int
806     ;
807
808     /** A ConditionalExpression is something like a "if-then-else" Expression in
809         a short Way. it looks like "a? b : c.
810         @attribute Condition A LogicalOrExpression which can (must) be boolean
811                               Statement
812         @attribute ifTrue A AssignmentExpression which is executed when Condition
813                           is a "true"-Statement
814         @attribute ifFalse A ConditionalExpression which is executed when Condition
815                            is a "false"-Statement
816     */
817     ConditionalExpression implements (LogicalOrExpression)=>Expression
818     returns Expression =
819         ret=LogicalOrExpression
820         (
821           astscript{!(Condition=ret;);}
822           "?" IfTrue:AssignmentExpression
823           ":" IfFalse:ConditionalExpression
824         )?
825     ;
826
827     ast ConditionalExpression =
828         Condition:Expression
829     ;
830
831     /** A LogicalOrExpression represents a "x || y"
832         with a || as the operator
833         @attribute LeftOperand The left operand is a LogicalAndExpression, the "x"
834         @attribute RightOperand The right Operand is a LogicalAndExpression, the "y"
835         @attribute Operator || (PIPEPIPE)
836     */
837     LogicalOrExpression:InfixExpression
838     implements (LogicalAndExpression)=>Expression returns Expression =
839         ret=LogicalAndExpression //ret accords to the left operand
840         (
841           astscript{!(LeftOperand=ret;);}
842           Operator:["||"] RightOperand:LogicalAndExpression
843         )*
844     ;
845
846     /** A LogicalAndExpression represents a "x && y"
847         with a && as the operator
848         @attribute LeftOperand The left operand is a BitWiseOrExpression, the "x"
849         @attribute RightOperand The right Operand is a BitWiseOrExpression, the "y"
850         @attribute Operator && (ANDAND)
851     */
852     LogicalAndExpression:InfixExpression
853     implements (BitWiseOrExpression)=>Expression returns Expression =
854         ret=BitWiseOrExpression //ret accords to the left operand
855         (
856           astscript{!(LeftOperand=ret;);}
857           Operator:["&&" ] RightOperand:BitWiseOrExpression
858         )*
859     ;
860
861     /** A BitWiseOrExpression represents a "x | y"
862         with a | as the operator
```

```
863         @attribute LeftOperand The left operand is a XorExpression, the "x"
864         @attribute RightOperand The right Operand is a XorExpression, the "y"
865         @attribute Operator | (PIPE)
866     */
867     BitWiseOrExpression:InfixExpression
868     implements (XorExpression)=>Expression returns Expression =
869         ret=XorExpression //ret accords to the left operand
870         (
871           astscript{!(LeftOperand=ret;);}
872           Operator:["|" ] RightOperand:XorExpression
873         )*
874     ;
875
876     /** A XorExpression represents a "x ^ y"
877         with a ^ as the operator
878         @attribute LeftOperand The left operand is a BitWiseAndExpression, the "x"
879         @attribute RightOperand The right Operand is a BitWiseAndExpression, the "y"
880         @attribute Operator ^ (ROOF)
881     */
882     XorExpression:InfixExpression
883     implements (BitWiseAndExpression)=>Expression returns Expression =
884         ret=BitWiseAndExpression //ret accords to the left operand
885         (
886           astscript{!(LeftOperand=ret;);}
887           Operator:["^"] RightOperand:BitWiseAndExpression
888         )*
889     ;
890
891     /** A BitWiseAndExpression represents a "x & y"
892         with a & as the operator
893         @attribute LeftOperand The left operand is a EqualityExpression, the "x"
894         @attribute RightOperand The right Operand is a EqualityExpression, the "y"
895         @attribute Operator & (AND)
896     */
897     BitWiseAndExpression:InfixExpression
898     implements (EqualityExpression)=>Expression returns Expression =
899         ret=EqualityExpression //ret accords to the left operand
900         (
901           astscript{!(LeftOperand=ret;);}
902           Operator:["&" ] RightOperand:EqualityExpression
903         )*
904     ;
905
906     /** A EqualityExpression represents a "x == y"
907         with a == or != as the operator
908         @attribute LeftOperand The left operand is a RelationalExpression, the "x"
909         @attribute RightOperand The right Operand is a RelationalExpression, the "y"
910         @attribute Operator == (EQUALSEQUALS), != (EXCLAMATIONMARKEQUALS)
911     */
912     EqualityExpression:InfixExpression
913     implements (RelationalExpression)=>Expression returns Expression =
914         ret=RelationalExpression //ret accords to the left operand
915         (
916           astscript{!(LeftOperand=ret;);}
917           Operator:["!=" | "=="] RightOperand:RelationalExpression
918         )*
919     ;
920
921     /** A RelationalExpression represents a "x > y"
922         with a <, >, <= or >= as the operator
923         @attribute LeftOperand The left operand is a ShiftExpression, the "x"
924         @attribute RightOperand The right Operand is a ShiftExpression, the "y"
```

```
925                               or a "instancOfExpression"
926         @attribute Operator < (LT), > (GT), <= (LTEQUALS), >= (GTEQUALS)
927     */
928     RelationalExpression:InfixExpression
929     implements (ShiftExpression)=>Expression returns Expression =
930         ret=ShiftExpression //ret accords to the left operand
931         (
932           (
933             astscript{!(LeftOperand=ret;);}
934             Operator:["<" | ">" | "<=" | ">="] RightOperand:ShiftExpression
935           )*
936           |
937           ret=InstanceOfExpression->[ret]
938         )
939     ;
940
941     /** A InstancOfExpression is a: "Expression instancof Type"
942         @attribute Expression The Expression which is checked to be the Type
943         @attribute Type The right Hand, a Type
944     */
945     InstanceOfExpression
946     implements ("instanceof" Type)=>Expression [Expression:Expression] =
947         "instanceof" Type:Type
948     ;
949
950     /** A ShiftExpression represents a "x >> y"
951         with a <<, >> or >>> as the operator
952         @attribute LeftOperand The left operand is a AdditiveExpression, the "x"
953         @attribute RightOperand The right Operand is a AdditiveExpression, the "y"
954         @attribute Operator << (LTLT), >> (GTGT), >>> (GTGTGT)
955     */
956     ShiftExpression:InfixExpression
957     implements (AdditiveExpression)=>Expression returns Expression =
958         ret=AdditiveExpression //ret accords to the left operand
959         (
960           astscript{!(LeftOperand=ret;);}
961           Operator:["<<" | ">>" | ">>>"] RightOperand:AdditiveExpression
962         )*
963     ;
964
965     /** A AdditiveExpression represents a simple "x + y"
966         with a "+" or "-" as the operator
967         @attribute LeftOperand The left operand is a MultiplicativeExpression,
968                                the "x"
969         @attribute RightOperand The right Operand is a MultiplicativeExpression,
970                                 the "y"
971         @attribute Operator + (PLUS), - (MINUS)
972     */
973     AdditiveExpression:InfixExpression
974     implements (MultiplicativeExpression)=>Expression returns Expression =
975         ret=MultiplicativeExpression //ret accords to the left operand
976         (
977           astscript{!(LeftOperand=ret;);}
978           Operator:["+" | "-"] RightOperand:MultiplicativeExpression
979         )*
980     ;
981
982     /** A MultiplicativeExpression represents a simple "x * y"
983         with a "*", "/" or "%" (= Modulo) as a Operand
984         @attribute LeftOperand The left operand is a PrefixExpression, the "x"
985         @attribute RightOperand The right Operand is a PrefixExpression, the "y"
986         @attribute Operator * (STAR), / (LASH) , % (PRECENT)
```

```
987     */
988     MultiplicativeExpression:InfixExpression
989     implements (PrefixExpression)=>Expression returns Expression =
990         ret = PrefixExpression
991         (
992           astscript { !(LeftOperand=ret;);}
993           Operator:["*"|"/" | "%"] RightOperand:PrefixExpression
994         )*
995     ;
996
997     /** An ASTPrefixExpression represents an prefix increment, decrement or
998         negotiation of an expression. This rule conforms approximately to
999         the unaryExpression rule of the antlr version of this grammar.
1000        @attribute Expression The expression to apply the operator to.
1001        @attribute Op ++ (PLUSPLUS), -- (MINUSMINUS), + (PLUS), - (MINUS),
1002                      ~ (TILDE) or ! (EXCLAMATIONMARK)
1003    */
1004    PrefixExpression
1005    implements (("++" | "--" | "-" | "+" | "~" | "!") | CastExpression)=>Expression
1006    returns Expression =
1007        (
1008          ( astscript{!();}
1009            Operator: ["++"  | "--" | "-" | "+" |  "~" | "!"]
1010            Expression:PrefixExpression
1011          )
1012          |
1013          ret=CastExpression
1014        )
1015    ;
1016
1017    /** An ASTCastExpression represents an cast of an expression.
1018        This rule conforms approximately to the unaryExpressionNotPlusMinus rule
1019        of the antlr version of this grammar.
1020        @attribute Expression The expression to apply the cast to.
1021        @attribute TargetType The cast type
1022    */
1023    CastExpression implements (
1024                    ("(" PrimitiveArrayType ")")=> ("(" PrimitiveArrayType ")")
1025                    |
1026                    ("(" ComplexArrayType ")" CastExpression)=>
1027                    ("(" ComplexArrayType ")" CastExpression)
1028                    |
1029                    PostfixExpression
1030                  )=>Expression returns Expression =
1031      (
1032        //BinaryNot and LogicalNot handling moved to the rule PrefixExpression
1033        // use predicate to skip cases like: (int.class)
1034        ("(" PrimitiveArrayType ")" )=>
1035        // explicit cast:
1036        ( astscript{!();}
1037          "(" TargetType:PrimitiveArrayType ")" Expression:PrefixExpression
1038        )
1039        // Have to backtrack to see if operator follows.  If no operator
1040        // follows, it's a typecast.  No semantic checking needed to parse.
1041        // if it _looks_ like a cast, it _is_ a cast; else it's a "(expr)"
1042        |
1043        ( "(" ComplexArrayType ")" CastExpression)=>
1044        // explicit cast:
1045        ( astscript{!();}
1046          "(" TargetType:ComplexArrayType ")" Expression:CastExpression
1047        )
1048        | ret=PostfixExpression
```

```
1049        )
1050    ;
1051
1052    /** An ASTPostfixExpression represents an postfix
1053        increment or decrement of an expression
1054        @attribute Expression The expression to increment/decrement
1055        @attribute Op ++ (PLUSPLUS) or -- (MINUSMINUS)
1056    */
1057    PostfixExpression
1058    implements (Primary ("++"|"--")?)=>Expression returns Expression =
1059        ret=Primary
1060        (astscript { !(Expression=ret;); } Operator:["++"|"--"] )?
1061    ;
1062
1063    ast PostfixExpression =
1064        Expression:Expression
1065    ;
1066
1067    /** ASTConstructorInvocation represents explicit constructor invocations of the
1068        own class (e.g. this();) or of the super class (e.g. super();)
1069        @attribute TypeArguments Generic arguments
1070        @attribute Reference This- or SuperReference
1071        @attribute Arguments List of parameter, which are handed to the constructor
1072    */
1073    ConstructorInvocation implements
1074      (TypeArguments? (ThisReference->[null] | SuperReference->[null])
1075      "(" (Expression ("," Expression)*)? ")" )=>Expression =
1076        //semantic: check that there are no wildcard types in the list
1077        (TypeArguments:TypeArguments )?
1078        (
1079          Reference:ThisReference->[null]
1080          | Reference:SuperReference->[null]
1081        )
1082        "("
1083          (Arguments: Expression ("," Arguments: Expression)*)?
1084        ")"
1085    ;
1086
1087    /** This rule is needed for explicit constructor invocations with a
1088        qualification like 'new Outer().super()'. Qualified invocations
1089        can just used with 'super'.
1090    */
1091    QualifiedConstructorInvocation: ConstructorInvocation
1092    implements (ConstructorInvocation "." )=>Expression [qualification=Expression] =
1093        (TypeArguments:TypeArguments )?
1094        //semantic: check, if qualification is a legal expression
1095        Reference:SuperReference->[qualification]
1096        "("
1097          (Arguments: Expression ("," Arguments: Expression)*)?
1098        ")"
1099    ;
1100
1101    ast ConstructorInvocation =
1102        Reference:Expression
1103    ;
1104
1105    /** The Primary parser rule is necessary to encapsulate some other rules like
1106        ThisReference, MethodInvocation, FielAccess... But there are never
1107        produced ASTPrimary objects.
1108    */
1109    Primary returns Expression =
1110        ret=PrimarySuffix
```

```
1111        (
1112          "." ret=ThisReference->[ret]
1113          |
1114          "."
1115          (
1116            ((TypeArguments)? MethodInvocationWithSingleName->[null,null])=>
1117            (typeArguments=TypeArguments)?
1118            ret=MethodInvocationWithSingleName->[ret, typeArguments]
1119            |
1120            (FieldAccess->[null])=>
1121            ret=FieldAccess->[ret]
1122            |
1123            (QualifiedConstructorInvocation->[null])=>
1124            ret=QualifiedConstructorInvocation->[ret]
1125            |
1126            superReference=SuperReference->[ret]  "."
1127            (
1128              ((TypeArguments)? Name "(")=>
1129              (typeArguments=TypeArguments)?
1130              ret=MethodInvocationWithSingleName->[superReference, typeArguments]
1131              |
1132              ret=FieldAccess->[superReference]
1133            )
1134            |
1135            ret=NewExpression->[ret]
1136          )
1137          |
1138          ret=ArrayAccessExpression->[ret]
1139        )*
1140    ;
1141
1142    /** The ASTThisReference represents just the 'this' keyword
1143        @attribute Qualification The qualification is the x in x.this
1144    */
1145    ThisReference implements ("this")=>Expression [Qualification:Expression] =
1146        "this"
1147    ;
1148
1149    /** The ASTFieldAccess represents an access to an field.
1150        @attribute Name Name of the field
1151        @attribute Qualification
1152    */
1153    FieldAccess implements (Name)=>Expression [Qualification:Expression]=
1154        Name
1155    ;
1156
1157    /** The ASTMethodInvocation represents a method invocation
1158        @attribute Name the single name of the method
1159        @attribute Qualification The qualification of method1 is the expression X
1160                                 of this example: X.qualification()
1161        @attribute TypeArguments The generic arguments of the method
1162    */
1163    MethodInvocationWithSingleName:MethodInvocation
1164    implements (Name "(")=>Expression
1165    [Qualification:Expression, TypeArguments:TypeArguments] =
1166        Name
1167        "("
1168          (Arguments: Expression ("," Arguments: Expression)*)?
1169        ")"
1170    ;
1171
1172    /** The ASTMethodInvocation represents a method invocation
```

```
1173        @attribute Name the qualified name of the method
1174        @attribute Qualification The qualification of method1 is the expression X
1175                                 of this example: X.qualification()
1176        @attribute TypeArguments The generic arguments of the method
1177    */
1178    MethodInvocationWithQualifiedName:MethodInvocation
1179    implements (Name (options{greedy=true;}: "." Name)* "("=>Expression
1180    [ Qualification:Expression, TypeArguments:TypeArguments] =
1181        Name
1182        (options{greedy=true;}: "." Name)*
1183        "("
1184          (Arguments: Expression ("," Arguments: Expression)*)?
1185        ")"
1186    ;
1187
1188    /** An ASTArrayAccessExpression stands for a simple: "a[b]"
1189        @attribute Receiver The Receiver is the "a" and must not be null
1190        @attribute Component The Component is the b in the brackets and
1191                             must not be null
1192    */
1193    ArrayAccessExpression
1194    implements ("[" Expression "]")=>Expression [Receiver:Expression] =
1195        "[" Component:Expression "]"
1196    ;
1197
1198    /** The PrimarySuffix parser rule is necessary to encapsulate some other rules
1199        garant a correct redirect
1200    */
1201    PrimarySuffix returns Expression =
1202        ((Type|VoidType) "." "class")=> ret=ClassAccess
1203        | (MethodInvocationWithQualifiedName->[null, null])=>
1204          (ret=MethodInvocationWithQualifiedName->[null, null])
1205        | ret=QualifiedName
1206        | ret=Literal
1207        | ret=NewExpression->[null]
1208        | (ConstructorInvocation)=>
1209          ret=ConstructorInvocation
1210        | ret=ThisReference->[null]
1211        | ret=SuperReference->[null]
1212        | ret=ParenthesizedExpression
1213    ;
1214
1215    /** ASTParenthesized represents any Epxression parenthesized with brackets
1216        '(...)'
1217        @attribute Expression The expression between the brackets.
1218    */
1219    ParenthesizedExpression implements ("(" AssignmentExpression ")")=>Expression =
1220        "(" Expression:AssignmentExpression ")"
1221    ;
1222
1223    /** An ASTClassAccess represents an
1224        @attribute Type Type represents the Type in a "Type.class"
1225    */
1226    ClassAccess implements ((Type|VoidType) "." "class")=>Expression =
1227        (Type|VoidType) "." "class"
1228    ;
1229
1230    /** An ASTSuperReference represents the Literal "super" and a
1231        qualification X used in the form "X.super..." which can be used for
1232        SuperMethodInvocations or FieldAccesses like used in the rule
1233        primary2
1234        @attribute Qualification Qualification is the X in "X.super..."
```

```
1235      */
1236      SuperReference implements ("super")=>Expression [Qualification:Expression] =
1237          "super"
1238      ;
1239
1240      /** The ASTNewExpression represents a ClassInstantiation or an
1241          ArrayInstantiation. It always begins with the keyword 'new'.
1242          There are never ASTNewExpression objects.
1243      */
1244      NewExpression [qualification=Expression] returns Expression =
1245          "new" (typeArgs=TypeArguments)?
1246          (type=ComplexType | type=PrimitiveType)
1247          (
1248            ret=ClassInstantiation->[qualification, typeArgs, type]
1249            | ret=ArrayInstantiation->[qualification, typeArgs, type]
1250          )
1251      ;
1252
1253      /** ASTInstantiation just combines the two classes ASTArrayInstantiation and
1254          ASTClassInstantiation
1255      */
1256      interface Instantiation;
1257
1258      ast Instantiation astextends Expression =
1259          Qualification:Expression
1260          TypeArguments:TypeArguments
1261          Type:Type
1262      ;
1263
1264      /** The ASTClassInstantiation represents the part of a class instantiation from
1265          the '(' on.
1266          @attribute Qualification The outer class if you instantiate a inner class
1267          @attribute TypeArguments The generic type arguments
1268          @attribute Type The class which is initialized
1269          @attribute MemberDeclarations The body of the class
1270          @attribute ConstructorArguments Arguments handed to the constructor
1271      */
1272      ClassInstantiation
1273      [Qualification:Expression, TypeArguments:TypeArguments, Type:Type] =
1274          "("
1275            (ConstructorArguments: Expression
1276            ("," ConstructorArguments: Expression)*)?
1277          ")"
1278          (
1279            options{greedy=true;}:
1280            "{"
1281              (MemberDeclarations:MemberDeclaration | ";")*
1282            "}"
1283            {a.getMemberDeclarations().set_Existent(true);}
1284          )? //This Line accords to '(classBody)?'. Classbody couldn't be converted
1285      ;
1286
1287      ast ClassInstantiation astimplements Instantiation;
1288
1289      /** The ASTArrayInstantiation represents the part of a array instantiation from
1290          the '[' on.
1291          @attribute Qualification The outer class if you instantiate a inner class
1292          @attribute TypeArguments The generic type arguments
1293          @attribute Type The class which is initialized
1294          @attribute Initializer Initial values for the Array
1295          @attribute ConstructorArguments Arguments handed to the constructor
1296      */
```

```
1297    ArrayInstantiation
1298    [Qualification:Expression, TypeArguments:TypeArguments, Type:Type] =
1299        (
1300          (options{greedy=true;}: "["(Sizes:Expression) "]")+
1301          (options{greedy=true;}: "[" "]" {a.setDims(a.getDims()+1);})*
1302          |
1303          ("[" "]" {a.setDims(a.getDims()+1);})+
1304          Initializer:ArrayInitializer
1305        )
1306    ;
1307
1308    ast ArrayInstantiation astimplements Instantiation =
1309        Dims:/int
1310    ;
1311
1312    // Override mc.types.Types
1313    /** The ASTQualifiedName represents a possibly qualified name in the AST.
1314        The different parts of a qualified name are separated by '.'; they are
1315        stored in an ASTStringList.
1316        @attribute parts A List of ASTStringList concludes all name parts
1317    */
1318    QualifiedName implements Expression =
1319        parts:Name (options{greedy=true;}: "." parts:Name)*;
1320
1321    // Override mc.literals.Literals
1322    interface Literal extends Expression;
1323
1324    /** ASTNullLiteral represents 'null'
1325    */
1326    NullLiteral implements Literal;
1327
1328    /** ASTBooleanLiteral represents "true" or "false"
1329        @attribute source String-representation (including '"').
1330    */
1331    BooleanLiteral implements Literal;
1332
1333    /** ASTCharLiteral represents any valid character parenthesized with "'".
1334        @attribute source String-representation (including "'").
1335    */
1336    CharLiteral implements Literal;
1337
1338    /** ASTStringLiteral represents any valid character sequence parenthesized
1339        with '"'.
1340        @attribute source String-representation (including '"').
1341    */
1342    StringLiteral implements Literal;
1343
1344    /** The interface ASTNumericLiteral combines the numeric literal types for
1345        Integer, Long, Float and Double
1346    */
1347    interface NumericLiteral extends Literal;
1348
1349    /** ASTIntLiteral represents an Integer number.
1350        @attribute source String-representation (including '"').
1351    */
1352    IntLiteral implements NumericLiteral;
1353
1354    /** ASTLongLiteral represents a Long number.
1355        @attribute source String-representation (including '"').
1356    */
1357    LongLiteral implements NumericLiteral;
1358
```

```
1359     /** ASTFloatLiteral represents a Float number.
1360         @attribute source String-representation (including '"'). 
1361     */
1362     FloatLiteral implements NumericLiteral;
1363 
1364     /** ASTDoubleLiteral represents a Double number.
1365         @attribute source String-representation (including '"'). 
1366     */
1367     DoubleLiteral implements NumericLiteral;
1368 
1369     concept attributes {
1370         syn JavaTypeEntry: /mc.javadsl.ets.entries.JavaTypeEntry;
1371         global SymbolTable: /interfaces2.helper.SymbolTableInterface;
1372     }
1373 
1374     concept antlr {
1375         lexer java {
1376           /** flag for enabling the "assert" keyword */
1377           private boolean assertEnabled = true;
1378           /** flag for enabling the "enum" keyword */
1379           private boolean enumEnabled = true;
1380           /** Enable the "assert" keyword */
1381           public void enableAssert() { assertEnabled = true; }
1382           /** Disable the "assert" keyword */
1383           public void disableAssert() { assertEnabled = false; }
1384           /** Query the "assert" keyword state */
1385           public boolean isAssertEnabled() { return assertEnabled; }
1386           /** Enable the "enum" keyword */
1387           public void enableEnum() { enumEnabled = true; }
1388           /** Disable the "enum" keyword */
1389           public void disableEnum() { enumEnabled = false; }
1390           /** Query the "enum" keyword state */
1391           public boolean isEnumEnabled() { return enumEnabled; }
1392 
1393           /** This method overrides one in the CharScannner. This is necessary,
1394            *  because the literal table returns for "enum" and "assert" always
1395            *  LITERAL_enum and LITERAL_assert. Allthough if this features has
1396            *  been disabled.
1397           */
1398           @Override
1399           public int testLiteralsTable(int ttype) {
1400             int ret;
1401             ret = super.testLiteralsTable(ttype);
1402             if (!assertEnabled
1403                 && "assert".equals(
1404                         new String(text.getBuffer(), 0, text.length()))) {
1405               ret = Name;
1406             }
1407             if (!enumEnabled
1408                 && "enum".equals(
1409                         new String(text.getBuffer(), 0, text.length()))) {
1410               ret = Name;
1411             }
1412             return ret;
1413           }
1414         }
1415     }
1416 }
```

Abbildung C.4.: Grammatik der Java/P Version 1.1.5 [FM07]

C.5. Klassendiagramme

MC

```
1  package mc.umlp.cd;
2
3  version "1.6";
4
5  /**
6   * Grammar for UML/P-Classdiagrams
7   *
8   * @author Martin Schindler
9   */
10 grammar CD extends mc.umlp.common.Common {
11
12     options {
13         compilationunit CDDefinition
14     }
15
16     /*=========================================================================*/
17     /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
18     /*=========================================================================*/
19
20     /** ASTCDElement represents all elements of a UML Classdiagram */
21     abstract / CDElement astextends /mc.umlp.common._ast.UMLPNode;
22
23     /** ASTCDType represents all type-defining elements of a UML Classdiagram */
24     abstract / CDType extends CDElement;
25
26     /** ASTCDCallable represents a common abstract class for
27         ASTCDConstructor and ASTCDMethod */
28     abstract / CDCallable astextends /mc.umlp.common._ast.UMLPNode;
29
30     // for editor
31     ast CDAssociation =
32         arrow:/String
33         method public String getArrow(){
34           if (leftToRight) {
35             return "->";
36           } else if (rightToLeft) {
37             return "<-";
38           } else if (bidirectional) {
39             return "<->";
40           }
41           return "--";
42         };
43
44     // common attributes of ASTCDClass, ASTCDInterface, and ASTCDEnum
45     ast CDType =
46         Completeness
47         Modifier
48         Name
49         interfaces:/mc.types._ast.ASTReferenceTypeList
50         cDAttributes:/ASTCDAttributeList
51         cDMethods:/ASTCDMethodList;
52
53     // common attributes of ASTCDConstructor and ASTCDMethod
54     ast CDCallable =
55         Modifier
56         TypeParameters
```

```
57          Name
58          cDParameters:/ASTCDParameterList
59          exceptions:/mc.types._ast.ASTQualifiedNameList
60          body:/mc.ast.ASTNode;
61
62      external Value;
63      external Body;
64
65
66      /*=================================================================*/
67      /*============================== GRAMMAR ==============================*/
68      /*=================================================================*/
69
70      /** ASTCDDefinition represents a UML Classdiagram
71          @attribute completeness   Optional Comleteness of this Classdiagramm
72          @attribute stereotype     Optional Stereotype
73          @attribute name           Name of this Classdiagram
74          @attribute cDClasses      List of Classes of this Classdiagram
75          @attribute cDInterfaces   List of Interfaces of this Classdiagram
76          @attribute cDEnums        List of Enums of this Classdiagram
77          @attribute cDAssociations List of Associations of this Classdiagram
78          @attribute invaritants    List of Invaritants of this Classdiagram
79      */
80      / CDDefinition astextends /mc.umlp.common._ast.UMLPNode =
81          Completeness?
82          Stereotype?
83          "classdiagram" Name
84          "{"
85          (
86            (Completeness? Modifier "class")=>
87            cDClasses:CDClass
88            |
89            (Completeness? Modifier "interface")=>
90            cDInterfaces:CDInterface
91            |
92            (Completeness? Modifier "enum")=>
93            cDEnums:CDEnum
94            |
95            (Stereotype? ("association"|"aggregation"|"composition"))=>
96            cDAssociations:CDAssociation
97            |
98            invariants:Invariant
99          )*
100         "}";
101
102     /** ASTCDClass represents a Class in a UML Classdiagram
103         @attribute completeness   Optional Comleteness of this Class
104         @attribute modifier       Modifier of this Class
105         @attribute name           Name of this Class
106         @attribute typeParameters Generic type parameters of this Class
107         @attribute superclasses   List of Superclasses of this Class
108         @attribute interfaces     List of Interfaces implemented by this Class
109         @attribute cDConstructors List of Constructors of this Class
110         @attribute cDMethods      List of Methods of this Class
111         @attribute cDAttributes   List of Attributes of this Class
112     */
113     / CDClass astextends CDType =
114         Completeness?
115         Modifier
116         "class" Name
117         (options{greedy=true;}: TypeParameters)?
118         ("extends" superclasses:ReferenceType
```

```
119                   ("," superclasses:ReferenceType)*)?
120         ("implements" interfaces:ReferenceType
121                   ("," interfaces:ReferenceType)*)?
122         (
123           ("{" (
124             (Modifier Type Name ("=" | ";"))
125             => cDAttributes:CDAttribute
126             |
127             (Modifier (options{greedy=true;}: TypeParameters)? Name "(")
128             => cDConstructors:CDConstructor
129             |
130             cDMethods:CDMethod
131           )* "}")
132           |
133           ";"
134         );
135
136
137       /** ASTCDInterface represents an Interface in a UML Classdiagram
138           @attribute completeness   Optional Comleteness of this Interface
139           @attribute modifier       Modifier of this Interface
140           @attribute name           Name of this Interface
141           @attribute typeParameters Generic type parameters of this Interface
142           @attribute interfaces     List of Interfaces extended by this Interface
143           @attribute cDMethods      List of Methods of this Interface
144           @attribute cDAttributes   List of Attributes of this Interface
145       */
146       / CDInterface astextends CDType =
147           Completeness?
148           Modifier
149           "interface" Name
150           (options{greedy=true;}: TypeParameters)?
151           ("extends" interfaces:ReferenceType
152                     ("," interfaces:ReferenceType)*)?
153           (
154             ("{" (
155               (Modifier Type Name ("=" | ";"))
156               => cDAttributes:CDAttribute
157               |
158               cDMethods:CDMethod
159             )* "}")
160             |
161             ";"
162           );
163
164       /** ASTCDEnum represents an Enumeration (enum type) in a UML Classdiagram
165           @attribute completeness    Optional Comleteness of this Enum
166           @attribute modifier        Modifier of this Enum
167           @attribute name            Name of this Enum
168           @attribute interfaces      List of Interfaces implemented by this Enum
169           @attribute cDEnumConstants List of the Enum Constants
170           @attribute cDConstructors  List of Constructors of this Enum
171           @attribute cDMethods       List of Methods of this Enum
172           @attribute cDAttributes    List of Attributes of this Enum
173       */
174       / CDEnum astextends CDType =
175           Completeness?
176           Modifier
177           "enum" Name
178           ("implements" interfaces:ReferenceType
179                       ("," interfaces:ReferenceType)*)?
180           (
```

```
181             ("{"(
182               cDEnumConstants:CDEnumConstant
183               ("," cDEnumConstants:CDEnumConstant)* ";"
184               (
185                 (Modifier Type Name ("=" | ";"))
186                 => cDAttributes:CDAttribute
187                 |
188                 (Modifier ("<" TypeVariableDeclaration
189                 ("," TypeVariableDeclaration)* (">" | ">>" | ">>>")?)?
190                 Name "("
191                 => cDConstructors:CDConstructor
192                 |
193                 cDMethods:CDMethod
194               )*
195             )?"}")
196             |
197             ";"
198         );
199
200
201     /** ASTCDEnumConstant represents a Constant of an Enumeration (enum type)
202         in a UML Classdiagram
203         @attribute name             Name of this Constant
204         @attribute cDEnumParameters List of optional parameters of this Constant
205     */
206     / CDEnumConstant astextends /mc.umlp.common._ast.UMLPNode =
207         Name ("("
208           cDEnumParameters:CDEnumParameter
209           ("," cDEnumParameters:CDEnumParameter)*
210         ")")?;
211
212
213     /** ASTCDEnumParameter represents a Parameter of an Enumeration Constant
214         @attribute Value Value of this Parameter
215     */
216     / CDEnumParameter astextends /mc.umlp.common._ast.UMLPNode =
217         Value;
218
219
220     /** ASTCDMethod represents a Method of a Class or Interface
221         @attribute modifier       Modifier of this Method
222         @attribute typeParameters Generic type parameters of this Method
223         @attribute returnType     Return-Type of the return value of this Method
224         @attribute name           Name of this Method
225         @attribute cDParameters   List of Parameters of this Method
226         @attribute exceptions     List of Exceptions thrown by this Method
227         @attribute body           Body of this Method
228     */
229     / CDMethod astextends CDCallable =
230         Modifier
231         (options{greedy=true;}: TypeParameters)?
232         ReturnType
233         Name "(" (
234           cDParameters:CDParameter
235           (options{greedy=true;}: "," cDParameters:CDParameter)*
236         )? ")"
237         (("throws")=> "throws"
238           exceptions:QualifiedName
239           (options{greedy=true;}: "," exceptions:QualifiedName)*)?
240         (("{")=>Body | ";");
241
242
```

```
243     /** ASTCDConstructor represents a Constructor of a Class or Interface
244         @attribute modifier       Modifier of this Constructor
245         @attribute typeParameters Generic type parameters of this Constructor
246         @attribute name           Name of this Constructor
247         @attribute cDParameters   List of Parameters of this Constructor
248         @attribute exceptions     List of Exceptions thrown by this Constructor
249         @attribute body           Body of this Constructor
250     */
251     / CDConstructor astextends CDCallable =
252         Modifier
253         (options{greedy=true;}: TypeParameters)?
254         Name
255         "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
256         (("throws")=> "throws"
257           exceptions:QualifiedName
258           (options{greedy=true;}: "," exceptions:QualifiedName)*)?
259         (("{")=> Body | ";");
260
261
262     /** ASTCDParameter represents a Parameter of a Constructor or Method
263         @attribute type Type of this Parameter
264         @attribute ellipsis Optional indicator for a variable Parameter length
265         @attribute name Name of this Parameter
266     */
267     / CDParameter astextends /mc.umlp.common._ast.UMLPNode =
268         Type (Ellipsis:["..."])? Name;
269
270
271     /** ASTCDAttribute represents an Attribute of a Class or Interface
272         @attribute modifier Modifier of this Attribute
273         @attribute type     Type of this Attribute
274         @attribute name     Name of this Attribute
275         @attribute value    Value of this Attribute
276     */
277     / CDAttribute astextends /mc.umlp.common._ast.UMLPNode =
278         Modifier
279         Type
280         Name
281         ("=" Value)? ";";
282
283
284     /** ASTQualifier represents a Qualifier of an Association
285         @attribute name     Abstract name of the Qualifier if not a type
286         @attribute type     Type of the Qualifier if not an abstract name
287                             (i.e. attribute name)
288     */
289     / CDQualifier astextends /mc.umlp.common._ast.UMLPNode =
290         (Name)=> Name | Type;
291
292
293     /** ASTCDAssociation represents an Association between Classes or Interfaces
294         @attribute stereotype         Optional Stereotype
295         @attribute Association        True if Association is of type "association"
296         @attribute Aggregation        True if Association is of type "aggregation"
297         @attribute Composition        True if Association is of type "composition"
298         @attribute type               Type of the Association (Association,
299                                       Aggregation, or Composition)
300         @attribute derived            True if this is a derived Association
301         @attribute name               Name of this Association
302         @attribute leftModifier       Optional left side Modifier
303         @attribute leftCardinality    Cardinality of the left side of this
304                                       Association
```

```
305         @attribute leftReferenceName  Name of the Class or Interface on the left
306                                       side of this Association
307         @attribute leftQualifier      Qualifier of the left side of this
308                                       Association
309         @attribute leftRole           Role of the Class or Interface on the left
310                                       side of this Association
311         @attribute leftToRight        True if Association is navigable from
312                                       left to right ("->")
313         @attribute rightToLeft        True if Association is navigable from
314                                       right to left ("<-")
315         @attribute bidirectional      True if Association is navigable in
316                                       both directions ("<->")
317         @attribute simple             True if navigation of Association is
318                                       not specified  ("--")
319         @attribute rightRole          Role of the Class or Interface on the right
320                                       side of this Association
321         @attribute rightQualifier     Qualifier of the right side of this
322                                       Association
323         @attribute rightReferenceName Name of the Class or Interface on the right
324                                       side of this Association
325         @attribute rightCardinality   Cardinality of the right side of this
326                                       Association
327         @attribute rightModifier      Optional right side Modifier
328     */
329     / CDAssociation astextends CDElement =
330         Stereotype?
331         (
332           Association:["association"]
333           | Aggregation:["aggregation"]
334           | Composition:["composition"]
335         )
336         (options{greedy=true;}: Derived:[DERIVED:"/"])?
337         //predicate necessary to prevent clash with LeftReference
338         ((Name Modifier Cardinality? QualifiedName)=> Name |)
339         leftModifier:Modifier
340         leftCardinality:Cardinality?
341         leftReferenceName:QualifiedName
342         ("[" leftQualifier:CDQualifier "]")?
343         ("(" leftRole:Name ")")?
344         (
345           leftToRight:["->"]
346           | rightToLeft:["<-"]
347           | bidirectional:["<->"]
348           | simple:["--"]
349         )
350         ("(" rightRole:Name ")")?
351         ("[" rightQualifier:CDQualifier "]")?
352         rightReferenceName:QualifiedName
353         rightCardinality:Cardinality?
354         rightModifier:Modifier ";";
355
356 }
```

Abbildung C.5.: Grammatik der UML/P Klassendiagramme Version 1.6 [Sch12]

C.6. Seitenbeschreibung

MC

```
1  package de.montiwis.page;
2
3  version "1.0";
4
5  /**
6   * Grammar for page description in MontiWIS
7   *
8   * @author Dirk Reiß
9   */
10 grammar Page extends mc.umlp.common.Common {
11
12   options {
13             parser lookahead=3
14             lexer lookahead=7
15             compilationunit PageDefintion
16   }
17
18   /*=========================================================================*/
19   /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
20   /*=========================================================================*/
21
22     interface PageElement;
23     interface TextContents;
24
25   /*=========================================================================*/
26   /*============================== GRAMMAR ==================================*/
27   /*=========================================================================*/
28
29    /** PageDefinition represents a page.
30        @attribute stereotype    optional stereotype for this page
31        @attribute name          the name of the page
32        @attribute parameters    optional parameters of the page
33        @attribute pageElement   List of an arbitrary number of PageElements
34     */
35     PageDefintion =
36         Stereotype?
37       "page" name:Name
38       (
39          (
40          Parameters?
41          "{"
42           PageElement*
43          "}"
44          )
45       |
46          (
47          ";"
48          )
49       );
50
51     /** Parameters represents an arbitrary number of Parameter elements.
52         @attribute param        List of an arbitrary number of Parameter elements
53     */
54     Parameters =
55       "(" param:Parameter? ("," param:Parameter )* ")";
56
```

```
57      /** Parameter represents a parameter of a page
58          @attribute type         the Type of the Parameter
59          @attribute name         the Name of the Parameter
60      */
61      Parameter =
62          Type Name;
63
64      /** StringContents represents a static String.
65          @attribute text         the String value
66      */
67      StringContents implements TextContents =
68          text:StringLiteral;
69
70      /** AttributeReferenceContents represents the value of an attribute
71          that is part of a passed parameter.
72          @attribute pageParameterName    the name of the passed parameter
73          @attribute attributeName        the name of the attribute of the
74                                          passed parameter
75      */
76      AttributeReferenceContents implements TextContents =
77           Output:["<"] pageParameterName:Name "." attributeName:Name;
78
79      /** SimpleAttributeReferenceContents represents the value of a passed
80          parameter that is one of the predefined ones.
81          @attribute pageParametername    the name of the passed parameter
82      */
83      SimpleAttributeReferenceContents implements TextContents =
84           Output:["<"] pageParametername:Name;
85
86      /** Head represents a heading element in a page.
87          @attribute textContents     List of TextContents elements that
88                                      form the contents of the heading
89      */
90      Head implements PageElement =
91        "head" "{" TextContents* "}";
92
93      /** Text represents a static text in a page.
94          @attribute  textContents    List of TextContents elements that
95                                      form the contents of the text
96      */
97      Text implements PageElement =
98        "text" "{" TextContents* "}";
99
100     /** Table represents a table in a page.
101         @attribute tableHead        optional header in a table
102         @attribute tableRows        List of rows in a table
103     */
104     Table implements PageElement =
105       "table"
106       "{"
107         TableHead?
108         tableRows:TableRow+
109       "}";
110
111     /** TableHead represents the head row of a table.
112         @attribute tableCells       List of TableCell elements that form
113                                     the columns of a header
114     */
115     TableHead =
116       "head"
117       "{"
118         tableCells:TableCell ("|" tableCells:TableCell )*
```

```
119       "}";
120
121     /** TableRow represents a contents row in a table.
122         @attribute tableCells       List of TableCell elements that form
123                                     the columns of the table contents
124     */
125     TableRow =
126       "row"
127       "{"
128         tableCells:TableCell ("|" tableCells:TableCell )*
129       "}";
130
131     /** TableCell represents the contents of a single cell in a table (header or row)
132         @attribute  langString      contents in case it is a reference
133                                     to an external String
134         @attribute  text            contents in case it is a static String
135         @attribute  attribureRef    contents in case it is a reference to a
136                                     passed parameter (or its attribute)
137     */
138     TableCell =
139       langString:QualifiedName | text:StringLiteral | attributeRef:AttributeReference;
140
141     /** AttributeReference represents a reference to a parameter or one of its attributes
142         @attribute  edit                indicator that the parameter should be editable
143         @attribute  output              indicator that the parameter should be non-editable
144         @attribute  infoOutput          indicator that the parameter should be non-editable
145                                         but allow a link to the linked object
146         @attribute  entityName          name of the referenced parameter
147         @attribute  attributeName       optional name of the referenced attribute (in case it
148                                         refers to a non-predefined parameter type)
149         @attribute  attributesToDisplay List of AttributeToDisplay elements that specify
150                                         which attributes should be display as link
151     */
152     AttributeReference =
153       (Edit:[">"] | Output:["<"] | InfoOutput:["<>"])
154       entityName:Name ("." attributeName:Name)?
155       ("("
156          attributesToDisplay:AttributeToDisplay (attributesToDisplay:AttributeToDisplay)*
157       ")")?;
158
159     /** AllAttributesShortcut represents a placeholder for all attributes of the given parameter.
160         @attribute  edit            indicator that all attributes should be editable
161         @attribute  output          indicator that all attributes should not be editable
162         @attribute  entityName      name of the passed parameter
163     */
164     AllAttributesShortcut implements PageElement =
165       "allAttributes"
166       "{"
167         (Edit:[">"] | Output:["<"]) entityName:Name
168       "}";
169
170     /** AttributeToDisplay represents an element to specify which attributes should
171         be display of a certain object.
172         @attribute  simpleAttribute         the name of the object in case it is a
173                                             simple attribute
174         @attribute  attributeName           the name of the object in case it is an object
175                                             of the domain model
176         @attribute  referencedAttributeName List of attribute names that should be displayed
177     */
178     AttributeToDisplay =
179         simpleAttribute:Name
180         |
```

```
181         attributeName:Name "(" referencedAttributeName:Name (referencedAttributeName:Name)* ")";
182
183     /** ListEntities represents an element that shows a list of all passed elements.
184         @attribute  edit                indicator that the list should contain links for
185                                         editing and deleting
186         @attribute  output              indicator that the list should only display the elements
187         @attribute  pageVariableName  the name of the passed parameter.
188                                         Should refer to a Collection.
189         @attribute  attributeToDisplay  List of the attributes that shall be displayed in the list
190     */
191     ListEntities implements PageElement =
192         "list"
193         "{"
194             (Edit:[">"] | Output:["<"]) pageVariableName:Name
195             ("("
196                 attributeToDisplay:AttributeToDisplay (attributeToDisplay:AttributeToDisplay)*
197             ")")?
198         "}";
199
200     /** ListEntitiesOnly represents an element that shows a list of all passed elements without
201         and further possibilities for interaction.
202         @attribute  pageVariableName    the name of the passed parameter.
203                                         Should refer to a Collection.
204         @attribute  attributesToDisplay List of the attributes that shall be displayed in the list
205     */
206     ListEntitiesOnly implements PageElement =
207         "listOnly"
208         "{"
209             pageVariableName:Name
210             ("("
211                 attributeToDisplay:AttributeToDisplay (attributeToDisplay:AttributeToDisplay)*
212              ")")?
213         "}";
214
215     /** ChooseEntities represents an element in the page that allows the selection of mulitple
216         elements from the list.
217         @attribute  pagevariableName    name of the passed parameter.
218                                         Should refer to a Collection.
219         @attribute  attributeToDisplay  List of the attributes to display in columns.
220     */
221     ChooseEntities implements PageElement =
222         "choose"
223         "{"
224             pageVariableName:Name
225             ("("
226                 attributeToDisplay:AttributeToDisplay (attributeToDisplay:AttributeToDisplay)*
227              ")")?
228         "}";
229
230     /** ChooseOneEntity represents an element in the page that allows the selection of exactly
231         one element from the list.
232         @attribute  pagevariableName    name of the passed parameter.
233                                         Should refer to a Collection.
234         @attribute  attributeToDisplay  List of the attributes to display in columns.
235     */
236     ChooseOneEntity implements PageElement =
237         "chooseOne"
238         "{"
239             pageVariableName:Name
240             ("("
241                 attributeToDisplay:AttributeToDisplay (attributeToDisplay:AttributeToDisplay)*
242              ")")?
```

```
243         "}";
244
245     /** HTMLElement represents an element in the page that can contain arbitrary HTML code plus
246         parts of the passed parameters.
247         @attribute  textContents    List of TextContents elements that form the content of the
248                                     HTMLElement.
249     */
250     HTMLElement implements PageElement =
251         "html" "{" TextContents* "}";
252 }
```

Abbildung C.6.: Grammatik der Seitenbeschreibungssprache Version 1.0

C.7. Aktivitätsdiagramme

MC

```
package de.montiwis.activities.main;

version "1.0";

/**
    Grammar
*/
grammar Activitydiagram extends mc.umlp.common.Common {

        options {
                        parser lookahead=3
                        lexer lookahead=7
                        compilationunit ADDefinition
        }

    /*=========================================================================*/
    /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
    /*=========================================================================*/

    external ADContent;

    interface ADElement;
    interface ADAssignment;

    /*=========================================================================*/
    /*============================== GRAMMAR ==================================*/
    /*=========================================================================*/

    /** ADDefintion represents the definition of an activity.
        @attribute  stereotype          optional stereotype
        @attribute  name                the name of the activity
        @attribute  aDInputParameters   optional input parameters
        @attribute  aDOutParameters     optional input parameters
        @attribute  aDElements          List of the contents of an activity
                                        (e.g., actions, transitions etc)
    */
    ADDefinition =
        Stereotype?
        "activity" name:Name
        "{"
            ADInputParameters?
            ADOutputParameters?
            (aDElements:ADElement)*
        "}";

    /*========================== ADELEMENTS ===================================*/

    /** ADAction represents an action.
        @attribute  stereotype              optional stereotype
        @attribute  name                    the name of the action
        @attribute  aDInputParameters       optional input parameter
        @attribute  aDOutParameters         optional input parameter
        @attribute  aDVariableDeclaration   optional declaration of variables within
                                            the action
        @attribute  aDCallStatement         a reference to an activity that specifies
                                            this actions' behavior
```

```
57          @attribute  aDActionContent        List of the contents of the action
58          @attribute  aDAssignment           List of the assignments of input- to outputparameters
59          @attribute  empty                  indicator that this action does not contain any
60                                             content (yet)
61      */
62      ADAction implements (Stereotype? "action")=>ADElement =
63          (Stereotype)?
64          "action" name:Name
65          (("{"
66              (("in" ":")=>ADInputParameters)?
67              (("out" ":")=>ADOutputParameters)?
68              (("var" ":")=>ADVariableDeclaration)?
69              (
70                  ("calls")=> ADCallStatement
71                  |
72                  (ADActionContent)*
73              )
74              (ADAssignment)*
75          "}")
76           |
77           (Empty:[";"])
78           );
79
80       /** ADRole represents the definition of a role / swimlane in an activity.
81          @attribute  stereotype      optional stereotype
82          @attribute  name            the name of the role
83          @attribute  actionNames     List of the names of actions that are executed
84                                      by this role
85       */
86       ADRole implements (Stereotype? "role")=>ADElement =
87          Stereotype?
88          "role" name:Name
89          "{"
90              actionNames:Name ("," actionNames:Name)* ";"
91          "}";
92
93      /** ADEdge represents an edge between nodes of arbitrary kinds.
94          @attribute  aDLeftEdgeSide  the nodes from which the edge starts
95          @attribute  aDRightEdgeSide the nodes to which the edge leads
96      */
97      ADEdge implements ADElement =
98          ADLeftEdgeSide "->" ADRightEdgeSide ";";
99
100     /** ADDecisionNode represents a decision node within an activity.
101         @attribute  stereotype          optional stereotype
102         @attribute  name                the name of the decision node
103         @attribute  aDBasicAssignment   List of assignments of input- to outputparameters
104     */
105     ADDecisionNode implements (Stereotype? "decision")=>ADElement =
106         Stereotype?
107         "decision" name:Name (
108         ";"
109         |
110         "{"
111             ADBasicAssignment*
112         "}");
113
114     /** ADMergeNode represents a merge node within the activity;
115         @attribute  Stereotype         optional stereotype
116         @attribute  name               name of the merge node
117         @attribute  aDBasicAssignment  List of assignments of input- to outputparameters
118     */
```

```
119     ADMergeNode implements (Stereotype? "merge")=>ADElement =
120         Stereotype?
121         "merge" name:Name (
122         ";"
123         |
124             "{"
125                 ADBasicAssignment*
126             "}");
127
128
129
130     /** ADJoinNode represents a joinnode
131         @attribute  stereotype         optional stereotype
132         @attribute  name               the name of the join node
133         @attribute  aDBasicAssignment  List of assignments of input- to outputparameters
134     */
135     ADJoinNode implements (Stereotype? "join")=>ADElement =
136        Stereotype?
137        "join" name:Name (
138        ";"
139        |
140        "{"
141             ADBasicAssignment*
142        "}");
143
144     /** ADForkNode represents a forknode
145         @attribute  Stereotype         optional stereotype
146         @attribute  name               the name of the fork node
147         @attribute  aDBasicAssignment  List of assignments of input- to outputparameters
148     */
149     ADForkNode implements (Stereotype? "fork")=>ADElement =
150        Stereotype?
151        "fork" name:Name (
152        ";"
153        |
154        "{"
155             ADBasicAssignment*
156        "}");
157
158     /*============================ EDGE ELEMENTS ================================*/
159
160     /** ADLeftEdgeSide represents the left hand side of an edge (the nodes from which
161         the edge originates).
162         @attribute  initial         an initial node
163         @attribute  aDJoinEdgeSide  a join construct
164         @attribute  aDMergeEdgeSide a merge construct
165         @attribute  aDNode          a node
166     */
167     ADLeftEdgeSide =
168         Initial:["initial"]
169         |
170         (Name ("." Name)? "&")=> ADJoinEdgeSide
171         |
172         (Name ("." Name)? "|")=> ADMergeEdgeSide
173         |
174         ADNode
175         ;
176
177     /** ADRightEdgeSide represents the right hand side of an edge (the nodes that
178         the transition points to).
179         @attribute  final                          a final node
180         @attribute  flowFinal                      a flowfinal node
```

```
181         @attribute  aDForkEdgeSide                  a fork construct
182         @attribute  aDSplitEdgeSide                 a split construct
183         @attribute  aDConditiionalSplitEdgeSide     a conditional split construct
184         @attribute  aDNode                          a node
185      */
186     ADRightEdgeSide =
187         Final:["final"]
188         |
189         FlowFinal:["flowfinal"]
190         |
191         (Name ("." Name)? "&" )=> ADForkEdgeSide
192         |
193         (((Name ("." Name)?) | "final" | "flowfinal") "|")=> ADSplitEdgeSide
194         |
195         ((Name ":")? "[" )=> ADConditionalSplitEdgeSide
196         |
197         ADNode
198         ;
199
200     /** ADForkEdgeSide represents a fork side on the right hand side of an edge.
201         @attribute forkNodes    List of the nodes to which the edge is being
202                                 forked to
203     */
204     ADForkEdgeSide =
205         forkNodes:ADNode ("&" forkNodes:ADNode)+;
206
207    /** ADJoinEdgeSide represents a join side on the left hand side of an edge.
208         @attribute joinNodes    List of the nodes from which the transition is being
209                                 joined
210     */
211     ADJoinEdgeSide =
212         joinNodes:ADNode ("&" joinNodes:ADNode)+;
213
214     /** ADSplitEdgeSide represents a split construct.
215         @attribute terms    List of ADSplitSideTerms
216     */
217     ADSplitEdgeSide =
218        terms:ADSplitSideTerm ("|" terms:ADSplitSideTerm)+;
219
220     /** ADSplitSideTerm represents the alternatives that can form a split construct.
221         @attribute aDNode       an ADNode
222         @attribute final        a Final node
223         @attribute flowFinal    a FlowFinal node
224     */
225     ADSplitSideTerm =
226         (ADNode | Final:["final"] | FlowFinal:["flowfinal"]);
227
228     /** ADMergeEdgeSide represents a merge construct.
229         @attribute mergeNodes   List of nodes that are part of the split construct
230     */
231     ADMergeEdgeSide =
232         mergeNodes:ADNode ("|" mergeNodes:ADNode)+;
233
234     /** ADConditionalSplitEdgeSide represents a split with a condition
235         under which a branch is chosen
236         @attribute terms    List of ADConditionalsplitTerms
237     */
238     ADConditionalSplitEdgeSide =
239         terms:ADConditionalSplitTerm ("|" terms:ADConditionalSplitTerm)* ;
240
241     /** ADConditionalSplitTerm represents a term consisting of a condition and a node that
242             is chosen when the condition is matched
```

```
243
244         @attribute  kind                indicates which kind of InvariantContent follows
245                                         (could be Java, OCL, ...)
246         @attribute  invariantContent    the content of this invariant
247         @attribute  aDSplitSideTerm  a ADSplitSideTerm
248     */
249     ADConditionalSplitTerm =
250          (kind:Name ":")?
251         "[" InvariantContent(parameter kind) "]"
252          ADSplitSideTerm;
253
254     /*========================== OTHER ELEMENTS ============================*/
255
256     /** ADNode represents a node
257         @attribute  name     the name of the node
258         @attribute  pinName  the name of the input / output pin on the node
259     */
260     ADNode = name:Name ("." pinName:Name)? ;
261
262     /** ADBasicAssignment represents assignment of input- to outputparameter
263         @attribute  inputParam   name of the input parameter
264         @attribute  outputParam  name of the output parameter
265     */
266     ADBasicAssignment =
267          outputParam:Name "=" inputParam:Name ";";
268
269     /** ADHierarchicalInputAssignment represents the assignment of inputparameter to
270         input parameter of the child activity.
271         @attribute  inputParam          the name of the input parameter
272         @attribute  childActivityName   the name of the child activity
273         @attribute  parameterName       the name of the input parameter of the child activity
274     */
275     ADHierarchicalInputAssignment implements (Name "->" Name "." Name ";")=>ADAssignment =
276          (inputParam:Name "->" childActivityName:Name "." parameterName:Name) ";";
277
278
279     /** ADHierarchicalOutputAssignment represents the assignment of an outputarameter
280         of the child activity to an output parameter of this activity.
281         @attribute  childActivityName   the name of the child activity
282         @attribute  parameterName       the name of the output parameter of the child activity
283         @attribute  outputParam         the name of the output parameter
284     */
285     ADHierarchicalOutputAssignment implements (Name "." Name "->" Name ";")=>ADAssignment =
286          (childActivityName:Name "." parameterName:Name "->" outputParam:Name) ";";
287
288     /** ADVariableDeclaration represents a variable declaration within a node.
289         @attribute  params  List of the parameters that declare the variables
290                             to be used within the node
291     */
292     ADVariableDeclaration  =
293         "var" ":" params:ADParameter ("," params:ADParameter)* ";";
294
295     /** ADInputParameters represents the input parameters.
296         @attribute  params   List of parameters
297     */
298     ADInputParameters  =
299         "in" ":" params:ADParameter ("," params:ADParameter)* ";";
300
301     /** ADOutputParameters represents the output parameters.
302         @attribute  params   List of parameters
303     */
304     ADOutputParameters  =
```

```
305         "out" ":" params:ADParameter ("," params:ADParameter)* ";";
306
307     /** ADParameter represents a parameter.
308         @attribute  type     a Type that represents the type of the parameter
309         @attribute  name     the name of the parameter
310     */
311     ADParameter =
312         type:Type name:Name;
313
314     /** ADCallStatement represents a call behaviour statement; used to reference another activity
315         that is a child of this action or target of the signal
316         @attribute  calleddActivity the name of the called activity
317     */
318     ADCallStatement =
319        "calls" calledActivity:Type ";";
320
321     /** ADActionContent represents the content of an action.
322         @attribute  kind        the kind of the content
323         @attribute  aDContent   the content of the action
324     */
325     ADActionContent =
326         kind:Name ":" ADContent(parameter kind);
327 }
```

Abbildung C.7.: Grammatik der Aktivitätsdiagramme Version 1.0

MC

```
1  package de.montiwis.activities.contents;
2
3  version "1.0";
4
5  /**
6    * Grammar for the ViewContents that represents the contents of an action in case
7    * a page is called.
8    *
9    *  @author Dirk Reiß
10   */
11 grammar ViewContent extends mc.umlp.common.Common {
12
13     options {
14         parser lookahead=3
15         lexer lookahead=7
16     }
17
18     /** View represents a call to a view.
19         @attribute pageName         the name of the page to call
20         @attribute params   List of parameters passed to the called page
21     */
22     View  =
23         (pageName:Name "(" params:Name? ("," params:Name)* ")" ";")
24         ;
25
26 }
```

Abbildung C.8.: Grammatik des Aktionsinhalts für Seitenaufrufe Version 1.0

MC

```
1  package de.montiwis.activities.contents;
2
3  version "1.0";
4
5
6  /**
7   * Grammar for the action content in case commands are used.
8   *
9   * @author: Dirk Reiß
10  */
11 grammar CommandContent {
12
13         options {
14             parser lookahead = 3
15             lexer lookahead = 7
16         }
17
18     /** Command represents a command. Delegates to the commands themselves.
19         @attribute  findCommand             a FindCommand
20         @attribute  saveCommand             a SaveCommand
21         @attribute  assignRolesCommand      an AssignRolesCommand
22         @attribute  loadAllCommand          a LoadAllCommand
23         @attribute  getActualUserCommand    a GetAllUserCommand
24         @attribute  saveAllCommand          a SaveAllCommand
```

```
25          @attribute  deleteCommand           a DeleteCommand
26          @attribute  setErrorsCommand        a SetErrorsCommand
27      */
28      Command =
29          ((Name "=")? Name "." "find")=>FindCommand
30          |
31          SaveCommand
32          |
33          ((Name "=")? Name "." "load")=>LoadCommand
34          |
35          AssignRolesCommand
36          |
37          ((Name "=")? Name "." "loadAll")=>LoadAllCommand
38          |
39          (Name "=" "getActualUser")=>GetActualUserCommand
40          |
41          SaveAllCommand
42          |
43          DeleteCommand
44          |
45          SetErrorsCommand
46          ;
47
48      /** FindCommand represents command to search for one or more entities.
49          @attribute  variableName    the variable that the result of the search shall be assigned to
50          @attribute  entityName      the entity (classname) that is to be searched.
51          @attribute  attributeName   the name of the attribute of that class that will be searched for
52          @attribute  value           the value that the objects should match when searched for.
53      */
54      FindCommand  =
55          (variableName:Name "=")?  entityName:Name "." "find" "(" attributeName:Name "," value:Name ")↩
                " ";";
56
57      /** SaveCommand represents a Command to save an entity.
58          @attribute  entityName      the name of the entity that will be saved.
59          @attribute  attributeName   the name of the variable that will be saved
60      */
61      SaveCommand  =
62          entityName:Name "." "save" "(" attributeName:Name ")" ";";
63
64      /** SaveAllCommand represents a Command to save a collection of entities;
65          @attribute  entityName      the name of the entity that will be saved.
66          @attribute  attributeName   the name of the variable that will be saved.
67                                      It should be of type Collection<Entity>
68                                      with Entity being the entityName.
69      */
70      SaveAllCommand =
71          entityName:Name "." "saveAll" "(" attributeName:Name ")" ";";
72
73      /** LoadCommand represents a Command to load an entity by id.
74          @attribute  variableName    the name of the variable that the loaded entity
75                                      will be assigned to
76          @attribute  entityName      the name of the entity that shall be loaded.
77          @attribute  id              the name of the variable that holds the id
78      */
79      LoadCommand  =
80          (variableName:Name "=")?  entityName:Name "." "load" "(" id:Name ")" ";";
81
82      /** LoadAllCommand represents a Command to load all entities of a certain kind.
83          @attribute  variableName    the name of the variable that the loaded entity
84                                      will be assigned to (should be a Collection)
85          @attribute  entityName      the name of the entitie that shall be loaded.
```

```
86      */
87      LoadAllCommand =
88          (variableName:Name "=")?  entityName:Name "." "loadAll" "(" ")" ";";
89
90      /** AssignRolesCommand represents a Command to assign certain users to a certain role.
91          @attribute  roleName        the name of the role to which users shall be assigned to
92          @attributee variableName    the name of a variable that contains the user objects
93                                      (should be a Collection)
94      */
95      AssignRolesCommand =
96          "assignRole" "(" roleName:Name "," variableName:Name ")" ";";
97
98      /** GetActualUserCommand represents a Command that retrieves the actually acting user from the ↩
             system.
99          @attribute  variableName    the variable that the retrieved user should be assigned to.
100     */
101     GetActualUserCommand =
102         variableName:Name "=" "getActualUser" "(" ")" ";";
103
104     /** DeleteCommand represents a Command to delete a certain object.
105         @attribute  entityName      the name of the entity that the object to delete belongs to.
106         @attribute  variableName    the name of the variable that contains the object
107                                     that shall be deleted
108     */
109     DeleteCommand =
110         entityName:Name "." "delete" "(" variableName:Name ")" ";";
111
112     /** SetErrorsCommand represents a Command to set error messages in the system.
113         @attribute  variableName    the name of the variable that contains the error messages.
114                                     Should be a List<String>.
115     */
116     SetErrorsCommand =
117         "setErrors" "(" variableName:Name ")" ";";
118 }
```

Abbildung C.9.: Grammatik des Aktionsinhalts für Kommandos Version 1.0

MC

```
1  package de.montiwis.activities.contents;
2
3  version "1.0";
4
5  /**
6   * Grammar for the Java content in a MontiWIS action.
7   *
8   * @author Dirk Reiß
9   */
10 grammar JavaContent extends mc.umlp.common.Common {
11
12     options {
13         parser lookahead=3
14         lexer lookahead=7
15     }
16
17     external BlockStatement;
18
19     /** JavaContents represents the actual Java contents of an action.
20         @attribute  blockstatement  a Java block statement
21     */
22     Java  =
23         blockStatement:BlockStatement;
24 }
```

Abbildung C.10.: Grammatik des Aktionsinhalts für Java Code Version 1.0

C.8. Applikationsprache

MC

```
1  package de.montiwis.application;
2
3  version "1.0";
4
5
6  /**
7   * Grammar for the MontiWIS application language.
8   *
9   * @author Dirk Reiß
10  */
11 grammar Application extends mc.umlp.common.Common {
12
13     options {
14         parser lookahead=3
15         lexer lookahead=7
16         compilationunit ApplicationDefinition
17     }
18
19     /*=========================================================================*/
20     /*================== INTERFACES AND EXTERNAL SYMBOLS ======================*/
21     /*=========================================================================*/
22
23     interface ApplicationElement;
24
25     /*=========================================================================*/
26     /*============================== GRAMMAR ==================================*/
27     /*=========================================================================*/
28
29     /** ApplicationDefinition represents the root element of the applicationDSL
30         @attribute  name               the name of the applicationDSL
31         @attribute  applicationElement  placeholder for every other content of
32                                        the applicationDSL
33     */
34     ApplicationDefinition =
35         "application" name:Name
36         "{"
37             ApplicationElement*
38         "}";
39
40     /*============================== Menu =====================================*/
41
42     /** Menu represents the definition of the menu of the application.
43         Each menu element represents one menu that is then referenced throughout
44         the application by its name.
45         @attribute  name     the name of the menu. It identifies this menu throughout
46                              the application (in templates for example)
47         @attribute menuEntry a link to an activity or a label within a menu
48     */
49     Menu implements ApplicationElement =
50         "menu" name:Name
51         "{"
52             MenuEntry+
53          "}";
54
55     /** MenuEntry represents either a link to an activity or a label within a menu.
56         @attribute  labelEntry                      a label within a menu
```

```
57          @attribute  activityOrPageReferenceEntry    a reference to an activity or page
58          @attribute  implicitViewEntry               a reference to a standard page (CRUD page)
59      */
60      MenuEntry =
61          (LevelBullet (StringLiteral | QualifiedName) ";")=>LabelEntry |
62          (LevelBullet (StringLiteral | QualifiedName) ":" QualifiedName ";")=>↩
                ActivityOrPageReferenceEntry |
63          (LevelBullet (StringLiteral | QualifiedName) ":" QualifiedName "." Name "()" ";")=>↩
                ImplicitViewEntry;
64
65      /** LabelEntry represents an entry in the menu that just acts as a label
66          (i.e., neither an activity nor a view is referenced).
67          @attribute  levelBullet     indicates which level this label is to be placed at
68          @attribute  label           the label that shall be used. Can either be a label itself
69          @attribute  labelRef        a reference to an entry in a message.properties file
70      */
71      LabelEntry =
72          LevelBullet (label:StringLiteral | labelRef:QualifiedName) ";";
73
74      /** ActivityOrPageReferenceEntry represents an entry that references either an activity or a view↩
                .
75          @attribute  levelBullet                 indicates which level this reference is to be placed ↩
                    at
76          @attribute  label                       the label of this entry
77          @attribute  labelRef                    a reference to an externalized string
78          @attribute  referencedActivityOrPage    the fullqualified name of the referenced activity or ↩
                    view
79      */
80      ActivityOrPageReferenceEntry =
81          LevelBullet (label:StringLiteral | labelRef:QualifiedName) ":" referencedActivityOrPage:↩
                QualifiedName ";";
82
83     /** ImplicitViewEntry represents a reference to a standard (CRUD) page.
84          @attribute  levelBullet                 indicates which level this reference is to be placed ↩
                    at
85          @attribute  label                       the label of this entry
86          @attribute  labelRef                    a reference to an externalized string
87          @attribute  referencedClass             the name of the entity that is referenced
88          @attribute  viewName                    the name of the view to call (e.g., new, list)
89      */
90      ImplicitViewEntry =
91          LevelBullet (label:StringLiteral | labelRef:QualifiedName) ":" referencedClass:QualifiedName ↩
                "." viewName:Name "()" ";";
92
93      /** LevelBullet represents the bullets (analogous to Wiki syntax) of the hierarchical
94          tree-like structure of the menu.
95          @attribute  levelOne    the first level in the menu
96          @attribute  levelTwo    the second level in the menu
97          @attribute  levelThree  the third level in the menu
98      */
99      LevelBullet =
100         LevelOne:["*"] | LevelTwo:["**"] | LevelThree:["***"] ;
101
102     /*============================== Rightdef ==================================*/
103
104     /** Right represents the applicationElement that declares the priviledges a certain right has.
105         It can either contain role assignments or not.
106         @attribute  rightNames      List of rights for whom the declared priviledges apply
107         @attribute  runnable    List of activities (or Views resp.) that this right has priviledges ↩
                to run
108     */
109     Right implements ApplicationElement =
```

```
110         "right" rightNames:Name ("," rightNames:Name)*
111         "{"
112             (
113                 "runs" Runnable ("," Runnable)* ";"
114             )
115         "}";
116 
117 
118     /** Runnable represents either a reference to a standard (CRUD) page or an activity or page.
119         @attribute  implicitView        a reference to a standard page
120         @attribute  activityOrPageName  a reference to an activity or page
121     */
122     Runnable =
123         (QualifiedName "." Name "()")=>ImplicitView
124         |
125         (QualifiedName)=>ActivityOrPageName ;
126 
127     /** ActivityOrPageName represents a reference to an activity or page.
128         @attribute  activityOrPageName  the name of the activty or page
129     */
130     ActivityOrPageName =
131         activityOrPageName:QualifiedName;
132 
133     /** ImplicitView represents a link to a standard page.
134         @attribute  referencedClass the name of the entity
135         @attribute  viewName        the kind of view
136     */
137     ImplicitView =
138         referencedClass:QualifiedName "." viewName:Name "()";
139 }
```

Abbildung C.11.: Grammatik der Applikationssprache Version 1.0

Anhang D.

Lebenslauf

Name	Reiß
Vorname	Dirk
Geburtstag	05.07.1977
Geburtsort	Braunschweig
Staatsangehörigkeit	Deutsch

seit 2011	Wissenschaftlicher Mitarbeiter am Institut für Gebäude- und Solartechnik der Technischen Universität Braunschweig
2009 - 2011	Wissenschaftlicher Mitarbeiter am Institut für Wirtschaftsinformatik der Technischen Universität Braunschweig
2005 - 2009	Wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering der Technischen Universität Braunschweig
2005	Abschluss als Diplom-Wirtschaftsinformatiker
1998 - 2005	Studium der Wirtschaftsinformatik an der Technischen Universität Braunschweig
1997 - 1998	Zivildienst in Salzgitter
1990 - 1997	Gymnasium in Braunschweig
1988 - 1990	Orientierungsstufe in Vechelde
1984 - 1988	Grundschule in Vallstedt

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 [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] 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] 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]. 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)

Many of our contributions build on UML/P, which is described in the two books [Rum11] 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]. 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] 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] allows the spe-

cification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, 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.

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] 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 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].

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]. [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 [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] describes 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]. 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] 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] 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. 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 [HHK+14], 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. TU Munich, Germany, 1997.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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*. Springer Berlin, 2te Edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Berlin, 2te Edition, Juni 2012.

[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.