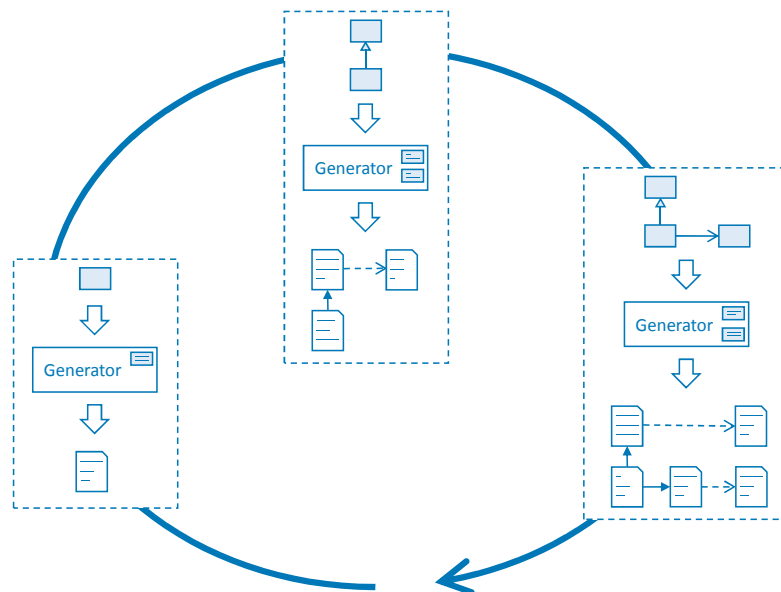


Klaus Müller

Modellbasierte Unterstützung der Software Evolution im industriellen Kontext





[Mul18] K. Müller:

Modellbasierte Unterstützung der Software Evolution im industriellen Kontext.

Shaker Verlag, ISBN 978-3-8440-6034-8. Aachener Informatik-Berichte, Software Engineering, Band 35. July 2018.

www.se-rwth.de/publications/

Modellbasierte Unterstützung der Software Evolution im industriellen Kontext

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

M.Sc.

Klaus Müller

aus Bernkastel-Wittlich

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Udo Kelter

Tag der mündlichen Prüfung: 27. Juli 2017

Kurzfassung

Software Evolution ist ein integraler Bestandteil der Softwareentwicklung, da jedes produktiv eingesetzte Softwaresystem kontinuierlich angepasst werden muss, damit es seinen Nutzen nicht verliert [Leh79]. Die Folge davon ist, dass die Komplexität der Softwaresysteme im Laufe der Zeit stetig ansteigt. Eine Möglichkeit um mit dieser steigenden Komplexität umzugehen besteht in dem Einsatz von Modellierungs- und Generierungstechniken [SVEH07, Sch12]. Diese führen allerdings zu weiteren Herausforderungen hinsichtlich der Software Evolution, da die eingesetzten Modelle und Generatoren ebenfalls kontinuierlich angepasst werden müssen. Eine besondere Herausforderung in diesem Kontext ist, dass jede Änderung eines Modells weitreichende Auswirkungen auf ein Softwaresystem haben kann, sobald auf Basis des Modells Generierungsaktivitäten durchgeführt werden. Eine wichtige Voraussetzung zur Unterstützung der Software Evolution besteht deshalb in der Ermittlung der Modelländerungen, die im Laufe der Zeit durchgeführt wurden.

Diese Arbeit zielt auf die modellbasierte Unterstützung der Software Evolution im industriellen Kontext ab. Der Fokus liegt auf Softwaresystemen, die generativ entwickelt und in denen Klassendiagramme zur Modellierung eingesetzt werden. Die wichtigsten wissenschaftlichen Beiträge lassen sich wie folgt zusammenfassen:

- Bei der automatischen Ermittlung von Modelländerungen kann im Allgemeinen nicht garantiert werden, dass stets die korrekten Modelländerungen erkannt werden. Zur Unterstützung der Modellierer wird in dieser Arbeit eine domänenspezifische Sprache (engl. domain specific language, DSL) bereitgestellt, die es Modellierern ermöglicht punktuell festzulegen, welche Modelländerungen für ausgewählte Modellelemente durchgeführt wurden. Diese Vorgaben werden in die automatische Differenzberechnung einbezogen, so dass Fehler in der Differenzberechnung vermieden werden können.
- Jede Modelländerung kann weitreichende Auswirkungen auf ein Softwaresystem haben. Aus diesem Grund wird in dieser Arbeit eine DSL bereitgestellt, über die es möglich ist, die Auswirkungen von Modelländerungen in Klassendiagrammen zu modellieren. Nachdem die durchgeführten Modelländerungen identifiziert wurden, wird auf Basis der modellierten Auswirkungen eine Checkliste generiert. Diese legt dar, welche Entwicklungsaktivitäten aufgrund der erkannten Modelländerungen durchgeführt werden müssen.
- Werden Datenmodelle zur Modellierung der Struktur der Daten eines Softwaresystems eingesetzt und generativ verarbeitet, können Änderungen der Datenmodelle die Durchführung von Datenmigrationen erfordern. Auf welche Weise diese Datenmigrationen vorgenommen werden müssen, richtet sich nach der Art der Modelländerungen. In dieser Arbeit werden die notwendigen Schritte zur Umsetzung von Datenmigrationen für verschiedene Arten von Modelländerungen dargelegt. Darüber hinaus wird eine Vorgehensweise zur Generierung von Migrationsklassen vorgestellt, welche diese Schritte kapselt.
- Zur Implementierung eines Softwaresystems oder einer Variante eines Softwaresystems kann es hilfreich sein neue Varianten bestehender Generatoren zu erzeugen. Zur Entwicklung solcher Varianten werden in dieser Arbeit mehrere DSLs bereitgestellt. Über diese ist

es möglich eine neue Generatorvariante zu definieren, indem modelliert wird, auf welche Weise der Inhalt bestehender Generatorartefakte für die neue Variante angepasst werden soll. Als Ergebnis werden auf Basis der modellierten Operationen automatisch neue Generatorartefakte erzeugt, welche genau die modellierten Änderungen abbilden.

- Die Restriktionen, die festlegen, welche Komponenten eines Systems mit welchen anderen Komponenten interagieren dürfen, sind ein wichtiger Bestandteil der Softwarearchitektur eines Systems. In dieser Arbeit wird die Modellierung solcher Abhängigkeitsrestriktionen für Plugin-basierte Softwaresysteme durch eine DSL ermöglicht. Diese Restriktionen drücken erlaubte und verbotene Abhängigkeiten zwischen Plugins des Softwaresystems aus. Zudem wurde eine Prüfkomponeente entwickelt, welche die Implementierung eines Softwaresystems kontinuierlich gegen die modellierten Restriktionen prüft.

Die genannten Teilgebiete der Software Evolution werden an einem als Herzstück eines Produkts im Diagnose-Bereich für Fahrzeuge eingesetzten Softwaresystem evaluiert. Die entwickelten Konzepte sind jedoch unabhängig von diesem Softwaresystem und lassen sich auf andere generativ entwickelte Softwaresysteme übertragen.

Abstract

Software evolution is an essential part of software development since every software system that is productively used needs to be adapted continuously or it becomes less useful over time [Leh79]. One possibility to cope with the increasing complexity arising from this is to use modeling and generation techniques [SVEH07, Sch12]. However, such techniques lead to further challenges when it comes to software evolution. One reason for this is that the used models and generators need to be adapted continuously too. One particular challenge in this context is that every change of a model can have severe impacts on the whole software system as soon as generation activities are performed based on the model. Thus, an important prerequisite to facilitate software evolution is to determine the model changes which are performed in the course of the time.

This work is targeted on supporting software evolution in an industrial context in a model-based way. In particular, the focus is on software systems that are being developed in a generative way and that use class diagrams for modeling purposes. The most important scientific contributions are summarized in the following:

- In general, an automatic approach to identify the performed model changes cannot guarantee that the reported model changes are always correct. To support the modelers in this context, a domain specific language (DSL) is provided. This DSL allows modelers to define for selected model elements which model changes have been performed. This knowledge is taken into account during the automatic identification of the model changes. In this way, errors in the automated identification of the model changes can be avoided.
- Every model change can have severe impacts on a software system. For this reason, a DSL is provided in this thesis that allows for modeling consequences of model changes in class diagrams. After having identified the performed model changes, a checklist is generated based on the modeled consequences. This checklist indicates which development activities need to be carried out due to the model changes.
- In case data models are used for modeling the structure of data in a software system, and these models are processed by a code generator, changes of data models can necessitate performing data migrations. In which way these data migrations need to be carried out depends on the types of model changes. In this thesis the steps that are necessary to perform data migrations for various types of model changes are explained. Moreover, a procedure to generate migration classes encapsulating these steps is presented.
- In order to implement a software system or a variant of a software system, it can be helpful to develop new variants of existing generators. For developing such variants multiple DSLs are provided in this thesis. These allow for defining a generator variant by modeling in which way the content of existing generator artifacts is to be adapted in the new variant. As a result, new generator artifacts are created automatically based on the modeled operations. These newly created artifacts conform to the modeled operations.

- The restrictions that indicate which component of a software system may interact with which other components are an important part of the software architecture of a system. To support software developers in developing a software system that conforms to such restrictions, a DSL is provided that makes it possible to model dependency restrictions for plugin-based software systems. These restrictions express allowed and forbidden dependencies between distinct plugins of the software system. Moreover, a checking component was developed to validate the implementation of a software system against the modeled restrictions continuously.

The stated areas of software evolution are evaluated by applying them to a software system that is at the core of a product in the area of vehicle diagnostics. However, the developed concepts are independent from this software system and can be transferred to other software systems that are developed in a model-driven way.

Danksagung

An dieser Stelle möchte ich mich bei all den Menschen bedanken, die mich während meiner Promotion unterstützt und so zum Erfolg dieser Dissertation beigetragen haben.

Mein ganz besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Betreuung meiner Promotion. Die interessanten und konstruktiven Diskussionen mit dir sowie deine wertvollen Ratschläge haben maßgeblich zu dem Erfolg dieser Arbeit beigetragen. Zudem möchte ich mich dafür bedanken, dass ich die Möglichkeit hatte in einem interessanten Industrieprojekt bei der DSA Daten- und Systemtechnik GmbH zu arbeiten und die entwickelten Ideen dort anzuwenden.

Darüber hinaus danke ich Prof. Dr. Udo Kelter für die Bereitschaft zur Übernahme des Zweitgutachtens. Ebenso möchte ich mich bei Prof. Dr. Ulrik Schroeder für die Abnahme der Prüfung in der praktischen Informatik und bei Prof. Dr. Erika Ábrahám für die Abnahme der Prüfung in der theoretischen Informatik sowie die Leitung der Prüfungskommission bedanken.

Weiterhin gilt mein Dank der Geschäftsführung der DSA Daten- und Systemtechnik GmbH und hier insbesondere Dr. Ansgar Schleicher. Gemeinsam mit Bernhard hast du meine Promotion erst möglich gemacht. Bedanken möchte ich mich auch für die Möglichkeit die Dissertation parallel zur Arbeit bei DSA fertigzustellen.

Großer Dank gilt auch allen Kollegen am Lehrstuhl. Die vielfältigen gemeinsamen Freizeitaktivitäten haben mir immer viel Freude bereitet und waren eine gelungene Abwechslung in so manch stressigen Zeiten. Besonders bedanken möchte ich mich bei Dr. Arne Haber, Dr. Andreas Wortmann und Dr. Markus Look. Ihr habt mir nicht nur wichtiges Feedback im Laufe der Promotion gegeben, sondern auch dafür gesorgt, dass ich mich schnell in Aachen heimisch gefühlt habe. Ebenso möchte ich an dieser Stelle Christoph Schulze danken. Auch wenn wir beide durch die Industrieprojekte nicht so häufig am Lehrstuhl waren, warst du stets ein sehr angenehmer Bürokollege. Die gemeinsamen Poker Matches gegen dich bzw. mit dir werde ich vermissen. Weiterhin möchte ich Dr. Pedram Mir Seyed Nazari danken. Du hast mich zwar leider zu häufig im Tischtennis besiegt, aber das kann ich dir verzeihen. Danke für deine Hilfe bei der Prüfungsvorbereitung und für die vielen netten Gespräche. Weiterhin gilt mein besonderer Dank Dr. Alexander Roth. Wir haben so einige Stunden beim gemeinsamen Schreiben von Papern verbracht. Du hast nicht nur stets gutes Feedback gegeben, auf dich konnte man sich auch immer verlassen und du hattest auch immer ein offenes Ohr für sonstige Gespräche. Die Konferenzreise nach Angers mit dir und Pedram werde ich stets in guter Erinnerung behalten. Natürlich möchte ich mich auch bei allen bedanken, die ein Review von Teilen meiner Arbeit durchgeführt haben: Dr. Andreas Wortmann, Dr. Arne Haber, Robert Heim, Achim Lindt, Dr. Pedram Mir Seyed Nazari, Dr. Markus Look, Dr. Alexander Roth sowie Timo Greifenberg. Sollte ich jemanden vergessen haben, tut es mir leid.

Weiterhin möchte ich mich herzlich bei meinen Kollegen von der DSA bedanken. Mein Dank gilt unter anderem Christoph Briem, David Scheuren, Detlef Wilkening sowie Oliver Küsters. Die Diskussionen mit euch und euer Feedback haben mir nicht nur den Einstieg in die Arbeit bei DSA erleichtert, sondern auch zum Gelingen dieser Arbeit beigetragen. Weiterhin gilt mein Dank Gregor Skibka und Jochen Wollgarten. Durch euch habe ich einen schnellen Einstieg in die Modellierung und Generierung bei DSA erhalten. Die gemeinsame Zeit mit euch im Büro

war zudem sehr unterhaltsam. Hervorheben möchte ich zudem auch noch Jens Carstensen. Wir haben echt viele lustige Dinge erlebt und viele lustige Unterhaltungen geführt - auch wenn es für unsere Kollegen glaube ich nicht immer so lustig war. Darüber hinaus gilt mein großer Dank Sven Ewald. Gemeinsam mit Jens und dir habe ich meine Leidenschaft für das Bouldern entdeckt. An die vielen lustigen Unterhaltungen und Events mit dir denke ich immer sehr gerne zurück.

Ebenso möchte ich an dieser Stelle Prof. Dr. Rainer Oechsle von der Hochschule Trier danken. Du hast mir während meines Studiums in Trier früh die Möglichkeit gegeben erste Praxiserfahrungen in einem interessanten Projekt zu sammeln. Auch in den darauffolgenden Jahren hast du mir die Möglichkeit gegeben mich in verschiedensten Bereichen zu entfalten. Diese Erfahrungen haben mit dazu beigetragen, dass ich mich für eine Promotion entschieden habe.

Abschließend möchte ich ein besonders großes Danke an meine Eltern sowie meine Geschwister richten. Ihr wart immer für mich da und habt mich auf meinem gesamten Lebensweg stets unterstützt. Dadurch habt ihr es mir überhaupt erst möglich gemacht diesen Weg zu gehen. Zudem habt ihr immer Verständnis gezeigt, wenn ich durch das Studium bzw. die Promotion mal weniger Zeit hatte, was leider häufig vorkam. Ebenso möchte ich meiner Patentante Inge Sliwka herzlich danken. Du hast früh mein Interesse für Informatik geweckt und dieses Interesse sowie mich im Allgemeinen gefördert. Zu guter letzt kann ich meiner Freundin Maria nicht genug danken. Du musstest wohl am meisten unter meiner Promotion leiden. Dessen ungeachtet hast du mich immer liebevoll unterstützt und motiviert. Du warst zudem immer verständnisvoll, wenn ich abends oder am Wochenende an der Dissertation oder an Papern gesessen habe. Ich weiß, dass das nicht selbstverständlich ist. Ohne deinen großen Rückhalt und deine Unterstützung wäre diese Arbeit nicht möglich gewesen.

Aachen, Juni 2018
Klaus Müller

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation und Kontext der Arbeit	2
1.2	Wissenschaftliche Fragestellung	5
1.3	Aufbau der Arbeit	8
2	Kontext der Arbeit und Methodik	9
2.1	Systemlandschaft bei DSA	9
2.2	Ermittlung von Modellunterschieden	11
2.3	Auswirkungsanalyse zur Generierung von Checklisten	13
2.4	Generierung von Migrationsfunktionalität zur Daten Evolution	15
2.5	Modellierung von Generatorvariabilität	19
2.6	Modellbasierte Prüfung der Architekturkonformität in Plugin-basierten Software-systemen	20
3	Grundlagen der DSL-Entwicklung	23
4	Ermittlung von Modellunterschieden	27
4.1	Grundlagen und Stand der Technik	27
4.2	Methodik zur Bestimmung von Modellunterschieden	31
4.2.1	Konvertierung von Eingabemodellen	33
4.2.2	Durchführung der Matchingphase	37
4.2.3	Integration von Delta-Benutzervorgaben in die Matchingphase	43
4.2.4	Durchführung der Differenzierungsphase	52
5	Auswirkungsanalyse zur Generierung von Checklisten	55
5.1	Checkliste	56
5.2	Beispiel	57
5.2.1	Auswirkungsregel zur Analyse von objektrelationalen Mapping-Dateien (ORM-AR)	57
5.2.2	Checkliste zur Auswirkungsregel ORM-AR	59
5.3	Methodik zur Generierung von Checklisten	61
5.4	Ansatz zur regelbasierten Auswirkungsanalyse	64
5.4.1	Spezifikation von Auswirkungsregeln	64
5.4.2	Generierung und Erweiterung von Auswirkungsregeln	70
5.4.3	Beispiele für die Modellierung von Auswirkungsregeln	72

5.5	Diskussion	88
5.5.1	Allgemeine Diskussion	88
5.5.2	Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen	89
5.6	Verwandte Arbeiten	91
5.7	Fallstudie in der Industrie	93
5.7.1	Ziele und Forschungsfragen	93
5.7.2	Aufbau und Gegenstand der Studie	93
5.7.3	Ergebnisse und Diskussion	95
5.7.4	Gültigkeit der Fallstudie	98
6	Generierung von Migrationsfunktionalität zur Evolution von Daten	101
6.1	Grundlagen	102
6.1.1	JSON	102
6.1.2	Durchführung von Datenmigrationen	104
6.2	Überblick zur Generierung von Migrationsfunktionalität	104
6.2.1	Einsatzszenarien	104
6.2.2	Methodik zur Generierung von Migrationsfunktionalität	107
6.3	Ermittlung von Pfaden zu Modellelementen	108
6.3.1	Ermittlung von Pfaden	108
6.3.2	Konstruktion von Graphen zur Abbildung von Pfaden	115
6.4	Vereinfachung des Differenzmodells	117
6.4.1	Verschiebung eines Attributs	118
6.4.2	Veränderung der Quellklasse einer Assoziation	119
6.4.3	Veränderung der Zielklasse einer Assoziation	121
6.4.4	Verschiebung eines Literals aus einer Enumeration	122
6.4.5	Verschiebung eines Pakets oder einer Klasse	123
6.4.6	Umbenennung eines Pakets	123
6.4.7	Löschen eines Pakets	123
6.4.8	Veränderung einer Klasse hin zu einer (nicht) abstrakten Klasse	123
6.4.9	Hinzufügen einer Vererbungsbeziehung	124
6.4.10	Nicht betrachtete Modelländerungen	125
6.5	Verarbeitung von Modelländerungen	126
6.5.1	Handhabung von Modelländerungen zu Attributen oder Assoziationen	127
6.5.2	Handhabung von Modelländerungen zu Enumerationen	140
6.5.3	Handhabung von Modelländerungen zu Klassen	142
6.5.4	Zusammenfassung	151
6.6	Generierung von Migrationsfunktionalität	153
6.6.1	Minimierung der Graphen	153
6.6.2	Generierung der Traversierungslogik	155
6.6.3	Generierung der Migrationslogik	158
6.6.4	Generierung der Migrationslogik für die Attributverschiebung	161

6.6.5	Generierung der Migrationslogik für die Veränderung der Quellklasse einer Assoziation	165
6.6.6	Durchführung manueller Anpassungen an generierter Migrationsfunktionalität	166
6.7	Diskussion	167
6.7.1	Allgemeine Diskussion	167
6.7.2	Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen	167
6.8	Verwandte Arbeiten	168
6.9	Fallstudie in der Industrie	170
6.9.1	Ziele und Forschungsfragen	170
6.9.2	Aufbau und Gegenstand der Studie	171
6.9.3	Ergebnisse und Diskussion	172
6.9.4	Gültigkeit der Fallstudie	174
7	Modellierung von Generatorvariabilität	175
7.1	Grundlagen	176
7.1.1	Feature-orientierte Programmierung	176
7.1.2	Template-basierte Generierung	177
7.2	Beispiel	179
7.3	Methodik zur Modellierung von Generatorvariabilität	181
7.4	Variabilitätskonzepte	183
7.4.1	Variabilitätsregionen	183
7.4.2	Definition von Kompositionsoperationen	185
7.5	Konfiguration von Generatorvarianten	187
7.5.1	Konfiguration von Generatorschichten	187
7.5.2	Konfiguration einer Generatorvariante	191
7.5.3	Automatische Ermittlung und Validitätsprüfung der benötigten CGLs	192
7.6	Kompositionale Erzeugung von Generatorvarianten	196
7.6.1	Überblick über die Schritte zur Durchführung der Komposition	199
7.6.2	Ermittlung der Auswirkungen der im Schichtdefinitionsmodell modellierten Kompositionsoperationen	200
7.6.3	Erzeugung der Generatorartefakte der Generatorvariante	205
7.6.4	Adaption der Generatorartefakte zur Anpassung der Namen	210
7.7	Diskussion	215
7.7.1	Allgemeine Diskussion	215
7.7.2	Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen	217
7.8	Verwandte Arbeiten	218
7.9	Fallstudie in der Industrie	219
7.9.1	Ziele und Forschungsfragen	219
7.9.2	Aufbau und Gegenstand der Studie	220
7.9.3	Ergebnisse und Diskussion	221

7.9.4	Gültigkeit der Fallstudie	224
8	Modellbasierte Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen	225
8.1	Grundlagen	226
8.1.1	Begriffsdefinitionen	226
8.1.2	Motivation im Projektkontext und Anforderungen	228
8.2	Überblick über den Ansatz zur Prüfung der Architekturkonformität	230
8.3	Domänenspezifische Sprache zur Definition von Abhängigkeitsmodellen	232
8.3.1	Definition einer Feature und Plugin-Basis	233
8.3.2	Definition von Feature und Plugin-Gruppen	235
8.3.3	Definition von Abhängigkeitsrestriktionen	238
8.4	Semantik eines Abhängigkeitsmodells	243
8.5	Vorgehensweise zur Prüfung der Architekturkonformität	245
8.5.1	Implementierung	245
8.5.2	Beispiel	250
8.6	Anwendungsmethodik	254
8.7	Diskussion	255
8.7.1	Allgemeine Diskussion	255
8.7.2	Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen	257
8.8	Verwandte Arbeiten	258
8.9	Fallstudie in der Industrie	261
8.9.1	Ziele und Forschungsfragen	261
8.9.2	Aufbau und Gegenstand der Studie	261
8.9.3	Ergebnisse und Diskussion	262
8.9.4	Gültigkeit der Fallstudie	269
9	Zusammenfassung und Ausblick	271
9.1	Zusammenfassung	271
9.2	Ausblick	274
	Literaturverzeichnis	277
A	Abkürzungsverzeichnis	311
B	Notationen	313
C	Checkliste	315
D	Vordefinierte Bedingungen der DSL IRS	319
E	Grammatiken	323
E.1	Grammatik der DSL DUPS	323
E.2	Grammatik der DSL IRS	324

E.3	Grammatik der DSL LDL	325
E.4	Grammatik der DSL PCL	326
E.5	Grammatik der DSL DepCoL	326
F	Kontextbedingungen	331
F.1	Kontextbedingungen für IRS	331
F.2	Kontextbedingungen für LDL	332
F.3	Kontextbedingungen für PCL	333
F.4	Kontextbedingungen für DepCoL	333
G	Fragebögen zu Fallstudien	337
G.1	Fragebögen bezüglich der Checklistenenerzeugung	337
G.2	Fragebögen bezüglich der Architekturkonformität	339
H	Glossar	341
I	Curriculum Vitae	345

Kapitel 1

Einleitung

Software ist ein wettbewerbsentscheidender Faktor, ohne den innovative Produkte nicht mehr denkbar sind [BJN⁺06]. So steigt der Softwareanteil in Automobilen bereits seit vielen Jahren drastisch an und wird voraussichtlich auch in den nächsten Jahren sukzessive ansteigen [Bro06, BKPS07, PBKS07]. Dies ist im Automobilkontext insbesondere darin begründet, dass die meisten Innovationen über die Elektronik eines Fahrzeugs getrieben werden, die maßgeblich durch Software realisiert wird [SZ10]. Darüber hinaus ist Software ein integraler Bestandteil in vielen weiteren Industrien wie der Luft- und Raumfahrtindustrie oder der Telekommunikationsbranche [BJN⁺06, KSW11, Rie13].

Der Lebenszyklus eines Softwaresystems endet jedoch nicht mit der initialen Implementierung und Auslieferung des Softwaresystems, sondern erstreckt sich weit darüber hinaus. Dies ist darin begründet, dass Softwaresysteme lange Zeit im Einsatz sein können und daher lange Zeit gewartet werden müssen. Das ist insbesondere insofern relevant, als dass die meisten Angaben zu der Verteilung der Kosten auf den Lebenszyklus eines Softwaresystems induzieren, dass die Wartung von Software die kostenintensivste Aktivität darstellt. So berichtet Lehman in [Leh79] darüber, dass 70 Prozent der Gesamtkosten auf Wartungsaktivitäten und lediglich 30 Prozent auf Entwicklungsaktivitäten entfallen. Andere Studien implizieren, dass die Softwarewartung für mindestens 50 Prozent [Lie83, Ben87, MD08, XCN09], teilweise aber sogar für über 90 Prozent der Gesamtkosten verantwortlich ist [MD08, XCN09]. Folglich besitzt die Wartung der Software nach der initialen Implementierung eine zentrale Bedeutung in der Softwareentwicklung.

In einigen Arbeiten bezieht sich der Begriff der Softwarewartung auf primär fehlerkorrigierende Modifikationen eines Softwaresystems nach dessen Auslieferung. Diese Ansicht wird bereits von Royce vertreten, der im Wasserfall-Vorgehensmodell die Wartungsphase als letzte Phase einordnet, nachdem ein System ausgeliefert wurde [Roy70]. Diese Auffassung legt jedoch nahe, dass ein Softwaresystem nach der initialen Entwicklung keinen wesentlichen Änderungen mehr unterliegt [GG08, MD08]. Erst einige Jahre später stieg das Bewusstsein dafür, dass ein Softwaresystem kontinuierlich angepasst werden muss, da es andernfalls seinen Nutzen verliert [Leh79]. Dieser Fokus auf kontinuierliche Änderungen eines Systems führte zu der Einführung des Begriffs der Software Evolution (siehe Definition 1).

In dieser Arbeit wird die Software Evolution im industriellen Kontext untersucht. In Abschnitt 1.1 wird das Thema dieser Arbeit genauer motiviert und der Kontext der Arbeit eingeführt. Darauf folgend wird in Abschnitt 1.2 die wissenschaftliche Fragestellung vorgestellt, die dieser Arbeit zugrunde liegt. Abschließend wird der Aufbau der Arbeit in Abschnitt 1.3 erläutert.

Definition 1 (Software Evolution)

Software Evolution ist ein kontinuierlicher Prozess im Lebenszyklus eines Softwaresystems [Leh79], in dem ein Softwaresystem über eine Folge von Änderungsoperationen angepasst wird. Die durchzuführenden Änderungen sind primär in neuen oder sich ändernden Anforderungen oder in Änderungen der Ausführungsumgebung eines Softwaresystems begründet [BR00]. Der Fokus der Änderungen liegt auf einer stetigen funktionalen Verbesserung des Softwaresystems, kann aber auch die Behebung von Fehlern betreffen [Art88].

1.1 Motivation und Kontext der Arbeit

Lehman postulierte bereits in den 80er Jahren, dass die Komplexität eines Softwaresystems im Laufe der Software Evolution stetig ansteigt, außer es werden explizite Schritte zur Reduktion der Komplexität umgesetzt [Leh79]. Ein Ansatz um mit dieser steigenden Komplexität umzugehen besteht in dem Einsatz von Modellen, die Aspekte eines Softwaresystems auf einem höheren Abstraktionsniveau beschreiben [SVEH07, Sch12]. Analog zu [Sch12] wird in dieser Arbeit der Standpunkt vertreten, dass ein Modell nur dann als Modell erachtet werden kann, wenn es eine reale Abstraktion mit Hinblick auf Detailauslassungen bietet. Programmcode einer allgemeinen Programmiersprache (engl. general purpose language, GPL) wie Java erfüllt diese Anforderungen nicht, so dass fortan zwischen Modellen und Code unterschieden wird.

In der modellgetriebenen Softwareentwicklung bilden Modelle die zentralen Entwicklungsartefakte. Dabei werden konstruktive Modelle unter dem Einsatz von Code-Generatoren, im Folgenden als Generatoren abgekürzt, in konkrete Implementierungen transformiert [FR07, FHR08, Rum12]. Auf diese Weise können technische Details durch Generatoren gekapselt werden und die Modelle können größtenteils unabhängig von technischen Details beschrieben werden. Aus Modellen wird jedoch nicht zwangsläufig Code generiert. Darüber hinaus können Interpreter zum Einsatz kommen, die Modelle automatisch verarbeiten und darauf aufbauend Code ausführen [SVEH07, BCW12]. Ferner können Modelle beispielsweise zu Analysezwecken eingesetzt werden, so dass unter anderem eine Konsistenzanalyse zwischen der Implementierung des Softwaresystems und den Modellen durchgeführt werden kann [Sch12, GMR15]. Überdies werden Modelle weiterhin auch zu Kommunikations- und Dokumentationszwecken eingesetzt [SVEH07, Gmb10, Sch12].

Zur Erzeugung von Modellen können verschiedenartige Modellierungssprachen zum Einsatz kommen. Mit der Unified Modeling Language (UML) [OMG10c, OMG10b] liegt ein Standard für die Modellierung vor, der nicht nur in der Forschung, sondern auch in der Industrie weit verbreitet ist [Kra10, HWRK11, Wei12]. Die UML stellt eine Sammlung von Modellierungssprachen dar, über die verschiedenste Aspekte von Softwaresystemen ausgedrückt werden können und mittels derer beliebige Softwaresysteme modelliert werden können [Sch12]. Im Gegensatz zu einer derartigen allgemeingültigen Modellierungssprache existieren darüber hinaus domänenspezifische Sprachen (engl. domain specific language, DSL), die auf eine bestimmte Domäne zugeschnitten sind [Fow10]. Der Fokus auf eine dedizierte Domäne ermöglicht es Experten der Domäne, die notwendigen Sachverhalte präziser auszudrücken als in einer allgemeingültigen Modellierungssprache wie der UML.

In der Industrie kommen sowohl UML als auch DSLs zum Einsatz. Gemäß einer Umfrage von Hutchinson [HWRK11, HWR14], nutzen 85 Prozent der Entwickler in der Industrie die UML und circa 40 Prozent der Entwickler DSLs zu Modellierungszwecken. Eine etwas geringere Verbreitung von Modellierungstechniken in der Industrie wird durch die Studien von Tomassetti [TTR⁺11, TTT⁺12] dargelegt, wonach 68 Prozent der befragten Personen Modellierungstechniken verwenden, wobei von diesen 76 Prozent die UML und 21 Prozent DSLs nutzen. Trotz dieser weiten Verbreitung von UML, berichtete beispielsweise Whittle [WHR14], dass die Unternehmen, die erfolgreich modellgetriebene Softwareentwicklung angewendet haben, eher DSLs als UML zur Modellierung eingesetzt haben. Dabei handelte es sich oft um sehr kleine DSLs mit einem sehr engen Fokus. In dieser Arbeit werden konform zu diesen Ergebnissen für spezifische Zwecke kleine DSLs mit einem sehr eng definierten Fokus eingesetzt.

Während die zuvor genannten Studien implizieren, dass Modellierungstechniken in der Industrie bereits relativ weit verbreitet sind, ist die generative Nutzung von Modellen weniger stark etabliert. Die Ergebnisse der Umfrage in [Gmb10] zeigen, dass circa 50 Prozent der Befragten Modellierung und aufbauend darauf Generierung einsetzen. Überdies legt [TTR⁺11, TTT⁺12] dar, dass lediglich 30 Prozent der Befragten der Studie auf Basis von Modellen automatisch Code generieren.

In dieser Arbeit liegt der Fokus auf Softwaresystemen, die modellgetrieben entwickelt und im industriellen Kontext eingesetzt werden. Das Hauptziel dieser Arbeit besteht darin zu veranschaulichen, wie die Software Evolution in solchen Systemen durch den Einsatz von Modellierungs- und Generierungstechniken unterstützt werden kann. Die Erkenntnisse dieser Arbeit sollen als Ausgangsbasis für derartige Softwaresysteme dienen, um dort die Software Evolution in spezifischen Problembereichen ebenfalls über Modellierungs- und Generierungstechniken unterstützen zu können. Auf diese Weise soll die industrielle Nutzung der modellgetriebenen Softwareentwicklung gefördert werden.

Die modellgetriebene Herangehensweise zur Entwicklung eines Softwaresystems führt dazu, dass im Zuge der Software Evolution neben dem Code eines Softwaresystems insbesondere auch die Modelle angepasst werden müssen. Werden diese Modelle zur Code-Generierung eingesetzt, können sich Änderungen in Modellen unmittelbar auf die generierten Artefakte und indirekt somit auch auf die handgeschriebenen Artefakte auswirken. Im Rahmen der Software Evolution müssen folglich auch die Modell Evolution (siehe Definition 2) und die damit verbundenen Auswirkungen betrachtet werden.

Definition 2 (Modell Evolution)

Modell Evolution bezeichnet einen kontinuierlichen Prozess im Lebenszyklus eines Modells, in dem ein Modell über eine Folge von Änderungsoperationen angepasst wird. Die durchzuführenden Änderungen sind primär in neuen oder sich ändernden Anforderungen oder in Änderungen der Ausführungsumgebung des modellierten Softwaresystems begründet.

Dies erfordert automatisierte Mechanismen zur Ermittlung von Modellunterschieden, da andernfalls nicht korrekt nachvollzogen werden kann, welche Modelländerungen im Rahmen der Modell Evolution vorgenommen wurden. Weiterhin ist die automatisierte Bestimmung von Modellunterschieden die Grundvoraussetzung dafür, Aussagen über die Auswirkungen der Modelländerungen treffen zu können. Dieser Aspekt ist sowohl für die Modell Evolution als auch für

die Software Evolution im Allgemeinen von entscheidender Bedeutung. Aus diesem Grund sind diese Aspekte für diese Arbeit besonders wichtig.

Die Entwicklung von Softwaresystemen, die in der Industrie eingesetzt werden, ist mit spezifischen Herausforderungen verbunden. So kann die Weiterentwicklung eines Systems normalerweise nicht für längere Zeit gestoppt werden, nur um in einem bestimmten Problembereich einen neuen Ansatz zu evaluieren und in das System zu integrieren. Stattdessen müssen sich neue Ansätze oder Änderungen des Entwicklungsprozesses möglichst nahtlos integrieren lassen, damit die Weiterentwicklung des Systems nicht beeinträchtigt wird. Darüber hinaus kann jeder Fehler, der durch eine Änderung in ein Softwaresystem eingeführt wird, zu sehr hohen Kosten führen. Ferner erhöht sich durch den produktiven Einsatz eines Softwaresystems die Wahrscheinlichkeit für Änderungen bedingt durch neue oder geänderte Anforderungen oder durch Anpassungen an die Systemlandschaft eines spezifischen Kunden. Diese genannten Aspekte gelten in Softwaresystemen, die lediglich zu Forschungszwecken oder für private Zwecke entwickelt werden, normalerweise in deutlich geringerem Maße.

Konkret sind die verschiedenen Ansätze dieser Arbeit durch Softwaresysteme bei dem Unternehmen DSA GmbH [DSAA] motiviert und werden in den dort in Entwicklung bzw. Wartung und Evolution befindlichen Produkten evaluiert. DSA zählt zu den führenden Herstellern von Hard- und Softwaresystemen zur Prüfung und Diagnose elektrischer und elektronischer Fahrzeugkomponenten [DSAd] mit zur Zeit circa 400 Mitarbeitern. Der Schwerpunkt der Untersuchungen liegt auf dem Softwaresystem PRODIS.Authoring, im Folgenden abgekürzt als Authoring, welches unter anderem zur Erstellung und Verwaltung von Prüfabläufen und Diagnose-daten eingesetzt wird [DSAb]. Prüfabläufe können entlang der gesamten Diagnoseprozesskette von der Entwicklung von Steuergeräten über die Produktion von Fahrzeugen bis zum Werkstattbetrieb eingesetzt werden. So können Prüfabläufe beispielsweise die Durchführung von Ak-tuatortests wie ABS-Tests sowie das Lesen von Messwerten oder Fehlercodes in Steuergeräten betreffen.

Der steigende Softwareanteil in Fahrzeugen und die steigende Softwarekomplexität in Fahr-zeugen führen zu stetig neuen Anforderungen an die Softwaresysteme bei DSA. Aus diesem Grund ist die Unterstützung der Software Evolution besonders wichtig. Aufgrund der Vielzahl der Themengebiete, die im Umfeld der Software Evolution existieren, kann in dieser Arbeit lediglich eine Auswahl von Teilgebieten der Software Evolution betrachtet werden. Die unter-suchten Teilgebiete wurden einerseits mit Hinblick auf die Ausgangssituation bei DSA aus-gewählt, andererseits wurde darauf geachtet, dass die Teilgebiete und die damit verbundenen Herausforderungen nicht spezifisch für die Softwaresysteme bei DSA sind. Stattdessen können die ausgewählten Themengebiete für Softwaresysteme relevant sein, die mindestens eines der folgenden Kriterien erfüllen:

- **C1:** Für die Entwicklung des Softwaresystems werden Modellierungstechniken einge-setzt. Aufgrund der Ausgangssituation bei DSA liegt der Fokus auf dem Einsatz von UML-Klassendiagrammen, im Folgenden als Klassendiagramme abgekürzt. Dies ist je-doch keine starke Einschränkung, da Klassendiagramme die am häufigsten eingesetzte Modellierungstechnik der UML sind [HWRK11, HWR14].
- **C2:** Das Softwaresystem wird modellgetrieben entwickelt und ein Generator erzeugt Teile

des Softwaresystems auf Basis von Modellen (siehe **C1**). Fehlende Teile des Softwaresystems oder erforderliche Anpassungen der generierten Logik werden über handgeschriebene Implementierungen bereitgestellt. Letztgenannter Punkt gilt normalerweise für jedes modellgetriebene Softwareentwicklungsprojekt, welches Code-Generierung einsetzt. Der Grund hierfür ist, dass es nicht praktikabel ist, jedes Detail eines Softwaresystems zu modellieren [Wil03, SVC06, GHK⁺15a, GHK⁺15b].

- **C3:** Die Struktur der von dem Softwaresystem verarbeiteten Daten wird über konstruktive Datenmodelle (siehe Definition 3) beschrieben. Änderungen der Datenmodelle können dadurch zu Veränderungen der Struktur der Daten führen. Bei diesem Kriterium handelt es sich um eine spezielle Ausprägung von **C1** und **C2**, da in diesen beiden Kriterien nicht eingeschränkt wird, welche Art von Modell entwickelt wird.

Definition 3 (Datenmodell)

Ein Datenmodell definiert die Struktur der Daten, die durch ein Softwaresystem verarbeitet werden.

- **C4:** Es werden verschiedene Varianten eines Softwaresystems entwickelt. Diese Varianten können sich einerseits mit Hinblick auf den generellen Funktionsumfang, aber auch mit Hinblick auf die Art der Implementierung spezifischer Systemteile unterscheiden. Zudem soll es auf einfache Weise möglich sein neue Varianten eines Softwaresystems zu erzeugen. Alternativ dazu ist dieses Kriterium auch erfüllt, wenn verschiedene Softwaresysteme entwickelt werden, in denen gleichartige Funktionalität in unterschiedlichen Ausprägungen eingesetzt wird.
- **C5:** Das Softwaresystem ist in verschiedene Plugins (siehe Definition 4) aufgeteilt, die jeweils eine Teilfunktionalität des Softwaresystems kapseln. Auf diese Weise lässt sich ein Softwaresystem besser modularisieren. Bei diesem Kriterium handelt es sich um eine spezielle Ausprägung von **C4**, da durch die Kombination von Plugins auf einfache Weise Varianten eines Softwaresystems erzeugt werden können.

Definition 4 (Plugin)

Ein Plugin stellt die kleinste betrachtete Komponente eines Plugin-basierten Systems dar, die im Wesentlichen aus einer Menge von Klassen und benötigten Bibliotheken besteht.

1.2 Wissenschaftliche Fragestellung

Die zentrale Fragestellung, die in dieser Arbeit beantwortet werden soll, lautet:

Wie kann die Evolution eines komplexen Softwaresystems, welches im industriellen Kontext modellgetrieben entwickelt wird, durch den Einsatz von Modellierungs- und Generierungstechniken unterstützt werden?

Diese Hauptforschungsfrage betrifft ein sehr breites Themenspektrum, welches in einer Arbeit keinesfalls ganzheitlich untersucht werden kann. Aus diesem Grund wird die Hauptforschungsfrage nachfolgend über verschiedene Teilfragen verfeinert. Diese Teilfragen sind motiviert durch die zuvor eingeführten Charakteristiken **C1**, **C2**, **C3**, **C4** und **C5**. Zu jeder Teilfrage ist daher vermerkt, welche Charakteristiken die jeweilige Teilfrage betrifft. Darüber hinaus wird für jede Teilfrage kurz dargelegt, inwiefern die Frage für die Arbeit relevant ist. In Kapitel 2 werden die einzelnen Forschungsfragen detailliert motiviert und der wissenschaftliche Beitrag zu jeder Frage zusammengefasst.

RQ1: Wie kann verhindert werden, dass die automatische Ermittlung von Modellunterschieden fehlerhafte Modelländerungen meldet (**C1**, **C2**, **C3**)?

Automatische Verfahren zur Bestimmung von Modellunterschieden sind eine grundlegende Voraussetzung zur Unterstützung der Modell Evolution. Die bestehenden Ansätze bieten jedoch keine Möglichkeit sicherzustellen, dass die korrekten Modellunterschiede erkannt werden [PMR13, MR14b]. Wenn auf Basis der erkannten Modelländerungen automatisch weitere Aktionen durchgeführt werden, kann dies allerdings die Modell Evolution und die Software Evolution beeinträchtigen.

RQ2: Wie kann die Kenntnis über Modelländerungen im weiteren Verlauf der Evolution eines Softwaresystems effizient genutzt werden (**C2**, **C3**)?

In dieser Arbeit werden insbesondere zwei Facetten dieser Fragestellung untersucht. Aus diesem Grund wird die Forschungsfrage in zwei Teilfragen gegliedert, die im Folgenden vorgestellt werden.

RQ2.1: Wie können Entwickler in einem generativen Softwareentwicklungsprozess dabei unterstützt werden die Auswirkungen von Modelländerungen vorherzusehen (**C2**, **C3**)?

Wie zuvor erläutert, sind Modelländerungen unabdingbar für die Software Evolution in generativen Softwareentwicklungsprozessen. Jede Modelländerung kann Auswirkungen auf eine Vielzahl von Artefakten eines Softwaresystems haben. Für Entwickler ist es jedoch in manchen Situationen schwierig die Auswirkungen der verschiedenen Modelländerungen einzuschätzen, so dass die Software Evolution dadurch beeinträchtigt wird.

RQ2.2: Wie können in einem Softwaresystem, in dem konstruktive Datenmodelle eingesetzt werden, automatisch passende Datenmigrationen auf Basis von Modelländerungen erzeugt werden (**C3**)?

Die Daten, die von einem Softwaresystem verarbeitet werden, unterliegen genauso der Evolution wie andere Bestandteile eines Softwaresystems. Durch die Modellierung von konstruktiven

Datenmodellen können Modelländerungen zu einer Veränderung dieser Struktur führen. Existierende Daten, die konform zu der vorherigen Struktur waren, können im Allgemeinen jedoch nicht verworfen werden, da für die Benutzer des Softwaresystems diese Daten das zentrale Gut darstellen. Folglich müssen die bestehenden Daten im Zuge der Software Evolution möglichst verlustfrei migriert werden.

RQ3: Wie kann die Konstruktion von Generatorvarianten vereinfacht werden (C2, C4)?

Generatoren können zur Generierung verschiedenster Teile eines Softwaresystems eingesetzt werden. Allerdings müssen nicht für jedes Softwaresystem stets komplett neue Generatoren entwickelt werden. Stattdessen bietet sich die Wiederverwendung von Generatoren an, wenn zuvor bereits passende Generatoren für den jeweiligen Anwendungskontext entwickelt wurden. Ein Problem in diesem Kontext ist, dass die bestehenden Generatoren nicht zwangsläufig unverändert übernommen werden können. Vielmehr kann es aufgrund spezifischer Anforderungen erforderlich sein dedizierte Teile eines Generators auf andere Weise umzusetzen. Dazu muss es allerdings möglich sein Variabilität in Generatoren zu integrieren, um Varianten von Generatoren (siehe Definition 5) entwickeln zu können.

Definition 5 (*Generatorvariante*)

Eine Generatorvariante ist ein konkreter Generator, der Teile bestehender Generatoren wiederverwendet, erweitert und/oder anpasst.

RQ4: Wie kann dem Problem entgegengewirkt werden, dass die Implementierung eines Plugin-basierten Softwaresystems von der Softwarearchitektur des Softwaresystems abweicht (C4, C5)?

In einem Plugin-basierten Softwaresystem ist die Gesamtfunktionalität des Softwaresystems auf verschiedene Plugins aufgeteilt. Die einzelnen Plugins eines Plugin-basierten Softwaresystems sind aber nicht zwangsläufig unabhängig voneinander. Stattdessen kann jedes Plugin Abhängigkeiten zu einer Vielzahl anderer Plugins besitzen. Sobald ein Plugin eine Abhängigkeit zu einem anderen Plugin besitzt, kann es auf dessen Klassen zugreifen.

Im Laufe der Software Evolution müssen die Plugins kontinuierlich angepasst werden. Dies kann sich auch auf die Abhängigkeiten zwischen den Plugins auswirken. Beispielsweise können Entwickler aufgrund neuer oder geänderter Anforderungen neue Abhängigkeiten einführen, um die Funktionalität eines anderen Plugins nutzen zu können. Die Softwarearchitektur eines Plugin-basierten Softwaresystems restringiert, welches Plugin von welchen anderen Plugins abhängen darf. Durch die Änderung der Plugins kann es in der Implementierung aber zu Abweichungen zu diesen Vorgaben kommen, was zu Qualitätsproblemen führen kann und daher vermieden werden sollte.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt.

Kapitel 2 beschreibt die verschiedenen Teilgebiete aus dem Bereich der Software Evolution, die in dieser Arbeit untersucht werden. Zu jedem Teilgebiet wird dargelegt, warum in dem Bereich Unterstützung hilfreich ist. Ferner wird jede Problemstellung auf den Kontext dieser Arbeit übertragen und erläutert, wie die Problemstellung in dieser Arbeit angegangen wird. Für jeden dazu entwickelten Ansatz wird zudem jeweils kurz dargelegt, wie die Entwickler den Ansatz konkret anwenden.

Kapitel 3 gibt einen Überblick über die Grundlagen der DSL-Entwicklung. An verschiedenen Stellen in dieser Arbeit werden DSLs vorgestellt. Dieses Kapitel beschreibt die wesentlichen Bestandteile der Grammatiken, anhand derer die DSLs in dieser Arbeit definiert werden.

Kapitel 4 gibt einen Überblick über den Stand der Technik bei der Ermittlung von Modellunterschieden und stellt den Ansatz vor, der in dieser Arbeit eingesetzt wird. Der Vergleich zweier Modelle bzw. Modellversionen und die Berechnung der Modellunterschiede bildet eine fundamentale Grundlage für diese Arbeit. So operieren die Aktivitäten, die in Kapitel 5 und Kapitel 6 beschrieben werden, auf Modellunterschieden.

Kapitel 5 beschreibt einen Ansatz zur Auswirkungsanalyse. In diesem Ansatz werden Auswirkungen von Modelländerungen in expliziten Auswirkungsregeln beschrieben. Diese Auswirkungsregeln werden genutzt um Checklisten zu generieren, die Entwickler auf die Auswirkungen der identifizierten Modellunterschiede hinweist.

Kapitel 6 stellt vor, wie Migrationsfunktionalität generiert wird, über die eine korrekte Daten Evolution sichergestellt wird. Die resultierende Datenmigrationsfunktionalität kann sowohl für die Datenmigration im Kontext der Client-Server Kommunikation als auch für die Datenmigration im Datenbankbereich eingesetzt werden.

Kapitel 7 stellt einen Ansatz zur Modellierung von Generatorvariabilität vor. Dieser Ansatz zielt darauf ab Varianten von Generatoren entwickeln zu können, indem Teile bestehender Generatoren durch neuen Inhalt ersetzt oder neuer Inhalt zu diesen hinzugefügt wird.

Kapitel 8 beschreibt einen Ansatz zur automatischen Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen. Zu diesem Zweck können Abhängigkeitsrestriktionen modelliert werden, die beschreiben, ob Abhängigkeiten zwischen spezifischen Elementen eines Plugin-basierten Softwaresystems erlaubt oder verboten sind. Eine Prüfkomponente stellt indes sicher, dass die Architekturkonformität automatisch geprüft wird. Die Entwickler werden auf diese Weise umgehend darüber benachrichtigt, wenn sie Verstöße gegen modellierte Abhängigkeitsrestriktionen einführen.

Kapitel 9 fasst abschließend die Ergebnisse der Arbeit zusammen und diskutiert mögliche Erweiterungen, die in zukünftigen Arbeiten untersucht werden können.

Kapitel 2

Kontext der Arbeit und Methodik

In diesem Kapitel wird der Kontext dieser Arbeit genauer vorgestellt. Dazu wird ein grober Überblick über ausgewählte Hard- und Softwaresysteme von DSA gegeben und dargestellt, wie diese in den Bereich der Fahrzeugdiagnose eingebettet sind. Anschließend werden die verschiedenen Teilgebiete der Software Evolution vorgestellt, die in dieser Arbeit untersucht werden. Zu Beginn jedes Teilgebiets wird die zugrundeliegende wissenschaftliche Fragestellung wiederholt, die in Abschnitt 1.2 bereits eingeführt wurde. Daraufgehend wird für jedes Teilgebiet die Problemstellung allgemein dargelegt und auf den Kontext dieser Arbeit übertragen. Ferner wird zu jedem Teilgebiet der wissenschaftliche Beitrag dieser Arbeit kurz zusammengefasst. Im Zuge dessen wird auch erläutert, wie die Entwickler die in dieser Arbeit entwickelten Ansätze einsetzen können.

2.1 Systemlandschaft bei DSA

DSA bietet verschiedenste Hard- und Softwarelösungen zur Prüfung und Diagnose elektrischer und elektronischer Fahrzeugkomponenten [DSAd]. Im Kontext der Hardwarelösungen bietet DSA beispielsweise mobile Diagnose-Interfaces (engl. mobile diagnostic interface, MDI) an, die in allen Bereichen für die Diagnosekommunikation mit den Steuergeräten eines Fahrzeugs eingesetzt werden können. Im Kontext der Softwarelösungen bietet DSA unter anderem mit PRODIS.Automation [DSAc], im Folgenden als Automation abgekürzt, ein Autorenwerkzeug für den Produktionsbereich und mit Authoring ein Autorenwerkzeug für den After-Sales Bereich an. Beide Systeme ermöglichen die Erstellung und Verwaltung von Diagnosedaten sowie die Erstellung von Prüfabläufen. Über die Diagnosedaten können beispielsweise die Fahrzeugtopologie, Bussysteme und Steuergeräte einheitlich beschrieben werden [ZS14]. Über die Prüfabläufe können hingegen Fahrzeuge in der Produktionslinie oder in der Werkstatt getestet und programmiert werden. Beiden Systemen ist zudem gemeinsam, dass es verschiedene Varianten der Systeme gibt. Von Automation existieren im Wesentlichen drei Varianten, die sich teilweise im Funktionsumfang unterscheiden. Zudem kann kundenspezifisch der Funktionsumfang definiert werden. Für Authoring existieren hingegen zwei Varianten, die sich ebenfalls in ihrem Funktionsumfang unterscheiden. Darüber hinaus unterstützen beide Systeme allerdings auch zahlreiche weitere unterschiedliche Funktionsbausteine, die sich aus den unterschiedlichen Anforderungen der Fahrzeugdiagnose in der Produktion und im After-Sales Bereich ergeben. Auf diese wird im Folgenden allerdings nicht genauer eingegangen, da diese für die Arbeit nicht relevant sind.

Zu beiden Autorenwerkzeugen existiert jeweils ein zugehöriges Diagnoselaufzeitsystem, PRODIS.RTS [DS Ae] für Automation und PRODIS.WTS [DS Af] für Authoring, im Folgenden als RTS und WTS abgekürzt. Diese Laufzeitsysteme können die in Automation und Authoring erstellten Diagnosedaten und Prüfabläufe interpretieren bzw. ausführen.

Abbildung 2.1 illustriert auf vereinfachte Weise den groben Zusammenhang zwischen Authoring, WTS, einem Diagnosetester sowie einem Fahrzeug in der Werkstatt. Auf der linken Seite ist dargestellt, dass über Authoring Diagnosepakete erstellt werden können, die unter anderem Diagnosedaten und Prüfabläufe enthalten können. Die Diagnosepakete, Diagnosedaten, Prüfabläufe sowie die weiteren Daten werden auf einem Server gespeichert und in einer Datenbank versioniert. Die erzeugten Diagnosepakete können anschließend nahtlos durch WTS ausgeführt werden. In der Abbildung ist dazu angedeutet, dass WTS in der Werkstatt auf einem Laptop installiert ist. Dieser Laptop ist beispielsweise über Kabel oder Bluetooth mit einem MDI verbunden, welches wiederum mit dem zu untersuchenden Fahrzeug verbunden ist. Auf diese Weise können die Mitarbeiter in der Werkstatt über den Laptop und das MDI mit dem Fahrzeug interagieren und beispielsweise Fehlercodes aus Steuergeräten auslesen oder sonstige Prüfungen anstoßen. Alternativ ist in der Abbildung angedeutet, dass an das zu untersuchende Fahrzeug ein schnurloses Diagnose-Interface (engl. wireless diagnostic interface, WDI) angeschlossen ist. Ein solches WDI kann mit dem Laptop über WLAN oder Bluetooth kommunizieren.

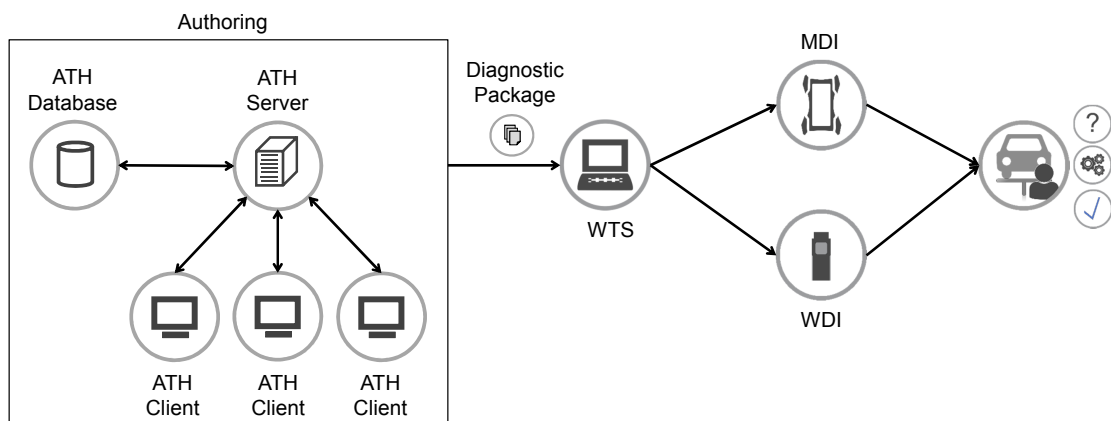


Abbildung 2.1: Zusammenhang zwischen Authoring, WTS, Diagnosegeräten und dem Fahrzeug in der Werkstatt.

Sämtliche zuvor genannten DSA-Softwaresysteme sind Java-Anwendungen, die über die Eclipse Rich Client Platform (RCP) [Vog15] entwickelt wurden. Eine RCP-Anwendung besteht aus verschiedenen Komponenten, die wiederum aus Features und Plugins bestehen. Der Vorteil dieser Architektur ist, dass die Features und Plugins auf einfache Weise wiederverwendet werden können. So werden einige Features und Plugins von allen vier genannten Softwaresystemen genutzt, andere hingegen beispielsweise nur von Automation und Authoring und wiederum andere sind spezifisch für eines der Softwaresysteme. Insgesamt existieren zu den Softwaresystemen bei DSA mehr als 120 Features und 1100 Plugins, wobei zwischen den Plugins insgesamt mehr als 10000 Abhängigkeiten bestehen. Alleine Authoring umfasst dabei insgesamt circa 70 Features

und 460 Plugins.

Darüber hinaus gilt es zu beachten, dass der Softwareentwicklungsprozess bei DSA dem modellgetriebenen Ansatz folgt. Dabei werden auf Basis von Klassendiagrammen zahlreiche Artefakte für Automation und Authoring generiert, wobei der Schwerpunkt auf Authoring liegt. Konkret erzeugt der Generator bei DSA auf Basis der Klassendiagramme unter anderem Java-Klassen zur Repräsentation der Entitäten und der Datenzugriffsobjekte (engl. Data Access Object, DAO) sowie Java-Klassen zum Zugriff auf die Geschäftslogik. Ferner werden für Authoring SQL-Skripte generiert, die zur Erzeugung des zugrundeliegenden Datenbankschemas eingesetzt werden. Die Klassendiagramme umfassen insgesamt über 500 Klassen, woraus circa 2000 Java-Dateien generiert werden, die circa 450000 Zeilen Java-Code umfassen. Dies entspricht im Falle von Authoring circa 17 Prozent des gesamten Java-Codes und 10 Prozent der gesamten Java-Dateien. Die SQL-Skripte umfassen insgesamt circa 340000 Zeilen SQL-Code, wovon fast 90 Prozent generiert sind.

Die zuvor genannten Zahlen verdeutlichen bereits, wie komplex die betrachteten Softwaresysteme sind. Diese hohe Komplexität und die hohen Qualitätsanforderungen an die Systeme machen die Evolution besonders anspruchsvoll. Die in dieser Arbeit vorgestellten Ansätze haben das Ziel, die Entwickler bei der Evolution zu unterstützen, so dass manche Aktionen automatisiert erfolgen können oder die Entwickler zumindest entlastet werden.

2.2 Ermittlung von Modellunterschieden

RQ1: Wie kann verhindert werden, dass die automatische Ermittlung von Modellunterschieden fehlerhafte Modelländerungen meldet (C1, C2, C3)?

In generativen Softwareentwicklungsprozessen müssen die Modelle im Laufe der Evolution der Softwaresysteme iterativ weiterentwickelt werden, um die erforderlichen Änderungen abbilden zu können [MD08, PMR13]. Zur Gewährleistung einer adäquaten Evolution der Modelle sind daher dedizierte Werkzeuge zur Versionierung und Verwaltung von Modellen unerlässlich [MWD⁺05, LRSS07, BKL⁺12, MR14b]. Dies gilt insbesondere bei der Verarbeitung von komplexen Modellen, bei denen es nicht praktikabel ist, die Modelle ausschließlich manuell zu analysieren.

Eine zentrale Funktionalität im Bereich der Modellversionierung ist die Berechnung von Modellunterschieden. Auf der Ebene von Quellcode sind Werkzeuge zur Berechnung und Visualisierung von Unterschieden schon seit einigen Jahren weit verbreitet [BKL⁺12]. Diese bestehenden textbasierten Techniken liefern angewandt auf grafische Modelle im Allgemeinen allerdings unbefriedigende Ergebnisse [KPP06, ABK⁺09, BKL⁺12]. Der Hauptgrund dafür ist, dass die textbasierten Techniken zeilenbasiert arbeiten und das bereits kleine Modelländerungen zu großen Unterschieden in der textuellen Repräsentation der Modelle führen können. Dies betrifft selbst Änderungen, die auf der Ebene von Modelländerungen nicht berücksichtigt werden müssen, wie Änderungen des Layouts der Modelle. Aus diesem Grund mussten für die Berechnung von Modellunterschieden eigenständige Techniken entwickelt werden. Erst durch die automatisierte Berechnung von Modellunterschieden kann auf praktikable Weise geprüft werden, ob

genau die erforderlichen Modelländerungen durchgeführt wurden. Darüber hinaus bildet die Berechnung von Modellunterschieden eine grundlegende Voraussetzung für eine Analyse der Modell Evolution, die Aufschluss über die Historie der Modellelemente gibt [WK08]. Ohne dedizierte Werkzeuge ist die Analyse der Modell Evolution eine sehr zeitaufwändige Tätigkeit.

Ein fundamentales Problem bei der Modellunterschiedsberechnung ist, dass ein automatischer Ansatz im Allgemeinen die Modellunterschiede nicht immer korrekt bestimmen kann [PMR13, MR14b]. Beispielsweise kann eine Umbenennung einer Klasse in einem Klassendiagramm sowohl als Umbenennung als auch als Löschen der Klasse mit dem alten Namen und Hinzufügen einer Klasse mit dem neuen Namen interpretiert werden. Problematisch sind fehlerhafte Modellunterschiede genau dann, wenn die Modellunterschiede automatisch weiterverarbeitet und aufbauend darauf weitere Aktionen automatisch durchgeführt werden. Dies gilt beispielsweise bei der Generierung von Migrationsfunktionalität. Werden auf Basis der Modellunterschiede automatisch Migrationsskripte generiert, können fehlerhafte Modellunterschiede dazu führen, dass fehlerhafte Migrationsskripte erzeugt werden.

Wie einleitend dargelegt, folgt der Softwareentwicklungsprozess bei DSA dem modellgetriebenen Ansatz. Da die dazu eingesetzten Klassendiagramme über 500 Klassen umfassen, ist eine manuelle Analyse der Modelle zu aufwändig. Um diesen Aufwand zu reduzieren, ist eine automatisierte Ermittlung von Modellunterschieden auf der Ebene von Klassendiagrammen unerlässlich.

Aufgrund der fundamentalen Bedeutung der automatisierten Ermittlung von Modellunterschieden beschreibt Kapitel 4 die in dieser Arbeit verfolgte Vorgehensweise zur Ermittlung von Modellunterschieden. Dabei wird ein Ansatz eingeführt, der es Modellierern über eine DSL ermöglicht, für ausgewählte Modellelemente gezielt das Ergebnis der automatischen Modellunterschiedsberechnung zu steuern. Auf diese Weise kann verhindert werden, dass im Zuge der automatischen Bestimmung der Modellunterschiede fehlerhafte Modellunterschiede zurückgeliefert werden. Die Anwendung des Ansatzes ist nachfolgend beschrieben:

1. Die Modellierer verändern die bestehenden Modelle, beispielsweise aufgrund von geänderten Anforderungen.
2. Als Nächstes führen die Modellierer das Werkzeug zur Differenzberechnung aus und analysieren die gemeldeten Modellunterschiede zur letzten Version.
3. Falls die Modellierer dabei erkennen, dass das Werkzeug unzutreffende Modellunterschiede ermittelt hat, definieren sie für die betroffenen Modellelemente manuell die durchgeführten Änderungsoperationen.
4. Das Werkzeug zur Differenzberechnung wird erneut ausgeführt, wobei automatisch die zuvor definierten Änderungsoperationen einbezogen werden.
5. Die Modellierer validieren die resultierenden Modelländerungen und kehren zum zweiten Schritt zurück, falls weiterhin unzutreffende Modelländerungen gemeldet werden.

2.3 Auswirkungsanalyse zur Generierung von Checklisten

RQ2: Wie kann die Kenntnis von Modelländerungen im weiteren Verlauf der Evolution eines Softwaresystems effizient genutzt werden (C2, C3)?

RQ2.1: Wie können Entwickler in einem generativen Softwareentwicklungsprozess dabei unterstützt werden die Auswirkungen von Modelländerungen vorherzusehen (C2, C3)?

Wie zuvor bereits beschrieben, dienen in einem generativen Softwareentwicklungsprozess abstrakte Modelle als Eingabe für Generatoren. Intelligente Generatoren können bereits essentielle Teile eines Softwaresystems generieren und die Größe und Anzahl der manuell zu erstellenden Artefakte dadurch drastisch reduzieren [Rum12, MR14a]. Dessen ungeachtet ist es in der Regel erforderlich weitere Artefakte wie Quellcodedateien, Konfigurationsdateien oder Property-Dateien manuell zu erstellen und zu pflegen. Handgeschriebene Artefakte werden im Wesentlichen bereitgestellt, um komplett fehlende Teile der Systeme zu implementieren und um die generierte Logik punktuell zu adaptieren oder zu ergänzen. Die Folge davon ist, dass handgeschriebene Artefakte in eine generierte Infrastruktur integriert werden müssen [MR14a, GHK⁺15a, GHK⁺15b]. Zur Verdeutlichung illustriert Abbildung 2.2 das Generation Gap Muster [MR14a, GHK⁺15a, GHK⁺15b] zur Integration von handgeschriebenem und generiertem Code. In diesem Muster wird jeweils eine abstrakte Basisklasse generiert und die handgeschriebenen Implementierungen müssen in einer Unterklasse dieser Basisklasse vorgenommen werden.

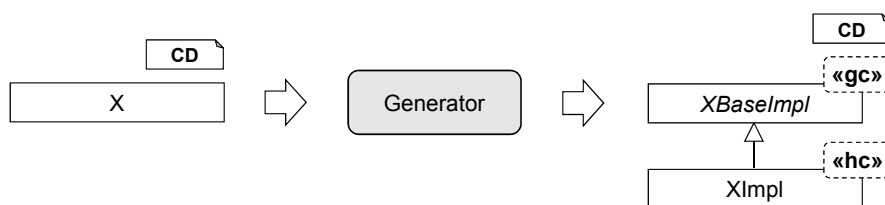


Abbildung 2.2: Klassisches Generation Gap Muster [GHK⁺15a, GHK⁺15b] zur Integration von handgeschriebenem und generiertem Code.

Aufgrund der Integration der handgeschriebenen und generierten Artefakte sind die handgeschriebenen Artefakte oftmals hochgradig abhängig von den generierten Artefakten [MR14a]. In einigen Fällen sind diese Abhängigkeiten direkt ersichtlich und einfach zu handhaben. So ist in Abbildung 2.2 direkt erkennbar, dass die handgeschriebene Klasse `XImpl` von der generierten Klasse `XBaseImpl` abhängt. Nutzt beispielsweise die Klasse `XImpl` eine generierte Methode der Klasse `XBaseImpl` und diese Methode wird aufgrund von Modelländerungen unter einem anderen Namen abgelegt, weist die Entwicklungsumgebung darauf hin, dass die Methode nicht mehr verfügbar ist.

Problematischer sind hingegen Abhängigkeiten, die für die Entwickler nicht offensichtlich sind. Zur Illustration dieses Problems wurde in Abbildung 2.3 das vorherige Generierungsschema um die Generierung eines SQL-Skripts zur Erzeugung einer Datenbanktabelle für eine Entität erweitert. In diesem Beispiel hängt die handgeschriebene Klasse `XImpl` von dem generierten Datenbankskript `tableX.sql` zu Entität `X` ab. Diese Abhängigkeit kann beispielsweise daraus resultieren, dass in der Klasse ein SQL-Query implementiert ist, welches gegen die Datenbanktabelle zu Entität `X` arbeitet. Aufgrund dieser Abhängigkeit kann es erforderlich sein die Klasse `XImpl` zu ändern, nachdem die Klasse `X` im zugehörigen Klassendiagramm angepasst wurde. Dies ist unter anderem dann gegeben, wenn für die Attribute der Klasse `X` Umbenennungen durchgeführt oder Attribute entfernt wurden. Durch eine derartige Modelländerung ändert sich der Aufbau der zugehörigen Datenbanktabelle und somit das Datenbankschema. Verweist das SQL-Query auf die von den Änderungen betroffenen Anteile, so muss das SQL-Query folglich angepasst werden, da es andernfalls nicht ordnungsgemäß ausgeführt werden kann. Für einen Entwickler ist dieses Problem aber ohne zusätzliche Werkzeugunterstützung nicht direkt erkennbar. Dadurch wirken sich die Änderungen der Klasse `X` gegebenenfalls erst zur Laufzeit des Softwaresystems in einem Fehler aus.

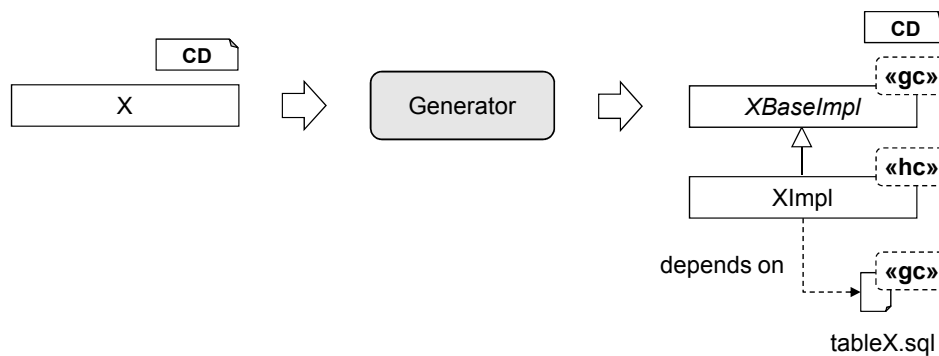


Abbildung 2.3: Klassisches Generation Gap Muster [GHK⁺15a, GHK⁺15b] zur Integration von handgeschriebenem und generiertem Code. Die Klasse `XImpl` hängt von dem generierten Datenbankskript `tableX.sql` ab.

Diese Beobachtung gilt nicht nur für Klassendiagramme oder diesen geschilderten Fall. Stattdessen besteht dieses Problem in jedem Softwaresystem, welches sowohl aus generierten als auch aus handgeschriebenen Artefakten besteht, zwischen denen Abhängigkeiten existieren. In solchen Systemen kann jede Änderung in Modellen maßgeblichen Einfluss auf die generierten und handgeschriebenen Artefakte und somit schwerwiegende Auswirkungen auf das gesamte Softwaresystem haben [MR14a]. Ohne Werkzeugunterstützung ist es nur schwer möglich festzustellen, welche Teile eines Softwaresystems auf welche Weise von einer konkreten Modelländerung betroffen sind. Ansätze zur Auswirkungsanalyse widmen sich diesem Problem, indem sie versuchen die Auswirkungen von Änderungen zu identifizieren [Boh95, Arn96]. Existierende Arbeiten zur Auswirkungsanalyse untersuchen jedoch hauptsächlich die Auswirkungen von Quellcodeänderungen [Leh11, LSLZ13] und nicht die von Modelländerungen.

Der Generator bei DSA erzeugt auf Basis der Klassendiagramme eine Vielzahl von Java-

Klassen. Zur Integration von handgeschriebenem und generiertem Code wird bei DSA eine angepasste Variante des zuvor kurz beschriebenen Generation Gap Musters eingesetzt. Die Entwickler müssen folglich zu jeder generierten Basisklasse eine handgeschriebene Unterklasse implementieren. Darüber hinaus existieren aber auch noch zahlreiche weitere handgeschriebene Klassen, die sowohl generierte als auch handgeschriebene Funktionalität verwenden. Eine ähnliche Situation liegt bei den SQL-Skripten vor. Die SQL-Skripte zur Erzeugung des Datenbankschemas sind zwar zu Großteilen generiert, allerdings existieren auch hier noch handgeschriebene Skripte. Das gesamte Datenbankschema ergibt sich somit aus einem Zusammenspiel generierter und handgeschriebener SQL-Skripte. Angesichts dieser Ausgangssituation kann jede Änderung eines Klassendiagramms für Authoring weitreichende Konsequenzen haben, da es erforderlich sein kann handgeschriebene Implementierungsklassen sowie SQL-Skripte nach Modelländerungen adaptieren zu müssen.

Aufgrund der beschriebenen Problematik wird in Kapitel 5 ein Ansatz zur Auswirkungsanalyse eingeführt. Dieser ermöglicht es die Auswirkungen von Modelländerungen explizit unter Nutzung einer DSL zu modellieren. Aufbauend auf den modellierten Auswirkungen erzeugt ein Generator für eine Menge von Modelländerungen eine Checkliste, die Aufschluss über die Auswirkungen der Modelländerungen gibt. Konkret führt eine Checkliste auf, welche Entwicklungsaktivitäten aufgrund der ermittelten Modelländerungen durchzuführen sind. Die Checklisten vereinfachen somit die Evolution eines Softwaresystems. Die Anwendung des Ansatzes ist nachfolgend beschrieben:

1. Die Entwickler modellieren für ausgewählte Problembereiche die Auswirkungen von Modelländerungen.
2. Nachdem das nächste Mal Modelländerungen durchgeführt wurden, wird das Differenzberechnungswerkzeug ausgeführt, um die vorgenommenen Modelländerungen zu ermitteln.
3. Anschließend wird der Checklistengenerator für diese Modelländerungen aufgerufen. Dieser prüft anhand der im ersten Schritt modellierten Auswirkungen, welche Modelländerungen welche Auswirkungen haben. Als Ergebnis erzeugt der Generator eine Checkliste, die diese Auswirkungen kapselt.
4. Die Entwickler arbeiten mit der generierten Checkliste und haken die durchgeführten Tätigkeiten ab.

2.4 Generierung von Migrationsfunktionalität zur Daten Evolution

RQ2: Wie kann die Kenntnis von Modelländerungen im weiteren Verlauf der Evolution eines Softwaresystems effizient genutzt werden (C2, C3)?

RQ2.2: Wie können in einem Softwaresystem, in dem konstruktive Datenmodelle eingesetzt werden, automatisch passende Datenmigrationen auf Basis von Modelländerungen erzeugt werden (C3)?

Ein wesentlicher Bestandteil bei der Modellierung eines Softwaresystems ist die Modellierung eines Datenmodells. Zu diesem Zweck können verschiedene Modellierungstechniken eingesetzt werden, wie Entity Relationship Modelle [Che75, Che02], Klassendiagramme [HB99] oder DSLs [VWV11].

Sobald konstruktive Datenmodelle modelliert werden, ist die Evolution dieser Modelle von besonderer Bedeutung für die Software Evolution. Der Grund hierfür ist, dass sich Modelländerungen auf die Struktur der Daten auswirken und somit die Bereitstellung von Datenmigrationen bedingen können. Ohne die Durchführung von Datenmigrationen können die Daten, die zu einer vorherigen Modellversion konform sind, nicht weiter verwendet werden. Dies ist bei dem produktiven Einsatz von Softwaresystemen nicht akzeptabel, da die gespeicherten Daten ein zentrales Gut in diesen Softwaresystemen darstellen. Folglich besteht eine Notwendigkeit zur Durchführung von Datenmigrationen. Ein primäres Ziel dieser Datenmigrationen ist es die bestehenden Daten möglichst verlustfrei zu erhalten, da Datenverluste zur Unzufriedenheit der Benutzer der Softwaresysteme führen können.

Bei DSA wird das Datenmodell über Klassendiagramme modelliert. Der Aspekt der Daten Evolution ist für Authoring im Kontext der Client-Server Kommunikation sowie im Bereich der Server-Datenbank relevant. Im Folgenden liegt der Schwerpunkt auf der Betrachtung der Client-Server Kommunikation.

Die Clients und die Server von Authoring können weltweit verteilt sein. Zwischen den Clients und den Servern werden bisher serialisierte Instanzen der handgeschriebenen Entitätsklassen ausgetauscht. Diese sind, wie in Abbildung 2.2 auf Seite 13 zu sehen, von den generierten Entitätsklassen abgeleitet. Das hat zur Folge, dass sich Modelländerungen auf die generierten Entitätsklassen und somit auch auf die handgeschriebenen Entitätsklassen auswirken können. Zur Serialisierung und Deserialisierung der konkreten Instanzen von Entitätsklassen wird bisher der Standard Java Serialisierungs- und Deserialisierungsmechanismus eingesetzt.

Ein Problem bei dieser Herangehensweise ist, dass die Daten eines Clients auf einem Server oder die Daten eines Servers auf einem Client nur dann vollständig und verlustfrei verarbeitet werden können, wenn die Entitätsklassen des Clients strukturell zu denen des Servers korrespondieren. In einigen Situationen ist dieser Ansatz jedoch zu restriktiv. Ein Beispielszenario hierfür ist in Abbildung 2.4 dargestellt. In diesem Beispiel empfängt ein Server in Version v_n von einem älteren Client in Version v_a Instanzen der Klasse ECU, die Steuergeräte repräsentiert. Hierbei wird angenommen, dass von Version v_a zu Version v_n lediglich ein Attribut der Klasse ECU umbenannt wurde. In diesem Fall ist es für den Server problemlos möglich die serialisierten Daten zu verarbeiten, da die Daten aus v_a unter Einbeziehung der Umbenennung auf einfache Weise in die Version v_n überführt werden können. In Abbildung 2.4 wird das in der ersten Variante veranschaulicht.

In dem beschriebenen Szenario ist es allerdings so, dass der Server die Umbenennung bei der Deserialisierung der Daten nicht beachtet. Somit wird das umbenannte Attribut bei der Deseria-

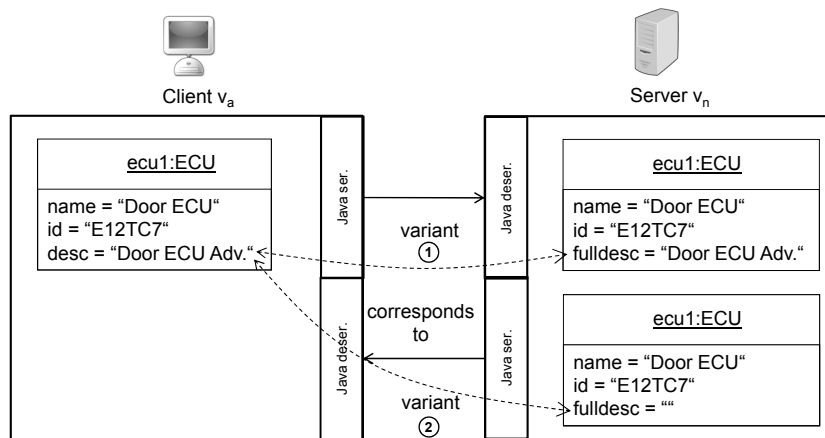


Abbildung 2.4: Kommunikation zwischen Client (Version v_a) und Server (Version v_n). Variante 1 (Soll-Situation): Server erkennt Umbenennung und übernimmt Daten aus umbenanntem Attribut. Variante 2 (Ist-Situation): Server erkennt Umbenennung nicht und verliert Daten zu Attribut-Wert.

lisierung nicht mit Daten gefüllt, selbst wenn der Serialisierungsschritt ursprünglich Daten für das Attribut serialisiert hat. In Abbildung 2.4 entspricht das der zweiten Variante der Datenübertragung. Infolgedessen würden Benutzer von Authoring mit unvollständigen Daten arbeiten. Da dies zu unerwarteten Fehlern bei der Bedienung von Authoring führen kann, wird diesem Szenario bisher entgegengewirkt, indem Clients nur mit Servern kommunizieren können, die Daten in der gleichen Version wie die Clients verarbeiten. Somit ist eine Client-Version bisher immer an eine Server-Version gebunden.

In der Praxis stellt diese Festlegung jedoch eine Einschränkung dar. Wie zuvor bereits erläutert, können die Clients und die Server weltweit auf verschiedene Standorte verteilt sein, wobei die Clients eines Kunden üblicherweise gegen genau einen zentralen Server arbeiten. Die Clients in einem Standort können folglich nicht aktualisiert werden, solange andere Standorte des gleichen Kunden noch eine ältere Version des Clients einsetzen. Möchte ein Kunde beispielsweise die Client-Version erst an spezifischen Standorten aktualisieren und im produktiven Betrieb testen, bevor die Version ebenfalls an anderen Standorten aktualisiert wird, ist dies mit der aktuellen Herangehensweise nicht direkt möglich.

Eine Möglichkeit um dieses Problem anzugehen besteht darin, die standardmäßig genutzte Serialisierungs- und Deserialisierungslogik im Falle von Modelländerungen zu überschreiben und eine angepasste Logik bereitzustellen. Konkret kann diese Logik beispielsweise so implementiert werden, dass die Werte zu den Attributen in einer bestimmten Reihenfolge in den Datenstrom geschrieben und bei der Deserialisierung auch wieder gelesen werden. Umbenennungen sind dadurch unproblematisch, da die Namen der Attribute bei dieser Strategie irrelevant sind. Infolgedessen können die Daten in dem aufgeführten Szenario verlustfrei deserialisiert werden. Bei dieser Vorgehensweise sind zwar Umbenennungen unkritisch, andere Arten von Modelländerungen können jedoch wiederum zu Problemen führen. Beispielsweise kommt es zu Datenverlusten, wenn ein einzelnes Attribut in eine Liste umgewandelt oder wenn

ein Attribut verschoben wird. Um auch mit diesen und weiteren Modelländerungen umgehen zu können, müssen Datenmigrationen durchgeführt werden. Diese beinhalten, wie bestehende Daten möglichst verlustfrei in ein neues Format überführt werden können. Unabhängig davon, welche konkrete eigene Serialisierungsstrategie oder Datenmigrationsstrategie eingesetzt wird, müsste diese Logik momentan manuell implementiert werden. Mit Hinblick auf die Größe und Komplexität der Systeme und der Modelle ist dies allerdings sehr zeitaufwändig.

Weiterhin gilt es zu beachten, dass ein Authoring-Server gegen eine Datenbank arbeitet. Sobald die Version des Servers aktualisiert werden soll, müssen folglich Datenmigrationen auf der Datenbank durchgeführt werden. Bisher werden diese Datenmigrationen manuell in SQL implementiert.

Um bei der Client-Server Kommunikation mehr Flexibilität bei Datenmigrationen zu haben, wird die Client-Server Kommunikation auf JSON¹ [Bra14] umgestellt. Parallel zu dem Projekt, in dessen Kontext diese Arbeit durchgeführt wurde, wird zudem untersucht, ob die Datenbank auf eine JSON-Datenbank umgestellt werden kann. Der große Vorteil einer einheitlichen Umstellung auf JSON ist, dass dann lediglich einmal Migrationsskripte für JSON-Dokumente bereitgestellt werden müssen und diese in beiden Fällen wiederverwendet werden können.

Zur Reduktion des manuellen Implementierungsaufwands werden die Datenmigrationen in dieser Arbeit generiert. In Kapitel 6 wird dargelegt, wie diese Generierung umgesetzt wird. Dazu wird ein Konzept vorgestellt, welches beschreibt, wie für die verschiedenen Arten von Modelländerungen Datenmigrationen durchgeführt werden können, so dass die Daten möglichst verlustfrei migriert werden. Ferner wird erläutert, wie vorgegangen wird, um die Migrationsfunktionalität zu generieren. Der resultierende Ansatz gliedert sich wie folgt in den Entwicklungsablauf der Entwickler ein:

1. Die Modellierer verändern die bestehenden Datenmodelle, beispielsweise aufgrund von geänderten Anforderungen.
2. Das Differenzberechnungswerkzeug wird ausgeführt, um die Modelländerungen zu ermitteln.
3. Der Generator für Datenmigrationen analysiert die vorgenommenen Änderungen und generiert Migrationsfunktionalität für sämtliche Modelländerungen, die eine Datenmigration erfordern.
4. Die Entwickler erweitern die generierten Migrationsklassen, für den Fall, dass weitere Datenmigrationen erforderlich sind. Dieser Schritt ist beispielsweise notwendig, wenn Datenkorrekturen erforderlich sind.
5. Die Entwickler führen die generierten Migrationsklassen auf den bestehenden Daten aus.

¹JSON = Java Script Object Notation

2.5 Modellierung von Generatorvariabilität

RQ3: Wie kann die Konstruktion von Generatorvarianten vereinfacht werden (C2, C4)?

Zur Umsetzung eines Generators existieren verschiedene Techniken, wie Visitor-basierte Generatoren oder Template-basierte Generatoren [CH03, CH06]. Bei Visitor-basierten Generatoren wird die interne Repräsentation des Eingabemodells von einem Visitor-Mechanismus [GHJV95] traversiert und dabei wird Text in einen Ausgabestrom geschrieben. Template-basierte Generatoren hingegen basieren darauf, dass der zu generierende Code in Templates gekapselt ist. Templates enthalten im Wesentlichen den zu generierenden Code der Zielsprache sowie zusätzliche Anweisungen der Templatesprache, z.B. zur Navigation innerhalb der internen Repräsentation des Eingabemodells. Komplexere Funktionalität wird üblicherweise in eingebettete Hilfsklassen [Fow10], im Folgenden abgekürzt als Hilfsklassen, ausgelagert. Diese Hilfsklassen werden normalerweise in objektorientierten Programmiersprachen implementiert und können innerhalb von Templates aufgerufen werden. Der Vorteil der Auslagerung komplexer Logik in Hilfsklassen ist, dass dadurch die Lesbarkeit von Templates verbessert wird und die ausgelagerte Logik in mehreren Templates verwendet werden kann. Der Generator bei DSA ist ein Template-basierter Generator, daher wird im weiteren Verlauf der Arbeit angenommen, dass zur Generierung Template-basierte Generatoren eingesetzt werden.

Viele Generatoren werden bisher als monolithische Systeme entwickelt. Der Einsatz solcher Generatoren für andere Anwendungsfälle, in denen ähnliche aber leicht angepasste Anforderungen an den generierten Code existieren, ist ohne weiteres in der Regel nicht möglich. Der Grund dafür ist, dass die monolithische Struktur der Generatoren eine gezielte Adaption von Generatoranteilen erschwert oder gar unmöglich macht. Im Falle von Template-basierten Generatoren würde sich eine gezielte Adaption darüber äußern, dass dedizierte Teile ausgewählter Templates oder Hilfsklassen in der neuen Generatorvariante auf andere Weise umgesetzt werden würden.

Ein simpler Ansatz zur Handhabung dieser Problematik besteht darin, einen bestehenden Generator zu kopieren und Änderungen an dieser Kopie vorzunehmen, falls eine neue Variante des bestehenden Generators entwickelt werden soll. Diese Art der informellen Wiederverwendung [Jör13a] kann zwar auf jede Art von Generator angewendet werden, besitzt aber mit Hinblick auf die Weiterentwicklung und Wartung von Generatoren gravierende Schwachstellen. Das Hauptproblem bei dieser Herangehensweise ist, dass Änderungen, die in einer Variante eines Generators eingebaut werden, für alle anderen Varianten des Generators relevant sein können. Die Entwickler müssen somit sicherstellen, dass die Änderungen in allen anderen erforderlichen Varianten nachgepflegt werden. Derartige Anpassungen sind besonders kritisch, wenn Fehler in einem Generator behoben wurden. Darüber hinaus können solche Anpassungen sehr aufwändig sein, da im Laufe der Zeit nicht gesichert werden kann, dass die Änderungen überall konsistent nachgetragen werden.

Eine Alternative dazu besteht in dem Einsatz spezifischer Generatorframeworks, die teilweise eine Adaption von Generatoranteilen ermöglichen, z.B. Acceleo [Acc] oder XTend [Xtea]. Ein wesentlicher Nachteil an dieser Alternative ist, dass die Generatorentwickler auf ein spezifisches Generatorframework festgelegt sind. Während dies bei Neuentwicklungen gegebenenfalls

noch praktikabel ist, führt dies bei bereits entwickelten Generatoren zu Problemen, falls diese mit einem anderen Generatorframework implementiert wurden. In diesem Fall müssen die bestehenden Generatoren in das neue Generatorframework migriert werden, was mit sehr hohem Aufwand und dadurch mit sehr hohen Kosten verbunden sein kann.

Der Generator bei DSA wurde mit Hilfe des Template-basierten Generatorframeworks open-ArchitectureWare [HVEK07] entwickelt, welches als Templatesprache XPand [XPa] einsetzt. Die verwendete Version des Generatorframeworks unterstützt keine Mechanismen zur Integration von Variabilität in Generatoren. Zudem kommt ein Umstieg auf ein komplett anderes Generatorframework aufgrund des damit verbundenen Aufwands nicht in Frage.

Angesichts der mit den zuvor dargelegten Alternativen verbundenen Nachteile wird in Kapitel 7 ein Ansatz vorgestellt, in dem die variablen Anteile eines Generators explizit modelliert werden können. Dieser Ansatz ist nicht auf ein spezifisches Generatorframework beschränkt. Die Entwicklung von Generatorvarianten wird dabei durch ein Konzept ermöglicht, durch das eine Generatorvariante eine andere Generatorvariante gezielt verändern kann. Auf diese Weise kann beispielsweise ein Teil eines Templates durch neuen Inhalt ersetzt werden. Zur Definition der gewünschten Änderungsoperationen können die Entwickler eine für diesen Zweck entwickelte DSL nutzen. Die Artefakte der resultierenden Generatorvariante werden automatisch unter Einbeziehung der ursprünglichen Generatorvariante und der definierten Änderungsoperationen neu erzeugt. Der Ansatz kann wie folgt angewendet werden. Dabei wird vorausgesetzt, dass eine neue Variante eines Generators entwickelt werden soll.

1. Die Entwickler analysieren den bestehenden Generator. Sie identifizieren die Teile des Generators, die für die neue Generatorvariante angepasst werden müssen.
2. Anschließend modellieren die Entwickler, auf welche Weise die identifizierten Generatorteile für die neue Generatorvariante angepasst werden sollen. Dies kann sowohl den Fall umfassen, dass bestehende Teile komplett ersetzt als auch das neue Logik gezielt vor oder hinter bestehende Generatorteile eingefügt werden soll.
3. Anschließend definieren die Entwickler, aus welchen Teilen des bestehenden Generators und aus welchen Erweiterungen bzw. angepassten Teilen des Generators die neue Generatorvariante bestehen soll.
4. Die Artefakte der neuen Generatorvariante werden abschließend automatisch erzeugt. Die Entwickler können die resultierende Generatorvariante daraufhin unmittelbar einsetzen.

2.6 Modellbasierte Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen

RQ4: Wie kann dem Problem entgegengewirkt werden, dass die Implementierung eines Plugin-basierten Softwaresystems von der Softwarearchitektur des Softwaresystems abweicht (C4, C5)?

Modularisierung ist eines der wichtigsten Prinzipien der Softwareentwicklung [Par72], da eine Aufteilung in Module die Austauschbarkeit der Module vereinfacht [Sch12]. Plugin-basierte Softwaresysteme folgen dieser Idee und teilen die Gesamtfunktionalität eines Softwaresystems auf einzelne Plugins auf, die jeweils eine Menge von Klassen sowie weitere benötigte Ressourcen, wie z.B. Property-Dateien, umfassen. Darüber hinaus wird davon ausgegangen, dass ein Plugin-basiertes Softwaresystem vorliegt, in dem Features eine zusammengehörige Gruppe von Plugins kapseln. Der große Vorteil dieses Ansatzes ist, dass mehrere Softwaresysteme Features und Plugins wiederverwenden können und das bestehende Softwaresysteme auf einfache Weise durch die Bereitstellung neuer Features und Plugins erweitert werden können.

Die Einteilung der Softwaresysteme in Features und Plugins ist ein wichtiger Bestandteil der Softwarearchitektur, die ein Softwaresystem in Komponenten strukturiert und definiert, welche Komponenten wie miteinander interagieren [PW92, GS94, BR07, MMR10]. Trotz der elementaren Bedeutung von Softwarearchitekturen werden Architekturbeschreibungen jedoch nur in seltenen Fällen gepflegt, falls die Softwarearchitektur überhaupt explizit dokumentiert wird [KP07].

Das Hauptproblem bei der Pflege der Architekturbeschreibungen ist, dass es in der Regel sehr zeitaufwändig ist manuell sicherzustellen, dass die Implementierung eines Softwaresystems konsistent zu der Softwarearchitektur ist. Im Zuge der Evolution eines Softwaresystems muss diese Prüfung zudem nicht nur einmalig, sondern kontinuierlich begleitend zur Weiterentwicklung des Softwaresystems durchgeführt werden. Im Falle eines Plugin-basierten Softwaresystems muss beispielsweise stets sichergestellt werden, dass die Features und Plugins nur mit dedizierten anderen Features und Plugins interagieren. In Plugin-basierten Softwaresystemen mit einer hohen Anzahl von Features und Plugins kann dies ohne gezielte Werkzeugunterstützung manuell nicht adäquat sichergestellt werden. Abweichungen zwischen der Softwarearchitektur und der Implementierung, als Architektur-Erosion bezeichnet [PW92], sollten jedoch vermieden werden, da sie sich negativ auf die Qualität, die Wartbarkeit und die Weiterentwickelbarkeit eines Softwaresystems auswirken [KP07, TVCB12, HR13]. Die Wiederverwendung von Features und Plugins führt indes dazu, dass sich eine Abweichung zwischen Softwarearchitektur und Implementierung negativ auf mehrere Softwaresysteme auswirken kann.

Um diesem Problem entgegenzuwirken, wurden eine Vielzahl verschiedener Ansätze zur automatischen Prüfung der Konformität zwischen Implementierung und Softwarearchitektur entwickelt [KP07, PTV⁺10]. Einige dieser Ansätze basieren auf der Definition von Architekturregeln, die von der Implementierung erfüllt werden müssen, andere definieren hingegen Architekturregeln, gegen welche die Implementierung nicht verstoßen darf [SJSJ05, TV08, TV09]. Ferner existieren Ansätze, die voraussetzen, dass die erwartete Softwarearchitektur definiert und die bestehende Softwarearchitektur automatisch aus dem Quellcode des Softwaresystems extrahiert wird [MNS01, DHHJ10]. Sobald eine Abweichung zwischen Ist- und Referenzarchitektur erkannt wird, werden die Entwickler darüber in Kenntnis gesetzt, so dass diese Abweichungen wieder aufgelöst werden können. Diese bestehenden Ansätze sind jedoch nicht dediziert auf die Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen ausgelegt. In solchen Systemen müssen Abhängigkeitsrestriktionen bezüglich der Abhängigkeiten zwischen Features und Plugins definiert und geprüft werden können.

Die betrachteten Softwaresysteme bei DSA sind allesamt RCP-Anwendungen. Zu vielen Fea-

tures und Plugins dieser RCP-Anwendungen existieren informelle Beschreibungen dazu, welche Features und Plugins mit welchen anderen Features und Plugins interagieren dürfen. Das Hauptproblem in diesem Kontext ist, dass nicht automatisch geprüft wird, ob diese Vorgaben eingehalten werden. Darüber hinaus betreffen die Beschreibungen nicht alle Teile der Softwaresysteme. Die Folge davon ist, dass die Entwickler und Systemarchitekten die Vorgaben bisher manuell prüfen müssen. Aufgrund der hohen Komplexität der Systeme ist diese manuelle Prüfung aber sehr zeitaufwändig.

Aufgrund der geschilderten Problematik wird in Kapitel 8 ein Ansatz zur Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen eingeführt. Dieser umfasst sowohl eine Modellierungssprache zur Definition von Abhängigkeitsrestriktionen als auch eine Prüfkomponente, welche die Konformität eines Softwaresystems zu diesen Abhängigkeitsrestriktionen automatisch prüft. Der große Vorteil dieser Prüfkomponente ist, dass sie in die Entwicklungsumgebung integriert werden kann, in der die Softwaresysteme implementiert werden. Die Prüfung auf die Architekturkonformität kann auf diese Weise kontinuierlich während der Weiterentwicklung der Softwaresysteme durchgeführt werden. Dadurch können Entwickler unmittelbar darauf hingewiesen werden, wenn eine Änderung, die sie in der Implementierung vorgenommen haben, gegen die Architekturvorgaben verstößt. Die Schritte zur Anwendung dieses Ansatzes sind wie folgt:

1. Die Entwickler integrieren die Prüfkomponente in die Entwicklungsumgebung, in der das Softwaresystem entwickelt wird.
2. Die Systemarchitekten modellieren iterativ eine Referenzarchitektur für das betroffene Softwaresystem. Diese drückt aus, welche Plugins und Features voneinander abhängen dürfen und welche Abhängigkeiten verboten sind. Die Prüfkomponente prüft die Konformität der Implementierung eines Softwaresystems gegen diese Referenzarchitektur.
3. Ändert ein Entwickler die Abhängigkeiten eines Plugins, wird die Prüfkomponente automatisch ausgeführt und die Änderungen des Entwicklers werden gegen die Referenzarchitektur geprüft.
4. Verstoßen die Änderungen des Entwicklers gegen die Referenzarchitektur, werden dem Entwickler unmittelbar Fehler oder Warnungen angezeigt. Der Entwickler kann die Änderungen daraufhin wieder rückgängig machen oder die Implementierung auf andere Weise so anpassen, dass die unerwünschten Abhängigkeiten nicht mehr bestehen.
5. Werden nach der Einführung des Ansatzes bereits Verstöße identifiziert, die nicht durch neue Änderungen entstanden sind, werden diese Verstöße nach Rücksprache mit den Systemarchitekten ebenfalls entfernt.

Kapitel 3

Grundlagen der DSL-Entwicklung

Im Rahmen dieser Arbeit wird die Language Workbench Xtext [Xteb, Bet13] zur Entwicklung von textuellen DSLs eingesetzt. Der Grund hierfür ist, dass DSA Xtext zur Entwicklung textueller DSLs nutzt. Alternativ zu Xtext gibt es aber eine Vielzahl weiterer Language Workbenches zur Entwicklung textueller DSLs [Mer10, EvdSV⁺13] wie beispielsweise MontiCore [KRV08, Kra10, KRV10] und Spoofax [KV10, Spo].

In diesem Kapitel werden die wesentlichen Xtext-Grundlagen vermittelt, die zum Verständnis der entwickelten DSLs benötigt werden. Generelle Herausforderungen bei der Entwicklung von DSLs und der Nutzung von Modellierungstechnologien werden in [FR07] diskutiert.

Mit Xtext entwickelte DSLs basieren auf einer erweiterten kontextfreien Grammatik [ASU86], welche die konkrete Syntax der DSL festlegt. Aufbauend auf der Grammatik wird in Xtext ein ECore [Ecl]-Modell generiert, welches die abstrakte Syntax der Sprache repräsentiert. Weiterhin wird aus der Grammatik unter anderem ein Lexer und ein Parser sowie eine Editor-Implementierung generiert. Intern verwendet Xtext als Lexer- und Parser-Generator ANTLR [PQ95].

Zur Verarbeitung eines konkreten Modells einer Sprache erzeugt der generierte Lexer aus dem Eingabestrom für das Modell eine Liste von Tokens. Anschließend verarbeitet der generierte Parser diese Liste von Tokens und erzeugt einen abstrakten Syntaxbaum (engl. abstract syntax tree, AST). Dieser AST kann darauffolgend von einem Generator oder anderen Komponenten verarbeitet werden. Abbildung 3.1 verdeutlicht diesen Ablauf grafisch.

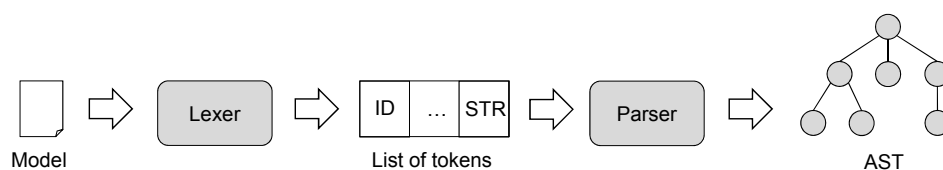


Abbildung 3.1: Modellverarbeitung durch Lexer und Parser.

Listing 3.1 zeigt anhand eines Beispiels die wichtigsten Bestandteile einer Xtext-Grammatik. Diese ist ein abgewandelter Auszug der Grammatik, welche in Abschnitt 8.3 eingeführt wird. Die Bestandteile dieser Grammatik werden im Folgenden von oben nach unten erläutert.

Zu Beginn jeder Grammatik wird ihr Name definiert (Zeile 1). Die vollqualifizierte Angabe des Grammatiknamens führt dazu, dass die Artefakte zu der DSL in das entsprechende Paket generiert werden. Darüber hinaus wird in Zeile 1 festgelegt, dass die Grammatik `org.eclipse.xtext.common.Terminals` wiederverwendet wird. Diese Wiederverwen-

```

1 grammar rwth.DepCoL with org.eclipse.xtext.common.Terminals
2
3 DependencyModel:
4     featureBase = FeatureBase
5     featureRefs += FeatureRef*
6 ;
7
8 FeatureBase:
9     'featurebase {'
10         (availableFeatures += FeatureName ';'')+
11     '}'
12 ;
13
14 FeatureName:
15     name = ID
16 ;
17
18 FeatureRef:
19     'feature' feature = [FeatureName | ID] '{'
20         dependencies += Dependency+
21     '}'
22 ;
23
24 Dependency:
25     severity = ('[critical]' | '[error]')?
26     'forbid to' toFeature = [FeatureName | ID] ';'
27 ;

```

Listing 3.1: Beispiel für eine Xtext-Grammatik.

dition erlaubt es Produktionsregeln der wiederverwendeten Grammatik zu nutzen. Ferner können Produktionsregeln der wiederverwendeten Grammatik überschrieben werden.

Den Kernbestandteil jeder Xtext-Grammatik bilden die darin enthaltenen Produktionsregeln. Jede Produktionsregel definiert ein Nichtterminalsymbol auf der linken Seite der Regel. Die rechte Seite der Regel kann auf Nichtterminalsymbole und Terminalsymbole verweisen. Ein Verweis auf ein Nichtterminalsymbol auf der rechten Seite führt dazu, dass im AST entsprechende Knoten für das Nichtterminalsymbol angelegt werden. Die linke und rechte Seite einer Regel sind in Xtext über einen Doppelpunkt voneinander getrennt.

Die Grammatik in Listing 3.1 definiert in Zeile 3 – 6 das Nichtterminalsymbol `DependencyModel`. Dieses verweist auf die Nichtterminalsymbole `FeatureBase` (Zeile 4) sowie `FeatureRef` (Zeile 5). Der Verweis auf `FeatureBase` ist verpflichtend, da keine Kardinalität explizit angegeben ist. Demgegenüber können beliebig viele Verweise auf `FeatureRef` vorliegen, ausgedrückt über den Stern hinter dem Verweis (Zeile 5). Beide Verweise sind benannt, so dass im AST von einem `DependencyModel` Knoten über den Namen zu den Kindknoten navigiert werden kann. So besitzt der Verweis auf `FeatureBase` den Namen `featureBase` (Zeile 4) und der Verweis auf `FeatureRef` den Namen `featureRefs` (Zeile 5). Xtext unterscheidet verschiedene Arten von benannten Zuweisungen:

- Die direkte Zuweisung „`=`“ drückt aus, dass dem Namen auf der linken Seite genau ein Element auf der rechten Seite zugeordnet wird.
- Die Zuweisung „`+=`“ drückt aus, dass der Name auf der linken Seite auf mehrere Elemente der rechten Seite verweisen kann.
- Die Zuweisung „`?`“ entspricht einem booleschen Zuweisungsoperator. Dieser ordnet dem Namen auf der linken Seite einen booleschen Wert zu, der ausdrückt, ob das Element auf der rechten Seite vorliegt oder nicht.

Anschließend wird das Nichtterminalsymbol `FeatureBase` eingeführt (Zeile 8 – 12). In den Zeilen 9 und 11 wird die konkrete Syntax der Sprache festgelegt. In Xtext werden dazu Zeichenketten in Hochkommata eingeschlossen. Somit muss in einem konkreten Modell zu Beginn die Zeichenkette „`featurebase {`“ stehen. Ferner drückt die Grammatik aus, dass mindestens ein Verweis auf das Nichtterminalsymbol `FeatureName` vorliegen muss, ausgedrückt durch das Zeichen „`+`“ hinter dem Verweis (Zeile 10).

Darauffolgend wird das Nichtterminalsymbol `FeatureName` definiert (Zeile 14 – 16). Dieses verweist auf das Nichtterminalsymbol `ID` (Zeile 15), welches in der Grammatik `org.eclipse.xtext.common.Terminals` definiert ist. Bei diesem handelt es sich um ein spezielles Nichtterminalsymbol, welches durch eine Lexerregel erzeugt wird. Lexerregeln werden durch den Lexer verarbeitet und beeinflussen, welche Tokens für den Eingabestrom erzeugt werden. Listing 3.2 zeigt die Definition der Lexerregel `ID`. Diese sagt aus, dass das Token `ID` für eine Zeichenkette erzeugt wird, die mit einem optionalen „`^`“ Zeichen beginnt, gefolgt von einem Kleinbuchstaben, einem Großbuchstaben oder einem Unterstrich. Im Anschluss daran können beliebig viele weitere Buchstaben, Ziffern und Unterstriche folgen.

```

1 terminal ID: '^'? ('a'..'z'|'A'..'Z'|'_')
2   ('a'..'z'|'A'..'Z'|'_'|'0'..'9') *;
```

Listing 3.2: Lexerregel `ID` der Xtext-Grammatik `org.eclipse.xtext.common.Terminals`.

Neben der Lexerregel `ID` wird in dieser Arbeit die Lexerregel `STRING` verwendet, die in Listing 3.1 allerdings nicht genutzt wird. Diese kennzeichnet eine Zeichenkette, die in Anführungszeichen gekapselt ist.

Die Produktionsregeln in Listing 3.1 werden durch den Parser verarbeitet und werden in Xtext folglich Parserregeln genannt. Der wesentliche Unterschied zu Lexerregeln ist, dass sie den Aufbau des ASTs festlegen. So existiert in einem AST, der auf Basis der gezeigten Grammatik erzeugt wird, beispielsweise ein Knoten `DependencyModel`, der einen Kindknoten `FeatureBase` besitzt und mehrere Kindknoten `FeatureRef` besitzen kann.

Als Nächstes wird in der Grammatik das Nichtterminalsymbol `FeatureRef` definiert (Zeile 18 – 22). Dieses verdeutlicht die Verwendung eines Querverweises in Xtext. Ein Querverweis wird über eine eckige Klammer gekennzeichnet und der Inhalt der Klammern verweist auf ein Nichtterminalsymbol. In der Definition von `FeatureRef` liegt ein Querverweis auf `FeatureName` vor (Zeile 19). Die Bedeutung von Querverweisen wird im Folgenden präzisiert.

Wie zuvor erläutert, baut der Xtext-Parser einen AST auf. Über Querverweise ist es möglich namensbasiert einen einzelnen Knoten dieses ASTs zu referenzieren. Dies wird anhand des Beispiels aus Listing 3.1 verdeutlicht. Gemäß dieser Grammatik werden initial Informationen zum Nichtterminalsymbol `FeatureBase` definiert. Über diese Definition werden in dem AST konkrete `FeatureName` Knoten eingeführt, da `FeatureBase` einen Verweis auf `FeatureName` besitzt. Der Querverweis in Zeile 19 drückt aus, dass ein existierender `FeatureName` Knoten referenziert werden muss. Das hat zur Folge, dass die referenzierte `FeatureName` Instanz zuvor über `FeatureBase` eingeführt werden muss.

Die konkrete Notation zur Angabe von Querverweisen ist in zwei Teile gegliedert. Die Angabe vor dem Zeichen „|“ drückt aus, auf welches Nichtterminalsymbol querverwiesen wird. Der zweite Teil des Querverweises, die Angabe nach „|“, wird angegeben, um die Syntax der zu parsenden Zeichenkette zu spezifizieren. Der Name einer `FeatureName` Instanz wird durch die Lexerregel `ID` festgelegt (Zeile 15). Folglich referenziert der Querverweis in Zeile 19 diese Lexerregel im zweiten Teil des Querverweises.

Als Letztes wird das Nichtterminalsymbol `Dependency` definiert (Zeile 24 – 27). Hier drückt die rechte Seite der Produktion aus, dass zu Beginn die Zeichenkette „[critical]“ oder die Zeichenkette „[error]“ stehen kann (Zeile 25). Diese Alternative wird in Xtext über das Zeichen „|“ ausgedrückt. Hervorzuheben hierbei ist, dass „|“ in Querverweisen eine andere Bedeutung hat, wie zuvor erläutert. Darüber hinaus enthält auch die Definition von `Dependency` einen Querverweis auf `FeatureName` (Zeile 26).

Abschließend ist in Listing 3.3 ein Modell gezeigt, welches konform zu der in diesem Kapitel eingeführten Grammatik ist. In diesem Modell werden in `FeatureBase` drei Features `f1`, `f2` und `f3` definiert. Darüber hinaus liegen zwei `FeatureRef` Instanzen vor, wobei die erste auf zwei und die zweite auf eine `Dependency` Instanz verweist.

```
1 featurebase {
2     f1;
3     f2;
4     f3;
5 }
6
7 feature f1 {
8     [critical] forbid to f2;
9     [critical] forbid to f3;
10 }
11
12 feature f3 {
13     [error] forbid to f1;
14 }
```

Listing 3.3: Beispielhaftes Modell, welches konform zu der Grammatik aus Listing 3.1 ist.

Kapitel 4

Ermittlung von Modellunterschieden

Die Ermittlung von Modellunterschieden ist eine grundlegende Voraussetzung für die Unterstützung der Software Evolution in der modellgetriebenen Softwareentwicklung. Der Hauptgrund dafür ist, dass die eingesetzten Modelle im Laufe der Zeit kontinuierlich angepasst werden müssen. Infolgedessen muss geprüft werden, ob die Modellierer genau die geforderten Modeländerungen durchgeführt haben. Diese ergeben sich üblicherweise aus neuen oder geänderten Anforderungen an das Softwaresystem. Weiterhin vereinfacht die Ermittlung von Modellunterschieden die Analyse der Modell Evolution, die Aufschluss über die Historie spezifischer Modellelemente gibt [WK08]. Darüber hinaus ermöglicht die Unterschiedsberechnung das Patchen von Modellen [KKK13] sowie die Zusammenführung von Modellen (engl. Model Merging) [BCE⁺06, Men02]. Ein Patch enthält Editieroperationen, die gezielt auf ein Modell angewandt werden können. Bei der Zusammenführung von Modellen werden die Modellinformationen aus verschiedenen Modellen abgeglichen und in ein integriertes Modell überführt [EPK06]. Eine Grundvoraussetzung hierfür ist, dass ermittelt werden kann, welche Elemente der Modelle übereinstimmen und welche verschieden sind.

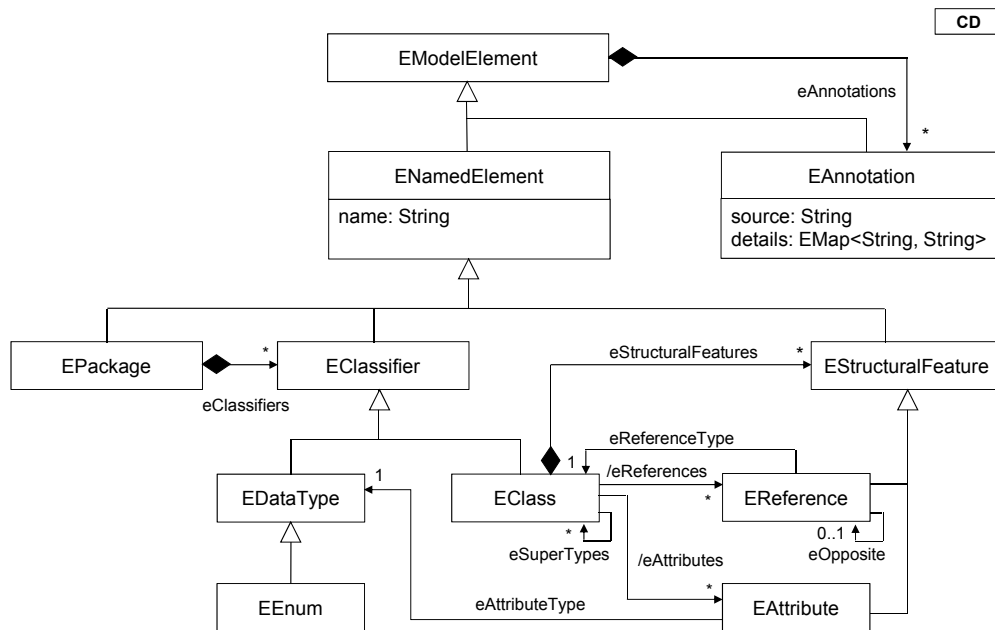
In dieser Arbeit bildet die Ermittlung von Modellunterschieden die Grundlage für die Generierung von Checklisten (in Kapitel 5 beschrieben) und für die Generierung von Migrationsfunktionalität zur Daten Evolution (in Kapitel 6 beschrieben). Das Kapitel baut auf den Publikationen [PMR13, MR14b] auf.

In Abschnitt 4.1 werden zunächst die Grundlagen für die Berechnung von Modellunterschieden vorgestellt. Ferner wird ein Überblick über den Stand der Technik gegeben. Im Anschluss stellt Abschnitt 4.2 eine Methodik zur Berechnung von Modellunterschieden vor, welche die Vorgehensweise zur Ermittlung von Modellunterschieden in dieser Arbeit darlegt.

4.1 Grundlagen und Stand der Technik

Das Eclipse Modeling Framework (EMF) [Ecl] ist ein Modellierungsframework, welches Möglichkeiten zur Code-Generierung bietet und zur Konstruktion von Anwendungen dient, die auf strukturierten Datenmodellen basieren [Ecl, SBPM09]. Eine Besonderheit von EMF ist, dass es Java, XML und Klassendiagramme vereint, da ein Modell in einer dieser drei Formen definiert werden kann und mittels EMF die zugehörigen Repräsentationen in den anderen Formen generiert werden können.

Den Kern von EMF bildet das Metamodell ECore zur Beschreibung von Datenmodellen. Ein vereinfachter und gekürzter Auszug aus dem ECore-Metamodell ist in Abbildung 4.1 dargestellt.



Im weiteren Verlauf dieser Arbeit werden die Assoziationen der Metaklassen, die keine Komposition darstellen, z.B. `eSuperTypes` der Metaklasse `EClass`, als Metaassoziationen bezeichnet. Weiterhin werden die Attribute der Metaklassen, z.B. `name` der Metaklasse `ENamedElement`, als Metaattribute bezeichnet. Die Gründe für diese Begriffseinführung werden nachfolgend aufgeführt.

Zu Assoziationen, die eine Komposition darstellen, z.B. `eStructuralFeatures`, können

Instanzen des Zieltyps erzeugt werden, die dann Kindknoten im entsprechenden Kontext sind. Besitzt in einem Klassendiagramm eine konkrete Instanz einer `EClass` beispielsweise eine Assoziation zu einer anderen Klasse, so ist diese Assoziation unter `eStructuralFeatures` abgelegt und ist ein Kindknoten der Instanz der Quellklasse der Assoziation. Eine Metaassoziation beschreibt demgegenüber Eigenschaften einer Klasse, wie den Supertyp einer Klasse. Im Rahmen der Berechnung von Modellunterschieden (siehe Abschnitt 4.2) soll dazwischen unterschieden werden können, ob ein bestehender Kindknoten entfernt bzw. ein komplett neues Modellelement als Kindknoten hinzugefügt wurde oder ob eine Eigenschaft einer Klasse über eine Metaassoziation geändert wurde. Aus diesem Grund wird in der Terminologie fortan zwischen Metaassoziationen und Assoziationen unterschieden. Ohne diese Differenzierung würden beide Arten von Änderungen gleichartig behandelt werden, obwohl sie unterschiedlich geartet sind. Ferner wird zwischen Metaattributen und Attributen unterschieden, um zwischen Attributen einer konkreten Klasse eines Klassendiagramms und den Attributen der Metaklassen differenzieren zu können.

Jedem Modellelement können darüber hinaus beliebig viele Annotationen zugeordnet sein. Eine Annotation besitzt einen Quellnamen (`source`) und kann ferner eine Menge von Schlüssel-Wert-Paaren besitzen (`details`). Annotationen werden in dieser Arbeit genutzt, um Informationen zu Stereotypen und Tagged Values, die in EMF nicht direkt unterstützt werden, abzubilden, wie in Abschnitt 4.2.1 noch genauer beschrieben wird.

Zur Berechnung von Modellunterschieden existieren eine Vielzahl verschiedener Ansätze [Bib, KDRPP09, SRC12b, BKL⁺12, SRC13]. Nachfolgend wird ein kurzer Überblick über die wesentlichen Verfahren gegeben. Zur Vereinfachung der Beschreibung wird in diesem Kapitel davon ausgegangen, dass die Unterschiede zwischen zwei verschiedenen Versionen v_a und v_n eines Modells berechnet werden, wobei die Version v_a die ältere Version ist.

Die Ansätze zur Berechnung von Modellunterschieden lassen sich auf verschiedene Weisen kategorisieren. Ein Differenzierungsmerkmal betrifft die Art der berechneten Modellunterschiede. Syntaktische Modellunterschiede ergeben sich aus dem Vergleich der konkreten oder abstrakten Syntax der Modelle [MRR11c]. Demgegenüber beziehen semantische Modellunterschiede nur die Semantik der Modelle ein, ignorieren aber rein syntaktische Unterschiede [MRR11a, MRR12]. So ermittelt die Arbeit in [MRR11c] semantische Modellunterschiede für Klassendiagramme oder die Arbeit in [MRR11b] semantische Modellunterschiede für Aktivitätsdiagramme. Die Unterscheidung zwischen syntaktischen und semantischen Modellunterschieden ist insofern relevant, als dass zwei Modelle syntaktisch verschieden sein können, dessen ungeachtet aber die gleiche Semantik besitzen können. Darüber hinaus können semantisch ähnliche Modelle syntaktisch sehr unterschiedlich sein [MRR11b, MRR11c, MRR12]. In dieser Arbeit werden ausschließlich syntaktische Modellunterschiede bestimmt und genutzt, so dass der Fokus im Folgenden auf Techniken zur Ermittlung syntaktischer Modellunterschiede liegt. Dies ist dadurch motiviert, dass in dieser Arbeit in einigen Fällen syntaktische Änderungsoperationen benötigt werden, um weitere Schritte durchführen zu können. So muss beispielsweise sowohl für die Generierung von Checklisten, in Kapitel 5 beschrieben, als auch für die Generierung der Datenmigrationsfunktionalität, in Kapitel 6 erläutert, eine Umbenennung und Verschiebung von Modellelementen erkannt werden. Änderungen dieser Art können unmittelbar aus den syntaktischen Änderungsoperationen abgelesen werden. Ein anderer Grund ist, dass die

bekannten Werkzeuge zur Bestimmung syntaktischer Modellunterschiede für Klassendiagramme auch für sehr große und komplexe Modelle problemlos ausgeführt werden können. Für die Ansätze zur Ermittlung semantischer Modellunterschiede gilt dies normalerweise lediglich in eingeschränktem Maße [MRR11c].

Ein weiteres Kriterium zur Differenzierung der Ansätze besteht in den Informationen, die zur Unterschiedsberechnung herangezogen werden:

- Operationsbasierte Ansätze [LO92, SZN04, HK10, KHWH10] berechnen die Unterschiede durch das Aufzeichnen der Editieroperationen, die Modellierer in einem Modellierungswerkzeug vorgenommen haben [KKPS12b].
- Zustandsbasierte Ansätze [AP03, KWN05, LGJ07, BP08, KGFE08] nutzen zur Unterschiedsberechnung ausschließlich die Modelle selbst und bedürfen keiner weiteren Zusatzinformationen [KKPS12b].

Nachfolgend werden kurz die wesentlichen Typen von zustandsbasierten Ansätzen dargelegt [KDRPP09]. Der Grund für den Fokus auf zustandsbasierte Ansätze ist, dass der Einsatz von operationsbasierten Ansätzen dedizierte Modellierungswerkzeuge erfordert, durch welche die Änderungen der Modellierer aufgezeichnet werden. Dadurch wird jedoch die Wahl des Modellierungswerkzeuges stark eingeschränkt. Diese Einschränkung wird in dieser Arbeit als zu stark erachtet, so dass in dieser Arbeit ein zustandsbasiertes Verfahren eingesetzt wird, wie im weiteren Verlauf des Kapitels noch ausführlicher erklärt wird.

Zustandsbasierte Ansätze sind üblicherweise in zwei Phasen unterteilt. In der ersten Phase, der Matchingphase, werden die in den beiden Modellen zueinander korrespondierenden Modellelemente identifiziert [SRC12b, BKL⁺12]. Das Ergebnis der Matchingphase ist ein Matchmodell (siehe Definition 6).

Definition 6 (Matchmodell)

Ein Matchmodell zeigt auf, welches Modellelement in einem Modell (bzw. in einer Modellversion) zu welchem Modellelement in einem anderen Modell (bzw. einer anderen Modellversion) korrespondiert. Überdies kann ein Matchmodell Modellelemente umfassen, die in der Matchingphase keinem anderen Modellelement zugeordnet werden konnten.

Die zweite Phase, die Differenzierungsphase, meldet aufbauend auf einem Matchmodell die konkreten Modellunterschiede und kapselt diese in einem Differenzmodell (siehe Definition 7).

Definition 7 (Differenzmodell)

Ein Differenzmodell kapselt sämtliche Modellunterschiede, die bei dem Vergleich zweier Modelle (bzw. Modellversionen) ermittelt wurden. Ein Differenzmodell wird aufbauend auf einem Matchmodell berechnet.

Die Matchingphase wird ausführlicher in Abschnitt 4.2.2 und die Differenzierungsphase ausführlicher in Abschnitt 4.2.4 beschrieben. Im Folgenden werden verschiedene Techniken zur Durchführung der Matchingphase vorgestellt.

Identifizierer-basierte Techniken [AP03, Pan07] setzen voraus, dass jedes Modellelement in einem Modell einen eindeutigen Identifizierer besitzt, der sich nach der Erzeugung des Modell-

elements nicht mehr ändert. Wird beispielsweise eine Klasse in einem Klassendiagramm umbenannt, wird vorausgesetzt, dass diese Umbenennung keine Auswirkungen auf den Identifizierer der Klasse hat. Zwei Modellelemente werden bei derartigen Ansätzen genau dann einander zugeordnet, wenn die Identifizierer beider Modellelemente gleich sind.

Eine weitere Gruppe von Ansätzen setzt Signaturen zur Bestimmung der zueinander korrespondierenden Modellelemente ein [RFG⁺05, LGJ07]. Eine Signatur ist eine von Benutzern definierte Funktion, welche die Identität eines Modellelements berechnet. Zwei Modellelemente gelten genau dann als zueinander korrespondierend, wenn die Signaturen beider Modellelemente identisch sind. Ein einfaches Beispiel für eine Signatur ist der Name des Modellelements. Beim Vergleich zweier Klassendiagramme würde demnach eine Klasse aus einem Klassendiagramm auf eine Klasse aus dem anderen Klassendiagramm abgebildet werden, wenn beide Klassen den gleichen Namen besitzen.

Eine andere Strategie zur Bestimmung eines Matchmodells basiert auf der Berechnung von Ähnlichkeiten zwischen Modellelementen [MGMR02, KWN05, TBWK07, LGJ07, BP08]. Ein Modellelement korrespondiert bei diesen Techniken nur dann zu einem anderen Modellelement, falls die Ähnlichkeit zwischen beiden Modellelementen einen zuvor definierten Schwellenwert überschreitet. Zur Berechnung der Ähnlichkeit werden normalerweise verschiedene Maße eingesetzt. Ein übliches Maß ist die Ähnlichkeit des Namens der untersuchten Modellelemente, z.B. unter Bestimmung der längsten gemeinsamen Zeichenkette beider Namen [PMR13]. Ein weiteres übliches Maß ist die strukturelle Ähnlichkeit von Modellelementen. Bei Klassen in Klassendiagrammen kann die strukturelle Ähnlichkeit beispielsweise über die Ähnlichkeit der enthaltenen Attribute und Assoziationen ermittelt werden.

Darüber hinaus existieren sprachspezifische Techniken, die auf spezifische Modellierungssprachen zugeschnitten sind und Modellunterschiede ausschließlich für Modelle in diesen Modellierungssprachen identifizieren können. Beispiele dafür sind die Arbeiten in [XS05] und [NSC⁺07], die auf Klassendiagramme und Statecharts zugeschnitten sind.

4.2 Methodik zur Bestimmung von Modellunterschieden

Zur Ermittlung von Modellunterschieden wird in dieser Arbeit ein zustandsbasierter Ansatz verfolgt, da ein derartiger Ansatz keine Restriktionen bezüglich der Wahl des eingesetzten Modellierungswerkzeugs macht. Wie bei zustandsbasierten Ansätzen üblich, erfolgt die Unterschiedsberechnung auch in dieser Arbeit in zwei Phasen: der Matching- und der Differenzierungsphase. Abbildung 4.2 gibt einen Überblick über die Schritte zur Bestimmung der Modellunterschiede. Diese werden im Folgenden zunächst kurz erläutert. Die darauffolgenden Abschnitte gehen genauer auf ausgewählte Schritte ein. Im Zuge dieser Beschreibung wird hervorgehoben, welche Schritte von der Standardvorgehensweise zustandsbasierter Ansätze abweichen.

1. Zu Beginn muss zunächst festgelegt werden, zwischen welchen Modellen die Unterschiede ermittelt werden sollen. Normalerweise werden zwei verschiedene Versionen des gleichen Modells ausgewählt, um die Evolution dieses Modells besser nachvollziehen zu können. Gemäß Abbildung 4.2 werden die Modellversionen v_a und v_n miteinander verglichen, wobei die Version v_a die ältere Version und v_n die neuere Version ist. Dessen unge-

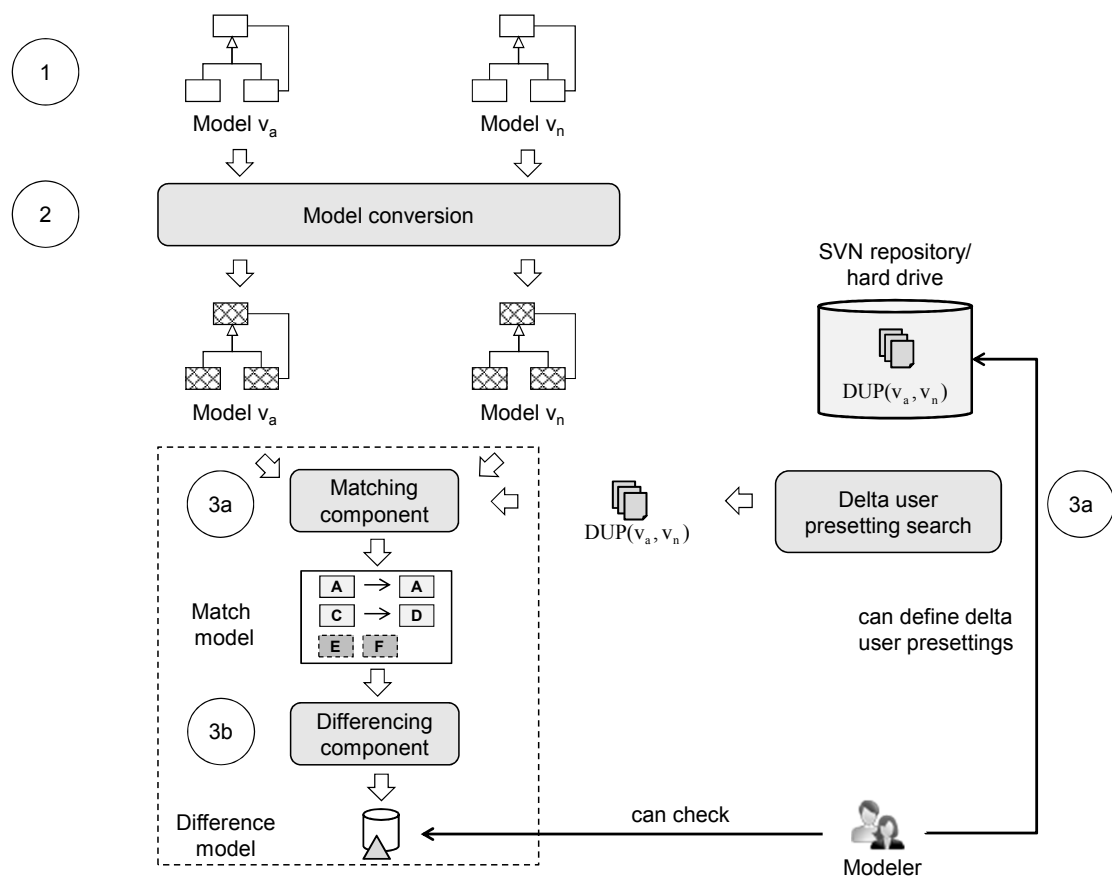


Abbildung 4.2: Überblick über die Schritte zur Identifikation von Modellunterschieden für zwei gegebene Modelle.

achtet können aber ebenso beliebige Modelle der gleichen Modellierungssprache miteinander verglichen werden.

2. Ein Differenzberechnungswerkzeug setzt voraus, dass die Eingabemodelle in einem spezifischen Format vorliegen. Falls die zu untersuchenden Modelle ein anderes Format besitzen, muss als vorbereitender Schritt eine Konvertierung der Eingabemodelle vorgenommen werden, so dass das gewählte Differenzberechnungswerkzeug die Eingabemodelle verarbeiten kann. Dieser Schritt wird in Abschnitt 4.2.1 genauer vorgestellt. In Abbildung 4.2 wird dieser Schritt durch die Modell-Konvertierungskomponente übernommen.
3. Anschließend erfolgt die Unterschiedsberechnung in zwei Phasen: einer Matchingphase und einer Differenzierungsphase.
 - a) Die Durchführung der Matchingphase findet in der Matchingkomponente statt. Als Ergebnis liefert die Matchingkomponente ein Matchmodell. Die Matchingphase wird genauer in Abschnitt 4.2.2 erläutert. In dieser Arbeit wird die klassische Matchingphase um die Verarbeitung von Delta-Benutzervorgaben (siehe Definition 8) erweitert. Die Suchkomponente für Delta-Benutzervorgaben ermittelt, ob für die durchzuführende Differenzberechnung Delta-Benutzervorgaben vorliegen, die in die Matchingphase einbezogen werden müssen. Abschnitt 4.2.3 erläutert diesen Sachverhalt genauer.

Definition 8 (Delta-Benutzervorgabe)

Eine Delta-Benutzervorgabe steuert, auf welches Modellelement ein spezifisches Modellelement in der Matchingphase abgebildet wird oder legt fest, dass ein Modellelement auf kein anderes Modellelement abgebildet werden soll.

- b) Aufbauend auf dem Matchmodell wird in der Differenzierungsphase durch die Differenzkomponente das Differenzmodell erzeugt. Die Vorgehensweise zur Ermittlung des Differenzmodells aus dem Matchmodell wird genauer in Abschnitt 4.2.4 dargestellt.

In dieser Arbeit wurde als Differenzberechnungswerkzeug EMF Compare [EMFa] ausgewählt und partiell erweitert. Da Klassendiagramme die am häufigsten eingesetzte Modellierungstechnik der UML sind [HWRK11, HWR14] und DSA vor allem Klassendiagramme zur Modellierung einsetzt, liegt der Fokus dieser Arbeit auf dem Einsatz von Klassendiagrammen. Aus diesem Grund betrachten die nachfolgenden Abschnitte insbesondere den Fall, dass die Bestimmung von Modellunterschieden für Klassendiagramme durchgeführt wird.

4.2.1 Konvertierung von Eingabemodellen

Falls die Modelle in einem Format vorliegen, dass nicht direkt von dem Differenzberechnungswerkzeug verarbeitet werden kann, muss als vorbereitender Schritt eine Konvertierung der Modelle in den passenden Modelltyp vorgenommen werden. In dem Projekt, in dessen Kontext diese Arbeit durchgeführt wurde, wurde ein Modellierungswerkzeug eingesetzt, welches die modellierten Klassendiagramme in einem proprietären Format speichert. Damit die Modelle

durch andere Komponenten, wie den Generator, verarbeitet werden können, bietet das Modellierungswerkzeug die Möglichkeit, die Modelle im XMI-Format [OMG15] zu exportieren. Demgegenüber erwartet EMF Compare als Eingabe ECore-Modelle. Infolgedessen muss eine Konvertierung vom XMI-Format in das ECore-Format durchgeführt werden. Nachfolgend wird angegeben, wie die wichtigsten Elemente aus Klassendiagrammen auf ECore abgebildet werden können:

- `UML:Package` → `EPackage`. Sowohl Klassendiagramme als auch ECore unterstützen direkt das Konzept von Paketen. Diese können direkt aufeinander abgebildet werden.
- `UML:Class` → `EClass`. Analog zum vorhergehenden Fall, unterstützen sowohl Klassendiagramme als auch ECore direkt das Konzept von Klassen. Diese können daher wiederum direkt aufeinander abgebildet werden.
- `UML:Interface` → `EClass`, wobei gilt `Interface == true`. In ECore werden sowohl Klassen als auch Interfaces über Instanzen vom Typ `EClass` ausgedrückt. Interfaces können in ECore von Klassen über die Eigenschaft `Interface` unterschieden werden.
- `UML:Attribute` → `EAttribute`. Attribute in Klassendiagrammen korrespondieren zu Instanzen vom Typ `EAttribute` in ECore. Für die Typen der Attribute wird nachfolgend spezifiziertes Typmapping durchgeführt:
 - `boolean` → `EBoolean`
 - `byte` → `EByte`
 - `char` → `EChar`
 - `java.util.Date` → `EDate`
 - `double` → `EDouble`
 - `float` → `EFloat`
 - `int` → `EInt`
 - `long` → `ELong`
 - `short` → `EShort`
 - `String` → `EString`
- `UML:Association` (nicht zu Enumerationen) → `EReference`. Für die Behandlung von Assoziationen muss eine Fallunterscheidung durchgeführt werden. Insofern eine Klasse eine Assoziation zu einer Klasse oder einem Interface besitzt, kann die Assoziation als Instanz vom Typ `EReference` ausgedrückt werden. In ECore ist es jedoch nicht möglich, dass eine Klasse über eine `EReference` auf eine Enumeration verweist, wie in Abschnitt 4.1 erläutert. Dieser Aspekt wird im nächsten Auflistungspunkt betrachtet. Nachstehend wird zusammengefasst, welche Sprachmittel der UML-Assoziationen bei diesem Schritt beachtet werden:

- Falls ein Rollenname für ein Assoziationsende gesetzt ist, wird dieser Name direkt für die Assoziation in ECore verwendet. Andernfalls wird der Name der Zielklasse als Rollename eingesetzt - per Konvention mit Kleinbuchstaben zu Beginn. Für mehrwertige Assoziationen wird zusätzlich ein „s“ an den Namen angehängen.
- Kardinalitäten werden unverändert von der Assoziation auf die EReference übertragen.
- Ist für eine Assoziation keine Pfeilrichtung angegeben, wird eine bidirektionale Assoziation angenommen, analog zu [Rum11]. Da jede Assoziation in ECore implizit gerichtet ist, werden in diesem Fall zwei Assoziationen erzeugt, je eine für eine Assoziationsrichtung. Andernfalls wird eine unidirektionale EReference erzeugt.
- Abgeleitete Assoziationen werden ebenfalls unterstützt. In diesem Fall wird `derived` auf den booleschen Wert `true` gesetzt.
- Bei qualifizierten Assoziationen wird lediglich der Spezialfall der `ordered` Assoziation beachtet. Hierbei wird `ordered` auf den booleschen Wert `true` gesetzt.

Einige Sprachmittel von Assoziationen werden bei der Konvertierung nicht beachtet: Assoziationsnamen, qualifizierte Assoziationen, mehrstellige Assoziationen sowie Assoziationsklassen. Diese Sprachmittel werden somit bei der Differenzberechnung nicht beachtet. Bei Bedarf können jedoch zusätzliche Informationen zu einer Assoziation, z.B. der Assoziationsname, über Annotationen zur Assoziation hinterlegt werden.

- `UML:Association` (zu Enumerationen) $\rightarrow$ `EAttribute`. Assoziationen zu Enumerationen werden in ECore als `Attribute` und nicht als Assoziationen dargestellt, wie in Abschnitt 4.1 beschrieben. Aus diesem Grund muss für eine Assoziation zu einer Enumeration ein Attribut mit der Enumeration als Attributtyp angelegt werden. Für die Namen des Attributs gelten die gleichen Regeln wie für die Namen der Assoziationen im vorherigen Auflistungspunkt.

Da ECore nicht direkt das Konzept von Stereotypen und Tagged Values unterstützt, werden diese Informationen bei der Konvertierung von Klassendiagrammen in ECore-Modelle über Annotationen abgebildet. Die Stereotypen eines Modellelements werden über eine dedizierte Annotation dargestellt, die als Werte in den Schlüssel-Wert-Paaren die Namen der angewandten Stereotypen kapselt. Analog wird zur Verwaltung von Tagged Values eine Annotation verwendet, die als Schlüssel die Namen der Tagged Values und als Werte die zugehörigen Werte kapselt. Korrespondierend dazu wird in jedem Modellelement eine Annotation genutzt, um den eindeutigen Identifizierer eines Modellelements zu verwalten, falls ein Modellelement einen solchen besitzt.

Nachfolgend wird die Konvertierung von Klassendiagrammen in ECore-Modelle unter Verwendung der zuvor definierten Abbildungsvorschriften veranschaulicht. Abbildung 4.3 zeigt dazu einen vereinfachten Ausschnitt aus einem Klassendiagramm für ein Bibliothekssystem.

Die drei Klassen `Library`, `Book` und `BookExemplar` werden auf `EClass` Instanzen, die zwei Attribute `title` und `id` auf `EAttribute` Instanzen und die zwei Assoziationen auf drei `EReference` Instanzen abgebildet. Wie zuvor dargelegt, werden für eine bidirektionale Assoziation zwei `EReference` Instanzen erzeugt, je eine für eine Assoziationsrichtung. Aus

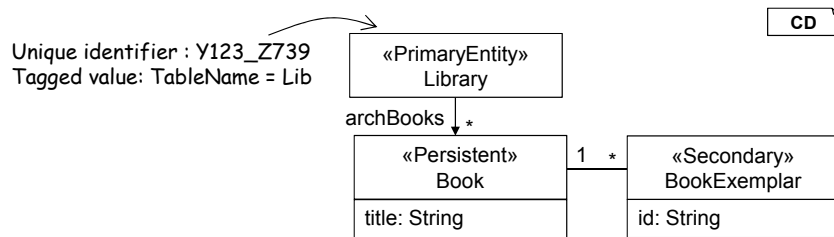


Abbildung 4.3: Auszug aus einem Klassendiagramm für ein Bibliothekssystem.

diesem Grund werden für die bidirektionale Assoziation zwischen `Book` und `BookExemplar` zwei `EReference` Instanzen erzeugt, wobei beide zur Abbildung der bidirektionalen Assoziation über `eOpposite` auf die jeweils andere Assoziationsrichtung verweisen.

Jedes dieser Modellelemente besitzt eine Annotation mit dem Quellnamen `Id` zur Verwaltung des eindeutigen Identifizierers des Modellelements, in dem Beispiel gespeichert unter dem Schlüsselnamen `ElementId`, z.B. `Y123_Z739` für die Klasse `Library`. Darüber hinaus besitzt jedes Modellelement, welches mit einem Stereotyp markiert ist, eine Annotation mit Quellnamen `Stereotype`. Der konkrete Stereotyp ist in dem Beispiel dem Schlüssel mit dem Namen `Stereotype` zugeordnet, z.B. `PrimaryEntity` für die Klasse `Library`. Ferner enthält jedes Modellelement, für das Tagged Values hinterlegt sind, eine Annotation mit Quellnamen `TaggedValues`. Der Name des Tagged Values bildet den Schlüssel und der Wert des Tagged Values den zugehörigen Wert. Beispielsweise ist der Klasse `Library` der Tagged Value `TableName` mit Wert „Lib“ zugeordnet. Das resultierende `ECore`-Modell ist auszugsweise in Abbildung 4.4 dargestellt.

Die Abbildung einer bidirektionalen Assoziation auf zwei separate `EReference` Instanzen hat Auswirkungen auf die Differenzberechnung zwischen zwei Modellen. Wird in einer Version eines Klassendiagramms die Navigierbarkeit einer Assoziation verändert, führt das dazu, dass in dem `ECore`-Modell zu der geänderten Version eine Assoziation hinzugefügt oder gelöscht wird. Dies wird im Folgenden präzisiert.

Wird die Navigierbarkeit einer Assoziation geändert, führt das in zwei Fällen dazu, dass in der neuen Version des Klassendiagramms auf `ECore`-Ebene eine `EReference` Instanz hinzugefügt wird: 1) ein Assoziationsende wird durch eine Modelländerung navigierbar gemacht und dadurch entsteht eine bidirektionale Assoziation und 2) die Navigierbarkeit eines Assoziationsendes wird aufgehoben und dadurch entsteht eine Assoziation ohne Angabe zur Navigierbarkeit, die somit als bidirektionale Assoziation betrachtet wird. In beiden Fällen muss in dem Klassendiagramm ursprünglich eine unidirektionale Assoziation bestanden haben, so dass die Modelländerung auf `ECore`-Ebene zu dem Hinzufügen einer Assoziation führt.

Demgegenüber führt die Änderung der Navigierbarkeit einer Assoziation in zwei Fällen dazu, dass in der neuen Version des Klassendiagramms auf `ECore`-Ebene eine Assoziation gelöscht wird: 1) ein Assoziationsende wird durch eine Modelländerung navigierbar gemacht und dadurch entsteht eine unidirektionale Assoziation und 2) die Navigierbarkeit eines Assoziationsendes wird aufgehoben und dadurch entsteht eine unidirektionale Assoziation. Im erstgenannten Fall muss zuvor eine Assoziation ohne Angabe bezüglich der Navigierbarkeit und im zweiten

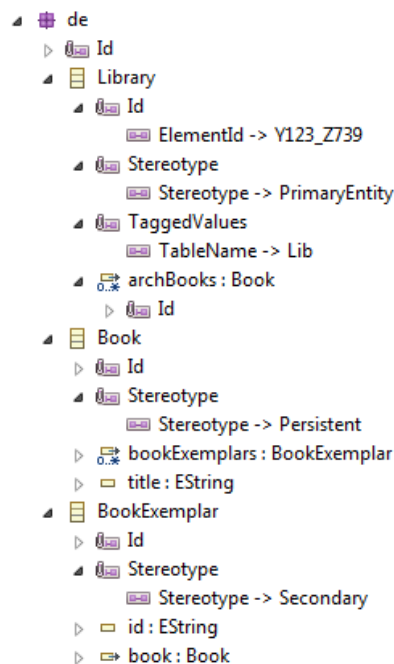


Abbildung 4.4: Ausschnitt aus dem ECore-Modell zum Klassendiagramm in Abbildung 4.3.

Fall eine Assoziation mit zwei navigierbaren Assoziationsenden vorgelegen haben. In beiden Konstellationen hätten zu der ursprünglichen Assoziation in ECore zwei Assoziationen existiert, so dass diese Modelländerungen zu dem Löschen einer Assoziation führen.

4.2.2 Durchführung der Matchingphase

Das Ziel der Matchingphase ist die Erstellung eines Matchmodells, auf dessen Grundlage in der Differenzierungsphase das Differenzmodell bestimmt wird. Abbildung 4.5 zeigt die wichtigsten Elemente des EMFCompare-Metamodells für Matchmodelle. Zur Vereinfachung der Beschreibung wurde die Struktur des Metamodells geringfügig geändert.

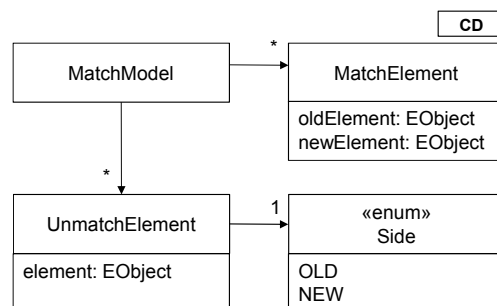


Abbildung 4.5: Metamodell für die Matchmodelle dieser Arbeit.

Das Metamodell kapselt über `MatchElement` die verschiedenen Matchelemente, also Paare von Modellelementen, die in der Matchingphase als zueinander korrespondierend identifiziert wurden. Ferner verweist `UnmatchElement` auf Modellelemente, die keinem anderen Modellelement zugeordnet werden konnten. Der Verweis auf die Enumeration `Side` gibt Auskunft darüber, ob dieses Modellelement in der alten oder der neuen Modellversion enthalten war.

Der Pseudocode zur Berechnung eines Matchmodells ist in Algorithmus 4.1 gezeigt. Nachdem zu Beginn zunächst ein leeres Matchmodell erzeugt wird (Zeile 1), werden darauffolgend die Wurzelemente der eingegebenen Modelle bestimmt (Zeile 2 – 3). Bei Klassendiagrammen können dies beispielsweise die Pakete sein, die sich auf oberster Hierarchieebene befinden. Für diese Wurzelemente wird anschließend die Funktion `mapLists` aufgerufen (Zeile 4). Diese untersucht, welche Modellelemente aus der ersten Liste zu Modellelementen aus der zweiten Liste korrespondieren. Der Pseudocode dieser und weiterer Funktionen wird im Anschluss noch genauer vorgestellt. Als Ergebnis liefert `mapLists` eine Liste von Matchelementen, die jeweils ein Modellelement aus der neuen Modellversion kapseln und angeben, welches Modellelement in der alten Version dazu korrespondiert. Ausgehend von diesen Modellelementen der Matchelemente wird daraufhin die Suche nach weiteren Matchelementen rekursiv initiiert (Zeile 6 – 7).

Algorithmus 4.1 : Function `matchModels()`

```

input  : modelOld: Old model version,
          modelNew: New model version
output : Match model for new and old model version

1 Create empty matchModel;
2 rootsNew  $\leftarrow$  getContent (modelNew) ;
3 rootsOld  $\leftarrow$  getContent (modelOld) ;
4 matches  $\leftarrow$  mapLists (rootsOld, rootsNew) ;

5 for each match : matches do
6   | recursiveMappings (old element of match, new element of match,
7   |   matchModel) ;

8 if stillToFindNew is not empty  $\wedge$  stillToFindOld is not empty then
9   | matches  $\leftarrow$  mapLists (stillToFindOld, stillToFindNew)
10  | for each match : matches do
11    | recursiveMappings (old element of match, new element of match,
12    |   matchModel) ;

13 createUnmatchedElements (stillToFindOld, stillToFindNew, matchModel) ;
return matchModel;

```

Wie in der weiteren Beschreibung noch gezeigt wird, werden jeweils die Modellelemente der Modelle miteinander verglichen, die auf der gleichen Hierarchieebene liegen. Im Falle von Klassendiagrammen werden beispielsweise die Klassen verglichen, die im Wurzel-Paket der Klassendiagramme liegen. Der Nachteil an dieser Vorgehensweise ist, dass so Verschiebungen von Modellelementen nicht erkannt werden. Aus diesem Grund verwaltet der Algorithmus zur

Erzeugung des Matchmodells in den Listen `stillToFindNew` und `stillToFindOld` zusätzlich die Modellelemente, die keinem anderen Modellelement zugeordnet werden konnten. Im Anschluss an den rekursiven Aufruf zur Bestimmung der Matchelemente (Zeile 6 – 7), wird daher die Suche nach Matchelementen für die Modellelemente veranlasst, die in diesen Listen enthalten sind (Zeile 9 – 12). Auf diese Weise können auch Verschiebungen von Modellelementen erkannt werden. Sämtliche Modellelemente, die nach diesem Schritt weiterhin in den Datenstrukturen `stillToFindNew` und `stillToFindOld` enthalten sind, konnten keinem anderen Modellelement zugeordnet werden. Diese Modellelemente werden entsprechend als einzelne Modellelemente ohne Zuordnung dem Matchmodell hinzugefügt (Zeile 13).

Algorithmus 4.2 zeigt den Pseudocode zur Funktion `mapLists`. Diese Funktion sucht zu jedem Modellelement aus der ersten übergebenen Liste das korrespondierende Modellelement in der zweiten übergebenen Liste. Dazu iteriert die Funktion über jedes Modellelement `obj1` der neuen Modellversion (Zeile 2 – 8) und bestimmt die Ähnlichkeitswerte zwischen `obj1` und den nicht zugeordneten Modellelementen der alten Version (Zeile 3). Das Modellelement mit der höchsten Ähnlichkeit zu `obj1` wird zurückgeliefert. Anschließend wird geprüft, ob die Ähnlichkeit des resultierenden Paares von Modellelementen groß genug ist (Zeile 4). Diese Bedingung ist genau dann erfüllt, wenn die berechnete Ähnlichkeit einen vordefinierten Schwellenwert überschreitet. Als Folge davon werden die beiden Modellelemente als zueinander korrespondierend erachtet. In diesem Fall wird ein Matchelement hinzugefügt, welches die beiden Modellelemente kapselt (Zeile 6). Die beiden Modellelemente werden anschließend aus den Listen `listNew` und `listOld` gelöscht (Zeile 7-8), damit jedes Modellelement maximal einem anderen Modellelement zugeordnet wird.

Algorithmus 4.2 : Function `mapLists()`

```

input  : listOld: List of model elements from old model version,
          listNew: List of model elements from new model version
output : List of matches when searching passed lists of model elements

1 Create empty list matches;
2 for each obj1 : listNew do
3   obj2 ← findMostSimilar (listOld, obj1);
4   similar = isSimilar(obj2, obj1);
5   if similar then
6     addMatchElement (matches, obj1, obj2);
7     Remove obj1 from listNew;
8     Remove obj2 from listOld;
9 Add all listNew to stillToFindNew;
10 Add all listOld to stillToFindOld;
11 return matches;
```

Diese Vorgehensweise hat zur Folge, dass die Reihenfolge der Elemente in den Listen die Menge an gefundenen Matchelementen beeinflussen kann. Dieser Fall tritt ein, wenn in den übergebenen Listen mehrere Elemente eine sehr hohe Ähnlichkeit zueinander haben, so dass

es mehrere Zuordnungsmöglichkeiten zu einem Modellelement gibt. Um diesem Problem entgegenzuwirken, können für sämtliche Paare von Modellelementen aus der ersten und zweiten übergebenen Liste die Ähnlichkeiten ermittelt werden. Basierend auf diesen Ähnlichkeiten kann dann die optimale Zuordnung bestimmt werden. Diese Vorgehensweise wird allerdings in dieser Arbeit nicht angewendet, um die Anzahl der zu betrachtenden Paare von Modellelementen und somit die Anzahl der notwendigen Berechnungen der Ähnlichkeiten zu reduzieren. In der Praxis tritt dieses Problem zudem üblicherweise nicht auf, da der verwendete Schwellenwert für den Ähnlichkeitswert hoch ist, so dass es normalerweise nicht mehrere Zuordnungsmöglichkeiten gibt. Sollte dieses Problem allerdings tatsächlich auftreten, wird in Abschnitt 4.2.3 ein Ansatz vorgestellt, über den Modellierer eine daraus resultierende falsche Zuordnung in der Matchingphase korrigieren können.

Sämtliche Modellelemente, die im Anschluss daran noch in `listNew` und `listOld` enthalten sind, konnten keinem Modellelement zugeordnet werden. Diese werden daher abschließend zu den Datenstrukturen `stillToFindNew` und `stillToFindOld` hinzugefügt (Zeile 9-10).

Nachfolgend wird in Algorithmus 4.3 die Berechnung der Ähnlichkeit zweier Modellelemente betrachtet. Diese Berechnung wird initiiert, damit zu einem gegebenen Modellelement das Modellelement mit der höchsten Ähnlichkeit ermittelt werden kann. Zu Beginn wird dazu die Ähnlichkeit der Namen der beiden Modellelemente berechnet (Zeile 1). Für diesen Berechnungsschritt gibt es verschiedene Ausprägungsmöglichkeiten [CRF03], beispielsweise die Bestimmung der längsten gemeinsamen Zeichenkette [PMR13] oder die Ermittlung der Editierdistanz [RY98]. Darauf folgend wird die strukturelle Ähnlichkeit berechnet (Zeile 2). Diese bezieht beispielsweise für Klassen die enthaltenen Attribute und Assoziationen ein. Eine einfache Vorgehensweise für diesen Berechnungsschritt besteht in der Konstruktion einer Zeichenkette, welche die Namen sämtlicher Attribute und die Rollennamen sämtlicher Assoziationen umfasst. Auf diese Weise kann die Berechnung der strukturellen Ähnlichkeit wiederum auf die Berechnung der Ähnlichkeit von Zeichenketten zurückgeführt werden. In EMF Compare wird ein derartiger Ansatz verfolgt. Abschließend wird die Gesamtähnlichkeit ermittelt (Zeile 3 – 4). Diese ergibt sich aus der gewichteten Addition der einzelnen Teilähnlichkeiten.

Algorithmus 4.3 : Function `calculateSimilarity()`

input : `objOld`: Model element from old model version,
 `objNew`: Model element from new model version
output : Similarity between `objOld` and `objNew`

```

1 nameSimilarity  $\leftarrow$  calculateNameSimilarity(objOld, objNew);
2 relationsSimilarity  $\leftarrow$  calculateRelationsSimilarity(objOld, objNew);
3 similarity  $\leftarrow$  (nameSimilarity  $\times$  nameWeight) +
4    (relationsSimilarity  $\times$  relationsWeight);
5 return similarity;

```

Als Nächstes wird in Algorithmus 4.4 der Pseudocode zu `recursiveMappings` vorgestellt. Dieser Funktion werden zwei Modellelemente übergeben, die gemäß Matchmodell einander zugeordnet sind. Für die in diesen Modellelementen enthaltenen Kindelemente wird rekursiv

die Suche nach weiteren Matchelementen veranlasst. Dazu werden die Kindelemente ermittelt, die unmittelbar in den beiden Modellelementen enthalten sind. Für ein Paket in einem Klassendiagramm sind dies beispielsweise die UML-Klassen, die auf oberster Ebene unmittelbar in dem Paket enthalten sind. Für die resultierenden Elemente wird die bekannte Funktion `mapLists` aufgerufen, so dass auf dieser Hierarchieebene wieder nach Matchelementen gesucht wird (Zeile 1). Die gefundenen Matchelemente werden danach zum Matchmodell hinzugefügt (Zeile 2) und die Funktion `recursiveMappings` wird wiederum für die Modellelemente der Matchelemente aufgerufen (Zeile 4 – 5). Diese Rekursion wird fortgesetzt, bis keine Matchelemente mehr gefunden werden können, z.B. weil ein Modellelement als Eingabe verarbeitet wird, welches keine weiteren Elemente enthält.

Algorithmus 4.4 : Function recursiveMappings()

```

input  : objOld: Model element from old model version,
          objNew: Model element from new model version,
          matchModel: Match model
output : matchModel: Possibly extended match model

1 matches ← mapLists (getContent (objOld) , getContent (objNew) ) ;
2 Add all matches to match model;
3 for each match : matches do
4   | recursiveMappings (old element of match, new element of match,
5   |   matchModel) ;

```

Zusammengefasst traversiert der Algorithmus zur Berechnung des Matchmodells rekursiv die Modellelemente der Modelle, die auf der gleichen Hierarchieebene liegen. Diese Modellelemente werden jeweils miteinander verglichen und ein Paar von Modellelementen wird zum Matchmodell hinzugefügt, falls die Ähnlichkeit der beiden Modellelemente einen gewissen Schwellenwert überschreitet. Im Anschluss daran wird der Vergleich der Modellelemente durchgeführt, die in der ersten Phase keinen anderen Modellelementen zugeordnet werden konnten. Dieser Schritt wird ausgeführt, um Verschiebungen von Modellelementen zu behandeln. Die Modellelemente die danach weiterhin keinem anderen Modellelement zugeordnet werden können, werden abschließend als Modellelemente ohne Match zum Matchmodell hinzugefügt.

Beispiel

Die Ermittlung eines Matchmodells wird im Folgenden auf Basis eines Beispiels erläutert. In diesem Beispiel wird angenommen, dass für die beiden Modelle aus Abbildung 4.6 und Abbildung 4.7 das Matchmodell bestimmt werden soll. Dabei stellt das Modell in Abbildung 4.6 die ältere Version dar. Ferner wird angenommen, dass das Wurzel-Paket in beiden Modellen das Paket `library` ist, die Klassen `Library`, `Book` und `BookExemplar` im Paket `core` liegen und die Klasse `Address` im Paket `ext` liegt. In der älteren Version sei die Klasse `Person` im Paket `core`, in der neueren Version aber im Paket `ext`.

Zu Beginn werden die Wurzelemente beider Modelle bestimmt und die zueinander korre-

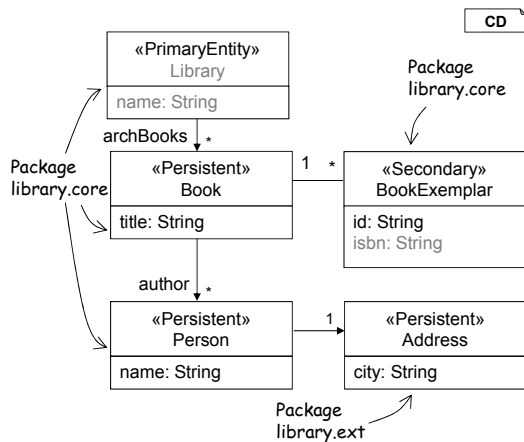


Abbildung 4.6: Klassendiagramm für ein Bibliothekssystem (ältere Version v_a).

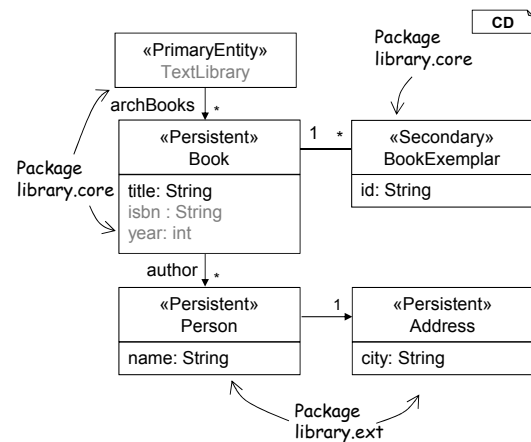


Abbildung 4.7: Klassendiagramm für ein Bibliothekssystem (neuere Version v_n).

spondierenden Wurzelemente ermittelt. In diesem Beispiel sind die Pakete `library` die Wurzelemente. Diese sind unverändert und werden folglich aufeinander abgebildet. Anschließend werden die unmittelbar in diesen Paketen enthaltenen Kindelemente bestimmt und dabei werden die beiden gleichnamigen Pakete `core` sowie die gleichnamigen Pakete `ext` als zueinander korrespondierend bestimmt. Als Nächstes werden die Inhalte der Pakete `core` auf Matchelemente hin untersucht. Im Zuge dessen werden die Paare bestehend aus den beiden `Book` sowie den beiden `BookExemplar` Klassen jeweils Matchelemente. Ferner wird trotz der Umbenennung die Klasse `Library` im Matchmodell der Klasse `TextLibrary` zugeordnet. Demgegenüber kann die Klasse `Person` keiner anderen Klasse zugeordnet werden, da nur die in den Paketen `core` enthaltenen Klassen im ersten Schritt betrachtet werden. Darauf folgend wird für die in den Matchelementen gekapselten Klassen nach weiteren Matchelementen gesucht. Bei den Aufrufen für diese drei Klassen können die Attribute `name` (Klasse `Library`), `isbn` (Klasse `Book`), `year` (Klasse `Book`) und `isbn` (Klasse `BookExemplar`) keinem anderen Attribut zugeordnet werden, da zunächst nur die Attribute der einander korrespondierenden Klassen betrachtet werden.

Analog wird für die Inhalte der Pakete `ext` verfahren und es werden die gleichnamigen Klassen `Address` aufeinander abgebildet. Die Klasse `Person` im neuen Paket `ext` wird hingegen keiner anderen Klasse zugeordnet.

In der ersten Phase des Matchingalgorithmus konnten folgende Modellelemente keinem anderen Modellelement zugeordnet werden: Attribut `name` (Klasse `Library`), Attribut `isbn` (Klasse `Book`), Attribut `year` (Klasse `Book`), Attribut `isbn` (Klasse `BookExemplar`) und die beiden `Person` Klassen. Aus diesem Grund wird versucht unter diesen bisher nicht zugeordneten Elementen weitere Matchelemente zu identifizieren. Dies gelingt für die beiden `Person` Klassen sowie für die Attribute `isbn` der Klassen `Book` und `BookExemplar`. Lediglich die Attribute `name` (Klasse `Library`) und `year` (Klasse `Book`) verbleiben ohne Zuordnung.

Abbildung 4.8 zeigt zusammenfassend das Matchmodell für das Beispiel in Form eines Ob-

jektdiagramms. Zur Verkleinerung des Objektdiagramms sind lediglich die Ausschnitte des Matchmodells zu sehen, die für die Erzeugung des späteren Differenzmodells relevant sind. Wie dort zu sehen, enthält das Matchmodell für die zwei Modellelemente ohne Zuordnung jeweils eine Instanz vom Typ `UnmatchElement`. Ferner zeigt es die drei Matchelemente, die für die Differenzberechnung relevant sind.

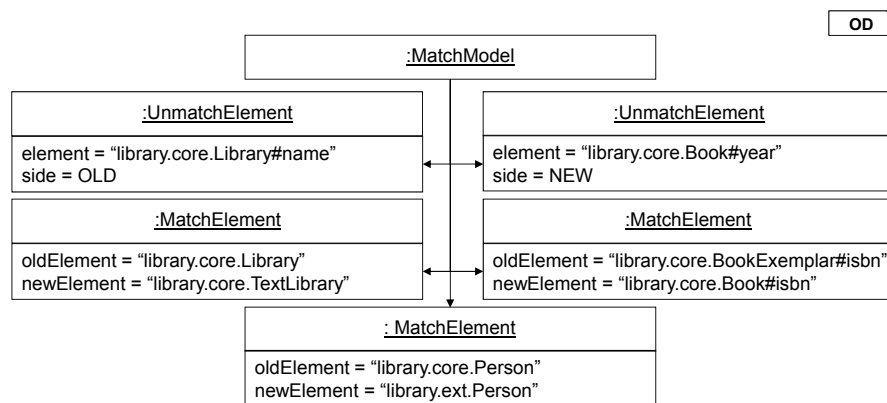


Abbildung 4.8: Matchmodell für die Modelle aus Abbildung 4.6 und Abbildung 4.7.

4.2.3 Integration von Delta-Benutzervorgaben in die Matchingphase

Ein fundamentales Problem bei der Modellunterschiedsberechnung ist, dass ein automatischer Ansatz die Modellunterschiede nicht immer korrekt bestimmen kann [PMR13, MR14b]. Dies ist darin begründet, dass es oftmals kein eindeutiges Matchmodell und somit auch kein eindeutiges Differenzmodell gibt. Findet beispielsweise in einem Klassendiagramm eine Umbenennung einer Klasse statt, kann diese Änderung nicht nur als Umbenennung sondern auch als das Löschen einer Klasse und Hinzufügen einer neuen Klasse angesehen werden [PMR13]. Im erstgenannten Fall wird die ursprüngliche Klasse in der Matchingphase der umbenannten Klasse zugeordnet. Im zweitgenannten Fall wird sowohl die ursprüngliche als auch die umbenannte Klasse keinem anderen Modellelement zugeordnet.

Die Wahrscheinlichkeit für das Auftreten solcher Probleme kann in manchen Fällen reduziert werden. So kann das Problem bezüglich der Umbenennung von Modellelementen dadurch adressiert werden, dass linguistische Datenbanken wie WordNet [Wor] in der Matchingphase genutzt werden. Auf diese Weise kann die Semantik von Zeichenketten in die Analyse einbezogen werden [PMR13]. Dies wirkt sich jedoch negativ auf die Laufzeit der Matchingalgorithmen aus und bietet darüber hinaus keine Garantie dafür, dass Umbenennungen immer korrekt erkannt werden. Alternativ dazu kann eine Identifizierer-basierte Technik zur Durchführung der Matchingphase eingesetzt werden. Diese Technik ist im Allgemeinen weniger anfällig für die beschriebene Problematik. Der Grund hierfür ist, dass durch die Modelländerungen die Identifizierer der bestehenden Modellelemente nicht geändert werden. Auf diese Weise ist es möglich unmittelbar abzuleiten, welche Modellelemente zueinander korrespondieren. Aber auch durch den Einsatz von Identifizierer-basierten Techniken kann nicht garantiert werden, dass die be-

schriebenen Probleme nicht auftreten. Angenommen, ein Modellierer möchte beispielsweise ein Attribut umbenennen, löscht dieses aber unbeabsichtigt und fügt es anschließend mit neuem Namen hinzu. Durch das Löschen und Hinzufügen des Attributs ändert sich der Identifizierer des Attributs. Folglich wird das neu hinzugefügte Attribut nicht dem ursprünglichen Attribut zugeordnet und es werden fehlerhafte Modelländerungen zurückgeliefert. Eine weitere Möglichkeit zur Reduktion der Wahrscheinlichkeit für die beschriebene Problematik besteht in dem Einsatz eines operationsbasierten Verfahrens. Durch die Aufzeichnung der Aktionen der Modellierer können operationsbasierte Verfahren die Modelländerungen genauer darlegen. Aber auch operationsbasierte Verfahren können nicht garantieren, dass immer korrekte Modellunterschiede gefunden werden. Das zuvor genannte Beispiel für das unbeabsichtigte Löschen und Hinzufügen eines Attributs anstelle der Umbenennung eines Attributs führt auch hier zu Problemen. In diesem Fall ist für ein operationsbasiertes Werkzeug ebenfalls nicht ersichtlich, dass eigentlich eine Umbenennung durchgeführt werden sollte. Überdies setzt diese Technik voraus, dass dedizierte Modellierungswerkzeuge eingesetzt werden - was nicht immer vorausgesetzt werden kann.

Solange sich die Nutzer von Differenzberechnungswerkzeugen über die beschriebene Problematik im Klaren sind, ist es nicht zwangsläufig kritisch, wenn fehlerhafte Modellunterschiede gemeldet werden. Problematisch wird es jedoch dann, wenn die ermittelten Modellunterschiede automatisch weiterverarbeitet werden und aufbauend darauf weitere Aktionen durchgeführt werden. In diesem Fall führen fehlerhafte Modellunterschiede dazu, dass unter Umständen falsche Aktionen im Anschluss durchgeführt werden. In dieser Arbeit liegt genau diese Situation vor. Sowohl bei der Generierung von Checklisten (siehe Kapitel 5) als auch bei der Generierung der Migrationsfunktionalität zur Daten Evolution (siehe Kapitel 6) werden die Modellunterschiede ermittelt und automatisch verarbeitet. Werden Modelländerungen falsch ermittelt, sind die resultierenden Checklisten bzw. die generierte Migrationsfunktionalität unbrauchbar.

Die Hauptursache für fehlerhafte Modellunterschiede ist, dass Modellelemente, die zueinander korrespondieren, nicht als solche erkannt werden [KKPS12a, PMR13]. Derartige Fehler führen zu einem fehlerhaften Matchmodell, so dass das daraus abgeleitete Differenzmodell ebenfalls fehlerhaft ist. Um diesem Problem entgegenzuwirken, werden in dieser Arbeit Delta-Benutzervorgaben (siehe Definition 8 auf Seite 33) eingeführt. Diese erlauben es den Modellierern zu steuern, auf welches Modellelement ein spezifisches Modellelement in der Matchingphase abgebildet wird. Auf diese Weise kann verhindert werden, dass ein fehlerhaftes Matchmodell berechnet wird.

Existierende Arbeiten zu der Modelldifferenzberechnung widmen sich diesem Problem nur indirekt. So wurde beispielsweise in [KKPS12a] die Adaption von Modelldifferenzberechnungswerkzeugen untersucht. Dort wird dargelegt, wie in dem Framework SiDiff [SiD] die Matchingphase an Benutzerpräferenzen angepasst werden kann. Allerdings liegt der Fokus hierbei auf der Möglichkeit konfigurierbare Algorithmen zur Durchführung der Matchingphase bereitzustellen. Aber auch diese konfigurierbaren Algorithmen können nicht sicherstellen, dass das berechnete Matchmodell den Erwartungen der Modellierer entspricht.

Im Folgenden wird als Erstes erläutert, wie Delta-Benutzervorgaben definiert werden können. Im Anschluss daran wird dargelegt, wie Delta-Benutzervorgaben in der Matchingphase bezogen werden. Abschließend wird die um die Verarbeitung von Delta-Benutzervorgaben erweiterte Matchingphase erläutert.

Definition von Delta-Benutzervorgaben

Delta-Benutzervorgaben haben den Zweck für ausgewählte Modellelemente zu steuern, auf welche Modellelemente sie in der Matchingphase abgebildet werden. Darüber hinaus können sie ausdrücken, dass einem Modellelement des alten oder des neuen Modells kein Modellelement zugeordnet werden soll. Folglich können Delta-Benutzervorgaben als eine Relation zwischen der Menge der alten und der Menge der neuen Modellelemente verstanden werden. Die Quell- und Zielmenge dieser Relation umfasst zusätzlich ein spezielles Element, welches ein nicht vorhandenes Modellelement repräsentiert. Auf diese Weise kann durch die Relation auch ausgedrückt werden, dass einem Modellelement kein anderes Modellelement zugeordnet werden soll. Allerdings handelt es sich bei dieser Relation um keine Funktion, da die Relation mehrere Tupel umfassen kann, die verschiedenen Elementen aus der Zielmenge das genannte Element zur Repräsentation eines nicht vorhandenen Modellelements zuordnet. Dieses Szenario liegt vor, wenn die Delta-Benutzervorgaben ausdrücken, dass verschiedenen Modellelementen aus dem neuen Modell kein Modellelement aus dem alten Modell zugeordnet werden soll.

Eine Notation welche genau dies ausdrückt ist in Listing 4.1 angegeben. Zu beachten ist, dass Delta-Benutzervorgaben nicht für sämtliche Modellelemente spezifiziert werden müssen, sondern nur für Modellelemente, für die festgelegt werden soll, welchem anderen Modellelement sie zugeordnet werden sollen.

```
1 'none' -> de.Library
2 de.Book#bookExemplars -> 'none'
3 de.BookExemplar#isbn -> de.Book#isbn
4 de.Book#title -> de.Book#fullTitle
5 de.BookExemplar#isbn -> de.Book#fullIsbn
```

Listing 4.1: Delta-Benutzervorgaben in Form einer Abbildungsvorschrift.

Die erste Angabe in Listing 4.1 drückt aus, dass kein Modellelement in der Ursprungsversion dem Modellelement `de.Library` zugeordnet werden soll. Dies führt in der Differenzierungsphase dazu, dass dieses Modellelement als hinzugefügtes Modellelement betrachtet wird. Die Differenzierungsphase wird ausführlicher in Abschnitt 4.2.4 erläutert. Das gegenteilige Szenario wird durch die zweite Angabe ausgedrückt. In diesem Fall wird der Assoziation `de.Book#bookExemplars` keine andere Assoziation zugeordnet. Folglich wird `de.Book#bookExemplars` als gelöscht interpretiert. Die anderen drei Angaben drücken aus, dass ein existierendes Modellelement einem anderen existierenden Modellelement zugeordnet werden soll. So beschreibt die dritte Angabe den Fall, dass das Attribut `isbn` der Klasse `de.BookExemplar` in die Klasse `de.Book` verschoben wurde. Die vierte Angabe definiert überdies, dass das Attribut `title` der Klasse `de.Book` in `fullTitle` umbenannt wurde und die letzte Angabe kennzeichnet eine Umbenennung und Verschiebung eines Attributs.

Wie diese Beispiele bereits verdeutlichen, ist eine fundamentale Voraussetzung für die Formulierung von Delta-Benutzervorgaben, dass die betroffenen Modellelemente eindeutig adressiert werden können. Die Angabe einer Berechnungsvorschrift für den Namen eines Modellelements ist abhängig vom jeweils betrachteten Modelltyp und muss daher für jeden Modelltyp spezifisch entwickelt werden. Nachfolgend wird die Konstruktion von Namen für die wichtigsten Elemente von Klassendiagrammen angegeben:

- Jede Klasse eines Klassendiagramms kann eindeutig über den vollqualifizierten Pfad zur Klasse angesprochen werden. Dabei werden alle Pakete und der Name der Klasse selbst einbezogen. Beispielsweise liegt gemäß Listing 4.1 die in der ersten Delta-Benutzervorgabe referenzierte Klasse `Library` im Paket `de`.
- Jedes Attribut einer Klasse eines Klassendiagramms kann eindeutig über den vollqualifizierten Pfad bis zur Klasse zuzüglich des Namens des Attributs adressiert werden. So verweist das Quellelement der dritten Delta-Benutzervorgabe in Listing 4.1 auf das Attribut `isbn` der Klasse `BookExemplar`. In diesem Fall wird noch ein weiteres Trennzeichen verwendet, um Klassennamen und Attributnamen voneinander zu trennen.
- Für Assoziationen einer Klasse wird analog an den vollqualifizierten Namen der Quellklasse der Assoziation der Rollename des Ziellozationsendes angehängt. Die zweite Delta-Benutzervorgabe in Listing 4.1 veranschaulicht den Fall, dass implizit ein Name für eine Assoziationsrolle vergeben wurde (`bookExemplars`), da im Modell kein Name explizit angegeben war. Ist ein Rollename für das Ziellozationsende gesetzt, verhält es sich analog.

In dieser Arbeit wird vorausgesetzt, dass die Namen von Klassen in einem Klassendiagramm, die Namen von Attributen einer Klasse und die Rollennamen von Assoziationen eindeutig sind. Dies ist konform zu den Kontextbedingungen der UML/P [Sch12]. Daraus folgt, dass die Klassen, Attribute und Assoziationen bei einem validen Klassendiagramm eindeutig referenziert werden können.

In den Modellen die in dem Projekt, in dessen Kontext diese Arbeit durchgeführt wurde, eingesetzt wurden, konnten in einigen Fällen fehlerhafte Matchmodelle identifiziert werden. Dazu wurde der vorgestellte Differenzberechnungsansatz ohne Delta-Benutzervorgaben auf die Historie der Modelle angewendet. Die resultierenden Matchmodelle wurden mit Hinblick auf Fehler in den Matchmodellen analysiert. Dabei wurde festgestellt, dass die folgenden Szenarien zu fehlerhaften Matchmodellen geführt haben:

- Ein Modellelement wird als gelöscht oder hinzugefügt gemeldet, obwohl es verschoben oder umbenannt wurde (oder beides).
- Ein Modellelement wird als umbenannt oder verschoben (oder beides) gemeldet, obwohl ein Modellelement gelöscht und ein anderes Modellelement hinzugefügt wurde.

Auf Grundlage dieser Erkenntnisse wird nachfolgend die DSL Delta User Presetting Specification (DUPS) zur Definition von Delta-Benutzervorgaben vorgestellt. Der Vorteil dieser Variante ist, dass direkt ersichtlich ist, welche Modelländerungen bezüglich der Modellelemente durchgeführt wurden. Die Grammatik von DUPS ist in Listing 4.2 gezeigt.

DUPS stellt fünf verschiedene Operationen bereit, über die ausgedrückt werden kann, welche Modelländerungen durchgeführt wurden:

- `added addedElem`: Ein Modellelement wurde hinzugefügt (Zeile 13 – 15).
- `deleted deletedElem`: Ein Modellelement wurde gelöscht (Zeile 17 – 19).

```

1 grammar rwth.DUPS with org.eclipse.xtext.common.Terminals
2 generate dups "http://www.se.de/rwth/DUPS"
3
4 DeltaUserPresettings:
5     presettings += DeltaUserPresettingSpecification+;
6
7 DeltaUserPresettingSpecification:
8     DeltaUserPresettingAdd | DeltaUserPresettingDelete |
9     DeltaUserPresettingMove | DeltaUserPresettingRename |
10    DeltaUserPresettingMoveAndRename
11 ;
12
13 DeltaUserPresettingAdd:
14     'added' addedElem = ModelElemName';'
15 ;
16
17 DeltaUserPresettingDelete:
18     'deleted' deletedElem = ModelElemName';'
19 ;
20
21 DeltaUserPresettingMove:
22     'moved' movedSrc = ModelElemName ' to '
23     movedTgt = ModelElemName';'
24 ;
25
26 DeltaUserPresettingRename:
27     'renamed' renamedOrg = ModelElemName ' to '
28     renamedNew = ModelElemName';'
29 ;
30
31 DeltaUserPresettingMoveAndRename:
32     'moved and renamed' orgElem = ModelElemName ' to '
33     newElem = ModelElemName';'
34 ;
35
36 ModelElemName:
37     Name* ('.' Name)* ('#' Name)?
38 ;
39
40 Name :
41     (ID | '0..9')
42 ;

```

Listing 4.2: Grammatik der DSL DUPS.

- moved *movedSrc* to *movedTgt*: Ein Modellelement wurde verschoben (Zeile 21 – 24).
- renamed *renamedOrg* to *renamedNew*: Ein Modellelement wurde umbenannt (Zeile 26 – 29).

- moved and renamed *orgElem* to *newElem*: Ein Modellelement wurde verschoben und umbenannt (Zeile 31 – 34).

Listing 4.3 illustriert die Angabe von Delta-Benutzervorgaben in DUPS. Die Reihenfolge der Operationen in Listing 4.3 entspricht der Reihenfolge der Delta-Benutzervorgaben in Listing 4.1 auf Seite 45.

```
1 // 'none' -> de.Library
2 added de.Library;
3
4 // de.Book#bookExemplars -> 'none'
5 deleted de.Book#bookExemplars;
6
7 // de.BookExemplar#isbn -> de.Book#isbn
8 moved de.BookExemplar#isbn to de.Book;
9
10 // de.Book#title -> de.Book#fullTitle
11 renamed de.Book#title to fullTitle;
12
13 // de.BookExemplar#isbn -> de.Book#fullIsbn
14 moved and renamed de.BookExemplar#isbn to "de.Book.fullIsbn";
```

DUPS

Listing 4.3: Spezifikation von Modelländerungen für Delta-Benutzervorgaben.

Der Vorteil an der Nutzung von DUPS ist, dass die durchgeführten Modelländerungen direkt ersichtlich sind und nicht nur indirekt daraus hervorgehen, wie in Listing 4.1. Aus diesem Grund wird in dieser Arbeit die DSL DUPS zur Definition von Delta-Benutzervorgaben genutzt. Dessen ungeachtet wird auch diese alternative Darstellungsform intern als eine Abbildung von einem Quellelement auf ein Zielelement dargestellt. Die Matchingphase ist dadurch unabhängig davon, welche Variante zur Definition von Delta-Benutzervorgaben eingesetzt wird.

Solange die Zuordnung von Modellelementen zueinander korrekt ist, wird auch das resultierende Differenzmodell korrekt sein. Aus diesem Grund stehen nur solche Delta-Benutzervorgaben zur Verfügung, die beeinflussen, welches Modellelement welchem anderen zugeordnet wird. Beispielsweise steht keine Delta-Benutzervorgabe zur Verfügung, die ausdrückt, dass eine Multiplizität einer Assoziation angepasst wurde. Der Grund dafür ist, dass diese Änderung fehlerfrei abgeleitet werden kann, wenn bekannt ist, welche Assoziationen zueinander korrespondieren. In der Differenzierungsphase werden diese einander zugeordneten Assoziationen auf Unterschiede hin untersucht. Dieser Aspekt wird ausführlicher in Abschnitt 4.2.4 erläutert. In dem genannten Fall erkennt die Differenzkomponente, dass die Multiplizitäten verschieden sind und fügt einen entsprechenden Modellunterschied zum Differenzmodell hinzu.

Obgleich die Untersuchungen ausschließlich mit Klassendiagrammen durchgeführt wurden, sind die zugrundeliegenden Probleme unabhängig von Klassendiagrammen. Auch bei anderen Modelltypen führen vor allem Umbenennungen und Verschiebungen zu Problemen bei der Unterschiedsberechnung [PMR13]. Delta-Benutzervorgaben können daher auch in anderen Kontexten eingesetzt werden, vorausgesetzt die Modellelemente können eindeutig über einen Namen angesprochen werden.

Die Idee für Delta-Benutzervorgaben, wie sie in Listing 4.3 dargestellt sind, ist durch die Delta-Modellierung [HRRS11, HHK⁺13, HHK⁺15] motiviert. Delta-Modellierung ist ein sprachunabhängiger Ansatz zur Modellierung von Variabilität, bei dem über Delta-Operationen Änderungen eines Kernsystems ausgedrückt werden [MR14b]. Übliche Delta-Operationen betreffen das Hinzufügen, Löschen, Modifizieren oder Ersetzen eines Elements des Kernsystems. Das Kernsystem entspricht im Kontext dieser Arbeit der Ursprungsversion des Modells, mit dem ein Modell verglichen wird. Listing 4.4 zeigt, wie die Umbenennung eines Attributs in einem typischen Delta-Modellierungsansatz ausgedrückt wird. Diese Delta-Operation korrespondiert zu der Delta-Benutzervorgabe 'renamed de.Book#title to fullTitle'.

```

1 delta RenameBookTitle {
2   modify attribute de.Book.title {
3     Rename to fullTitle;
4   }
5 }

```

Delta

Listing 4.4: Spezifikation einer Delta-Operation zur Umbenennung eines Attributs.

Wichtig ist jedoch, dass die Delta-Modellierungsansätze einen anderen Fokus besitzen als Delta-Benutzervorgaben. Delta-Modellierung ist ein allgemeingültiger Ansatz zur Modellierung von Variabilität und ist nicht für die Modifikation von Match- oder Differenzmodellen gedacht.

Ablage und Ermittlung relevanter Delta-Benutzervorgaben

Bevor die Matchingphase ausgeführt werden kann, muss ermittelt werden, ob für die betrachteten Modelle Delta-Benutzervorgaben existieren. So wird automatisch eine Suche nach relevanten Delta-Benutzervorgaben in einem Versionskontrollsystem initiiert, falls die Modellierer beim Aufruf des Differenzberechnungswerkzeugs nicht direkt die zu verwendenden Delta-Benutzervorgaben übergeben haben.

Der Vorteil an der Ablage von Delta-Benutzervorgaben in einem Versionskontrollsystem oder einem anderen vergleichbaren Speicherort ist, dass sie automatisch ermittelt werden können, sobald die entsprechenden Modellversionen verglichen werden. Insbesondere gilt dies unabhängig davon, welcher Benutzer die Unterschiedsberechnung veranlasst hat. Angenommen, Delta-Benutzervorgaben wurden für die Unterschiedsberechnung zwischen den Modellversionen v_a und v_n spezifiziert. Das nächste Mal, wenn diese beiden Versionen wieder miteinander verglichen werden, können die bestehenden Delta-Benutzervorgaben wiederverwendet werden, ohne dass ein Benutzer dies explizit angeben muss. Durch die automatische Suche nach passenden Delta-Benutzervorgaben sind die Modellierer zudem nicht selbst dafür zuständig, stets die passenden Delta-Benutzervorgaben zu suchen und zu übergeben.

Damit die Delta-Benutzervorgaben auf diese Weise verarbeitet werden können, muss bekannt sein, auf welche Modellversionen die Delta-Benutzervorgaben bezogen sind. Um dies auf einfache Weise zu ermöglichen, wurde für die Benennung von Dateien, welche die Delta-Benutzervorgaben enthalten, eine einfache Namenskonvention etabliert: dup_<quellRevision>_-<zielRevision>. Die Datei dup_240_271 beinhaltet beispielsweise Delta-Benutzervorgaben, die sich auf den Vergleich zwischen der Modellversion 240 und der Modellversion 271 beziehen. Zusätzlich zu dieser Namenskonvention wurde noch eine weitere Namenskonvention

angewandt, um auf einfache Weise ableiten zu können, zu welchem konkreten Modell die Delta-Benutzervorgaben gehören. Zu diesem Zweck werden, falls das Modell im Versionskontrollsystem unter dem Pfad `https://<parent_folder>/<model>` abgelegt ist, die Delta-Benutzervorgaben unter dem Pfad `https://<parent_folder>/dup_<model>` abgelegt.

Unter Nutzung dieser beiden Konventionen können die Delta-Benutzervorgaben, die bei der Differenzberechnung zwischen zwei spezifischen Modellversionen betrachtet werden müssen, auf einfache Weise ermittelt werden.

Erweiterte Matchingphase

Die Integration von Delta-Benutzervorgaben in die Unterschiedsberechnung wird durch eine Erweiterung der Matchingphase erreicht. Der angepasste Matchingalgorithmus ist in Algorithmus 4.5 gezeigt. Bevor die eigentliche Berechnung des Matchmodells initiiert wird, werden die Delta-Benutzervorgaben bezogen, die für die untersuchten Modelle relevant sind (Zeile 1). Dieser Schritt wurde im vorhergehenden Abschnitt erläutert.

Algorithmus 4.5 : Function `matchModels()` - Um Delta-Benutzervorgaben erweitert

```

input  : modelOld: Old model version,
          modelNew: New model version
output : Match model

1 deltaUserPresettings ← getDeltaUserPresettings (modelOld, modelNew);
2 if !areDeltaUserPresettingsValid (deltaUserPresettings,
3   modelOld, modelNew) then
4   | abort model matching phase;
   /* ...original procedure... */

```

Falls Delta-Benutzervorgaben gefunden wurden, wird als Erstes validiert, ob die Delta-Benutzervorgaben nur auf Modellelemente verweisen, die in den Modellen existieren. Wird bei dieser Validierung festgestellt, dass auf ein nicht vorhandenes Modellelement verwiesen wird, wird der Benutzer darüber informiert und die Matchingphase wird abgebrochen. Ferner wird geprüft, ob die Delta-Benutzervorgaben widerspruchsfrei sind. Ein Widerspruch tritt auf, wenn in den Delta-Benutzervorgaben angegeben ist, dass ein Modellelement sowohl gelöscht als auch hinzugefügt wurde oder falls sich zwei verschiedene Vorgaben auf das gleiche Quell- oder Zielelement beziehen. Beispielsweise widerspricht die Delta-Benutzervorgabe `'renamed de.Y to X'` der Delta-Benutzervorgabe `'moved de.core.X to de'`, da beide Angaben unterschiedliche Quellelemente auf das gleiche Zielelement `de.X` abbilden. Analog zum vorherigen Fall wird der Benutzer über Widersprüche informiert und die Matchingphase abgebrochen. Diese Gültigkeitsprüfung ist in Algorithmus 4.5 in den Zeilen 2 – 3 dargestellt.

Im Anschluss daran wird die erweiterte Matchingphase angestoßen. In dieser wird, wie üblich, für Modellelemente der einen Modellversion nach korrespondierenden Modellelementen in der anderen Modellversion gesucht. Dies ist in Algorithmus 4.6 dargestellt. Ein Unterschied zur Vorgehensweise ohne Beachtung der Delta-Benutzervorgaben, die zuvor in Algorithmus 4.2 auf

Seite 39 vorgestellt wurde, ergibt sich genau dann, wenn ein Modellelement verarbeitet wird, für das eine Delta-Benutzervorgabe existiert (Zeile 2 – 11). In diesem Fall wird das Modellelement bestimmt, welches gemäß der Delta-Benutzervorgabe zu dem Modellelement korrespondiert (Zeile 4). Zu diesem Zweck wird zunächst unter den Elementen aus `listOld` nach einem solchen Modellelement gesucht. Sollte unter diesen nicht das passende Element enthalten sein, wird global nach dem zugehörigen Element gesucht. Die Zuordnung vom entsprechenden Quellelement zum zugehörigen Zielelement wird daraufhin zum Matchmodell hinzugefügt (Zeile 6). Sollte kein korrespondierendes Modellelement existieren, wird das Modellelement einzeln zum Matchmodell hinzugefügt (Zeile 10). Dies ist dann der Fall, wenn die Delta-Benutzervorgabe ausdrückt, dass das Modellelement als hinzugefügt oder gelöscht angesehen werden soll.

Algorithmus 4.6 : Function `mapLists()` - Um Delta-Benutzervorgaben erweitert

```

input  : listOld: List of model elements from old model version,
          listNew: List of model elements from new model version,
          matchModel: Match model
output : List of matches

1 Create empty list matches;
2 for each obj1 : listNew do
3   if existsDeltaUpForElement (obj1) then
4     obj2 ← findElementAccordingToDeltaUp (listOld, obj1);
5     if obj2 found then
6       addMatchElement (matches, obj2, obj1);
7       Remove obj1 from listNew;
8       Remove obj2 from listOld;
9     else
10      createUnmatchedElements (obj1, matchModel);
11      Remove obj1 from listNew;
12 Remove elements from listOld for which delta user presetting exists;
    /* ...original procedure... */
  
```

Bevor der ursprüngliche Matchingalgorithmus anschließend ausgeführt wird, werden vorab sämtliche Elemente aus `listOld` entfernt, die von einer Delta-Benutzervorgabe betroffen sind (Zeile 12). Auf diese Weise wird verhindert, dass ein anderes Modellelement aus der neueren Modellversion einem Modellelement in der älteren Modellversion, zu dem eine Delta-Benutzervorgabe existiert, zugeordnet wird. Andernfalls kann nicht sichergestellt werden, dass die Delta-Benutzervorgaben stets umgesetzt werden.

Die ursprüngliche Matchingphase wird also nur für Modellelemente unterbrochen, für die Delta-Benutzervorgaben existieren. In allen anderen Fällen wird die Matchingphase so vollzogen, als gäbe es keine Delta-Benutzervorgaben.

4.2.4 Durchführung der Differenzierungsphase

Das Ziel der Differenzierungsphase ist es aufbauend auf einem Matchmodell ein Differenzmodell zu erzeugen. Abbildung 4.12 zeigt die wichtigsten Elemente des EMFCompare-Metamodells für Differenzmodelle. Zur Vereinfachung der Beschreibung wurden die Namen mancher Metamodellklassen angepasst und die Struktur des Metamodells geringfügig geändert. Aus Platzgründen sind zudem die Assoziationen zu den Klassen `EObject`, `EReference` sowie `EAttribute` als Attribute modelliert.

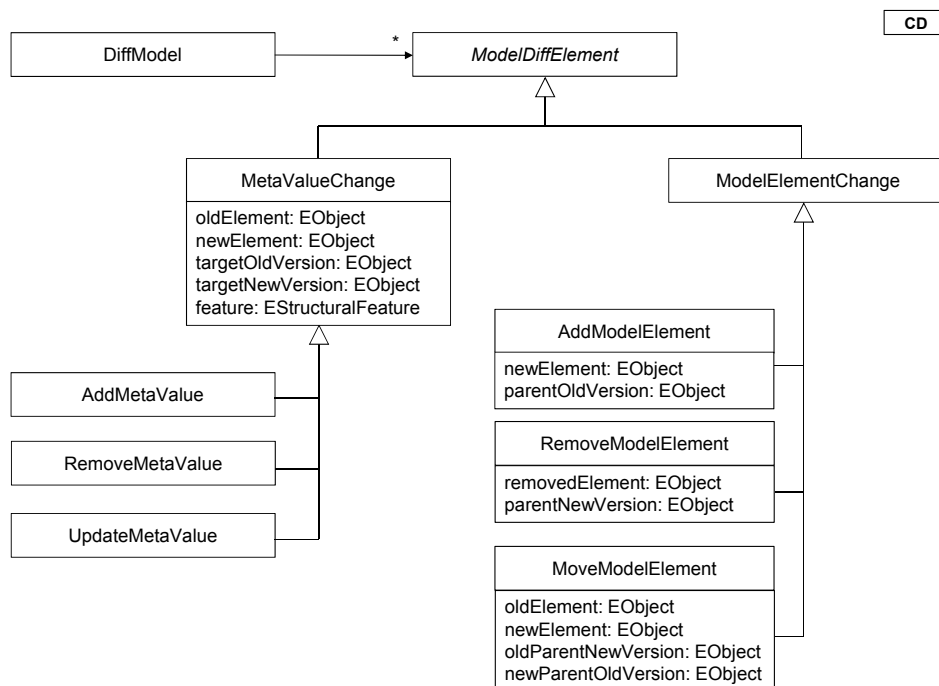


Abbildung 4.12: Metamodell für die Differenzmodelle dieser Arbeit.

Die Klasse `ModelDiffElement` bildet das Wurzelement des Metamodells. Jede Klasse, die einen Modellunterschiedstyp repräsentiert, ist von dieser Klasse abgeleitet. Es wird zwischen zwei Arten von Modellunterschieden differenziert:

- `MetaValueChange` Instanzen kapseln Modellunterschiede in den Metaattributen und Metaassoziationen eines Modellelements. Dazu zählt beispielsweise die Änderung des Namens oder des Supertypen einer Klasse. Das Attribut `feature` kapselt das betroffene Metaattribut bzw. die betroffene Metaassoziation.
- `ModelElementChange` Instanzen kapseln die weiteren Änderungen von Modellelementen bzw. der in diesen enthaltenen Modellelementen. Zu diesen zählt beispielsweise das Hinzufügen oder Entfernen einer Klasse oder eines Attributs.

Die Schritte zur Ableitung eines Differenzmodells aus einem Matchmodell sind nachfolgend dargelegt.

- Für ein Modellelement $me_{a,k}$ aus Version v_a , zu dem kein passendes Modellelement in Version v_n gefunden werden konnte: Es wird ein Modellunterschied zum Differenzmodell hinzugefügt, der ausdrückt, dass das Modellelement $me_{a,k}$ gelöscht wurde.
- Für ein Modellelement $me_{n,i}$ aus Version v_n , zu dem kein passendes Modellelement in Version v_a gefunden werden konnte: Es wird ein Modellunterschied zum Differenzmodell hinzugefügt, welcher ausdrückt, dass das Modellelement $me_{n,i}$ hinzugefügt wurde.
- Für ein Modellelement $me_{n,i}$ aus Version v_a , zu dem ein korrespondierendes Modellelement $me_{a,k}$ in Version v_n gefunden wurde: In diesem Fall ergeben sich die konkreten Modellunterschiede aus dem Vergleich von $me_{n,i}$ und $me_{a,k}$:
 - Der Wert jedes Metaattributs von $me_{n,i}$ wird mit dem Wert des zugehörigen Metaattributs von $me_{a,k}$ verglichen. Falls die Werte verschieden sind, wird ein Modellunterschied hinzugefügt, der kapselt, für welches Metaattribut die Werte verschieden sind und wie die verschiedenen Werte sind.
 - Analog wird verglichen, welche Objekte den Metaassoziationen von $me_{a,k}$ und welche Objekte den analogen Metaassoziationen von $me_{n,i}$ zugewiesen sind. Sind der gleichen Metaassoziation mr_r in $me_{a,k}$ und $me_{n,i}$ nicht die gleichen Objekte zugeordnet, wird ein Modellunterschied hinzugefügt, der ausdrückt, dass:
 - * ein Objekt o_p zu der Metaassoziation mr_r hinzugefügt wurde, falls o_p nur $me_{n,i}$, nicht aber $me_{a,k}$ zugeordnet ist.
 - * ein Objekt o_p aus der Metaassoziation mr_r gelöscht wurde, falls o_p nur $me_{a,k}$, nicht aber $me_{n,i}$ zugeordnet ist.
 - Falls das Elternelement von $me_{n,i}$, also das Element in dem $me_{n,i}$ unmittelbar enthalten ist, unterschiedlich zu dem Elternelement von $me_{a,k}$ ist, wird ein Modellunterschied hinzugefügt, der ausdrückt, dass $me_{n,i}$ verschoben wurde.

Beispiel

Die Konstruktion eines Differenzmodells auf Basis eines Matchmodells wird im Folgenden beispielhaft verdeutlicht. Abbildung 4.8 auf Seite 43 zeigt das Matchmodell, welches aus dem Vergleich der Modelle aus Abbildung 4.6 und Abbildung 4.7 auf Seite 42 entstanden ist.

Gemäß Matchmodell ist das Attribut `name` der Klasse `Library` ohne Zuordnung. Da es lediglich in der alten Version enthalten war, wird das Modellelement als gelöscht interpretiert. Folglich wird zum Differenzmodell eine Modelländerung vom Typ `RemoveModelElement` hinzugefügt. Für das zweite Attribut ohne Zuordnung, das Attribut `year` der Klasse `Book`, wird hingegen eine Modelländerung vom Typ `AddModelElement` zum Differenzmodell hinzugefügt. Der Grund dafür ist, dass das Modellelement lediglich in der neuen Version vorliegt und somit hinzugefügt worden sein muss. Ferner wird für die Klasse `library.core.Library` eine Umbenennung in `library.core.TextLibrary` erkannt, was darin resultiert, dass eine Instanz vom Typ `UpdateMetaValue` hinzugefügt wird. In dieser ist das Metaattribut `name` von der Modelländerung betroffen. Überdies werden für die Verschiebung der Klasse `Person`

und die des Attributs `isbn` zwei Instanzen vom Typ `MoveModelElement` zum Differenzmodell hinzugefügt. Abbildung 4.13 zeigt zusammenfassend das resultierende Differenzmodell.

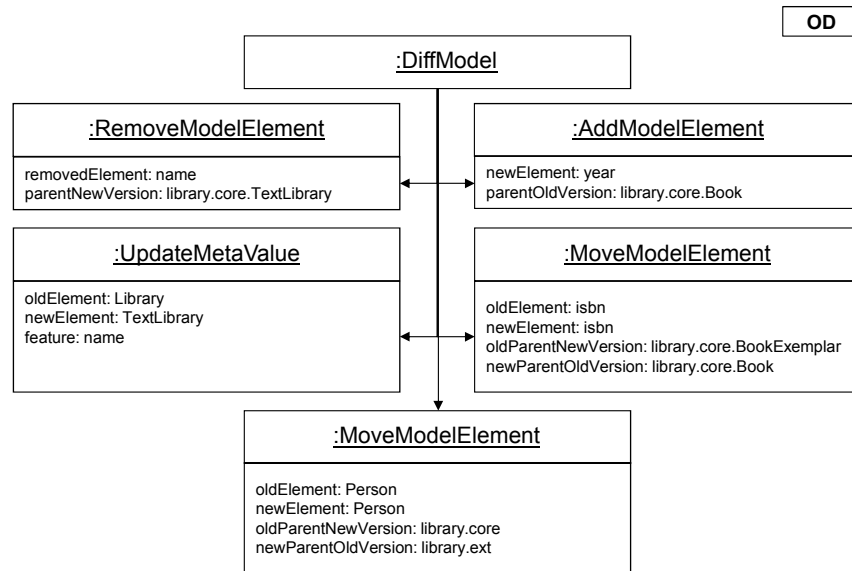


Abbildung 4.13: Differenzmodell für die Modelle aus Abbildung 4.6 und Abbildung 4.7 (auf Seite 42).

Kapitel 5

Auswirkungsanalyse zur Generierung von Checklisten

Ein fundamentales Problem bei der Weiterentwicklung eines Softwaresystems besteht darin, dass jede Änderung des Softwaresystems weitreichende Auswirkungen auf das gesamte Softwaresystem haben kann. In modellgetriebenen Softwareentwicklungsprojekten gilt das insbesondere für Änderungen der Modelle, aus denen oftmals große Teile eines Softwaresystems generiert werden. Um diesem Problem entgegenzuwirken, wird in dieser Arbeit ein modellbasierter Ansatz zur Auswirkungsanalyse vorgestellt. In diesem werden die Auswirkungen von Modelländerungen in expliziten Auswirkungsregeln (siehe Definition 9) modelliert. Die Grundlage für dieses Kapitel bilden die beiden Publikationen [MR14a, MR15].

Definition 9 (Auswirkungsregel)

Eine Auswirkungsregel beschreibt die Auswirkungen von Modelländerungen auf ein Softwaresystem. Dazu definiert eine Auswirkungsregel Bedingungen und legt dar, welche Auswirkungen eine Modelländerung hat, wenn sie eine der Bedingungen erfüllt.

Zur Modellierung von Auswirkungsregeln wird in dieser Arbeit die DSL Impact Rule Specification (IRS) eingeführt. Der große Vorteil bei der expliziten Modellierung von Auswirkungsregeln besteht darin, dass es möglich ist Expertenwissen zu Abhängigkeiten oder sonstigen Charakteristiken eines Softwaresystems explizit festzuhalten.

Sobald die Änderungen bekannt sind, die für ein Modell durchgeführt wurden (siehe Kapitel 4), können die Auswirkungsregeln auf die identifizierten Modelländerungen angewendet werden. Erfüllt eine Modelländerung eine der Bedingungen, die in einer Auswirkungsregel spezifiziert sind, kann der zugehörige Beschreibungstext der Auswirkung an die Entwickler gemeldet werden. Die verschiedenen relevanten Beschreibungstexte werden für die Entwickler gebündelt in einer Checkliste (siehe Definition 10) aufbereitet.

Definition 10 (Checkliste)

Eine Checkliste beschreibt die Auswirkungen von Modelländerungen. Konkret kapselt eine Checkliste Hinweise zu den Entwicklungsaktivitäten, die in einem Softwaresystem aufgrund der Modelländerungen durchgeführt werden müssen.

Im Zuge des Evolutionsprozesses können diese Checklisten von Entwicklern abgearbeitet und abgehakt werden. Die Checklisten bieten dadurch einen strukturierten Ansatz zur Weiterentwicklung eines Softwaresystems. Die Motivation für die Erzeugung von Checklisten ist, dass Entwickler sich möglicherweise nicht immer an sämtliche Entwicklungsaktivitäten erinnern, die

nach spezifischen Modelländerungen durchgeführt werden müssen. Das gilt vor allem in komplexen Softwaresystemen. Dies wurde in Diskussionsrunden von Entwicklern berichtet. Darüber hinaus können die Entwickler die notwendigen Änderungen gezielter ausführen, da die Checkliste auf diese hinweist. Beispielsweise kann die Checkliste direkt die Artefakte auflisten, die anzupassen sind, so dass Entwickler nicht selbst nach diesen suchen müssen. Durch die explizite Definition von Produktwissen in Auswirkungsregeln wird einem weiteren Problem entgegengewirkt: falls Entwickler ein Unternehmen verlassen, kann ihr Expertenwissen zumindest partiell verloren gehen. Durch die explizite Definition des Expertenwissens in Auswirkungsregeln wird dieses Problem zumindest etwas abgeschwächt.

In Abschnitt 5.1 wird das Konzept der Checkliste genauer eingeführt. Daraufhin wird in Abschnitt 5.2 einleitend ein Beispiel für eine Auswirkungsregel und darauf aufbauend ein Beispiel für eine Checkliste vorgestellt. Abschnitt 5.3 gibt anschließend einen Überblick über den Ansatz zur Generierung von Checklisten. Der Ansatz wird darauffolgend detaillierter in Abschnitt 5.4 vorgestellt. Als Nächstes folgt eine Diskussion verschiedener Aspekte des Ansatzes in Abschnitt 5.5, mit anschließender Darlegung verwandter Arbeiten in Abschnitt 5.6. Abschließend wird in Abschnitt 5.7 eine Fallstudie beschrieben, in welcher der Ansatz evaluiert wurde.

5.1 Checkliste

Eine Checkliste wird stets unter Anwendung von Auswirkungsregeln auf eine Menge von Modellunterschieden generiert. Die betrachteten Modellunterschiede resultieren aus der Differenzberechnung zwischen zwei Modellen bzw. Modellversionen. Eine Checkliste liefert dadurch Aufschluss darüber, welche Entwicklungsaktivitäten aufgrund der identifizierten Modelländerungen durchgeführt werden müssen. Das Hauptziel einer Checkliste ist es, durch die explizite Auflistung der Auswirkungen der Modelländerungen die Evolution des Softwaresystems zu vereinfachen.

Jede Checkliste ist aus verschiedenen Bereichen aufgebaut. Diese werden im Folgenden zusammengefasst:

- **Allgemeine Informationen:** Zu Beginn einer Checkliste ist angegeben, wann die Checkliste erzeugt wurde und welche Modelle bzw. Modellversionen im Zuge der Checklisten-erzeugung verglichen wurden.
- **Auflistung der identifizierten Modelländerungen:** Anschließend werden die Modelländerungen aufgelistet, auf Basis derer die Checkliste generiert wurde. Diese Modelländerungen beziehen sich auf die Modelle bzw. Modellversionen, die vorab in den allgemeinen Informationen angegeben sind. Die identifizierten Modelländerungen sind für eine Checkliste von zentraler Bedeutung, da sich die von den Auswirkungsregeln erzeugten Beschreibungen der Auswirkungen nach der Art der festgestellten Modelländerungen richten.
- **Übersicht über die genutzten Auswirkungsregeln:** Darauffolgend wird dargelegt, welche Auswirkungsregeln zur Erzeugung der Checkliste herangezogen wurden. Vor dem Aufruf der Checklistengenerierung kann konfiguriert werden, welche Auswirkungsregeln einge-

setzt werden sollen. Da somit nicht immer alle verfügbaren Auswirkungsregeln zum Einsatz kommen, werden die verwendeten Auswirkungsregeln aufgeführt.

- **Ergebnisse der Auswirkungsregeln:** Für jede verwendete Auswirkungsregel ist einleitend eine Beschreibung angegeben. Diese fasst zusammen, welchen Zweck die Auswirkungsregel hat. Überdies kann eine Checkliste Informationen darüber enthalten, wie kritisch die Durchführung der Entwicklungsaktivitäten ist, auf die durch die Auswirkungsregel hingewiesen wird. Weiterhin kann eine Checkliste darlegen, wie wahrscheinlich es ist, dass die von der Auswirkungsregel gemeldeten Entwicklungsaktivitäten tatsächlich durchgeführt werden müssen. Sämtliche genannten Angaben werden von den Modellierern der Auswirkungsregel selbst definiert und aus der modellierten Auswirkungsregel in die Checkliste übernommen. Darauf folgend sind die Checklistenhinweise aufgeführt, die durch die betrachtete Auswirkungsregel gemeldet wurden. Diese beschreiben, welche Entwicklungsaktivitäten aufgrund der jeweiligen Auswirkungsregel durchzuführen sind. Dieser Teil einer Checkliste ist abhängig davon, welche Auswirkungen die Modellierer für die verschiedenen Bedingungen definiert haben. Zusätzlich zu den Checklistenhinweisen selbst wird durch den Checklistengenerator zudem eine Beschreibung der Modelländerung, die zur Erzeugung des jeweiligen Checklistenhinweises geführt hat, in die Checkliste eingefügt. Diese Beschreibung wird angegeben, damit die Entwickler ein besseres Verständnis für die Gründe der in der Checkliste aufgeführten Entwicklungsaktivitäten haben.

5.2 Beispiel

In diesem Abschnitt wird ein konkreter Anwendungsfall beschrieben, in dem eine Auswirkungsregel helfen kann, die Evolution von Authoring zu unterstützen. Der Fokus liegt dabei auf der Verdeutlichung des Nutzens der Auswirkungsregel und nicht auf einer detaillierten Modellierung der Regel. Die reale Spezifikation der Auswirkungsregel erfolgt später in Abschnitt 5.4.3. Dort sind darüber hinaus weitere konkrete Beispiele zu Auswirkungsregeln samt der zugehörigen Modellierung der Auswirkungsregeln dargelegt. Ferner wird in diesem Abschnitt ein Beispiel für eine Checkliste gezeigt, die unter Anwendung dieser Auswirkungsregel generiert wurde. Die konkrete Vorgehensweise zur Generierung von Checklisten wird in Abschnitt 5.3 sowie Abschnitt 5.4 detailliert erklärt.

5.2.1 Auswirkungsregel zur Analyse von objektrelationalen Mapping-Dateien (ORM-AR)

Authoring verwendet einen objektrelationalen Mapper (engl. object relational mapping, ORM), um das objektorientierte Modell auf eine relationale Datenbank abzubilden. Zur Definition dieser Abbildung setzen ORMs Mapping-Dateien ein, in denen diese Abbildung genau beschrieben ist. Die in den Mapping-Dateien referenzierten Entitätsklassen sind dabei aus Klassendiagrammen generiert, wie in Kapitel 2 eingeführt. Folglich können Änderungen in den Klassendiagrammen zu Änderungen in den Entitätsklassen führen und somit erfordern, dass die Mapping-Dateien ebenfalls angepasst werden müssen.

Die Auswirkungsregel ORM-AR kapselt genau die Bedingungen, die Änderungen in den Mapping-Dateien erfordern. Konkret weist sie darauf hin, welche Änderungen in den Klassendiagrammen dazu führen, dass neue Elemente zu den Mapping-Dateien hinzugefügt, aus diesen entfernt oder in diesen aktualisiert werden müssen.

Die Motivation für die Einführung der Auswirkungsregel ORM-AR ist, dass der Zugang zur Datenbank nur dann ordnungsgemäß funktioniert, wenn die Einträge in den Mapping-Dateien korrekt sind und keine Einträge fehlen. Fehler in den Mapping-Dateien oder fehlende Einträge können daher problematisch für das gesamte System und folglich auch für die Benutzer des Systems sein. Indem die Auswirkungsregel ORM-AR auf die notwendigen Anpassungen der Mapping-Dateien verweist, kann sie die Entwickler somit bei der Evolution von Authoring unterstützen.

Listing 5.1 zeigt eine vereinfachte Modellierung für diese Auswirkungsregel. Die vollständige Modellierung ist in Abschnitt 5.4.3 dargelegt, nachdem die DSL IRS in Abschnitt 5.4.1 vollständig eingeführt wurde. Zu Beginn der Auswirkungsregel ist zunächst eine Beschreibung der Auswirkungsregel angegeben (Zeile 2 – 4). Anschließend sind auf der linken Seite (Zeile 9, 12) die Bedingungen der Auswirkungsregel spezifiziert. Auf der zugehörigen rechten Seite (Zeile 10, 13 – 14) steht der Text, der zu der Checkliste hinzugefügt wird, wenn die vorangestellte Bedingung erfüllt ist. Aus der vereinfachten Modellierung geht hervor, dass nach dem Löschen eines persistenten Modellelements oder dem Umbenennen eines persistenten Attributs ein Hinweis zur Checkliste hinzugefügt werden muss. Beiden Checklistenhinweisen ist das Gruppenkürzel `ForClass` zugeordnet (Zeile 10, 13). Sämtliche Checklistenhinweise, denen der gleiche Gruppentext zugeordnet ist, erscheinen in der resultierenden Checkliste gebündelt zu diesem Gruppentext.

```

1 impactRule "ORM File Analysis" {
2   description = "This rule checks whether it is necessary to
3     change an object relational mapping file by adding entries,
4     changing existing entries or deleting existing entries."
5
6   impact {
7     group ForClass "Entries concerning class {Class}:"
8
9     pc.concernsPersistentModelElement() && pc.deletedModelElement()
10      => group ForClass, hint "Delete entry from mapping file."
11
12     pc.concernsPersistentModelElement() && pc.renamedAttribute()
13      => group ForClass, hint "Rename old entry {OldEntry}
14        to {NewEntry} in mapping file."
15   }
16 }

```

Listing 5.1: Vereinfachte Modellierung der Auswirkungsregel ORM-AR.

In der gezeigten Spezifikation des Gruppentextes zum Gruppenkürzel `ForClass` ist ein sogenannter Platzhalter enthalten (Zeile 7). Dieser trägt den Namen `Class` und wird zur Laufzeit der Checklistengenerierung durch den Namen der Klasse ersetzt, die von dem konkreten Checklistenhinweis betroffen ist. Auf diese Weise sind sämtliche Hinweise, welche die gleiche Klasse

betreffen, gebündelt in der Checkliste angegeben. Ferner enthält der Checklistenhinweis zur zweiten Bedingung zwei Platzhalter `OldEntry` (Zeile 13) und `NewEntry` (Zeile 14). Diese werden zur Laufzeit der Checklistengenerierung durch den alten und neuen Namen des umbenannten Attributs ersetzt. Im nächsten Abschnitt wird eine Checkliste gezeigt, die unter Nutzung dieser Auswirkungsregel generiert wurde. Diese Checkliste verdeutlicht die Bedeutung einer Gruppenzuordnung und zeigt zudem die Auswirkungen der Ersetzung eines Platzhalters.

5.2.2 Checkliste zur Auswirkungsregel ORM-AR

Listing 5.2 zeigt eine Checkliste, die unter Nutzung der Auswirkungsregel ORM-AR generiert wurde. Eine vollständige Checkliste, die unter Anwendung sämtlicher Auswirkungsregeln aus Abschnitt 5.4.3 generiert wurde, ist im Anhang in Listing C.1 aufgeführt. Für das Beispiel wird angenommen, dass folgende Modelländerungen vorgenommen wurden:

- Aus der Klasse `core.Library` wurde das Attribut `name` entfernt.
- Das Attribut `title` aus der Klasse `core.Book` wurde in `fullTitle` umbenannt.
- Das Attribut `year` wurde aus der Klasse `core.Book` gelöscht.
- Das Attribut `isbn` wurde aus der Klasse `core.BookExemplar` in die Klasse `core.Book` verschoben.
- Die Klasse `ext.Address` wurde gelöscht.

Die in der Checkliste aufgeführten Modelländerungen (Zeile 5 – 11) entsprechen genau den zuvor genannten Modelländerungen. Hervorzuheben hierbei ist, dass diese Modelländerungen aus dem Vergleich der beiden in der Checkliste genannten Modellversionen (Zeile 2 – 3) resultieren. Weiterhin zeigt die Checkliste auf, dass ausschließlich die Auswirkungsregel ORM-AR zur Generierung der Checkliste genutzt wurde (Zeile 13 – 15).

Anschließend zeigt die Checkliste die Ergebnisse zur Auswirkungsregel ORM-AR (Zeile 17 – 36). Eingeleitet werden diese durch eine Beschreibung zur Auswirkungsregel (Zeile 19 – 21), die unverändert aus dem Modell zu ORM-AR entnommen ist (siehe Listing 5.1, Zeile 2 – 4). Darauffolgend sind die Checklistenhinweise zu ORM-AR gebündelt aufgeführt (Zeile 23 – 36). Gemäß dieser Hinweise müssen die Entwickler drei Einträge aus den Mapping-Dateien entfernen (Zeile 24 – 25, 31 – 32, 35 – 36) und einen Eintrag umbenennen (Zeile 28 – 30). Dies ist dadurch bedingt, dass drei Modellelemente gelöscht wurden und ein Modellelement umbenannt wurde. Die Information darüber, welche Modelländerung zu der Erzeugung eines Checklistenhinweises geführt hat, wird automatisch im Zuge der Checklistengenerierung hinzugefügt, wie zuvor bereits in Abschnitt 5.1 eingeführt.

Für die Verschiebung des Attributs `isbn` aus der Klasse `core.BookExemplar` in die Klasse `core.Book` wurde durch ORM-AR hingegen kein Checklistenhinweis erzeugt. Der Grund dafür ist, dass in der vereinfachten Modellierung von ORM-AR (siehe Listing 5.1) diese Bedingung nicht aufgeführt ist und somit nicht von ORM-AR beachtet wird.

Wie zuvor bereits erläutert, können die Checklistenhinweise einer Auswirkungsregel verschiedenen Gruppen zugeordnet sein. Gruppen ermöglichen es mehrere Checklistenhinweise

logisch zusammenzufassen. Dies zeigt sich in der Checkliste in Listing 5.2 daran, dass zur Klasse `core.Book` zwei Änderungen aufgeführt sind (Zeile 27–32). Bei der Gruppenbeschreibung und den Hinweisen in der Checkliste gilt es zu beachten, dass die Platzhalter aus dem Modell zu ORM-AR zur Laufzeit ersetzt wurden. So zeigen die Gruppenüberschriften in der Checkliste den Namen der betroffenen Klasse anstelle des Platzhalters `Class`. Ferner zeigt der Hinweis zu der Umbenennung des Eintrags der Mapping-Datei den alten und neuen Namen anstelle der Platzhalter `OldEntry` und `NewEntry`. In Abschnitt 5.4 wird dargelegt, wie die Ersetzung der Platzhalter genau erfolgt.

```

1 Creation time: 06.12.2016, 14:53
2 Old model: library, revision 4
3 New model: library, revision 5
4
5 Found model differences:
6 -----
7 - Deleted attribute 'core.Library#name' (type: String)
8 - Renamed attribute 'core.Book#title' (type: String) to 'fullTitle'
9 - Deleted attribute 'core.Book#year' (type: int)
10 - Moved attribute 'core.BookExemplar#isbn' to class 'core.Book'
11 - Deleted class 'ext.Address'
12
13 Used impact rules:
14 -----
15 - ORM File Analysis
16
17 ORM File Analysis:
18 =====
19 This rule checks whether it is necessary to change an object
20 relational mapping file, by adding entries, changing existing
21 entries or deleting existing entries.
22
23 Entries concerning class 'core.Library':
24 - Delete entry from mapping file. (Reason: Deleted attribute
25   'core.Library#name' (type: String))
26
27 Entries concerning class 'core.Book':
28 - Rename old entry 'title' to 'fullTitle' in mapping file.
29   (Reason: Renamed attribute 'core.Book#title' (type: String)
30   to 'fullTitle')
31 - Delete entry from mapping file. (Reason: Deleted attribute
32   'core.Book#year' (type: int))
33
34 Entries concerning class 'ext.Address':
35 - Delete entry from mapping file. (Reason: Deleted class
36   'ext.Address')
```

Listing 5.2: Beispielhafte Checkliste unter Nutzung der Auswirkungsregel ORM-AR.

5.3 Methodik zur Generierung von Checklisten

Der Ansatz zur Generierung von Checklisten besteht aus drei Hauptschritten: 1. der Bereitstellung von Auswirkungsregeln, 2. der Identifikation von Modellunterschieden und 3. der Erzeugung einer Checkliste basierend auf den Auswirkungsregeln und den Modellunterschieden.

Nachfolgend wird ein detaillierter Überblick über die Schritte zur Generierung einer Checkliste gegeben. Abbildung 5.2 veranschaulicht die einzelnen Teilschritte. Dabei entsprechen die Schritte in der Abbildung den Schritten in der nachfolgenden Aufzählung. Aus Gründen der Übersichtlichkeit wurden in Abbildung 5.2 und in der Aufzählung einige Teilschritte zur Bestimmung der Modellunterschiede weggelassen und werden nachfolgend nur kurz erläutert. Diese Phase des Ansatzes ist im Detail in Kapitel 4 beschrieben.

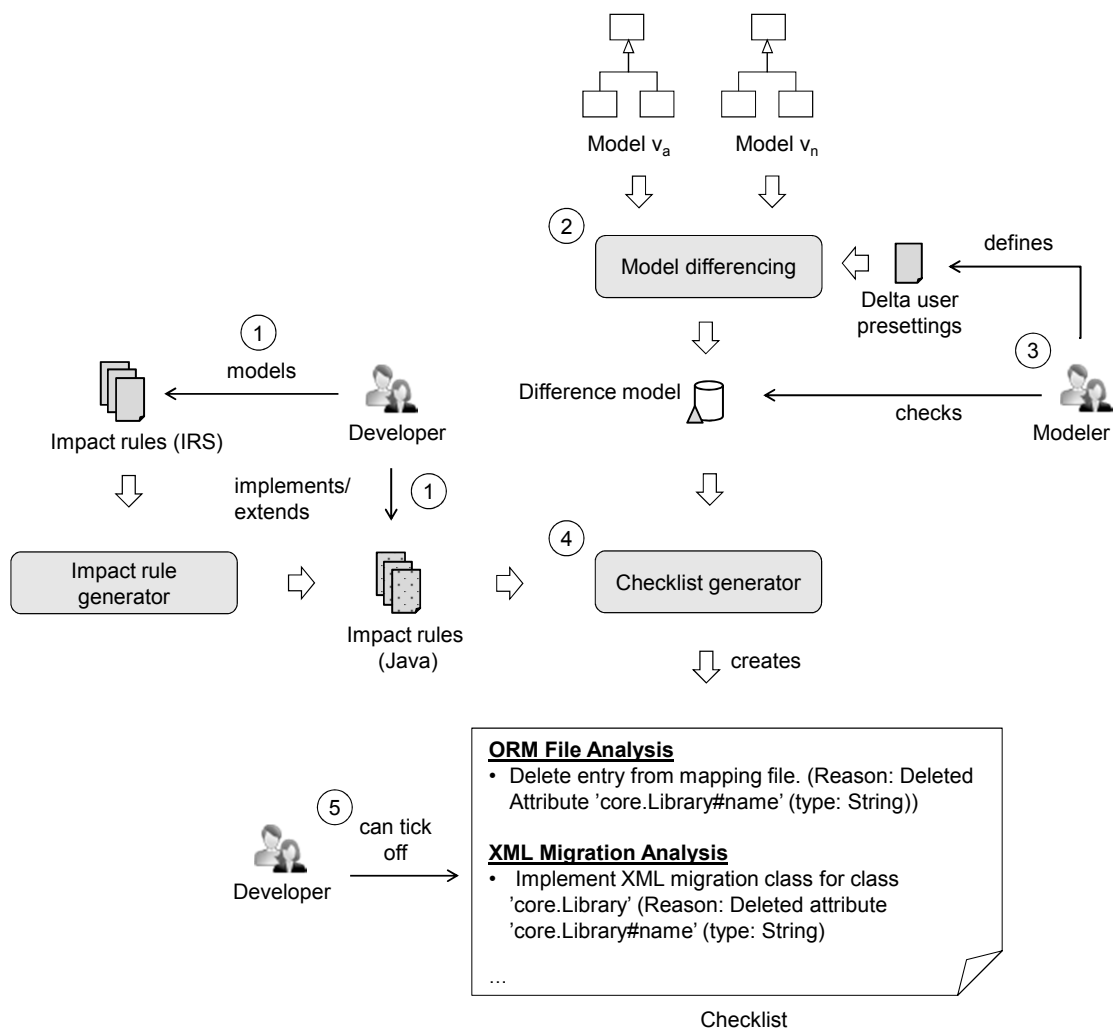


Abbildung 5.2: Überblick über den Ansatz zur Generierung von Checklisten.

1. Die Entwickler modellieren Auswirkungsregeln unter Nutzung der DSL IRS. Auf Basis dieser Modelle werden über einen Generator Java-Implementierungen der Auswirkungsregeln generiert. Diese generierten Implementierungen können über handgeschriebene Klassen erweitert werden. Ferner können Entwickler selbstständig Auswirkungsregeln direkt in Java implementieren, ohne diese modellieren zu müssen. Als Resultat liegen Implementierungen von Auswirkungsregeln vor, die im vierten Schritt vom Checklisten-generator eingesetzt werden.
2. Ein Differenzberechnungswerkzeug bestimmt die Modelländerungen, die zwischen zwei Modellversionen v_a und v_n eines Modells durchgeführt wurden. Sämtliche identifizierten Modelländerungen sind in einem Differenzmodell gekapselt, welches das Ergebnis der Differenzberechnung darstellt. Die später resultierende Checkliste gibt Aufschluss darüber, welche Entwicklungsaktivitäten aufgrund der Modelländerungen aus diesem Differenzmodell durchgeführt werden müssen.
3. Die Modellierer prüfen das resultierende Differenzmodell. Stellen sie dabei fest, dass das Differenzmodell fehlerhafte Modelländerungen beinhaltet, müssen sie Delta-Benutzervorgaben spezifizieren und diese bereitstellen, wie in Abschnitt 4.2.3 erklärt. Danach muss die Checklistengenerierung neu initiiert werden. Andernfalls ist die daraus resultierende Checkliste unbrauchbar, da sie auf Basis von fehlerhaften Modelländerungen unter Umständen fälschlicherweise auf spezifische Entwicklungsaktivitäten verweist.
4. Aufbauend auf einem Differenzmodell und den im ersten Schritt bereitgestellten Implementierungen von Auswirkungsregeln wird eine Checkliste generiert. Der Pseudocode des zugrundeliegenden Algorithmus ist in Algorithmus 5.1 angegeben. Zur Erzeugung einer Checkliste traversiert der Checklistengenerator für jede Auswirkungsregel jeden Modellunterschied des Differenzmodells (Zeile 3 – 9). In jeder Auswirkungsregel können mehrere Einträge definiert sein, die jeweils aus Bedingungen und einer zugehörigen Beschreibung der Auswirkungen bestehen. Der Checklistengenerator untersucht alle vorhandenen Einträge (Zeile 4 – 9) und prüft für jeden Modellunterschied, ob die Bedingungen erfüllt sind, die im jeweiligen Eintrag spezifiziert sind (Zeile 5 – 6). Falls der Modellunterschied die Bedingungen erfüllt, wird ein Tupel gespeichert, welches aus der Auswirkungsbeschreibung und dem Modellunterschied besteht (Zeile 9). Der Modellunterschied wird im Tupel gespeichert, um innerhalb der Checkliste angeben zu können, welcher Modellunterschied zur Erzeugung eines Checklistenhinweises geführt hat. Ein Beispiel hierfür ist in Zeile 24 – 25 in Listing 5.2 auf Seite 60 zu sehen. Dieses Tupel wird einer Gruppenbeschreibung zugeordnet (Zeile 9), damit später bekannt ist, welche Auswirkungsbeschreibungen zu welcher Gruppe gehören. Ist für einen Checklistenhinweis nicht explizit eine Gruppe angegeben, wird ein Checklistenhinweis einer Standardgruppe zugeordnet.

Falls die identifizierten Modellunterschiede auf mindestens eine der in der Auswirkungsregel definierten Bedingungen zugetroffen haben (Zeile 10), wird die Checkliste um die Hinweise zu der aktuell untersuchten Auswirkungsregel erweitert (Zeile 11 – 19). Dazu wird zu der Checkliste hinzugefügt:

- Der Name der Auswirkungsregel (Zeile 11).

- Eine Beschreibung zur Auswirkungsregel (Zeile 12).
 - Die Namen der Gruppen, die identifiziert wurden (Zeile 14). Zu jeder Gruppe werden die Checklistenhinweise angegeben, die der Gruppe zugeordnet sind. Dabei wird ein erweiterter Checklistenhinweis erzeugt (Zeile 17 – 18). Dieser besteht aus dem Hinweistext, der in der Auswirkungsregel angegeben ist und der um eine Beschreibung des zugehörigen Modellunterschieds erweitert wird. Zu der Standardgruppe werden die Hinweise hingegen direkt aufgelistet, ohne einleitende Gruppenbeschreibung.
5. Die generierte Checkliste, die in Form einer Textdatei vorliegt, wird abschließend an die Entwickler weitergereicht. Diese können die aufgeführten Entwicklungsaktivitäten daraufhin durchführen und die Checkliste abhaken.

Algorithmus 5.1 : Function generateChecklist()

```

input : diffModel: Difference model,
        impactRules: Impact rules
output : Checklist generated by applying impact rules on difference model

1 for each ir : impactRules do
2   Create empty map groupIRMap;
3   for each diff : diffModel do
4     for each irEntry : getImpactEntries (ir) do
5       if modelDiffFulfillsCondition (diff,
6         getCondition (irEntry) ) then
7         hint ← getChecklistHint (irEntry);
8         /* If no group is explicitly stated, default group
9           is assumed */
10        groupName ← getChecklistHintGroup (irEntry);
11        Add <groupName, <hint, diff > > to groupIRMap;
12
13 if groupIRMap is not empty then
14   addToChecklist (getName (ir) );
15   addToChecklist (getDescription (ir) );
16   for each groupName : groupIRMap do
17     addToChecklist (groupName);
18     hintsForIR ← getHintsForGroup (groupName);
19     for each hintEntry : hintsForIR do
20       extHint = createHint (getHint (hintEntry),
21         getMDDesc (hintEntry) );
22       addToChecklist (extHint);
23
24 return checklist;

```

5.4 Ansatz zur regelbasierten Auswirkungsanalyse

Zur Erhöhung der Verständlichkeit von Auswirkungsregeln wurde die DSL IRS entwickelt. Mittels IRS kann definiert werden, welche Bedingungen erfüllt sein müssen, damit beim Erkennen einer bestimmten Art von Modelländerung ein Checklistenhinweis erzeugt wird. Die Definition über IRS ist die empfohlene Variante zur Spezifikation von Auswirkungsregeln. Dessen ungeachtet kann auch die vollständige oder partielle manuelle Implementierung einer Auswirkungsregel in Java hilfreich sein. Der Grund dafür ist, dass das Expertenwissen, welches durch die Auswirkungsregel ausgedrückt wird, dann ebenfalls explizit formuliert ist. Dieses Wissen kann darauffolgend wiederverwendet werden, um die Evolution eines Systems zu vereinfachen.

5.4.1 Spezifikation von Auswirkungsregeln

Die Grammatik der DSL IRS ist in zwei Teile gegliedert: 1. einen beschreibenden Teil zur Auswirkungsregel und 2. die Definition der Bedingungen, unter denen Modelländerungen Auswirkungen haben. Listing 5.3 zeigt den Teil der Grammatik, in dem eine Auswirkungsregel allgemein beschrieben wird.

```

1 grammar rwth.IRS with org.eclipse.xtext.common.Terminals
2 generate iRS "http://www.se.de/rwth/IRS"
3
4 ImpactRule:
5   'impactRule' name = STRING '{'
6     'description' description = STRING
7     ('severity = ' severity = ('minor' | 'normal' | 'critical'))?
8     ('probability = ' probability = ('low' | 'medium' | 'high'))?
9     ('relevantFor = ' relevantFor = STRING)?
10
11     (impactSpec = ImpactSpecificationBlock)?
12   '}'
13 ;

```

Listing 5.3: Teil der Grammatik der DSL IRS, in dem eine Auswirkungsregel allgemein beschrieben wird.

Wie in dieser Grammatik zu sehen, besitzt jede Auswirkungsregel einen Namen (Zeile 5) und eine Beschreibung (Zeile 6), die deutlich macht, auf welche Arten von Auswirkungen die Regel sich bezieht. Darüber hinaus kann ein Schweregrad angegeben werden (Zeile 7). Dieser kennzeichnet, wie kritisch die Durchführung der Entwicklungsaktivitäten ist, auf die durch die Auswirkungsregel hingewiesen wird. Mögliche Werte hierfür sind „minor“, „normal“ und „critical“. Das Wissen über den Schweregrad kann von Entwicklern dazu genutzt werden die Reihenfolge, in der die verschiedenen Entwicklungsaktivitäten durchgeführt werden, zu bestimmen. Weiterhin kann für eine Auswirkungsregel angegeben werden, wie wahrscheinlich es ist, dass die gemeldeten Entwicklungsaktivitäten tatsächlich ausgeführt werden müssen (Zeile 8). Als mögliche Werte können „low“, „medium“ oder „high“ genutzt werden. Zusammen mit der Information bezüglich des Schweregrads kann diese Angabe zur Priorisierung der dargelegten Schritte eingesetzt werden. Zusätzlich können für jede Auswirkungsregel mehrere Personen oder

Rollen von Personen angegeben werden, für die von der Regel gemeldete Hinweise relevant sind (Zeile 9). Auf diese Weise können personenspezifische Checklisten generiert werden, so dass Entwickler nur die für sie bzw. ihre Rolle relevanten Checklistenhinweise sehen.

Im Anschluss an den beschreibenden Teil einer Auswirkungsregel können in einem Bedingungsblock (Zeile 11) die Bedingungen definiert werden, unter denen Modelländerungen Auswirkungen haben. Listing 5.4 zeigt den zugehörigen Teil der IRS-Grammatik. Zur Vereinfachung der Beschreibung wurden in der gezeigten Grammatik einige Xtext-Details weggelassen. Die vollständige Grammatik für IRS ist in Listing E.2 auf Seite 324 aufgeführt.

```

1 ImpactSpecificationBlock:
2   'impact' (incomplete ?= '[incomplete]')? '{'
3     (groupSpec += GroupSpecification)*
4     (impactSpec += ImpactSpecification)+
5   '}'
6 ;
7
8 GroupSpecification:
9   'group ' name = ID STRING ';'
10 ;
11
12 ImpactSpecification:
13   condition = ImpactRuleCondition '=>'
14     ('group ' (abbrGroup = [GroupSpecification | ID] |
15       group = STRING)', hint')? impactDesc = STRING
16 ;
17
18 ImpactRuleCondition:
19   BooleanExpr
20 ;
21
22 BooleanExpr:
23   PrefixedExpr (op = ('&&' | '||') PrefixedExpr)*;
24 ;
25
26 PrefixedExpr:
27   Atomic | negation = '!' Atomic;
28 ;
29
30 Atomic:
31   parenthesizedExpr = '(' ImpactRuleCondition ')' | MethodName
32 ;
33
34 MethodName:
35   ID'()' | 'pc.' ID'()'
36 ;

```

Listing 5.4: Teil der Grammatik der DSL IRS, über den die Definition der Bedingungen einer Auswirkungsregel erfolgt.

Zu Beginn des Bedingungsblocks können einleitend Gruppen definiert werden (Zeile 3, 8 – 10). Dazu muss ein Kürzel für die Gruppe sowie eine Beschreibung der Gruppe angegeben werden (Zeile 9). Das eingeführte Kürzel kann von Checklistenhinweisen referenziert werden (Zeile 14), um zu kennzeichnen, welcher Gruppe ein Checklistenhinweis zugeordnet wird. Die Gruppenbeschreibung legt indes den Text fest, der für die Gruppe in der Checkliste angezeigt wird. Die Definition einer Gruppe zu Beginn des Bedingungsblocks bietet sich dann an, wenn an mehreren Stellen in der Auswirkungsregel auf die gleiche Gruppe verwiesen werden soll. Die Definition von Gruppen ist jedoch optional.

Darauffolgend wird definiert, welche Auswirkungen eine Modelländerung hat, die spezifische Bedingungen erfüllt. Für jede Auswirkungsregel können mehrere Paare von solchen Bedingungen und zugehöriger Auswirkungsbeschreibung definiert werden (Zeile 4, 12 – 16). Zusätzlich zu der Beschreibung der eigentlichen Auswirkung kann entweder eine zuvor definierte Gruppe referenziert (Zeile 14) oder eine Gruppenbeschreibung angegeben werden (Zeile 15). Wird weder eine zuvor definierte Gruppe referenziert, noch eine Gruppenbeschreibung angegeben, wird der Checklistenhinweis einer Standardgruppe zugeordnet. In der Checkliste werden sämtliche Checklistenhinweise aus der Standardgruppe gemeinsam ohne einleitende Gruppenbeschreibung angezeigt. Die Auswirkungen der Nutzung von Gruppen zeigen sich in der Checkliste in Listing 5.2 auf Seite 60. Dort existiert für die Auswirkungsregel ORM-AR beispielsweise eine Gruppe für Änderungen der Klasse `core.Book`. Diese Gruppe umfasst zwei Checklistenhinweise (Listing 5.2, Zeile 27 – 32).

In jeder Bedingungsspezifikation können mehrere Bedingungen kombiniert werden (Zeile 18 – 36), so wie auch boolesche Ausdrücke in Java über logische Operatoren kombiniert werden können. Zu diesem Zweck unterstützt IRS die logischen Operatoren `&&` und `||` (Zeile 23). Zusätzlich bietet IRS einen Negationsoperator `!` an (Zeile 27) und erlaubt die Klammerung von Bedingungen (Zeile 31). Die Bedeutung dieser Notationen ist die gleiche wie in Java. Der Bedingungsanteil kann auf zwei Arten von Bedingungen verweisen: vordefinierte Bedingungen (Zeile 35, linke Alternative) sowie benutzerdefinierte Bedingungen (Zeile 35, rechte Alternative). Die genaue Bedeutung dieser beiden Bedingungstypen wird im Anschluss an diesen Absatz genauer dargelegt.

Der Nutzer der DSL kann den Auswirkungsblock alternativ auch leer lassen (Listing 5.3, Zeile 11). In diesem Fall müssen die Bedingungen samt zugehöriger Auswirkungsbeschreibung vom Benutzer selbst in Java implementiert werden. Überdies kann der Modellierer angeben, dass die Modellierung der Auswirkungsregel unvollständig ist (Listing 5.4, Zeile 2). In diesem Fall muss ein Entwickler handgeschrieben weitere Bedingungsprüfungen und Checklistenhinweise implementieren. Dieser Sachverhalt wird genauer in Abschnitt 5.4.2 beschrieben.

Verwendung vordefinierter Bedingungen

Vordefinierte Bedingungen können direkt innerhalb jeder Auswirkungsregel angegeben werden, ohne dass die Nutzer von IRS selbst festlegen müssen, wie die Bedingungen implementiert sind. Derartige Bedingungen wurden bereitgestellt, um die Spezifikation von Bedingungen in Auswirkungsregeln zu vereinfachen.

In dieser Arbeit wurden für die wichtigsten Arten von Modelländerungen, die im Rahmen der Modelldifferenzberechnung ermittelt werden können, vordefinierte Bedingungen implementiert.

Diese prüfen jeweils, ob die zugehörige Modelländerung aufgetreten ist. Da Authoring als Eingabemodell Klassendiagramme verarbeitet, werden nachfolgend einige ausgewählte vordefinierte Bedingungen aufgeführt, die im Kontext von Klassendiagrammen genutzt werden können. Eine vollständige Auflistung der vordefinierten Bedingungen ist in Anhang D dargelegt.

- Prüfung auf Modelländerungen bezüglich von Modellelementen allgemein (Pakete, Klassen, Assoziationen oder Attributen):
 - `addedModelElement()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen eines Modellelements repräsentiert.
 - `deletedModelElement()`: Liefert `true`, wenn die Modelländerung das Löschen eines Modellelements darstellt.
- Prüfung auf Modelländerungen bezüglich von Paketen:
 - `addedPackage()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen eines Pakets repräsentiert.
 - `deletedPackage()`: Gibt `true` zurück, falls die Modelländerung das Löschen eines Pakets darstellt.
 - `renamedPackage()`: Liefert `true`, falls die Modelländerung das Umbenennen eines Pakets repräsentiert.
 - `movedPackage()`: Gibt `true` zurück, falls die Modelländerung das Verschieben eines Pakets darstellt.
- Prüfung auf Modelländerungen bezüglich von Klassen:
 - `addedClass()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen einer Klasse repräsentiert.
 - `deletedClass()`: Gibt `true` zurück, falls die Modelländerung das Löschen einer Klasse darstellt.
 - `renamedClass()`: Liefert `true`, falls die Modelländerung das Umbenennen einer Klasse repräsentiert.
 - `movedClass()`: Gibt `true` zurück, falls die Modelländerung das Verschieben einer Klasse darstellt.
- Prüfung auf Modelländerungen bezüglich von Assoziationen:
 - `addedAssociation()`: Gibt `true` zurück, insofern die Modelländerung das Hinzufügen einer Assoziation repräsentiert.
 - `deletedAssociation()`: Liefert `true`, falls die Modelländerung das Löschen einer Assoziation repräsentiert.
 - `renamedTgtAssociationEnd()`: Gibt `true` zurück, falls die Modelländerung das Umbenennen des Rollenamens des Zilassoziationsendes der Assoziation darstellt.

- `changedTgtAssociationEndCardinalityFromOneToStar()`: Liefert `true`, wenn die Modelländerung die Veränderung der Kardinalität des Zielassoziationsendes von einer einwertigen zu einer mehrwertigen Assoziation repräsentiert.
- `changedTgtAssociationEndCardinalityFromStarToOne()`: Liefert `true`, wenn die Modelländerung die Veränderung der Kardinalität des Zielassoziationsendes von einer mehrwertigen zu einer einwertigen Assoziation ausdrückt.

Für die Veränderung des Quellassoziationsendes einer Assoziation werden hierbei bewusst keine vordefinierten Bedingungen angeboten. Der Grund dafür ist, dass sämtliche Assoziationen in ECore implizit gerichtet sind und unmittelbar keine Informationen zu einem Quellassoziationsende spezifiziert werden können. Folglich können sich auch keine Modelländerungen eines Quellassoziationsendes ergeben.

- Prüfung auf Modelländerungen bezüglich von Attributen:
 - `addedAttribute()`: Gibt `true` zurück, insofern die Modelländerung das Hinzufügen eines Attributs repräsentiert.
 - `deletedAttribute()`: Liefert `true`, wenn die Modelländerung das Löschen eines Attributs repräsentiert.
 - `renamedAttribute()`: Gibt `true` zurück, insofern die Modelländerung das Umbenennen eines Attributs repräsentiert.
 - `movedAttribute()`: Liefert `true`, falls die Modelländerung die Verschiebung eines Attributs darstellt.
 - `changedAttributeCardinality()`: Gibt `true` zurück, falls die Modelländerung das Ändern der Kardinalität eines Attributs ausdrückt.

Hervorzuheben ist hierbei, dass die Bedingung `addedModelElement()` genau dann erfüllt ist, wenn die Bedingung `addedPackage()`, `addedClass()`, `addedAssociation()` oder `addedAttribute()` erfüllt ist. Die Nutzung dieser Bedingung bietet sich an, wenn allgemein geprüft werden soll, ob eines dieser Modellelemente hinzugefügt wurde. Analog ist die Bedingung `deletedModelElement()` genau dann erfüllt, wenn die Bedingung `deletedPackage()`, `deletedClass()`, `deletedAssociation()` oder `deletedAttribute()` erfüllt ist.

Die genannten vordefinierten Bedingungen können in allen Softwareentwicklungsprojekten eingesetzt werden, die Klassendiagramme zur Modellierung verwenden. Darüber hinaus wurden in dieser Arbeit projektspezifische vordefinierte Bedingungen eingeführt. In diesen ist Wissen über Spezifika des Projekts, in dessen Kontext diese Arbeit durchgeführt wurde, gekapselt. Nachfolgend sind sämtliche vordefinierte Bedingungen aufgelistet, die projektspezifisch abgeleitet wurden:

- `concernsPersistentModelElement()`: Liefert `true`, falls die Modelländerung ein persistentes Modellelement betrifft. Persistente Modellelemente werden über verschiedenartige Stereotypen gekennzeichnet und können daher über diese Stereotypen identifiziert werden.

- `concernsPOClass()`: Gibt `true` zurück, wenn die Modelländerung eine Klasse betrifft, die einen Stereotyp `PO` besitzt. Dieser Stereotyp kennzeichnet Klassen, die eine besondere Bedeutung in Authoring haben.
- `concernsNonPOClass()`: Liefert `true`, falls die Modelländerung eine Klasse betrifft, die einen Stereotyp `NonPO` besitzt. Dieser Stereotyp kennzeichnet Klassen, die eine besondere Bedeutung in Authoring haben. Neben Klassen mit den Stereotypen `PO` sowie `NonPO` gibt es Klassen mit weiteren Stereotypen, so dass aus dem Wahrheitswert dieser Bedingung nicht auf den Wahrheitswert der Bedingung `concernsPOClass()` geschlossen werden kann.

Verwendung benutzerdefinierter Bedingungen

Es wurden nur solche Bedingungen vordefiniert, die für alle Auswirkungsregeln hilfreich sein können. Da die vordefinierten Bedingungen aber nicht zwangsläufig ausreichen, um die erforderlichen Bedingungen für Auswirkungsregeln zu spezifizieren, wird zusätzlich die Möglichkeit geboten benutzerdefinierte Bedingungen im Bedingungsanteil einzuführen. Über diese hat der Entwickler die Möglichkeit auszudrücken, dass eine weiterführende Bedingungsprüfung erforderlich ist. Die Implementierung einer solchen Bedingungsprüfung muss ein Entwickler manuell hinzufügen. Um zwischen vordefinierten und benutzerdefinierten Bedingungen unterscheiden zu können, werden Erstere durch das Schlüsselwort `pc` (engl. predefined condition) gefolgt von einem Punkt eingeleitet (Listing 5.4 auf Seite 65, Zeile 35, rechte Alternative).

Verwendung von Platzhaltern in Checklistenhinweisen

Um die Spezifikation von Auswirkungsregeln flexibler zu gestalten, ist es in IRS möglich Platzhalter in einen Checklistenhinweis oder eine Gruppenbeschreibung einzubauen. Ein Platzhalter ist ein speziell gekennzeichnete Bereich in einer Zeichenkette. Zur Laufzeit der Checklisten-generierung wird der Platzhalter durch eine andere Zeichenkette ersetzt. Ein Entwickler muss dazu Code bereitstellen, der die Zeichenkette zurückgibt, die den Platzhalter ersetzt. Dies kann dann erforderlich sein, wenn der Checklistenhinweis bzw. die Gruppenbeschreibung nicht komplett statisch vordefiniert werden kann. In der Implementierung können hingegen dynamisch weiterführende Informationen in die jeweilige Zeichenkette eingebaut werden.

Standardmäßig wird der vordefinierte Platzhalter `#name` angeboten. Dieser wird automatisch durch eine Repräsentation des Namens des betroffenen Modellelements ersetzt. Die Entwickler müssen folglich keinen Code für diesen Platzhalter implementieren.

Der Vorteil bei der Verwendung von benutzerdefinierten Bedingungen und Platzhaltern liegt darin, dass abstrakte Bedingungen formuliert und die Auswirkungen von Änderungen abstrakt beschrieben werden können. Dies erhöht die Lesbarkeit einer Auswirkungsregel, da leichter verständlich ist, unter welchen Bedingungen welcher Checklistenhinweis erzeugt wird, ohne dass die Implementierungsdetails dazu bekannt sein müssen.

5.4.2 Generierung und Erweiterung von Auswirkungsregeln

Aus den Spezifikationen der Auswirkungsregeln in IRS werden Java-Implementierungen generiert. Jede generierte Implementierungsklasse ist von der Klasse `AbstractIRBase` abgeleitet. Abbildung 5.5 zeigt das Klassendiagramm zu dieser Basisklasse.

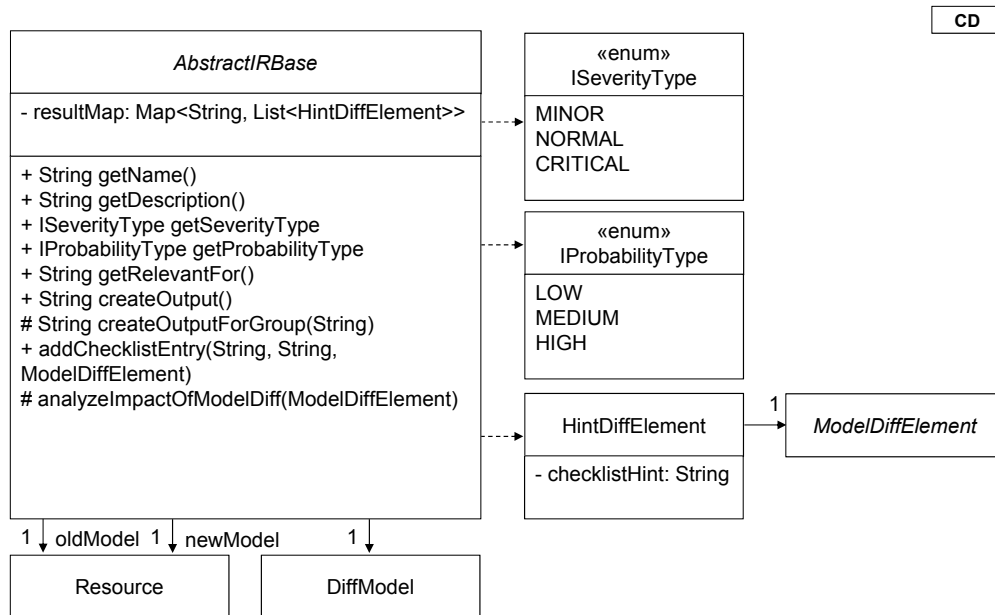


Abbildung 5.5: Überblick über die Basisklasse für alle generierten Implementierungen von Auswirkungsregeln.

Sämtliche Checklistenhinweise, die von einer Auswirkungsregel identifiziert wurden, sind in der Datenstruktur `resultMap` gekapselt. Als Schlüssel in dieser Datenstruktur dient der Name der Gruppe und als Wert ist eine Liste hinterlegt. Jeder Eintrag einer solchen Liste besteht aus dem eigentlichen Checklistenhinweis sowie der zugehörigen Modelländerung. Jede konkrete Unterklasse der abstrakten Basisklasse steuert über die Methode `addChecklistEntry()` neue Einträge zu dieser Datenstruktur bei. Die Bedeutung der Assoziationen zu den Klassen `Resource` und `DiffModel` wird im weiteren Verlauf dieses Abschnitts noch erläutert.

Ferner stellt die Klasse Methoden bereit, über die auf die allgemeinen Informationen zu einer Auswirkungsregel zugegriffen werden kann:

- `getName()` zur Ermittlung des Namens,
- `getDescription()` zur Ermittlung der Beschreibung,
- `getSeverityType()` zur Ermittlung des Schweregrads von Verstößen gegen die Auswirkungsregel,
- `getProbabilityType()` zur Bestimmung der Wahrscheinlichkeit, dass die von der Auswirkungsregel gemeldeten Hinweise tatsächlich für die Entwickler relevant sind und

- `getRelevantFor()` zur Bestimmung der Rollen bzw. Personen, für die von der Auswirkungsregel gemeldete Checklistenhinweise relevant sind.

Überdies liefert die Methode `createOutput()` eine Zeichenkette zurück, welche die Ergebnisse der Auswirkungsregel repräsentiert. Diese Zeichenkette wird in die Checkliste integriert, um die Ergebnisse der Auswirkungsregel gebündelt anzugeben. Zu diesem Zweck greift die Methode unter anderem auf die verschiedenen zuvor genannten `get`-Methoden zu, um die einleitende Beschreibung zu einer Auswirkungsregel zu erzeugen. Darüber hinaus wird die Methode `createOutputForGroup()` verwendet, die eine Zeichenkette zurückliefert, welche die Ergebnisse zu der übergebenen Gruppe kapselt.

Weiterhin enthält die Klasse `AbstractIRBase` die Methode `analyzeImpactOfModelDiff()`, über die geprüft wird, ob eine Modelländerung eine Auswirkung auf das Softwaresystem hat oder nicht. Die konkrete Logik zur Durchführung dieser Prüfungen muss durch die Unterklassen der Klasse `AbstractIRBase` bereitgestellt werden.

Aus einem IRS-Modell wird stets genau eine Klasse für die Auswirkungsregel generiert. Die Generierung dieser Klasse wird nach folgendem Schema vollzogen:

- Der Name der generierten Klasse bildet sich aus dem Präfix `IR`, dem Namen der Auswirkungsregel ohne Leerzeichen und dem Suffix `Base`. Jede generierte Klasse erweitert die abstrakte Basisklasse `AbstractIRBase`.
- In jeder generierten Klasse wird die Methode `analyzeImpactOfModelDiff()` gemäß dem Muster Schablonenmethode (engl. Template Method Pattern) [GHJV95] implementiert. In dieser Methode wird eine `if` Bedingung für jeden definierten Bedingungsteil der Auswirkungsregel generiert. Als Bedingungen werden genau die definierten Bedingungen übernommen, da die Syntax in IRS in diesem Kontext konform ist zu der Spezifikation von Bedingungen in Java. Dieser Teil der Bedingungsprüfungen kann nicht in Unterklassen überschrieben werden. Der letzte Auflistungspunkt dieser Auflistung legt dar, wie Entwickler weitere Bedingungsprüfungen einbringen können.
- Für jede benutzerdefinierte Bedingung mit dem Namen `X` wird eine abstrakte Methode mit dem Namen `X` generiert. Diese Methode gibt einen booleschen Wert zurück und erwartet einen Parameter vom Typ `ModelDiffElement`. Diese Klasse ist die Basisklasse für jede Art von Modelländerung, wie in Abschnitt 4.2.4 erklärt.
- Für jeden Platzhalter mit dem Namen `Y` wird eine abstrakte Methode mit dem Namen `getY` generiert. Diese Methode gibt eine Zeichenkette zurück und erwartet ebenfalls einen Parameter vom Typ `ModelDiffElement`.
- Ist für eine Auswirkungsregel angegeben, dass die Modellierung der Auswirkungsregel unvollständig ist, wird eine abstrakte Methode `analyzeImpactOfModelDiffNonGenerated()` generiert. Diese wird am Ende von `analyzeImpactOfModelDiff()` aufgerufen. Durch die Implementierung dieser Methode wird den Entwicklern die Möglichkeit gegeben weiterführende Bedingungsprüfungen manuell zu integrieren.

Die Erweiterung der generierten Logik erfolgt über die Bereitstellung einer handgeschriebenen Unterklasse zur generierten Basisklasse. Aus dem zuvor dargelegten Generierungsschema kann unmittelbar abgeleitet werden, dass nur in folgenden Fällen eine handgeschriebene Unterklasse implementiert werden muss:

- Die Auswirkungsregel enthält mindestens eine benutzerdefinierte Bedingung.
- Mindestens ein Checklistenhinweis der Auswirkungsregel enthält einen Platzhalter.
- Die Auswirkungsregel ist als unvollständig markiert.

In diesen Fällen enthält die generierte Klasse für die Auswirkungsregel abstrakte Methoden, die in der Unterklasse implementiert werden müssen. Damit zur Laufzeit des Checklistengenerators Instanzen der handgeschriebenen Unterklassen erzeugt werden können, wird per Konvention angenommen, dass der Name der Unterklasse mit dem Präfix `IR` beginnt und darauf der Name der Auswirkungsregel ohne Leerzeichen folgt. In Abschnitt 5.4.3 werden konkrete Beispiele für Auswirkungsregeln gezeigt. Im Zuge dessen wird ebenso dargelegt, wie eine konkrete generierte und eine zugehörige handgeschriebene Erweiterung aussieht.

Bei den bisherigen Erläuterungen wurden ausschließlich die Auswirkungen von einzelnen Modelländerungen betrachtet. Der Ansatz zur Auswirkungsanalyse kann jedoch auch in Situationen eingesetzt werden, in denen die Auswirkungen von mehreren Modelländerungen abhängen. Obgleich IRS keine direkten Mechanismen zur Handhabung solcher Fälle bietet, können derartige Szenarien über die handgeschriebenen Erweiterungen zu den generierten Implementierungen umgesetzt werden. Das ist möglich, da die Basisklasse `AbstractIRBase` Zugriff auf das gesamte Differenzmodell besitzt, wie in Abbildung 5.5 auf Seite 70 über die Assoziation zur Klasse `DiffModel` gekennzeichnet. Daher kann jede Auswirkungsregel bei Bedarf das gesamte Differenzmodell traversieren. Weiterhin besitzt die Klasse `AbstractIRBase` zwei Assoziationen zu der Klasse `Resource`, welche Zugriff auf die ursprüngliche und die geänderte Version des zugehörigen Klassendiagramms bieten. Dies kann hilfreich sein in Fällen, in denen Informationen zu Elementen aus den Klassendiagrammen bezogen werden müssen. Konkret können derartige Anteile in der Methode `analyzeImpactOfModelDiffNonGenerated()` in der handgeschriebenen Erweiterungsklasse implementiert werden. Aufgrund dieser Struktur können selbst komplexe Auswirkungsregeln umgesetzt werden und der Ansatz ist nicht auf simple Auswirkungsregeln beschränkt.

5.4.3 Beispiele für die Modellierung von Auswirkungsregeln

Nachfolgend werden verschiedene Beispiele für Auswirkungsregeln vorgestellt, welche die Evolution von Authoring vereinfachen können. Zu jeder Auswirkungsregel ist eine Motivation für die Einführung der Auswirkungsregel angegeben. Ferner ist jeweils aufgeführt, welche Konsequenzen damit verbunden sind, wenn die Entwickler vergessen, die von der Auswirkungsregel gemeldeten Entwicklungsaktivitäten durchzuführen. Für die Auswirkungsregel ORM-AR wird dieser allgemeine Teil in diesem Abschnitt hingegen nicht dargelegt, da die Auswirkungsregel einleitend bereits in Abschnitt 5.2.1 eingeführt wurde. Abschließend wird zu jeder Auswirkungsregel die Modellierung in IRS verdeutlicht. Die gezeigten Modelle für die Auswirkungsregeln

decken allerdings nicht immer vollumfänglich alle denkbaren Szenarien ab. So wurde in manchen Fällen insbesondere von Spezialfällen abstrahiert, die sehr projektspezifisch sind. Dies gilt vor allem für die Auswirkungsregeln XMLMig-AR sowie DBMig-AR.

Zudem wurden in den Beispielen in keiner Regel die Auswirkungen einer Änderung der Attributkardinalität betrachtet. Dies ist darin begründet, dass der DSA-Generator stets einwertige Attribute generiert. Folglich haben Änderungen der Attributkardinalitäten keinen Einfluss auf die generierten Artefakte, so dass deren Auswirkungen nicht betrachtet werden. Weiterhin werden zur Verkürzung der Beispiele die Auswirkungen von Paketänderungen nicht betrachtet. Dies stellt jedoch keine starke Einschränkung dar, denn die Paketänderungen lassen sich auf Änderungen der darin enthaltenen Klassen übertragen.

Modellierung der Auswirkungsregel zur XML-Migrationsanalyse (XMLMig-AR)

Der lokale Arbeitsbereich eines Benutzers von Authoring kann Instanzen verschiedener Entitäten enthalten, z.B. von Steuergeräten oder deren Fehlercodes. Authoring erlaubt es diese Instanzen nach XML zu exportieren oder XML-Daten, die zuvor exportiert wurden, in den lokalen Arbeitsbereich eines Benutzers zu importieren. In diesem Kontext gilt es zu beachten, dass die Entitätsklassen aus den Klassendiagrammen generiert werden, wie in Kapitel 2 eingeführt. Die Folge davon ist, dass sich Änderungen in den Klassendiagrammen nicht nur auf die Entitätsklassen auswirken, sondern auch auf die zugehörigen exportierten XML-Daten. Wird beispielsweise ein Attribut einer Klasse umbenannt, so spiegelt sich diese Umbenennung auch in den XML-Daten wider, die zu dieser Klasse erzeugt werden. Dadurch können XML-Daten, die in einer Authoring Version exportiert wurden, unter Umständen nicht unmittelbar in ein Authoring in einer anderen Version importiert werden. Verschiedenartige Änderungen in den Klassendiagrammen können somit die Implementierung von XML-Migrationsklassen erfordern. Aus diesem Grund müssen die zu importierenden XML-Daten vorab migriert werden.

Die Auswirkungsregel XMLMig-AR kapselt die Bedingungen, unter denen eine XML-Migrationsklasse implementiert werden muss. Durch die Ausführung dieser Auswirkungsregel werden Entwickler darüber benachrichtigt, wenn XML-Migrationsklassen zu implementieren sind. Ferner schlägt die Auswirkungsregel fertige XML-Migrationsklassen vor, so dass die Entwickler diese nicht manuell implementieren müssen.

Die Modellierung der Auswirkungsregel XMLMig-AR ist in Listing 5.5 angegeben. Falls Entwickler vergessen adäquate XML-Migrationsklassen anzulegen, ist es den Benutzern von Authoring nicht möglich bestimmte Typen von XML-Daten zu importieren. Das kann dazu führen, dass sie Authoring für ihren Anwendungsfall nicht länger einsetzen können. Aus diesem Grund sind die Hinweise auf die zu implementierenden XML-Migrationsklassen kritisch (Zeile 4). Zudem ist die Wahrscheinlichkeit, dass die gemeldeten XML-Migrationsklassen tatsächlich zu implementieren sind, hoch (Zeile 5).

Die Auswirkungsregel drückt aus, dass in mindestens drei Fällen eine XML-Migration durchgeführt werden muss:

- Es wurde ein Attribut oder eine Assoziation gelöscht (Zeile 13) und keine Oberklasse der betroffenen Klasse enthält das gelöschte Attribut bzw. die gelöschte Assoziation (Zeile 11). Ferner muss die betroffene Klasse von einer Klasse ausgehend erreichbar sein, welche

```

1 impactRule "XML Migration Analysis" {
2   description = "This rule checks whether it is necessary to
3     implement XML migration classes."
4   severity = critical
5   probability = high
6
7   impact [incomplete] {
8     group forPOClass "Implement XML migration class for class
9       {MigClass}:";
10
11     !superClassContainsDeletedFeatures() &&
12       concernedClassReachableViaPOClass() &&
13       (pc.deletedAttribute() || pc.deletedAssociation()) =>
14     group forPOClass, hint "Write migration to delete
15       XML element {#name}."
16
17     (pc.renamedAttribute() || pc.renamedTargetAssociationEnd)
18       && concernedClassReachableViaPOClass()
19     => group forPOClass, hint "Write migration to rename
20       XML element {OldName} to {NewName}."
21
22     pc.movedAttribute() && concernedClassReachableViaPOClass()
23     => group forPOClass, hint "Write migration to
24       move XML element {#name} to {NewLocation}."
25   }
26 }

```

Listing 5.5: Modellierung der Auswirkungsregel XMLMig-AR.

mit dem Stereotyp PO markiert ist. Letztgenannter Sachverhalt wird durch die benutzerdefinierte Bedingung `concernedClassReachableViaPOClass()` ausgedrückt (Zeile 12). Aufgrund dieser benutzerdefinierten Bedingung muss eine handgeschriebene Erweiterungsklasse bereitgestellt werden, in der die Methode `concernedClassReachableViaPOClass()` zu implementieren ist. Der resultierende Checklistenhinweis weist darauf hin, dass im Rahmen einer XML-Migration ein XML-Element gelöscht werden muss. Dazu wird auf den vordefinierten Platzhalter `#name` verwiesen, der durch den Namen des betroffenen Modellelements ersetzt wird.

- Es wurde ein Attribut oder ein Zielloassoziationsende einer Assoziation umbenannt (Zeile 17) und die betroffene Klasse erfüllt die benutzerdefinierte Bedingung `concernedClassReachableViaPOClass()` (Zeile 18). Im Checklistenhinweis wird auf den Namen des alten Elements verwiesen, welches umbenannt werden muss. Hierzu werden die Platzhalter `OldName` und `NewName` eingeführt (Zeile 20). Infolgedessen müssen in der handgeschriebenen Erweiterungsklasse zur generierten Implementierung die Methoden `getOldName()` und `getNewName()` implementiert werden.
- Ein Attribut wurde verschoben (Zeile 22) und die betroffene Klasse erfüllt wiederum die benutzerdefinierte Bedingung `concernedClassReachableViaPOClass()` (Zeile

le 22). Der Platzhalter `NewLocation` (Zeile 24) verweist auf das Element, in welches das jeweilige XML-Element verschoben werden muss. Damit der Platzhalter zur Laufzeit der Checklisten-generierung durch die korrekte Zeichenkette ersetzt wird, muss in der handgeschriebenen Erweiterungsklasse die Methode `getNewLocation()` implementiert werden.

Weiterhin drückt die Auswirkungsregel aus, dass die Modellierung unvollständig ist (Zeile 7). Die Folge dieser Angabe ist, dass in die generierte Basisklasse die abstrakte Methode `analyzeImpactOfModelDiffNonGenerated()` generiert wird. Diese muss in der handgeschriebenen Erweiterungsklasse implementiert werden, um die weiteren Bedingungsprüfungen in die Auswirkungsregel integrieren zu können.

Als Nächstes wird die Generierung einer Implementierung für eine Auswirkungsregel am Beispiel der Auswirkungsregel XMLMig-AR illustriert. Im Anschluss daran wird die Bereitstellung einer handgeschriebenen Erweiterung auf Basis dieser generierten Implementierung erläutert.

Generierung der Implementierung zu XMLMig-AR Listing 5.6 zeigt einleitend den Teil der generierten Implementierung, in dem die allgemeinen Informationen zu der Auswirkungsregel über `get`-Methoden zurückgeliefert werden. Die generierte Basisklasse folgt dem zuvor eingeführten Namensschema und ist abstrakt, da die Auswirkungsregel benutzerdefinierte Bedingungen sowie Platzhalter enthält und als unvollständig deklariert wurde. Die Implementierung der `get`-Methoden ergibt sich unmittelbar aus dem in Listing 5.5 gezeigten Modell für die Auswirkungsregel. So wird in der Implementierung der dort spezifizierte Name (Listing 5.6, Zeile 5 – 7), die Beschreibung (Zeile 9 – 12), der Schweregrad (Zeile 14 – 16) und die Wahrscheinlichkeit für das Eintreten der gemeldeten Checklistenhinweise (Zeile 18 – 20) zurückgeliefert.

Im Anschluss an den Teil mit den genannten allgemeinen Informationen folgt der Teil, der aufgrund der definierten Platzhalter und benutzerdefinierten Bedingungen der Auswirkungsregel generiert wurde (Zeile 26 – 39). Für die vier Platzhalter wurde jeweils eine abstrakte `get`-Methode generiert, die eine Zeichenkette zurückliefert und der als Parameter die aktuell betrachtete Modelländerung übergeben wird (Zeile 27 – 30). Die von diesen Methoden zurückgelieferten Zeichenketten ersetzen in den Checklistenhinweisen die entsprechenden Platzhalter. Weiterhin wurden für die zwei benutzerdefinierten Bedingungen abstrakte Methoden generiert, die einen booleschen Wert zurückliefern und so benannt sind wie die benutzerdefinierten Bedingungen (Zeile 33 – 36). Auch diese Methoden erhalten als Parameter die aktuell betrachtete Modelländerung. Ferner wird eine abstrakte Methode `analyzeImpactOfModelDiffNonGenerated()` generiert (Zeile 38 – 39), da die Auswirkungsregel als unvollständig modelliert wurde. Die Verwendung dieser Methode ergibt sich im weiteren Verlauf dieses Abschnitts.

Die Prüfung der modellierten Bedingungen und die Erzeugung der zugehörigen Checklistenhinweise erfolgt in der Methode `analyzeImpactOfModelDiff()`, deren Implementierung in Listing 5.7 auf Seite 77 gezeigt ist.

Innerhalb der Methode `analyzeImpactOfModelDiff()` wird für jeden Eintrag der Auswirkungsregel, bestehend aus Bedingungsspezifikation und Checklistenhinweis, ein Bedingungsblock generiert. In diesen wird die jeweilige Bedingung aus der Auswirkungsregel übernommen. Dabei werden die abstrakten Methoden für die benutzerdefinierten Bedingungen aufgerufen, wenn in einer Bedingungsspezifikation eine benutzerdefinierte Bedingung verwendet wurde

```

1 public abstract class IRXMLMigrationAnalysisBase
2     extends AbstractIRBase {
3
4     //General information concerning impact rule
5     public String getName() {
6         return "XML Migration Analysis";
7     }
8
9     public String getDescription() {
10        return "This rule checks whether it is necessary to " +
11            "implement XML migration classes";
12    }
13
14    public ISeverityType getSeverityType() {
15        return ISeverityType.CRITICAL;
16    }
17
18    public IProbabilityType getProbabilityType() {
19        return IProbabilityType.HIGH;
20    }
21
22    public String getRelevantFor() {
23        return null;
24    }
25
26    //Methods for returning placeholder strings
27    abstract public String getMigClass(ModelDiffElement diffElem);
28    abstract public String getOldName(ModelDiffElement diffElem);
29    abstract public String getNewName(ModelDiffElement diffElem);
30    abstract public String getNewLocation(ModelDiffElement diffElem);
31
32    //Methods for checking userdefined conditions
33    abstract public boolean superClassContainsDeletedFeatures(
34        ModelDiffElement diffElem);
35    abstract public boolean concernedClassReachableViaPOClass(
36        ModelDiffElement diffElem);
37
38    abstract public void analyzeImpactOfModelDiffNonGenerated(
39        ModelDiffElement diffElem);
40 }

```

Listing 5.6: Generierte Implementierung der Auswirkungsregel XMLMig-AR (siehe Listing 5.5 auf Seite 74) mit Methoden zu allgemeinen Informationen, Platzhaltern sowie benutzerdefinierten Bedingungen.

(Zeile 9 – 10, 18, 25). Die Checklistenhinweise werden über die Methode `addChecklistEntry()` zur Checkliste hinzugefügt. An den Stellen der Checklistenhinweise, an denen in der Modellierung Platzhalter angegeben sind, erfolgt der Aufruf der Methoden zur Rückgabe

```

1 public abstract class IRXMLMigrationAnalysisBase
2     extends AbstractIRBase {
3
4     public final void analyzeImpactOfModelDiff(
5         ModelDiffElement diffElem) {
6         String forPOClass = "Concerning entry for class " +
7             getMigClass(diffElem) + ":";
8
9         if (!superClassContainsDeletedFeatures(diffElem) &&
10             concernedClassReachableViaPOClass(diffElem) &&
11             (pc.deletedAttribute(diffElem) ||
12              pc.deletedAssociation(diffElem)) ) {
13             addChecklistEntry(forPOClass, "Write migration to delete " +
14                 "XML element " + getName(diffElem) + ".", diffElem);
15         }
16
17         if (pc.renamedAttribute(diffElem) &&
18             concernedClassReachableViaPOClass(diffElem)) {
19             addChecklistEntry(forPOClass, "Write migration to rename " +
20                 "XML element " + getOldName(diffElem) + " to " +
21                 getNewName(diffElem) + ".", diffElem);
22         }
23
24         if (pc.movedAttribute(diffElem) &&
25             concernedClassReachableViaPOClass(diffElem)) {
26             addChecklistEntry(forPOClass, "Write migration class to " +
27                 "move XML element " + getName(diffElem) + " to " +
28                 getNewLocation(diffElem) + ".", diffElem);
29         }
30
31         analyzeImpactOfModelDiffNonGenerated(diffElem);
32     }
33 }

```

Listing 5.7: Generierte Implementierung der Auswirkungsregel XMLMig-AR (siehe Listing 5.5 auf Seite 74) zur Prüfung der Bedingungen der Auswirkungsregel.

der Zeichenkette für die Platzhalter. Beispielsweise werden im zweiten Bedingungsblock die Methoden `getOldName()` (Zeile 20) und `getNewName()` (Zeile 21) aufgerufen, um diese fehlenden Teile des Checklistenhinweises zu ermitteln. Ferner wird jeweils die betrachtete Modelländerung übergeben, damit in der Checkliste angegeben werden kann, welche Modelländerung zu der Erzeugung des Checklistenhinweises geführt hat.

Zum Abschluss der Methode `analyzeImpactOfModelDiff()` wird die Methode `analyzeImpactOfModelDiffNonGenerated()` aufgerufen. Dieser Aufruf wird in die Methode generiert, da die Auswirkungsregel als unvollständig modelliert wurde.

Implementierung einer handgeschriebenen Erweiterungsklasse zu XMLMig-AR

In der handgeschriebenen Erweiterungsklasse `IRXMLMigrationAnalysis` müssen die vier

abstrakten Methoden zur Handhabung der Platzhalter sowie die zwei abstrakten Methoden zur Implementierung der benutzerdefinierten Bedingungen implementiert werden. Weiterhin muss die Methode `analyzeImpactOfModelDiffNonGenerated()` implementiert werden, da die Auswirkungsregel als unvollständig modelliert wurde. In diese letztgenannte Methode können Entwickler weitere Bedingungsprüfungen hinzufügen.

Eine spezielle Anforderung an die Auswirkungsregel XMLMig-AR war, dass in der Checkliste am Ende jeder identifizierten Gruppe ein Vorschlag für eine XML-Migrationsklasse gezeigt werden soll. Diese Migrationsklasse soll sämtliche Modelländerungen betreffen, die den Checklistenhinweisen der Gruppe zugeordnet sind, und passend dazu die XML-Migration verdeutlichen. Die Umsetzung einer derartigen Anforderung kann darüber erreicht werden, dass in der handgeschriebenen Erweiterungsklasse die Methode `createOutputForGroup()` überschrieben wird. Diese ist, wie in Abschnitt 5.4.2 erklärt, dafür zuständig eine Zeichenkette zurückzuliefern, welche die Ergebnisse einer Gruppe repräsentiert. Die Implementierung der Methode ist in Listing 5.8 dargestellt. Wie dort zu sehen, werden als Erstes die Gruppenbeschreibung (Zeile 6) und anschließend die einzelnen ermittelten Checklistenhinweise zur Gruppe einbezogen (Zeile 8 – 12). Anschließend wird auf Basis der Modelländerungen, die zu den Checklistenhinweisen der Gruppe hinterlegt sind, die XML-Migrationsklasse ermittelt.

```

1 public class IRXMLMigrationAnalysis extends
2     IRXMLMigrationAnalysisBase {
3
4     public String createOutputForGroup(String groupName) {
5         StringBuilder groupBuilder = new StringBuilder();
6         groupBuilder.append("- " + groupName + "\n");
7         List<HintDiffElement> hintList = resultMap.get(groupName);
8         for (HintDiffElement hint : hintList) {
9             groupBuilder.append("    - " + hint.getChecklistHint()
10                 + "(Reason: " + ModelDiffHelper.getDescriptionOfMD(
11                     hint.getModelDiffElement() + ")");
12         }
13         groupBuilder.append("\nProposed XML migration class:\n");
14         //construction of proposed XML migration class
15         return groupBuilder.toString();
16     }
17 }

```

Listing 5.8: Implementierung der Methode `createOutputForGroup()`.

Eine Alternative zu dieser Vorgehensweise besteht darin, in die Checklistenhinweise einen Platzhalter einzufügen, der durch den Quellcode einer Migrationsklasse ersetzt wird. Diese Herangehensweise hat jedoch den Nachteil, dass dann eine Migrationsklasse pro Checklistenhinweis vorgeschlagen wird. Das hätte zur Folge, dass die Entwickler auf Basis der einzelnen Migrationsklassen selbstständig eine Migrationsklasse für alle Änderungen zur Gruppe erzeugen müssten. Zur Erhöhung der Lesbarkeit der Checklisten und zur Reduktion des Arbeitsaufwands für die Entwickler wurde von dieser Alternative abgesehen.

Modellierung der Auswirkungsregel zur Analyse von objektrelationalen Mapping-Dateien (ORM-AR)

Die Motivation für die Auswirkungsregel ORM-AR wurde bereits in Abschnitt 5.2.1 vorgestellt. Aus diesem Grund liegt der Fokus in diesem Abschnitt auf der Modellierung von ORM-AR in IRS. Das Modell zu ORM-AR ist in Listing 5.9 illustriert. Fehlerhafte Einträge in den Mapping-Dateien sind demnach kritisch (Zeile 6) und die Wahrscheinlichkeit, dass die gemeldeten Änderungen tatsächlich durchgeführt werden müssen, ist hoch (Zeile 7).

Die Auswirkungsregel drückt aus, dass die Mapping-Dateien in folgenden Fällen geändert werden müssen:

- Falls ein persistentes Modellelement hinzugefügt wird, muss ein neuer Eintrag zur Mapping-Datei hinzugefügt werden (Zeile 12 – 14).
- Insofern ein persistentes Modellelement gelöscht wird, muss der bestehende Eintrag gelöscht werden (Zeile 16 – 18).
- Wenn eine persistente Klasse oder ein persistentes Attribut verschoben wird, muss die Mapping-Datei ebenfalls aktualisiert werden (Zeile 20 – 23). Zudem drückt die Auswirkungsregel aus, dass der bestehende Eintrag in einen bestimmten Eintrag verschoben werden kann. Für die Pfade zum alten und neuen Eintrag wurden die Platzhalter `OldLocation` (Zeile 22) und `NewLocation` eingefügt (Zeile 23).
- Falls die Quellklasse einer persistenten Assoziation verändert wird, muss der bestehende Eintrag bezüglich der alten Quellklasse entfernt und in die neue Quellklasse eingetragen werden (Zeile 25 – 29). Die alte und neue Quellklasse werden über die Platzhalter `OldSourceClass` (Zeile 28) und `NewSourceClass` (Zeile 29) eingefügt.
- Falls die Zielklasse einer persistenten Assoziation verändert wird, muss der bestehende Eintrag bezüglich der alten Zielklasse entfernt und ein neuer Eintrag für die neue Zielklasse eingetragen werden (Zeile 31 – 34).
- Ein bestehender Eintrag muss umbenannt werden, wenn ein persistentes Modellelement geändert wird und einer der folgenden Fälle vorliegt (Zeile 36 – 42):
 - eine Umbenennung einer Klasse, eines Attributs oder des Ziellozationsendes einer Assoziation oder
 - eine Veränderung der Kardinalität des Ziellozationsendes von einer einwertigen zu einer mehrwertigen Assoziation oder umgekehrt
- Falls sich die Informationen bezüglich der Oberklasse einer persistenten Klasse ändern, d.h. wenn eine Oberklasse hinzugefügt, entfernt oder geändert wurde (Zeile 44 – 47), muss der bestehende Eintrag angepasst werden.

IRS

```

1 impactRule "ORM File Analysis" {
2   description = "This rule checks whether it is necessary to
3     change an object relational mapping file, by adding
4     entries, changing existing entries or deleting
5     existing entries."
6   severity = critical
7   probability = high
8
9   impact [incomplete] {
10    group ForClass "Entries concerning class {Class}:"
11
12    pc.concernsPersistentModelElement() &&
13      pc.addedModelElement() => group ForClass,
14      hint "Add new entry to mapping file."
15
16    pc.concernsPersistentModelElement() &&
17      pc.deletedModelElement() => group ForClass,
18      hint "Delete entry from mapping file."
19
20    pc.concernsPersistentModelElement() &&
21      (pc.movedClass() || pc.movedAttribute()) =>
22      group ForClass, hint "Move entry {OldLocation} to
23      new location {NewLocation} in mapping file."
24
25    pc.concernsPersistentModelElement() &&
26      pc.changedSourceClassOfAssociation() => group ForClass,
27      hint "Delete old entry from mapping file from old class
28      {OldSourceClass}. Add new entry to mapping file to class
29      {NewSourceClass}."
30
31    pc.concernsPersistentModelElement() &&
32      pc.changedTgtClassOfAssociation() => group ForClass,
33      hint "Delete old entry from mapping file. Add new entry to
34      mapping file for new target class."
35
36    pc.concernsPersistentModelElement() &&
37      (pc.renamedClass() || pc.renamedAttribute() ||
38      pc.renamedTargetAssociationEnd() ||
39      pc.changedTgtAssociationEndCardinalityFromOneToStar() ||
40      pc.changedTgtAssociationEndCardinalityFromStarToOne())
41    => group ForClass, hint "Rename old entry {OldEntry}
42    to {NewEntry} in mapping file."
43
44    pc.concernsPersistentModelElement() && (pc.deletedSuperType()
45    pc.changedSuperTypeToNewSuperType() ||
46    pc.addedSuperType() || => group ForClass, hint "Adapt
47    mapping file information concerning super type of class."
48  }
49 }

```

Listing 5.9: Modellierung der Auswirkungsregel ORM-AR.

Modellierung der Auswirkungsregel zur Datenmigrationsanalyse in Datenbanken (DBMig-AR)

Auf Basis des alten und des neuen Datenbankschemas wird für Authoring werkzeugunterstützt ein Skript erzeugt, welches die Migration des Datenbankschemas durchführt. Dieses Skript muss jedoch manuell um Anweisungen zur Durchführung einer Datenmigration erweitert werden, da die Migration der Daten in diesem Kontext nicht beachtet wird.

In der Auswirkungsregel DBMig-AR sind sämtliche Bedingungen spezifiziert, die eine Datenmigration erforderlich machen. Insbesondere weist die Auswirkungsregel darauf hin, welche Arten von Datenmigrationen durchzuführen sind.

Listing 5.10 zeigt die Modellierung der Auswirkungsregel DBMig-AR. Eine fehlende oder fehlerhafte Datenmigration kann dazu führen, dass Daten verloren gehen. Die Folge davon ist, dass Benutzer von Authoring nicht mehr mit den zuvor bekannten Daten arbeiten können. Fehler in diesem Gebiet haben also schwerwiegende Auswirkungen für die Benutzer von Authoring. Aus diesem Grund werden die Hinweise dieser Auswirkungsregel als kritisch eingestuft (Zeile 4). Die Wahrscheinlichkeit, dass die gemeldeten Datenmigrationen tatsächlich erforderlich sind, ist zudem hoch (Zeile 5).

Konkret kapselt die Auswirkungsregel, dass in folgenden Fällen Datenmigrationen durchzuführen sind:

- Es wird eine persistente Klasse umbenannt. In diesem Fall muss die durch die Klasse repräsentierte Tabelle umbenannt werden (Zeile 10 – 13). Dies gilt allerdings nur dann, wenn die Klasse keine Oberklasse besitzt. Andernfalls existiert zu der geänderten Klasse keine eigene Tabelle, sondern nur eine Tabelle für die Oberklasse.
- Ein persistentes Attribut oder ein persistentes Ziellosoziationsende wird umbenannt. Hierbei muss die Spalte, die das Attribut oder das Ziellosoziationsende repräsentiert, umbenannt werden (Zeile 15 – 18).
- Ein persistentes Attribut wird verschoben. In diesem Fall muss eine Datenmigration geschrieben werden, durch die der Inhalt der Spalte in die neue Zieltabelle verschoben wird (Zeile 20 – 25). Dies gilt allerdings nur dann, wenn das Attribut nicht innerhalb einer Vererbungshierarchie verschoben wird (Zeile 21), da diese Vererbungshierarchie durch die gleiche Tabelle ausgedrückt wird. Weiterhin muss die neue Zielklasse des Attributs von der alten Quellklasse aus erreichbar sein (Zeile 22), da andernfalls der verschobene Wert nicht ordnungsgemäß verschoben werden kann.
- Bei einer persistenten Assoziation wird das Ziellosoziationsende von einer einwertigen zu einer mehrwertigen Assoziation verändert. In einer Datenmigration kann auf Basis der bestehenden Daten die neue Spalte in der Tabelle für die Zielklasse der Assoziation befüllt werden (Zeile 27 – 31).
- Bei einer persistenten Assoziation wird das Ziellosoziationsende von einer mehrwertigen zu einer einwertigen Assoziation verändert. Dann kann in einer Datenmigration, analog zum vorherigen Fall, auf Basis der bestehenden Daten die neue Spalte in der Tabelle für die Quellklasse der Assoziation befüllt werden (Zeile 33 – 37).

```

1 impactRule "Database Data Migration Analysis" {
2   description = "This rule checks whether it is necessary to
3     implement data migration scripts for the database."
4   severity = critical
5   probability = high
6
7   impact [incomplete] {
8     group MigTable "Migrations concerning table '{Table}':"
9
10    pc.concernsPersistentModelElement() && pc.renamedClass() &&
11      classDoesNotHaveSuperclass()
12    => group MigTable, hint "Write data migration to rename
13      table '{#name}' to '{NewName}'."
14
15    pc.concernsPersistentModelElement() &&
16      (pc.renamedAttribute() || pc.renamedTgtAssociationEnd())
17    => group MigTable, hint "Write data migration to rename column
18      '{#name}' to '{NewName}'."
19
20    pc.movedPersistentAttribute() &&
21      !movedAttributeToSuperTypeOrSubType() &&
22      movedAttributeToClassReachableViaOldClass()
23    => group MigTable, hint "Write data migration to move column
24      content '{#name}' from table '{OldTableBeforeMove}' to table
25      '{NewTableAfterMove}'."
26
27    pc.concernsPersistentModelElement() &&
28      pc.changedTgtAssociationEndCardinalityFromOneToStar()
29    => group MigTable, hint "Write data migration to fill column
30      which was added/will be added due to to-star association in
31      table for target class."
32
33    pc.concernsPersistentModelElement() &&
34      pc.changedTgtAssociationEndCardinalityFromStarToOne()
35    => group MigTable, hint "Write data migration to fill column
36      which was added/will be added due to to-one association in
37      table for source class."
38   }
39 }

```

IRS

...

Listing 5.10: Modellierung der Auswirkungsregel DBMig-AR.

Modellierung der Auswirkungsregel zur Property-Dateien Analyse (Prop-AR)

Für bestimmte Typen von Modellelementen in den Klassendiagrammen erzeugt der DSA-Generator Code, der auf Schlüssel in Property-Dateien verweist. Entwickler müssen zu diesen Schlüsseln Beschreibungen hinzufügen, die später in Authoring angezeigt werden.

Die Auswirkungsregel Prop-AR gibt Aufschluss darüber, unter welchen Bedingungen die Property-Dateien angepasst werden müssen. Dazu legt die Auswirkungsregel dar, welche Einträge zu Property-Dateien hinzugefügt oder daraus gelöscht bzw. welche bestehenden Einträge angepasst werden müssen.

Das Modell zur Auswirkungsregel Prop-AR ist in mehrere Listings aufgeteilt. Listing 5.11 zeigt den Teil der Auswirkungsregel mit den allgemeinen Informationen. Weiterhin sind dort die Auswirkungen der Änderungen, die Klassen betreffen, definiert. Fehlende Einträge in den betroffenen Property-Dateien führen nicht zwangsläufig zu Fehlern, können aber darin resultieren, dass unpassende Zeichenketten in Authoring angezeigt werden. Dies kann die Benutzbarkeit von Authoring beeinträchtigen. Dessen ungeachtet werden die Hinweise als unkritisch eingestuft (Zeile 5). Ferner ist die Wahrscheinlichkeit, dass die Änderungen an den Property-Dateien tatsächlich vorgenommen werden müssen, hoch (Zeile 6).

```

1 impactRule "Property File Analysis" {
2   description = "This rule checks in which way property files
3     have to be changed. This can comprise adding new entries,
4     changing existing entries or deleting existing entries."
5   severity = minor
6   probability = high
7
8   impact {
9     group PFile "Changes in properties file {File}:"
10
11     //Concerning class changes:
12     pc.addedClass() => group PFile, hint "Add entity.typeName.
13       singular.{NewName} and entity.typeName.plural.{NewName}."
14
15     pc.deletedClass() => group PFile, hint "Remove entity.typeName.
16       singular.{NewName} and entity.typeName.plural.{NewName}."
17
18     pc.renamedClass() => group PFile, hint "Add entity.typeName.
19       singular.{NewName} and entity.typeName.plural.{NewName}.
20       Check whether entries entity.typeName.singular.{OldName}
21       and entity.typeName.singular.{OldName} exist. You might take
22       the associated values for the new entries."
23
24     //further conditions shown in next listings
25   }
26 }

```

Listing 5.11: Modellierung der allgemeinen Informationen zur Auswirkungsregel Prop-AR und Definition der Auswirkungen von Änderungen, die Klassen betreffen.

Gemäß der Auswirkungsregel muss eine Property-Datei angepasst werden, wenn eine Klasse hinzugefügt (Zeile 12 – 13), gelöscht (Zeile 15 – 16) oder umbenannt (Zeile 18 – 22) wird. In letztgenanntem Fall weist der Checklistenhinweis darauf hin, dass als Wert zu den hinzuzufügenden Schlüsseln gegebenenfalls die Werte zu den Schlüsseln mit dem alten Namen übernommen werden können.

Ähnlich verhält es sich für Änderungen von Attributen, wie in Listing 5.12 gezeigt. Wenn ein Attribut hinzugefügt (Zeile 1 – 2) oder gelöscht (Zeile 4 – 5) wird, müssen Schlüssel zu einer Property-Datei hinzugefügt oder gelöscht werden. Wird ein Attribut verschoben, so muss unter Umständen ein neuer Eintrag hinzugefügt werden oder es kann ein bestehender Eintrag wiederverwendet werden (Zeile 7 – 10). Im Falle einer Umbenennung eines Attributs kann gegebenenfalls ebenfalls ein bestehender Eintrag umbenannt werden (Zeile 12 – 15).

```

1 pc.addedAttribute() => group PFile, hint "Add
2   field.{FieldNameAttribute}."
3
4 pc.deletedAttribute() => group PFile,
5   hint "Remove field.{FieldNameAttribute}."
6
7 pc.movedAttribute() => group PFile, hint "Add field.
8   {OldFieldNameAttribute}. Check whether entry field.
9   {NewFieldNameAttribute} already exists. You might take
10  the associated value for the new entry."
11
12 pc.renamedAttribute()
13   => group PFile, hint "Add field.{FieldNameAttribute}. Check
14   whether entry field.{OldFieldNameAttribute} already exists.
15   You might take the associated value for the new entry."

```

Listing 5.12: Modellierung der Auswirkungsregel Prop-AR (siehe Listing 5.11). Definition der Auswirkungen von Änderungen, die Attribute betreffen.

Auf vergleichbare Weise wird mit Änderungen von Assoziationen verfahren, wie in Listing 5.13 verdeutlicht. So zieht unter anderem das Hinzufügen oder Löschen einer Assoziation Änderungen einer Property-Datei nach sich (Zeile 1 – 2, 4 – 5). Ferner müssen Property-Dateien angepasst werden, wenn die Quell- oder Zielklasse einer Assoziation verändert wird (Zeile 7 – 10). Zudem genügt unter Umständen eine Umbenennung eines bestehenden Eintrags, wenn ein Zielloziationssende umbenannt wird oder sich die Kardinalität eines Zielloziationssendes verändert (Zeile 12 – 16).

Modellierung der Auswirkungsregel zur Business Rule Analyse (BR-AR)

In Authoring können verschiedene Objekte erzeugt, mit anderen Objekten verbunden und in der Datenbank gespeichert werden. Ohne Prüfung der Objekthinhalte kann es dazu kommen, dass ungültige Objekte angelegt oder Objekte auf ungültige Weise geändert werden. Um dies zu verhindern, werden in Authoring Business Rules eingesetzt, um die Gültigkeit von Objekten zu prüfen und auf Fehler hinzuweisen.

```

1 pc.addedAssociation() => group PFile, hint "Add entry field.
2   {FieldNameAssociation}."
3
4 pc.deletedAssociation() => group PFile, hint "Remove entry
5   field.{FieldNameAssociation}."
6
7 pc.changedSourceClassOfAssociation() ||
8   pc.changedTgtClassOfAssociation() => group PFile,
9   hint "Remove entry field.{OldFieldNameAssociation}.
10  Add entry field.{NewFieldNameAssociation}."
11
12 pc.renamedTargetAssociationEnd() ||
13   pc.changedTgtAssociationEndCardinality() => group PFile,
14   hint "Add entry field.{FieldNameAssociation}. Check whether
15   entry field.{OldFieldNameAssociation} already exists. You
16   might take the associated value for the new entry."

```

Listing 5.13: Modellierung der Auswirkungsregel Prop-AR (siehe Listing 5.11). Definition der Auswirkungen von Änderungen, die Assoziationen betreffen.

Die Auswirkungsregel BR-AR kapselt Bedingungen, die dazu führen können, dass eine Business Rule hinzugefügt werden muss.

Das Modell für die Auswirkungsregel BR-AR ist in Listing 5.14 dargestellt. Die Hinweise dieser Auswirkungsregel sind mit Unsicherheit behaftet, da nicht direkt absehbar ist, ob eine Business Rule tatsächlich erforderlich ist oder nicht. Dies muss stattdessen von den verantwortlichen Entwicklern (Zeile 5) geklärt werden. Aus diesem Grund ist lediglich eine geringe Wahrscheinlichkeit für die Gültigkeit der Hinweise vermerkt (Zeile 4). Die Auswirkungsregel schlägt in fünf Szenarien vor, dass eine Business Rule hinzugefügt werden kann:

- Es wurde eine neue Klasse hinzugefügt und diese Klasse besitzt den Stereotyp PO (Zeile 8 – 10). In diesem Fall kann eine Business Rule hilfreich sein, welche die Eindeutigkeit des Namens der Instanzen dieser neuen Klasse sicherstellt.
- Es wurde ein Attribut hinzugefügt, welches nicht aus einer Oberklasse entfernt wurde (Zeile 12 – 14). In diesem Fall kann eine neue Business Rule sicherstellen, dass nur spezifische Werte für dieses Attribut hinterlegt sein dürfen.
- Falls ein Attribut verschoben wurde, kann es erforderlich sein eine bestehende Business Rule zu der ursprünglichen Quellklasse des Attributs anzupassen und etwaige Prüfungen auf zulässige Werte in einer Business Rule für die Zielklasse zu verankern (Zeile 16 – 18).
- Es wurde eine reflexive Assoziation hinzugefügt (Zeile 20 – 25). In einem solchen Szenario kann eine Business Rule prüfen, ob Zyklen in den Objektstrukturen zulässig sind oder nicht. Der Platzhalter `ClassContainingAssociation` (Zeile 23) wird dazu verwendet, um auf die Klasse hinzuweisen, zu der die reflexive Assoziation hinzugefügt wurde.

```

1 impactRule "Business Rule Analysis" {
2   description = "This rule checks whether it might be
3     required to add new or adapt existing BR's."
4   probability = low
5   relevantFor = "ab"
6
7   impact [incomplete] {
8     pc.addedClass() && pc.concernsPOClass() => "Check whether BR
9       should be added which checks uniqueness of display name for
10      new instances of new class."
11
12     pc.addedAttribute() && !removedSameAttributeFromSuperClass() =>
13       "Check whether BR should be added which checks whether values
14       of added attribute should be restricted."
15
16     pc.movedAttribute() => "Check whether BR exists concerning
17       moved attribute. In this case, the BR needs to be adapted
18       to new class of the attribute."
19
20     addedReflexiveAssociation() => "Due to the addition of a
21       reflexive association, cycles could occur in the resulting
22       object structures, i.e., an instance of type
23       {ClassContainingAssociation} could refer to the same instance
24       via the reflexive association. Check whether a BR should be
25       added which forbids such cycles."
26
27     addedAssociationSoThatClassIsReachableViaMultiplePaths() =>
28       "Due to the addition of an association, the class
29       {TargetClass} is reachable via the paths {PathsDescription}.
30       Check whether new BR should be added ensuring consistency of
31       class that is reachable via multiple paths."
32   }
33 }

```

IRS

Listing 5.14: Modellierung der Auswirkungsregel BR-AR.

- Es wurde eine Assoziation hinzugefügt, so dass von einer Klasse ausgehend eine andere Klasse über mehrere verschiedene Pfade erreichbar ist (Zeile 27 – 31). Dieser Fall wird in Abbildung 5.16 verdeutlicht. In der ursprünglichen Version des Modells (linke Seite in Abbildung 5.16) ist die Klasse E von der Klasse A ausgehend lediglich über einen Pfad erreichbar. In der neuen Version des Modells (rechte Seite in Abbildung 5.16) ist die Klasse E von A ausgehend über zwei Pfade erreichbar. In diesem speziellen Szenario ist es denkbar, dass Instanzen der Klasse E, die über einen Pfad mit einer Instanz der Klasse A verbunden sind auch über den anderen Pfad mit dieser Instanz verbunden sein müssen. Derartige Business Rules existieren bereits für Spezialfälle. Der Platzhalter `TargetClass` (Zeile 29) soll auf die Klasse verweisen, die über mehrere Pfade erreichbar ist, in vorherigem Beispiel ist das die Klasse E. Weiterhin soll der Platzhalter `PathsDescription` (Zeile 29) die bisherigen Pfade zu der Zielklasse auflisten.

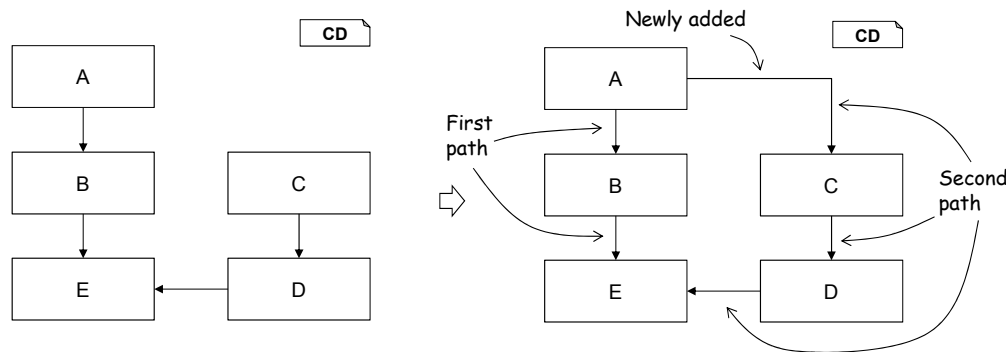


Abbildung 5.16: Beispiel für ein Szenario, in dem eine Klasse durch eine Modelländerung über verschiedene Pfade von einer anderen Klasse aus erreichbar ist.

Modellierung der Auswirkungsregel zur Changelog Analyse (Changelog-AR)

In Authoring existieren für alle Klassendiagramme, die als Eingabe für den Generator dienen, Changelogs. Diese werden von den Modellierern genutzt, um die durchgeführten Modelländerungen zu dokumentieren. Dadurch sollen Changelogs den Entwicklern einen einfachen Überblick über vergangene Änderungen bieten, um die Evolution der Modelle schneller erfassen zu können. Listing 5.15 zeigt einen Auszug aus einem Changelog, wobei aus Gründen der Geheimhaltung lediglich fiktive Modelländerungen dokumentiert sind. Konkret drückt dieser Changelog aus, dass am 06.03.2017 der Modellierer *km* eine Verschiebung der Klasse *Person* aus dem Paket *library.core* in das Paket *library.ext* vorgenommen hat (Zeile 8). Der gleiche Modellierer hat am 14.02.2017 zudem eine Assoziation hinzugefügt und ein Attribut umbenannt (Zeile 3 – 4).

```

1 2017-02-14, km
2 -----
3 - Added association from 'Library' to 'BookExemplar'
4 - Renamed 'isbn' in 'BookExemplar' to 'fullIsbn'
5
6 2017-03-06, km
7 -----
8 - Moved 'Person' from package 'library.core' to 'library.ext'
```

Listing 5.15: Auszug aus einem fiktiven Changelog. Der Changelog weist auf drei Modelländerungen hin, die von einem Modellierer durchgeführt wurden.

Die Auswirkungsregel Changelog-AR informiert die Entwickler darüber, dass die Changelogs modifiziert werden müssen, sobald eine Modelländerung festgestellt wird. Dabei schlägt die Auswirkungsregel einen Text vor, der in die Changelogs eingetragen werden kann.

Die Modellierung der Auswirkungsregel Changelog-AR ist in Listing 5.16 dargestellt. Änderungen bezüglich der Changelogs sind prinzipiell unkritisch (Zeile 6), da fehlerhafte oder veraltete Changelogs keine unmittelbaren Auswirkungen auf die Softwaresysteme haben. Allerdings werden dann die Changelogs als Informationsquelle zum Nachvollziehen der Evolution

der Modelle nutzlos. Die Wahrscheinlichkeit, dass die gemeldeten Checklistenhinweise tatsächlich relevant sind, ist hoch (Zeile 7). Der Grund dafür ist, dass die Auswirkungsregel auf jede Modelländerung hinweist und jede Modelländerung in den Changelogs dokumentiert werden muss.

```

1 impactRule "Changelog Analysis" {
2   description = "This rule checks whether new entries have to
3     be added to the changelog file. Take the following textual
4     summary of the identified model differences and copy it to
5     the changelog file."
6   severity = minor
7   probability = high
8
9   impact {
10    anyModelDiff() => {MDDesc}
11  }
12 }

```

IRS

Listing 5.16: Modellierung der Auswirkungsregel Changelog-AR.

Als einzige Bedingung für die Auswirkungsregel ist die benutzerdefinierte Bedingung `anyModelDiff()` (Zeile 10) angegeben. Diese Bedingung drückt aus, dass der zugeordnete Checklistenhinweis für jede Modelländerung gemeldet wird. Der Platzhalter `MDDesc` muss dabei durch eine Beschreibung der Modelländerung ersetzt werden.

5.5 Diskussion

5.5.1 Allgemeine Diskussion

Auswirkungsregeln verkörpern in dieser Arbeit die Auswirkungen von Änderungen von Klassendiagrammen auf ein Softwaresystem. Klassendiagramme eignen sich sehr gut als Anwendungsbeispiel, da Klassendiagramme häufig zur Modellierung der Struktur bzw. Architektur eines Softwaresystems eingesetzt werden [Rum11, HWRK11, Rum12, HWR14]. Ferner existieren ausgereifte Werkzeuge zur Identifizierung von Modelländerungen für Klassendiagramme. Das Konzept der Auswirkungsregeln ist aber nicht auf die Anwendung bei Klassendiagrammen beschränkt. Die Idee zur Modellierung von Auswirkungen und der darauf aufbauenden Checklistengenerierung lässt sich analog auf andere Modelltypen übertragen. Wie in Abschnitt 4.2.1 dargelegt, wird zu Beginn der Modelldifferenzberechnung eine Konvertierung der Klassendiagramme in das ECore-Format durchgeführt. Eine technische Voraussetzung für die Ausweitung des Ansatzes auf andere Modellarten ist folglich, dass weitere Konvertierungskomponenten bereitgestellt werden. Die größte Herausforderung besteht jedoch darin, die Bedingungen zu identifizieren, unter denen spezifische Modellunterschiede Auswirkungen auf ein Softwaresystem haben. Darüber hinaus lässt sich die Idee zur Modellierung von Auswirkungen von Änderungen konzeptuell ebenfalls auf Änderungen von Code übertragen. Dies erfordert allerdings, dass eine Differenzkomponente die Änderungen des Codes klassifiziert, so dass Auswirkungsregeln mit Hinblick auf verschiedene Typen von Code-Änderungen formuliert werden können.

Darüber hinaus gilt es zu beachten, dass die Entwickler in manchen Szenarien abwägen müssen, ob sie eine Auswirkungsregel bereitstellen oder den Generator direkt passend erweitern. Beispielsweise ist die Auswirkungsregel ORM-AR überflüssig, wenn der Generator direkt die passenden Mapping-Dateien generiert, ohne dass diese nach Modelländerungen manuell angepasst werden müssen. In einer derartigen Konstellation kann es allerdings dennoch sinnvoll sein eine Auswirkungsregel bereitzustellen. Dies gilt beispielsweise, wenn die Modellierung der Auswirkungsregel gegenüber der vollumfänglichen Erweiterung des Generators deutlich weniger aufwändig ist. Zudem kann eine Auswirkungsregel genutzt werden, während parallel der Generator erweitert wird. Dies stellt zwar einen Mehraufwand dar, führt aber dazu, dass die Entwickler begleitend zur Generatorerweiterung durch die Auswirkungsregel unterstützt werden.

Abschließend wird das Verhältnis zwischen den Kosten für die Entwicklung der Auswirkungsregeln und dem Nutzen der Auswirkungsregeln betrachtet. Die Hauptfrage, die vor der Implementierung einer Auswirkungsregel beantwortet werden muss, ist, ob sich der Spezifikations- und Implementierungsaufwand mit Hinblick auf den Nutzen der Auswirkungsregel rechnet. Diese Frage kann nicht pauschal beantwortet werden, sondern muss individuell für jede Auswirkungsregel separat analysiert werden. Ein wichtiger Gesichtspunkt bei der Bewertung des Nutzens einer Auswirkungsregel ist, dass eine Auswirkungsregel zu jeder Zeit ausgeführt werden kann, um zu prüfen, welche Auswirkungen Modelländerungen haben. Die Prüfung erfolgt zudem immer auf die gleiche Weise. Das kann einen entscheidenden positiven Einfluss auf die Software-Qualität haben, da es unwahrscheinlicher wird, dass die Entwickler vergessen die gemeldeten Entwicklungsaktivitäten auszuführen. Darüber hinaus können die Checklisten die Durchführung der Entwicklungsaktivitäten vereinfachen, da die Entwickler z.B. nicht selbst prüfen müssen, welche Modelländerungen Auswirkungen auf die Mapping-Dateien haben oder auf welche Weise die Property-Dateien angepasst werden müssen. Weiterhin muss bei der Einschätzung einer Auswirkungsregel beachtet werden, wie oft voraussichtlich neue Versionen des relevanten Softwaresystems erscheinen werden. Je öfter neue Versionen des Systems veröffentlicht werden, desto häufiger kann eine Auswirkungsregel angewendet werden und die Entwickler bei ihren Tätigkeiten unterstützen. Der Spezifikations- und Implementierungsaufwand kann sich daher im Laufe der Zeit amortisieren, da die Entwickler kontinuierlich unterstützt werden. Insbesondere muss in diesem Kontext beachtet werden, dass die manuelle Definition von Auswirkungsregeln die Erzeugung von präzisen Hinweisen, die auf spezifische Entwicklergruppen zugeschnitten sein können, ermöglicht. Ein automatischer Ansatz kann dies nicht auf die gleiche Weise umsetzen.

5.5.2 Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen

Die Integration des Ansatzes in den Softwareentwicklungsprozess eines produktiv eingesetzten Softwaresystems wird vor allem dadurch erleichtert, dass die Entwickler die Möglichkeit haben, Auswirkungsregeln für ausgewählte Problembereiche iterativ zu entwickeln. So können die Entwickler beispielsweise damit beginnen erste Bedingungen zu definieren, welche auf die kritischsten Auswirkungen hinweisen. Anschließend können sie die Regel sukzessive um weitere Bedingungen erweitern, um weitere Fälle abzudecken. In einem solchen iterativen Vorgehen

muss den Entwicklern aber stets klar sein, welche Modelländerungen in einem Bereich bereits zu einem bestimmten Zeitpunkt abgedeckt werden und welche nicht.

Ein weiterer Aspekt, der die Anwendung des Ansatzes erleichtert, ist, dass durch die Definition von benutzerdefinierten Bedingungen und Platzhaltern von Implementierungsdetails abstrahiert werden kann. Auf diese Weise können Auswirkungsregeln zunächst abstrakt definiert werden, ohne direkt sämtliche Implementierungsdetails festlegen zu müssen. Diese können stattdessen im Anschluss daran sukzessive implementiert werden. Dadurch kann einerseits die Entwicklung von Auswirkungsregeln vereinfacht, aber auch die Verständlichkeit der Auswirkungsregeln erhöht werden.

Die Anwendung des Ansatzes wird zudem darüber gefördert, dass durch den Ansatz nicht eingeschränkt wird, auf welche Arten von Auswirkungen die Auswirkungsregeln hinweisen. So können die Regeln unter anderem darauf hinweisen, dass bestehende Artefakte angepasst oder gelöscht werden müssen. Gleichermäßen können sie ausdrücken, dass neue Artefakte angelegt oder sonstige Entwicklungsaktivitäten durchgeführt werden müssen.

Die Möglichkeit für ausgewählte Modellelemente Delta-Benutzervorgaben definieren zu können, vereinfacht die Integration des Ansatzes in einen Softwareentwicklungsprozess zusätzlich. Dies ist dann relevant, wenn fehlerhafte Modellunterschiede gemeldet werden. Ohne Delta-Benutzervorgaben können die daraus resultierenden Checklisten unter Umständen nicht eingesetzt werden, da diese aufgrund der fehlerhaften Modellunterschiede auf unnötige oder fehlerhafte Entwicklungsaktivitäten verweisen können.

Ein Problem, das beim praktischen Einsatz von Auswirkungsregeln auftreten kann, ist, dass falsch-positive Hinweise durch Auswirkungsregeln gemeldet werden. Ein falsch-positiver Hinweis liegt dann vor, wenn gemäß Checkliste eine Entwicklungsaktivität durchgeführt werden muss, dies faktisch aber nicht erforderlich ist oder die Durchführung im schlimmsten Fall sogar zu Fehlern führt. Falls die Checklisten im Laufe der Zeit viele derartige falsch-positive Hinweise enthalten, könnte die Akzeptanz unter den Entwicklern rapide sinken. Dies könnte schließlich dazu führen, dass der Ansatz irgendwann nicht weiter eingesetzt wird. Um dieses Problem zu verringern, bietet der Ansatz die Möglichkeit die Wahrscheinlichkeit für die Korrektheit der gemeldeten Hinweise zu deklarieren. Folglich können die Entwickler als Erstes die Hinweise abarbeiten, die eine höhere Eintrittswahrscheinlichkeit besitzen. Bei den anderen Hinweisen ist dann direkt ersichtlich, dass die Hinweise unter Umständen nicht zutreffen.

Weiterhin gilt es zu beachten, dass der Ansatz bisher keine Mechanismen zum Konfliktmanagement zwischen verschiedenen Auswirkungsregeln beinhaltet. Das bedeutet, dass der Ansatz bisher keine Unterstützung dabei bietet zu erkennen, ob verschiedene Auswirkungsregeln einander widersprechen. Stattdessen wird vorausgesetzt, dass die Modellierer von Auswirkungsregeln oder die Entwickler, welche die Checklisten verarbeiten, dies selbstständig prüfen und etwaige Konflikte auflösen.

Die praktische Anwendung des Ansatzes wird in Abschnitt 5.7 im Rahmen einer Fallstudie verdeutlicht.

5.6 Verwandte Arbeiten

Existierende Arbeiten zur Auswirkungsanalyse fokussieren sich weitestgehend auf die Auswirkungen von Code-Änderungen [Leh11, LSLZ13]. Diese Ansätze nutzen üblicherweise statische oder dynamische Code-Analyse-Techniken zur Identifikation von beeinflussten Code-Teilen. Darüber hinaus existieren verschiedenste weitere Ansätze, z.B. Information Retrieval Techniken [CC05, CC06, BL10], Ansätze die Traceability-Techniken [vKG03, DLFO08, LDLL10] nutzen oder Ansätze die auf dem Mining von Software Repositories [CC05, KCM05, CC06, KCM07] basieren. Eine weitere Gruppe von Ansätzen basiert auf expliziten Auswirkungsregeln, die ausdrücken, welche Auswirkungen eine Änderung hat. Im weiteren Verlauf dieses Abschnitts liegt der Fokus auf diesen regelbasierten Ansätzen, da der in dieser Arbeit vorgestellte Ansatz ebenfalls regelbasiert ist. Im Folgenden werden zunächst Code-basierte Ansätze vorgestellt. Anschließend werden Techniken beschrieben, welche, wie der Ansatz in dieser Arbeit, die Auswirkungen von Modelländerungen untersuchen.

In [SLT⁺10] wird eine statische Code-Analyse-Technik vorgestellt. Diese beruht darauf, dass die Auswirkung einer Änderung von der Art der Änderung und der Abhängigkeit zwischen der geänderten Entität und anderen Entitäten abhängt. Dadurch sind die Auswirkungsregeln spezifisch für Änderungstypen und drücken aus, welche Entitäten (Klassen, Methoden, Attribute) von Änderungen eines bestimmten Typs einer bestimmten Entität betroffen sein können. Auswirkungsregeln in IRS drücken primär aus, dass die Auswirkungen von dem Änderungstypen abhängen. Darüber hinaus kann aber auch in IRS die Abhängigkeit eines geänderten Modellelements zu anderen Modellelementen beachtet werden. Beispielsweise hat das Löschen eines Attributs aus einer Klasse in einem bestimmten Kontext keine Auswirkungen, wenn das zugehörige Attribut zu einer Oberklasse der Klasse hinzugefügt wurde. Derartige Konstellationen können ebenfalls in IRS modelliert werden.

Auf ähnliche Weise setzt die Arbeit in [CKKL02] voraus, dass die Auswirkung einer Änderung von der Art der Änderung und von der Beziehung zwischen den Artefakten abhängt. Die Auswirkungsregeln in [CKKL02] sind boolesche Ausdrücke, die auf spezifische Änderungstypen und Beziehungstypen verweisen. Analog sind auch Auswirkungsregeln in IRS boolesche Ausdrücke, die ausgewertet werden, um zu entscheiden, ob eine Modelländerung Auswirkungen hat.

Ähnlich zu diesen Arbeiten basiert auch die Technik in [QVWM94] auf der Annahme, dass die Auswirkung von Änderungen von der Art der Änderung, der Abhängigkeitsbeziehung zwischen Entitäten und der Art der geänderten Entität abhängt. Zusätzlich dazu werden Propagierungsregeln genutzt, die darlegen, welche Arten von Änderungen auf den betroffenen Entitäten durchgeführt werden müssen. Eine Erweiterung dieser Technik wird in [BBE⁺95] beschrieben. Dort werden *potentielle Propagierungsregeln* eingeführt, die potentiell betroffene Entitäten finden können. Eine *potentielle Regel* ist vergleichbar zur Modellierung einer IRS-Auswirkungsregel, deren Wahrscheinlichkeit mit einem mittleren oder niedrigen Wert angegeben ist.

Im Gegensatz zu oben genannten Arbeiten untersucht der Ansatz in dieser Arbeit Modelländerungen und deren Auswirkungen. Weiterhin zielt diese Arbeit nicht primär auf die Identifikation von betroffenen Entitäten ab. Stattdessen liegt die Hauptmotivation in der Erzeugung von kon-

kreten Hinweisen bezüglich weiterer durchzuführender Aktivitäten.

In [BLO03a, BLO03b, BLOS06] wird ein regelbasierter Ansatz zur Identifikation von Auswirkungen von Modelländerungen in UML-Modellen vorgestellt. Das Ziel dieses Ansatzes besteht darin, über Auswirkungsregeln die Modellelemente zu identifizieren, die von den Modelländerungen betroffen sind. Dazu werden zu Beginn die Modellunterschiede zwischen zwei Modellversionen identifiziert. Im Anschluss daran werden die Auswirkungen der gefundenen Änderungen basierend auf Auswirkungsregeln berechnet, die in der Sprache OCL (Object Constraint Language) [OMG10a] spezifiziert sind. Diese OCL-Auswirkungsregeln liefern die Modellelemente, die von den Modelländerungen betroffen sind. Die Entwickler müssen folglich diese Modellelemente untersuchen und die zugehörigen UML-Modelle gegebenenfalls anpassen, um die Modelle konsistent zu den geänderten UML-Modellen zu halten.

Der Ansatz dieser Arbeit ist durch die Arbeiten [BLO03a, BLO03b, BLOS06] motiviert, da dort ebenfalls Auswirkungen von Modelländerungen untersucht werden. Weiterhin werden dort ebenfalls explizit modellierte Auswirkungsregeln eingesetzt, mit dem Unterschied, dass diese dort in OCL und in dieser Arbeit in der eigens entwickelten DSL IRS formuliert sind. Aus diesem Grund werden die Gemeinsamkeiten und Unterschiede der Ansätze im Folgenden detaillierter erläutert. In beiden Fällen werden initial die Modelländerungen mit einem Differenzberechnungsverfahren ermittelt und die Auswirkungsregeln auf die Modelländerungen angewendet. Ein Unterschied besteht darin, dass Fehler in den Modelländerungen in [BLO03a, BLO03b, BLOS06] bei jeder Ausführung manuell korrigiert werden müssen. Ein weiterer Unterschied ist, dass in [BLO03a, BLO03b, BLOS06] für jeden Änderungstyp genau eine Auswirkungsregel definiert ist. Bei der Verwendung von IRS ist es ebenfalls möglich eine Auswirkungsregel für jeden relevanten Änderungstyp anzugeben. Die modellierten Auswirkungsregeln kapseln jedoch üblicherweise die Auswirkungen in einem bestimmten Problembereich. Auf diese Weise können die Experten aus den entsprechenden Problembereichen die Auswirkungen von Modelländerungen definieren. Andernfalls müsste eine Vielzahl von Entwicklern kollaborieren, um die Auswirkungen zu definieren, die von einem bestimmten Änderungstyp ausgehen. Der wichtigste Unterschied zwischen den Arbeiten besteht aber darin, dass die Auswirkungsregeln in [BLO03a, BLO03b, BLOS06] dazu genutzt werden, um UML-Modellelemente zu identifizieren, die von den UML-Modelländerungen betroffen sind. Im Gegensatz dazu zielen die Auswirkungsregeln in dieser Arbeit darauf ab auf sämtliche Arten von Auswirkungen von Modelländerungen hinzuweisen.

In [LFR13] wird eine weitere Technik vorgestellt, welche die Auswirkungen von Änderungen von verschiedenen Arten von Artefakten, wie z.B. UML-Modellen oder Java-Code, untersucht. Als Erstes werden die verschiedenen Artefakte dazu in ein vereinheitlichtes EMF-Modell transformiert, über das alle relevanten Arten von Artefakten abgebildet werden können. Die resultierenden EMF-Modelle werden in EMFStore [EMFb] importiert, so dass sämtliche Artefakte über dieses Model Repository erreichbar sind. Explizit definierte Regeln bestimmen daraufhin zwischen welchen Arten von Artefakten welche Art von Abhängigkeitslink erzeugt werden sollte. Basierend darauf wird eine Änderungspropagierung über explizite Propagierungsregeln durchgeführt. Diese Regeln drücken im Wesentlichen aus, dass die Auswirkung einer Änderung von der Art der Abhängigkeitsbeziehung zwischen Artefakten und der Art der Änderung abhängt, wie bereits bei einigen zuvor beschriebenen Ansätzen.

Die Arbeit in [LFR13] ist allgemeingültiger als der Ansatz, der in dieser Arbeit beschrieben wird, da dort die Auswirkungen von Änderungen von verschiedenen Typen von Artefakten ermittelt werden. Eine Gemeinsamkeit beider Ansätze ist, dass Auswirkungen von Änderungen in Klassendiagrammen betrachtet werden können und dass zur Ermittlung der Auswirkungen explizite Regeln eingesetzt werden. Eine weitere Gemeinsamkeit besteht darin, dass in diesen Regeln Bedingungen formuliert werden können, die erfüllt sind, wenn bestimmte Arten von Modelländerungen identifiziert wurden. Ein Unterschied besteht darin, dass in [LFR13] im Rahmen der Änderungspropagierung ausschließlich darauf hingewiesen wird, welche Elemente der Artefakte von den Änderungen betroffen sind, z.B. dass eine spezifische Methode umbenannt werden muss. Demgegenüber kann über Auswirkungsregeln in dieser Arbeit auf notwendige Entwicklungsaktivitäten verwiesen werden, die nicht auf existierende Artefakte bezogen sind. Zudem ermöglicht es der Ansatz dieser Arbeit auf einfache Weise genaue Hinweise auf die weiteren Entwicklungsaktivitäten zu erzeugen, um so die Evolution eines Softwaresystems gezielt zu beherrschen.

Der Ansatz zur Auswirkungsanalyse der in dieser Arbeit vorgestellt wurde, ist somit sowohl durch die Arbeiten [BLO03a, BLO03b, BLOS06] als auch durch [LFR13] motiviert. In dieser Arbeit liegt der Fokus jedoch auf der freien Definition der Auswirkungen, so dass nicht nur betroffene Modellelemente oder Artefakte identifiziert werden können.

5.7 Fallstudie in der Industrie

5.7.1 Ziele und Forschungsfragen

Das Ziel dieser Fallstudie ist es zu validieren, dass der vorgestellte Ansatz in einem realen Softwaresystem anwendbar ist. Zu diesem Zweck wurden die folgenden Forschungsfragen abgeleitet:

- **RQ1:** Ist es mit Hinblick auf die Laufzeit des Ansatzes praktikabel den Ansatz für reale Modelle mit einer realen Evolutionshistorie anzuwenden?
- **RQ2:** Ist es über die Auswirkungsregeln möglich die relevanten nächsten Entwicklungsaktivitäten zu identifizieren?
- **RQ3:** Wie bewerten Entwickler des Softwaresystems den Ansatz?

5.7.2 Aufbau und Gegenstand der Studie

Der Ansatz wird in der Studie auf Authoring angewendet, da Authoring ein komplexes Softwaresystem ist, welches modellgetrieben entwickelt wurde. Der erste Schritt in der Durchführung der Studie war die Berechnung der Modellunterschiede zwischen 20 Versionen der drei wichtigsten Klassendiagramme für Authoring. Die Klassendiagramme enthielten zwischen 100 und 300 Klassen. Alle aufeinanderfolgenden Versionen wurden paarweise verglichen, so dass $(20 - 1) \cdot 3 = 57$ Modellvergleiche durchgeführt wurden. Die Änderungen der Klassendiagramme erfolgten im Wesentlichen innerhalb eines Jahres.

Zur Beantwortung der ersten Forschungsfrage wurde der Ansatz, wie zuvor beschrieben, jeweils auf zwei aufeinanderfolgende Klassendiagrammversionen angewendet und die Ausführungszeit protokolliert. Weiterhin wurde mit Hilfe eines Klassendiagrammgenerators ausgehend von einem existierenden Klassendiagramm aus der realen Modellhistorie ein Klassendiagramm generiert, welches mit 4000 Klassen eine deutlich höhere Anzahl von Klassen beinhaltete. Dadurch wurde validiert, dass der Ansatz auch bei sehr großen Klassendiagrammen eingesetzt werden kann. Auf dem generierten Klassendiagramm wurden anschließend 500 Änderungsoperationen ausgeführt. Diese deckten die verschiedenen Änderungsarten ab, die für die entwickelten Auswirkungsregeln relevant sind. Der Ansatz wurde schließlich auf das ursprüngliche Klassendiagramm und das resultierende geänderte Klassendiagramm angewendet und die Ausführungszeit wurde erneut protokolliert. Die Experimente wurden jeweils auf einem Dell Latitude E3620 (8GB RAM; Intel i7-2620M 2.7 Ghz) Rechner ausgeführt.

Die zweite Forschungsfrage wurde beantwortet, indem sämtliche Auswirkungsregeln aus Abschnitt 5.4.3 angewendet wurden. Die Klassendiagrammversionen wurden dazu paarweise verglichen, wie zuvor erläutert. Anschließend wurde jeweils eine Checkliste für die Modelländerungen generiert, die zwischen den verglichenen Klassendiagrammversionen identifiziert wurden. Die von den Auswirkungsregeln gemeldeten Entwicklungsaktivitäten wurden anschließend manuell analysiert und geprüft, ob die Aktivitäten wirklich durchgeführt werden mussten und ob sie durchgeführt wurden.

Um die dritte Forschungsfrage zu beantworten, wurden fünf Checklisten ausgewählt, die bei der Anwendung des Ansatzes auf die verschiedenen Klassendiagrammversionen entstanden sind. Diese Checklisten wurden mit Hinblick auf deren Größe ausgewählt, um verschiedene Szenarien abzudecken. Weiterhin wurde darauf geachtet, dass die Checklisten auf verschiedenartige Entwicklungsaktivitäten hinweisen, um ein möglichst großes Spektrum der Auswirkungsregeln abzudecken. Die ausgewählten Checklisten wurden an fünf Entwickler übergeben. Unter diesen waren drei erfahrene Authoring-Entwickler und die anderen beiden kannten das System zwar gut, mussten aber nur selten Entwicklungsaktivitäten durchführen. Darüber hinaus wurden Fragebögen mit den folgenden Fragen an die Entwickler verteilt. Diese konnten die Entwickler mit Freitext beantworten. Zudem konnten die Entwickler einen Freitext als weiteres Feedback niederschreiben. Die vollständigen Antworten aller Entwickler zu diesen Fragebögen sind in Abschnitt G.1 aufgeführt.

- **DQ1:** Wie sehr vereinfachen die Checklisten die Ausführung der weiteren Entwicklungsaktivitäten?
- **DQ2:** Wie bewerten Sie den Aufwand zur Abarbeitung der Checklisten?
- **DQ3:** Welche Gründe sprechen für Sie für und gegen die Nutzung der Checklisten in Zukunft?

5.7.3 Ergebnisse und Diskussion

Forschungsfrage RQ1

RQ1: Ist es mit Hinblick auf die Laufzeit des Ansatzes praktikabel den Ansatz für reale Modelle mit einer realen Evolutionshistorie anzuwenden?

Die komplette Ausführungszeit für die Ausführung aller 57 Modellvergleiche samt der Erzeugung der Checklisten betrug 25 Sekunden. Für das Klassendiagramm mit den 4000 Klassen betrug die gesamte Ausführungszeit durchschnittlich 12 Sekunden, inklusive der Checklisten-erzeugung. In beiden Fällen wurde der Ansatz jeweils zehn Mal ausgeführt und die angegebenen Ausführungszeiten gemittelt.

Diese Messungen dienen als starker Indikator dafür, dass die Anwendung des Ansatzes in einem realen Projektumfeld mit realen Modellen und selbst bei einer hohen Anzahl von Modelländerungen praktikabel ist. Die gemessenen Werte sind dabei etwas schneller als die gemessenen Werte in der Fallstudie in [MR14a], trotz der höheren Anzahl von Auswirkungsregeln. Der Grund dafür ist, dass das Tooling mit Hinblick auf die Performance weiter optimiert wurde.

Forschungsfrage RQ2

RQ2: Ist es über die Auswirkungsregeln möglich die relevanten nächsten Entwicklungsaktivitäten zu identifizieren?

Ergebnisse zu XMLMig-AR Die Auswirkungsregel XMLMig-AR hat in 15 Fällen auf die Notwendigkeit der Implementierung einer XML-Migrationsklasse hingewiesen. In diesen 15 Fällen waren sämtliche neun Migrationsklassen enthalten, die tatsächlich von den Entwicklern bereitgestellt wurden. An dieser Stelle sei noch angemerkt, dass acht dieser neun Migrationsklassen nicht direkt nach den relevanten Modelländerungen erstellt wurden. Stattdessen wurden sie erst später im Entwicklungsprozess bereitgestellt, nachdem das Test-Team Fehler gefunden hat. Unter den restlichen sechs Migrationsklassen war eine fälschlicherweise gemeldete Migrationsklasse und zwei der Migrationsklassen wurden nicht implementiert, obwohl sie hätten implementiert werden müssen. Die fälschlicherweise gemeldete Migrationsklasse war darin begründet, dass in der betroffenen Entitätsklasse handgeschriebene Änderungen vorgenommen wurden, welche die Durchführung der XML-Migration überflüssig gemacht haben. Die beiden fehlenden Migrationsklassen wurden auf Basis dieser Erkenntnisse nachträglich implementiert. Zudem müssen die drei restlichen Migrationsklassen für das nächste Release implementiert werden.

Die Auswirkungsregel hat aber nicht nur auf die tatsächlich umgesetzten Migrationsklassen hingewiesen. In zwölf Fällen wurden konkrete Migrationsklassen vorgeschlagen, von denen zehn Migrationsklassen ohne Veränderung direkt hätten eingesetzt werden können. Die anderen beiden Migrationsklassen betrafen Sonderfälle, die von der Logik, welche die Migrationsklassen vorschlägt, nicht beachtet wurden.

Im Rahmen der Modellierung der Auswirkungsregel XMLMig-AR wurden große Teile der Auswirkungsregel in Java implementiert. Der Grund dafür war, dass es viele zu prüfende Bedingungen gab, die teilweise stark verschachtelt werden mussten. Wären sämtliche dieser Bedingungen über IRS ausgedruckt worden, wäre die resultierende Auswirkungsregel sehr unübersichtlich geworden. Stattdessen wurden für diese Prüfungen benutzerdefinierte Bedingungen bereitgestellt, die anschließend in Java implementiert wurden. In komplexen Szenarien ist dies die einzige Möglichkeit, um die Komplexität der Auswirkungsregeln zu reduzieren. Die Erfahrungen bei der Modellierung und Implementierung dieser Auswirkungsregel legen nahe, dass Entwickler im Allgemeinen in komplexen Szenarien größere Teile der Auswirkungsregel selbst implementieren müssen.

Ergebnisse zu ORM-AR Die Untersuchung der Ergebnisse der Auswirkungsregel zeigte, dass die Auswirkungsregel auf sämtliche Änderungen verwiesen hat, die auf den Mapping-Dateien tatsächlich durchgeführt wurden. Dies lässt darauf schließen, dass die Wahrscheinlichkeit, dass Entwickler vergessen, die Mapping-Dateien ordnungsgemäß anzupassen, gering ist.

Ergebnisse zu DBMig-AR Die Auswirkungsregel DBMig-AR lieferte insgesamt sieben Hinweise auf Modelländerungen, die eine Datenmigration in der Datenbank erforderten. Diese Datenmigrationen wurden allesamt auch tatsächlich implementiert. Darüber hinaus wurden noch weitere Datenmigrationen von den Entwicklern implementiert. Diese ließen sich aber nicht auf Modelländerungen zurückführen, sondern waren in anderen Änderungen der Anforderungen oder in Fehlerkorrekturen in den Daten begründet. Analog zu der Auswirkungsregel ORM-AR zeigen diese Ergebnisse, dass die Wahrscheinlichkeit dafür, dass die Entwickler vergessen, die erforderlichen Änderungen durchzuführen, gering ist.

Ergebnisse zu Prop-AR Eine Analyse der Ergebnisse dieser Auswirkungsregel zeigte, dass von den 275 gemeldeten Einträgen, die zu den Property-Dateien hätten hinzugefügt werden müssen, 254 hinzugefügt wurden. Die anderen Einträge fehlen in der aktuellen Version des Systems und müssen noch hinzugefügt werden. Die genannten Zahlen stammen aus der Fallstudie, die in [MR14a] vorgestellt wurde. Aufgrund von kleinen Änderungen in dem Umfeld der Property-Dateien wurden die Ergebnisse zu dieser Auswirkungsregel nach den Änderungen erneut bestimmt. Diese erneute Ausführung zeigte, dass von 260 gemeldeten Einträgen 255 tatsächlich hinzugefügt wurden. Die fehlenden fünf Einträge stimmen im Übrigen nicht mit den Einträgen überein, die in der ursprünglichen Fallstudie bereits als fehlend identifiziert wurden. Das zeigt, dass die Entwickler im Laufe der Zeit einige fehlende Einträge noch hinzugefügt haben.

Ergebnisse zu BR-AR Die Auswirkungsregel BR-AR hat insgesamt 163 Hinweise gemeldet, die auf mögliche neue Business Rules hingedeutet haben. Von diesen wurden lediglich fünf Hinweise zu Business Rules tatsächlich umgesetzt. Ob die verbliebenen 158 Hinweise auf Business Rules hingewiesen haben, die hätten implementiert werden müssen, lässt sich rückblickend nur schwer beurteilen. Eine Suche nach Bugs in diesem Kontext ergab zumindest keine Treffer. Aus diesem Grund kann davon ausgegangen werden, dass die meisten dieser weiteren Hinweise

nicht relevant waren. Auch unter Beachtung der Tatsache, dass für die Auswirkungsregel angegeben ist, dass die Wahrscheinlichkeit für das Eintreffen der gemeldeten Hinweise gering ist, ist die Anzahl der irrelevanten Hinweise im Vergleich zu den relevanten Hinweisen zu hoch. Auf Basis dieser Ergebnisse kann abgeleitet werden, dass diese Auswirkungsregel nur sehr bedingt hilfreich ist, da sie zu viele irrelevante Hinweise meldet.

Ergebnisse zu Changelog-AR Die Auswirkungsregel Changelog-AR lieferte 415 Hinweise auf Änderungen der Changelog-Dateien. Dabei wurde sowohl für jede Änderung eines Tagged Values als auch für jede Änderung eines Modellelements ein separater Hinweis gemeldet. Von diesen wurden lediglich 208 tatsächlich dokumentiert. Eine genauere Analyse zeigte, dass von den fehlenden 207 Änderungen insgesamt 114 Änderungen einen einzelnen Commit und 161 Änderungen vier Commits betreffen. Dies zeigt, dass die meisten Commits korrekt dokumentiert wurden und die fehlenden Einträge auf wenige Commits verteilt sind. Die weiteren fehlenden Einträge betrafen fast ausschließlich Änderungen von Tagged Values. Demgegenüber wurden die sonstigen Änderungen von Modellelementen größtenteils ordnungsgemäß dokumentiert. Dessen ungeachtet zeigt dieses Ergebnis, dass die Auswirkungsregel dabei helfen kann, die Changelog-Dateien auf dem aktuellen Stand zu halten.

Zusammenfassung Die Auswirkungsregeln waren, abgesehen von der Auswirkungsregel BR-AR, in allen untersuchten Fällen in der Lage, auf die durchgeführten Entwicklungsaktivitäten hinzuweisen. Vor allem bei der Implementierung der XML-Migrationen und der Pflege der Changelog-Dateien wurden Entwicklungsaktivitäten identifiziert, die vergessen wurden durchzuführen. Bei der Implementierung der XML-Migrationen könnte dies darin begründet sein, dass es komplexe Bedingungen gibt, welche das Bereitstellen der Migrationsklassen erfordern. Bei den Changelog-Dateien zeigte sich zudem, dass manchmal komplette Commits nicht eingearbeitet wurden und vor allem Änderungen der Tagged Values nicht vollständig dokumentiert wurden. Bei den Auswirkungsregeln ORM-AR und DBMig-AR haben die Entwickler die notwendigen Änderungen auch ohne Auswirkungsregeln durchgeführt. Dessen ungeachtet kann die Verwendung dieser Auswirkungsregel hilfreich sein. So weisen die Auswirkungsregeln unmittelbar auf die Modelländerungen hin, welche die erforderlichen Änderungen induzieren. Die Entwickler können daraufhin die Änderungen gezielt durchführen. Darüber hinaus erleichtert das explizite Hinweisen auf die notwendigen Änderungen die Ausführung der Änderungen durch andere Entwickler, so dass die Änderungen leichter auf weitere Entwickler verteilt werden können. Bei der Auswirkungsregel BR-AR zeigte sich hingegen, dass zu viele irrelevante Hinweise gemeldet werden, so dass diese Regel die Entwickler nur sehr bedingt unterstützt.

Forschungsfrage RQ3

RQ3: Wie bewerten Entwickler des Softwaresystems den Ansatz?

- **DQ1:** Wie sehr vereinfachen die Checklisten die Ausführung der weiteren Entwicklungsaktivitäten?

- **DQ2:** Wie bewerten Sie den Aufwand zur Abarbeitung der Checklisten?
- **DQ3:** Welche Gründe sprechen für Sie für und gegen die Nutzung der Checklisten in Zukunft?

Die vollständigen Antworten zu den Fragebögen sind in Abschnitt G.1 dargelegt. Nachfolgend werden lediglich einige dieser Antworten vorgestellt. Der Fokus liegt darauf, die grundsätzliche Rückmeldung der Entwickler zu den Fragen vorzustellen.

Bezüglich der Frage DQ1 antworteten vier Entwickler, dass die Checklisten die Durchführung der weiteren erforderlichen Entwicklungsaktivitäten erleichtern. Demgegenüber merkte ein Entwickler an, dass die Checklisten hilfreich wären, wenn sie weniger Details beinhalten würden. Beispielsweise antwortete ein Entwickler: „Die Informationen, die in den Checklisten enthalten sind, sind sehr hilfreich.“. Ein anderer Entwickler schrieb: „Die Wahrscheinlichkeit, dass eine erforderliche Entwicklungstätigkeit vergessen wird, sinkt. Die Checklisten animieren dazu auf strukturierte Weise zu arbeiten.“.

Der gleiche Entwickler, der die Größe der Checkliste bei der ersten Frage kritisierte, antwortete bei der zweiten Frage DQ2 analog: „Die Checklisten sind größer als erforderlich. Ihre Größe wirkte zu Beginn auf mich abschreckend.“ Die anderen vier Entwickler antworteten, dass die Kosten zum Abarbeiten der Checklisten akzeptabel sind. Allerdings merkte auch unter diesen ein Entwickler an, dass einige Details in den Checklisten weggelassen werden könnten. Beispielsweise gab ein Entwickler an: „Ich erwarte, dass die Checklisten die Entwicklungsaktivitäten beschleunigen.“. Ferner merkte ein Entwickler an, dass der Detailgrad der Checklisten genau richtig sei.

Eine Analyse der Antworten zur dritten Frage DQ3 ergab, dass vier Entwickler die Checklisten in Zukunft gerne nutzen möchten und dass ein Entwickler sie nutzen würde, wenn sie kürzer wären. Einer der vier Entwickler antwortete beispielsweise: „Ich würde die Checklisten immer verwenden. Auf diese Weise kann man nicht vergessen bestimmte Entwicklungsaktivitäten auszuführen, unabhängig davon, wie groß oder klein die Änderungen sind.“.

Im Bereich zur Angabe von offenem Feedback schlugen zwei Entwickler Optimierungen für das Format der Checkliste vor, um den Umgang mit den Checklisten zu vereinfachen.

Alle Entwickler gaben an, dass sie die Idee zur Erzeugung von Checklisten sehr gut finden. Die Rückmeldung der Entwickler offenbarte aber auch, dass es unterschiedliche Meinungen bezüglich des Detailgrads der Checklisten gibt. Aus diesem Grund plant DSA den Ansatz zu erweitern und Checklisten mit unterschiedlichem Detailgrad zu erzeugen - insbesondere eine umfangreiche Checkliste mit Detailinformationen und eine kurze Checkliste, die nur die wesentlichen Informationen beinhaltet.

5.7.4 Gültigkeit der Fallstudie

Eine Gefahr für die Gültigkeit der Fallstudie besteht darin, dass sich die Ergebnisse der Fallstudie nicht auf andere Softwaresysteme übertragen lassen, da der Ansatz nur auf ein einzelnes Softwaresystem angewendet wurde. Durch die Auswahl des untersuchten Softwaresystems wird

diese Gefahr jedoch abgeschwächt. Ein Grund hierfür ist, dass die Tatsache, dass ein elementarer Teil des Softwaresystems auf Basis von Klassendiagrammen generiert wird und dass Änderungen in diesen Klassendiagrammen weitere manuelle Änderungen bedingen können, nicht spezifisch für Authoring ist. Darüber hinaus wurden in der Fallstudie keine spezifischen Eigenschaften von Authoring vorausgesetzt, die dazu geführt haben, dass der Ansatz besonders gut im Kontext von Authoring angewendet werden konnte. Aus diesen Gründen kann davon ausgegangen werden, dass die Fallstudie dafür geeignet ist, auch Rückschlüsse auf andere Softwaresysteme ziehen zu können und dass der Ansatz ebenfalls in anderen Systemen eingesetzt werden kann, um dort die Software Evolution zu unterstützen.

Eine weitere mögliche Gefahr für die Gültigkeit einer Fallstudie dieser Art ist, dass andere Änderungen als die identifizierten Modelländerungen die gemeldeten Entwicklungsaktivitäten erfordert haben. Um diesem Problem entgegenzuwirken wurden die gemeldeten Entwicklungsaktivitäten genau analysiert. Dabei wurde unter anderem geprüft, ob sich die generierten Artefakte auf die erwartete Weise geändert haben, so dass z.B. ein Erzeugen oder Überarbeiten bestimmter handgeschriebener Artefakte unabdingbar war. Darüber hinaus wurde geprüft, ob die gemeldeten Änderungen durchgeführt wurden, nachdem die jeweiligen Modelländerungen vorgenommen wurden. Dieser Prüfung liegt die Annahme zu Grunde, dass die Änderungen nicht pro-aktiv vorgenommen wurden. Gemäß dieser Prüfungen haben ausschließlich die identifizierten Modelländerungen die Durchführung der gemeldeten Entwicklungsaktivitäten erfordert.

Kapitel 6

Generierung von Migrationsfunktionalität zur Evolution von Daten

Datenmodelle definieren die Struktur der Daten, die durch ein Softwaresystem verarbeitet werden. Damit bilden sie einen wichtigen Bestandteil bei der Modellierung eines Softwaresystems [FHR08, VV08]. In dieser Arbeit werden zur Modellierung von Datenmodellen Klassendiagramme eingesetzt. Sobald ein Generator aufbauend auf Datenmodellen Artefakte generiert, auf Basis derer Daten erzeugt werden, können Änderungen der Datenmodelle zu Änderungen der Struktur dieser Daten führen. Dieser Fall tritt beispielsweise ein, wenn aus Datenmodellen Klassen generiert werden und in einem Softwaresystem werden Instanzen dieser Klassen serialisiert und deserialisiert.

Im Laufe der Zeit erzeugen die Benutzer eines Softwaresystems üblicherweise eine Vielzahl von Daten. Wird ein Datenmodell geändert, sind diese Daten unter Umständen nicht mehr konform zu dem Datenmodell. Dies hat zur Folge, dass sie in der neuen Version des Softwaresystems nicht mehr ordnungsgemäß verarbeitet werden können. Für die Benutzer des Softwaresystems sind diese Daten jedoch von zentraler Bedeutung, da ein Verlust der Daten bedeutet, dass sie neu angelegt werden müssen. Aus diesem Grund müssen Datenmigrationen bereitgestellt werden, an die es zwei primäre Anforderungen gibt: 1. die Daten müssen so migriert werden, dass sie konform zu der neuen Version der Datenmodelle sind und 2. die Migrationen sollten so implementiert sein, dass möglichst viele der bestehenden Daten beibehalten werden können. Hierbei gilt es zu beachten, dass die bestehenden Daten nicht in allen Situationen beibehalten werden können. Dieser Aspekt wird an verschiedenen Stellen in diesem Kapitel noch ausführlicher betrachtet.

Auf der Ebene der Klassendiagramme, welche die Datenmodelle beschreiben, wird in dieser Arbeit zwischen primären (siehe Definition 11) und sekundären Klassen (siehe Definition 12) unterschieden.

Definition 11 (Primäre Klasse)

Eine primäre Klasse eines Klassendiagramms zeichnet sich dadurch aus, dass zu ihr eigenständige Daten vorliegen können. Sämtliche zu migrierenden Daten sind stets Daten, die zu primären Klassen erzeugt wurden.

Definition 12 (Sekundäre Klasse)

Eine sekundäre Klasse eines Klassendiagramms ist dadurch gekennzeichnet, dass zu ihr keine eigenständigen Daten vorliegen. Stattdessen sind die Daten zu sekundären Klassen stets in den Daten zu primären Klassen eingebettet.

Sowohl primäre als auch sekundäre Klassen sind in den Klassendiagrammen durch spezifische Stereotypen markiert. Erstere werden in diesem Kapitel über den Stereotypen `PE` und letztere über den Stereotypen `SC` markiert. Zur Vereinfachung der Beispiele werden die Stereotypen weggelassen, wenn sie für ein Beispiel unerheblich sind. Jede Klasse, die keinen dieser Stereotypen besitzt, ist für die Datenmigration nicht relevant und wird im Folgenden nicht betrachtet.

Im Falle von Authoring werden einerseits Daten zwischen Clients und Servern ausgetauscht und darüber hinaus Daten serverseitig in einer Datenbank abgelegt. In beiden Fällen soll als Format zum Austauschen und Speichern der Daten JSON eingesetzt werden. Aus Definition 11 folgt somit, dass in beiden Fällen JSON-Daten zu primären Klassen migriert werden müssen. Zur Vereinfachung der Beschreibung wird dabei davon ausgegangen, dass die Datenmigrationen JSON-Dokumente als Eingabe erhalten und die migrierten JSON-Dokumente als Ergebnis liefern. Der Ansatz, der in diesem Kapitel vorgestellt wird, ist jedoch nicht auf die Migration von Daten im JSON-Format beschränkt. Die Ausführungen lassen sich analog auf sämtliche textbasierte Formate übertragen, die hierarchisch strukturiert sind.

In Abschnitt 6.1 werden einleitend die Grundlagen dargelegt, auf denen die darauffolgenden Erklärungen aufbauen. Danach gibt Abschnitt 6.2 einen Überblick über den Ansatz zur Generierung von Migrationsfunktionalität. Der erste Teil dieses Ansatzes besteht in der Ermittlung von Pfaden zu Modellelementen und wird in Abschnitt 6.3 erklärt. Auf Basis dieser Pfade kann zu den Daten navigiert werden, die migriert werden müssen. Im Anschluss daran wird in Abschnitt 6.4 dargelegt, wie das Differenzmodell, auf dessen Basis die Migrationsfunktionalität generiert wird, vereinfacht werden kann. Anschließend wird in Abschnitt 6.5 beschrieben, wie Datenmigrationen für spezifische Arten von Modelländerungen umgesetzt werden. Die Generierung dieser Datenmigrationen wird danach in Abschnitt 6.6 erläutert. Es folgt eine Diskussion des Ansatzes in Abschnitt 6.7 und eine Betrachtung der verwandten Arbeiten in Abschnitt 6.8. Abschließend werden in Abschnitt 6.9 die Ergebnisse einer Fallstudie präsentiert, in welcher der Ansatz angewendet wurde.

6.1 Grundlagen

6.1.1 JSON

Die JavaScript Object Notation (JSON) [Bra14] ist ein textbasiertes und menschenlesbares Format zum Austausch von strukturierten Daten zwischen Softwaresystemen. JSON ist stark an JavaScript-Objekte angelehnt, obgleich es kleinere Unterschiede zwischen JSON und JavaScript-Objekten gibt. Diese werden in diesem Kapitel aber nicht näher betrachtet. Darüber hinaus ist JSON unabhängig von einer konkreten Programmiersprache. Zur Verarbeitung von JSON existieren jedoch Parser in vielen verschiedenen Programmiersprachen.

Jedes JSON-Dokument startet mit einer geöffneten und endet mit einer geschlossenen geschweiften Klammer. Innerhalb dieser Klammern können Schlüssel-Wert-Paare spezifiziert sein, wobei die Schlüssel jeweils in Anführungszeichen gesetzt werden müssen. Ein Schlüssel wird gefolgt von einem Doppelpunkt und die einzelnen Schlüssel-Wert-Paare müssen durch Komma voneinander getrennt sein. Als Wert zu einem Schlüssel kann ein Nullwert, ein boolescher Wert, ein Zahlenwert, eine Zeichenkette, ein JSON-Objekt oder eine Liste angegeben werden. Ein

JSON-Objekt wird wiederum durch eine geöffnete geschweifte Klammer eingeleitet und endet mit einer geschlossenen geschweiften Klammer. Innerhalb dieser Klammern können abermals Schlüssel-Wert Paare stehen, die Auskunft über die Werte des gekapselten JSON-Objekts geben. Eine Liste enthält eine durch Kommata separierte Liste von Werten (Nullwerte, boolesche Werte, Zahlenwerte), JSON-Objekten oder anderen Listen.

Abbildung 6.1 zeigt auf der linken Seite ein beispielhaftes Datenmodell. Dieses enthält die primäre Klasse $x.y.A$ sowie die beiden sekundären Klassen $x.y.C$ sowie $x.y.G$. In der Mitte in Abbildung 6.1 ist ein Objektdiagramm gezeigt, welches konform zu diesem Datenmodell ist. Auf der rechten Seite ist das JSON-Dokument gezeigt, welches zu diesem Objektdiagramm korrespondiert. In diesem JSON-Dokument ist zu dem Schlüssel b eine Liste von JSON-Objekten hinterlegt, da die zugehörige Assoziation mehrwertig ist. Diese enthält ein JSON-Objekt für eine $x.y.C$ Instanz, in dem Werte zu den Schlüsseln d und e hinterlegt sind. Darüber hinaus ist eine Liste zu dem Schlüssel f hinterlegt. Diese kapselt zwei JSON-Objekte für $x.y.G$ Instanzen, in denen jeweils der Schlüssel h angegeben ist.

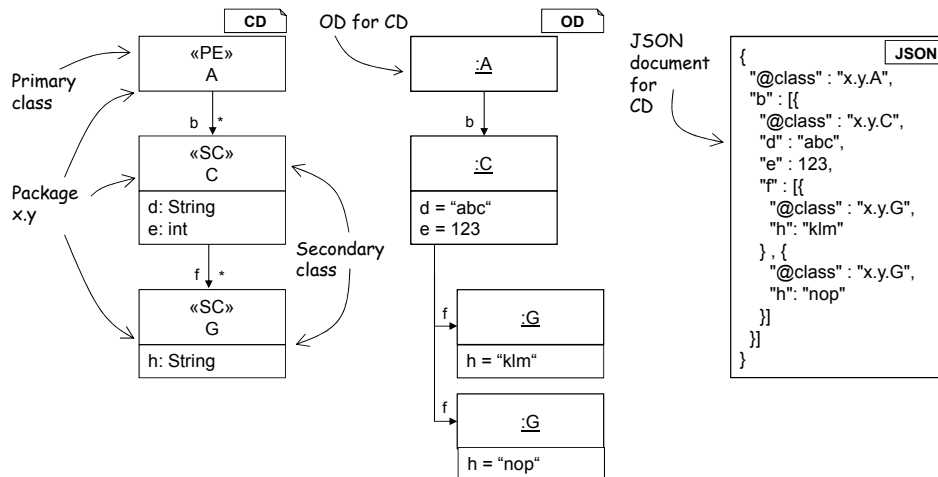


Abbildung 6.1: Beispielhaftes Datenmodell (linke Seite), beispielhaftes Objektdiagramm zu dem Datenmodell (Mitte) und JSON-Dokument konform zu diesem Objektdiagramm (rechte Seite).

In dem gezeigten JSON-Dokument gibt der Schlüssel `@class` den vollqualifizierten Namen der Klasse an, deren Instanz durch das JSON-Objekt repräsentiert wird. Diese Information liegt standardmäßig nicht in JSON-Dokumenten vor. Stattdessen ist dies in der Nutzung von Jackson [Jac] zur Serialisierung von Java-Objekten nach JSON und zur Deserialisierung von JSON-Dokumenten nach Java begründet. Mittels Jackson können diese Typinformationen in JSON-Dokumenten abgelegt werden. Für die praktische Nutzung von JSON-Dokumenten sind diese expliziten Informationen von elementarer Bedeutung, da ohne sie bei der Deserialisierung keine JSON-Daten zu polymorphen Typen verarbeitet werden können. In diesem Kapitel werden nur in solchen Beispielen die `@class` Schlüssel angegeben, in denen das Wissen über die repräsentierte Instanz benötigt wird.

Ein weiteres wichtiges Feature von Jackson ist die Unterstützung für die Handhabung von zyklischen Objektstrukturen. Intern verwaltet Jackson dazu für jedes JSON-Objekt einen eindeutigen Identifizierer. Über diesen kann erkannt werden, ob ein JSON-Objekt bei der Serialisierung bereits betrachtet wurde. In den gezeigten JSON-Dokumenten wird zur Vereinfachung der Beispiele aber von diesem Sachverhalt abstrahiert.

Darüber hinaus gilt es zu beachten, dass in dieser Arbeit davon ausgegangen wird, dass jedes JSON-Dokument ausschließlich Daten zu genau einer Instanz einer primären Klasse enthält. Verweist diese Instanz auf eine andere Instanz einer primären Klasse, enthält das JSON-Dokument lediglich einen Verweis auf einen eindeutigen Identifizierer zu dieser Instanz. Die Daten zu dieser referenzierten Instanz sind hingegen in einem anderen JSON-Dokument abgelegt.

Jackson wird in dieser Arbeit nicht nur zur Serialisierung und Deserialisierung von JSON-Dokumenten, sondern auch zur Migration von JSON-Dokumenten eingesetzt. Dazu werden die JSON-Dokumente über die Jackson API eingelesen und der resultierende JSON-Objektbaum wird modifiziert. Alternativ können die Datenmigrationen auch über andere vergleichbare Bibliotheken oder über JavaScript-Funktionen umgesetzt werden.

6.1.2 Durchführung von Datenmigrationen

Änderungen in den Datenmodellen können zu Änderungen in der Struktur der Daten führen. Angenommen das Datenmodell aus Abbildung 6.1 wird geändert, in dem beispielsweise das Attribut h der Klasse $x.y.G$ in h_{New} umbenannt wird. Anschließend ist das in der Abbildung gezeigte JSON-Dokument nicht mehr konform zu dem neuen Datenmodell. Damit das gezeigte JSON-Dokument dennoch in der neuen Version verarbeitet werden kann, muss es migriert werden. Dazu werden in dieser Arbeit die Pfade zu der Klasse bestimmt, die von einer Modelländerung betroffen ist. In dem beschriebenen Szenario ist das die Klasse $x.y.G$, da diese das umbenannte Attribut enthält. Ferner müssen die Pfade von einer primären Klasse ausgehend starten. In diesem Beispiel resultiert dies für die primäre Klasse $x.y.A$ in dem Pfad $\{b, f\}$. Über diesen Pfad kann ausgehend von dem Wurzelobjekt eines JSON-Dokuments zu den JSON-Objekten, die von der Modelländerung betroffen sind, navigiert werden. In dem Beispiel werden als Erstes die JSON-Objekte zu dem Schlüssel b bestimmt. Für die resultierenden JSON-Objekte werden wiederum die JSON-Objekte zu dem Schlüssel f ermittelt. Für diese kann die Umbenennung des Schlüssels h in h_{New} erfolgen, wie in Abbildung 6.2 verdeutlicht.

Wie diese Pfade ermittelt werden und wie aufbauend darauf die Datenmigrationsfunktionalität generiert wird, wird in den weiteren Abschnitten dieses Kapitels ausführlich erläutert.

6.2 Überblick zur Generierung von Migrationsfunktionalität

6.2.1 Einsatzszenarien

Für die generierte Datenmigrationsfunktionalität gibt es in dieser Arbeit zwei Einsatzszenarien:

- Die Migration von Datenbankinhalten
- Die Umwandlung von Daten, die zwischen Clients- und Servern versendet werden

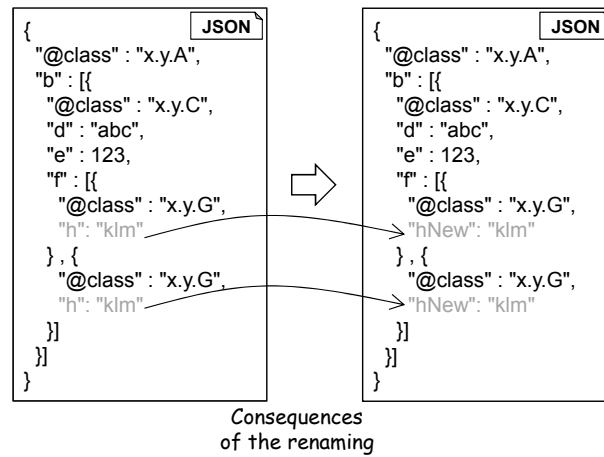


Abbildung 6.2: Auswirkungen der Umbenennung des Attributs h der Klasse $x.y.G$ in $hNew$.

Eine Herausforderung in beiden Einsatzszenarien ist, dass es möglich sein muss, Daten aus einem beliebigen Versionsstand v_a in einen beliebigen anderen Versionsstand v_n zu migrieren. Zur Bewältigung dieser Herausforderung wird in dieser Arbeit nach jedem Release v_n von Authoring die Datenmigrationsfunktionalität für den Übergang von Version v_{n-1} zu v_n generiert. Auf diese Weise lässt sich jeder beliebige Versionssprung einer älteren Version v_a zu v_n über eine schrittweise Ausführung der Migrationsfunktionalität realisieren:

1. Bestimmung des Versionsverlaufs von v_a bis v_n . Der Verlauf der einzelnen Versionen ist in einem separaten Artefakt vermerkt. Mit Hilfe dieses Artefakts kann unmittelbar der Versionsverlauf von v_a bis v_n bestimmt werden.
2. Ausführung der Migrationsfunktionalität für alle Paare (v_i, v_{i+1}) , die Elemente dieses Versionsverlaufs sind, mit $a \leq i < n$.

Der große Vorteil an dieser Herangehensweise ist, dass Daten für beliebige Versionssprünge migriert werden können. Ein Nachteil daran ist, dass durch die schrittweise Ausführung unter Umständen unnötige Migrationen durchgeführt werden, so dass die Ausführungsgeschwindigkeit der Gesamtmigration reduziert wird. Dieses Problem tritt beispielsweise auf, wenn in einem Versionssprung in einem Modellbereich Änderungen durchgeführt wurden und dieser Modellbereich wurde in einem späteren Versionssprung entfernt oder erneut umstrukturiert. In diesem Fall ist die unmittelbare Generierung von Migrationsfunktionalität für den Übergang von Version v_a nach v_n vorteilhaft.

Bei der Datenmigration für die Client-Server-Kommunikation ist diese Herangehensweise jedoch nicht durchführbar. Der Grund dafür ist, dass Clients in beliebigen Versionen mit einem Server kommunizieren können. Dadurch ist es nicht praktikabel für alle möglichen Versionskombinationen vorab die passende Migrationsfunktionalität zu generieren. Bei der Migration von Datenbankinhalten ist ein derartiger Ansatz hingegen umsetzbar, da diese Datenmigrationen

explizit durch die Entwickler initiiert werden. Dadurch können sie vorab die passende Datenmigrationsfunktionalität generieren. Nachteilhaft hieran ist jedoch, dass die Entwickler für einen größeren Versionssprung Datenmigrationen hinzufügen müssen, die für die einzelnen Zwischenschritte nachträglich manuell hinzugefügt wurden. Aus diesen Gründen wird in dieser Arbeit eine schrittweise Anwendung der Datenmigrationen für die Versionssprünge vorgenommen.

Abbildung 6.3 illustriert die Einbettung der Migrationsfunktionalität in die Client-Server-Kommunikation. In diesem Beispiel kommunizieren zwei Clients mit unterschiedlichen Versionen v_a und v_b mit einem Server in einer neueren Version v_n . Client-seitig findet eine Serialisierung von Java-Objekten nach JSON statt. Die serialisierten JSON-Dokumente werden anschließend an den Server gesendet. Dort transformiert eine Migrationskomponente die JSON-Daten unter Beachtung der Version des Clients. Bei der Kommunikation zwischen Client v_a und dem Server v_n wird die Migrationsfunktionalität für die Paare $(v_a, v_{a+1}), (v_{a+1}, v_{a+2}), \dots, (v_{n-1}, v_n)$ nacheinander ausgeführt. Sendet der Server hingegen Daten an Client v_a , werden die Migrationen in der umgekehrten Reihenfolge ausgeführt. Dies ist möglich, da bei einem Versionssprung von v_k nach v_{k+1} nicht nur Migrationen von v_k nach v_{k+1} sondern zusätzlich auch Migrationen von v_{k+1} nach v_k generiert werden. Neben Client in Version v_a kann ein weiterer Client in Version v_b mit dem gleichen Server kommunizieren. In diesem Fall werden konform zu dem vorherigen Fall initial die Migrationen für die Paare $(v_b, v_{b+1}), (v_{b+1}, v_{b+2}), \dots, (v_{n-1}, v_n)$ nacheinander ausgeführt.

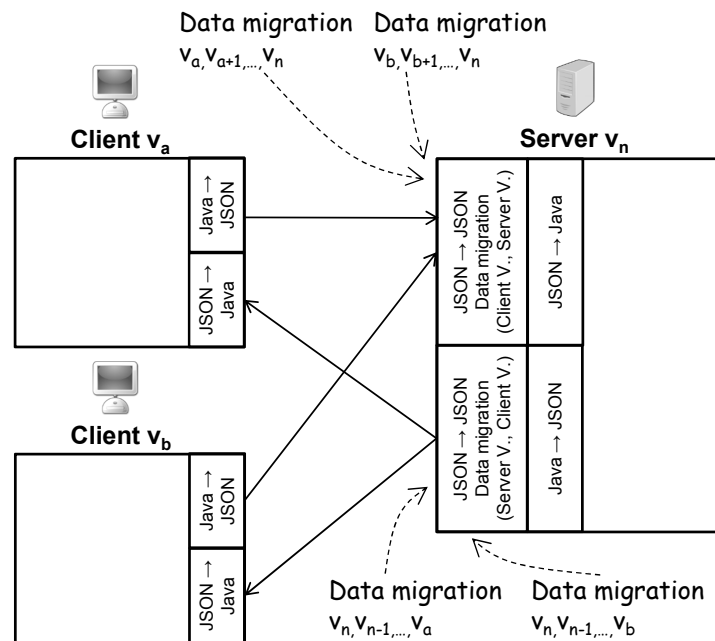


Abbildung 6.3: Verwendung der Datenmigrationsfunktionalität in der Client/Server-Kommunikation. Clients in verschiedenen Versionen können mit dem gleichen Server kommunizieren.

6.2.2 Methodik zur Generierung von Migrationsfunktionalität

Der Ansatz zur Generierung von Migrationsfunktionalität gliedert sich in vier Hauptschritte. Diese werden im Folgenden kurz beschrieben und in den weiteren Abschnitten dieses Kapitels ausführlich erläutert. Der dazugehörige Pseudocode ist in Algorithmus 6.1 auf Seite 108 dargestellt.

1. Ein Differenzberechnungswerkzeug bestimmt die Modelländerungen, die zwischen zwei Modellversionen v_a und v_n durchgeführt wurden (Zeile 1). Das Ziel der Datenmigration ist es, die zu v_a konformen Daten verlustfrei zu migrieren, so dass die resultierenden Daten konform zu v_n sind. In manchen Szenarien können die Daten allerdings nur unter Datenverlust migriert werden. Dieser Aspekt wird in Abschnitt 6.4 und Abschnitt 6.5 genauer erklärt. Die Ermittlung von Modellunterschieden wurde im Detail in Kapitel 4 beschrieben.
2. Das aus dem ersten Schritt resultierende Differenzmodell kann sämtliche Arten von Modelländerungen enthalten, die in einem Klassendiagramm vorgenommen werden können. Einige dieser Modelländerungsarten werden als Nächstes in einem Vereinfachungsschritt (Zeile 2) durch andere Modelländerungsarten ausgedrückt, z.B. durch eine Folge von Operationen zum Hinzufügen oder Löschen von Attributen. Dieser Schritt wird durchgeführt, damit die Generierung der Migrationsfunktionalität für eine kleinere Menge von Änderungsarten implementiert werden muss. Ferner werden solche Modelländerungen aus dem Differenzmodell entfernt, für die auf keinen Fall Datenmigrationen generiert werden müssen. Auf diese Weise operieren die folgenden Schritte auf einem Differenzmodell, welches ausschließlich relevante Modelländerungsarten enthält. Dieser Schritt wird genauer in Abschnitt 6.4 erklärt.
3. Für jede Modelländerung des vereinfachten Differenzmodells werden anschließend folgende Schritte ausgeführt (Zeile 3 – 6):
 - a) Bestimmung der von der Modelländerung betroffenen Klasse (Zeile 4). Da zur Modellierung der Datenmodelle Klassendiagramme eingesetzt werden, lässt sich jede Modelländerung darauf zurückführen, dass eine spezifische Klasse unmittelbar von einer Modelländerung betroffen ist. Beispielsweise ist bei einer Umbenennung eines Attributs die Klasse betroffen, die das umbenannte Attribut beinhaltet.
 - b) Ermittlung der möglichen Pfade zu der von der Modelländerung betroffenen Klasse (Zeile 5). Eine Datenmigration wird nur dann initiiert, wenn die betroffene Klasse von mindestens einer primären Klasse ausgehend über mindestens einen Pfad von Assoziationen erreichbar ist. Aus den resultierenden Pfaden kann unmittelbar abgeleitet werden, wie in einem JSON-Dokument zu den von der Änderung betroffenen JSON-Objekten navigiert werden kann. Die Ermittlung von Pfaden wird im Detail in Abschnitt 6.3 erläutert. Zu beachten ist hierbei, dass die Pfade stets Pfade darstellen, die in der alten Version v_a bestehen. Der Grund dafür ist, dass die zu migrierenden Daten gemäß Voraussetzung konform zu Version v_a sein müssen. Folglich müssen die Strukturen aus Version v_a als Ausgangsbasis genommen werden, um die Daten zu migrieren.

- c) Als Nächstes werden auf Basis der zuvor ermittelten Pfade Graphen konstruiert (Zeile 6). Diese beinhalten, analog zu den Pfaden, die Informationen, die zur Traversierung der JSON-Dokumente benötigt werden. Existierte durch die vorherige Verarbeitung bereits ein Graph zu einer primären Klasse, wird dieser Graph um die vorliegenden Pfadinformationen erweitert. Dieser Schritt wird ausgeführt, um zirkuläre Pfade besser handhaben zu können. Die allgemeine Vorgehensweise zur Konstruktion eines Graphen auf Basis von Pfaden wird in Abschnitt 6.3 beschrieben.
4. Abschließend wird für die zuvor aufgebauten Graphen die Migrationsfunktionalität generiert. Nachdem alle Modelländerungen verarbeitet wurden, wird dazu initial für jeden Graph versucht, den Graph zu minimieren (Zeile 8). Durch diese Minimierung wird verhindert, dass Migrationsoperationen unnötig generiert und ausgeführt werden. Basierend auf einem minimierten Graphen wird anschließend die Migrationsfunktionalität generiert (Zeile 9). In Abschnitt 6.5 wird allgemein erläutert, wie für welche Art der Modelländerung vorgegangen wird, um eine Datenmigration durchzuführen. Darauf aufbauend wird in Abschnitt 6.6 erklärt, wie die Migrationsfunktionalität generiert wird.

Algorithmus 6.1 : Function generateMigrationFunctionalityVersions()

```

input  :  $v_a$ : Old model version,
           $v_n$ : New model version
output : Generated migration functionality

1 diffModel  $\leftarrow$  calculateModelDiff( $v_a$ ,  $v_n$ );
2 adaptDiffModel(diffModel);
3 for each modelDiff : diffModel do
4   | affectedClass  $\leftarrow$  determineAffectedClass(modelDiff);
5   | paths  $\leftarrow$  findPathsToPrimaryClasses(affectedClass);
6   | createOrExtendGraphs(paths, modelDiff);
7 for each g : graphs do
8   | minG  $\leftarrow$  minimizeGraph(g);
9   | generateMigrationFunctionality(minG);
  
```

6.3 Ermittlung von Pfaden zu Modellelementen

6.3.1 Ermittlung von Pfaden

Die Ermittlung von Pfaden zu einer spezifischen Zielklasse bildet eine wichtige Grundlage für die Generierung der Migrationsfunktionalität. Die Unterscheidung zwischen primären und sekundären Klassen ist dabei von zentraler Bedeutung:

- Jedes JSON-Dokument enthält ausschließlich Daten zu genau einer Instanz einer primären Klasse. Dadurch können Modelländerungen, die sich direkt auf eine primäre Klasse beziehen, nur Auswirkungen auf Daten zu dieser primären Klasse haben. Die Folge davon

ist, dass keine Pfade ermittelt werden müssen, wenn die Zielklasse eine primäre Klasse darstellt.

- Um zu bestimmen, auf welche bestehenden Daten sich eine Modelländerung zu einer sekundären Klasse auswirken kann, müssen die Pfade ermittelt werden, die von primären Klassen ausgehend zu der betroffenen sekundären Klasse führen. Infolgedessen müssen die Daten zu diesen primären Klassen migriert werden.

Die Kernidee zur Ermittlung der Pfade ist, dass die jeweils eingehenden Assoziationen einer Klasse rückwärts verfolgt werden. Dieser Prozess beginnt mit einer spezifischen Zielklasse und wird wiederholt, bis:

- eine primäre Klasse erreicht wird,
- die jeweilige Klasse keine eingehenden Assoziationen besitzt oder
- die Quellklasse der eingehenden Assoziation oder eine Unter- bzw. Oberklasse davon für den aktuellen Pfad bereits traversiert wurde.

Algorithmus 6.2 zeigt den Pseudocode, der diese Kernidee umsetzt und darlegt, wie die Pfade bestimmt werden. Ein Pfad besteht dabei aus einer Liste von Pfadeinträgen und jeder Pfadeintrag verweist auf genau eine Assoziation eines Klassendiagramms. Für jede Modelländerung des vereinfachten Differenzmodells wird Algorithmus 6.2 für eine spezifische Zielklasse aufgerufen. Diese Zielklasse stellt die Klasse dar, welche von der Modelländerung direkt betroffen ist. Details dazu, mit welcher Zielklasse der Algorithmus für welche Art von Modelländerung aufgerufen wird, sind in Abschnitt 6.5 dargelegt. Als Ergebnis liefert der Algorithmus sämtliche Pfade zurück, die von primären Klassen ausgehend zu der Zielklasse führen. Sobald von einer primären Klasse ausgehend ein Pfad zu der Zielklasse existiert, können die Daten zu der entsprechenden primären Klasse von der zugehörigen Modelländerung betroffen sein. Folglich muss für jede derart bestimmte primäre Klasse eine Datenmigration durchgeführt werden.

Algorithmus 6.2 : Function findPathsToPrimaryClasses()

input : targetClass: Class to which paths are determined

output : List of paths from primary classes to class targetClass

```

1 Create list primaryClassPaths;
2 if targetClass is primary class then
3   Create special path emptyPrimaryClassPath using targetClass as start node;
4   Add path emptyPrimaryClassPath to primaryClassPaths;
5   return primaryClassPaths;
6 Create empty path path;
7 currentPaths ← traverseAssociations(targetClass, path, true);
8 findPathsToPrimaryClassesInternal(currentPaths, primaryClassPaths);
9 return primaryClassPaths;
```

Im Folgenden wird Algorithmus 6.2 genauer vorgestellt. Zu Beginn wird geprüft, ob die Zielklasse eine primäre Klasse ist (Zeile 2). In diesem Fall wird ein spezieller Pfad erzeugt (Zeile 3), der lediglich aus der primären Klasse besteht. Dieser Pfad wird darauffolgend zu der Liste der Pfade hinzugefügt (Zeile 4) und diese Liste wird zurückgegeben (Zeile 5). Diese Sonderbehandlung wird durchgeführt, damit aus den zurückgelieferten Pfaden direkt ersichtlich ist, ob eine Datenmigration durchzuführen ist.

Handelt es sich bei der Zielklasse hingegen um keine primäre Klasse (Zeile 6 – 9), werden die eingehenden Assoziationen zu der übergebenen Klasse ermittelt und es wird eine erste Liste von Pfaden erzeugt (Zeile 7). Diese Liste enthält für jede eingehende Assoziation der Zielklasse je einen Pfad, der genau aus der zugehörigen Assoziation besteht. Darauffolgend wird versucht diese Pfade weiter zu expandieren (Zeile 8). Abschließend wird die Liste mit den gefundenen Pfaden zurückgegeben (Zeile 9).

Der Pseudocode zur Expansion der Pfade ist in Algorithmus 6.3 angegeben. Basierend auf einem existierenden Pfad werden die eingehenden Assoziationen der Zielklasse traversiert (Zeile 2 – 10). Aufbauend auf einer spezifischen eingehenden Assoziation wird danach ein neuer Pfadeintrag erzeugt (Zeile 5). Der ursprünglich übergebene Pfad wird daraufhin kopiert und der neue Pfadeintrag wird vorne in den Pfad eingefügt (Zeile 6). Auf diese Weise wird der übergebene Pfad jeweils mit einer eingehenden Assoziation expandiert.

Algorithmus 6.3 : Function `traverseAssociations()`

```

input  : targetClass: Class whose incoming associations are traversed,
          currentPath: Current path which is extended by incoming associations,
          considerInheritedAssocs: Boolean flag indicating whether inherited associations
          should be taken into account

output : Potentially expanded list of paths extendedPaths

1 Create empty list extendedPaths;
2 for each incomingAssoc : getIncomingAssociations (targetClass,
3   considerInheritedAssocs) do
4   newSrcClass ← Get source class of incomingAssoc;
5   newEntry ← createPathEntry (incomingAssoc,targetClass) ;
6   tempPath ← Copy currentPath and insert newEntry at beginning;
7   if tempPath does not contain cycle then
8     | Add tempPath to extendedPaths;
9   else
10  | Add tempPath to cyclePaths;
11 if targetClass is abstract then
12  | /* Explained later...                               */
13 return extendedPaths;

```

Der expandierte Pfad wird allerdings nur dann zu der Liste der zurückzuliefernden Pfade hinzugefügt, wenn durch die Expansion des Pfads kein Zyklus entstanden ist (Zeile 7 – 8). Ein

Zyklus entsteht genau dann, wenn:

- die Quellklasse der eingehenden Assoziation,
- eine Oberklasse der Quellklasse der eingehenden Assoziation
- oder eine Unterklasse der Quellklasse der eingehenden Assoziation

zuvor bei der Traversierung des aktuell betrachteten Pfads schon besucht wurde. Für den Fall, dass durch die Erweiterung um das zuletzt erzeugte Pfadelement ein Zyklus entsteht, wird dieser Pfad zu einer Liste von zirkulären Pfaden hinzugefügt (Zeile 9 – 10). Zirkuläre Pfade werden verwaltet, um die nicht-zirkulären Pfade später bei Bedarf erweitern zu können. Dieser Aspekt wird im weiteren Verlauf dieses Abschnitts noch im Detail erläutert.

Die zuvor beschriebene Vorgehensweise zur Expansion eines Pfads wird schließlich rekursiv für die Quellklassen der eingehenden Assoziationen fortgeführt.

Der bisher vorgestellte Teil des Algorithmus zur Ermittlung von Pfaden zu einer Zielklasse wird im Folgenden anhand von zwei Beispielen illustriert. Abbildung 6.4 zeigt ein fiktives Datenmodell, in dem lediglich die Klasse A eine primäre Klasse darstellt. Ferner wird angenommen, dass die Pfade zu der Zielklasse D konstruiert werden sollen.

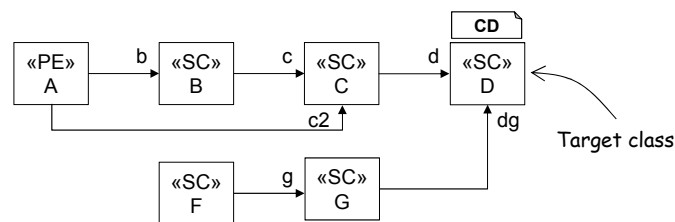


Abbildung 6.4: Beispielhaftes Klassendiagramm zur Bestimmung von Pfaden zur Klasse D.

Im ersten Schritt werden die eingehenden Assoziationen der Klasse D untersucht. Infolgedessen wird eine initiale Liste mit zwei Pfaden konstruiert: $\{\{d\}, \{dg\}\}$. Anschließend werden diese Pfade weiter expandiert. Als Erstes wird dazu der Pfad $\{d\}$ als Ausgangsbasis genommen und die eingehenden Assoziationen der Klasse C ermittelt. Die Klasse hat zwei eingehende Assoziationen und diese Assoziationen werden jeweils in eine Kopie des Pfads vorne eingefügt. Dies resultiert in einer expandierten Liste mit zwei Pfaden: $\{\{c2, d\}, \{c, d\}\}$. Der Algorithmus wird daraufhin mit diesen expandierten Pfaden erneut aufgerufen. Dabei wird festgestellt, dass die Quellklasse des Pfads $\{c2, d\}$ bereits eine primäre Klasse ist. Dieser Pfad kann also nicht weiter expandiert werden. Zur Expansion des anderen Pfads wird die eingehende Assoziation b in den Pfad kopiert, was in dem neuen Pfad $\{b, c, d\}$ resultiert. Anschließend wird auch für diesen Pfad festgestellt, dass die Quellklasse des Pfads eine primäre Klasse ist.

Ferner wird der Pfad $\{dg\}$ aus der initialen Erstellung der Pfade weiter expandiert. Dazu wird die eingehende Assoziation der Klasse G vorne in den Pfad eingefügt, so dass der Pfad $\{g, dg\}$ entsteht. Dieser Pfad kann nicht weiter expandiert werden, da die Klasse F keine weiteren eingehenden Assoziationen besitzt. Insgesamt wurden durch den Algorithmus also drei

Pfade ermittelt: $\{\{c2, d\}, \{b, c, d\}, \{g, dg\}\}$. Die ersten beiden Pfade sind für den Algorithmus relevant, da die beiden Pfade von der primären Klasse A ausgehend starten. Der dritte Pfad wird nicht weiter betrachtet, da er nicht von einer primären Klasse ausgehend startet.

Als Nächstes wird ein Beispiel zur Behandlung von zirkulären Strukturen vorgestellt. Abbildung 6.5 zeigt ein beispielhaftes Klassendiagramm, bei dem ein Zyklus zwischen den Klassen B, C und E besteht und bei dem Pfade zu der Klasse D ermittelt werden sollen.

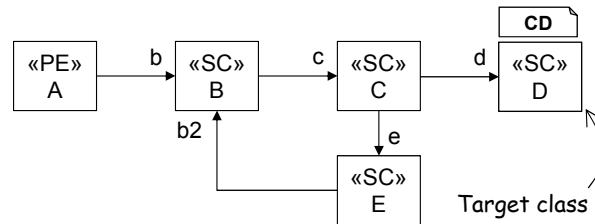


Abbildung 6.5: Beispielhaftes Klassendiagramm zur Bestimmung von Pfaden zur Klasse D - unter Beachtung von Zyklen.

Der Algorithmus erzeugt initial den Pfad $\{d\}$ und fährt danach mit der Abarbeitung der Klasse C fort. Der Pfad wird hierbei zu dem Pfad $\{c, d\}$ expandiert und der Algorithmus fährt mit der Klasse B fort. Die zwei eingehenden Assoziationen führen zu zwei expandierten Pfaden $\{\{b, c, d\}, \{b2, c, d\}\}$. Der Pfad $\{b, c, d\}$ wird allerdings nicht weiter expandiert, da die Quellklasse des Pfads bereits eine primäre Klasse ist. Der zweite Pfad wird hingegen weiter zu $\{e, b2, c, d\}$ expandiert. Durch die Hinzunahme der Assoziation e in den Pfad entsteht jedoch ein Zyklus. Der Algorithmus kann aufgrund dessen nicht nach dem bekannten Schema fortgesetzt werden, da daraus eine endlose Suche nach Pfaden resultieren würde. Stattdessen wird der zirkuläre Pfad gespeichert, wie in Algorithmus 6.3 auf Seite 110 in Zeile 10 gezeigt. Die zirkulären Pfade werden bewusst nicht verworfen, da die Kenntnis über diese Pfade für Datenmigrationen wichtig sein kann. Würden die zirkulären Pfade nicht verwaltet werden, könnte innerhalb der JSON-Dokumente nicht in die Teile der JSON-Dokumente navigiert werden, die nur über die zirkulären Pfade erreichbar sind. Somit würden Teile der JSON-Dokumente nicht migriert werden, obwohl diese migriert werden müssen. Aus diesem Grund werden die zirkulären Pfade nicht verworfen, sondern bei der Ausführung des Algorithmus separat verwaltet.

Als Nächstes wird der Teil des Traversierungsalgorithmus vorgestellt, der den Fall behandelt, dass die Pfade zu einer abstrakten Klasse expandiert werden sollen. Algorithmus 6.4 zeigt den zugehörigen Pseudocode, der den Pseudocode aus Algorithmus 6.3 auf Seite 110 erweitert.

Für jede konkrete Unterklasse der abstrakten Klasse wird die zuvor beschriebene Logik erneut ausgeführt. Das ist erforderlich, da jede dieser Unterklassen weitere eingehende Assoziationen besitzen kann, die ebenfalls analysiert werden müssen. Eine Besonderheit bei der Analyse der Unterklassen ist, dass bei dem Aufruf in Zeile 4 angegeben ist, dass die geerbten eingehenden Assoziationen nicht in die Analyse einbezogen werden sollen. Dies wird darüber ausgedrückt, dass `traverseAssociations()` mit dem Wahrheitswert `false` für `considerInheritedAssocs` aufgerufen wird. Der Grund dafür ist, dass für die abstrakte Klasse die eingehenden Assoziationen bereits betrachtet wurden und eine erneute Analyse dieser Assoziationen

Algorithmus 6.4 : Function `traverseAssociations()` - Second part

input : `targetClass`: Class whose incoming associations are traversed,
 `currentPath`: Current path which is extended by incoming associations,
 `considerInheritedAssocs`: Boolean flag indicating whether inherited associations
 should be taken into account

output : Potentially expanded list of paths `extendedPaths`

```

1 Create empty list extendedPaths;
  /* Explained before... */
2 if targetClass is abstract then
3   for each subclass : getConcreteSubclasses(targetClass) do
4     tempPath ← traverseAssociations(subclass,currentPath,false);
5     Add tempPath to extendedPaths;
6 return extendedPaths;
```

somit überflüssig ist. Andernfalls würden als Ergebnis mehrere gleiche Pfade zurückgeliefert werden.

Eine Sonderbehandlung erfahren abstrakte Klassen auch bei der Erzeugung eines Pfadeintrags, wie in Algorithmus 6.5 gezeigt.

Algorithmus 6.5 : Function `createPathEntry()`

input : `incomingAssoc`: Association for which path entry is created,
 `targetClass`: Class who owns incoming association `incomingAssoc`

output : Path entry for `incomingAssoc`

```

1 Create path entry newEntry with incomingAssoc;
  /* Holds if incomingAssoc refers to abstract class and class
   is subclass of this class. */
2 if Target class of incomingAssoc is not targetClass then
3   if targetClass is abstract then
4     Add constraints to newEntry that relevant class is one of the concrete subclasses of
5     targetClass;
6   else
7     Add constraint to newEntry that relevant class is targetClass;
8 return newEntry;
```

Jeder Pfadeintrag besteht aus einer Assoziation (Zeile 1). Darüber hinaus können zu einem Pfadeintrag Restriktionen hinterlegt sein. Diese beschreiben, für die Instanzen welcher Klasse der Pfadeintrag gilt. Die Motivation für die Verwaltung derartiger Restriktionen wird nachfolgend dargelegt. Pfade werden konstruiert, um von dem Wurzelobjekt eines JSON-Dokuments zu den Teilen des JSON-Dokuments navigieren zu können, in denen die Datenmigrationen durch-

geführt werden müssen. Verweist eine der Assoziationen eines Pfads auf eine abstrakte Klasse und darauffolgende Assoziationen des Pfads starten ausgehend von spezifischen Unterklassen, muss für die entsprechenden Pfadeinträge vermerkt sein, für welche Unterklassen der Pfadeintrag gültig ist. Diese Informationen werden in Pfadeinträgen über Restriktionen verwaltet.

Das Vermerken von Restriktionen ist allerdings nur dann erforderlich, wenn die aktuell betrachtete Assoziation auf eine Zielklasse, die nicht die aktuell betrachtete Klasse ist, verweist (Zeile 2). Dies gilt genau dann, wenn die Assoziation auf eine abstrakte Klasse zeigt und die aktuell betrachtete Klasse eine Unterklasse dieser Klasse ist. Falls die betrachtete Klasse selbst abstrakt ist, so können in JSON-Dokumenten Instanzen zu sämtlichen konkreten Unterklassen dieser Klasse weiter verfolgt werden, so dass sämtliche konkrete Unterklassen als relevante Klassen vermerkt werden (Zeile 4 – 5). Andernfalls wird vermerkt, dass die aktuell betrachtete Klasse die relevante Unterklasse ist.

Die Handhabung von Vererbungshierarchien bei der Pfadermittlung wird im Folgenden anhand von Abbildung 6.6 verdeutlicht. In diesem Beispiel wird angenommen, dass Pfade zu der Zielklasse E identifiziert werden sollen.

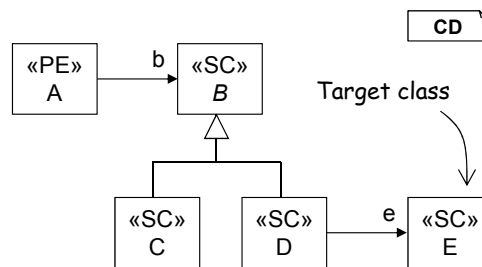


Abbildung 6.6: Beispielhaftes Klassendiagramm zur Bestimmung von Pfaden zur Klasse E unter Betrachtung von Vererbung.

Der Algorithmus erzeugt initial den Pfad $\{e\}$, da die Klasse E lediglich eine eingehende Assoziation besitzt. Der nächste Schritt besteht in der Expansion dieses Pfads unter Betrachtung der eingehenden Assoziationen der Klasse D. Die einzige eingehende Assoziation dieser Klasse ist die geerbte Assoziation b. Gemäß Algorithmus 6.5 werden Restriktionen zu einem Pfadeintrag hinzugefügt, wenn die Zielklasse einer zu untersuchenden eingehenden Assoziation ungleich der betrachteten Klasse ist (Zeile 2, Algorithmus 6.5). Diese Bedingung ist erfüllt, da die Assoziation b auf die Klasse B verweist und die Klasse D untersucht wird. Infolgedessen wird zu dem Pfadeintrag die Restriktion hinzugefügt, dass die Klasse D die relevante Unterklasse ist. Dies resultiert in dem erweiterten Pfad $\{b(\text{subclass D}), e\}$. Aus diesem Pfad kann direkt abgeleitet werden, dass bei einer Traversierung eines JSON-Dokuments die Assoziation e nur verfolgt werden darf, wenn diese von einer Instanz der Klasse D ausgeht. Für den Fall, dass die Klasse D eine abstrakte Klasse wäre und konkrete Unterklassen besitzen würde, würden Restriktionen zu diesen konkreten Unterklassen hinterlegt werden.

6.3.2 Konstruktion von Graphen zur Abbildung von Pfaden

Nach dem Aufruf von Algorithmus 6.2 wird jeder zurückgelieferte Pfad als Graph dargestellt. Die Abbildung auf Graphen erfolgt, um zirkuläre Strukturen einfacher handhaben zu können. Diese Graphen werden später zur Generierung der Migrationsfunktionalität traversiert. Die Abbildung eines Pfads in einen Graphen wird wie folgt gehandhabt:

- Der Pfad wird auf Ebene des Klassendiagramms von Anfang bis Ende verfolgt. Für jede dabei unmittelbar betroffene Klasse wird ein Knoten zu dem Graphen hinzugefügt. Als Name eines Knoten wird der vollqualifizierte Name der Klasse gesetzt. Diese Entscheidung wurde getroffen, um Verschiebungen und Umbenennungen von Paketen und Klassen handhaben zu können. Dieser Aspekt wird in Abschnitt 6.5.3 noch ausführlicher beschrieben. Standardmäßig wird zu dem Knoten, der für die Zielklasse des Pfads erzeugt wurde, zusätzlich noch die Modelländerung hinterlegt, die zu dem Aufruf des Pfadermittlungsalgorithmus geführt hat. Abweichungen von dieser Vorgehensweise werden in Abschnitt 6.5.1 explizit hervorgehoben. Die Information darüber, welche Modelländerung mit welchem Knoten verbunden ist, ist für die spätere Generierung der Migrationsfunktionalität relevant.
- Eine Kante zwischen zwei Knoten wird genau dann eingefügt, wenn zwischen den zugehörigen Klassen eine Assoziation besteht. Der Name des Assoziationsendes, die Richtung der Navigierbarkeit und die Kardinalitäten werden für die Kanten beibehalten. In den konkreten Beispielen werden in den Graphen jedoch nur die jeweils erforderlichen Informationen gezeigt.
- Die Restriktionen, die zu Pfadeinträgen hinterlegt sind, werden auf die zugehörigen Kanten des Graphen übertragen. Dadurch kann das Wissen über die zulässigen Unterklassen in die spätere Generierung der Migrationsfunktionalität einbezogen werden.
- Eine Sonderstellung bei der Umwandlung in einen Graphen nehmen Vererbungsbeziehungen ein. Angenommen eine abstrakte Klasse A ist unmittelbar Teil eines Pfads. Dann sind Unterklassen von A als Restriktionen im zugehörigen Pfadeintrag hinterlegt. Für jede dieser hinterlegten Unterklassen wird implizit ein Knoten angelegt, falls dieser nicht bereits zuvor erzeugt wurde. Darüber hinaus werden die Knoten für die Unterklassen zur Kennzeichnung der Vererbungsbeziehung über eine eingehende Kante mit dem Knoten für die Klasse A verbunden. Dies gilt allerdings nur in Fällen, in denen die Klasse A auch in der neuen Modellversion eine Oberklasse ist. Die Motivation für diese Sonderbehandlung ist, dass auf Basis der Graphen die Migrationslogik generiert wird, wie in Abschnitt 6.6 noch genauer dargelegt. Ohne die Fallunterscheidung würde für ein JSON-Objekt auch die Migrationslogik ausgeführt werden, die spezifisch für eine Oberklasse bzw. Unterklasse ist, selbst wenn die Vererbungsbeziehung in der neuen Version gar nicht mehr vorliegt.

Ein konkreter Pfad betrifft stets genau eine primäre Klasse, da jeder zurückgelieferte Pfad mit einer spezifischen primären Klasse startet. Für jede betroffene primäre Klasse wird ein solcher Graph aufgebaut. Dazu wird ausgehend von dem ersten gefundenen Pfad für eine primäre

Klasse ein Graph nach obigem Schema konstruiert. Anschließend werden ebenfalls die weiteren ermittelten Pfade für denselben Typ abgearbeitet. In diesen Fällen werden jedoch keine neuen Graphen erzeugt, sondern der bestehende Graph wird erweitert. Falls ein Knoten für eine Klasse erzeugt werden soll, für die im Graphen bereits ein Knoten erzeugt wurde, wird der bestehende Knoten beibehalten. Handelt es sich bei diesem Knoten um den Knoten für die Zielklasse des Pfads, wird der Knoten noch um die Modelländerung, die zu dem Aufruf des Pfadermittlungsalgorithmus geführt hat, erweitert. Mit einem Knoten im Graphen können dadurch mehrere Modelländerungen assoziiert sein. Analog wird in den Graphen nur dann eine Kante hinzugefügt, wenn die Kante nicht bereits im Graphen existiert.

Eine Sonderstellung nehmen die Pfade, die aufgrund der Verschiebung von Attributen und der Veränderung der Quellklasse einer Assoziation zurückgeliefert wurden, ein. Für jede derartige Modelländerung wird ein separater Graph aufgebaut. Dies ist darin begründet, dass zur Handhabung dieser Fälle eine separate Migrationsfunktionalität generiert wird. In Abschnitt 6.6.4 und Abschnitt 6.6.5 wird dieser Aspekt detaillierter erläutert.

Anschließend wird geprüft, ob die zuvor erzeugten Graphen um die Informationen aus den zirkulären Pfaden erweitert werden können. Innerhalb eines zirkulären Pfads ist die erste Assoziation des Pfads die Assoziation, die dazu geführt hat, dass ein Zyklus aufgetreten ist. Ein Graph g kann daher genau dann um die Informationen aus einem zirkulären Pfad cp erweitert werden, wenn die erste Assoziation von cp eine eingehende oder ausgehende Assoziation einer Klasse oder einer Unter- bzw. Oberklasse einer Klasse ist, für die im Graphen bereits ein Knoten erzeugt wurde. In diesem Fall wird für die erste Assoziation eines zirkulären Pfads eine Kante zu dem Graphen hinzugefügt, falls diese noch nicht existiert. Falls für die Quell- oder Zielklasse dieser Assoziation noch keine Knoten im Graphen vorliegen, so werden diese ebenfalls erzeugt. Anschließend wird analog für die weiteren Pfadeinträge eines zirkulären Pfads verfahren.

Nachfolgend werden die Graphen zu zwei zuvor eingeführten Beispielen angegeben. Abbildung 6.7 zeigt den Graphen der zu dem Beispiel in Abbildung 6.6 auf Seite 114 erzeugt wurde. Dabei wird angenommen, dass die Modelländerung, die zu der Pfadermittlung geführt hat, das Umbenennen des Attributs `name` der Klasse `E` in `newName` war. Das Hauptaugenmerk liegt in diesem Beispiel auf der Kante zwischen den Knoten `B` und `D`. Diese Kante ist als Vererbungskante markiert. Abgesehen davon ist der Aufbau dieses Graphen analog zu dem Aufbau des relevanten Teils des zugehörigen Klassendiagramms. Hervorzuheben ist in diesem Beispiel, dass der Graph keinen Knoten für die Klasse `C` enthält. Der Grund dafür ist, dass die Unterklasse `C` nicht als Restriktion zu dem Pfadeintrag für die Assoziation `b` hinterlegt ist.

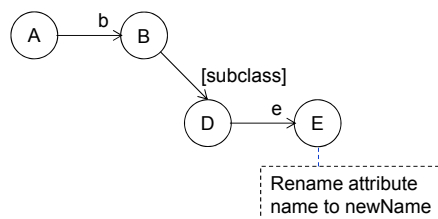


Abbildung 6.7: Graph der zu den Pfaden aus Abbildung 6.6 erzeugt wurde.

Abbildung 6.8 zeigt überdies den Graphen zu dem Pfad $\{b, c, d\}$ aus Abbildung 6.5 auf Seite 112. In diesem Fall sei angenommen, dass die initiiierende Modelländerung die Umbenennung der Klasse D war. Aus diesem Grund ist diese Modelländerung mit dem Knoten D verbunden. Zusätzlich zu diesem Pfad wurde für Abbildung 6.5 der zirkuläre Pfad $\{e, b2, c, d\}$ identifiziert. Dieser zirkuläre Pfad ist für den Pfad $\{b, c, d\}$ relevant, da die erste Assoziation des zirkulären Pfads, die Assoziation e , eine ausgehende Assoziation der Klasse C ist, die Teil des Pfads $\{b, c, d\}$ ist. Folglich kann der in Abbildung 6.8 gezeigte Graph um die Knoten und Kanten des zirkulären Pfads erweitert werden. Dies führt in dem Beispiel dazu, dass der Knoten E sowie die beiden Kanten e und $b2$ zum Graphen hinzugefügt werden. Die Graph-Elemente der weiteren Teile des zirkulären Pfads existierten schon durch die ursprüngliche Erzeugung des initialen Graphen und müssen daher nicht zum Graphen hinzugefügt werden. Abbildung 6.9 zeigt den Graphen, der aus der Erweiterung des Graphen um den zirkulären Pfad resultiert.

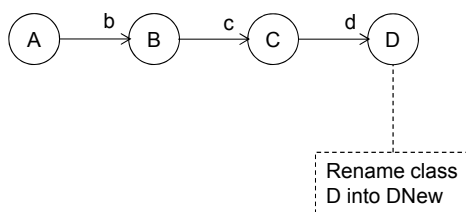


Abbildung 6.8: Klassendiagramm für ein Bibliothekssystem (ältere Version a).

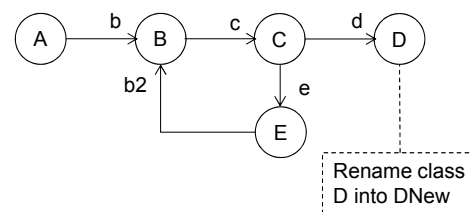


Abbildung 6.9: Klassendiagramm für ein Bibliothekssystem (neuere Version n).

6.4 Vereinfachung des Differenzmodells

Bevor das Differenzmodell an den Generator übergeben wird, erfolgt eine Vereinfachung des Differenzmodells. Dieser Vereinfachungsschritt beinhaltet unter anderem, dass spezifische Arten von Modelländerungen auf andere Arten von Modelländerungen abgebildet werden. Die Motivation hinter diesem Schritt ist, dass die Generierung der Migrationslogik dadurch vereinfacht wird, da sie eine kleinere Menge von Modelländerungsarten beachten muss. Dieser Abschnitt erklärt, in welchen Konstellationen diese Rückführung auf andere Modelländerungsarten vollzogen wird. Darüber hinaus werden einige Modelländerungsarten aus dem Differenzmodell entfernt, da für sie keine Datenmigrationen generiert werden müssen. Dadurch kann das Differenzmodell weiter vereinfacht werden.

Obgleich es eines der primären Ziele bei der Durchführung von Datenmigrationen ist, bestehende Daten beizubehalten, ist dies nicht immer möglich. Das äußert sich in dem Vereinfachungsschritt in der Weise, dass manche Arten von Modelländerungen auf das Löschen von Attributen oder Assoziationen zurückgeführt werden. Das führt bei der Generierung der Migrationen auf Basis des vereinfachten Differenzmodells dazu, dass bestehende Daten gelöscht werden. Eine derartige Anpassung des Differenzmodells wird allerdings nur in zwei Fällen vorgenommen:

- Durch die Modelländerung wird ausgedrückt, dass in einem bestimmten Kontext nur noch ein eingeschränkter Teil der bestehenden Daten zulässig ist. Die anderen Daten müssen im Zuge der Datenmigration gelöscht werden, da die Daten andernfalls in der neuen Version nicht mehr zulässig sind.
- Die Modelländerung entspricht dem Löschen eines bestehenden Modellelements und dem Hinzufügen eines neuen Modellelements und es ist nicht möglich automatisch abzuleiten, ob und wie bestehende Daten in die neue Version überführt werden sollen. Damit die migrierten Daten konform zu der neuen Version der Datenmodelle sind, müssen die Daten daher gelöscht werden.

Die Entwickler werden in beiden Fällen darüber informiert, wenn Modelländerungen auf löschende Modelländerungen abgebildet werden. Sie können daraufhin prüfen, ob die erkannten Modelländerungen tatsächlich korrekt sind. Im Zuge dessen können sie Modelländerungen, die fehlerhaft identifiziert wurden, korrigieren. Dieser Sachverhalt wurde zuvor bereits ausführlich in Abschnitt 4.2.3 erläutert. Darüber hinaus können Entwickler Funktionalität implementieren, worüber bestehende Daten beibehalten werden können. Dazu müssen sie allerdings die generierte Migrationslogik manuell adaptieren. Dieser Aspekt wird genauer in Abschnitt 6.6.6 betrachtet.

6.4.1 Verschiebung eines Attributs

In diesem Abschnitt bezieht sich der Begriff der Quellklasse auf die Klasse, die das verschobene Attribut ursprünglich beinhaltet hat, während die Zielklasse auf die Klasse verweist, in die das Attribut verschoben wurde. Bei dieser Modelländerung müssen verschiedene Fälle betrachtet werden:

- Die Zielklasse ist eine Oberklasse der Quellklasse. Diese Konstellation wird für die Unterklassen der Zielklasse auf das Hinzufügen eines neuen Attributs zurückgeführt. Ausgenommen davon ist lediglich die ursprüngliche Quellklasse. In Abbildung 6.10 liegt diese Situation vor. Dieser Fall wird für die Unterklassen C und D wie das Hinzufügen des Attributs x behandelt.

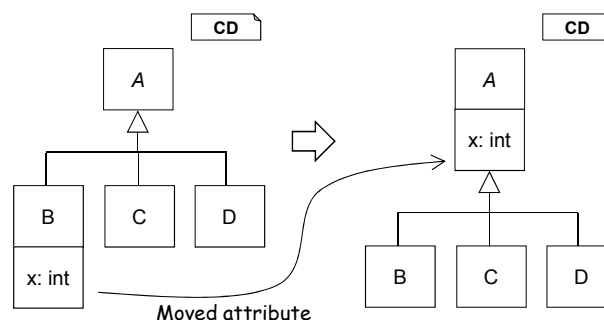


Abbildung 6.10: Verschiebung eines Attributs in eine Oberklasse der Quellklasse.

- Die Zielklasse ist eine Unterklasse der Quellklasse. In diesem Fall wird das Attribut nicht mehr an alle Unterklassen der ursprünglichen Quellklasse vererbt, sondern liegt nur noch in der neuen Zielklasse vor. Demzufolge darf in den Daten zu den anderen Unterklassen kein Wert mehr zu dem Attribut eingetragen sein. Aus diesem Grund wird dieser Fall für die Unterklassen der ursprünglichen Quellklasse auf das Löschen eines Attributs zurückgeführt. Ausgenommen davon ist lediglich die neue Zielklasse. Auf diese hat die Modelländerung keine Auswirkungen und die Daten hierzu können beibehalten werden. In Abbildung 6.10 liegt diese Situation vor, wenn das linke Klassendiagramm die neue und das rechte Klassendiagramm die alte Modellversion darstellt. Für die Klassen C und D wird somit das Löschen des Attributs x vermerkt.
- Von der Quellklasse kann über Assoziationen nicht zu der Zielklasse und von der Zielklasse nicht zu der Quellklasse navigiert werden. Die Quellklasse und die Zielklasse sind somit unabhängig voneinander. Die Folge davon ist, dass bei der Datenmigration nicht automatisch abgeleitet werden kann, aus welcher Instanz der Quellklasse der jeweilige Wert des verschobenen Attributs in welche Instanz der Zielklasse verschoben werden soll. Somit besteht keine Möglichkeit automatisch festzulegen, ob und auf welche Weise die bestehenden Daten zu dem verschobenen Attribut beibehalten werden sollen. Die einzige Möglichkeit um dieses Szenario automatisch zu behandeln, besteht darin, die Modelländerung auf das Löschen des verschobenen Attributs aus der Quellklasse und das Hinzufügen des verschobenen Attributs zu der Zielklasse zurückzuführen. Genau dieser Ansatz wird in dieser Arbeit verfolgt. In Abbildung 6.11 sind die Klassen A und B unabhängig voneinander. Somit wird die Verschiebung hier durch das Löschen des Attributs aus Klasse A und das Hinzufügen des Attributs zu Klasse B ausgedrückt.

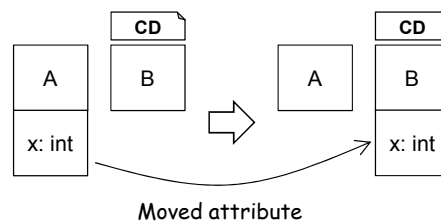


Abbildung 6.11: Verschiebung eines Attributs in eine Zielklasse, die unabhängig von der Quellklasse ist.

Liegt keiner der zuvor beschriebenen Fälle vor, kann für die Modelländerung eine Datenmigration vorgenommen werden, in der die bestehenden Daten beibehalten werden können. Details dazu sind in Abschnitt 6.5.1 erklärt.

6.4.2 Veränderung der Quellklasse einer Assoziation

Bei der Veränderung der Quellklasse einer Assoziation müssen folgende Fälle unterschieden werden:

- Die neue Quellklasse ist eine Oberklasse der alten Quellklasse. In diesem Fall erbt in der neuen Modellversion die alte Quellklasse die betroffene Assoziation, so dass diese Modelländerung keine Auswirkungen auf die alte Quellklasse hat. Stattdessen wird diese Konstellation darauf zurückgeführt, dass eine Assoziation zu den Unterklassen der neuen Quellklasse hinzugefügt wurde. Ausgenommen hiervon ist lediglich die ursprüngliche Quellklasse. In Abbildung 6.12 liegt diese Situation vor. Dieser Fall wird für die Unterklasse C wie das Hinzufügen der Assoziation hin zur Klasse E behandelt.

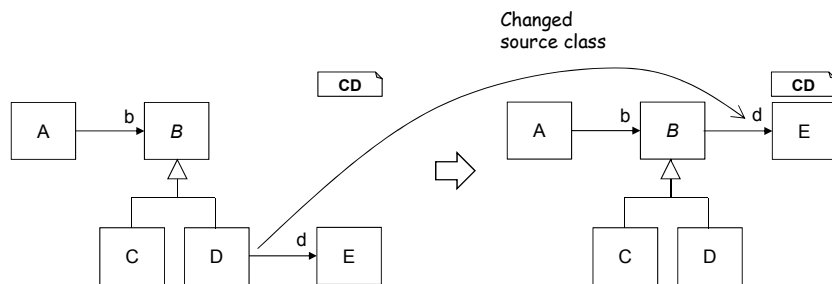


Abbildung 6.12: Änderung der Quellklasse einer Assoziation, so dass die neue Quellklasse eine Oberklasse der alten Quellklasse ist.

- Die neue Quellklasse ist eine Unterklasse der alten Quellklasse. Die anderen Unterklassen der alten Quellklasse erben somit die betroffene Assoziation in der neuen Version nicht mehr. Die Folge davon ist, dass bei der Datenmigration die Daten, die zur neuen Quellklasse hinterlegt sind, beibehalten werden können. Demgegenüber sind die Daten, die zu den anderen Unterklassen der ursprünglichen Quellklasse vorgelegen haben, nicht mehr zulässig, so dass diese gelöscht werden müssen. Diese Situation wird daher darauf zurückgeführt, dass eine Assoziation aus den Unterklassen der alten Quellklasse entfernt wurde, mit Ausnahme der neuen Quellklasse. In Abbildung 6.12 liegt diese Situation vor, wenn das linke Klassendiagramm als neue und das rechte Klassendiagramm als alte Modellversion angenommen wird. Folglich wird in diesem Fall die Modelländerung für die Unterklasse C wie das Löschen der Assoziation zur Klasse E behandelt.
- Liegt keiner der beiden zuvor genannten Fälle vor und von der alten kann nicht zur neuen und von der neuen nicht zur alten Quellklasse navigiert werden, stehen die alte und neue Quellklasse in keiner Beziehung zueinander. Aus der Modelländerung folgt in diesem Fall, dass die Assoziation von der alten Quellklasse ausgehend nicht mehr existiert, dafür aber eine neue Assoziation von der neuen Quellklasse ausgehend. Da die Daten zu der alten Quellklasse in keiner Beziehung zu der neuen Quellklasse stehen, können die existierenden Daten zu der Assoziation nicht beibehalten werden. Folglich wird die Modelländerung auf das Löschen einer Assoziation aus der alten Quellklasse und das Hinzufügen einer Assoziation zu der neuen Quellklasse zurückgeführt. In Abbildung 6.13 wird die Modelländerung für die Klasse B folglich wie das Löschen der Assoziation zur Klasse C und für die Klasse D wie das Hinzufügen der Assoziation zu dieser Klasse behandelt.

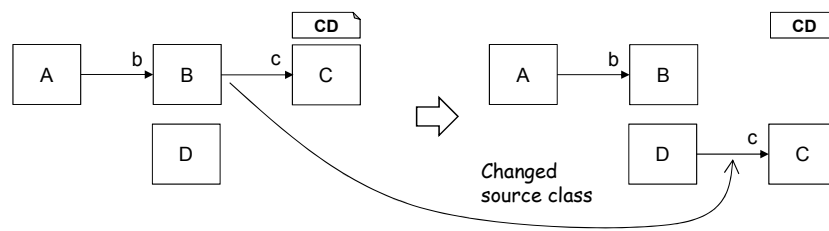


Abbildung 6.13: Änderung der Quellklasse einer Assoziation, so dass die neue Quellklasse in keiner Beziehung zu der alten Quellklasse steht.

Ist keine der drei zuvor genannten Bedingungen erfüllt, kann von der alten zur neuen Quellklasse oder von der neuen zur alten Quellklasse navigiert werden. Die Durchführung einer Datenmigration in diesem Szenario wird in Abschnitt 6.5.1 erklärt.

6.4.3 Veränderung der Zielklasse einer Assoziation

Im Falle einer Veränderung der Zielklasse einer Assoziation muss ebenfalls eine Fallunterscheidung durchgeführt werden:

- Ist die neue Zielklasse eine Oberklasse der bisherigen Zielklasse, können die bestehenden Daten beibehalten und es muss keine Datenmigration durchgeführt werden. Dies ist darin begründet, dass bei den bestehenden Daten zu dem Assoziationsende Instanzen der bisherigen Zielklasse hinterlegt sein können und diese konform zu der neuen Zielklasse sind. Folglich kann die Modelländerung in dieser Konstellation aus dem Differenzmodell entfernt werden. Abbildung 6.14 verdeutlicht dieses Szenario. Hierbei können die Daten, die zu der Assoziation mit dem Assoziationsende *c* hinterlegt sind, beibehalten werden.

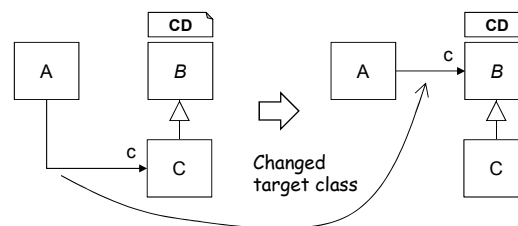


Abbildung 6.14: Änderung der Zielklasse einer Assoziation, so dass die neue Zielklasse eine Oberklasse der alten Zielklasse ist.

- Ist die neue Zielklasse der Assoziation weder eine Oberklasse noch eine Unterklasse der bisherigen Zielklasse, sind sämtliche zu der bisherigen Assoziation hinterlegten Daten nicht mehr zulässig. Der Grund hierfür ist, dass die alte und neue Zielklasse unabhängig voneinander sind und daher nicht automatisch abgeleitet werden kann, wie die bestehenden Daten zu der alten Zielklasse in die neue Version überführt werden können. Stattdes-

sen können in der neuen Version neue Daten zu der Assoziation zu der neuen Zielklasse hinterlegt sein. Aus diesem Grund wird die Modelländerung für die Quellklasse der Assoziation auf das Löschen einer Assoziation zu der alten Zielklasse und das Hinzufügen einer Assoziation zu der neuen Zielklasse zurückgeführt. Liegt die Situation wie in Abbildung 6.15 gezeigt vor, wird die Änderung demzufolge für die Klasse A wie das Löschen der Assoziation zu der Klasse B und das Hinzufügen der Assoziation zu der Klasse C behandelt.

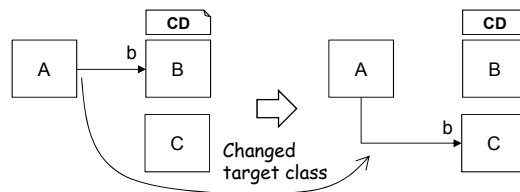


Abbildung 6.15: Änderung der Zielklasse einer Assoziation, so dass die neue Zielklasse keine Oberklasse oder Unterklasse der alten Zielklasse ist.

- Für den Fall, dass die neue Zielklasse der Assoziation eine Unterklasse der bisherigen Zielklasse ist, wird die Modelländerung unverändert beibehalten und nicht aus dem Differenzmodell entfernt. Diese Konstellation wird in Abschnitt 6.5.1 erläutert.

6.4.4 Verschiebung eines Literals aus einer Enumeration

Die Verschiebung eines Literals aus einer Enumeration in eine andere Enumeration führt dazu, dass in den Daten zu den Attributen, welche auf die betroffene Enumeration verweisen, nicht mehr das verschobene Literal als Wert hinterlegt sein darf. Folglich kann die Verschiebung auf das Löschen eines Literals aus der ursprünglichen Enumeration zurückgeführt werden. Die Tatsache, dass durch diese Art der Modelländerung das verschobene Literal zu einer anderen Enumeration hinzugefügt wird, hat keine Auswirkungen auf die Datenmigration. Dies wird genauer in Abschnitt 6.4.10 erläutert. In Abbildung 6.16 wird die Modelländerung somit wie das Löschen des Literals D aus der Enumeration B interpretiert.

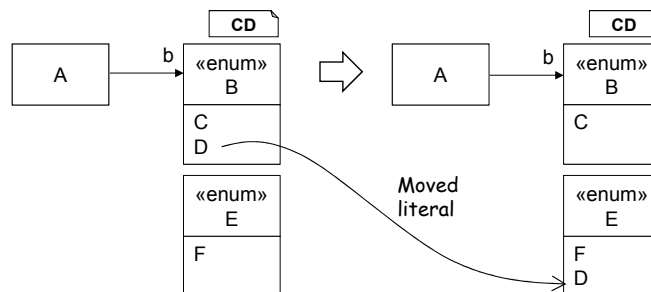


Abbildung 6.16: Verschiebung eines Literals aus einer Enumeration.

6.4.5 Verschiebung eines Pakets oder einer Klasse

Die Verschiebung von Paketen oder Klassen wird im Rahmen der Datenmigration durch die Modelländerung ersetzt, welche eine Umbenennung von Klassen ausdrückt. Die Motivation hierfür ist, dass sich eine Verschiebung von Paketen oder Klassen, analog zu der Umbenennung einer Klasse, lediglich auf den vollqualifizierten Klassennamen in JSON-Dokumenten auswirkt.

6.4.6 Umbenennung eines Pakets

Die Umbenennung von Paketen wird im Rahmen der Datenmigration, analog zu der Verschiebung von Paketen, durch die Modelländerung ersetzt, welche eine Umbenennung der im Paket enthaltenen Klassen ausdrückt. Der Grund hierfür ist, dass sich eine Umbenennung von Paketen für die darin enthaltenen Klassen lediglich auf deren vollqualifizierten Klassennamen in JSON-Dokumenten auswirkt.

6.4.7 Löschen eines Pakets

Das Löschen eines Pakets wird auf das Löschen der im Paket enthaltenen Klassen, Enumerationen und Assoziationen zurückgeführt. Diese Operation ist zulässig, da zusätzlich zu den Migrationsschritten, die bei dem Löschen dieser Klassen, Enumerationen und Assoziationen durchgeführt werden müssen, keine weiteren Migrationsschritte zum Löschen eines Pakets notwendig sind.

6.4.8 Veränderung einer Klasse hin zu einer (nicht) abstrakten Klasse

In dieser Arbeit wird vorausgesetzt, dass lediglich eine abstrakte Klasse Unterklassen besitzen kann. Diese Annahme ist durch das Projekt motiviert, in dessen Kontext diese Arbeit durchgeführt wurde. Am Ende dieses Abschnitts wird zusätzlich dargelegt, wie mit dieser Art der Modelländerung verfahren wird, wenn diese Annahme nicht gilt.

Aufgrund der zuvor genannten Annahme kann eine abstrakte Klasse X nur in zwei Fällen zu einer nicht abstrakten Klasse geändert werden:

1. die Unterklassen von X wurden entfernt bzw. die Vererbungsbeziehungen zu X wurden entfernt oder
2. X hatte keine Unterklassen

Andernfalls würde die resultierende nicht abstrakte Klasse X Unterklassen besitzen und dies ist gemäß Voraussetzung nicht erlaubt.

In dem erstgenannten Fall wird die Modelländerung beibehalten und nicht aus dem Differenzmodell entfernt oder auf andere Modelländerungen zurückgeführt. In Abschnitt 6.5.3 wird beschrieben, wie in diesem Fall Datenmigrationen durchgeführt werden.

Im zweitgenannten Fall können zu einer Assoziation, die auf X verweist, keine Daten hinterlegt gewesen sein, da zu X keine Unterklassen existierten. In der neuen Modellversion können

hingegen Daten zu der Klasse X vermerkt werden. Folglich wird die Modelländerung für sämtliche Klassen, die Assoziationen zu der Klasse X besitzen, auf das Hinzufügen einer Assoziation hin zu der nun nicht abstrakten Klasse X zurückgeführt.

Der Fall, dass eine nicht abstrakte Klasse X zu einer abstrakten Klasse verändert wird, ist gemäß Voraussetzung erlaubt. Eine derartige Modelländerung macht jedoch in der Regel nur dann Sinn, wenn zusätzlich konkrete Unterklassen zu X hinzugefügt wurden. Verweist eine Klasse A über eine Assoziation auf die Klasse X , können über diese Assoziation bereits Daten zu X hinterlegt sein. In den JSON-Objekten zu diesen bestehenden Daten ist vermerkt, dass die Daten zu dem Typ X erzeugt wurden. In der neuen Modellversion sind diese Daten dadurch nicht mehr gültig, da in der neuen Modellversion keine Daten unmittelbar zu der abstrakten Klasse X vorliegen können. Folglich müssen die Daten im Zuge der Datenmigration gelöscht werden. Aus diesem Grund wird dieser Fall durch das Löschen einer Assoziation ausgedrückt. Eine Ausnahme hiervon bildet die Konstellation, dass die in der neuen Version abstrakte Klasse genau eine Unterklasse besitzt. In diesem Szenario kann eine Datenmigration unter Beibehaltung der Daten durchgeführt werden. In Abschnitt 6.5.3 wird genau dieses Szenario erläutert.

Abschließend wird der Fall betrachtet, dass die Annahme, dass nur abstrakte Klassen Unterklassen besitzen können, nicht gilt. Als Erstes wird nachfolgend davon ausgegangen, dass eine abstrakte Klasse X zu einer nicht abstrakten Klasse verändert wurde. Falls X keine Unterklassen hatte, unterscheidet sich die Vorgehensweise hierbei nicht von der zuvor beschriebenen Vorgehensweise. Demzufolge wird die Modelländerung für sämtliche Klassen, die Assoziationen zu X besitzen, auf das Hinzufügen einer Assoziation hin zu der nun nicht mehr abstrakten Klasse zurückgeführt. Besitzt X hingegen Unterklassen, hat die Modelländerung keine Auswirkungen auf eine Datenmigration. Verweist eine Klasse nämlich auf X , können zu der entsprechenden Assoziation bereits Daten hinterlegt sein, die dadurch nicht verändert werden müssen.

Eine Änderung einer nicht abstrakten Klasse X hin zu einer abstrakten Klasse kann hingegen Auswirkungen auf die bestehenden Daten haben. Zu sämtlichen Assoziationen, die auf X verweisen, dürfen fortan keine Daten mehr direkt zu X eingetragen sein. In einer Datenmigration muss folglich über die Instanzen, die zu den betroffenen Assoziationen hinterlegt sind, iteriert werden und die Instanzen von X müssen entfernt werden. Ein Spezialfall liegt vor, wenn die abermals nicht abstrakte Klasse X genau eine Unterklasse besitzt. Die Herangehensweise in diesem Szenario wird wiederum in Abschnitt 6.5.3 beschrieben.

6.4.9 Hinzufügen einer Vererbungsbeziehung

Zur Vereinfachung der Beschreibung wird im Folgenden davon ausgegangen, dass die Klasse C die Unterklasse und die Klasse P die Oberklasse ist. Das Hinzufügen einer Vererbungsbeziehung wirkt sich in der Art aus, dass die Unterklasse C , sowie alle Unterklassen dieser Klasse, in der neuen Version ebenfalls über die Attribute und Assoziationen der Klasse P und deren Oberklassen verfügen. Folglich wird die betroffene Modelländerung durch Modelländerungen ersetzt, welche das Hinzufügen der Attribute und Assoziationen der neuen Oberklasse P und deren Oberklassen zu der Klasse C bzw. zu deren konkreten Unterklassen ausdrücken. Diese Situation wird in Abbildung 6.17 illustriert. Hier wird die Modelländerung auf das Hinzufügen des Attributs x zurückgeführt, da dieses in der neuen Oberklasse enthalten ist. Zu beachten ist,

dass wenn ein Attribut von C oder einer konkreten Unterklasse in die neue Oberklasse oder eine Oberklasse davon verschoben wurde, dieses Attribut nicht als neu hinzugefügt erachtet wird.

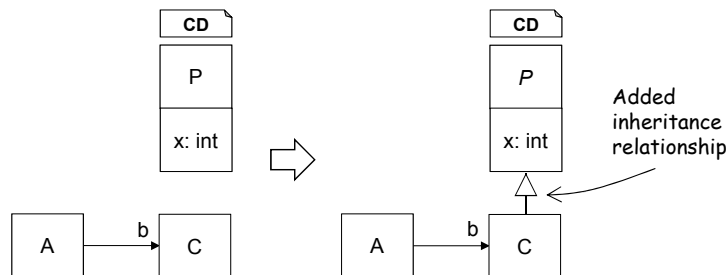


Abbildung 6.17: Hinzufügen einer Vererbungsbeziehung.

Wie mit dem Löschen einer Vererbungsbeziehung verfahren wird, ist abhängig von dem Kontext der Modelländerung und kann nicht allgemeingültig im Differenzmodell angepasst werden. Die Handhabung dieser Modelländerung wird detailliert in Abschnitt 6.5.3 erklärt.

6.4.10 Nicht betrachtete Modelländerungen

Nicht jede Modelländerung erfordert die Durchführung einer Datenmigration. Modelländerungen diesen Typs sind nachfolgend aufgelistet, mit einer kurzen Erklärung dazu, warum keine Datenmigration durchgeführt werden muss. Derartige Modelländerungen werden in dem Vereinfachungsschritt aus dem Differenzmodell entfernt:

- Hinzufügen eines Pakets oder einer Klasse: Diese Art von Modelländerung kann nur dann eine Datenmigration nötig machen, wenn zusätzlich zu dem Paket oder der Klasse mindestens eine Assoziation hinzugefügt wurde. Andernfalls können die Klassen des neu hinzugefügten Pakets bzw. die neu hinzugefügte Klasse von keiner bestehenden primären Klasse aus erreichbar sein. Aus diesem Grund wird eine Modelländerung dieser Art nicht explizit betrachtet, da die zugehörige Datenmigration bereits für das Hinzufügen der neuen Assoziation durchgeführt wird.
- Hinzufügen einer Enumeration: Analog zu dem Hinzufügen eines Pakets oder einer Klasse wird auch diese Modelländerung aus analogen Gründen nicht beachtet: Wurde zusätzlich zu der Enumeration kein Attribut hinzugefügt, welches auf diese Enumeration verweist, ist diese Änderung für die Datenmigration irrelevant. Andernfalls wird die Migration bereits über das Hinzufügen des Attributs korrekt behandelt.
- Löschen einer Enumeration: Diese Modelländerung wird aus den gleichen Gründen wie das Hinzufügen einer Enumeration nicht betrachtet: Wurde eine Enumeration entfernt, besitzen die Attribute, die auf die Enumeration verwiesen haben, keinen Datentyp mehr. Das resultierende Klassendiagramm wäre dadurch ungültig. Folglich müssen zusätzlich die Attribute entfernt worden sein, welche auf die gelöschte Enumeration verwiesen haben. Für diese Löschoperationen wird die Datenmigration bereits korrekt durchgeführt.

Daher muss das Löschen einer Enumeration zusätzlich nicht auch noch betrachtet werden.

- Umbenennung einer Enumeration: Die Werte, die zu einem Attribut hinterlegt sind, welches auf eine Enumeration verweist, enthalten keinen `@class` Schlüssel. Folglich kann die Umbenennung einer Enumeration keinen Einfluss auf bestehende Daten haben, so dass diese Modelländerung nicht behandelt wird.
- Verschiebung einer Enumeration: Diese Modelländerung wird aus den gleichen Gründen wie die Umbenennung einer Enumeration nicht betrachtet: Die Verschiebung wirkt sich lediglich auf den vollqualifizierten Namen einer Enumeration aus und dies hat keinen Einfluss auf die bestehenden Daten.
- Hinzufügen eines Literals zu einer Enumeration: Durch das Hinzufügen eines neuen Literals zu einer bestehenden Enumeration müssen die bestehenden Daten nicht angepasst werden. Aus diesem Grund wird diese Art der Modelländerung für die Datenmigrationen nicht beachtet.

Für die anderen Modelländerungen, die in diesem Abschnitt nicht betrachtet wurden, wird in Abschnitt 6.5 dargelegt, wie sie mit Hinblick auf die Generierung der Migrationsfunktionalität behandelt werden.

6.5 Verarbeitung von Modelländerungen

Die Vorgehensweise zur Durchführung einer Datenmigration ist abhängig davon, welche Arten von Modelländerungen vorgenommen wurden. Die Schritte, die erforderlich sind, um für eine Modelländerung eine passende Datenmigration durchzuführen, werden in diesem Abschnitt vorgestellt.

Eine Modelländerung erfordert jedoch nicht zwangsläufig eine Datenmigration. Um festzustellen, ob für eine konkrete Modelländerung des vereinfachten Differenzmodells eine Datenmigration erforderlich ist, wird Algorithmus 6.2 auf Seite 109 für die Klasse aufgerufen, die von der jeweiligen Modelländerung betroffen ist. Ergibt dieser Aufruf des Algorithmus, dass die betroffene Klasse von keiner primären Klasse ausgehend erreichbar ist, besteht keine Notwendigkeit zur Durchführung einer Datenmigration. Andernfalls werden für die primären Klassen, von denen ausgehend Pfade zu der betroffenen Klasse gefunden wurden, Graphen erzeugt, wie zuvor in Abschnitt 6.3.2 erklärt. Dabei wird in den Graphen standardmäßig dem Knoten, der die von der Modelländerung betroffene Klasse repräsentiert, die jeweilige Modelländerung zugeordnet. Abweichungen von diesem Standardfall werden in diesem Abschnitt explizit hervorgehoben. Die Zuordnung einer Modelländerung zu einem Knoten ist wichtig, da für jeden Knoten, dem eine Modelländerung zugeordnet ist, Datenmigrationsfunktionalität generiert wird. Dies wird im Detail in Abschnitt 6.6 erläutert. Welchem Knoten eine Modelländerung zugeordnet ist, beeinflusst folglich, ausgehend von welchen JSON-Objekten Datenmigrationen für die verschiedenen Modelländerungen initiiert werden.

6.5.1 Handhabung von Modelländerungen zu Attributen oder Assoziationen

Hinzufügen eines Attributs oder einer Assoziation

Das Hinzufügen eines Attributs oder einer Assoziation kann auf zwei Weisen gehandhabt werden: 1. die bestehenden JSON-Dokumente werden nicht migriert oder 2. die JSON-Dokumente werden migriert, indem ein Wert für das neue Attribut oder eine Instanz der Zielklasse zur neuen Assoziation eingetragen wird.

Die erste Option ist nur dann praktikabel, wenn ein Serialisierungsframework eingesetzt wird, welches bei der Deserialisierung nicht erfordert, dass zu allen Attributen und Assoziationen einer Klasse Werte hinterlegt sind. Diese Bedingung wird durch das in dieser Arbeit eingesetzte Framework Jackson erfüllt. Falls Jackson feststellt, dass ein Wert zu einem Attribut oder einer Assoziation fehlt, wird das Attribut bzw. die Assoziation mit dem Standardwert aus Java initialisiert.

Soll stattdessen die zweite Variante angewendet werden, muss zunächst geprüft werden, ob eine Datenmigration überhaupt notwendig ist. Dazu erfolgt der Aufruf von Algorithmus 6.2 mit der Klasse, welche das hinzugefügte Attribut enthält bzw. mit der Klasse, welche die Quellklasse der hinzugefügten Assoziation darstellt. Ergibt dieser Aufruf, dass eine Datenmigration erforderlich ist, stellt sich die Frage, welche Werte zu dem hinzugefügten Attribut bzw. zu der hinzugefügten Assoziation angegeben werden sollen. Abgesehen von dem Standardwert für den Datentyp kann automatisch kein sinnvoller Wert angenommen werden. Alternativ besteht die Möglichkeit, dass die Entwickler Initialwerte angeben.

Bei dem Hinzufügen einer Assoziation gilt es zudem zu beachten, dass in der zweiten Variante nur dann Daten migriert werden müssen, wenn die untere Multiplizität der hinzugefügten Assoziation ausdrückt, dass die Assoziation auf mindestens ein Element verweist. In den anderen Fällen müssen gemäß Datenmodell keine Daten zu der hinzugefügten Assoziation vorliegen.

Zur Durchführung einer Datenmigration müssen folgende Schritte vorgenommen werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Klasse repräsentieren, zu der das Attribut bzw. die Assoziation hinzugefügt wurde.
2. Ermittlung des Namens `$addedKey` des hinzugefügten Attributs bzw. des Assoziationsendes der hinzugefügten Assoziation.
3. Hinzufügen des Schlüssels `$addedKey` zu dem aktuellen JSON-Objekt.
4. Initialisierung des Schlüssels `$addedKey` mit einem spezifischen Wert.

Ein konkretes Beispiel für diese Datenmigration ist in Abbildung 6.18 angegeben. Das Beispiel zeigt den Fall, dass ein Attribut `c` zu einer Klasse `A` hinzugefügt wurde. Das im unteren Teil von Abbildung 6.18 gezeigte JSON-Dokument wird in diesem Fall explizit um einen Schlüssel `c` erweitert. Als Wert wird in diesem Beispiel die Zeichenkette „efg“ eingetragen.

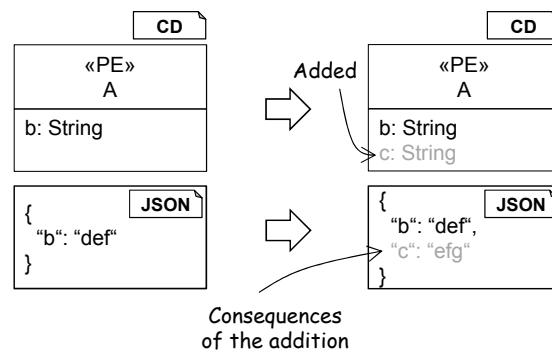


Abbildung 6.18: Datenmigration nach dem Hinzufügen eines Attributs.

Löschen eines Attributs oder einer Assoziation

Das Löschen eines Attributs oder einer Assoziation kann analog zu dem Hinzufügen eines Attributs oder einer Assoziation auf zwei Arten gehandhabt werden: 1. die bestehenden JSON-Dokumente werden nicht migriert oder 2. die JSON-Dokumente werden migriert, indem die Schlüssel zum gelöschten Attribut bzw. zur gelöschten Assoziation aus den JSON-Dokumenten entfernt werden.

Die erste Variante ist nur dann anwendbar, wenn ein Serialisierungsframework eingesetzt wird, welches bei der Deserialisierung unbekannte Schlüssel und die zugehörigen Werte ignorieren kann. Andernfalls beinhaltet das unveränderte JSON-Dokument weiterhin Werte zu dem gelöschten Attribut oder der gelöschten Assoziation. Diese Werte können von einem Parser für die aktualisierte Version nicht verarbeitet werden. Jackson bietet in diesem Kontext die Möglichkeit, Werte zu unbekannten Schlüsseln zu ignorieren. Aus diesem Grund können auch JSON-Dokumente verarbeitet werden, die Werte zu alten Schlüsseln enthalten.

Bei einer Entscheidung für die zweite Variante muss wiederum zu Beginn geprüft werden, ob durch das Löschen eine Datenmigration notwendig ist. Algorithmus 6.2 wird dazu mit der Klasse aufgerufen, welche das gelöschte Attribut enthält bzw. von der die gelöschte Assoziation ausgeht. Zur Durchführung einer Datenmigration müssen folgende Schritte durchgeführt werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Klasse repräsentieren, aus der das Attribut bzw. die Assoziation gelöscht wurde.
2. Bestimmung des Namens `$removedKey` des gelöschten Attributs bzw. des Assoziationsendes der gelöschten Assoziation.
3. Löschen des Schlüssels `$removedKey` aus dem aktuellen JSON-Objekt.

Unter der Voraussetzung, dass das linke Klassendiagramm die neue und das rechte Klassendiagramm die alte Modellversion darstellt, illustriert das zuvor gezeigte Beispiel aus Abbildung 6.18 die Auswirkungen einer Modelländerung dieser Art.

Umbenennung eines Attributs oder eines Assoziationsendes

Analog zu den beiden zuvor beschriebenen Fällen muss auch bei der Umbenennung eines Attributs oder eines Assoziationsendes geprüft werden, ob eine Datenmigration erforderlich ist. Dazu wird Algorithmus 6.2 mit der Klasse, die das umbenannte Attribut bzw. die Assoziation mit dem umbenannten Assoziationsende beinhaltet, aufgerufen. Ist diese Klasse von primären Klassen ausgehend erreichbar, müssen die Umbenennungen in den JSON-Dokumenten unter Durchführung folgender Schritte vollzogen werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Klasse repräsentieren, die das umbenannte Attribut bzw. die Assoziation, deren Ziellozationsende umbenannt wurde, enthalten.
2. Bestimmung des alten Namens `$oldName` des umbenannten Attributs bzw. Assoziationsendes.
3. Ermittlung des neuen Namens `$newName` des umbenannten Attributs bzw. Assoziationsendes.
4. Bestimmung des Werts `$value`, der mit dem Schlüssel `$oldName` des aktuellen JSON-Objekts verbunden ist.
5. Einfügen des Schlüssels mit dem Namen `$newName` und dem Wert `$value` in das aktuelle JSON-Objekt.
6. Löschen des Schlüssels mit dem Namen `$oldName` aus dem aktuellen JSON-Objekt.

Eine Umbenennung gemäß der geschilderten Herangehensweise wird allerdings nur dann vorgenommen, wenn das umbenannte Attribut nicht zusätzlich auch verschoben wurde. In diesem Fall wird durch die Verschiebung der neue Name bereits gesetzt, so dass keine Umbenennung durchgeführt werden muss. Gleichmaßen wird der neue Name eines Assoziationsendes ebenfalls implizit beachtet, wenn ein Assoziationsende umbenannt oder die Quellklasse einer Assoziation verändert wurde. Die Gründe hierfür werden später in Abschnitt 6.6.4 und Abschnitt 6.6.5 aufgeführt.

Abbildung 6.19 zeigt ein Beispiel für eine Datenmigration hinsichtlich der Umbenennung eines Attributs. In dem Beispiel wird das Attribut `c` der Klasse `A` in `cNew` umbenannt. Folglich müssen in den zugehörigen JSON-Dokumenten die Schlüssel `c` in `cNew` umbenannt werden.

Verschiebung eines Attributs

In diesem Abschnitt bezieht sich der Begriff der Quellklasse auf die Klasse, die das verschobene Attribut ursprünglich beinhaltet hat. Demgegenüber verweist der Begriff der Zielklasse auf die Klasse, in die das Attribut verschoben wurde. Wenn Quell- und Zielklasse der Verschiebung gemäß Algorithmus 6.2 nicht von primären Klassen ausgehend erreichbar sind, muss keine Datenmigration durchgeführt werden.

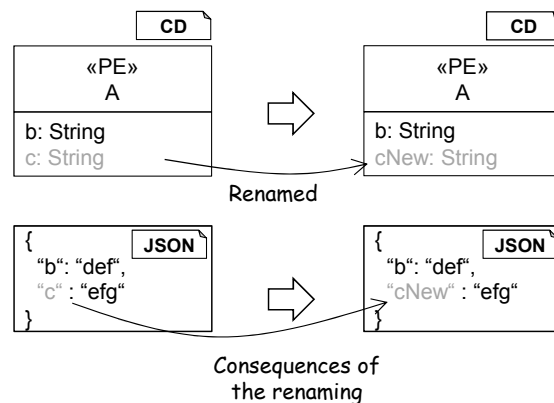


Abbildung 6.19: Datenmigration nach der Umbenennung eines Attributs.

Durch den Vereinfachungsschritt wurden für die Verschiebung eines Attributs bereits die Fälle abgehandelt, dass das Attribut innerhalb einer Vererbungshierarchie oder zwischen zwei voneinander unabhängigen Klassen verschoben wurde. Diese Fälle wurden in Abschnitt 6.4.1 dargestellt. Aus diesem Grund liegt im vereinfachten Differenzmodell nur noch eine Verschiebung eines Attributs vor, wenn von der Quellklasse über Assoziationen zu der Zielklasse oder von der Zielklasse über Assoziationen zu der Quellklasse navigiert werden kann. Dabei wird in diesem Abschnitt vorausgesetzt, dass von einer primären Klasse ausgehend sowohl Quell- als auch Zielklasse erreichbar sind. Andernfalls können die Daten nicht innerhalb der JSON-Dokumente zu der primären Klasse verschoben werden. Ist diese Bedingung nicht erfüllt und lediglich die Zielklasse ist erreichbar, wird die Modelländerung wie das Hinzufügen eines Attributs behandelt. Dies ist darin begründet, dass das verschobene Attribut für die Zielklasse ein neues Attribut darstellt. Demgegenüber wird die Modelländerung wie das Löschen eines Attributs behandelt, wenn lediglich die Quellklasse erreichbar ist. Die Daten müssen in diesem Szenario gelöscht werden, da das Attribut aus der Sicht der Quellklasse gelöscht wurde. Zudem sind die JSON-Objekte für die Zielklasse nicht innerhalb eines JSON-Dokuments für die primäre Klasse erreichbar, so dass die Daten nicht innerhalb des JSON-Dokuments beibehalten werden können.

Die konkrete Herangehensweise zur Durchführung einer Datenmigration richtet sich danach, ob die Zielklasse von der Quellklasse ausgehend erreichbar ist oder andersherum. Als Erstes wird nachfolgend dargestellt, wie die Vorgehensweise ist, wenn die Zielklasse von der Quellklasse ausgehend erreichbar ist. Diese Schritte sind so ausgelegt, dass auch eine etwaige Umbenennung des verschobenen Attributs beachtet wird.

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Quellklasse repräsentieren.
2. Bestimmung des alten Namens `$oldName` des verschobenen Attributs.
3. Ermittlung des Werts `$valueMovedAtt` zu dem Schlüssel `$oldName` des aktuellen JSON-Objekts.

4. Löschen des Schlüssels `$oldName` aus dem aktuellen JSON-Objekt.
5. Ausgehend von dem aktuellen JSON-Objekt: Navigation zu den JSON-Objekten, die Instanzen der Zielklasse repräsentieren.
6. Bestimmung des neuen Namens `$newName` des verschobenen Attributs.
7. Hinzufügen des Schlüssels `$newName` zu dem aktuellen JSON-Objekt. Als Wert für den Schlüssel wird `$valueMovedAtt` gesetzt.

Abbildung 6.20 zeigt ein Beispiel zu dieser Herangehensweise, in dem das Attribut `c` von der Klasse `A` in die Klasse `E` verschoben wurde. Die beiden Klassen sind über die einwertige Assoziation `d` verbunden. Auf diese Weise kann unmittelbar von `A` nach `E` navigiert werden und in den JSON-Dokumenten können die Werte zu dem `c` Attribut direkt in das zugeordnete JSON-Objekt vom Typ `E` verschoben werden. Der untere Teil von Abbildung 6.20 veranschaulicht das für ein JSON-Dokument. Dort wird der Schlüssel `c` samt Wert in das JSON-Objekt verschoben, welches zu dem Schlüssel `d` hinterlegt ist.

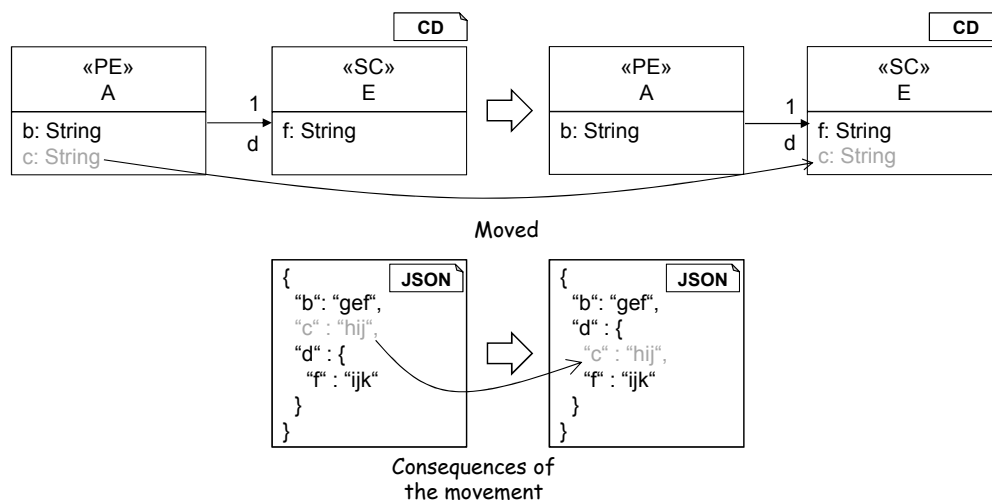


Abbildung 6.20: Datenmigration nach der Verschiebung eines Attributs. Die Zielklasse der Verschiebung ist von der Quellklasse der Verschiebung ausgehend erreichbar.

Ist hingegen die Quellklasse von der Zielklasse ausgehend erreichbar, erfolgt die Datenmigration wie nachstehend aufgeführt. Auch diese Schritte sind so ausgelegt, dass eine etwaige Umbenennung des verschobenen Attributs beachtet wird.

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Zielklasse repräsentieren.
2. Bestimmung des neuen Namens `$newName` des verschobenen Attributs.

3. Ausgehend von dem aktuellen JSON-Objekt: Navigation zu den JSON-Objekten, die Instanzen der Quellklasse repräsentieren.
4. Bestimmung des alten Namens `$oldName` des verschobenen Attributs.
5. Ermittlung des Werts `$valueMovedAtt` zu dem Schlüssel `$oldName` des aktuellen JSON-Objekts.
6. Löschen des Schlüssels `$oldName` aus dem aktuellen JSON-Objekt.
7. Hinzufügen des Schlüssels `$newName` zu dem JSON-Objekt aus dem zweiten Schritt. Als Wert für den Schlüssel wird `$valueMovedAtt` gesetzt.

Eine Besonderheit in diesem Szenario liegt vor, wenn von einem JSON-Objekt der Zielklasse zu mehreren JSON-Objekten der Quellklasse navigiert werden kann und die Navigations-Relation somit nicht injektiv ist. Dieser Fall besteht genau dann, wenn auf dem Pfad von Ziel- zu Quellklasse mindestens eine mehrwertige Assoziation existiert. Die wesentliche Frage in diesem Kontext ist, der Attributwert welches JSON-Objekts der Quellklasse in ein JSON-Objekt der Zielklasse verschoben wird. Abbildung 6.21 veranschaulicht dieses Szenario anhand eines Beispiels, in dem das Attribut `f` aus der Klasse `D` in die Klasse `A` verschoben wird. In dem beispielhaften JSON-Dokument, welches im unteren Teil von Abbildung 6.21 gezeigt ist, existieren zwei `D` Instanzen mit dem gleichen Wert für den Schlüssel `f`. Dieser Wert kann in das übergeordnete JSON-Objekt verschoben werden.

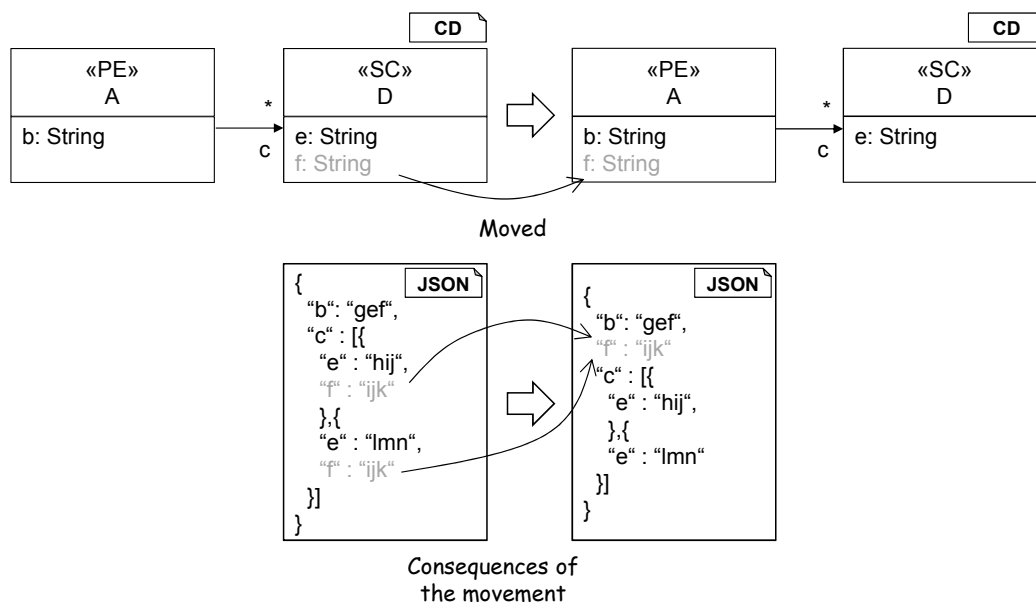


Abbildung 6.21: Datenmigration nach der Verschiebung eines Attributs. Die Quellklasse der Verschiebung ist von der Zielklasse der Verschiebung ausgehend erreichbar.

In dem gezeigten Szenario ist die Verschiebung unproblematisch, da jeweils der gleiche Wert zu dem Attribut angegeben ist. Wären stattdessen verschiedene Werte angegeben, müsste entweder ein spezifischer Wert ausgewählt werden und es käme zu Datenverlusten oder die Datenmigration könnte nicht automatisch durchgeführt werden.

Veränderung der Quellklasse einer Assoziation

Die Veränderung der Quellklasse einer Assoziation erfordert genau dann keine Datenmigration, wenn die alte und die neue Quellklasse der betroffenen Assoziation gemäß Algorithmus 6.2 nicht von primären Klassen ausgehend erreichbar sind.

Aufgrund der vorbereitenden Vereinfachung des Differenzmodells, siehe Abschnitt 6.4, kann eine derartige Modelländerung nur noch in zwei Fällen in dem Differenzmodell vorliegen: 1. die alte Quellklasse ist über Assoziationen von der neuen Quellklasse oder 2. die neue Quellklasse ist über Assoziationen von der alten Quellklasse ausgehend erreichbar. Die Herangehensweise zur Durchführung einer Datenmigration richtet sich danach, welcher der beiden Fälle vorliegt. Im Folgenden wird zunächst der erstgenannte Fall betrachtet. Analog zu der Verschiebung eines Attributs beziehen auch diese Schritte die Möglichkeit ein, dass das Assoziationsende zur Zielklasse hin umbenannt worden sein konnte.

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der neuen Quellklasse repräsentieren.
2. Bestimmung des neuen Namens `$newName` des Assoziationsendes der Zielklasse.
3. Hinzufügen des Schlüssels `$newName` zum aktuellen JSON-Objekt. Der Wert zu diesem Schlüssel wird in den nächsten Schritten gesetzt.
4. Ausgehend von dem aktuellen JSON-Objekt: Navigation zu den JSON-Objekten, die Instanzen der alten Quellklasse repräsentieren.
5. Bestimmung des alten Namens `$oldName` des Assoziationsendes der Zielklasse.
6. Ermittlung des Werts `$valueAssocName` zu dem Schlüssel `$oldName` des aktuellen JSON-Objekts.
7. Löschen des Schlüssels `$oldName` aus dem aktuellen JSON-Objekt.
8. Hinzufügen des Werts `$valueAssocName` zum Schlüssel `$newName` des JSON-Objekts aus dem zweiten Schritt.

Diese Vorgehensweise ist analog zu der Vorgehensweise bei der Verschiebung eines Attributs, bei der die Quellklasse von der Zielklasse ausgehend erreichbar ist. Aus diesem Grund stellt sich analog dazu auch hier die Frage, wie der Fall behandelt wird, dass von einer Instanz der neuen Quellklasse ausgehend zu mehreren Instanzen der alten Quellklasse navigiert werden kann. Diese Konstellation liegt vor, wenn auf dem Pfad von der neuen Quellklasse zu der alten Quellklasse mindestens eine mehrwertige Assoziation existiert. Das Problem in diesem Szenario ist, dass zu

mehreren JSON-Objekten der alten Quellklasse Objekte bei der betroffenen Assoziation hinterlegt sein können, die in das gleiche JSON-Objekt zu der neuen Quellklasse verschoben werden sollen. Sind zu den verschiedenen Instanzen der betroffenen alten Quellklasse nicht jeweils die gleichen Objekte hinterlegt, kann folglich keine automatische Datenmigration ohne Datenverluste durchgeführt werden. Andernfalls gehen bei der Verschiebung der JSON-Objekte definitiv Daten verloren.

Abbildung 6.22 veranschaulicht die zuvor beschriebene Herangehensweise zur Datenmigration anhand eines Beispiels. In diesem wird die Quellklasse der Assoziation e von der Klasse C in die Klasse A geändert. In dem gezeigten JSON-Dokument sind zu der Instanz vom Typ C zwei JSON-Objekte über den Schlüssel e hinterlegt. Diese werden in das JSON-Objekt zur übergeordneten Instanz vom Typ A verschoben, da dies die neue Quellklasse der Assoziation ist. Die bestehenden Daten können in diesem Szenario somit vollkommen erhalten werden.

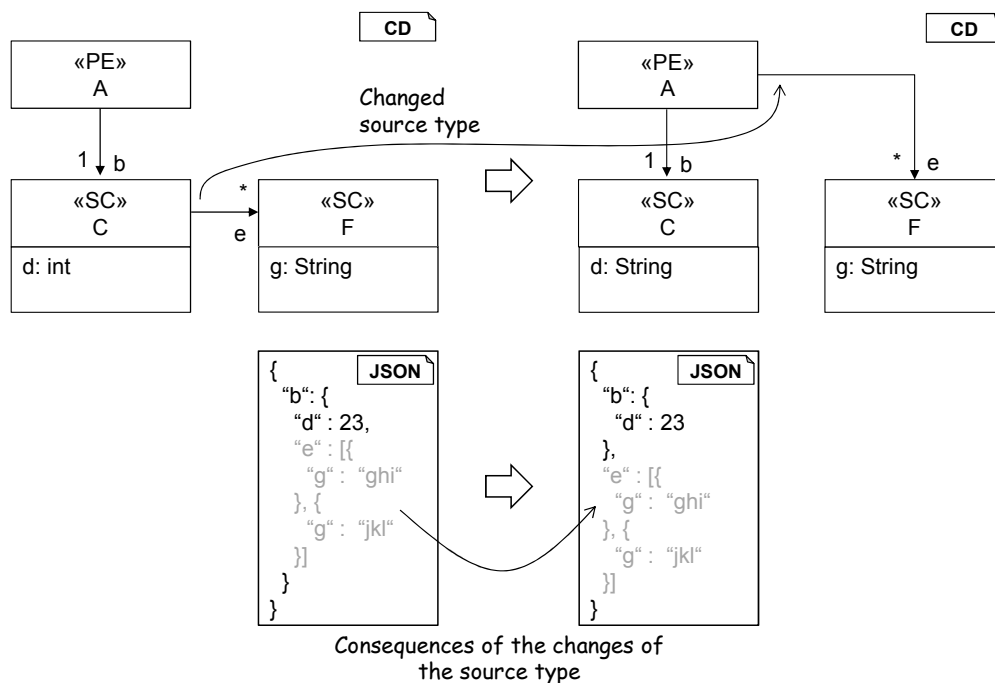


Abbildung 6.22: Datenmigration nach der Veränderung der Quellklasse einer Assoziation. Die alte Quellklasse ist von der neuen Quellklasse ausgehend erreichbar.

Die zuvor aufgeführte Herangehensweise ist nur dann valide, wenn von einer primären Klasse ausgehend zu der neuen und somit implizit auch zu der alten Quellklasse navigiert werden kann. Ist hingegen nur die alte Quellklasse von einer primären Klasse ausgehend erreichbar, wird eine Datenmigration durchgeführt, in der die Modelländerung wie das Löschen der Assoziation behandelt wird. Dies ist darin begründet, dass die zu der alten Quellklasse hinterlegten Daten in der neuen Version nicht mehr gültig sind und somit entfernt werden müssen. In Abbildung 6.22 liegt dieser Fall vor, wenn die Klasse A keine primäre Klasse ist und von einer primären Klas-

se ausgehend lediglich die Klasse *C* erreichbar ist. Für die JSON-Objekte zu Klasse *C* müssten folglich die beim Schlüssel *e* hinterlegten Daten gelöscht werden.

Als Nächstes wird dargelegt, wie eine Datenmigration durchgeführt wird, wenn die neue Quellklasse von der alten Quellklasse ausgehend erreichbar ist.

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der alten Quellklasse repräsentieren.
2. Bestimmung des alten Namens `$oldName` des Assoziationsendes der Zielklasse.
3. Ermittlung des Werts `$assocValue` zu dem Schlüssel `$oldName` des aktuellen JSON-Objekts.
4. Löschen des Schlüssels `$oldName` aus dem aktuellen JSON-Objekt.
5. Ausgehend von dem aktuellen JSON-Objekt: Navigation zu den JSON-Objekten, die Instanzen der neuen Quellklasse repräsentieren.
6. Bestimmung des neuen Namens `$newName` des Assoziationsendes der Zielklasse.
7. Hinzufügen des Schlüssels `$newName` zu dem aktuellen JSON-Objekt. Als Wert für den Schlüssel wird `$assocValue` gesetzt.

Abbildung 6.23 illustriert diese Vorgehensweise anhand eines Beispiels. In diesem wird die Quellklasse der Assoziation *b* von der Klasse *A* in die Klasse *F* geändert. In dem gezeigten JSON-Dokument zu der alten Modellversion ist zu dem Schlüssel *b* ein JSON-Objekt hinterlegt. Dieses JSON-Objekt wird aus dem JSON-Objekt zu Typ *A* entfernt und in alle JSON-Objekte vom Typ *F* hinzugefügt, die über den Pfad von Klasse *A* zu Klasse *F* erreichbar sind. In diesem Szenario können die Daten somit erhalten werden.

Analog zum vorherigen Fall setzt auch die zuletzt geschilderte Vorgehensweise voraus, dass die alte und somit implizit auch die neue Quellklasse von einer primären Klasse ausgehend erreicht werden können. Ist in diesem Szenario hingegen nur die neue Quellklasse von einer primären Klasse ausgehend erreichbar, wird eine Datenmigration durchgeführt, in der die Modelländerung wie das Hinzufügen der Assoziation behandelt wird. Dies ist darin begründet, dass zu der neuen Quellklasse in der neuen Version Daten zu der Assoziation hinterlegt werden können. In Abbildung 6.23 ist dies der Fall, wenn die Klasse *A* keine primäre Klasse ist und von einer primären Klasse ausgehend nur die Klasse *F* erreichbar ist. Für die JSON-Objekte zu Klasse *F* kann folglich ein Schlüssel *b* hinzugefügt werden.

Die zuvor beschriebene Herangehensweise ist darüber hinaus mit der Einschränkung verbunden, dass die Zielklasse der betroffenen Assoziation nicht auf dem Pfad von der alten Quellklasse zu der neuen Quellklasse liegen darf. In diesem Fall entsteht durch die Veränderung der Quellklasse ein Zyklus, wie in Abbildung 6.24 für die Assoziation *b* beispielhaft verdeutlicht. Werden zu den JSON-Objekten der neuen Quellklasse nun die JSON-Objekte hinzugefügt, die über die Assoziation zu der alten Quellklasse hinterlegt waren, kann ein Zyklus in den Daten entstehen. Zur Vereinfachung der Beschreibung wird dieser Sonderfall im weiteren Verlauf allerdings nicht näher betrachtet, da die Umsetzung stark von der Handhabung der Zyklen des betrachteten Serialisierungsframeworks abhängt.

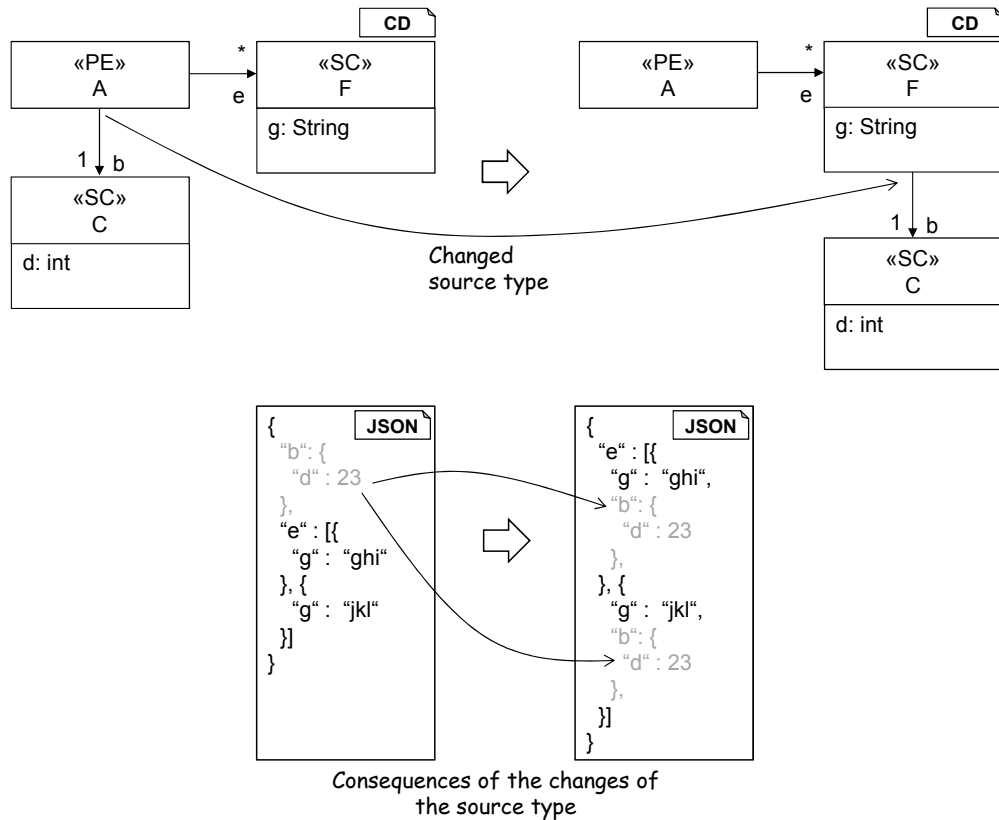


Abbildung 6.23: Datenmigration nach der Veränderung der Quellklasse einer Assoziation. Die neue Quellklasse ist von der alten Quellklasse ausgehend erreichbar.

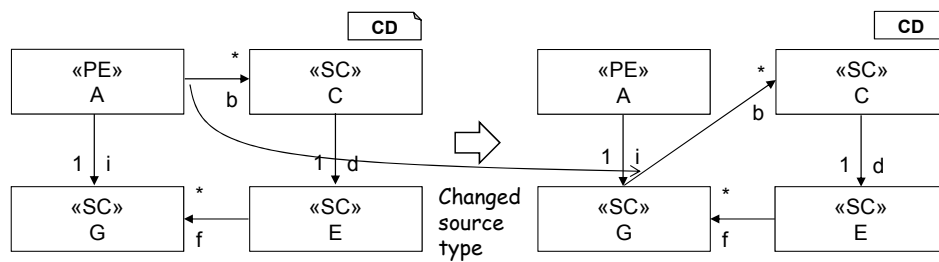


Abbildung 6.24: Veränderung der Quellklasse einer Assoziation. Die Zielklasse, der von der Änderung betroffenen Assoziation, liegt auf dem Pfad von der alten Quellklasse zu der neuen Quellklasse.

Veränderung der Zielklasse einer Assoziation

Im Falle einer Veränderung der Zielklasse einer Assoziation ist keine Datenmigration erforderlich, wenn die Quellklasse der betroffenen Assoziation gemäß Algorithmus 6.2 von keiner primären Klasse ausgehend erreichbar ist.

Aufgrund der vorbereitenden Vereinfachung des Differenzmodells, siehe Abschnitt 6.4, kann eine derartige Modelländerung in dem Differenzmodell nur noch vorliegen, wenn die neue Zielklasse eine Unterklasse der alten Zielklasse ist. Eine Datenmigration wird in diesem Fall wie folgt vorgenommen:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Quellklasse der betroffenen Assoziation darstellen.
2. Bestimmung des Namens `$newTargetType` der neuen Zielklasse.
3. Ermittlung des Namens `$assocName` des betroffenen Assoziationsendes.
4. Ermittlung der Werte `$values`, die zu dem Schlüssel `$assocName` hinterlegt sind.
5. Iteration über alle JSON-Objekte aus `$values`:
 - a) Bestimmung des Werts zu dem Schlüssel `@class` für das aktuell betrachtete JSON-Objekt aus `$values`.
 - b) Prüfung, ob das aktuell betrachtete JSON-Objekt aus `$values` auf `$newTargetType` verweist, unter Nutzung des zuvor ermittelten Schlüssels `@class`. Falls diese Bedingung nicht erfüllt ist, muss das aktuell betrachtete JSON-Objekt aus dem JSON-Dokument gelöscht werden. Andernfalls kann der hinterlegte Wert beibehalten werden.

Ein konkretes Beispiel für diese Datenmigration zeigt Abbildung 6.25. Das Beispiel verdeutlicht den Fall, dass die Zielklasse der Assoziation `b` von der Klasse `B` auf die Unterklasse `D` verändert wird. Das im unteren Teil von Abbildung 6.25 gezeigte JSON-Dokument wird in diesem Fall in der Art migriert, dass das JSON-Objekt zu der Klasse `E` aus der Liste zu dem Schlüssel `b` entfernt wird, da dieses JSON-Objekt nicht kompatibel zu der neuen Zielklasse ist.

Veränderung der Kardinalität eines Assoziationsendes von einwertig zu mehrwertig

Zur Prüfung, ob aufgrund der Veränderung der Kardinalität eines Assoziationsendes eine Datenmigration durchgeführt werden muss, wird Algorithmus 6.2 mit der Quellklasse der betroffenen Assoziation aufgerufen. Ergibt dieser Aufruf des Algorithmus, dass eine Datenmigration erforderlich ist, wird diese wie folgt durchgeführt:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Klasse repräsentieren, von der die Assoziation mit der veränderten Kardinalität ausgeht.

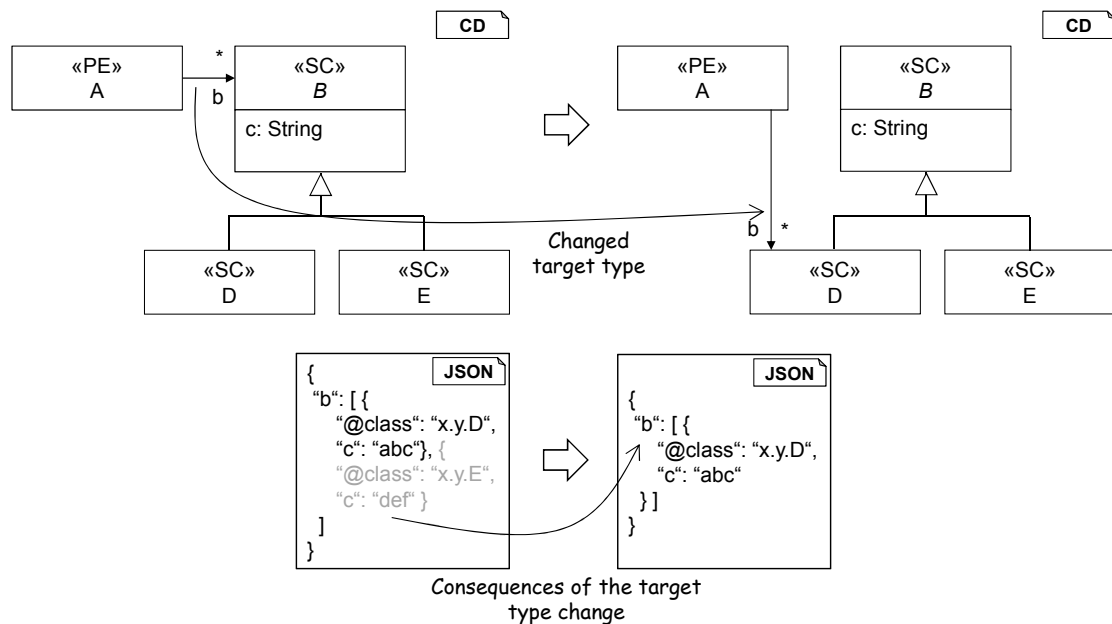


Abbildung 6.25: Datenmigration nach der Veränderung der Zielklasse einer Assoziation.

2. Bestimmung des Namens `$assocName` des Assoziationsendes, dessen Kardinalität verändert wurde.
3. Ermittlung des Werts `$valueForKey` zu dem Schlüssel `$assocName` des aktuellen JSON-Objekts.
4. Erzeugung einer Liste von JSON-Objekten. In diese Liste wird der zuvor ermittelte Wert `$valueForKey` hinein kopiert.
5. Überschreiben des existierenden Werts des Schlüssels `$assocName` durch das Setzen dieser Liste als neuen Wert in dem aktuellen JSON-Objekt.

Abbildung 6.26 zeigt ein Beispiel für die Modelländerung, in dem die Kardinalität des Assoziationsendes `c` hin zu einem mehrwertigen Assoziationsende geändert wurde. Im unteren Teil von Abbildung 6.26 ist ein JSON-Dokument für eine Instanz vom Typ `A` dargestellt, welche auf eine Instanz vom Typ `D` verweist. Dieser Verweis wird in der neuen Version in einer Liste gekapselt, die nur die hinterlegte Instanz enthält.

Veränderung der Kardinalität eines Assoziationsendes von mehrwertig zu einwertig

Wird ein mehrwertiges Assoziationsende in ein einwertiges Assoziationsende verändert, erfolgt die Prüfung auf die Notwendigkeit einer Datenmigration analog zu dem vorherigen Fall. Da in der Ausgangssituation ursprünglich ein mehrwertiges Assoziationsende vorgelegen hat, ist in

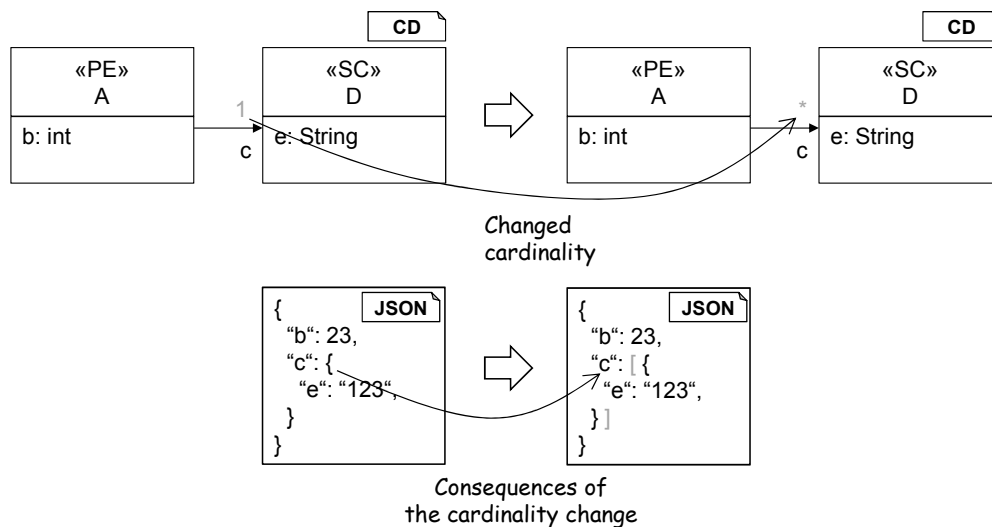


Abbildung 6.26: Datenmigration nach der Veränderung der Kardinalität eines Assoziationsendes von einwertig zu mehrwertig.

einem JSON-Dokument zu dem von der Modelländerung betroffenen Schlüssel eine Liste von JSON-Objekten hinterlegt. Infolge der Datenmigration darf der Schlüssel aber nur noch auf ein einzelnes JSON-Objekt verweisen. Das kann erreicht werden, indem aus der Liste von JSON-Objekten ein JSON-Objekt ausgewählt wird und in dem JSON-Dokument zu dem Schlüssel hinterlegt wird. Die Kernfrage in diesem Kontext ist allerdings, welches konkrete JSON-Objekt ausgewählt werden soll. Hierbei gibt es verschiedene Alternativen:

- Im Zuge der Datenmigration wird automatisch ein spezifisches Element ausgewählt, z.B. das erste Element der Liste. Enthält die Liste jedoch zwei oder mehr unterschiedliche JSON-Objekte, führt dieser Ansatz zu Datenverlusten.
- Eine Datenmigration wird nur dann ausgeführt, wenn zu dem Schlüssel eine Liste von JSON-Objekten angegeben ist, die nur ein JSON-Objekt enthält. Dieses JSON-Objekt kann als neuer Wert für den Schlüssel verwendet werden und die Datenmigration führt zu keinem Datenverlust. Enthält die Liste mehr als ein JSON-Objekt, wird die Datenmigration hingegen nicht automatisch ausgeführt. Stattdessen müssen die Entwickler die generierten Migrationsskripte erweitern und fallabhängig festlegen, wie mit welchen JSON-Dokumenten verfahren wird.

Sobald die Entscheidung für eine dieser Strategien getroffen wurde, kann die Datenmigration wie folgt durchgeführt werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Instanzen der Klasse repräsentieren, von der die Assoziation mit der veränderten Kardinalität ausgeht.

2. Bestimmung des Namens `$assocName` des Assoziationsendes, dessen Kardinalität verändert wurde.
3. Ermittlung des Werts `$valueForKey` zu dem Schlüssel `$assocName` des aktuellen JSON-Objekts.
4. Extraktion eines Werts `$fieldValue` aus der Liste `$valueForKey`, z.B. Extraktion des ersten Elements der Liste.
5. Überschreiben des existierenden Werts des Schlüssels `$assocName` in dem aktuellen JSON-Objekt. Dazu wird `$fieldValue` als neuer Wert für den Schlüssel gesetzt.

Die Vorgehensweise ist also analog zu dem Fall, dass ein einwertiges Assoziationsende hin zu einem mehrwertigen Assoziationsende verändert wird. Der einzige Unterschied ist, dass als neuer Wert der einzelne kopierte Wert und nicht die Liste mit dem kopierten Wert gesetzt wird. Folglich illustriert Abbildung 6.26 die Auswirkungen einer Modelländerung dieser Art, unter der Annahme, dass das linke Klassendiagramm die neue und das rechte Klassendiagramm die alte Modellversion darstellt.

6.5.2 Handhabung von Modelländerungen zu Enumerationen

Löschen eines Literals aus einer Enumeration

Das Löschen eines Literals aus einer Enumeration erfordert dann eine Datenmigration, wenn das Literal aus einer Enumeration gelöscht wurde, welche von einer primären Klasse ausgehend erreichbar ist. Demzufolge wird Algorithmus 6.2 mit dieser Enumeration aufgerufen, um herauszufinden, ob diese Bedingung erfüllt ist. Ist eine Datenmigration erforderlich, wird dazu wie folgt vorgegangen:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Schlüssel enthalten, die auf die betroffene Enumeration verweisen.
2. Ermittlung des Namens des Attributs `$enumAttName`, welches die betroffene Enumeration referenziert.
3. Bestimmung des Namens `$deletedLiteral` des gelöschten Literals.
4. Ermittlung des Werts `$enumValue`, der dem Schlüssel `$enumAttName` des aktuellen JSON-Objekts zugeordnet ist.
5. Entspricht `$enumValue` dem Wert `$deletedLiteral`, wird der Schlüssel `$enumAttName` aus dem aktuellen JSON-Objekt entfernt.

Die Modelländerung wird in diesem Fall dem Knoten für die Klasse zugeordnet, die ein Attribut enthält, welches auf die betroffene Enumeration verweist.

Abbildung 6.27 illustriert die Auswirkungen dieser Datenmigration. Hier wird das Literal `H` aus der Enumeration `F` entfernt. Das JSON-Dokument im unteren Teil von Abbildung 6.27 kapselt zwei Instanzen vom Typ `C`, die auf die betroffene Enumeration verweisen. Aus der Instanz,

die auf das gelöschte Literal verweist, muss der Schlüssel `e` entfernt werden, da der Verweis auf das Literal unzulässig ist.

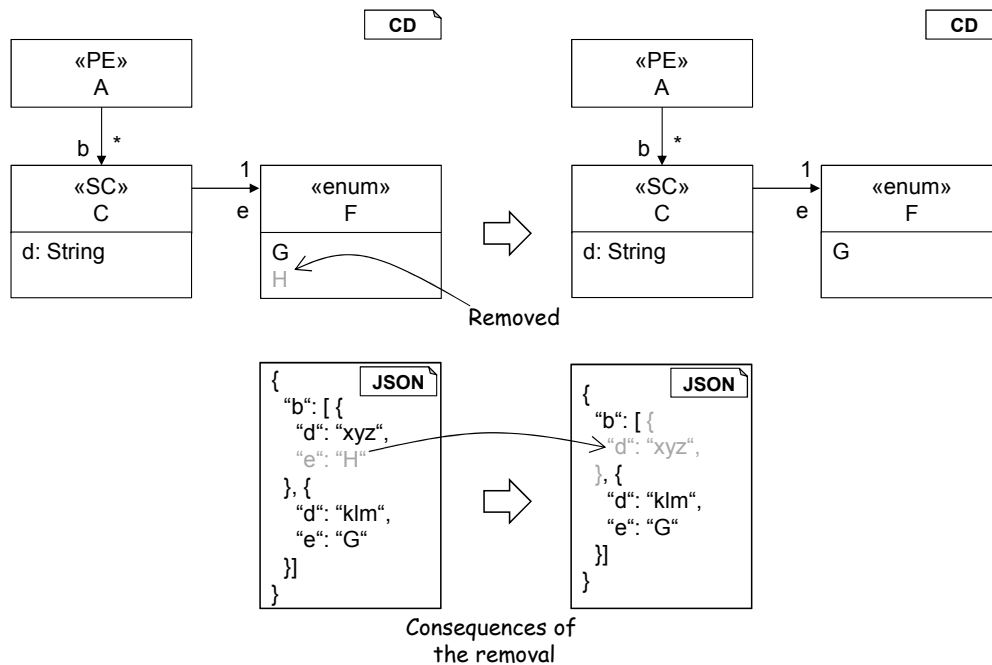


Abbildung 6.27: Datenmigration nach dem Löschen eines Literals einer Enumeration.

Umbenennung eines Literals in einer Enumeration

Analog zu dem Löschen eines Literals aus einer Enumeration bedingt die Umbenennung eines Literals in einer Enumeration genau dann eine Datenmigration, wenn die betroffene Enumeration von einer primären Klasse ausgehend erreichbar ist. Zur Handhabung der Datenmigration wird in diesem Fall wie folgt verfahren:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Schlüssel enthalten, die auf die betroffene Enumeration verweisen.
2. Ermittlung des Namens des Attributs `$enumAttName`, welches die betroffene Enumeration referenziert.
3. Bestimmung des ursprünglichen Namens `$oldName` des umbenannten Literals.
4. Ermittlung des neuen Namens `$newName` des umbenannten Literals.
5. Bestimmung des Werts `$enumValue`, der dem Schlüssel `$enumAttName` des aktuellen JSON-Objekts zugeordnet ist.

6. Entspricht der Wert `$enumValue` dem Wert `$oldName`, wird in dem aktuellen JSON-Objekt dem Schlüssel `$enumAttName` der Wert `$newName` zugewiesen.

Analog zum vorherigen Fall, wird die Modelländerung dem Knoten für die Klasse zugeordnet, die ein Attribut mit einem Verweis auf die betroffene Enumeration enthält.

Abbildung 6.28 demonstriert die Auswirkungen dieser Datenmigration. Hierbei müssen sämtliche Werte zu Schlüsseln angepasst werden, die auf das umbenannte Literal verweisen. In diesem Beispiel müssen folglich sämtliche Verweise auf das Literal `H` in `HNEW` umbenannt werden.

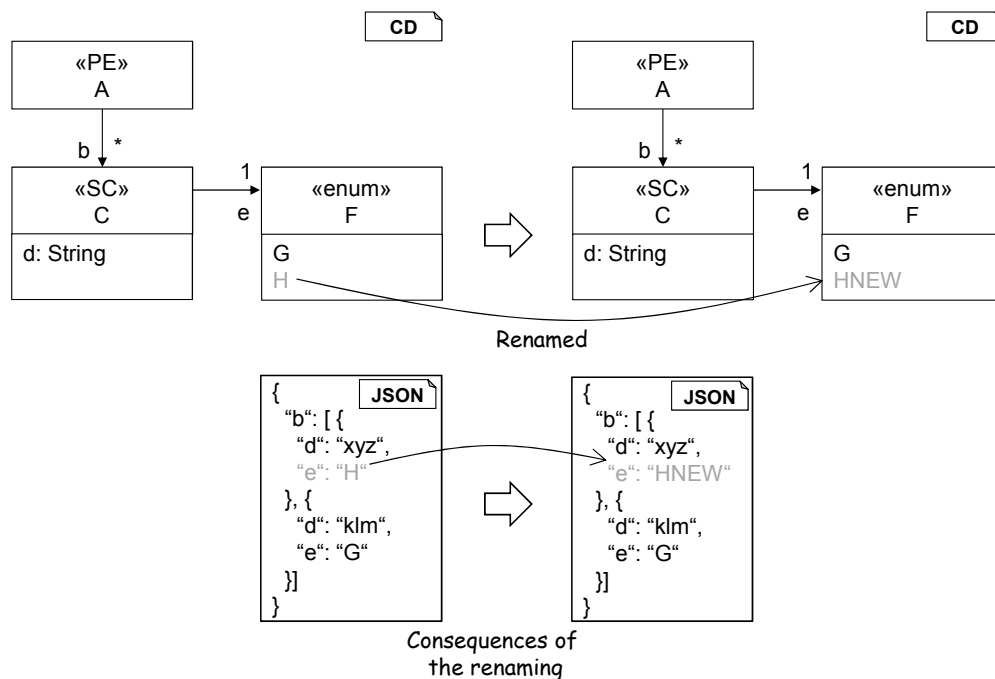


Abbildung 6.28: Datenmigration nach der Umbenennung eines Literals einer Enumeration.

6.5.3 Handhabung von Modelländerungen zu Klassen

Löschen einer Klasse

Zur Prüfung, ob durch das Löschen einer Klasse eine Datenmigration erforderlich ist, wird Algorithmus 6.2 für die gelöschte Klasse aufgerufen. Zur Durchführung einer Datenmigration müssen in diesem Fall folgende Schritte durchgeführt werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Schlüssel enthalten, deren Werte auf Instanzen der gelöschten Klasse verweisen können.
2. Falls eine Assoziation direkt auf die gelöschte Klasse verweist, muss im Zuge der Löschoperation der Klasse ebenfalls die Assoziation gelöscht worden sein. Andernfalls verweist

die Assoziation nicht auf eine Zielklasse und dies ist unzulässig. Aus diesem Grund wird die Modelländerung in dieser Konstellation keinem Knoten zugeordnet, da die Datenmigration bereits über das Löschen der entsprechenden Assoziation korrekt vollzogen wird. Das Löschen einer Klasse kann im Übrigen nicht im Vereinfachungsschritt für das Differenzmodell, siehe Abschnitt 6.4, behandelt werden, da die Modelländerung nicht pauschal aus dem Differenzmodell entfernt werden kann. Der Grund dafür ist, dass andere Assoziationen auf eine Unter- bzw. Oberklasse der gelöschten Klasse verweisen können und diese Fälle anders behandelt werden müssen, wie nachfolgend noch gezeigt wird.

3. Falls eine Assoziation auf eine Oberklasse der gelöschten Klasse verweist, kann der Schlüssel samt Wert nicht unmittelbar gelöscht werden. Stattdessen muss eine Fallunterscheidung unternommen werden:
 - Für einwertige Assoziationen kann der Schlüssel nur dann entfernt werden, wenn der Wert des Schlüssels auf eine Instanz der gelöschten Klasse verweist. Handelt es sich bei der gelöschten Klasse um eine abstrakte Klasse, wird geprüft, ob eine Instanz einer Unterklasse dieser Klasse vorliegt.
 - Für mehrwertige Assoziationen muss über sämtliche JSON-Objekte iteriert werden, die dem Schlüssel zugeordnet sind. Ein Element der Liste wird genau dann gelöscht, wenn das Element auf die gelöschte Klasse verweist. Im Falle einer gelöschten abstrakten Klasse wird wie im vorherigen Punkt beschrieben verfahren.

Die Modelländerung wird in diesen Fällen dem Knoten für die Quellklasse der betroffenen Assoziation zugeordnet.

4. Falls eine Assoziation auf eine Unterklasse der gelöschten Klasse verweist, muss im Zuge der Löschoperation der Klasse ebenfalls eine Vererbungsbeziehung gelöscht oder geändert worden sein. Andernfalls bestünde eine Vererbungsbeziehung zu einer nicht länger vorhandenen Klasse und dies ist unzulässig. Aus diesem Grund wird die Modelländerung in dieser Konstellation keinem Knoten zugeordnet, da die Datenmigration bereits über die entsprechende Änderung der Vererbungsbeziehung korrekt vollzogen wird.

Abbildung 6.29 zeigt die Durchführung der Datenmigration für einen Fall auf, in dem die Klasse *F* entfernt wird. Dies führt dazu, dass alle über *b* referenzierten *F* Instanzen aus der Liste entfernt werden, wie im unteren Teil von Abbildung 6.29 für ein beispielhaftes JSON-Dokument gezeigt.

Umbenennung einer Klasse

Wie bei den zuvor beschriebenen Modelländerungen wird initial geprüft, ob die konkrete Umbenennung eine Datenmigration erfordert. Algorithmus 6.2 wird zu diesem Zweck mit der umbenannten Klasse aufgerufen. Wird dabei festgestellt, dass eine Datenmigration erforderlich ist, müssen folgende Schritte zur Durchführung der Datenmigration ausgeführt werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Schlüssel enthalten, deren Werte auf Instanzen der umbenannten Klasse verweisen können.

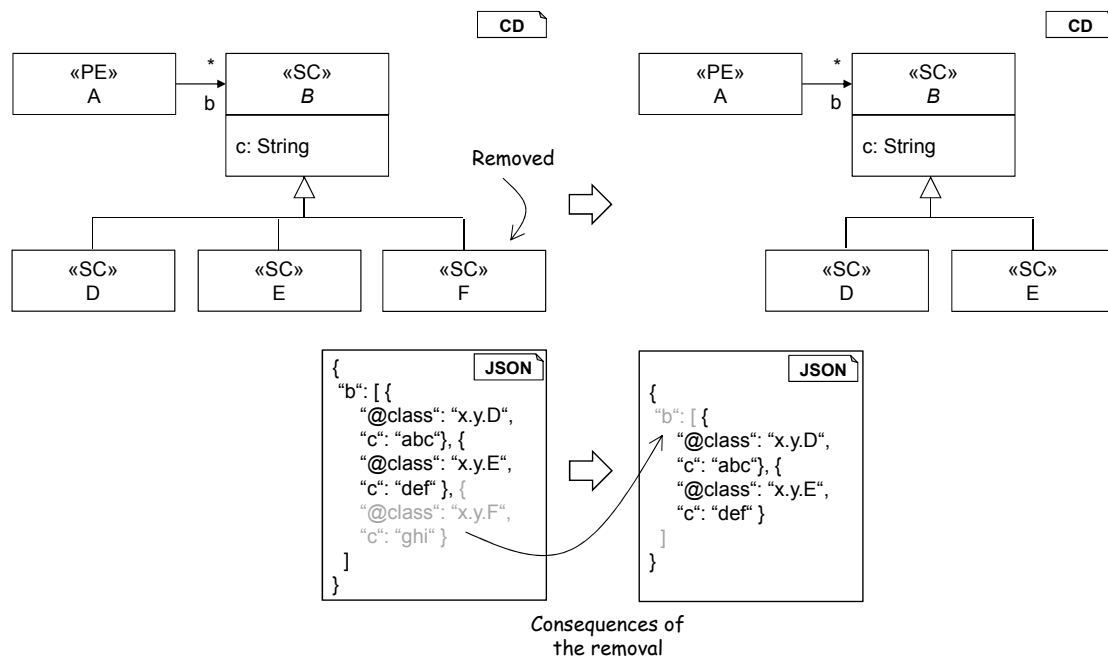


Abbildung 6.29: Datenmigration nach dem Löschen einer Klasse.

2. Bestimmung des alten vollqualifizierten Namens `$oldNameClass` der umbenannten Klasse.
3. Ermittlung des neuen vollqualifizierten Namens `$newNameClass` der umbenannten Klasse.
4. Bestimmung des Werts des Schlüssels `@class` für jedes einzelne in Frage kommende JSON-Objekt.
5. Prüfung, ob das aktuell betrachtete JSON-Objekt auf eine Instanz der Klasse `$oldNameClass` verweist. Dies erfolgt unter Prüfung des Werts des zuvor ermittelten Schlüssels `@class`. Ist diese Bedingung erfüllt, kann mit dem nächsten Schritt fortgefahren werden.
6. Aktualisierung des Werts des Schlüssels `@class` durch das Setzen auf den Namen `$newNameClass`.

Ein solches Szenario wird in Abbildung 6.30 veranschaulicht. In dem Beispiel wird die Klasse `x.y.C` in `x.y.CNew` umbenannt. Das im unteren Teil von Abbildung 6.30 gezeigte JSON-Dokument enthält zwei Verweise auf Instanzen der Klasse `C`, so dass für diese der Wert des Schlüssels `@class` in `x.y.CNew` geändert werden muss.

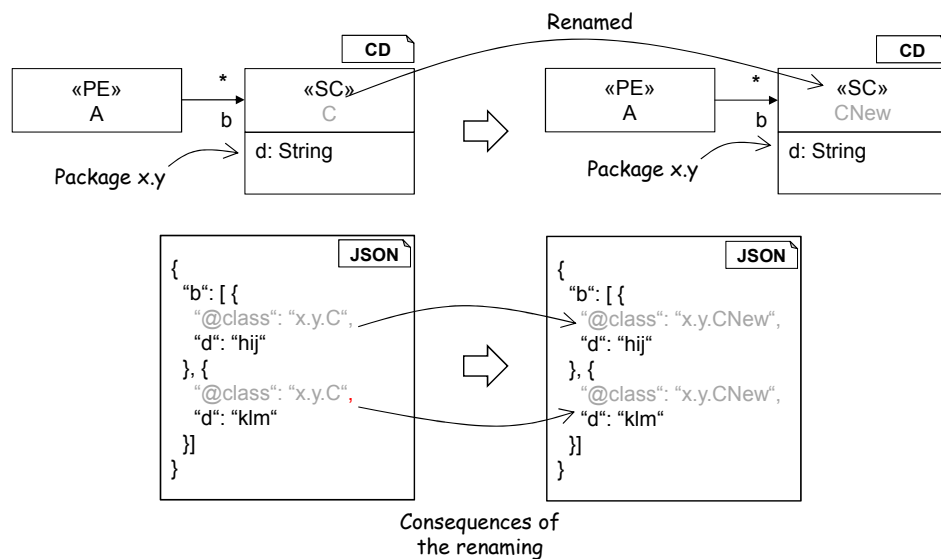


Abbildung 6.30: Datenmigration nach der Umbenennung einer Klasse.

Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse

Wie in Abschnitt 6.4.8 bereits dargelegt, wird in dieser Arbeit vorausgesetzt, dass lediglich abstrakte Klassen Unterklassen besitzen können. Diese Annahme ist durch das Projekt motiviert, in dessen Kontext die Arbeit durchgeführt wurde. Aufgrund der vorbereitenden Vereinfachung des Differenzmodells, siehe Abschnitt 6.4, kann eine Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse nur noch in zwei Fällen im vereinfachten Differenzmodell vorliegen: 1. die Unterklassen der abermals abstrakten Klasse oder 2. die Vererbungsbeziehungen dieser Unterklassen zu der betroffenen abstrakten Klasse wurden gelöscht. Zur Vereinfachung der Beschreibung wird davon ausgegangen, dass die Klasse P die abermals abstrakte Klasse ist. Die Prüfung, ob für die Modelländerung eine Datenmigration erforderlich ist, erfolgt durch den Aufruf von Algorithmus 6.2 mit der Klasse P . Ist diese Klasse von mindestens einer primären Klasse ausgehend erreichbar, muss eine Datenmigration wie folgt durchgeführt werden:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Schlüssel enthalten, deren Werte auf Instanzen der Klasse P verweisen können.
2. Bestimmung des vollqualifizierten Namens $\$nameClass$ der Klasse P .
3. Falls eine Assoziation a direkt auf die Klasse P oder eine Oberklasse davon verweist:
 - a) Ermittlung der Werte $\$values$, die zu der betroffenen Assoziation a hinterlegt sind.
 - b) Iteration über alle JSON-Objekte aus $\$values$.
 - c) Falls das aktuell betrachtete JSON-Objekt auf eine Instanz einer Unterklasse der Klasse P verweist: Aktualisierung des Werts des Schlüssels `@class` des betrach-

teten JSON-Objekts auf den Wert `$nameClass`. Sämtliche JSON-Objekte, die Instanzen einer Unterklasse der Klasse `P` waren, werden fortan als Instanzen der nun nicht mehr abstrakten Klasse `P` angesehen.

- d) Löschung sämtlicher Schlüssel aus dem aktuellen JSON-Objekt, die keine Attribute oder Assoziationen der Klasse `P` repräsentieren. Ausgenommen sind hiervon die Attribute der Unterklassen, die in die Klasse `P` oder eine Oberklasse davon verschoben wurden. Auf diese Weise werden aus den JSON-Objekten sämtliche Schlüssel entfernt, die spezifisch für die Unterklassen der Klasse `P` waren.

Die Modelländerung wird in diesem Fällen dem Knoten für die Quellklasse der betroffenen Assoziation zugeordnet. Ferner werden aus diesem Knoten abschließend die Modelländerungen entfernt, die auf das Löschen der Unterklassen der Klasse `P` oder auf Änderungen in der Vererbungshierarchie unterhalb der Klasse `P` verweisen. Durch die Umbenennung der Werte zum Schlüssel `@class` haben die Migrationen zu diesen Modelländerungen keine Auswirkungen, so dass sie zur Vereinfachung der Migrationsklasse direkt entfernt werden können.

4. Der Fall, dass eine Assoziation in der alten Version auf eine Unterklasse von `P` verweist, muss für diese Modelländerung hingegen nicht betrachtet werden. Bei dem Knoten für die Quellklasse der betroffenen Assoziation wird stattdessen das Löschen der Vererbungsbeziehung hinterlegt sein, wenn die Klasse selbst nicht gelöscht wurde. Im Zuge der Behandlung dieser Modelländerung werden bereits die notwendigen Migrationsschritte ausgeführt, so dass keine separaten Migrationsschritte für die Veränderung der abstrakten Klasse hin zu einer nicht abstrakten Klasse durchgeführt werden müssen.

Abbildung 6.31 illustriert die Durchführung der Datenmigration in diesem Szenario. In dem Beispiel wird die abstrakte Klasse `x.y.B` hin zu einer nicht abstrakten Klasse verändert und die Unterklassen werden entfernt. In dem zugehörigen JSON-Dokument im unteren Teil von Abbildung 6.31 müssen dadurch die zur Assoziation `b` hinterlegten JSON-Objekte angepasst werden. Konkret muss als Typ für die verschiedenen JSON-Objekte nun `x.y.B` eingetragen werden und zudem müssen die Schlüssel `g` und `h` entfernt werden, da diese zu Attributen der Unterklassen gehören.

Veränderung einer nicht abstrakten Klasse hin zu einer abstrakten Klasse

Aufgrund der vorbereitenden Vereinfachung des Differenzmodells, siehe Abschnitt 6.4, kann diese Modelländerung nur noch vorliegen, wenn die abermals nicht abstrakte Klasse in der neuen Version genau eine Unterklasse besitzt. Zur Vereinfachung der Beschreibung wird davon ausgegangen, dass die Klasse `C` die neue Unterklasse und die Klasse `P` die abermals nicht abstrakte Klasse ist. Eine Datenmigration ist in diesem Fall nur dann erforderlich, wenn die Klasse `P` gemäß Algorithmus 6.2 von mindestens einer primären Klasse ausgehend erreichbar ist. Die erforderlichen Migrationsschritte werden nachfolgend dargelegt:

1. Navigation vom Wurzelobjekt des JSON-Dokuments zu den JSON-Objekten, die Schlüssel enthalten, deren Werte auf Instanzen von `P` verweisen können.

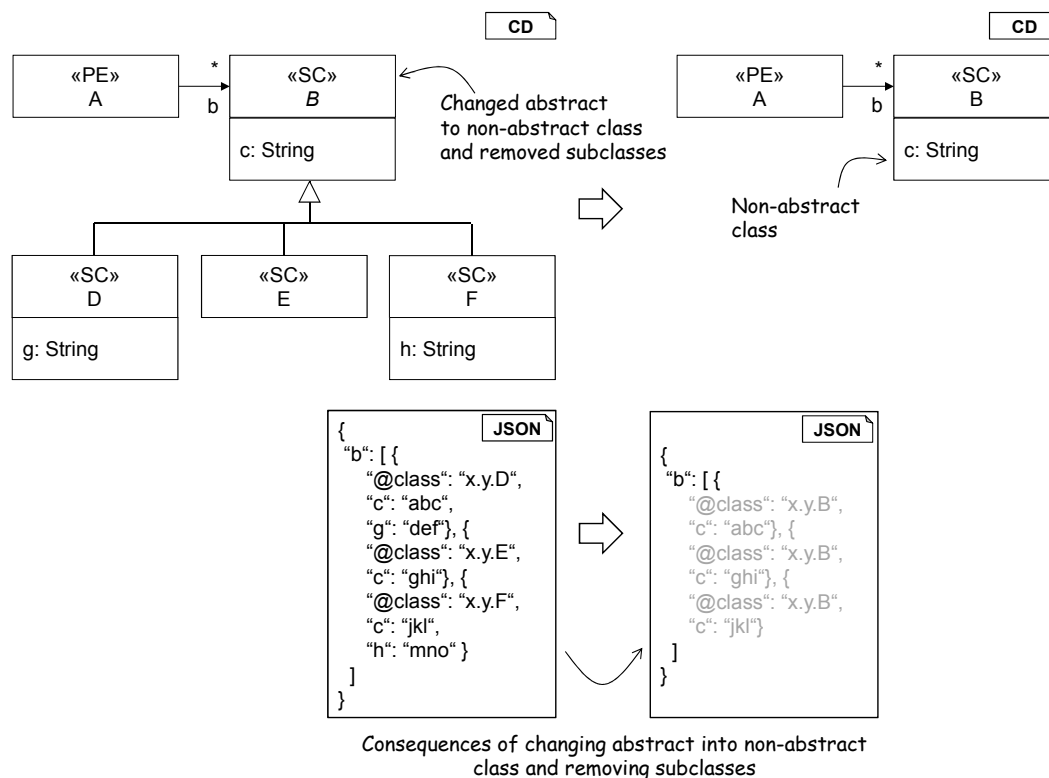


Abbildung 6.31: Datenmigration nach dem Ändern einer abstrakten Klasse hin zu einer nicht-abstrakten Klasse.

2. Bestimmung des vollqualifizierten Namens `$nameSubclass` der Klasse `C`.
3. Falls eine Assoziation `a` direkt auf `P` oder eine Oberklasse davon verweist:
 - a) Ermittlung der Werte `$values`, die zu der Assoziation `a` hinterlegt sind.
 - b) Iteration über alle JSON-Objekte aus `$values`.
 - c) Falls das aktuell betrachtete JSON-Objekt auf eine Instanz vom Typ `P` verweist: Aktualisierung des Werts des Schlüssels `@class` des betrachteten JSON-Objekts auf den Wert `$nameSubclass`. Sämtliche JSON-Objekte, die Instanzen vom Typ `P` waren, werden fortan als Instanzen vom Typ `C` angesehen.
 - d) Hinzufügen sämtlicher Schlüssel zu dem aktuellen JSON-Objekt, die Attribute oder Assoziationen von `C` repräsentieren. Ausgenommen hiervon sind lediglich die Attribute, die von der Klasse `P` oder einer Oberklasse davon in die Klasse `C` verschoben wurden.

Da gemäß Voraussetzung `P` eine konkrete Klasse war, kann eine Assoziation nicht auf Unterklassen von `P` in der alten Version verweisen. Folglich muss dieser Fall nicht betrachtet werden.

Unter der Einschränkung, dass die Klasse $x.y.B$ nur eine Unterklasse hat, verdeutlicht Abbildung 6.31 dieses Szenario. Das rechte Klassendiagramm stellt hierbei die alte und das linke Klassendiagramm die neue Version dar. Für sämtliche JSON-Objekte, die zum Schlüssel b hinterlegt waren, muss folglich der Typ des JSON-Objekts angepasst werden. Ferner wird der Schlüssel g hinzugefügt, unter der Voraussetzung, dass die Klasse $x.y.D$ die neue Unterklasse von Klasse $x.y.B$ ist.

Löschen einer Vererbungsbeziehung

Zur Vereinfachung der Beschreibung wird im Folgenden davon ausgegangen, dass die Klasse C die von dem Löschen der Vererbungsbeziehung betroffene Unterklasse und die Klasse P die davon betroffene Oberklasse ist. Nach dem Löschen einer Vererbungsbeziehung zwischen C und P ist dann keine Datenmigration notwendig, wenn die Klasse C gemäß Algorithmus 6.2 nicht von einer primären Klasse ausgehend erreichbar ist. Sollte eine Datenmigration erforderlich sein, wird eine Fallunterscheidung vorgenommen:

- Angenommen eine Assoziation a verweist auf die Klasse C oder eine Unterklasse davon. Dann wird das Löschen der Vererbungsbeziehung für die zu a hinterlegten JSON-Objekte wie das Löschen der Attribute und Assoziationen der vorherigen Oberklasse P und deren Oberklassen behandelt. Für die Klasse C bzw. deren konkreten Unterklassen sind solche Attribute und Assoziationen davon ausgenommen, die von P oder einer Oberklasse in die jeweilige Klasse verschoben wurden. In Abbildung 6.32 ist ein Beispiel für diesen ersten Fall dargestellt. In diesem Beispiel wird der Schlüssel d aus dem JSON-Objekt für die Instanz C gelöscht, da das Attribut d in der ehemaligen Oberklasse P enthalten war.

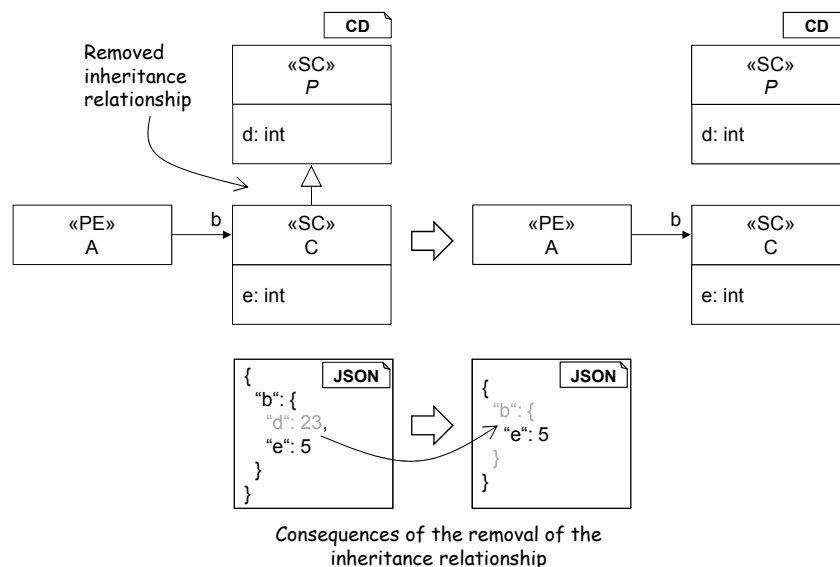


Abbildung 6.32: Datenmigration nach dem Löschen einer Vererbungsbeziehung.

- Liegt stattdessen eine Assoziation *a* auf die Klasse *P* oder eine Oberklasse davon vor, müssen JSON-Objekte, die dem zugehörigen Schlüssel für *a* hinterlegt sind, potentiell entfernt werden. Für diese Konstellation wird das Szenario wie das Löschen der Klasse *C* behandelt:
 - Für einwertige Assoziationen kann der Schlüssel nur dann entfernt werden, wenn der Wert des Schlüssels auf eine Instanz der Klasse *C* verweist. Handelt es sich bei *C* um eine abstrakte Klasse, wird die Instanz gelöscht, wenn auf eine Instanz einer Unterklasse dieser Klasse verwiesen wird.
 - Für mehrwertige Assoziationen muss über sämtliche in der Liste enthaltenen JSON-Objekte iteriert werden und ein Element der Liste wird genau dann gelöscht, wenn das Element auf die Klasse *C* verweist. Auch hier wird auf die Unterklassen von *C* geprüft, falls *C* abstrakt ist.

Im Gegensatz zu dem ersten Fall wird die Modelländerung nicht dem Knoten für die Klasse *C* bzw. deren konkreten Unterklassen zugeordnet, sondern dem Knoten für die Quellklasse der betroffenen Assoziation. Für dieses Szenario ist kein eigenständiges Beispiel angegeben, da das Beispiel für das Löschen einer Klasse diesen Sachverhalt verdeutlicht.

Veränderung der Oberklasse einer Klasse

Zur Vereinfachung der Beschreibung wird im Folgenden davon ausgegangen, dass die Klasse *C* die Klasse ist, deren Oberklasse geändert wurde, *N* die neue Oberklasse und *O* die alte Oberklasse ist. Nach dem Verändern der Oberklasse einer Klasse ist dann keine Datenmigration notwendig, wenn die Klasse *C* gemäß Algorithmus 6.2 nicht von einer primären Klasse ausgehend erreichbar ist. Unter der Voraussetzung, dass eine Datenmigration erforderlich ist, wird eine Fallunterscheidung durchgeführt, um die notwendigen Schritte zur Durchführung einer Datenmigration abzuleiten. Zur Verdeutlichung der Situation, die bei der Modelländerung jeweils vorliegt, werden beispielhafte Klassendiagramme angegeben. In den Beispielen ist jedoch keine Migration beispielhafter JSON-Dokumente dargestellt, da die prinzipiellen Migrationsschritte in den vorherigen Beispielen bereits erläutert wurden.

- Eine Assoziation *a* verweist auf die Klasse *C* oder eine Unterklasse von *C*:
 - Die Klasse *O* ist eine Unterklasse von *N*, siehe Abbildung 6.33: Dieser Fall wird für die Klasse *C* wie das Löschen der Attribute und Assoziationen aus der Klasse *O* und allen Oberklassen davon ausschließlich *N* behandelt.
 - Die Klasse *O* ist eine Oberklasse von *N*, wie in Abbildung 6.34 dargestellt: Dieses Szenario wird für die Klasse *C* wie das Hinzufügen der Attribute und Assoziationen aus der Klasse *N* und allen Oberklassen davon ausschließlich *O* behandelt.
 - Die Klassen *O* und *N* sind nicht Teil der gleichen Vererbungshierarchie: Diese Konstellation wird für die Klasse *C* wie das Löschen der Vererbungsbeziehung zu *O* und das Hinzufügen der Vererbungsbeziehung zu *N* erachtet.
- Eine Assoziation *a* verweist auf die Klasse *O* oder Oberklassen davon (ausgenommen *N*):

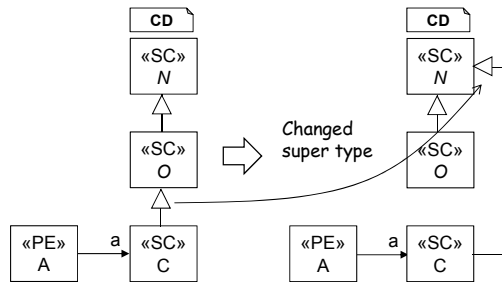


Abbildung 6.33: Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Unterklasse von neuer Oberklasse).

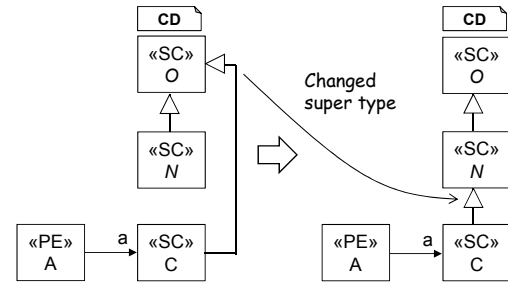


Abbildung 6.34: Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Oberklasse von neuer Oberklasse).

- Die Klasse O ist eine Unterklasse von N, wie in Abbildung 6.35 zu sehen: Dieser Fall wird für die Assoziation a wie das Löschen der Klasse C behandelt, da zu der Assoziation a keine C Instanzen mehr hinterlegt sein können.
- Die Klasse O ist eine Oberklasse von N, wie in Abbildung 6.36 gezeigt: Dieses Szenario wird für die Klasse C wie das Hinzufügen der Attribute und Assoziationen aus der Klasse N und allen Oberklassen davon ausschließlich O behandelt.

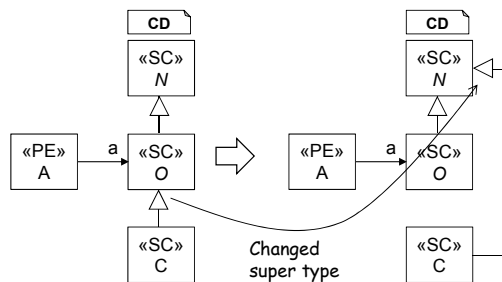


Abbildung 6.35: Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Unterklasse von neuer Oberklasse).

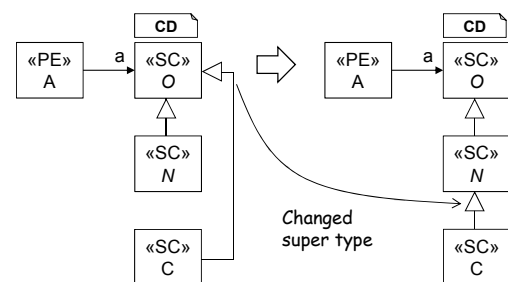


Abbildung 6.36: Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Oberklasse von neuer Oberklasse).

- Die Klassen O und N sind nicht Teil der gleichen Vererbungshierarchie: Diese Konstellation wird für die Assoziation a wiederum wie das Löschen der Klasse C behandelt, da zu der Assoziation a keine C Instanzen mehr hinterlegt sein können.
- Eine Assoziation a verweist auf die Klasse N oder Oberklassen davon:
 - Die Klasse O ist eine Unterklasse von N, illustriert in Abbildung 6.37: Dieser Fall wird für die Klasse C wie das Löschen der Attribute und Assoziationen aus der Klasse O und allen Oberklassen davon ausschließlich N behandelt.

- Die Klasse O ist eine Oberklasse von N, zu sehen in Abbildung 6.38: Ist N bereits eine abstrakte Klasse, ist in diesem Szenario keine Datenmigration notwendig, da zu der neuen Oberklasse lediglich eine neue Unterklasse eingeführt wurde und dies keinen Einfluss auf die bestehenden Daten hat. Andernfalls wird die Änderung bereits durch die Veränderung der Klasse N von einer nicht-abstrakten in eine abstrakte Klasse abgedeckt.

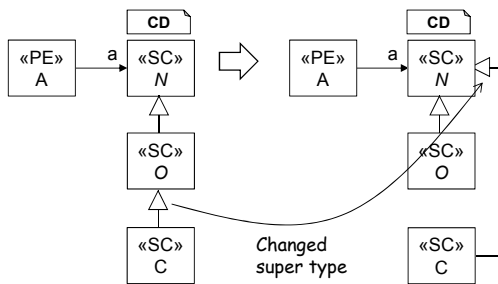


Abbildung 6.37: Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Unterklasse von neuer Oberklasse).

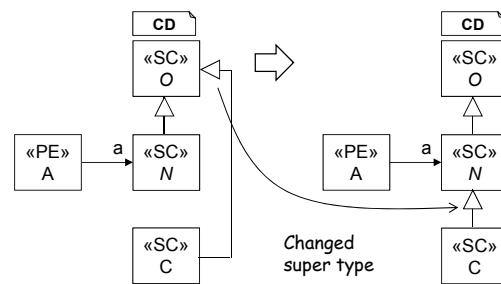


Abbildung 6.38: Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Oberklasse von neuer Oberklasse).

- Die Klassen O und N sind nicht Teil der gleichen Vererbungshierarchie: Dieses Szenario wird analog zum vorherigen Fall gehandhabt.

Welchem Knoten im Graphen die Modelländerung zugeordnet wird, hängt ab davon, auf welche Modelländerung die Veränderung der Oberklasse zurückgeführt wird. Für jede genannte Modelländerung wurde dies zuvor in diesem Abschnitt dargelegt. Beispielsweise wird die Modelländerung dem Knoten für die Quellklasse einer Assoziation zugeordnet, wenn die Modelländerung für die betroffene Assoziation wie das Löschen der Klasse C behandelt wird.

Wird die Modelländerung wie das Hinzufügen von Attributen und Assoziationen behandelt, sind bestimmte Attribute und Assoziationen davon ausgenommen. Wurde ein Attribut oder eine Assoziation von der neuen Oberklasse oder einer Oberklasse davon nach C bzw. eine konkrete Unterklasse von C verschoben, so gilt es für die betroffene Klasse nicht als neu hinzugefügt. Wird die Modelländerung wie das Löschen von Attributen und Assoziationen behandelt, verhält es sich analog. In diesem Fall gilt ein Attribut bzw. eine Assoziation für die Klasse C bzw. eine konkrete Unterklasse davon nicht als gelöscht, wenn es von der alten Oberklasse oder einer Oberklasse davon in die betroffene Klasse verschoben wurde.

6.5.4 Zusammenfassung

Die betrachteten Arten von Modelländerungen werden in Anlehnung an [VWV11] abschließend mit Hinblick auf deren Auswirkungen kategorisiert. Diese Kategorisierung setzt jedoch voraus, dass zuvor eine Vereinfachung des Differenzmodells vorgenommen wurde. Demzufolge liegen manche Arten von Modelländerungen nach der Vereinfachung des Differenzmodells nur noch in spezifischen Situationen vor, wie ausführlich in Abschnitt 6.4 erläutert.

- Konservative Modifikationen verändern ausschließlich die bestehenden Daten, ohne Daten zu entfernen oder hinzuzufügen:
 - Umbenennung eines Attributs oder eines Assoziationsendes
 - Veränderung der Kardinalität eines Assoziationsendes von einwertig zu mehrwertig
 - Umbenennung eines Literals in einer Enumeration
 - Umbenennung einer Klasse
- Verlustbehaftete Modifikationen führen definitiv zu Datenverlusten:
 - Löschen eines Attributs oder einer Assoziation
 - Veränderung der Zielklasse einer Assoziation
 - Löschen eines Literals aus einer Enumeration
 - Löschen einer Klasse
- Potentiell verlustbehaftete Modifikationen können zu Datenverlusten führen:
 - Verschiebung eines Attributs
 - Veränderung der Quellklasse einer Assoziation
 - Veränderung der Kardinalität eines Assoziationsendes von mehrwertig zu einwertig
 - Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse
 - Löschen einer Vererbungsbeziehung
 - Veränderung der Oberklasse einer Klasse
- Additive Modifikationen fügen neue Daten zu den bestehenden Daten hinzu:
 - Hinzufügen eines Attributs oder einer Assoziation
- Potentiell additive Modifikationen können neue Daten zu den bestehenden Daten hinzufügen:
 - Veränderung einer nicht abstrakten Klasse hin zu einer abstrakten Klasse
 - Veränderung der Oberklasse einer Klasse

Gemäß dieser Kategorisierung sind das Löschen einer Vererbungsbeziehung und die Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse lediglich potentiell verlustbehaftet. Der Grund hierfür ist, dass es beim Löschen einer Vererbungsbeziehung zu keinen Datenverlusten kommt, wenn die Oberklassen der betroffenen Klassen keine Attribute und Assoziationen besessen haben. Bei der zweitgenannten Modelländerung kommt es hingegen nicht zu Datenverlusten, wenn die Unterklassen der abstrakten Klassen keine Attribute und Assoziationen besessen haben.

Die dargelegte Kategorisierung kann im Zuge der Datenmigrationen als Ausgangsbasis genommen werden, um die Auswirkungen der Datenmigration besser einschätzen zu können. Umfasst eine Datenmigration ausschließlich konservative Modifikationen, so ist die Datenmigration

unproblematisch, da keine bestehenden Daten verloren gehen. Da verlustbehaftete Modifikationen hingegen Daten löschen, sollte für diese vorab geprüft werden, ob sie tatsächlich durchgeführt werden sollen. Ähnlich verhält es sich für potentiell verlustbehaftete Modifikationen.

6.6 Generierung von Migrationsfunktionalität

Die Generierung der Migrationsfunktionalität erfolgt auf Basis der aus den Pfaden konstruierten Graphen. Die Herangehensweise zur Generierung ist wie folgt:

1. Minimierung der Graphen. Nach der Verarbeitung sämtlicher Modelländerungen des vereinfachten Differenzmodells liegen für die verschiedenen primären Klassen Graphen vor. Diese legen dar, wie von der primären Klasse zu den von den Modelländerungen betroffenen Elementen navigiert werden kann. Zwar kann die Generierung der Migrationsfunktionalität unmittelbar auf diesen Graphen ausgeführt werden, dies kann jedoch in manchen Fällen zu unnötig komplizierter Migrationsfunktionalität führen. Aus diesem Grund wird versucht eine Minimierung der Graphen durchzuführen. Dieser Schritt wird in Abschnitt 6.6.1 detailliert beschrieben.
2. Einlesen der zu migrierenden JSON-Dokumente. Bei der Nutzung der Jackson API und vergleichbaren APIs werden die einzelnen JSON-Dokumente dazu in eine Baumstruktur überführt und das Wurzelobjekt dieses Baums wird als Ergebnis zurückgeliefert.
3. Generierung einer Traversierungslogik. Diese navigiert durch die JSON-Dokumente zu den von den Modelländerungen betroffenen Elementen. Dabei startet sie ausgehend von den Wurzelobjekten der JSON-Dokumente, die im zweiten Schritt bestimmt wurden. Dieser Schritt wird in Abschnitt 6.6.2 im Detail erläutert.
4. Durchführung der Datenmigration. Abhängig von der Art der durchgeführten Modelländerungen, wird abschließend die Migrationslogik zur Datenmigration generiert. Dieser Sachverhalt wird in Abschnitt 6.6.3 beschrieben. Die Verschiebung von Attributen wird auf eine Weise umgesetzt, die von dem Schema zur Handhabung der anderen Arten von Modelländerungen abweicht. Details hierzu sind in Abschnitt 6.6.4 dargelegt. Gleiches gilt für die Veränderung der Quellklasse einer Assoziation, wie in Abschnitt 6.6.5 erläutert.

6.6.1 Minimierung der Graphen

Die Hauptmotivation für die Minimierung der Graphen ist, dass dadurch die Generierung von unnötigen Migrationsanteilen verhindert werden kann. Abbildung 6.39 zeigt ein Beispiel hierfür.

In diesem Beispiel ist mit Knoten A die Modelländerung, die das Löschen der Assoziation `b` ausdrückt, verknüpft. Ferner ist mit Knoten C die Modelländerung, welche die Umbenennung des Attributs `name` der Klasse C in `newName` ausdrückt, verbunden. In diesem Szenario ist es überflüssig, als Erstes die Migrationsoperation zur Umbenennung des Attributs durchzuführen, da die Ausführung der Löschoperation diese überflüssig macht. Wird in einem JSON-Dokument

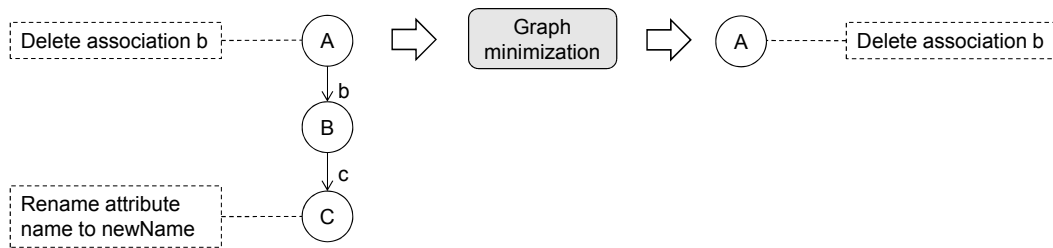


Abbildung 6.39: Beispiel zur Minimierung von Graphen. Die Knoten B und C können aufgrund des Löschens der Kante b ebenfalls entfernt werden.

die Löschoperation ausgeführt, führt das dazu, dass von einem JSON-Objekt zu Knoten A ausgehend der Schlüssel zu Assoziation b gelöscht wird. Anschließend ist es nicht mehr möglich über das Wurzelobjekt zu dem Knoten für die Klasse C zu navigieren.

Wird aus dem Graphen die Migrationslogik generiert, ohne den Graphen vorab zu minimieren, wird unnötigerweise die Logik zur Durchführung der Umbenennungsoperation generiert. Aus diesem Grund wird vor der Generierung der Migrationsfunktionalität geprüft, ob ein Graph minimiert werden kann. Auf diese Weise kann die Komplexität der generierten Migrationsfunktionalität reduziert werden. In Abbildung 6.39 ist der minimierte Graph im rechten Teil der Abbildung zu sehen. Die beiden Knoten B und C konnten entfernt werden, da die Löschoperation zu Assoziation b die Ausführung der Umbenennungsoperation überflüssig macht.

Zur Minimierung eines Graphen werden sämtliche Knoten des Graphen untersucht. Eine Minimierung ist nur dann möglich, wenn mit einem Knoten das Löschen einer Assoziation oder das Löschen einer Klasse verbunden ist. Welche Teile des Graphen gelöscht werden ergibt sich aus der konkret vorliegenden Modelländerung:

- **Löschen einer Assoziation:** Ist mit einem Knoten das Löschen einer Assoziation verbunden, wird die Kante e , die zu der gelöschten Assoziation korrespondiert, aus dem Graphen entfernt. Falls der Zielknoten der Kante e nach dem Löschen keine eingehenden Kanten mehr besitzt, wird dieser Knoten samt der von diesem ausgehenden Kanten ebenfalls entfernt. Falls durch diese Operationen weitere Knoten, die ursprünglich über den Zielknoten der Kante e erreichbar waren, keine eingehenden Kanten mehr besitzen, wird für diese Knoten analog verfahren.
- **Löschen einer Klasse:** Ist mit einem Knoten das Löschen einer Klasse verbunden, wird der Knoten aus dem Graphen entfernt, der zu der gelöschten Klasse korrespondiert. Analog zum vorherigen Fall werden zusätzlich die Kanten entfernt, die von diesem Knoten ausgehen. Besitzen dadurch andere Knoten keine eingehenden Kanten mehr, werden diese auf die gleiche Weise ebenfalls gelöscht. Aufgrund der in Abschnitt 6.5.3 vorgestellten Vorgehensweise zur Datenmigration kann das Löschen einer Klasse nur noch im Graphen vorliegen, falls eine Assoziation auf eine Ober- oder Unterklasse der gelöschten Klasse verweist. In diesen Fällen ist das Löschen der Klasse mit dem Knoten für die Quellklasse der betroffenen Assoziation verbunden. Referenziert beispielsweise eine Klasse A über

eine Assoziation eine Oberklasse P der gelöschten Klasse C , ist das Löschen der Klasse C mit Knoten A verbunden. In dem genannten Beispiel wird also nicht der Knoten A gelöscht, mit dem das Löschen der Klasse C verbunden ist, sondern der Knoten C , da dieser die gelöschte Klasse repräsentiert.

Angewendet auf den linken Graphen in Abbildung 6.39 bedeutet die zuvor beschriebene Vorgehensweise, dass die ausgehende Kante b des Knotens A entfernt wird. Der Zielknoten dieser Kante, Knoten B , besitzt keine weiteren eingehenden Kanten und ist nicht der Startknoten. Dadurch können er und seine ausgehende Kante ebenfalls entfernt werden. Aus den gleichen Gründen kann anschließend auch Knoten C entfernt werden.

Abbildung 6.40 zeigt ein weiteres Beispiel, welches ähnlich zu dem Beispiel aus Abbildung 6.39 ist. Der Unterschied zwischen den beiden Beispielen ist, dass der Knoten A über eine weitere Kante $c2$ zu Knoten C verfügt. Übertragen auf den linken Graphen in dieser Abbildung führt die dargelegte Vorgehensweise dazu, dass ebenfalls die ausgehende Kante b des Knotens A entfernt wird. Darüber hinaus kann wiederum der Zielknoten B gelöscht werden. Demgegenüber kann in diesem Beispiel der mit diesem verbundene Knoten C nicht gelöscht werden, da C noch über den Knoten A erreichbar ist. Aus diesem Grund wird lediglich der Knoten B und dessen eingehende und ausgehende Kante aus dem Graphen entfernt.

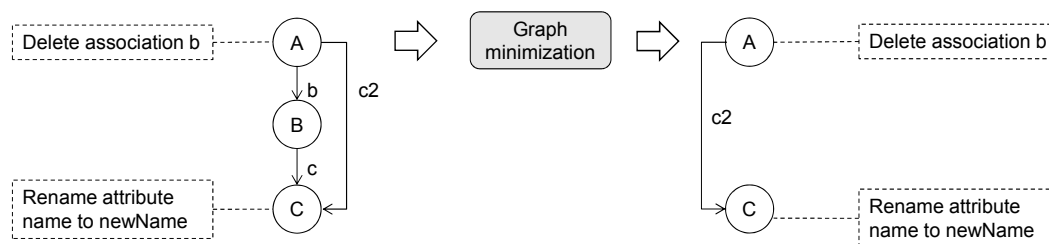


Abbildung 6.40: Minimierung von Graphen. Der auf den gelöschten Knoten folgende Knoten kann nicht gelöscht werden.

6.6.2 Generierung der Traversierungslogik

Die Generierung der Traversierungslogik erfolgt auf Basis der Graphen, die im ersten Schritt minimiert wurden. Ausgehend von diesen Graphen wird eine Logik generiert, die von den Wurzelobjekten der JSON-Dokumente zu den JSON-Objekten, für die Datenmigrationen durchgeführt werden müssen, navigiert. Auf diese Weise kann in allen JSON-Dokumenten, die zu einer bestimmten primären Klasse existieren, zu den von der Datenmigration betroffenen Teilen navigiert werden. Das Generierungsschema wird nachfolgend allgemein dargelegt.

Generierung der Migrationsklassen

In dieser Arbeit wird für jeden Graphen genau eine Migrationsklasse generiert. Die Herangehensweise für die Generierung dieser Migrationsklasse ist wie folgt:

1. Generierung einer Migrationsmethode `performDataMigration()`. Diese Methode wird mit dem Wurzelobjekt des zu migrierenden JSON-Dokuments aufgerufen und ist für die Initiierung der Datenmigration für dieses JSON-Dokument zuständig.
2. Generierung einer Migrationsmethode `performDataMigrationForX()` für jeden Knoten X des Graphen.
3. Falls mit einem Knoten X Modelländerungen verbunden sind, wird in die Migrationsmethode `performDataMigrationForX()` die Migrationslogik zur Handhabung der Datenmigrationen für diese Modelländerungen generiert. In Abschnitt 6.6.3 wird dieser Schritt dargelegt.
4. Für jede Kante z , die zwischen dem Knoten X und dem Knoten Y existiert und die keine Unterklassenkante darstellt, wird in `performDataMigrationForX()` der Wert zu dem Schlüssel z ermittelt. Liegt als Wert ein einzelnes JSON-Objekt vor, wird die Methode `performDataMigrationForY()` mit diesem Objekt als Argument aufgerufen. Andernfalls wird über alle Elemente der Liste iteriert und die einzelnen Objekte werden jeweils als Argumente an die Methode übergeben.
5. Besitzt ein Knoten X eine ausgehende Unterklassenkante zu einem Knoten Y und Y repräsentiert eine konkrete Klasse, muss in der Migrationsmethode `performDataMigrationForX()` sichergestellt werden, dass das aktuell betrachtete JSON-Objekt eine Instanz vom Typ Y darstellt. Ist diese Bedingung erfüllt, wird die Migrationsmethode `performDataMigrationForY()` mit dem aktuellen JSON-Objekt als Argument aufgerufen.
6. Besitzt ein Knoten X hingegen eine ausgehende Unterklassenkante zu einem Knoten Y und Y repräsentiert eine abstrakte Klasse, ist die zuvor beschriebene Vorgehensweise unzulässig. Der Grund dafür ist, dass in der Traversierungslogik auf den vollqualifizierten Namen der Zielklasse geprüft wird und keine Instanz mit dem vollqualifizierten Namen der abstrakten Zielklasse vorliegen kann. Stattdessen wird die Migrationsmethode `performDataMigrationForY()` aufgerufen, wenn eine Instanz einer konkreten Klasse vorliegt, die eine Unterklasse vom Typ Y ist.
7. Für einen Knoten Y , der eine eingehende Unterklassenkante zu einem Knoten X hat, wird die Migrationsmethode `performDataMigrationForX()` mit dem aktuellen JSON-Objekt als Argument aufgerufen. Dadurch wird sichergestellt, dass die Migrationslogik für den Knoten X und die mit diesem verbundenen Knoten ausgeführt wird.
8. Um Endlosschleifen bei der Traversierung der JSON-Dokumente zu vermeiden, werden in die Migrationsmethoden für abstrakte Klassen und Unterklassen von abstrakten Klassen boolesche Parameter generiert. Diese geben Aufschluss darüber, ob eine Migrationsmethode durch einen Knoten einer Oberklasse oder durch einen Knoten einer Unterklasse aufgerufen wurde. Folglich wird eine Migrationsmethode für eine Oberklasse nur aufgerufen, wenn der Aufruf der Methode selbst nicht durch die Oberklasse initiiert wurde. Gleichmaßen wird die Migrationsmethode einer Unterklasse nur dann aufgerufen, wenn der Aufruf der Methode nicht durch eine Unterklasse initiiert wurde.

Beispiel

Das Ergebnis der Anwendung dieser Vorgehensweise auf den Graphen aus Abbildung 6.7 auf Seite 116 ist in Listing 6.1 dargestellt. Der Ausgangsgraph besitzt die Knoten A, B, D und E, wobei lediglich mit dem Knoten E eine Modelländerung verbunden ist. Für jeden dieser Knoten wird eine Migrationsmethode generiert. Innerhalb jeder Migrationsmethode wird der Übergang von dem aktuellen Knoten zu einem nachfolgenden Knoten gehandhabt, so dass innerhalb des JSON-Dokuments vom Wurzelobjekt zu den relevanten Unterobjekten navigiert werden kann. Beispielsweise wird in `performDataMigrationA()` (Zeile 5 – 10) die Traversierung vom Wurzelobjekt zu dem Unterobjekt, welches über den Schlüssel `b` hinterlegt ist, vorgenommen. Hervorzuheben ist hierbei, dass als Argument für den booleschen Parameter `true` übergeben wird, damit die Migrationslogik für die Unterklassen aufgerufen wird. Demgegenüber wird die gleiche Methode in der Migrationsmethode `performDataMigrationForD()` (Zeile 22 – 28) mit dem Argument `false` aufgerufen, da der Aufruf aus der Methode für die Klasse D erfolgt, die eine Unterklasse von B ist.

```
1 public void performDataMigration(JsonNode rootNode) {
2     performDataMigrationA(rootNode);
3 }
4
5 public void performDataMigrationA(JsonNode a) {
6     JsonNode bNode = a.get("b");
7     if (bNode != null) {
8         performDataMigrationForB(bNode, true);
9     }
10 }
11
12 public void performDataMigrationForB(JsonNode b,
13     boolean considerSubclasses) {
14     if (considerSubclasses) {
15         JsonNode classValue = b.get("@class");
16         if (classValue != null && "D".equals(classValue)) {
17             performDataMigrationForD(b, true);
18         }
19     }
20 }
21
22 public void performDataMigrationForD(JsonNode d,
23     boolean calledBySuperclass) {
24     ...
25     if (!calledBySuperclass) {
26         performDataMigrationForB(this, false);
27     }
28 }
29
30 public void performDataMigrationForE(JsonNode f) {...}
```

Listing 6.1: Generat mit Traversierungslogik zu Abbildung 6.6 bzw. Abbildung 6.7.

Der Grund für die Generierung verschiedener Migrationsmethoden ist, dass auf diese Weise Zyklen leicht gehandhabt werden können. Bei zirkulären Graphen kann die gesamte Generierungslogik hingegen nicht sauber innerhalb einer Methode generiert werden.

Die gekürzten Teile in Listing 6.1 folgen dem gleichen Schema und sind daher in Listing 6.1 nicht gezeigt. Da lediglich mit dem Knoten E eine Modelländerung assoziiert ist, wird lediglich in die Migrationsmethode `performDataMigrationForE()` (Zeile 30) Migrationslogik generiert. Dieser Sachverhalt wird im nächsten Unterabschnitt präzisiert.

6.6.3 Generierung der Migrationslogik

Wird bei der Graphtraversierung ein Knoten erreicht, mit dem Modelländerungen verknüpft sind, muss in die zugehörige Migrationsmethode Migrationslogik generiert werden. Diese muss die Datenmigration passend zu den hinterlegten Modelländerungen durchführen. In Abschnitt 6.5 wurde dazu allgemeingültig erklärt, wie für verschiedene Modelländerungen vorgegangen wird, um eine Datenmigration durchzuführen. Die dort dargelegten Schritte werden in eine separate Migrationsmethode generiert. Diese wird innerhalb der Migrationsmethode für den entsprechenden Knoten aufgerufen. Auf diese Weise ist es Entwicklern möglich, die generierte Migration punktuell zu adaptieren. So legt Listing 6.2 die Implementierung der Methode `performDataMigrationForE()` dar, in der die in Listing 6.1 fehlende Datenmigration zur Handhabung der Umbenennung des Schlüssels `name` nach `newName` umgesetzt ist.

```

1 public void performDataMigrationForE(JsonNode e) {
2     performRenameNameToNewNameForE(e);
3 }
4
5 public void performRenameNameToNewNameForE(JsonNode e) {
6     String oldName = "name";
7     String newName = "newName";
8     JsonNode value = e.get(oldName);
9     if (value != null) {
10         ((ObjectNode) e).set(newName, value);
11     }
12     ((ObjectNode) e).remove(oldName);
13 }

```

Listing 6.2: Generat mit Datenmigrationslogik zur Handhabung der Umbenennung des Attributs `name` nach `newName` für den Knoten E.

Durch die Ausführung der Migrationslogik für eine Modelländerung kann allerdings die Ausführung weiterer Migrationslogik beeinträchtigt werden. Abbildung 6.43 zeigt zur Verdeutlichung einen Graphen, bei dem für Knoten C hinterlegt ist, dass das Assoziationsende `d` nach `newD` umbenannt wurde. Ferner ist bei Knoten C vermerkt, dass das Assoziationsende `d` von einer einwertigen zu einer mehrwertigen Assoziation verändert wurde.

Wird die Migrationslogik für beide Modelländerungen in beliebiger Reihenfolge generiert, kann dies zu einer fehlerhaften Datenmigration führen. Dieser Fall tritt in dem Beispiel ein, wenn als Erstes die Logik zur Umbenennung und anschließend die Logik zur Kardinalitätsänderung aufgerufen wird. In diesem Szenario kann die Datenmigration zur Umbenennung des

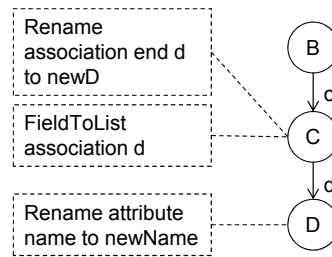


Abbildung 6.43: Graph in dem die Migrationslogik für die zu Knoten C hinterlegten Modelländerungen nicht in beliebiger Reihenfolge ausgeführt werden können.

Schlüssels `d` nach `newD` problemlos ausgeführt werden. Die darauffolgende Migration zur Kapselung des zu dem Schlüssel `d` hinterlegten Werts in einer Liste bleibt jedoch zwangsläufig ohne Auswirkungen. Der Grund dafür ist, dass aufgrund der zuvor durchgeführten Umbenennung des Schlüssels kein Wert mehr zu dem Schlüssel `d` eingetragen sein kann.

Um zu vermeiden, dass die Migrationslogik für eine Modelländerung das Ergebnis der Migrationslogik für andere Modelländerungen beeinflusst, wird die Migrationslogik abhängig von der Art der Modelländerung geordnet. Die Verschiebung eines Attributs und die Veränderung der Quellklasse einer Assoziation sind hiervon ausgenommen, da für diese Fälle separate Migrationsklassen generiert werden. Details hierzu sind in Abschnitt 6.6.4 und Abschnitt 6.6.5 zu finden.

Ordnung der Modelländerungen

In dem vorverarbeitenden Vereinfachungsschritt für das Differenzmodell (siehe Abschnitt 6.4) werden einige Arten von Modelländerungen auf das Löschen eines Attributs oder einer Assoziation und darauffolgend das Hinzufügen eines Attributs oder einer Assoziation zurückgeführt. Eine derartige Ersetzung einer Modelländerung wird dann vollzogen, wenn nicht automatisch abgeleitet werden kann, ob und wie die zugehörigen Daten in die neue Version überführt werden können. In diesen Fällen müssen in einem ersten Schritt die bestehenden Daten zu einem Schlüssel gelöscht werden, bevor anschließend neue Daten hinzugefügt werden können. Die umgekehrte Reihenfolge ist nicht zielführend, da dann die neu hinzugefügten Daten direkt wieder gelöscht werden würden. Aus diesem Grund müssen die Migrationen zu den beiden löschenden Modelländerungsarten vor den beiden additiven Modelländerungsarten ausgeführt werden.

Darüber hinaus existieren Arten von Modelländerungen, die sich nicht gegenseitig beeinflussen können und daher in beliebiger Reihenfolge ausgeführt werden können:

- Veränderung der Zielklasse einer Assoziation
- Löschen eines Literals aus einer Enumeration
- Umbenennung eines Literals aus einer Enumeration
- Löschen einer Klasse

Bei der ersten und letzten Änderung wird ausgehend von einem JSON-Objekt über die Werte iteriert, die einem spezifischen Schlüssel zugeordnet sind. Ist einem Schlüssel lediglich ein JSON-Objekt zugeordnet, wird der Schlüssel aus dem JSON-Objekt entfernt, falls das JSON-Objekt eine Instanz einer bestimmten Klasse repräsentiert. Ist einem Schlüssel hingegen eine Liste von JSON-Objekten zugeordnet, werden sämtliche JSON-Objekte aus der Liste entfernt, die Instanzen einer bestimmten Klasse repräsentieren. Beide Operationen sind unabhängig voneinander und auch unabhängig von den anderen beiden genannten Modelländerungsarten. Die Datenmigrationen zu den anderen beiden Operationen löschen lediglich einen Schlüssel mit einem Enumerationswert oder benennen den Enumerationswert zu einem Schlüssel um.

Die Migrationen zur Handhabung der Umbenennung eines Attributs oder eines Assoziationsendes verändern den Namen eines Schlüssels, auf den in den weiteren Migrationen zugegriffen wird. In Abbildung 6.43 wurde dieses Problem anhand einer Umbenennung und einer Kardinalitätsänderung dargelegt. Damit die vorherigen Migrationen auf dem ursprünglichen Namen operieren können, muss eine derartige Migration als Letztes ausgeführt werden.

Die Migrationen zur Handhabung der Kardinalitätsänderungen verändern den Wert, der einem Schlüssel zugeordnet ist. Dabei wird ein einzelner Wert in einer Liste gekapselt oder aus einer Liste ein einzelner Wert entnommen und dem Schlüssel zugeordnet. Dies kann Einfluss auf die Migrationen zu den verbliebenen Modelländerungen haben, da für diese über einen Schlüssel zu den Werten navigiert werden muss, die einem Schlüssel zugeordnet sind. Aus diesem Grund müssen die Migrationen für Kardinalitätsanpassungen als Vorletztes ausgeführt werden.

Die Migrationen zur Umbenennung von Klassen beeinflussen die Typangaben, die zu JSON-Objekten hinterlegt sind. Zur Durchführung der Migrationen zu den Änderungen in der Vererbungshierarchie und der Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse oder andersherum wird jedoch auf diese Typangaben zugegriffen. Aufgrund dieses Sachverhalts muss die Umbenennung von Klassen nach diesen Migrationen vorgenommen werden, damit die Migrationen nicht durch eine frühzeitige Umbenennung der Klassen beeinträchtigt werden.

Bezüglich der Handhabung des Löschens einer Vererbungsbeziehung und der Veränderung der Oberklasse einer Klasse gibt es keine fest vorgegebene Ausführungsreihenfolge. Der Grund hierfür ist, dass für eine Klasse nicht gleichzeitig die Vererbungsbeziehung gelöscht und die Oberklasse verändert worden sein kann. Stattdessen kann für eine Klasse nur eine der beiden Modelländerungsarten vorliegen. Die Migrationsschritte für die beiden genannten Änderungsarten beeinflussen zudem nicht die Migrationsschritte, die bei der Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse vorgenommen werden oder andersherum. Das gleiche gilt für die Veränderung einer nicht abstrakten Klasse hin zu einer abstrakten Klasse. Somit ist die Ausführungsreihenfolge bei den vier genannten Modelländerungsarten irrelevant, da sich die Operationen nicht gegenseitig beeinflussen.

Nachfolgende Auflistung fasst zusammen, in welcher Reihenfolge die Migrationslogik für welche Art von Modelländerung generiert wird. Sind einem Auflistungspunkt mehrere Arten von Modelländerungen zugeordnet, können diese in beliebiger Reihenfolge ausgeführt werden.

1. Löschen eines Attributs oder einer Assoziation
2. Hinzufügen eines Attributs oder einer Assoziation

3. Veränderung der Zielklasse einer Assoziation, Löschen eines Literals aus einer Enumeration, Umbenennung eines Literals aus einer Enumeration, Löschen eines Pakets oder einer Klasse
4. Veränderung einer abstrakten Klasse hin zu einer nicht abstrakten Klasse, Veränderung einer nicht abstrakten Klasse hin zu einer abstrakten Klasse, Löschen einer Vererbungsbeziehung, Veränderung der Oberklasse einer Klasse
5. Umbenennung eines Pakets oder einer Klasse
6. Veränderung der Kardinalität eines Assoziationsendes von einwertig zu mehrwertig, Veränderung der Kardinalität eines Assoziationsendes von mehrwertig zu einwertig
7. Umbenennung eines Attributs oder eines Assoziationsendes

6.6.4 Generierung der Migrationslogik für die Attributverschiebung

Die Durchführung einer Datenmigration für die Verschiebung eines Attributs erfordert das Kopieren eines Werts aus einem JSON-Objekt und das Übertragen dieses Werts in ein anderes JSON-Objekt. Das konkrete Generierungsschema zur Umsetzung solcher Datenmigrationen ist wie folgt:

- In der Migrationsmethode für den Quellknoten der Verschiebung wird das JSON-Objekt bestimmt, welches zu dem verschobenen Attribut hinterlegt ist. Dieses JSON-Objekt wird zwischengespeichert und darauffolgend wird der Schlüssel zu dem verschobenen Attribut aus dem JSON-Objekt entfernt.
- Falls die Zielklasse von der Quellklasse ausgehend erreichbar ist:
 - Ist ein Knoten X von dem Knoten für die Quellklasse der Verschiebung ausgehend erreichbar und befindet sich auf dem Pfad zu der Zielklasse, wird die Migrationsmethode für X um einen Parameter erweitert, der ein JSON-Objekt aufnehmen kann. Das im ersten Schritt ermittelte JSON-Objekt zu dem verschobenen Attribut wird an die Migrationsmethoden für die Folgeknoten übergeben. Auf diese Weise kann das JSON-Objekt zu dem verschobenen Attribut aus den JSON-Objekten für die Quellklasse an die JSON-Objekte der Zielklasse übermittelt werden.
 - In der Migrationsmethode für den Zielknoten wird zu dem aktuellen JSON-Objekt ein Schlüssel mit dem Namen des verschobenen Attributs hinzugefügt. Als Wert erhält dieser Schlüssel das an die Migrationsmethode übergebene JSON-Objekt für den Wert des verschobenen Attributs.
- Falls die Quellklasse von der Zielklasse ausgehend erreichbar ist:
 - Ist ein Knoten X von dem Knoten für die Zielklasse der Verschiebung ausgehend erreichbar und befindet sich auf dem Pfad zu der Quellklasse, wird die Migrationsmethode für X so generiert, dass die Methode ein JSON-Objekt zurückliefert. Auf diese

Weise kann das JSON-Objekt zu dem verschobenen Attribut von der Migrationsmethode für die Quellklasse der Verschiebung an das JSON-Objekt für die Zielklasse der Verschiebung zurückgegeben werden.

- Angenommen die Klasse X liegt auf dem Pfad von der Zielklasse zu der Quellklasse. Existieren von X ausgehend mehrere Pfade zu dem Knoten für die Quellklasse, ruft die Migrationsmethode zu X mehrere Migrationsmethoden auf, die jeweils JSON-Objekte zurückliefern. Unterscheiden sich diese Rückgabewerte, muss eine Entscheidung getroffen werden, welcher dieser Werte durch die Migrationsmethode zu X zurückgeliefert wird:
 - * Es wird kein Wert übernommen, da nicht eindeutig bestimmt werden kann, welcher Rückgabewert korrekt ist für die Datenmigration.
 - * Es wird ein spezifischer Wert übernommen, beispielsweise der Rückgabewert, der von einem bestimmten Pfad zurückgeliefert wurde.
- In der Migrationsmethode für den Quellknoten wird das im ersten Schritt ermittelte JSON-Objekt zu dem verschobenen Attribut zurückgegeben.
- In der Migrationsmethode für den Zielknoten wird ein neuer Schlüssel für das verschobene Attribut zu dem aktuellen JSON-Objekt hinzugefügt. Als Wert wird der Rückgabewert bzw. einer der Rückgabewerte übernommen, die von den aufgerufenen Migrationsmethoden zurückgeliefert wurden.

Da sich das Generierungsschema für die Verschiebung von Attributen von dem Generierungsschema für die anderen Modelländerungen unterscheidet, werden zur Vereinfachung des Generierungsprozesses separate Migrationsklassen erzeugt. Aus diesem Grund wird bei der Erzeugung der Graphen für jede Attributverschiebung ein eigenständiger Graph verwaltet. Dadurch kann für jede Attributverschiebung die passende Migrationslogik generiert werden. Diese Maßnahme erfordert jedoch, dass im Rahmen der Verschiebung eines Attributs implizit auch der Fall beachtet wird, dass das verschobene Attribut umbenannt worden sein konnte. Diese Modelländerung ist in dem separaten Graphen nicht hinterlegt und wird daher im Zuge der Datenmigration automatisch mit beachtet.

Die daraus resultierenden Migrationsklassen müssen vor den Migrationsklassen für die weiteren Modelländerungen ausgeführt werden, da diese weiteren Modelländerungen die Struktur der Daten beeinflussen können, die für die Verschiebung eines Attributs vorausgesetzt wird.

Beispiel

Die beschriebene Herangehensweise wird nachfolgend anhand eines Beispiels erläutert. Abbildung 6.44 zeigt hierzu einen Graphen, der ausdrückt, dass das Attribut `name` von Klasse `B` in Klasse `D` verschoben wurde. Hervorzuheben ist hierbei, dass die Modelländerung sowohl dem Knoten für die Quellklasse als auch dem Knoten für die Zielklasse der Verschiebung zugeordnet ist. Die Markierung dieser Knoten ist für die Generierung der Migrationsklassen wichtig.

Aufbauend auf diesem Graphen wird die Migrationslogik für die Verschiebung generiert. Die resultierende Migrationsklasse ist in Listing 6.3 gezeigt. In diesem Fall wird in der Migrationsmethode zu Knoten `B` das JSON-Objekt zu dem verschobenen Attribut `name` ermittelt (Zeile 3),

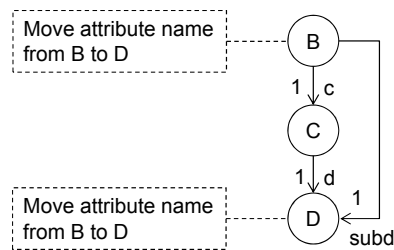


Abbildung 6.44: Graph für die Verschiebung des Attributs name von Klasse B in die Klasse D.

aus dem aktuellen JSON-Objekt entfernt (Zeile 4) und an die Migrationsmethoden für die Folgeknoten übergeben (Zeile 8, 12). Auf diese Weise wird das JSON-Objekt für das verschobene Attribut an die JSON-Objekte für die Zielklasse D übermittelt, so dass das verschobene Attribut zu diesen hinzugefügt werden kann (Zeile 26).

```

1 public void performMigrationForNodeB(JsonNode root) {
2     String orgAttName = "name";
3     JsonNode name = root.get(orgAttName);
4     ((ObjectNode) root).remove(orgAttName);
5
6     JsonNode c = root.get("c");
7     if (c != null) {
8         performMigrationForNodeC(c, name);
9     }
10    JsonNode subd = root.get("subd");
11    if (subd != null) {
12        performMigrationForNodeD(subd, name);
13    }
14 }
15
16 public void performMigrationForNodeC(JsonNode c, JsonNode name) {
17     JsonNode d = c.get("d");
18     if (d != null) {
19         performMigrationForNodeD(d, name);
20     }
21 }
22
23 public void performMigrationForNodeD(JsonNode d, JsonNode name) {
24     if (name != null) {
25         String newAttName = "name";
26         ((ObjectNode) d).set(newAttName, name);
27     }
28 }
  
```

Listing 6.3: Generierte Datenmigrationslogik für die Verschiebung des Attributs von der primären Klasse B in die Zielklasse D.

Darüber hinaus zeigt Listing 6.4 auf Seite 164 die generierte Migrationslogik für den Fall, dass der Wert wieder zurück von der vorherigen Zielklasse in die ursprüngliche Quellklasse verschoben wird. Hierbei wird das JSON-Objekt in der Migrationsmethode zu Knoten D ermittelt (Zeile 33), aus dem aktuellen JSON-Objekt entfernt (Zeile 34) und zurückgegeben (Zeile 35). Dieser Rückgabewert wird in dem JSON-Objekt für die Zielklasse dazu genutzt das verschobene Attribut zu dem aktuellen JSON-Objekt hinzuzufügen (Zeile 18).

```

1 public void performMigrationForNodeB(JsonNode root) {
2     JsonNode name = null;
3     JsonNode c = root.get("c");
4     if (c != null) {
5         name = performMigrationForNodeC(c);
6     }
7
8     JsonNode subd = root.get("subd");
9     if (subd != null) {
10        JsonNode nameFromSubd = performMigrationForNodeD(subd);
11        if (!nameFromSubd.equals(name)) {
12            //Here we need to decide whether to
13            //choose value, discard value etc.
14        }
15    }
16    String newAttName = "name";
17    if (name != null) {
18        ((ObjectNode) root).set(newAttName, name);
19    }
20 }
21
22 public JsonNode performMigrationForNodeC(JsonNode c) {
23     JsonNode d = c.get("d");
24     if (d != null) {
25         return performMigrationForNodeD(d);
26     }
27
28     return null;
29 }
30
31 public JsonNode performMigrationForNodeD(JsonNode d) {
32     String orgAttName = "name";
33     JsonNode name = d.get(orgAttName);
34     ((ObjectNode) d).remove(orgAttName);
35     return name;
36 }

```

Listing 6.4: Generierte Datenmigrationslogik für die Verschiebung des Attributs von der Quellklasse D in die Zielklasse B.

6.6.5 Generierung der Migrationslogik für die Veränderung der Quellklasse einer Assoziation

Die Handhabung der Veränderung der Quellklasse einer Assoziation ist konzeptuell ähnlich zu der Vorgehensweise bei der Verschiebung eines Attributs. Dadurch ist auch das Schema zur Generierung der Migrationsklassen in diesen Fällen vergleichbar. So werden bei der Erzeugung von Graphen ebenfalls für jede Veränderung der Quellklasse einer Assoziation eigenständige Graphen erzeugt und auf Basis dieser Graphen eigenständige Migrationsklassen generiert. Zudem sind auch diese Migrationsklassen so ausgelegt, dass etwaige Umbenennungen der betroffenen Assoziationsenden automatisch beachtet werden. Aufgrund der Ähnlichkeit zur Herangehensweise bei der Verschiebung eines Attributs, werden die weiteren Details zum Generierungsschema nicht betrachtet.

Die wesentliche Frage in diesem Kontext ist, wann die Migrationsklassen für die Veränderungen der Quellklasse von Assoziationen ausgeführt werden. Dies gilt insbesondere mit Hinblick darauf, dass diese Migrationsklassen die Struktur der Daten beeinflussen können, auf denen die Migrationen für die weiteren Modelländerungen arbeiten und anders herum. Zur Adressierung dieses Problems wird in dieser Arbeit die Festlegung getroffen, dass die Migrationen für die Veränderungen der Quellklasse von Assoziationen vor den anderen Modelländerungen ausgeführt werden. Diese Festlegung alleine genügt jedoch nicht, um die beschriebene Problematik zu lösen. Nachdem auf Basis eines Graphen für die Veränderung der Quellklasse einer Assoziation eine Migrationsklasse generiert wurde, wird die Veränderung der Quellklasse zudem auf Graphebene in die anderen Graphen abgebildet. Konkret wird in den anderen Graphen somit die Kante für die Assoziation zur alten Quellklasse entfernt und eine neue Kante für die Assoziation von der neuen Quellklasse ausgehend eingefügt. Durch diese Änderung der Graphen können die weiteren Migrationen für die angepasste Struktur der Daten generiert werden. Wurden also bereits Daten aufgrund einer Migration für die Veränderung der Quellklasse einer Assoziation verändert, können die weiteren Migrationen auf diesen aktualisierten Daten korrekt arbeiten.

Abbildung 6.47 illustriert die zuvor geschilderte Anpassung eines Graphen. Der gezeigte Graph entspricht dem Graphen für das Beispiel in Abbildung 6.22 auf Seite 134, wobei zusätzlich angenommen wird, dass das Attribut *g* der Klasse *F* gelöscht wird. Aus dem ursprünglich vorliegenden Graphen, gezeigt auf der linken Seite, wird die Kante *e* von Knoten *C* zu Knoten *F* gelöscht und eine Kante *e* von Knoten *A* zu Knoten *F* eingefügt. Die Migrationslogik wird anschließend für den angepassten Graphen, auf der rechten Seite gezeigt, generiert.

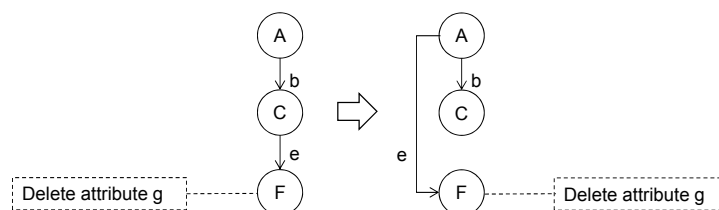


Abbildung 6.47: Graph wird angepasst, da die Quellklasse der Assoziation *e* von der Klasse *C* in die Klasse *A* geändert wurde.

6.6.6 Durchführung manueller Anpassungen an generierter Migrationsfunktionalität

Wie zuvor erläutert, wird für die relevanten Modelländerungen automatisch Migrationsfunktionalität generiert. Die Entwickler müssen diese allerdings nicht unverändert ausführen, sondern haben auch die Möglichkeit sie zu adaptieren und zu erweitern.

Zur Integration der generierten und der handgeschriebenen Migrationslogik wird in dieser Arbeit eine angepasste Variante des Generation Gap Musters [GHK⁺15a, GHK⁺15b] eingesetzt. Dieses Integrationsmuster sieht vor, dass eine Basisklasse generiert und eine Unterklasse davon manuell angelegt wird. In dieser Unterklasse kann die generierte Logik überschrieben werden.

Ein Aspekt von dem bisher abstrahiert wurde betrifft die Erzeugung von Instanzen der Migrationsklassen. Zu diesem Zweck wird in jede Migrationsklasse eine statische Methode generiert, welche eine Instanz der generierten Klasse zurückliefert. Falls Entwickler jedoch eine handgeschriebene Erweiterungsklasse bereitstellen, liefert diese Methode nicht länger eine Instanz der generierten sondern der handgeschriebenen Erweiterungsklasse zurück. Dieser Sachverhalt ist in Abbildung 6.48 illustriert.

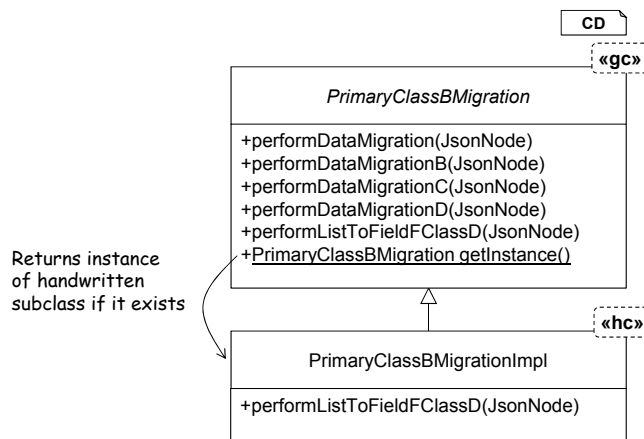


Abbildung 6.48: Generierte Migrationsklasse verwendet handgeschriebene Erweiterungsklasse zur Durchführung der Datenmigration.

Die Umsetzung dieses Ansatzes setzt voraus, dass die handgeschriebene Erweiterungsklasse nach einem festen Namensschema benannt ist. In dieser Arbeit wird konkret vorausgesetzt, dass der Name der handgeschriebenen Erweiterungsklasse den Namen der generierten Basisklasse mit angehängtem Suffix `Impl` besitzen muss. Dadurch kann bei der Generierung der Migrationsklassen geprüft werden, ob eine handgeschriebene Erweiterungsklasse zu einer generierten Basisklasse existiert. In diesem Fall kann in der Methode dann entsprechend eine Instanz dieser Erweiterungsklasse zurückgeliefert werden. Alternativ dazu kann angenommen werden, dass immer eine Erweiterungsklasse angelegt wird. Dadurch ist die zusätzliche Prüfung auf die Existenz der Erweiterungsklasse überflüssig. Diese Varianten des Generation Gap Musters samt der Vor- und Nachteile werden in [GHK⁺15a, GHK⁺15b] genauer vorgestellt.

6.7 Diskussion

6.7.1 Allgemeine Diskussion

Die Herangehensweise zur Generierung von Migrationsfunktionalität ist in dieser Arbeit auf Daten des JSON-Formats ausgelegt. Dies ist jedoch keine problematische Restriktion, da JSON ein weit verbreitetes Datenformat darstellt, welches in vielen Gebieten Anwendung findet [Smi15]. Darüber hinaus ist die Vorgehensweise nicht spezifisch für JSON, sondern kann auf andere textbasierte Datenformate, die ebenfalls hierarchisch aufgebaut sind, übertragen werden. Zu diesen zählt beispielsweise XML.

Die in dieser Arbeit vorgestellte Vorgehensweise zur Generierung von Migrationsfunktionalität ermöglicht die Migration von Daten, die konform zu einer Version v_a sind, hin zu Daten, die konform zu einer beliebigen anderen Version v_n sind. In der Praxis ist diese theoretische Flexibilität allerdings nur eingeschränkt sinnvoll. Kommuniziert beispielsweise ein neuer Client in Version v_n mit einem älteren Server in Version v_a , so kann der Client zwar Daten von dem Server beziehen, unter Umständen decken diese Daten allerdings wichtige Bereiche des neuen Clients nicht vollständig ab. Dies gilt vor allem dann, wenn das Client-Server System nach Version v_a in größerem Maße umstrukturiert wurde. Die von dem Server zurückgelieferten Daten sind dann zwar konform zu dem Datenschema des Clients, der Client kann dann aber nur bedingt mit diesen Daten weiterarbeiten, da wichtige Daten fehlen können. In der Praxis werden hingegen die Client- und Server-Versionen üblicherweise nicht maßgeblich voneinander abweichen. Vollkommen unproblematisch für die praktische Anwendung sind lediglich konservative Datenmigrationen, wie in Abschnitt 6.5.4 beschrieben. Alle anderen Modifikationen können zu Problemen führen.

Alternativ zu der Generierung der Migrationsfunktionalität auf Basis der Graphen, ist es möglich eine generische Implementierung für die Datenmigrationen bereitzustellen. Eine solche Implementierung kann einen gegebenen Graphen automatisch traversieren und bei Erreichen eines Knotens mit Modelländerungen die zugehörigen Migrationsoperationen ausführen. Der Vorteil einer derartigen Vorgehensweise besteht darin, dass die generische Implementierung für alle Graphen wiederverwendet werden kann. Nachteilig ist demgegenüber, dass auf Basis der generischen Implementierung nicht direkt offensichtlich ist, welche Stelle eines JSON-Dokuments zu einem spezifischen Zeitpunkt verarbeitet wird. Dies kann mit Hinblick auf die Erkennung von Fehlern problematisch sein. Der größte Nachteil besteht jedoch darin, dass es umständlicher ist handgeschriebene Erweiterungen bereitzustellen, die von dem Standardschema abweichen. Bei der Generierung der Migrationsfunktionalität können auf einfache Weise dedizierte Teile durch handgeschriebene Implementierungen überschrieben werden. Dies ist in dieser Arbeit der Hauptgrund für die Generierung anstelle der Nutzung einer generischen Implementierung.

6.7.2 Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen

Der in diesem Kapitel vorgestellte Ansatz lässt sich ohne viel Aufwand in den Softwareentwicklungsprozess eines produktiv eingesetzten Softwaresystems integrieren, da der Ansatz automatisch Migrationsfunktionalität auf Basis der erkannten Modelländerungen generiert. Ein zentra-

ler Aspekt bei dieser Integration ist, dass die Entwickler die Möglichkeit haben, die generierte Migrationsfunktionalität gezielt anzupassen. Zu diesem Zweck können sie sowohl die generierte Traversierungslogik als auch die generierte Logik zur Durchführung der einzelnen Datenmigrationen über die Bereitstellung von Unterklassen zu den generierten Klassen manuell anpassen. Auf diese Weise können Abweichungen von dem Standardschema umgesetzt und beispielsweise auch weitere erforderliche Datenanpassungen, die nicht aus den Modelländerungen resultieren, in den Migrationsprozess integriert werden.

Die in Abschnitt 6.5.4 vorgestellte Klassifikation der Auswirkungen der Modelländerungen kann die Entwickler zudem bei der Anwendung der resultierenden Datenmigrationen unterstützen. Auf Basis dieser Klassifikation kann automatisch abgeleitet werden, ob die Datenmigrationen die bestehenden Daten auf jeden Fall beibehalten, also konservative Modifikationen durchführen, oder ob sie beispielsweise zu Datenverlusten führen können. Im erstgenannten Fall können die Datenmigrationen bedenkenlos durchgeführt werden. Im zweitgenannten Fall möchten die Entwickler hingegen unter Umständen prüfen, ob die Daten in dem jeweils vorliegenden Fall eventuell zumindest teilweise doch beibehalten werden können.

Weiterhin ist es über Delta-Benutzervorgaben möglich, für ausgewählte Modellelemente die Ergebnisse der automatischen Differenzberechnung zu beeinflussen. Dies ist ein weiterer Faktor, der die Anwendung des Ansatzes vereinfacht. Dies ist dann wichtig, wenn fehlerhafte Modellunterschiede als Ergebnis gemeldet werden. Ohne Delta-Benutzervorgaben werden unter Umständen fehlerhafte Datenmigrationen generiert, durch die es beispielsweise zu vermeidbaren Datenverlusten kommen kann.

Die praktische Anwendung des Ansatzes wird in Abschnitt 6.9 im Rahmen einer Fallstudie verdeutlicht.

6.8 Verwandte Arbeiten

Die Problemstellung der Koevolution von Artefakten ist auf viele Bereiche übertragbar und nicht beschränkt auf die Evolution von Daten [Läm04, VV08]. Beispielsweise gibt es zahlreiche Untersuchungen dazu, wie Änderungen von Metamodellen an die Modelle propagiert werden können, die konform zu dem geänderten Metamodell sind [Wac07, CREP08, WKS⁺10, Her11, MV11]. Ferner müssen Programme migriert werden, wenn sich die Programmiersprache oder die DSL, in der sie geschrieben wurden, ändert [PJ07a, PJ07b]. Im Folgenden liegt der Fokus jedoch auf Ansätzen zur Koevolution im Kontext der Daten Evolution.

Die in Abschnitt 6.5.4 vorgestellte Kategorisierung von Migrationsaktivitäten ist durch die Arbeit [VWV11] motiviert. Dort werden Schema-Modifikationen, konservative Modifikationen sowie verlustbehaftete Modifikationen vorgeschlagen. Schema-Modifikationen werden in dieser Arbeit nicht betrachtet, da für die Daten kein JSON-Schema existiert.

Der größte Schwerpunkt der bisherigen Arbeiten in diesem Bereich liegt in der Schema- und Datenmigration für Datenbanken [ALP91, Rod92, Rod95, DLZ03, CMZ08, VWV11, CMDZ13]. In diesen Ansätzen werden auf Basis von Schemaänderungen üblicherweise SQL-Skripte generiert, welche die Schema- und die Datenmigration durchführen. Hervorzuheben hierbei ist, dass in [CMZ08, CMZ09, VWV11, CMDZ13] die Spezifikation der relevanten Schemaänderungen

über eine DSL erfolgt. In diesen Fällen müssen die Entwickler folglich die Änderungsoperationen explizit modellieren.

Eine Verallgemeinerung dieser Idee wird in [HHK⁺13, HHK⁺15] vorgestellt. Konkret wird hierbei eine Methodik dargelegt, die zeigt, wie allgemein vorgegangen werden kann, um zu einer textuellen Sprache eine Delta-Sprache zu entwickeln. Eine derartige Delta-Sprache bietet Modifikationsoperatoren an, über die verschiedenartige Änderungen von Modellen der zugehörigen textuellen Sprache definiert werden können. Die resultierende Delta-Sprache kann somit, abhängig von der ursprünglichen textuellen Sprache, ebenfalls zur Definition von Schema-modifikationen bzw. Datenmigrationen genutzt werden. So basiert beispielsweise [Loo17] auf der in [HHK⁺13, HHK⁺15] vorgestellten Methodik und führt eine klassendiagrammspezifische Delta-Sprache ein. Diese Delta-Sprache ermöglicht die Definition von Delta-Operationen, über die Modifikationen von Klassendiagrammen beschrieben werden können. Analog zu dieser Arbeit werden auch in [Loo17] Klassendiagramme zur Modellierung von Datenmodellen genutzt. Darüber hinaus wird auf Basis der modellierten Delta-Operationen eine Infrastruktur generiert, die es ermöglicht Objektdiagramme zu migrieren. Da die persistent gespeicherten Daten über Objektdiagramme beschrieben werden, können über diese Infrastruktur folglich die Datenmigrationen durchgeführt werden.

Im Gegensatz zu diesen und weiteren Arbeiten, die im Folgenden noch genauer vorgestellt werden, basiert der Ansatz dieser Arbeit darauf, die durchgeführten Änderungen aus den Modellversionen des Datenmodells automatisch abzuleiten. Eine Erweiterung des Ansatzes um eine DSL zur Definition von Änderungsoperatoren ist jedoch problemlos möglich.

Neben dem Gebiet der Koevolution eines Datenbankschemas und des Datenbankinhalts legen einige Arbeiten auch einen Fokus auf spezielle Typen von Daten, z.B. auf die Migration von XML-Dokumenten nach Änderungen des XML-Schemas. Derartige Ansätze sind für diese Arbeit relevant, da XML-Dokumente ebenfalls textbasierte hierarchische Dokumente darstellen wie JSON-Dokumente. So wird in [DLRZ05] ein Ansatz vorgestellt, in dem die Struktur von Daten über Klassendiagramme modelliert ist, aus denen XML-Schemata erzeugt werden. Änderungen der Klassendiagramme werden auf die XML-Schemata und auf die XML-Dokumente zu diesen Schemata propagiert. Die durchgeführten Änderungen werden nicht über eine nachträgliche Unterschiedsberechnung erkannt, wie in dieser Arbeit, sondern über die explizite Bereitstellung von Änderungsoperationen. Diese Änderungsoperationen werden intern auf verschiedene Änderungsprozeduren abgebildet, wie das Hinzufügen eines neuen Kindelements oder das Hinzufügen eines Schlüssels. Motiviert durch diesen Ansatz, werden auch in dieser Arbeit einige Änderungsoperationen auf andere Änderungsoperationen abgebildet, wie in Abschnitt 6.4 erklärt. Infolgedessen muss nicht für sämtliche Änderungsoperationen Migrationslogik generiert werden. Sowohl in dieser Arbeit als auch in [DLRZ05] sind Änderungen von Klassendiagrammen die Treiber für die Generierung von Datenmigrationen. Ein wesentlicher Unterschied zu dieser Arbeit besteht darin, dass die Benutzer selbst die Änderungsoperationen angeben und das in diesen die Pfade zu den betroffenen Elementen vermerkt sind. In dieser Arbeit werden hingegen die Pfade zu den betroffenen Elementen automatisch ermittelt, was für den Benutzer in geringerem Aufwand resultiert. Darüber hinaus wird in dieser Arbeit noch eine Minimierung vorgenommen, so dass die zu generierende Migrationsfunktionalität reduziert werden kann.

In [Kle07] wird ein Ansatz zur Evolution von XML-Dokumenten vorgestellt, der auf der

Modellierung eines konzeptuellen Modells basiert, aus dem dann ein XML-Schema abgeleitet wird. In diesem Ansatz werden alle Änderungen eines Modellierers an dem konzeptuellen Modell aufgezeichnet. Diese aufgezeichneten Änderungsoperationen werden, falls möglich, minimiert. Eine Minimierung wird beispielsweise durchgeführt, wenn mehrere aufgezeichnete Änderungsoperationen zusammengefasst werden oder sich gegenseitig aufheben können. Die Idee zur Minimierung der Graphen in dieser Arbeit ist hierdurch motiviert, auch wenn sich die Minimierungen unterscheiden. Aus der minimierten Menge von Änderungsoperationen werden Anweisungen zur Schema- und Daten Evolution generiert. Die Migration der XML-Dokumente erfolgt schließlich unter Nutzung von XML-Aktualisierungssprachen. Ein wesentlicher Unterschied zu dieser Arbeit besteht in der Aufzeichnung der Änderungsoperationen wie in der operationsbasierten Modelldifferenzberechnung [LO92, SZN04, HK10, KHW10]. Zudem setzt der Ansatz voraus, dass Aktualisierungssprachen existieren, so dass die Dokumente durch diese migriert werden können. In dieser Arbeit werden die durchgeführten Modelländerungen hingegen nachträglich bestimmt und darauf aufbauend die Pfade in den JSON-Dokumenten abgeleitet. Zudem lassen sich die entwickelten Migrationsschritte auch ohne dedizierte Aktualisierungssprache umsetzen.

Darüber hinaus existieren Arbeiten zur Daten Evolution für NoSQL-Datenbanken. Diese sind für diese Arbeit relevant, da die meisten NoSQL-Datenbanken JSON als primäre Datenrepräsentation nutzen. In [SKS13] wird eine Evolutionssprache zur Modellierung von Schemaevolutionsoperationen für NoSQL-Datenbanken vorgestellt. Als Teil der Evolutionsoperationen muss der Modellierer die Pfade zu den betroffenen Elementen, wie bei den zuvor genannten Arbeiten, explizit angeben. Die Sprache bietet darüber hinaus die Möglichkeit, Bedingungen zu formulieren, so dass eine Evolutionsoperation nur für bestimmte Datensätze ausgeführt wird. In [SKS15] wird mit Cleager ein zugehöriges Framework vorgestellt, welches die Evolutionsoperationen auf der Google Cloud Platform [Goo] ausführen kann. Für den Anwendungsfall der Datenbankmigrationen ließen sich die erkannten Pfade in dieser Arbeit zumindest partiell auf die Evolutionsoperationen aus [SKS13, SKS15] abbilden und so die Datenmigrationen durchführen. Dazu müssten nach dem in dieser Arbeit gezeigten Schema aber erst die Pfade automatisch bestimmt werden. Für den Anwendungsfall der Datenmigrationen bei der Client/Server Kommunikation ist dies jedoch nicht möglich, da die Daten dort nicht in einer JSON-Datenbank vorliegen.

Ein elementarer Unterschied zwischen dieser Arbeit und den bestehenden Arbeiten liegt somit in der automatischen Ermittlung der Pfade zu den betroffenen Modellelementen. Darüber hinaus gibt es nur vereinzelt Ansätze zur Minimierung der zu generierenden Migrationsfunktionalität. Zudem können die dargelegten Schritte zur Umsetzung der Datenmigrationen auf andere hierarchische Dokumentenarten übertragen werden.

6.9 Fallstudie in der Industrie

6.9.1 Ziele und Forschungsfragen

Das Ziel dieser Fallstudie ist es zu validieren, dass der vorgestellte Ansatz in einem realen Softwaresystem zur Durchführung der Datenmigrationen anwendbar ist. Zu diesem Zweck wurden die folgenden Forschungsfragen abgeleitet:

- **RQ1:** Ist es praktikabel den Ansatz für Datenmigrationen in einem realen Softwaresystem anzuwenden?
- **RQ2:** Führt die Ermittlung der Pfade zu einer Verbesserung der Ausführungszeit der Datenmigrationen im Vergleich zur Datenmigration ohne Pfadermittlung?
- **RQ3:** Führt die Minimierung der Graphen vor der Code-Generierung in realen Migrationsszenarien tatsächlich zu einer Reduktion der Komplexität der generierten Migrationsfunktionalität?

6.9.2 Aufbau und Gegenstand der Studie

Der Ansatz wurde in der Fallstudie auf Daten angewendet, die für Authoring erstellt wurden. Zum Zeitpunkt der Durchführung dieser Fallstudie nutzte Authoring noch eine relationale Datenbank sowie den Serialisierungsmechanismus von Java für die Client-Server-Kommunikation. Um den Ansatz dennoch evaluieren zu können, wurden die bestehenden Daten in einem Vorverarbeitungsschritt eingelesen und durch Jackson nach JSON serialisiert. Der Ansatz wurde daraufhin auf diese JSON-Dokumente angewendet. Insgesamt wurden auf diese Weise circa 7000 JSON-Dokumente erzeugt.

Zur Beantwortung der ersten Forschungsfrage wurde die Migrationsfunktionalität für die drei wichtigsten Klassendiagramme zu den letzten fünf Authoring-Releases generiert. Diese Klassendiagramme enthielten zwischen 100 und 300 Klassen. Konkret wurde die Migrationsfunktionalität jeweils für den Übergang von einem Release hin zum Vorrelease generiert. Als Startpunkt für diese Datenmigrationen dienten die initial erzeugten JSON-Dokumente zu dem aktuell vorliegenden Release. Diese Daten wurden schrittweise hin zu den älteren Versionen migriert. Abschließend wurden die JSON-Dokumente nach der Ausführung der verschiedenen Migrationsklassen manuell analysiert. In dieser Analyse wurden JSON-Dokumente zu verschiedenen primären Klassen und mit unterschiedlichen Größen betrachtet.

Zur Beantwortung der zweiten Forschungsfrage wurde die Generierung der Migrationsfunktionalität so angepasst, dass von dem Wurzelobjekt eines JSON-Dokuments ausgehend erschöpfend sämtliche Pfade innerhalb des JSON-Dokuments verfolgt wurden. Sobald ein JSON-Objekt gefunden wurde, für das eine Datenmigration durchzuführen war, wurde diese Migration wie üblich durchgeführt. Zum Vergleich der beiden Migrationsansätze wurden die Migrationsklassen in den beiden Varianten für die Versionsübergänge ausgeführt, die in der ersten Forschungsfrage zuvor beschrieben wurden. Die Ausführungszeiten wurden dabei protokolliert und miteinander verglichen. Die Experimente wurden jeweils auf einem Dell Latitude E3620 (8GB RAM; Intel i7-2620M 2.7 Ghz) Rechner ausgeführt.

Die dritte Forschungsfrage wurde beantwortet, in dem analysiert wurde, ob in den vorliegenden Fällen eine Minimierung der Graphen durchgeführt wurde.

6.9.3 Ergebnisse und Diskussion

Forschungsfrage RQ1

RQ1: Ist es praktikabel den Ansatz für Datenmigrationen in einem realen Softwaresystem anzuwenden?

Für die erste Forschungsfrage wurden insgesamt 40 Migrationsklassen für primäre Klassen generiert. Von den 113 Migrationsoperationen, die in diesen Migrationsklassen durchgeführt wurden, waren insgesamt circa 55 Prozent verlustbehaftete und circa 45 Prozent konservative Migrationen. Der Grund für diesen hohen Anteil an verlustbehafteten Migrationen ist, dass die Migrationen von der aktuellen Version hin zu den älteren Versionen generiert wurden. Im Zuge der Evolution von Authoring werden vor allem additive Änderungen durchgeführt. Werden nun Migrationen von der neuen zu älteren Versionen durchgeführt, werden folglich zahlreiche löschende Änderungen vorgenommen. Diese führen schließlich zu verlustbehafteten Migrationen.

Für die (potentiell) additiven Migrationsoperationen wurden hingegen keine Migrationen generiert. Dies ist darin begründet, dass es durch die Nutzung des Frameworks Jackson nicht notwendig ist, Schlüssel zu allen Attributen und Assoziationen einer Klasse zu hinterlegen. Stattdessen initialisiert Jackson diese mit den Standardwerten aus Java. In den betrachteten Fällen konnten für die neuen Schlüssel zudem keine sinnvollen Initialwerte eingetragen werden.

In der darauffolgenden Analyse der migrierten Daten wurde manuell untersucht, ob die Migrationsoperationen korrekt auf die untersuchten JSON-Dokumente übertragen wurden. Diese Analyse ergab keine Auffälligkeiten. Stattdessen wurden die identifizierten Migrationsoperationen korrekt durchgeführt.

Insgesamt legen diese Ergebnisse nahe, dass das in dieser Arbeit entwickelte Generierungsschema praktikabel eingesetzt werden kann, da die untersuchten Daten reale Kundendaten waren und die reale Versionshistorie untersucht wurde.

Forschungsfrage RQ2

RQ2: Führt die Ermittlung der Pfade zu einer Verbesserung der Ausführungszeit der Datenmigrationen im Vergleich zur Datenmigration ohne Pfadermittlung?

Die Ausführungszeit für die schrittweise Anwendung der verschiedenen Migrationsklassen auf die verschiedenen JSON-Dokumente betrug in beiden Varianten circa zehn Sekunden. Die Ermittlung der Pfade und die Steuerung der Traversierung durch diese Pfadermittlung hat sich folglich nicht positiv auf die Ausführungszeit der Datenmigrationen ausgewirkt.

Aufgrund dieses Ergebnisses wurden die Ausführungszeiten in beiden Szenarien genauer untersucht, wie in Tabelle 6.49 aufgeführt. Diese Analyse führte zu der Erkenntnis, dass von den zehn Sekunden Gesamtzeit etwas weniger als vier Sekunden für das Parsen der JSON-Dokumente durch Jackson benötigt wurden. Dieser Schritt ist notwendig, da die Migrationsklassen auf dem geparsten JSON-Objektbaum arbeiten. Mit etwas weniger als sechs Sekunden

wurde der Großteil der Ausführungszeit hingegen für das Schreiben der JSON-Dokumente auf Basis der veränderten JSON-Objektbäume benötigt. Demgegenüber war der Anteil der Ausführungszeit der eigentlichen Migrationsfunktionalität an der gesamten Ausführungszeit mit circa 350 Millisekunden sehr gering. Folglich bot sich in diesem Kontext nicht viel Optimierungspotential durch die Steuerung der Traversierung der JSON-Dokumente.

	Unter Steuerung der Traversierung	Ohne Steuerung der Traversierung
Parsen der JSON-Dokumente	3826 Millisekunden	3943 Millisekunden
Schreiben der geänderten Objektbäume	5978 Millisekunden	5922 Millisekunden
Ausführung Migrationsfunktionalität	346 Millisekunden	367 Millisekunden
Gesamtzeit	10150 Millisekunden	10232 Millisekunden

Tabelle 6.49: Übersicht über die Ausführungszeit der Datenmigrationen mit und ohne Steuerung der Traversierung.

Diese Ergebnisse zeigen, dass es nicht zwingend erforderlich ist die Traversierung der zu migrierenden JSON-Dokumente gezielt zu steuern, sondern das auch die erschöpfende Traversierung eingesetzt werden kann. Diese Erkenntnis lässt sich jedoch nicht verallgemeinern, da sich beispielsweise bei der Nutzung eines anderen Serialisierungsframeworks oder eines leicht angepassten Migrationsansatzes Unterschiede in den Ausführungszeiten ergeben können.

Forschungsfrage RQ3

RQ3: Führt die Minimierung der Graphen vor der Code-Generierung in realen Migrationsszenarien tatsächlich zu einer Reduktion der Komplexität der generierten Migrationsfunktionalität?

Die Migrationsklassen für die verschiedenen Versionsübergänge wurden sowohl unter Minimierung der Graphen als auch ohne Minimierung der Graphen generiert. Die Ergebnisse dazu sind in Tabelle 6.50 dargestellt. Wie in dieser Tabelle zu sehen, wurde unter Minimierung der Graphen insgesamt genau eine Traversierungsmethode und genau eine Migrationsmethode weniger generiert als ohne Minimierung der Graphen. Eine Traversierungsmethode stellt dabei eine Methode dar, über die innerhalb des JSON-Dokuments navigiert wird. Eine Migrationsmethode kapselt hingegen die Ausführung einer Migrationsoperation.

	Unter Minimierung der Graphen	Ohne Minimierung der Graphen
Anzahl Traversierungsmethoden	555	556
Anzahl Migrationsmethoden	113	114

Tabelle 6.50: Übersicht über die Anzahl der Traversierungsmethoden und Migrationsmethoden mit und ohne Minimierung der Graphen.

Dieses Ergebnis zeigt, dass die Minimierung der Graphen in dieser Fallstudie lediglich zu einer sehr marginalen Reduktion der Komplexität der generierten Migrationsfunktionalität geführt hat. Dessen ungeachtet ist der Minimierungsschritt sinnvoll, da in manchen Szenarien auf diese Weise unter Umständen zahlreiche Migrationsoperationen weniger ausgeführt werden müssen.

6.9.4 Gültigkeit der Fallstudie

Die Hauptgefahr für die Gültigkeit dieser Fallstudie ist, dass der Ansatz lediglich auf ein einzelnes Softwaresystem angewendet wurde. Die Erkenntnis, dass der Ansatz dafür geeignet ist in einem realen Softwaresystem eingesetzt zu werden, lässt sich aber trotzdem auf andere Softwaresysteme übertragen. Dies ist vor allem darin begründet, dass in dem betrachteten Bereich der realen Modellhistorie verschiedenste Arten von Modelländerungen vorgelegen haben, die wahrscheinlich auch bei anderen Softwaresystemen vorzufinden sein werden. Für diese verschiedenen Arten von Modelländerungen konnten passende Datenmigrationen generiert werden, so dass auch in anderen Softwaresystemen passende Datenmigrationen für diese Fälle generiert werden können. Zudem ist der Ansatz nicht dediziert auf nur wenige Arten von Modelländerungen ausgelegt, sondern betrachtet zahlreiche Änderungsarten bei Klassendiagrammen. Dies ist keine zu restriktive Einschränkung, da Klassendiagramme die am häufigsten eingesetzte Modellierungstechnik der UML sind [HWRK11, HWR14].

Andere Ergebnisse können sich hingegen bei der Analyse der Auswirkungen der Pfadermittlung ergeben, wenn ein anderes Framework zur Verarbeitung der JSON-Daten eingesetzt wird. Da der Traversierungsaufwand durch die Pfadermittlung aber reduziert wird, werden sich andere Frameworks sehr wahrscheinlich lediglich positiv auf diese Forschungsfrage auswirken.

Kapitel 7

Modellierung von Generatorvariabilität

Die Wiederverwendung von Software ist ein wichtiges Grundprinzip in der Softwareentwicklung [MMM95]. In modellgetriebenen Softwareentwicklungsprojekten ist nicht nur die Wiederverwendung von handgeschriebenen Artefakten, sondern auch die Wiederverwendung von Generatoren wünschenswert. In dieser Arbeit wird ein Ansatz vorgestellt, der explizit auf die Wiederverwendung von Generatoren ausgelegt ist. In diesem Ansatz kann die Variabilität von Generatoren modelliert werden, so dass auf einfache Weise neue Varianten von Generatoren (siehe Definition 5 auf Seite 7) definiert werden können. Die Grundlage für dieses Kapitel bildet die Publikation [GMR⁺16].

Den Kernbestandteil dieses Ansatzes bildet die Möglichkeit, für frei definierbare Regionen in Generatorartefakten Kompositionsoperationen (siehe Definition 13) modellieren zu können.

Definition 13 (Kompositionsoperation)

Eine Kompositionsoperation drückt aus, dass auf Basis zweier Artefakte X und Y ein neues Artefakt Z erzeugt werden soll, indem eine bestimmte Region innerhalb des Artefakts X durch Inhalt aus dem Artefakt Y ersetzt oder Inhalt aus Y zum Artefakt X hinzugefügt wird.

Kompositionsoperationen bieten somit die Möglichkeit auszudrücken, auf welche Weise für eine neue Generatorvariante neue Varianten bestehender Artefakte erzeugt werden sollen.

Ein weiterer wichtiger Aspekt des Ansatzes ist, dass die Generatorartefakte in wiederverwendbaren Schichten organisiert werden. So können für Generatorartefakte einer Schicht Kompositionsoperationen definiert werden, die ausdrücken, wie diese Generatorartefakte mit den Generatorartefakten einer anderen Schicht komponiert werden. Die Erzeugung einer konkreten Generatorvariante erfolgt schließlich über die Komposition der Generatorartefakte, die in ausgewählten Schichten enthalten sind. Dabei werden die Kompositionsoperationen angewendet, die für die betroffenen Generatorartefakte definiert sind.

Im weiteren Verlauf dieses Kapitels werden in Abschnitt 7.1 zunächst die Grundlagen des Ansatzes eingeführt und in Abschnitt 7.2 ein Beispiel eingeführt, an dem die Konzepte erläutert werden. Daraufgehend wird in Abschnitt 7.3 ein Überblick über die Methodik zur Modellierung der Generatorvariabilität gegeben, bevor die verfügbaren Variabilitätskonzepte zur Entwicklung von Generatorvarianten in Abschnitt 7.4 eingeführt werden. Anschließend beschreibt Abschnitt 7.5 die Mechanismen zur Konfiguration von Generatorvarianten, gefolgt von einer Erläuterung des Kompositionsmechanismus zur Erzeugung von Generatorvarianten in Abschnitt 7.6. In Abschnitt 7.7 wird der Ansatz daraufhin diskutiert, bevor in Abschnitt 7.8 verwandte Arbeiten vorgestellt werden. Abschließend werden in Abschnitt 7.9 die Ergebnisse einer Fallstudie dargelegt, in welcher der Ansatz zur Erzeugung von Generatorvarianten eingesetzt wurde.

7.1 Grundlagen

7.1.1 Feature-orientierte Programmierung

Die Konzepte der Organisation von Artefakten in wiederverwendbaren Schichten sowie der Auswahl und Komposition dieser Schichten stammen aus der Feature-orientierten Programmierung (FOP) [ABKS13]. Diese Technik dient der Implementierung von Softwareproduktlinien (siehe Definition 14), in der Softwaresysteme durch die Komposition von Features (siehe Definition 15) entstehen.

Definition 14 (Softwareproduktlinie)

Eine Softwareproduktlinie (engl. software product line, SPL) besteht aus einer Menge von Komponenten. Ein konkretes Produkt einer SPL wird durch die Auswahl einer Teilmenge dieser Komponenten erzeugt [RR15, GMR⁺16].

Definition 15 (Feature (Kontext Feature-orientierte Programmierung))

Ein Feature repräsentiert eine konfigurierbare Einheit eines Softwaresystems, die dazu beiträgt, dass eine Anforderung an das Softwaresystem umgesetzt wird [ALS06, ABKS13].

In FOP werden Features in Schichten angeordnet, die Artefakte enthalten [SB02, BSR03, ALS05]. Die Wiederverwendung von Artefakten kann unter anderem dadurch realisiert werden, dass ein Artefakt einer Schicht mit einem oder mehreren Artefakten aus anderen Schichten komponiert wird [ABKS13]. Überdies können Artefakte unverändert übernommen werden. Abbildung 7.1 zeigt ein Beispiel für einen FOP-Stapel bestehend aus drei Schichten.

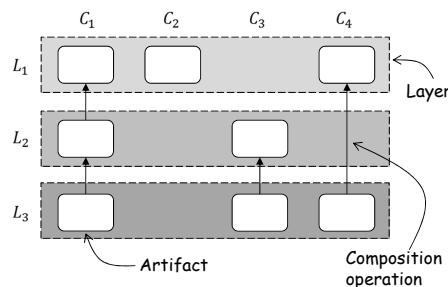


Abbildung 7.1: Beispiel für drei FOP-Schichten L_1 bis L_3 mit vier Kompositionsoptionen.

In diesem Beispiel enthält die erste Schicht die drei Artefakte C_1 , C_2 und C_4 . Die zweite Schicht enthält ein neues Artefakt C_3 sowie ein Artefakt C_1 , welches mit C_1 aus der ersten Schicht komponiert wird. Die dritte Schicht enthält nur Artefakte, die mit Artefakten anderer Schichten komponiert werden: C_1 mit C_1 aus der zweiten Schicht, C_3 mit C_3 aus der zweiten Schicht sowie C_4 mit C_4 aus der ersten Schicht.

Die wesentliche Motivation für die Verwendung von Schichten besteht darin, dass dadurch die Erzeugung von Varianten vereinfacht wird. So werden in FOP konkrete Produktvarianten durch die Komposition von Schichten gebildet. Im Zuge dessen werden die Kompositionsoptionen, die zwischen den Artefakten der betroffenen Schichten definiert sind, angewendet. Beispielsweise führt die Komposition der Schichten L_1 und L_2 in Abbildung 7.1 dazu, dass das Artefakt

C_1 aus L_1 mit C_1 aus L_2 komponiert wird. Da überdies keine weiteren Kompositionsoperationen zwischen diesen beiden Schichten definiert sind, werden die Artefakte C_2 , C_3 und C_4 unverändert in die resultierende Variante übernommen.

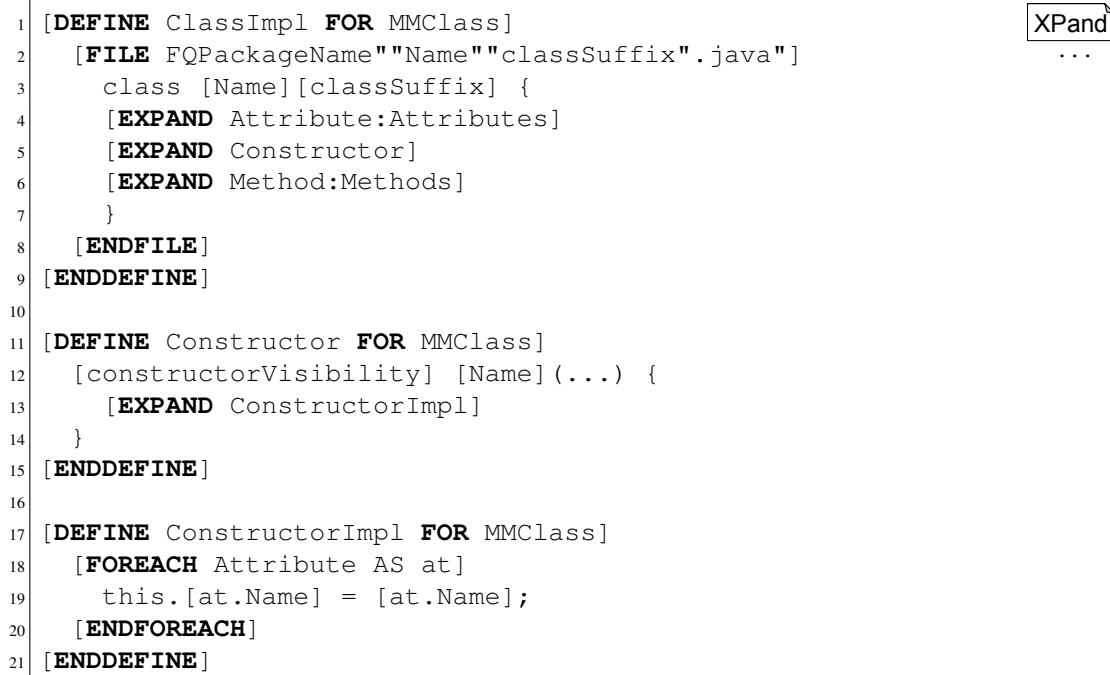
In dieser Arbeit enthalten die Schichten Generatorartefakte und werden daher fortan als Code-Generator-Schichten (engl. code generator layer, CGL) bezeichnet. Für die Realisierung von CGLs gibt es verschiedene Möglichkeiten. In dieser Arbeit wird, analog zu [ALS05], angenommen, dass die Artefakte einer CGL im Dateisystem in einem bestimmten Verzeichnis abgelegt sind. Obgleich zur Entwicklung von Generatorvarianten mehrere CGLs definiert werden können, liegt als Ergebnis nicht zwangsläufig eine reine Schichtenanordnung vor. Stattdessen können die zu komponierenden Schichten ebenso in einer Baumstruktur resultieren. Da der Ansatz auf FOP basiert und die Terminologie dort von Schichten spricht, wird in dieser Arbeit dessen ungeachtet ebenfalls der Begriff der Schicht verwendet.

7.1.2 Template-basierte Generierung

Der in dieser Arbeit entwickelte Ansatz ist auf Template-basierte Generatoren [CH03, CH06] ausgelegt. Diese bestehen insbesondere aus Templates, welche den zu generierenden Code kapseln. Zur Steuerung des zu generierenden Codes greifen solche Generatoren auf Anweisungen einer Templatesprache zurück. Ferner kann innerhalb von Templates üblicherweise auf Hilfsklassen [Fow10] zugegriffen werden, so dass Logik aus Templates in Hilfsklassen ausgelagert werden kann. In dieser Arbeit werden Generatoren eingesetzt, die auf Basis des Generatorframeworks openArchitectureWare entwickelt werden, die als Templatesprache XPand nutzen und die Hilfsklassen in Java implementieren.

In der Templatesprache XPand kann ein Template in verschiedene Blöcke unterteilt werden. Diese Blöcke beginnen mit dem Schlüsselwort `DEFINE` und werden daher fortan als `DEFINE` Blöcke bezeichnet. Jeder `DEFINE` Block besitzt einen Namen, kann optional eine Liste von Parametern besitzen und ist für einen spezifischen Typ von Eingabemodellelement, im weiteren Verlauf des Kapitels als Metamodellklasse bezeichnet, definiert. Die Bedeutung dieser Metamodellklassen wird im Folgenden erläutert. Zu Beginn des Generierungsprozesses werden die Eingabemodelle des Generators eingelesen und in einen AST überführt. Ein dediziertes Template, das Haupttemplate, dient als Startpunkt der Generierung [Sch12]. Ausgehend von einem spezifischen `DEFINE` Block in diesem Haupttemplate werden die weiteren Templates aufgerufen und so der Code generiert. Dieser `DEFINE` Block im Haupttemplate wird für den Wurzelknoten des ASTs ausgeführt. Dabei navigiert das Haupttemplate durch den AST oder ruft andere Templates auf, die durch den AST navigieren. Der Typ der Metamodellklasse eines `DEFINE` Blocks gibt an, welchen AST-Typ der `DEFINE` Block verarbeiten kann. Beispielsweise kann ein `DEFINE` Block, welcher für den Typ `MMClass` definiert ist, konkrete Klassen eines Klassendiagramms verarbeiten und dafür Code generieren.

Listing 7.1 zeigt eine Realisierung eines Templates, welches auf die Generierung von Java-Code aus einem Klassendiagramm abzielt. In diesem Template sind drei `DEFINE` Blöcke definiert: `ClassImpl` (Zeile 1 – 9), `Constructor` (Zeile 11 – 15) und `ConstructorImpl` (Zeile 17 – 21). Die Definition von `DEFINE` Blöcken in XPand ist vergleichbar zu der Deklaration von Methoden in objektorientierten Programmiersprachen, wobei das Template dann zu der Deklaration der Klasse korrespondiert.



```

1 [DEFINE ClassImpl FOR MMClass]
2   [FILE FQPackageName "Name" "classSuffix".java"]
3     class [Name][classSuffix] {
4       [EXPAND Attribute:Attributes]
5       [EXPAND Constructor]
6       [EXPAND Method:Methods]
7     }
8   [ENDFILE]
9 [ENDDFINE]
10
11 [DEFINE Constructor FOR MMClass]
12   [constructorVisibility] [Name](...) {
13     [EXPAND ConstructorImpl]
14   }
15 [ENDDFINE]
16
17 [DEFINE ConstructorImpl FOR MMClass]
18   [FOREACH Attribute AS at]
19     this.[at.Name] = [at.Name];
20   [ENDFOREACH]
21 [ENDDFINE]

```

Listing 7.1: Template Class.tpl zur Generierung des Grundgerüsts einer Java-Klasse.

Im ersten **DEFINE** Block wird zu Beginn über das **FILE** Kommando veranlasst, dass Code in eine spezifische Datei generiert wird. In welche Datei der Code generiert wird, ergibt sich aus dem Aufruf der drei Hilfsmethoden `FQPackageName()`, `Name()` und `classSuffix()`, die allesamt in der Metamodellklasse `MMClass` oder einer Oberklasse davon implementiert sein müssen. Diese drei Methoden liefern jeweils eine Zeichenkette zurück. Dadurch kann für das **FILE** Kommando die Zeichenkette konstruiert werden, die den Pfad zu der Datei angibt, in die generiert wird. Der Code, der in diese Datei generiert wird, ergibt sich aus dem im **FILE** Block statisch angegebenen Code und der Auswertung der enthaltenen **XPand**-Kommandos.

Außerhalb des **FILE** Kommandos wird der Aufruf einer Hilfsmethode über eckige Klammern ausgedrückt - ohne zusätzliche **XPand**-Kommandos. Beispielsweise ergibt sich der Name der zu generierenden Klasse aus dem Rückgabewert der Methode `Name()` konkateniert mit dem Rückgabewert der Methode `classSuffix()` (Zeile 3).

Ein weiteres wichtiges Konstrukt in **XPand** ist das **EXPAND** Kommando, worüber ein **DEFINE** Block aufgerufen wird. Im Rahmen der Ausführung dieses **DEFINE** Blocks wird eine Zeichenkette konstruiert, die an die Stelle der **EXPAND** Anweisung in die Ausgabe eingefügt wird. So wird in Zeile 4 der **DEFINE** Block `Attributes` des Templates `Attribute` aufgerufen. Die dort konstruierte Zeichenkette wird an den Anfang der Klassendeklaration eingefügt, so dass in diesem Fall die Attributdeklarationen einer Klasse zu Beginn erscheinen. Danach wird der **DEFINE** Block `Constructor` aufgerufen (Zeile 5), um den Konstruktor zu generieren. Abschließend erfolgt der Aufruf des **DEFINE** Blocks `Methods` des Templates `Method` (Zeile 6), um die Methodenimplementierungen zu generieren.

XPand bietet darüber hinaus die Möglichkeit, Bedingungen zu überprüfen und Schleifenkon-

strukture auszudrücken. So wird durch die Schleife in Zeile 18–20 über alle Attribute einer Klasse iteriert, jedes Attribut zwischengespeichert in der Variable `at` und danach wird im Konstruktor jedes Attribut mit dem Wert des übergebenen Parameters initialisiert.

Schließlich können in XPand über `REM` und `ENDREM` Kommentare eingeführt werden. Dieses Konstrukt wird später eingesetzt, um zu zeigen, wie Regionen in Generatorartefakten definiert werden können, deren Inhalt verändert werden soll.

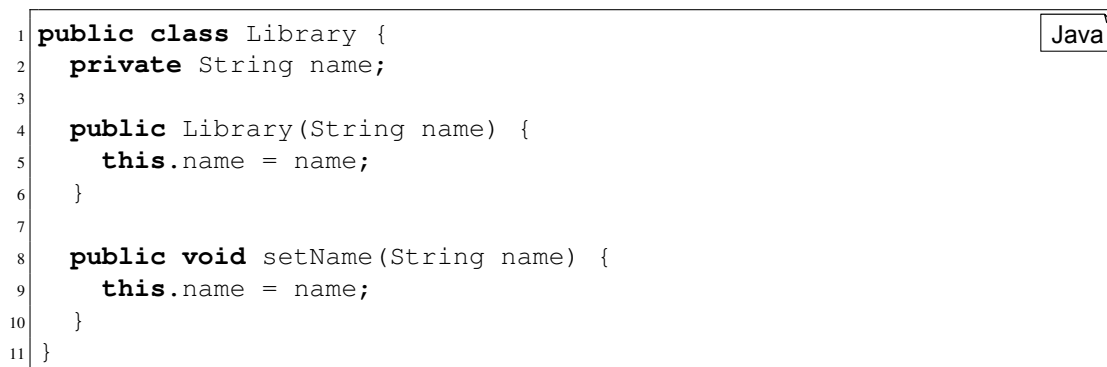
7.2 Beispiel

Nachfolgend wird ein einfaches Beispiel eingeführt, welches innerhalb des Kapitels zur Verdeutlichung verschiedener Konzepte eingesetzt wird. In diesem Beispiel wird angenommen, dass ein Generator ein Klassendiagramm als Eingabemodell erwartet und aus diesem Java-Code wie folgt generiert:

- Aus der UML-Klasse `X` wird eine gleichnamige Java-Klasse `X` generiert.
- Für jedes Attribut `y` einer UML-Klasse `X` wird in die Java-Klasse `X` ein gleichnamiges Java-Attribut `y` generiert. Die Sichtbarkeit dieses Attributs ergibt sich aus dem Klassendiagramm. Analog wird für Assoziationen verfahren. Zusätzlich wird zu jedem Attribut und zu jeder Assoziation eine `set`-Zugriffsmethode erzeugt. Von der Generierung einer `get`-Zugriffsmethode wurde zur Verkürzung des Beispiels abgesehen.
- Die Klasse `X` erhält einen parametrisierten öffentlichen Konstruktor, der als Argumente alle Attribute der UML-Klasse erwartet.

Das Generat für ein Klassendiagramm, welches lediglich die Klasse `Library` mit dem Attribut `name` vom Typ `String` umfasst, ist in Listing 7.2 dargestellt.

```
1 public class Library {
2     private String name;
3
4     public Library(String name) {
5         this.name = name;
6     }
7
8     public void setName(String name) {
9         this.name = name;
10    }
11 }
```



Listing 7.2: Basisvariante der Java-Klasse `Library` mit einem Attribut, einem öffentlichen Konstruktor sowie einer `set`-Zugriffsmethode.

Die Umsetzung dieses Generators unter Nutzung der Templatesprache XPAND wird im Folgenden vorgestellt. In Listing 7.1 auf Seite 178 wurde vorab bereits das Haupttemplate gezeigt, welches für die Generierung einer Java-Klasse zuständig ist. Dieses Haupttemplate bedient sich unter anderem der Hilfsmethoden `classSuffix()` und `constructorVisibility()`,

deren Implementierung in Listing 7.3 gezeigt ist. Wie dort zu sehen, liefert die Hilfsmethode `classSuffix()` standardmäßig die leere Zeichenkette zurück (Zeile 2 – 4), so dass für eine Klasse standardmäßig kein Suffix generiert wird. Weiterhin wird als Sichtbarkeit für Konstruktoren `public` angenommen (Zeile 5 – 7).

```

1 public class MMClass extends Class {
2     public String classSuffix() {
3         return "";
4     }
5     public String constructorVisibility() {
6         return "public";
7     }
8 }

```

Java
...

Listing 7.3: Die Metamodellklasse `MMClass.java` kapselt Hilfsfunktionalität, die innerhalb von Templates aufgerufen werden kann.

Das Template `Attribute` generiert die Attributdeklarationen für eine Klasse, wie in Listing 7.4 dargestellt. Für jedes Attribut der UML-Klasse wird ein Attribut generiert, dessen Sichtbarkeit, Typ und Name sich nach dem Attribut in der UML-Klasse richtet (Zeile 2 – 4).

```

1 [DEFINE Attributes FOR MMClass]
2     [FOREACH Attribute AS at]
3         [at.Visibility] [at.Type] [at.Name];
4     [ENDFOREACH]
5 [ENDDEFINE]

```

XPand

Listing 7.4: Template `Attribute.tpl` zur Generierung der Attribute für eine Klasse.

Abschließend zeigt Listing 7.5 das Template `Method`, welches für die Generierung der Methodenimplementierungen zuständig ist. Für jedes Attribut wird eine `set`-Methode generiert (Zeile 2 – 4). Die Generierung der `set`-Methode ist aufgeteilt in den Methodenkopf (Zeile 8 – 12) und die Methodenimplementierung (Zeile 14 – 16). Überdies enthält das Template einen leeren `DEFINE` Block `FurtherMethods` (Zeile 19 – 20), der als Erweiterungspunkt für die Generierung weiterer Methoden verwendet werden kann.

Für einen anderen Anwendungsfall wird angenommen, dass folgender Code generiert werden muss:

- an den Namen der Klasse soll das Suffix `Impl` angehängen werden
- die Sichtbarkeit des Konstruktors soll von `public` auf `protected` geändert werden
- die Argumente der `set`-Zugriffsmethoden sollen geprüft werden
- es soll eine statische Methode generiert werden, welche zur Instanziierung von Objekten der Klasse aufgerufen werden kann

Anstatt den bestehenden Generator zu kopieren und die erforderlichen Änderungen in die Kopie einzuarbeiten, soll der bestehende Generator wiederverwendet werden.

```

1 [DEFINE Methods FOR MMClass]
2   [FOREACH Attribute AS at]
3     [EXPAND SetterMethod FOR at]
4   [ENDFOREACH]
5   [EXPAND FurtherMethods]
6 [ENDDFINE]
7
8 [DEFINE SetterMethod FOR MMAttribute]
9   public void set[UpperCaseName]([Type] [Name]) {
10     [EXPAND SetterMethodBody]
11   }
12 [ENDDFINE]
13
14 [DEFINE SetterMethodBody FOR MMAttribute]
15   this.[Name] = [Name];
16 [ENDDFINE]
17
18 [REM]Hookpoint for future extensions[ENDREM]
19 [DEFINE FurtherMethods FOR MMClass]
20 [ENDDFINE]

```

Listing 7.5: Template Method.tpl zur Generierung der Methoden für eine Klasse.

Die angepasste Variante des Codes der zur beschriebenen UML-Klasse `Library` generiert wird, ist in Listing 7.6 dargelegt. Wie dort zu sehen ist, entspricht der Name der Java-Klasse dem Namen der UML-Klasse mit angehängtem Suffix `Impl` (Zeile 1, 4, 13, 15), die Konstruktorsichtbarkeit wurde zu `protected` geändert (Zeile 4), zu Beginn der Methode `setName()` findet eine Prüfung des übergebenen Werts statt (Zeile 9) und es gibt eine statische Methode `create()` zur Erzeugung von `LibraryImpl` Instanzen (Zeile 13 – 16).

7.3 Methodik zur Modellierung von Generatorvariabilität

Der Ansatz zur Modellierung von Generatorvariabilität gliedert sich in fünf Schritte. Diese werden im Folgenden kurz beschrieben. In den weiteren Abschnitten dieses Kapitels werden die letzten vier Schritte zudem ausführlich erläutert.

1. Die Entwickler analysieren die bestehenden Generatorvarianten. Das Hauptziel dieser Analyse ist es, die Teile der Generatorartefakte zu identifizieren, die für die neue Generatorvariante angepasst werden müssen. Daraus folgt zugleich, welche neuen Generatorartefakte für die neue Generatorvariante bereitgestellt werden müssen.
2. Anschließend implementieren die Entwickler die neu benötigten Generatorartefakte sowie die Generatorartefakte, die dazu genutzt werden, den Inhalt der bestehenden Generatorartefakte anzupassen. Diese werden von den Entwicklern in CGLs organisiert. Aufbauend auf diesen Artefakten und den CGLs müssen die Entwickler die erforderlichen Kompositionsoperationen modellieren. In Abschnitt 7.4 wird ein Überblick über die verfügbaren

```
1 public class LibraryImpl {
2     private String name;
3
4     protected LibraryImpl(String name) {
5         this.name = name;
6     }
7
8     public void setName(String name) {
9         assert(name != null);
10        this.name = name;
11    }
12
13    public static LibraryImpl create(String name) {
14        assert(name != null);
15        return new LibraryImpl(name);
16    }
17 }
```

Listing 7.6: Erweiterte Variante der Java-Klasse mit einem Attribut, einer set-Zugriffsmethode sowie einer statischen Instanziierungsmethode.

Kompositionsoptionen gegeben. Die Modellierung dieser Kompositionsoptionen ist eine grundlegende Voraussetzung zur Definition einer Generatorvariante und wird im Detail in Abschnitt 7.5.1 erläutert.

3. Als Nächstes definieren die Entwickler, aus welchen Teilen der bestehenden Generatorvarianten und aus welchen Erweiterungen bzw. angepassten Teilen der Generatorvarianten die neue Generatorvariante bestehen soll. Konkret definieren die Entwickler hierzu die CGLs, aus denen die Generatorvariante zusammengesetzt werden soll. Dieser Teil der Konfiguration einer Generatorvariante wird in Abschnitt 7.5.2 beschrieben.
4. Daraufgehend wird automatisch geprüft, ob zusätzlich zu den im vorherigen Schritt manuell festgelegten CGLs weitere CGLs zur Erzeugung der Generatorvariante benötigt werden. Die daraus resultierende CGL-Auswahl wird automatisch validiert mit Hinblick darauf, ob es zulässig ist, eine Generatorvariante auf Basis der entsprechenden CGLs zu erzeugen. Falls die Konfiguration der Generatorvariante nicht zulässig ist, müssen die Entwickler entweder die modellierten Kompositionsoptionen oder die Generatorartefakte, die für die Komposition verwendet werden, ändern. Dazu müssen sie in dieser Methodik in den zweiten oder dritten Schritt zurückkehren. Die Details zu dieser automatischen Validierung werden in Abschnitt 7.5.3 erklärt.
5. Abschließend werden für die zuvor konfigurierte Generatorvariante automatisch die passenden Generatorartefakte erzeugt. Die Entwickler können die resultierende Generatorvariante daraufhin unmittelbar einsetzen. Die Erzeugung der Generatorartefakte wird im Detail in Abschnitt 7.6 beschrieben.

7.4 Variabilitätskonzepte

7.4.1 Variabilitätsregionen

Die Voraussetzung für die Definition von Kompositionsoperationen zwischen Artefakten ist, dass festgelegt werden kann, welche Teile eines Artefakts auf welche Weise miteinander komponiert werden sollen. Eine Herausforderung bei der Anwendung von FOP auf Generatoren besteht darin, dass in der Generatorentwicklung verschiedene Arten von Sprachen zum Einsatz kommen. So wird neben einer Templatesprache oftmals eine weitere Sprache zur Implementierung von Hilfsfunktionalität eingesetzt. Dadurch müssen Kompositionsoperationen für verschiedenartige Artefakte definiert werden können. Im Folgenden liegt der Fokus auf der Komposition von Artefakten der beiden genannten Sprachtypen.

Obgleich es Generator-Frameworks für Template-basierte Generatoren gibt, die zu gewissen Teilen Adaptionen von Generatoren unterstützen [Acc, Xtea], gibt es bisher kein allgemeingültig anwendbares Konzept, welches auf verschiedene Generator-Frameworks angewendet werden kann. Der in dieser Arbeit vorgestellte Ansatz überwindet diese Einschränkung durch die Einführung von Generatorregionen (siehe Definition 16).

Definition 16 (Generatorregion)

Eine Generatorregion (engl. generator region, GR) ist eine dedizierte Region innerhalb eines Generatorartefakts, die eindeutig referenziert werden kann. Eine GR kann entweder einen Bereich innerhalb eines Generatorartefakts markieren, der Text enthält oder eine leere Region kennzeichnen, die keinen Text enthält.

Die gezeigte Definition deckt auch den Fall ab, dass der komplette Inhalt eines Artefakts als GR betrachtet werden kann.

Im Falle eines Templates kann eine GR Templateanweisungen oder statischen Code, der im Template enthalten ist, umfassen. Im Falle einer Hilfsklasse kann eine GR hingegen auf Code der Hilfsklasse verweisen.

Der Begriff der GR verweist ganz allgemein betrachtet auf eine klar definierte Region innerhalb eines Generatorartefakts. In dieser Arbeit werden zwei spezielle Ausprägungen von GRs unterschieden: Variabilitätsregionen (siehe Definition 17) und Varianten von GRs (siehe Definition 18).

Definition 17 (Variabilitätsregion)

Eine Variabilitätsregion (engl. variability region, VR) ist eine GR, für die über mindestens eine Kompositionsoperation definiert ist, dass ihr Inhalt verändert werden soll.

Definition 18 (Variante einer Generatorregion)

Eine Variante einer Generatorregion (engl. generator region variant, GRV) ist eine GR, die eine VR ersetzt.

Wird im weiteren Verlauf des Kapitels der Begriff der GR genutzt, umfasst dies sowohl den Fall, dass eine VR als auch den Fall dass eine GRV vorliegt. In der Erläuterung der konkreten Beispiele wird hingegen zwischen VRs und GRVs unterschieden. Eine Besonderheit liegt vor, wenn eine GRV durch eine andere GRV ersetzt wird. In diesem Fall stellt die ersetzte GRV für

die ersetzende GRV eine VR dar. Zudem wird in den Beispielen der Begriff der GR genutzt, wenn eine Region vor oder hinter eine VR eingefügt wird, wie im nächsten Abschnitt noch erläutert wird.

Zur Referenzierung einer GR wird in dieser Arbeit vorausgesetzt, dass jede GR einen Namen besitzt, auf den verwiesen werden kann. Um eine Region innerhalb eines Artefakts als GR zu kennzeichnen, kommen verschiedene Herangehensweisen in Frage. Nachstehend werden zwei mögliche Ansätze vorgestellt, die auch kombiniert werden können:

- Es werden spezifische Sprachkonstrukte genutzt, um GRs abzuleiten bzw. zu markieren.
- Es werden sprachunabhängige Anweisungen eingeführt, z.B. Kommentare, welche die Semantik eines Artefakts nicht beeinflussen, über die eine GR aber explizit gekennzeichnet werden kann.

Das erstgenannte Konzept lässt sich auf jede Sprache anwenden, die Sprachkonstrukte bietet, die zur Kennzeichnung von GRs geeignet sind. Dies sind vor allem Sprachkonstrukte, die auf einen Namen verweisen, der zur Referenzierung einer GR verwendet werden kann. Überdies sind solche Sprachkonstrukte gut geeignet, die mehrere zusammengehörige Anweisungen einer Sprache kapseln können. Das zweitgenannte Konzept lässt sich hingegen auf jede Sprache anwenden, welche die Möglichkeit zur Formulierung von Kommentaren bietet. Beide Ansätze werden im Folgenden durch die Anwendung auf XPand und Java verdeutlicht.

Eine Variante des zweiten Ansatzes ist es, Anweisungen in die Generatorartefakte einzuführen, die bei der Erzeugung der Generatorartefakte für die neue Generatorvariante wieder entfernt werden. Auf diese Weise müssen die eingeführten Anweisungen zur Markierung von GRs nicht bei der Ausführung der resultierenden Generatorartefakte verarbeitet werden. Diese Variante ist vergleichbar zu der Nutzung von Präprozessoranweisungen, wird im Folgenden allerdings nicht näher betrachtet.

Generatorregionen in XPand und Java

Übertragen auf die Templatesprache XPand eignen sich insbesondere `DEFINE` Blöcke zur Markierung von GRs. Der Grund dafür ist, dass `DEFINE` Blöcke die Gruppierung eines Templates in verschiedene Bereiche ermöglichen. Ferner besitzt jeder `DEFINE` Block einen Namen und kann im Template über diesen Namen referenziert werden. Eine nicht-leere Region in einem Template kann daher durch einen nicht-leeren `DEFINE` Block gekennzeichnet werden. Das Template in Listing 7.5 auf Seite 181 besitzt somit die nicht-leeren GRs mit den Namen `Methods`, `SetterMethod` sowie `SetterMethodBody`. Analog kann eine leere Region in einem Template durch einen leeren `DEFINE` Block umgesetzt werden. In Listing 7.5 auf Seite 181 ist mit `FurtherMethods` eine solche leere GR angegeben.

Übertragen auf die Programmiersprache Java kann auf ähnliche Weise abgeleitet werden, dass jede Methode eine GR markiert. Eine nicht-leere Region in einer Java-Klasse korrespondiert folglich zu einer nicht-leeren Methodenimplementierung. Analog dazu kann eine leere Region durch eine Methode ohne implementierten Methodenkörper modelliert werden. Die in Listing 7.3 auf Seite 180 gezeigte Hilfsklasse beinhaltet demnach die beiden nicht-leeren GRs `classSuffix` und `constructorVisibility`.

Generatorregionen über Kommentare

Nicht jede Sprache bietet Sprachkonstrukte wie Methoden oder `DEFINE` Blöcke zur Strukturierung von Artefakten an. In diesen Fällen können GRs sprachunabhängig, beispielsweise über Kommentarblöcke, definiert werden. Listing 7.7 verdeutlicht die Idee für die Generierung von `set`-Methoden für das Beispiel aus Abschnitt 7.2. Diese Implementierung kapselt die Generierung einer `set`-Methode über zwei Kommentarblöcke. Diese Kommentarblöcke können durch die Einführung eines Namens in den Kommentaren (`SetterMethod` sowie `SetterMethodBody`) adressiert werden, so dass die Kommentarblöcke als GR verwendet werden können.

```

1 [REM]BEGIN VR:SetterMethod[ENDREM]
2   public void set[UpperCaseName]([Type] [Name]) {
3       [REM]BEGIN VR:SetterMethodBody[ENDREM]
4           this.[Name] = [Name];
5       [REM]END VR:SetterMethodBody[ENDREM]
6   }
7 [REM]END VR:SetterMethod[ENDREM]
```



Listing 7.7: Alternative Template-Implementierung für die Generierung einer Methode mit Methodenkörper unter Nutzung von Kommentaren zur Kennzeichnung von GRs.

Der Kommentar-basierte Ansatz kann auf jede Sprache übertragen werden, die das Konzept der Kommentare bereitstellt. Dies ist in den meisten Sprachen der Fall. Der Hauptnachteil dieses Ansatzes ist jedoch, dass auf diese Weise induzierte Kommentare fragil sind, da leicht ein GR-Kommentar unabsichtlich verändert werden kann, ohne dass dies Auswirkungen auf die Ausführbarkeit des Templates hat. Ferner können zusätzlich zu den GR-Kommentaren auch normale Kommentare existieren. Dadurch kann es Entwicklern schwerer fallen, die Kommentare zu unterscheiden. Dessen ungeachtet können auf diese Weise GRs in jeglichen Artefakten gekennzeichnet werden.

7.4.2 Definition von Kompositionsoperationen

Wie einleitend bereits in Definition 13 auf Seite 175 dargelegt, drückt eine Kompositionsoperation aus, dass auf Basis zweier Artefakte *X* und *Y* ein neues Artefakt *Z* erzeugt werden soll. Durch die Definition passender Kompositionsoperationen kann dabei der Inhalt von *X* gezielt durch den Inhalt von *Y* verändert werden, so dass als Ergebnis eine neue Variante des Artefakts *X* vorliegt. Im weiteren Verlauf des Kapitels wird daher auf das Artefakt *X* als das adaptierte Artefakt verwiesen, da dessen Inhalt verändert wird. Analog dazu wird auf das Artefakt *Y* als das adaptierende Artefakt verwiesen, da *X* auf Basis von *Y* verändert wird.

In dieser Arbeit werden drei Arten von Kompositionsoperationen unterschieden. Zur Verdeutlichung der Auswirkungen der Kompositionsoperationen sind in der nachfolgenden Aufzählung die Namen der Artefakte, für welche die Kompositionsoperationen definiert sind, als Indizes angegeben.

1. `replaceY,X`. Drückt aus, dass eine GRV aus Artefakt *Y* eine VR aus Artefakt *X* komplett ersetzt.

2. $\text{before}_{Y,X}$. Definiert, dass eine GR aus Artefakt Y vor eine VR aus Artefakt X eingefügt wird.
3. $\text{after}_{Y,X}$. Drückt aus, dass eine GR aus Artefakt Y hinter eine VR aus Artefakt X eingefügt wird.

Zur Vereinfachung der Schreibweise werden im Folgenden die Indizes der Kompositionsoperationen nicht mehr explizit angegeben.

In Abbildung 7.9 ist ein Beispiel für die Kompositionsoperationen gezeigt. In diesem Beispiel besteht ein Artefakt K aus einer GR mit dem Namen B, einer GRV mit dem Namen R und einer GR mit dem Namen A. Ferner enthält ein Artefakt L eine VR mit dem Namen C und eine GR mit dem Namen D. Die GR B wird in diesem Beispiel vor und die GR A wird hinter die VR C eingefügt. Ferner ersetzt die GRV R die VR C. Demgegenüber ist keine Kompositionsoperation für die GR D definiert. Die Auswirkungen dieser Kompositionsoperationen sind auf der rechten Seite in Abbildung 7.9 zu sehen.

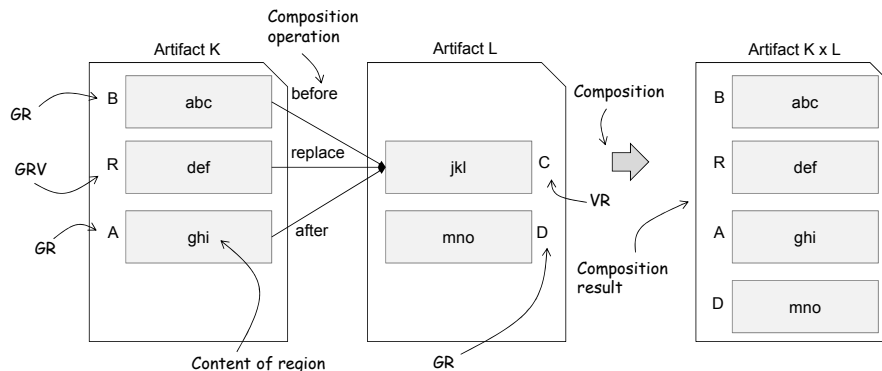


Abbildung 7.9: Auswirkungen der Kompositionsoperationen *before*, *replace* und *after*.

Wie in Abbildung 7.9 dargestellt, wirkt sich die *replace* Operation in der Art aus, dass das resultierende Artefakt die GRV R anstelle der VR C enthält. Die Semantik der Kompositionsoperationen *before* und *after* hängt von der Sprache ab, in der die betroffenen VRs implementiert sind. Repräsentiert eine VR beispielsweise eine Methode in einer Programmiersprache wie Java, ist die Bedeutung von *before* und *after*, dass vor oder nach der Ausführung dieser Methode eine andere Methode ausgeführt werden soll. In diesem Fall stellen die beiden Kompositionsoperationen Konzepte der Aspektorientierten Programmierung (engl. aspect oriented programming, AOP) für Java nach [Lad09]. Demgegenüber ist die Bedeutung bei der Anwendung auf Templates, dass vor oder hinter die VR auf Template-Ebene eine andere GR eingefügt werden soll, wie in Abbildung 7.9 gezeigt. In diesem Beispiel wird die GR B vor die VR C und die GR A hinter C eingefügt. Analog zum vorherigen Beispiel repräsentieren die beiden Operationen wiederum AOP Konzepte, in diesem Beispiel jedoch angewendet auf Templates.

Ein Sachverhalt, der im Kontext der Kompositionsoperation *replace* beachtet werden muss, betrifft das Verhalten der *replace* Operation, wenn vor oder hinter die ersetzende GRV Inhalt eingefügt wurde. In diesem Fall wird die betroffene VR durch die gesamte GRV ersetzt, einschließlich der GRs, die davor oder dahinter eingefügt wurden. In Abbildung 7.10 ist ein

Beispiel hierfür angegeben. In diesem Beispiel wird die VR D durch die GRV C ersetzt. Dabei werden die vor (B) und hinter (A) die GRV C eingefügten GRs einbezogen.

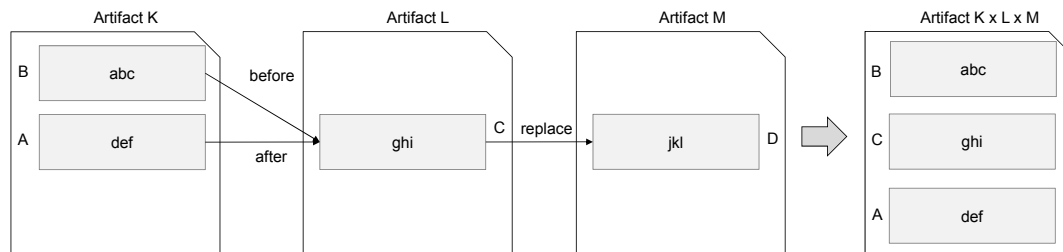


Abbildung 7.10: GRV C wird samt der GRs die vor und hinter die GRV eingefügt wurden (B und A) zu neuem Artefakt hinzugefügt.

7.5 Konfiguration von Generatorvarianten

7.5.1 Konfiguration von Generatorschichten

Die Grundvoraussetzung für die Konfiguration von Generatorvarianten besteht in der Definition der Kompositionsoperationen, die für die Artefakte der einzelnen CGLs vorgenommen werden sollen. Diese Kompositionsoperationen werden gebündelt in einem Schichtendefinitionsmodell (siehe Definition 19) festgelegt.

Definition 19 (Schichtendefinitionsmodell)

Ein Schichtendefinitionsmodell kapselt für sämtliche CGLs, welche GRVs welche VRs ersetzen und welche GRs vor und hinter welche VRs eingefügt werden.

Zur Modellierung eines Schichtendefinitionsmodells wird in dieser Arbeit die DSL Layer Definition Language (LDL) bereitgestellt. Mittels LDL kann definiert werden, wie die Artefakte eines CGLs mit den Artefakten eines anderen CGLs komponiert werden sollen. Die Grammatik zu LDL ist in Listing 7.8 dargestellt.

Jedes Schichtendefinitionsmodell besteht aus mindestens einer Schichtenspezifikation (Zeile 5). Jede Schichtenspezifikation gibt Aufschluss darüber, welche Schicht mit welcher anderen Schicht komponiert wird (Zeile 9 – 10). Innerhalb einer Schichtenspezifikation muss mindestens eine Kompositionsoperation modelliert werden (Zeile 11). Über diese Kompositionsoperationen kann definiert werden, auf welche Weise der Inhalt einer VR im Zuge der Komposition verändert werden soll. Für jede Kompositionsoperation müssen dazu die Signaturen (siehe Definition 20) der adaptierenden GR bzw. GRV (*adaptingGR*) sowie der adaptierten VR (*adaptedVR*) angegeben werden (Zeile 16 – 17, 27 – 29).

Definition 20 (Signatur einer Generatorregion)

Eine Signatur ist eine Zeichenkette, über die eine GR innerhalb eines Artefakts eindeutig identifiziert werden kann.

```

1 grammar rwth.LDL with org.eclipse.xtext.common.Terminals
2 generate LDL "http://www.se.de/rwth/LDL"
3
4 LayerDefinitionModel:
5     cgls = CGLSpecification+
6 ;
7
8 CGLSpecification:
9     'compose layer ' adaptingLayerName = ID ' with layer '
10         adaptedLayerName = [ID | CGLSpecification] '{'
11         vrCompositions = CompositionOperation+
12     '}'
13 ;
14
15 CompositionOperation:
16     adaptingGR = GRSignature compositionType = CompositionType
17     adaptedVR = GRSignature ';'
18 ;
19
20 CompositionType:
21     replace = 'replace'
22     | extends = 'extends'
23     | before = 'before'
24     | after = 'after'
25 ;
26
27 GRSignature:
28     signature = Name
29 ;
30
31 Name :
32     ID* (('.' | '-' | ':' ) ID)*
33 ;

```

Listing 7.8: Grammatik zur DSL LDL.

Zur Definition von Signaturen kann ausgenutzt werden, dass jede GR über einen Namen adressiert werden kann. In dieser Arbeit sind Signaturen in zwei Bereiche aufgeteilt. Der erste Teil kennzeichnet das Artefakt, in dem die GR enthalten ist. Dazu wird der Pfad zu dem Artefakt relativ zu dem CGL-Verzeichnis angegeben. Der zweite Teil kennzeichnet den Pfad zu der GR innerhalb dieses Artefakts. Zu diesem Zweck wird der Name der GR verwendet. Zur Unterscheidung der beiden Teile werden die beiden Teile durch einen Doppelpunkt voneinander getrennt. Dies stellt jedoch lediglich eine mögliche Vorgehensweise zur Konstruktion einer Signatur dar. Konzeptuell ist die einzige Bedingung, dass eine Signatur eine GR eindeutig identifiziert.

Als Kompositionsoperationen können unter anderem die drei in Abschnitt 7.4.2 eingeführten Kompositionsoperationen definiert werden: `replace` (Zeile 21), `before` (Zeile 23) und `after` (Zeile 24). So können die Kompositionsoperationen aus Abbildung 7.9 auf Seite 186 wie folgt in LDL definiert werden:

- K:B before L:C
- K:R replace L:C
- K:A after L:C

Zusätzlich zu den drei genannten Kompositionsoperationen bietet LDL als Komfortoperation die Kompositionsoperation `extends` (Zeile 22) an. Die Semantik dieser Kompositionsoperation ist, dass sämtliche GRVs in dem adaptierenden Artefakt sämtliche VRs in dem adaptierten Artefakt ersetzen, deren Signatur innerhalb des Artefakts identisch ist. Die Terminologie ist angelehnt an die objektorientierte Programmierung, da der beschriebene Fall dazu korrespondiert, dass eine Methode einer Unterklasse eine Methode einer Oberklasse überschreibt. Einen ähnlichen Effekt hat die `extends` Operation bei Artefakten, da die GRVs aus einem Artefakt die VRs eines anderen Artefakts überschreiben, falls diese die gleiche Signatur besitzen. Auf diese Weise kann die mehrfache Spezifikation von `replace` Operationen durch eine einzelne `extends` Spezifikation ersetzt werden. Abbildung 7.12 zeigt hierfür ein Beispiel, in dem drei `replace` Operationen angegeben sind:

- K:R1 replace L:R1
- K:R2 replace L:R2
- K:R3 replace L:R3

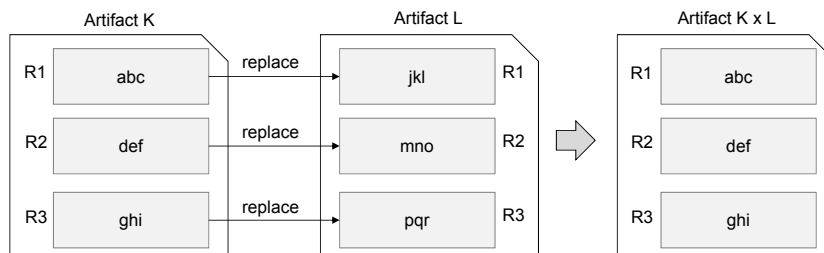


Abbildung 7.12: Mehrfache Anwendung der Kompositionsoperation `replace` für mehrere GRVs mit der gleichen Signatur wie die ersetzte VR.

Anstelle dieser drei Kompositionsoperationen kann der gleiche Sachverhalt mit Hilfe der `extends` Operation wie folgt ausgedrückt werden: `K extends L`. Der Grund dafür ist, dass durch die einzelnen `replace` Operationen jeweils eine VR im Artefakt `L` durch die GRV aus dem Artefakt `K` ersetzt wird, welche die gleiche Signatur hat.

Enthält das adaptierende Artefakt noch weitere GRs mit Namen, die im adaptierten Artefakt nicht vorkommen, so werden diese GRs im Zuge der `extends` Operation ebenfalls beachtet. In den Anwendungsbeispielen dieser Arbeit kann ein Template beispielsweise noch weitere `DEFINE` Blöcke enthalten. Im Zuge der Komposition werden diese zum adaptierten Artefakt hinzugefügt. Auch dieses Verhalten ist wiederum angelehnt an den Vererbungsmechanismus bei Klassen, da dort eine Unterklasse nicht nur Methoden überschreiben, sondern auch neue Methoden hinzufügen kann.

Anwendung auf das Beispiel

Nachfolgend wird das Konzept des Schichtdefinitionsmodells auf das in Abschnitt 7.2 eingeführte Beispiel angewendet. Da ein Schichtdefinitionsmodell eng mit den adaptierenden Templates und Hilfsklassen in Beziehung steht, werden diese nachfolgend als Erstes vorgestellt.

Das Template, über das die neue Variante der Methodengenerierung umgesetzt wird, ist in Listing 7.9 dargestellt. Der **DEFINE** Block **SetterMethodBody** (Zeile 1 – 3) kapselt die Generierung einer **assert**-Anweisung zu Beginn einer **set**-Methode. Der **DEFINE** Block **FurtherMethods** (Zeile 5 – 12) führt zudem eine statische Methode zur Erzeugung einer Instanz der entsprechenden Klasse ein.

```

1 [DEFINE SetterMethodBody FOR MMAttribute]
2   assert([Name] != null);
3 [ENDDEFINE]
4
5 [DEFINE FurtherMethods FOR MMClass]
6   public static [Name][classSuffix] create(...) {
7     [FOREACH Attribute AS at]
8       assert([at.Name] != null);
9     [ENDFOREACH]
10    return new [Name][classSuffix] (...);
11  }
12 [ENDDEFINE]
```

Listing 7.9: Das Template Methods für die angepasste Generierung von **set**-Methoden und die Generierung einer statischen Instanziierungsmethode.

Überdies zeigt Listing 7.10 die adaptierende Hilfsklasse **MMClassFactoryVar**, welche als **Suffix** für Klassen **Impl** festlegt (Zeile 2 – 4) und als Sichtbarkeit für Konstruktoren **protected** zurückliefert (Zeile 6 – 8).

```

1 public class MMClassFactoryVar extends MMClass {
2   public String classSuffix() {
3     return "Impl";
4   }
5
6   public String constructorVisibility() {
7     return "protected";
8   }
9 }
```

Listing 7.10: Die Hilfsklasse **MMClassFactoryVar**, welche das Klassen-Suffix **Impl** und eine **protected** Sichtbarkeit für Konstruktoren festlegt.

Aufbauend auf diesen adaptierenden Artefakten ist in Listing 7.11 das Schichtdefinitionsmodell gezeigt, in dem die erforderlichen Kompositionsoperationen definiert sind.

Die erste Kompositionsoperation (Zeile 3 – 4) drückt eine **before** Operation aus. Hierdurch wird die GR **SetterMethodBody** (siehe Listing 7.9, Zeile 1 – 3) aus dem Artefakt **Methods** des CGLs **factoryVariant** vor die gleichnamige VR des Artefakts **Methods** (siehe Lis-

```

1 compose layer factoryVariant with layer baseVariant {
2   // Template adaptations
3   ext.Methods:SetterMethodBody before
4     core.Methods:SetterMethodBody;
5   ext.Methods:FurtherMethods replace
6     core.Methods:FurtherMethods;
7   // Helper class adaptations
8   de.se-rwth.MMClassFactoryVar extends de.se-rwth.MMClass;
9 }

```

Listing 7.11: Schichtdefinitionsmodell für das Beispiel aus Abschnitt 7.2.

ting 7.5 auf Seite 181, Zeile 14 – 16) des CGLs `baseVariant` eingefügt. Infolge dieser Kompositionsoption wird an den Anfang jeder `set`-Methode eine `assert`-Anweisung eingefügt, welche die Gültigkeit des übergebenen Arguments prüft.

Die zweite Kompositionsoption (Zeile 5 – 6) ist eine `replace` Operation. Diese drückt aus, dass die GRV `FurtherMethods` (siehe Listing 7.9, Zeile 5 – 12) die gleichnamige VR (siehe Listing 7.5, Zeile 19 – 20) ersetzen soll. Erstgenannte GRV liegt im CGL `factoryVariant` im Artefakt `Methods` im Verzeichnis `ext`, letztgenannte VR im CGL `baseVariant` im gleichnamigen Artefakt im Verzeichnis `core`. Durch diese Kompositionsoption wird die Generierung der statischen Methode zur Erzeugung der Instanzen in das Template integriert.

Die letzte Kompositionsoption drückt eine `extends` Operation bezüglich der Hilfsklassen aus (Zeile 8). Dadurch ersetzen die Methoden `classSuffix()` und `constructorVisibility()` aus der Klasse `MMClassFactoryVar` die gleichnamigen Methoden aus der Klasse `MMClass`.

7.5.2 Konfiguration einer Generatorvariante

Basierend auf einem Schichtdefinitionsmodell wird eine konkrete Generatorvariante durch die Auswahl von CGLs festgelegt. Diese Auswahl erfolgt in einem Produktkonfigurationsmodell (siehe Definition 21), welches in der DSL `Product Configuration Language (PCL)` modelliert wird. Listing 7.12 zeigt die Grammatik zu dieser DSL.

Definition 21 (*Produktkonfigurationsmodell*)

Über ein Produktkonfigurationsmodell wird eine konkrete Generatorvariante konfiguriert. Der wichtigste Bestandteil dieser Konfiguration ist die Festlegung der CGLs, die für die Generatorvariante ausgewählt werden. Aus dieser Selektion wird abgeleitet, aus welchen CGLs die Generatorvariante komponiert werden muss.

Jedes Produktkonfigurationsmodell muss für die ausgewählte Variante einen eindeutigen Namen festlegen (Zeile 5). Darüber hinaus muss definiert werden, welche CGLs für die Generatorvariante ausgewählt werden (Zeile 6). Die spätere Komposition der Generatorartefakte erfolgt auf Basis dieser CGL-Auswahl. Die Entwickler müssen hierbei allerdings nicht vollständig sämtliche relevanten CGLs angeben. Stattdessen wird auf Basis der definierten CGLs automa-

```

1 grammar rwth.PCL with org.eclipse.xtext.common.Terminals
2 generate pCL "http://www.se.de/rwth/PCL"
3
4 productConfigurationModel:
5   'generator ' variantName = ID '{'
6   'layer = ' layer = ID (',' furtherLayers += ID)* ';'
7   ('output = ' output = STRING ';' )?
8   '}'
9 ;

```

Listing 7.12: Grammatik zur DSL PCL.

tisch abgeleitet, ob und welche weiteren CGLs für die Generatorvariante beachtet werden müssen. Dieser Aspekt wird in Abschnitt 7.5.3 erläutert.

Ferner kann in einem Produktkonfigurationsmodell festgelegt werden, in welches Ausgabeverzeichnis die zu erzeugende Generatorvariante geschrieben werden soll. Dies erfolgt über den Konfigurationsparameter `output` (Zeile 7). Fehlt eine `output` Angabe in einem Produktkonfigurationsmodell, werden die Artefakte der Generatorvariante in ein Standardverzeichnis geschrieben.

Anwendung auf das Beispiel

Ein Produktkonfigurationsmodell für das Beispiel aus Abschnitt 7.2 ist in Listing 7.13 angegeben. Die Generatorvariante, die durch dieses Produktkonfigurationsmodell konfiguriert wird, heißt `FactoryGenerator` (Zeile 1), wählt den CGL `factoryVariant` (Zeile 4) aus und die Artefakte der Generatorvariante werden in das Verzeichnis `genFactoryGenerator` geschrieben (Zeile 5).

```

1 generator FactoryGenerator {
2   //Composition operations for selected layer defined in
3   //layer definition model
4   layers = factoryVariant;
5   output = "genFactoryGenerator";
6 }

```

Listing 7.13: Produktkonfigurationsmodell durch das der CGL `factoryVariant` ausgewählt wird.

7.5.3 Automatische Ermittlung und Validitätsprüfung der benötigten CGLs

Bevor eine konkrete Generatorvariante erzeugt werden kann, wird vorab ermittelt, ob und welche CGLs zusätzlich zu den manuell im Produktkonfigurationsmodell selektierten CGLs automatisch selektiert werden müssen. Sobald sämtliche relevanten CGLs bekannt sind, wird eine Validitätsprüfung durchgeführt. Diese stellt sicher, dass die Komposition der resultierenden

CGLs möglich und valide ist. Diese beiden Schritte werden über die Abbildung des Schichtendefinitionsmodells auf einen gefärbten Graphen vollzogen. In diesem Graphen repräsentiert jeder Knoten eine GR, jede Kante zwischen zwei Knoten eine Kompositionsoperation zwischen den GRs und die Farbe eines Knotens repräsentiert den CGL, zu der eine GR gehört. Zur Konstruktion des Graphen wird das Schichtendefinitionsmodell ausgehend von den im Produktkonfigurationsmodell selektierten CGLs analysiert. Der Pseudocode dazu ist in Algorithmus 7.1 angegeben.

Zu jedem übergebenen CGL werden zu Beginn sämtliche Schichtenspezifikationen ermittelt, in denen Kompositionsoperationen für den CGL definiert sind (Zeile 2 – 3). Für jede Kompositionsoperation einer konkreten Schichtenspezifikation, die keine `extends` Operation darstellt, werden zwei Knoten erzeugt, falls diese nicht bereits existieren (Zeile 5 – 14): ein Knoten für die adaptierende GR bzw. GRV (Zeile 9 – 10) und ein Knoten für die adaptierte VR (Zeile 11 – 12). Ferner wird eine gerichtete Kante von dem Knoten für die adaptierende GR bzw. GRV hin zu dem Knoten für die adaptierte VR eingeführt. Diese ist mit dem Namen der Kompositionsoperation beschriftet (Zeile 13 – 14).

Kompositionsoperationen vom Typ `extends` werden speziell gehandhabt (Zeile 16 – 23). In einem solchen Fall werden die vorliegenden Artefakte untersucht und es wird bestimmt, welche VR gemäß `extends` durch welche GRV zu ersetzen ist (Zeile 16 – 17). Auf diese Paare wird die zuvor bereits vorgestellte Vorgehensweise angewendet. Es werden also Knoten für die GRVs und VRs erzeugt, falls sie nicht bereits existieren, und die Knoten werden über eine Kante verbunden (Zeile 18 – 23).

Jeder Knoten, der eine GR des aktuell betrachteten CGLs darstellt, wird in der gleichen Farbe gefärbt. Analog wird ein Knoten, der eine VR eines adaptierten CGLs darstellt, mit einer anderen Farbe gefärbt. Nachdem sämtliche Kompositionsoperationen des aktuell betrachteten CGLs verarbeitet wurden, wird die beschriebene Vorgehensweise für die adaptierten CGLs wiederholt, vorausgesetzt diese wurden bisher nicht verarbeitet (Zeile 24 – 25).

Aus dem resultierenden Graphen kann direkt abgeleitet werden, aus welchen CGLs die zu erzeugende Generatorvariante besteht. Jede Farbe, die in dem Graphen zur Färbung eines Knotens verwendet wurde, repräsentiert einen spezifischen CGL. Folglich werden sämtliche vorkommenden Farben bestimmt und daraus abgeleitet, welche CGLs benötigt werden. Ferner wird aufbauend auf dem resultierenden Graphen eine Validitätsprüfung durchgeführt. Ein Graph ist dann valide, wenn folgende Bedingungen erfüllt sind:

- Der Graph enthält keine Zyklen. Liegt im Graphen ein Zyklus vor, kann die Komposition nicht durchgeführt werden, da dies in einer Endlosschleife von Kompositionen ohne dedizierten Endpunkt resultieren würde. Ein solches Szenario muss manuell aufgelöst werden, beispielsweise durch eine Änderung der Kompositionsoperationen oder der CGL-Struktur.
- Jeder Knoten des Graphen besitzt maximal eine eingehende Kante für eine Ersetzungsoperation. Falls im Graphen ein Knoten mehrere eingehende Kanten für Ersetzungsoperationen besitzt, drückt das aus, dass eine VR durch mehrere GRVs ersetzt werden soll. Bei einer derartigen Situation kann nicht automatisch aufgelöst werden, durch welche GRV die VR im Zuge des Kompositionsschritts letztlich ersetzt werden soll. Wie im vorherigen Fall muss ein derartiges Szenario manuell durch eine Veränderung der Kompositi-

Algorithmus 7.1 : Function createCompositionGraph()

input : cgls: Manually selected CGLs according to product configuration model**output** : Composition graph

```

1 Create empty graph;
2 for each curCGL : cgls do
3   for each cglSpec : getCGLSpecsForLayer (curCGL) do
4     adaptedCGL ← getAdaptedCGL (cglSpec);
5     for each compositionOp : getCompositionOperations (cglSpec) do
6       adaptingGR ← getAdaptingGR (compositionOp);
7       adaptedVR ← getAdaptedVR (compositionOp);
8       if compositionOp does not represent extends operation then
9         adaptingVertex ← getOrCreateVertex (
10          graph, adaptingGR, curCGL);
11         adaptedVertex ← getOrCreateVertex (
12          graph, adaptedVR, adaptedCGL);
13         addEdge (graph, adaptingVertex,
14          compositionOp, adaptedVR);
15       else
16         for each adaptedVRTuple :
17           getAdaptedVRs (adaptingGR, adaptedVR) do
18             /* All vertices of the same CGL are colored with
19              the same color */
20             adaptingVertex ← getOrCreateVertex (graph,
21              adapting GR from adaptedVRTuple, curCGL);
22             adaptedVertex ← getOrCreateVertex (graph,
23              adapted VR from adaptedVRTuple, adaptedCGL);
24             addEdge (graph, adaptingVertex,
25              compositionOp, adaptedVR);
26         if adaptedCGL not visited then
27           createCompositionGraph (adaptedCGL);
28 return graph;

```

operationen oder eine Adaption der CGL-Struktur aufgelöst werden. Alternativ dazu ist es möglich, die Ersetzungsoperationen in einer festen Reihenfolge abzuarbeiten. Beispielsweise kann festgelegt werden, dass stets die erste Ersetzungsoperation automatisch verwendet wird. Ein derartiger Ansatz kann jedoch bei den Entwicklern zu Verwirrung führen, so dass diese Konstellation verboten wird.

Beispiele

Der Validierungsprozess wird nachfolgend auf Basis von Beispielen erläutert. Zur Vereinfachung der Beschreibung wird angenommen, dass sämtliche modellierten Kompositionsoperationen Ersetzungsoperationen darstellen.

Abbildung 7.18 zeigt auf der linken Seite die grafische Repräsentation eines Schichtendefinitionsmodells. Es wird davon ausgegangen, dass im Produktkonfigurationsmodell die CGLs L_3 und L_4 manuell ausgewählt wurden. Die Knoten des CGLs L_1 werden mit horizontalen Streifen, Knoten des CGLs L_2 mit vertikalen Streifen, Knoten des CGLs L_3 ohne Muster und Knoten des CGLs L_4 mit gepunktetem Muster dargestellt. Darüber hinaus gibt der hochgestellte Index eines Knotens C_i^j den CGL an, in dem das zugehörige Artefakt enthalten ist und der tiefgestellte Index kennzeichnet das betrachtete Artefakt. Somit repräsentiert beispielsweise der Knoten C_1^3 das Artefakt C_1 des CGLs L_3 .

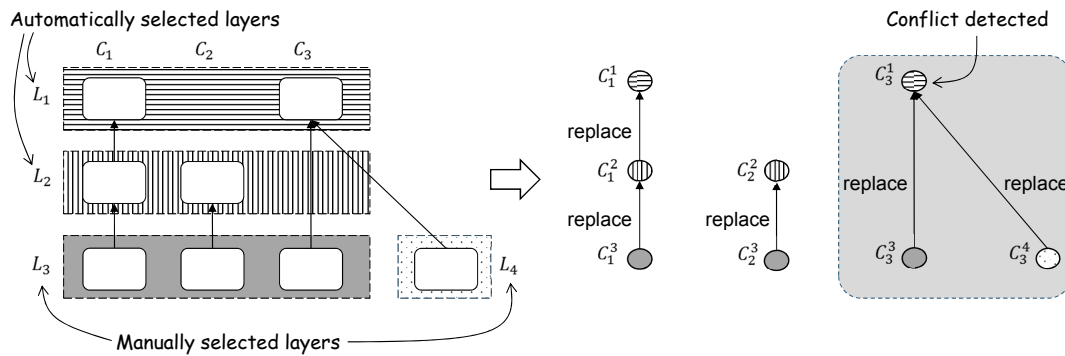


Abbildung 7.18: Manuelle Selektion zweier CGLs. Der Knoten C_3^1 enthält zwei eingehende Kanten, so dass die Validierung einen Fehler meldet.

Wird als Erstes die Komposition von L_3 und L_2 betrachtet, wird initial der Knoten C_1^3 erzeugt. Aufgrund der Kompositionsoperation zwischen C_1^3 und C_1^2 , wird der Knoten C_1^2 für den CGL L_2 erzeugt und C_1^3 wird mit diesem verbunden. Analog wird für die nächste Kompositionsoperation für L_3 der Knoten C_2^3 und für L_2 der Knoten C_2^2 erzeugt und C_2^3 mit C_2^2 verbunden. Da keine weiteren Kompositionsoperationen zwischen L_2 und L_3 definiert sind, fährt der Algorithmus mit der Schichtenspezifikation fort, gemäß der L_2 mit L_1 komponiert wird. Für diese Schichtenspezifikation ist lediglich eine Kompositionsoperation definiert, so dass für diese der Knoten C_1^1 erzeugt und eine Kante von C_1^1 zu diesem hinzugefügt wird. Daraufhin werden die Kompositionsoperationen, die zwischen L_3 und L_1 definiert sind, analysiert. Infolgedessen werden die Knoten C_3^1 und C_3^3 erzeugt und C_3^1 mit C_3^3 verbunden. Abschließend werden die

Kompositionsoperationen, die zwischen L_4 und L_1 definiert sind, betrachtet. Im Zuge dessen wird der Knoten C_3^4 erzeugt und mit C_3^1 verbunden. Eine weitere Expansion des Graphen ist nicht möglich, da sämtliche Kompositionsoperationen der ausgewählten CGLs untersucht wurden. Der resultierende Graph ist auf der rechten Seite in Abbildung 7.18 zu sehen. Aus diesem Graph kann inferiert werden, dass zusätzlich zu den manuell selektierten CGLs L_3 und L_4 die CGLs L_1 sowie L_2 automatisch selektiert werden müssen. Dies ist jedoch in diesem Beispiel hinfällig, da der Graph invalide ist. Der Grund dafür ist, dass der Knoten C_3^1 zwei eingehende Kanten besitzt, die Ersetzungsoperationen repräsentieren. Diese Konstellation stellt einen Konflikt dar, da im Zuge der Komposition nicht konfliktfrei automatisch entschieden werden kann, ob die VR nun durch die GRV aus L_3 oder L_4 ersetzt werden soll. Würde stattdessen gemäß Produktkonfigurationsmodell lediglich der CGL L_3 ausgewählt werden, würde dieser Konflikt nicht entstehen und der resultierende Graph wäre gültig.

Abschließend wird zudem das Szenario betrachtet, dass im Zuge der Validitätsprüfung ein Fehler aufgrund eines Zyklus gemeldet wird. Abbildung 7.19 zeigt zu diesem Zweck erneut eine grafische Repräsentation eines Schichtdefinitionsmodells, unter der Annahme das gemäß Produktkonfigurationsmodell der CGL L_3 manuell ausgewählt wurde. Der resultierende Graph enthält den Pfad (C_1^3, C_1^1) , da C_1^3 mit C_1^2 und C_1^2 mit C_1^1 komponiert wird. Darüber hinaus enthält der Graph den Pfad (C_1^1, C_1^3) , da C_1^1 mit C_1^3 komponiert wird. Im Graphen entsteht aufgrund dieser Konstellation ein Zyklus. Der Versuch diese Struktur bei der Komposition automatisch aufzulösen würde scheitern, da nicht klar bestimmt werden kann, welche VR durch welche GRV ersetzt werden soll und wann die Kette der Ersetzungsoperationen endet. Auch in diesem Beispiel würden automatisch die CGLs L_1 und L_2 selektiert werden, doch auch diese automatische CGL-Auswahl ist unerheblich, da der Graph invalide ist.

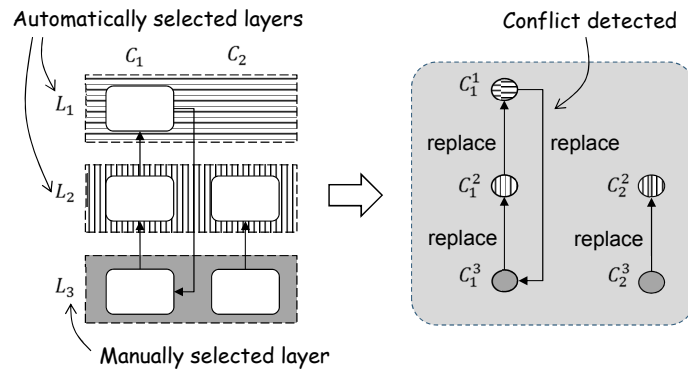


Abbildung 7.19: Manuelle Selektion eines CGLs. Der resultierende Graph enthält einen Zyklus, da ein Pfad (C_1^3, C_1^1) und ein Pfad (C_1^1, C_1^3) existiert.

7.6 Kompositionale Erzeugung von Generatorvarianten

Zur Durchführung der Komposition der CGLs gibt es im Allgemeinen zwei Optionen. Die erste Option betrifft die Komposition zur Laufzeit des Generators. Hierbei muss das Generatorfra-

network erweitert werden, so dass die Kompositionsoperationen zur Laufzeit des Generators umgesetzt werden können. Abbildung 7.20 verdeutlicht diesen Ansatz. In diesem Beispiel ruft die GR B die VR D auf, die durch die GRV F ersetzt wird. Die Komposition wird hierbei durchgeführt, in dem zur Laufzeit des Generators festgestellt wird, dass D durch F ersetzt werden soll. Das hat zur Folge, dass der Generator nicht die VR D aufruft, wie im Generatorartefakt angegeben, sondern das stattdessen die GRV F aufgerufen wird. Hervorzuheben ist hierbei, dass zur Abbildung der Komposition keine neuen Artefakte erzeugt werden.

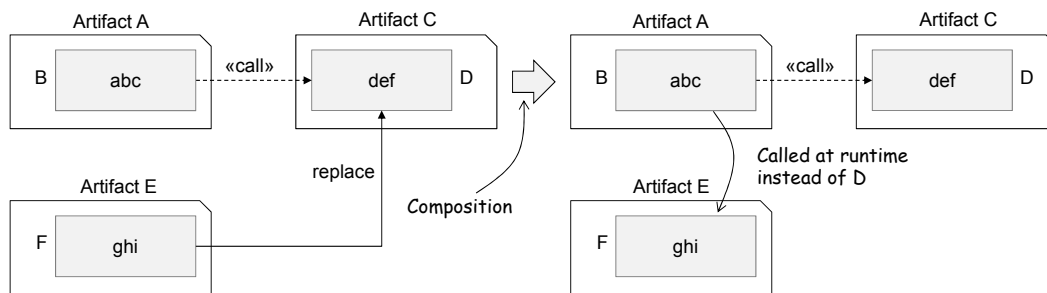


Abbildung 7.20: Komposition zur Laufzeit des Generators, ohne dass neue Generatorartefakte hierfür erzeugt werden.

Alternativ dazu kann die Komposition durchgeführt werden, indem auf Basis der bestehenden Generatorartefakte neue Generatorartefakte erzeugt werden. Diese neuen Artefakte werden so erzeugt, dass der Aufbau und Inhalt der neuen Artefakte die Kompositionsoperationen abbilden. Eine Kompositionsoperation `replace` führt in diesem Fall beispielsweise dazu, dass ein Teil eines Templates tatsächlich durch anderen Inhalt ersetzt wird. Dies wird in Abbildung 7.21 anhand eines Beispiels verdeutlicht. Der wesentliche Unterschied zum vorherigen Beispiel ist, dass dieses Mal zur Umsetzung der Komposition neue Generatorartefakte A' sowie C' erzeugt werden. Hervorzuheben ist hierbei, dass das Artefakt C' die GRV F anstelle der VR D enthält.

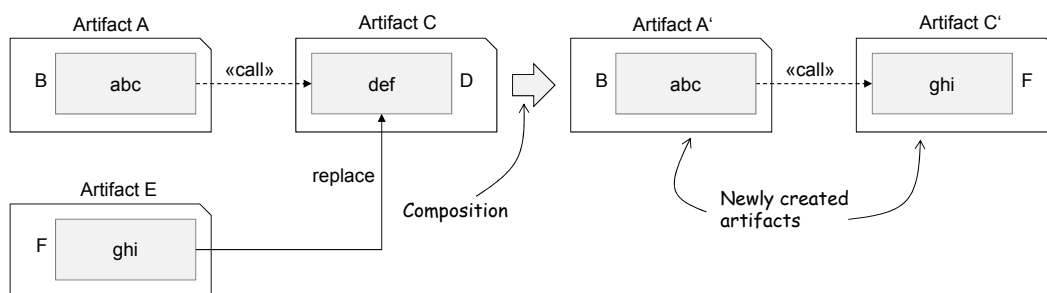


Abbildung 7.21: Komposition durch die Erzeugung neuer Artefakte. Die ursprüngliche VR wird in einem neuen Artefakt durch eine andere GRV ersetzt.

Die Hauptfrage bei diesem Kompositionsansatz ist, wie `before` und `after` Operationen umgesetzt werden. Wird eine `before` oder `after` Operation auf ein Template angewendet, kann eine Komposition beispielsweise durchgeführt werden, indem vor oder hinter den Aufruf

des Templates eine zusätzliche Anweisung eingeführt wird. Über diese kann dann das gewünschte Template aufgerufen werden. Abbildung 7.22 verdeutlicht diese Herangehensweise. In dem ursprünglichen Generator ruft die GR *B* die VR *D* auf. In dem Beispiel ist angegeben, dass die GR *F* vor diese VR eingefügt werden soll. Im Zuge der Komposition werden die neuen Generatorartefakte so erzeugt, dass die GR *B* vor dem Aufruf von VR *D* erst die GR *F* aufruft.

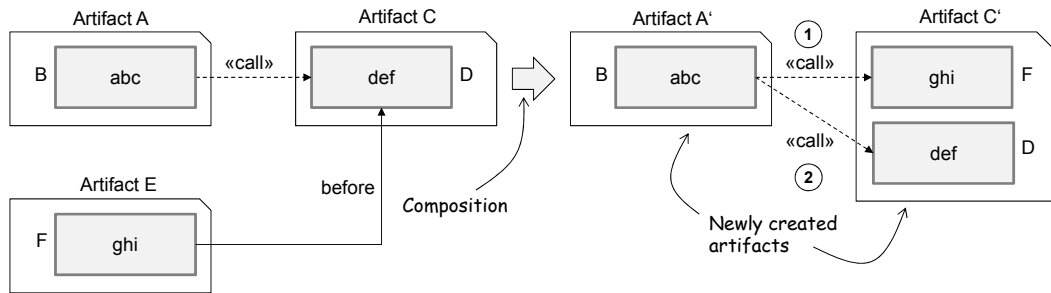


Abbildung 7.22: Komposition durch die Erzeugung neuer Artefakte. Eine GR wird vor einer VR aufgerufen, um die *before* Kompositionsoption abzubilden.

Alternativ dazu kann zur Umsetzung einer *before* Operation beispielsweise an den Anfang einer VR eine Anweisung eingefügt werden, über welche die GR aufgerufen wird, die vor die VR eingefügt werden soll. Analog kann mit *after* Operationen verfahren werden. Dieser Ansatz wird in dieser Arbeit verfolgt, wie noch detailliert in Abschnitt 7.6.3 erläutert wird. In Abbildung 7.22 würde folglich an den Anfang von VR *D* der Aufruf der GR *F* eingefügt werden.

Die Definition von *before* und *after* Kompositionsoptionen ist für Hilfsklassen normalerweise nicht sinnvoll, wie in Abschnitt 7.7 noch diskutiert wird. Prinzipiell sind die Auswirkungen dieser Operationen jedoch analog bei der Anwendung auf Hilfsklassen.

Der Vorteil an der Komposition durch die Erzeugung neuer Artefakte ist, dass sich die resultierenden Generatorartefakte syntaktisch nicht von Generatorartefakten unterscheiden, die ohne den Kompositionsansatz erstellt wurden. Ein Generatorframework muss folglich nicht angepasst werden, um das Kompositionsergebnis ausführen zu können. Diese Option kann daher in allen Generatorframeworks eingesetzt werden. Dies ist auch der Grund dafür, dass in dieser Arbeit dieser zweite Kompositionsansatz angewendet wird. Zusammenfassend zeigt Definition 22 die daraus resultierende Definition der Komposition, die dieser Arbeit zugrunde liegt.

Definition 22 (Komposition von Artefakten)

Komposition bezeichnet eine Operation, die zwei oder mehrere Artefakte zu einem neuen Artefakt zusammenfasst [Kra10].

Angewendet auf diese Arbeit bedeutet diese Definition, dass auf Basis der manuell und automatisch ermittelten CGLs neue Generatorartefakte erzeugt werden, welche die konfigurierte Generatorvariante repräsentieren. Dazu erfolgt eine Komposition dieser CGLs. Bei dieser wird jedes zu adaptierende Artefakt mit den zugehörigen adaptierenden Artefakten komponiert, um die entsprechende Variante des zu adaptierenden Artefakts zu erzeugen. Die genaue Vorgehensweise zur Durchführung der Komposition von CGLs wird im weiteren Verlauf dieses Abschnitts detailliert erläutert.

Die wesentlichen Schritte der Komposition werden auf Basis des Schichtdefinitionsmodells, welches in Abbildung 7.23 grafisch dargestellt ist, erläutert. Dieses Beispiel enthält im Gegensatz zu dem in Abschnitt 7.2 eingeführten Beispiel zusätzliche Besonderheiten, die es ermöglichen, die einzelnen Schritte der Komposition besser zu verdeutlichen. Eine Kompositionsoperation wird über einen Pfeil ausgedrückt, der mit dem Namen der Kompositionsoperation beschriftet ist. Ferner kennzeichnen gestrichelte Pfeile, dass ein Template ein anderes Template aufruft. Aus Platzgründen sind die in einem Template enthaltenen `DEFINE` Blöcke nebeneinander angegeben und nicht wie in einem realen Template untereinander. Überdies enthalten die `DEFINE` Blöcke lediglich fiktiven Inhalt. Es wird davon ausgegangen, dass in einem Produktkonfigurationsmodell mindestens die Layer L_3 und L_4 ausgewählt wurden, so dass sämtliche vier CGLs komponiert werden müssen.

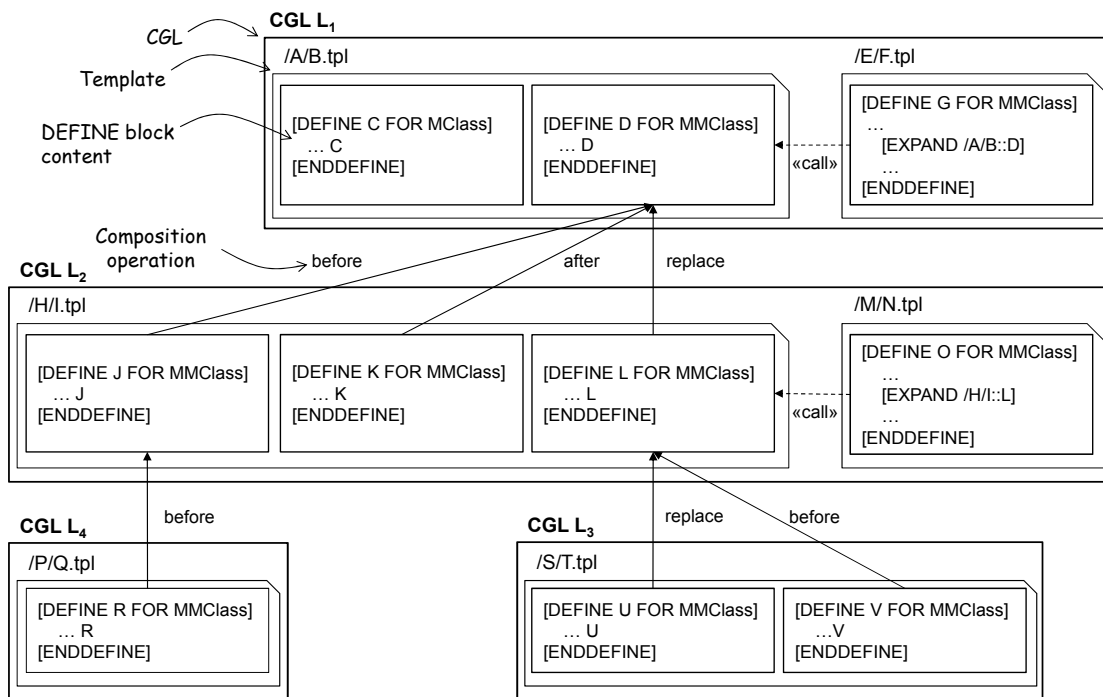


Abbildung 7.23: Grafische Repräsentation eines Schichtdefinitionsmodells, welches vier CGLs umfasst, in denen Kompositionsoperationen definiert sind.

7.6.1 Überblick über die Schritte zur Durchführung der Komposition

Nachfolgend wird ein Überblick über die einzelnen Schritte zur Durchführung der Komposition gegeben. Jeder Schritt wird im Anschluss daran detailliert erklärt.

1. Ermittlung der Auswirkungen der im Schichtdefinitionsmodell modellierten Kompositionsoperationen. In diesem Schritt werden für die manuell und automatisch selektierten CGLs sämtliche Kompositionsoperationen aus dem Schichtdefinitionsmodell verarbeitet. Auf diese Weise wird inferiert, wie die Struktur der Artefakte für die zu erzeugende

Generatorvariante sein muss. Die resultierende Datenstruktur gibt an, welche GRs vor oder hinter eine VR eingefügt werden sollen und ob und durch welche GRV eine VR ersetzt werden soll. Dieser Schritt wird in Abschnitt 7.6.2 erläutert.

2. Erzeugung der Generatorartefakte für die Generatorvariante. Aufbauend auf der im vorherigen Schritt konstruierten Struktur der Generatorartefakte werden neue Generatorartefakte erzeugt. Deren Inhalt bildet genau die Auswirkungen der modellierten Kompositionsoptionen ab. So werden vor und hinter VRs die entsprechenden GRs eingefügt und VRs werden durch die entsprechenden GRVs ersetzt. Dieser Schritt wird in Abschnitt 7.6.3 beschrieben.
3. Adaption der Generatorartefakte. Durch die Komposition der adaptierenden Artefakte mit den adaptierten Artefakten kann es dazu kommen, dass in den neu erzeugten Generatorartefakten auf die ursprünglichen Namen von VRs verwiesen wird, obwohl diese durch GRVs mit einem anderen Namen ersetzt wurden. Diese Aufrufstellen müssen angepasst werden, damit die neuen GRVs korrekt referenziert werden. Dieser Adaptionsschritt wird in Abschnitt 7.6.4 vorgestellt.

7.6.2 Ermittlung der Auswirkungen der im Schichtdefinitionsmodell modellierten Kompositionsoptionen

In diesem Schritt wird bestimmt, wie sich die im Schichtdefinitionsmodell modellierten Kompositionsoptionen auf die VRs der manuell und automatisch selektierten CGLs auswirken. Die Grundidee zur Umsetzung dieses Schritts wird nachfolgend kurz dargelegt. Darauf folgend wird die Umsetzung detaillierter vorgestellt.

Das Schichtdefinitionsmodell kapselt für sämtliche VRs, durch welche GRVs sie ersetzt werden und welche GRs vor oder hinter die VR eingefügt werden. Folglich muss für diesen Schritt das komplette Schichtdefinitionsmodell traversiert werden. Dabei werden zu den VRs jeweils folgende Informationen vermerkt:

- die Liste von GRs, die vor die VR eingefügt werden sollen
- die GRV, durch welche die VR ersetzt werden soll
- die Liste von GRs, die hinter die VR eingefügt werden sollen

Die Umsetzung dieses Schritts wird im Folgenden genauer vorgestellt. Der Pseudocode zum Aufbau einer Datenstruktur mit den zuvor genannten Informationen ist in Algorithmus 7.2 angegeben. Dieser Algorithmus unterscheidet sich geringfügig von der zuvor skizzierten Grundidee:

- Die genannten Informationen werden nur zu solchen VRs vermerkt, die nicht gleichzeitig GRVs darstellen, die also keine andere VR ersetzen. Sind Kompositionsoptionen für eine GRV angegeben, werden diese Informationen zu der ersetzten VR verwaltet. Auf diese Weise wirken sich die Kompositionsoptionen so aus, als wären sie direkt zu der ersetzten VR definiert wurden. Dadurch wird die Erzeugung der Generatorartefakte im nächsten Schritt vereinfacht.

- Zu einer VR wird eine Liste von GRVs verwaltet, welche die VR ersetzen. Enthält diese Liste zu einem Schlüssel A mehrere GRV Namen, z.B. B, C und D, so bedeutet das, dass A durch B, B durch C und C durch D ersetzt werden soll. Die verschiedenen GRVs werden verwaltet, um die Erzeugung der Generatorartefakte im nächsten Schritt zu vereinfachen.

Algorithmus 7.2 : Function `determineCompositionOperationConsequences()`

input : `ldm`: Layer definition model

output : Data structure `compositionResultVR`

```

1 Create empty map compositionResultVR;
2 graph ← createGraphFromLayerDefinitionModel(ldm);
3 introduceNumberingOfEdges(graph);
4 for each rootNode : getRootNodes(graph) do
5   | traverseEdgesOfNode(compositionResultVR, rootNode, rootNode);
6 return compositionResultVR;

```

Der erste Schritt zur Ermittlung der Auswirkungen der Kompositionsoperationen besteht in der Konstruktion des Graphen, der auch für die Validitätsprüfung aus dem Schichtendefinitionsmodell erzeugt wird, wie in Abschnitt 7.5.3 beschrieben (Zeile 2). Dieser Graph kapselt, analog zum Schichtendefinitionsmodell, auf welche Weise VRs mit anderen GRs komponiert werden. Um die Auswirkungen der Kompositionsoperationen auf die verschiedenen VRs abzuleiten, wird der komplette Graph traversiert.

Bevor diese Traversierung startet, wird der Graph um eine Kantenummerierung erweitert (Zeile 3). Diese Nummerierung wird eingeführt, damit bei der Traversierung entschieden werden kann, in welcher Reihenfolge die Kanten des Graphen zu traversieren sind. Die Kantenummerierung ist genau dann relevant, wenn ein Knoten mehrere eingehende Kanten für dieselbe Kompositionsoperation besitzt. Diese verschiedenen Kanten müssen deterministisch in einer bestimmten Reihenfolge durchlaufen werden. Für jeden Knoten des Graphen werden dabei die eingehenden Kanten nummeriert. Die Nummerierung der Kanten richtet sich danach, in welcher Reihenfolge im Schichtendefinitionsmodell die zugehörige Kompositionsoperation definiert ist. Besitzt ein Knoten n eingehende Kanten, erhält die Kante die Nummer eins, welche als Erstes im Schichtendefinitionsmodell definiert ist. Entsprechend wird der Kante die Nummer n zugeordnet, welche als Letztes im Schichtendefinitionsmodell definiert ist. Aufbauend auf dem um die Kantenummerierung erweiterten Graphen kann anschließend die Traversierung des Graphen durchgeführt werden. Diese startet top-down ausgehend von den Knoten des Graphen, die keine ausgehende Kante besitzen (Zeile 4 – 5).

Traversierung des Graphen

Die Traversierung des Graphen ist in Algorithmus 7.3 verdeutlicht. Bei dieser Traversierung wird zwischen dem Knoten unterschieden, der aktuell untersucht wird (`curNode`) und dem Knoten, auf den sich die Kompositionsoperationen für `curNode` auswirken (`affectedNode`). Diese Unterscheidung ist wichtig in Fällen, in denen ein Knoten A durch einen Knoten

B ersetzt wird. Sämtliche Kompositionsoperationen für Knoten B oder Knoten, die diesen ersetzen, werden zu Knoten A eingetragen. Dadurch wirken sich Kompositionsoperationen für einen ersetzenden Knoten so aus, als wären sie direkt für den Ursprungsknoten definiert wurden.

Algorithmus 7.3 : Function `traverseEdgesOfNode()`

```

input   : compositionResultVR: Data structure reflecting consequences of composition
           operations,
           affectedNode: Node the composition operations are stored for,
           curNode: Node whose incoming edges are currently analyzed
modifies: compositionResultVR: Data structure is filled with entries in the course of the
           graph traversal

1 crEntry ← Get entry for name of affectedNode from compositionResultVR;
2 if crEntry does not exist then
3   Create empty list beforeList;
4   Create empty list replacedByList;
5   Create empty list afterList;
6   crEntry ← <Name of affectedNode, <beforeList, replacedByList, afterList > >;
7   Add crEntry to compositionResultVR;
8 for each edge : getIncomingEdgesSortedByEdgeNumber (curNode) do
9   sourceName ← getSourceNodeName (edge) ;
10  crEntrySN ← Get entry for sourceName from compositionResultVR;
11  if edge represents before operation then
12    Append sourceName to crEntry.beforeList;
13  else if edge represents after operation then
14    Append sourceName to crEntry.afterList;
15  else if edge represents replace operation then
16    Append sourceName to crEntry.replacedByList;
17 for each edge : getIncomingEdgesSortedByEdgeNumber (curNode) do
18   sourceNode ← getSourceNodeName (edge) ;
19   nextAffectedNode ← sourceNode;
20   if edge represents replace operation then
21     nextAffectedNode ← affectedNode;
22   traverseEdgesOfNode (compositionResultVR, nextAffectedNode,
23     sourceNode) ;

```

Zu Beginn des Algorithmus wird der Eintrag ermittelt, der in `compositionResultVR` zu `affectedNode` hinterlegt ist (Zeile 1). Existiert dieser Eintrag noch nicht, wird ein neuer Eintrag zu diesem Schlüssel hinzugefügt (Zeile 2 – 7). Dieser neue Eintrag enthält leere Listen `beforeList` (Zeile 3), `replacedByList` (Zeile 4) und `afterList` (Zeile 5).

Zur Durchführung der Traversierung werden sämtliche eingehenden Kanten des aktuell be-

trachteten Knotens `curNode` abgearbeitet, gemäß der Reihenfolge, die sich aus der Nummerierung der Kanten ergibt (Zeile 8 – 16). Für jede Kante wird als Erstes der Name des Quellknotens der Kante bestimmt (Zeile 9). Dieser Name identifiziert die GR, mit dem der aktuell betrachtete Knoten `curNode` (bzw. `affectedNode`) komponiert wird. Die Behandlung einer Kante richtet sich danach, welche Kompositionsoperation durch die Kante ausgedrückt wird:

- Handelt es sich um eine Kante für eine `before` Kompositionsoperation, so dass gilt `sourceName before curNode`, wird `sourceName` an das Ende der Liste `beforeList` des Tupels `crEntry` angehängen (Zeile 11 – 12).
- Handelt es sich um eine Kante für eine `after` Kompositionsoperation, so dass gilt `sourceName after curNode`, wird `sourceName` analog an das Ende der Liste `afterList` des Tupels `crEntry` angehängen (Zeile 13 – 14).
- Repräsentiert die Kante eine `replace` Kompositionsoperation, so dass gilt `sourceName replace curNode`, wird `sourceName` gleichermaßen an das Ende der Liste `replacedByList` des Tupels `crEntry` angehängen (Zeile 15 – 16). Zu beachten ist hierbei, dass eine `replace` Kante im Graphen nicht nur vorliegt, wenn der Entwickler eine `replace` Kompositionsoperation unmittelbar angibt. Stattdessen werden bei der Konstruktion des Graphen ebenfalls `extends` Kompositionsoperationen auf einzelne `replace` Kompositionsoperationen abgebildet. Folglich muss bei dieser Prüfung nicht auf `extends` geprüft werden.

Aus dieser Behandlung der eingehenden Kanten kann direkt abgeleitet werden, dass die Nummerierung der Kanten das Kompositionsergebnis beeinflussen kann. Besitzt ein Knoten zwei oder mehr eingehende Kanten zur gleichen Kompositionsoperation, entscheidet die Nummerierung der Kanten, wie der zugehörige Quellknoten in `beforeList` oder `afterList` positioniert ist. Der Fall, dass ein Knoten zwei eingehende `replace` Kanten besitzt, wird durch die Validitätsprüfung des Graphen verhindert, wie in Abschnitt 7.5.3 erläutert.

Die beschriebene Vorgehensweise wird nach der Abarbeitung sämtlicher eingehender Kanten für die Quellknoten dieser Kanten wiederholt (Zeile 17 – 23). Besonders hervorzuheben ist hierbei die Konstellation, dass die Traversierung für einen Quellknoten fortgesetzt wird, der eine GRV repräsentiert (Zeile 20 – 21). In dieser Situation bleibt der von den Änderungen betroffene Knoten `affectedNode` unverändert. Auf diese Weise wirken sich Kompositionsoperationen, die für den betrachteten Quellknoten definiert sind, so aus, als wären sie direkt für den Knoten `affectedNode` definiert wurden. Wird der Quellknoten beispielsweise durch einen anderen Knoten ersetzt, muss sich diese Ersetzungsoperation auf die Informationen auswirken, die in `compositionResultVR` zu `affectedNode` vermerkt sind.

Da Zyklen in dem Graphen verboten sind, wie in der Validierungsphase in Abschnitt 7.5.3 erläutert, ist gesichert, dass die Traversierung nicht endlos ausgeführt wird.

Die beschriebene Herangehensweise ist darauf ausgerichtet, dass eine GR nicht vor oder hinter mehrere VRs eingefügt wird und nicht mehrere VRs ersetzt. Diese Fälle können durch eine leichte Adaptierung der Vorgehensweise ebenfalls in die Komposition einbezogen werden. In der Beschreibung wurde allerdings davon abstrahiert, um den Fokus auf den wesentlichen Teil des Algorithmus zu legen.

Anwendung auf das Beispiel

Die beschriebene Vorgehensweise wird nachfolgend auf das Beispiel aus Abbildung 7.23 auf Seite 199 angewendet. Abbildung 7.24 zeigt zu diesem Zweck den Graphen, der für das Schichtendefinitionsmodell erzeugt wird. Dabei wird angenommen, dass die CGLs L_3 und L_4 im Produktkonfigurationsmodell ausgewählt wurden. Da die textuelle Definition des Schichtendefinitionsmodells nicht gezeigt wurde, richtet sich die Nummerierung der Kanten nach dem Vorkommen der Kanten von links nach rechts. In dem Graphen ist die Nummerierung allerdings irrelevant, da kein Knoten mehrere eingehende Kanten derselben Kompositionsoperation besitzt.

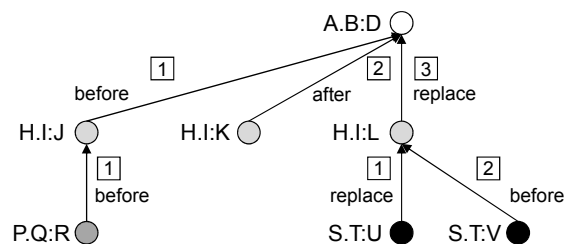


Abbildung 7.24: Graph zu dem Schichtendefinitionsmodell aus Abbildung 7.23 auf Seite 199.

Bei diesem Graphen startet die Traversierung mit dem Knoten $A.B:D$, da dieser keine ausgehenden Kanten besitzt. Aufgrund der eingehenden *before* Kante wird $H.I:J$ zu *beforeList* von $A.B:D$ hinzugefügt. Analog wird die GR $H.I:K$ zu *afterList* und die GRV $H.I:L$ zu *replacedByList* von $A.B:D$ hinzugefügt. Abbildung 7.25 illustriert den Inhalt der Datenstruktur *compositionResultVR* nach der Betrachtung aller eingehenden Kanten des Knotens $A.B:D$. Bereits verarbeitete Knoten sind mit einem gestrichelten Muster versehen und bereits verarbeitete Kanten sind gestrichelt markiert.

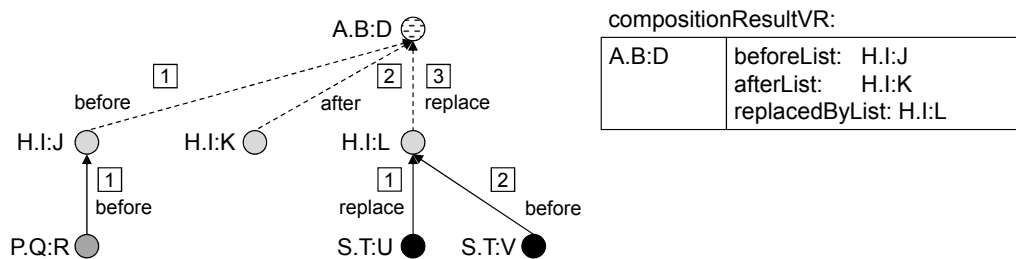


Abbildung 7.25: Graph nach Betrachtung des Knotens $A.B:D$.

Darauffolgend werden die eingehenden Kanten des Knotens $H.I:J$ verarbeitet. Da noch kein Eintrag zum Schlüssel $H.I:J$ existiert, wird ein neuer Eintrag zu *compositionResultVR* hinzugefügt. Anschließend wird die GR $P.Q:R$ zu *beforeList* zu diesem neuen Eintrag hinzugefügt. Abbildung 7.26 illustriert den Inhalt der Datenstruktur *compositionResultVR* nach der Betrachtung aller eingehenden Kanten des Knotens $H.I:J$.

Die Traversierung fährt anschließend mit dem Knoten $P.Q:R$ fort. Dieser besitzt jedoch keine

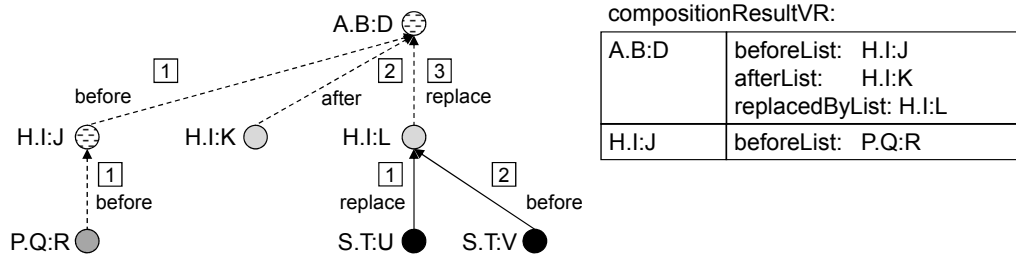


Abbildung 7.26: Graph nach Betrachtung des Knotens H.I:J.

eingehenden Kanten, so dass diese Iteration keine Auswirkungen auf `compositionResultVR` hat. Gleiches gilt für die darauffolgende Betrachtung des Knotens H.I:K.

Daraufhin wird der Knoten H.I:L betrachtet. In diesem Fall wird als Knoten, der von den Änderungen betroffen ist, der vorherige Knoten A.B:D beibehalten. Der Grund hierfür ist, dass die GRV H.I:L die VR A.B:D ersetzt. Die für diesen Knoten eingehenden Kanten sollen folglich so behandelt werden, als wären es eingehende Kanten des Knotens A.B:D. Als Ergebnis dieses Traversierungsschritts wird die GRV S.T:U an die Liste `replacedByList` zum Schlüssel A.B:D angehängen und die GR S.T:V wird zu `beforeList` zum Schlüssel A.B:D hinzugefügt. Abbildung 7.27 zeigt den Zustand des Graphen und den Inhalt von `compositionResultVR` nach Behandlung des Knotens H.I:L. An diesem Inhalt ändert sich in diesem Beispiel nichts mehr, da die weiteren Knoten keine eingehenden Kanten besitzen.

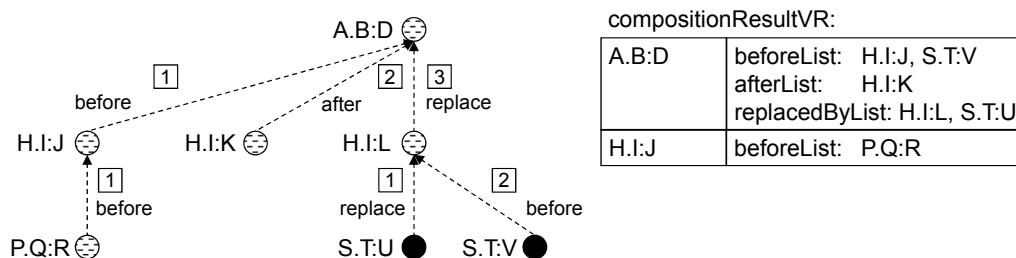


Abbildung 7.27: Graph nach Betrachtung des Knotens H.I:L.

Aus dem resultierenden Inhalt der Datenstruktur `compositionResultVR` kann abgelesen werden, dass die GRs H.I:J und S.T:V vor die VR A.B:D und die GR H.I:K hinter die VR eingefügt werden soll. Weiterhin geht daraus hervor, dass die VR A.B:D durch die GRV H.I:L und diese durch S.T:U ersetzt werden soll und das P.Q:R vor H.I:J eingefügt werden soll.

7.6.3 Erzeugung der Generatorartefakte der Generatorvariante

Auf Basis auf der im vorherigen Schritt aufgebauten Datenstruktur `compositionResultVR` werden neue Generatorartefakte für die Generatorvariante erzeugt. Dabei wird ausgenutzt, dass diese Datenstruktur unmittelbar angibt, durch welche GRV eine VR ersetzt werden muss und welche GRs vor und hinter eine VR eingefügt werden müssen. Der wichtigste Schritt zur Erzeugung der Generatorartefakte besteht folglich darin, die modellierten Kompositionsoperationen

auf die betroffenen Artefakte abzubilden. Dies wird wie folgt gehandhabt, wobei davon ausgegangen wird, dass die VR X_i ein Schlüssel in der Datenstruktur `compositionResultVR` ist:

1. Sind in `compositionResultVR` zu X_i Werte in `replacedByList` eingetragen, wird X_i durch die GRV ersetzt, die als Letztes in `replacedByList` eingetragen ist. Angenommen die GRV Y_j ist diese GRV. Enthält `compositionResultVR` Einträge zu VRs, die in Y_j enthalten sind, werden erst die Operationen für diese VRs verarbeitet, bevor X_i durch Y_j ersetzt wird. Dies ist darin begründet, dass der Inhalt von Y_j durch diese weiteren Kompositionsoperationen beeinflusst wird. Werden erst diese Kompositionsoperationen ausgewertet, ist gesichert, dass alle für Y_j relevanten Änderungen beachtet wurden. Folglich kann anschließend X_i durch Y_j ersetzt werden.
2. Falls die Datenstruktur `compositionResultVR` ausdrückt, dass GRs vor X_i eingefügt werden sollen, so wird an den Beginn der VR X_i für jede einzufügende GR eine Anweisung eingefügt, über welche die entsprechende GR aufgerufen wird. Im Falle von Templates wird also ein anderes Template (bzw. ein Teil eines anderen Templates) aufgerufen. Alternativ dazu kann der Inhalt aus der GR extrahiert und an den Beginn der VR eingefügt werden.
3. Analog zur Handhabung der GRs die vor eine VR eingefügt werden sollen, werden die GRs behandelt, die hinter X_i eingefügt werden sollen. In diesem Fall werden die entsprechenden Anweisungen allerdings an das Ende von X_i eingefügt.

Abschließend können die Generatorartefakte in manchen Fällen noch bereinigt werden. Für jede GRV X_j eines Artefakts X kann X_j aus X entfernt werden. Dies ist möglich, da die ersetzende GRV durch die vorherigen Schritte in das adaptierte Artefakt integriert wurde. Dieser Schritt ist nicht zwingend erforderlich, sondern sorgt lediglich dafür, dass die Artefakte einer konstruierten Generatorvariante keine GRVs enthalten, die nicht mehr benötigt werden.

Im Folgenden wird dargelegt, wie das Einfügen von Anweisungen zur Handhabung von `before` und `after` Operationen umgesetzt wird. Darüber hinaus wird ein Spezialfall bei der Umsetzung von Ersetzungsoperationen vorgestellt.

Einfügen von Anweisungen zur Handhabung von `before` und `after`

Die Datenstruktur `compositionResultVR` gibt an, welche GRs vor oder hinter welche VRs eingefügt werden sollen. Dazu verweist die Datenstruktur auf die Signaturen der betroffenen GRs und VRs. Auf Basis dieser Informationen werden Anweisungen zum Aufrufen der GRs eingefügt. Weicht die Notation zum Aufrufen von Artefakten davon ab, wie die Signaturen angegeben werden, ist aber nicht direkt ersichtlich, wie diese Anweisungen eingefügt werden können.

Zur Lösung dieses Problems wurde eine Ableitungsvorschrift entwickelt. Diese konstruiert auf Basis einer GR- bzw. VR-Signatur den Namen, wie er im Artefakt angegeben wird, um den durch die GR repräsentierten Bereich in der VR aufzurufen. Soll also eine spezifische GR vor oder hinter eine VR eingefügt werden, wird mit Hilfe dieser Ableitungsvorschrift der Name

konstruiert, über den die GR korrekt aufgerufen werden kann. Dieser Name wird anschließend innerhalb der Anweisung genutzt, die an den Anfang oder das Ende der VR eingefügt wird, um die `before` oder `after` Operation abzubilden.

In dieser Arbeit, in der XPand als Templatesprache eingesetzt wird, werden `EXPAND` Anweisungen an den Anfang oder das Ende einer VR eingeführt. Über diese werden die `DEFINE` Blöcke aufgerufen, die vor oder hinter die VR eingefügt werden sollen. Dazu wird ausgenutzt, dass in dieser Arbeit GR-Signaturen aus zwei Teilen bestehen: dem Pfad zu dem Artefakt und dem Pfad zu der GR innerhalb des Artefakts. In den Beispielen in dieser Arbeit sind die einzelnen Elemente des Pfads zum Artefakt über einen Punkt voneinander getrennt. Darauf folgend trennt ein Doppelpunkt die Pfadangaben zum Artefakt von den Angaben zur Navigation hin zur GR. Für derartige Signaturen sieht die Ableitungsvorschrift für XPand Templates wie folgt aus:

- Die Punkte in der Pfadangabe zum Artefakt werden durch Slashes ersetzt und an den Anfang wird einleitend ebenfalls ein Slash eingefügt.
- Der Doppelpunkt wird durch zwei Doppelpunkte ersetzt.
- Auf den Doppelpunkt folgt der Name der GR - bei XPand der Name des `DEFINE` Blocks.

Auf diese Weise wird beispielsweise die Signatur `ext.Methods.SetterMethodBody` durch den Namen `/ext/Methods::SetterMethodBody` ersetzt. Soll eine GR mit der genannten Signatur vor eine VR eingefügt werden, wird somit folgende Anweisung an den Beginn der VR eingefügt: `[EXPAND /ext/Methods::SetterMethodBody]`.

Für Java-Hilfsklassen wurde keine Ableitungsvorschrift entwickelt, da die Anwendung von `before` oder `after` Kompositionsoperationen für Hilfsklassen nicht unterstützt wird und die Ableitungsvorschrift nur in diesen Fällen benötigt wird.

Handhabung von `super` Methodenaufrufen

Gemäß der bisher beschriebenen Vorgehensweise wird eine VR X_i durch eine GRV Y_j ersetzt, falls Y_j als Letztes in `replacedByList` zu X_i eingetragen ist. Dies kann jedoch zu Problemen führen, wenn Y_j eine Methode einer Hilfsklasse repräsentiert und `super` Aufrufe enthält. Wird X_i hier unmittelbar durch Y_j ersetzt und es werden keine weiteren Schritte unternommen, verweist Y_j weiterhin auf `super` Aufrufe. Diese Aufrufe sind aber unter Umständen nicht mehr gültig, da Y_j durch den Ersetzungsschritt in die adaptierte Hilfsklasse eingefügt wurde, die X_i enthält. Die resultierende Hilfsklasse kann aufgrund der `super` Aufrufe somit fehlerhaft sein.

Um dieses Problem zu vermeiden, wird initial ermittelt, ob eine VR X_i einer Hilfsklasse durch eine GRV Y_j ersetzt werden soll, die `super` Aufrufe enthält. Liegt dieser Fall vor, wird die VR X_i nicht unmittelbar durch die GRV Y_i ersetzt. Stattdessen werden in Y_i diese `super` Aufrufe angepasst. Ferner wird die über `super` aufgerufene Methode unter einem neuen Namen in die Hilfsklasse integriert, wenn der `super` Aufruf an die zu ersetzende Methode selbst geht. Die Umsetzung dieser Schritte ist nachfolgend genauer beschrieben. Der Pseudocode in Algorithmus 7.4 zeigt die Umsetzung dieser Schritte. Als Resultat liefert der Pseudocode den Code, durch den die VR X_i ersetzt werden muss.

1. Erzeugung des ASTs zu der ersetzenden Methode (Zeile 2).
2. Ermittlung sämtlicher `super` Methodenaufrufe in der ersetzenden Methode auf Basis des zuvor erzeugten ASTs (Zeile 3).
3. Anpassung der `super` Methodenaufrufe durch die Entfernung des Schlüsselworts `super` (Zeile 5). Als Resultat liegt ein normaler Methodenaufruf vor. Der Grund für diesen Schritt ist, dass die GRVs in die ursprüngliche Hilfsklasse eingefügt werden. Ein `super` Methodenaufruf kann sich aber auf eine Methode dieser ursprünglichen Hilfsklasse bezogen haben. Würde ein solcher `super` Methodenaufruf beibehalten werden, wäre die resultierende Hilfsklasse syntaktisch falsch.
4. Handelt es sich bei dem `super` Aufruf um den Aufruf der Methode, die ersetzt werden soll, wird die ursprüngliche Implementierung der zu ersetzenden Methode aus der Oberklasse unter einem anderen Namen in die Hilfsklasse eingefügt (Zeile 6 – 9). Weiterhin wird der ursprüngliche Methodenaufruf so angepasst, dass der Name der aufgerufenen Methode auf den neuen Methodennamen angepasst wird (Zeile 10). In diesem Sonderfall findet also keine direkte Ersetzung von X_i durch Y_j statt, um die zuvor beschriebenen Probleme zu vermeiden.

Algorithmus 7.4 : Function `handleSuperInvocations()`

```

input    : method: Method which is analyzed for super invocations,
            compositionCode: Code through which  $X_i$  will be replaced by,
            extendedReplacedByList: List of GRVs replacing a specific VR,
            index: Index of element in list, which is currently analyzed
modifies : compositionCode: Code with adapted super invocations
            through which  $X_i$  will be replaced by

1 curReplacedElem  $\leftarrow$  Element of extendedReplacedByList at position index - 1;
2 ast  $\leftarrow$  Create AST for method;
3 superInvocations  $\leftarrow$  getSuperInvocations (ast);
4 for each superInvocation : superInvocations do
5     Remove super keyword from superInvocation;
6     if superInvocation invokes method curReplacedElem then
7         method  $\leftarrow$  curReplacedElem;
8         Append code of method to compositionCode;
9         Rename name of method into other unique name newName;
10        Adapt superInvocation to invoke newName;
11 if superInvocations not empty and index - 1 > 0 then
12     handleSuperInvocations (method, compositionCode,
13         extendedReplacedByList, index - 1);

```

Im Zuge der Ausführung von Algorithmus 7.4 wird der Code ermittelt, durch den die VR X_i ersetzt werden muss. Initial handelt es sich dabei um die Implementierung der Methode Y_j , die

durch den Algorithmus als Erstes auf `super` Aufrufe hin untersucht wird. Ferner wird dem Algorithmus eine Liste übergeben, die sich aus dem Inhalt von `replacedByList` zu X_i ergibt, wobei zusätzlich als erstes Element noch X_i in diese Liste hinzugefügt wird. Dadurch beinhaltet die Liste an Position k ein Element, welches durch das Element an Position $k + 1$ ersetzt wird. Der Algorithmus startet die Analyse dieser Liste von hinten nach vorne. Die Vorgehensweise wird so lange wiederholt, bis entweder alle Elemente der Liste `extendedReplacedByList` abgearbeitet wurden oder bis die aktuell betrachtete Methode keine `super` Aufrufe enthält (Zeile 11 – 13).

Im Folgenden wird die Handhabung der `super` Methodenaufrufe demonstriert. Dazu wird das Beispiel aus Abschnitt 7.2 fiktiv so angepasst, dass die Methode `classSuffix()` als Suffix „Impl“ zurückliefert und die überschreibende Hilfsklasse die ursprüngliche Methode über `super` aufruft. Dies ist in Abbildung 7.28 dargestellt. In diesem Szenario enthält die Datenstruktur `compositionResultVR` zur VR `de.MMClass: classSuffix` in `replacedByList` den Wert `de.MMClassFactoryVar: classSuffix`. Aus diesem Grund wird initial untersucht, welche `super` Methodenaufrufe in `de.MMClassFactoryVar: classSuffix` enthalten sind. Da die Methode einen `super` Methodenaufruf an die zu ersetzende Methode enthält, wird diese aus `de.MMClass` zu dem Kompositionscode hinzugefügt, durch den die Methode nachher ersetzt wird (Zeile 8 in Algorithmus 7.4). Weiterhin wird diese Methode umbenannt (Zeile 9), in `classSuffixL1` in diesem Fall, und der ursprüngliche `super` Methodenaufruf wird so angepasst, dass diese umbenannte Methode aufgerufen wird (Zeile 10). Folglich wird die Implementierung der Methode `classSuffix()` aus `MMClass` im Zuge der Komposition durch die Methode `classSuffix()` aus `MMClassFactoryVar` und die Methode `classSuffixL1()` ersetzt. Das Resultat der Komposition ist auf der rechten Seite in Abbildung 7.28 zu sehen.

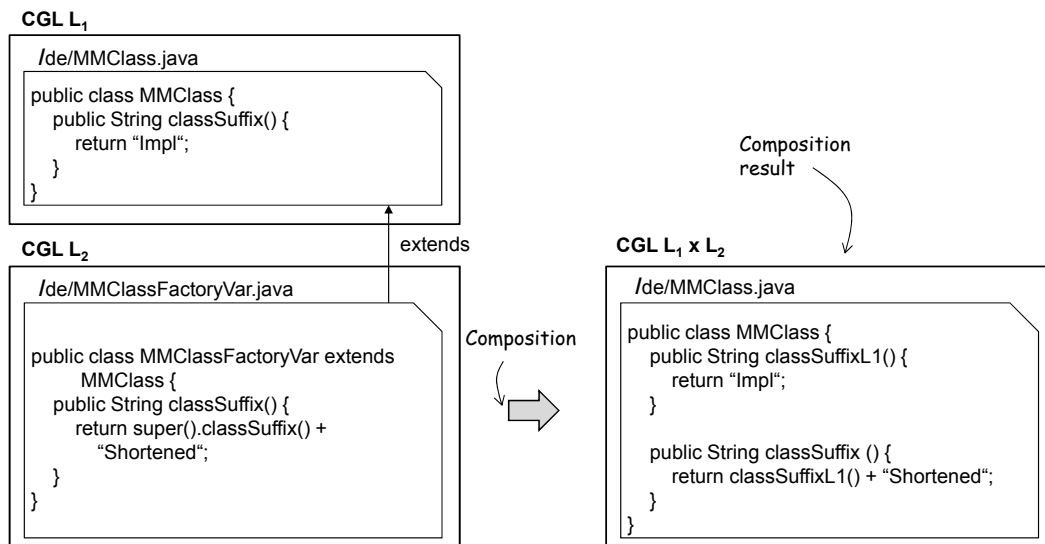


Abbildung 7.28: Handhabung von `super` Methodenaufrufen bei der Komposition von Java Hilfsklassen.

Anwendung auf das Beispiel

Die Vorgehensweise zur Erzeugung der Generatorartefakte wird nachfolgend auf das Beispiel aus Abbildung 7.23 auf Seite 199 angewendet. Der Inhalt der Datenstruktur `composition-ResultVR` für das Beispiel ist in Abbildung 7.27 auf Seite 205 zu sehen, wobei nur die Einträge gezeigt sind, die mit Werten befüllt sind.

Zu der VR `A.B:D` enthält die Datenstruktur sowohl Angaben zu `beforeList`, `afterList` als auch zu `replacedByList`. Folglich wird die VR als Erstes durch den Inhalt von `S.T:U` ersetzt, da diese GRV als Letztes in `replacedByList` eingetragen ist. Daraufaufgehend werden zwei `EXPAND` Anweisungen an den Anfang dieser VR eingefügt, da die Datenstruktur zwei Einträge zu `beforeList` enthält. Die beiden Einträge werden überführt in die beiden folgenden Anweisungen: `[EXPAND /H/I::J]` und `[EXPAND /S/T::V]`. Analog wird mit dem Eintrag in `afterList` verfahren. Hierbei wird an das Ende der VR die GR `H.I:K` über folgende Anweisung eingefügt: `[EXPAND /H/I::K]`.

Gleichermaßen wird für die VR `H.I:J` verfahren. Die Datenstruktur `compositionResultVR` drückt aus, dass die GR `P.Q:R` vor diese VR eingefügt werden soll. Folglich wird an den Anfang der VR folgende Anweisung eingefügt: `[EXPAND /P/Q::R]`.

Ferner wird die GRV `H.I:L` aus dem Template `I` und die GRV `S.T:U` aus dem Template `T` entfernt, da beide GRVs eine VR ersetzen. Der ursprüngliche Inhalt der GRV `S.T:U` wurde in das Template `B` integriert, so dass die Löschoperationen unproblematisch sind. Abbildung 7.29 zeigt den Inhalt der Artefakte nach diesen Änderungsoperationen.

7.6.4 Adaption der Generatorartefakte zur Anpassung der Namen

Die Generatorartefakte, die im vorherigen Schritt erzeugt wurden, können nicht zwangsläufig unverändert in einem Generator eingesetzt werden. Dies ist darin begründet, dass die aus der Komposition resultierenden Artefakte unter Umständen auf Artefakte verweisen, die in der finalen Generatorvariante gar nicht mehr existieren. Darüber hinaus kann es vorkommen, dass die Artefakte auf Bereiche verweisen, die durch die Komposition umbenannt wurden. Dieses Problem wird nachfolgend für Abbildung 7.29 verdeutlicht. Daraufaufgehend wird dargelegt, wie dieses Problem adressiert wird.

In Abbildung 7.29 ist der Inhalt der Generatorartefakte nach Ausführung des letzten Schritts dargestellt. Wie dort zu sehen ist, verweist der `DEFINE` Block `G` aus Template `F` auf den `DEFINE` Block `D` des Templates `B`. Im Zuge der Komposition wurde dieser `DEFINE` Block allerdings durch den `DEFINE` Block `U` des Templates `T` ersetzt. Die Folge davon ist, dass das Template `B` den `DEFINE` Block `U` anstelle des referenzierten `DEFINE` Blocks `D` enthält. Darüber hinaus verweist der `DEFINE` Block `O` aus Template `N` auf den `DEFINE` Block `L` des Templates `I`. Diese Referenz ist ebenfalls unzulässig, da der referenzierte `DEFINE` Block ebenso durch den `DEFINE` Block `U` ersetzt wurde. In beiden Fällen rufen die `DEFINE` Blöcke also `DEFINE` Blöcke auf, die im Kompositionsergebnis nicht mehr unter den ursprünglichen Namen existieren. Ein Generator bestehend aus diesen Artefakten kann somit nicht erfolgreich ausgeführt werden.

Aufgrund der geschilderten Problematik muss für die Generatorartefakte geprüft werden, ob sie angepasst werden müssen. Die Idee zur Durchführung dieser Prüfung ist nachfolgend aufgeführt. Die Umsetzung wird daraufaufgehend genauer erläutert.

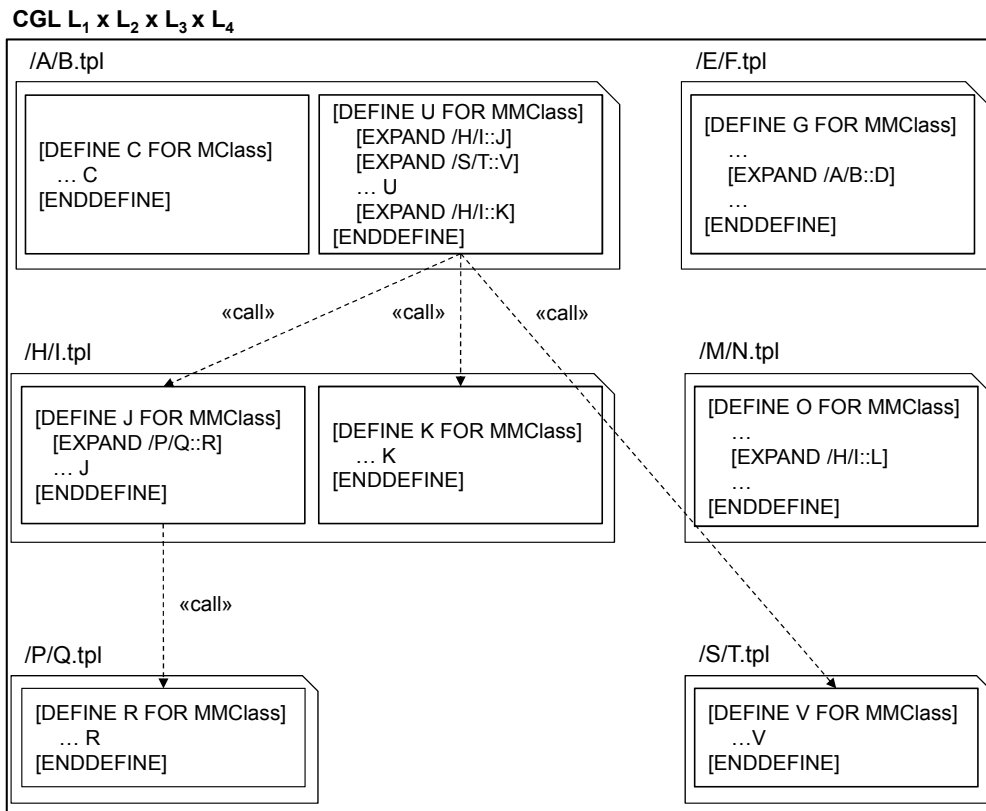


Abbildung 7.29: Inhalt der Generatorartefakte für die Generatorvariante, die auf Basis des Schichtdefinitionsmodells aus Abbildung 7.23 erzeugt wurde.

1. Bestimmung der Ersetzungsoperationen. Für jede VR muss ermittelt werden, durch welche GRVs sie ersetzt wurde.
2. Verarbeitung der Generatorartefakte und Prüfung der referenzierten Namen. Für jedes Generatorartefakt muss daraufhin ermittelt werden, ob eine VR oder GRV referenziert wird.
3. Falls eine Referenz auf eine VR oder GRV gefunden wird, muss ermittelt werden, wie diese Referenz im Kompositionsschritt auf die neu erzeugten Generatorartefakte abgebildet wurde. Die Referenz muss daraufhin an den neuen Namen angepasst werden.

Als Nächstes werden diese Schritte präzisiert und genauer erklärt, wie die Vorgehensweise zur Umsetzung der Schritte ist. Der Pseudocode dazu ist in Algorithmus 7.5 auf Seite 213 angegeben.

1. Aufbau einer Datenstruktur `vrReplacementMap`, die verwaltet, welche VR durch welche GRV ersetzt wird (Zeile 1). Die Erzeugung dieser Datenstruktur erfolgt auf Basis des Schichtdefinitionsmodells. Für jede Schichtenspezifikation werden alle definierten Kompositionsoperationen untersucht. Liegt eine Ersetzungsoperation `X replace Y`

vor, wird zu der Datenstruktur `vrReplacementMap` ein Eintrag mit dem Schlüssel `Y` und dem Wert `X` hinzugefügt. Die Datenstruktur vermerkt somit, durch welche GRV eine VR unmittelbar ersetzt wird.

2. Anwendung der Ableitungsvorschrift auf jeden Eintrag aus `vrReplacementMap` (Zeile 2 – 5). Dieser Schritt wird ausgeführt, damit im nächsten Schritt auf einfache Weise geprüft werden kann, ob ein Teil eines Artefakts aufgerufen wird, der durch die Komposition nicht mehr über diesen Namen erreichbar ist. Dazu wird die Ableitungsvorschrift, die in Abschnitt 7.6.3 eingeführt wurde, auf jeden Schlüssel und Wert eines Eintrags aus `vrReplacementMap` angewendet. Das resultierende neue Tupel aus Schlüssel und Wert wird in der Datenstruktur `adaptedReplacementMap` abgespeichert (Zeile 5), auf die in den weiteren Schritten des Algorithmus zugegriffen wird. Nach Anwendung der Ableitungsvorschrift auf alle Einträge in `vrReplacementMap` enthält die Datenstruktur `adaptedReplacementMap` Informationen darüber, ob ein Aufruf eines Artefakts durch einen anderen Aufruf ersetzt werden muss.
3. Für jedes Artefakt wird anschließend wie folgt verfahren (Zeile 6 – 20):
 - a) Parsen des Artefakts zur Erzeugung eines ASTs zum Artefakt (Zeile 7).
 - b) Identifikation der AST-Knoten, die auf Namen verweisen können, die sich im Zuge der Komposition verändert haben können (Zeile 9).
 - c) Für jeden betroffenen AST-Knoten: Prüfung, ob der referenzierte Name durch die Komposition verändert wurde (Zeile 11). Diese Bedingung ist dann erfüllt, wenn der AST-Knoten auf einen Namen verweist, der als Schlüssel oder Wert in der Datenstruktur `adaptedReplacementMap` enthalten ist. Falls die Bedingung erfüllt ist, muss der referenzierte Name auf den neuen Namen angepasst werden (Zeile 12 – 19).

Aktualisierung des referenzierten Namens

Wie zuvor dargelegt, kann es erforderlich sein den Namen anzupassen, der durch einen AST-Knoten referenziert wird. Die Vorgehensweise hierzu wird im Folgenden ausführlicher erläutert, wobei davon ausgegangen wird, dass im AST-Knoten der Name `refName` referenziert wird (Pseudocode 7.5, Zeile 10).

1. Prüfung, ob über `refName` eine VR referenziert wird, die durch eine GRV ersetzt wurde. Sollte dies der Fall sein, wird die entsprechende GRV bestimmt (Zeile 12). Dazu wird geprüft, ob `adaptedReplacementMap` den Schlüssel `refName` enthält. Ist dies nicht der Fall, wird `refName` durch keine GRV ersetzt und es wird mit dem nächsten Schritt fortgefahren. Andernfalls wird die GRV wie folgt bestimmt: Enthält `adaptedReplacementMap` zum Schlüssel `refName` den Wert `Y`, wird der Wert zu Schlüssel `Y` bestimmt. Dies wird solange wiederholt, bis zu einem Schlüssel kein Wert mehr vermerkt ist. Der Name des letzten Werts, der auf diese Weise gefunden wurde, repräsentiert den

Algorithmus 7.5 : Function `replaceReferencedNamesByNewNames()`

```

input    : ldm: Layer definition model,
           artifacts: Artifacts that might need to be adapted
modifies : artifacts: Referenced names, which are influenced by composition,
           are updated in the artifacts

1 vrReplacementMap  $\leftarrow$  createAndFillReplacementMap (ldm);
2 for each entry : vrReplacementMap do
3   newKeyName  $\leftarrow$  returnNameInNewNotation (Key of entry);
4   newValueName  $\leftarrow$  returnNameInNewNotation (Value of entry);
5   Add entry <newKeyName, newValueName> to adaptedReplacementMap;
6 for each artifact : artifacts do
7   ast  $\leftarrow$  createAST (artifact);
8   for each astNode : getASTNodes (ast) do
9     if astNode potentially affected by name changes then
10      refName  $\leftarrow$  getNameReferencedByASTNode (astNode);
11      if adaptedReplacementMap contains key or value refName then
12        /* Update referenced name */
13        grvReplacingRefName  $\leftarrow$  getGRVReplacingVR (refName);
14        vrReplacedByRefName  $\leftarrow$  getVRReplacedByGRV (refName);
15        pathToArtifact  $\leftarrow$  getPathToArtifactOfVR (
16          vrReplacedByRefName );
17        pathToVRInArtifact  $\leftarrow$  getPathToVRInArtifact (
18          grvReplacingRefName );
19        newName  $\leftarrow$  Combine pathToArtifact and pathToVRInArtifact;
20        Update refName to newName in astNode;
21  Update artifact in case AST nodes were updated;

```

Namen der GRV, durch welche die VR `refName` ersetzt wurde. Diese GRV ist wichtig, da die referenzierte VR `refName` über den Namen dieser GRV im neuen Artefakt abgelegt ist.

2. Prüfung, ob über `refName` eine GRV referenziert wird. Sollte dies der Fall sein, wird die VR ermittelt, die im Ausgangsartefakt durch die GRV `refName` ersetzt wird (Zeile 13). Dazu wird geprüft, ob `adaptedReplacementMap` einen Wert enthält, der `refName` entspricht. Ist das nicht der Fall, handelt es sich bei `refName` um keine GRV. Andernfalls wird die Vorgehensweise wiederholt für den Schlüssel zu dem Wert, bis kein Eintrag mehr in `adaptedReplacementMap` gefunden wird, so dass ein Schlüssel zu dem entsprechenden Wert existiert. Der Name des letzten Schlüssels, der auf diese Weise gefunden wurde, ist der Name der VR, die durch die GRV `refName` ersetzt wird. Diese VR ist wichtig, da diese VR Auskunft darüber gibt, in welchem Artefakt die ersetzende GRV im Kompositionsschritt abgelegt wurde. Wird eine VR ersetzt, findet diese Ersetzung innerhalb des Ausgangsartefakts statt. Folglich muss bekannt sein, in welchem Ausgangsartefakt die ursprüngliche VR enthalten war.
3. Konstruktion des neuen Namens (Zeile 18). Aus den vorherigen beiden Schritten wird schließlich der neue Name ermittelt. Der neue Name ergibt sich aus dem Pfad zum Artefakt welches die Ausgangs-VR enthält (Zeile 14 – 15) und dem Namen der GRV, welche die referenzierte VR ersetzt (Zeile 16 – 17).
4. Ersetzung des alten Namens durch den neuen Namen (Zeile 19).

Anwendung auf das Beispiel

Diese Vorgehensweise wird nachfolgend auf die Templates, deren Inhalt in Abbildung 7.29 auf Seite 211 zu sehen ist, angewendet. Die grafische Repräsentation des Schichtdefinitionsmodells ist in Abbildung 7.23 auf Seite 199 gezeigt. Für die dort gezeigten Ersetzungsoperationen sieht die Datenstruktur `vrReplacementMap` wie in Tabelle 7.30 gezeigt aus. Auf diese Einträge wird anschließend die Ableitungsvorschrift angewendet. Das Ergebnis dieser Anwendung ist in Tabelle 7.31 zu sehen.

Schlüssel	Wert
A.B:D	H.I:L
H.I:L	S.T:U

Abbildung 7.30: Inhalt der Datenstruktur `adaptedReplacementMap` vor Anwendung der Ableitungsvorschrift.

Schlüssel	Wert
/A/B::D	/H/I::L
/H/I::L	/S/T::U

Abbildung 7.31: Inhalt der Datenstruktur `adaptedReplacementMap` nach Anwendung der Ableitungsvorschrift.

Anschließend wird die Adaption der einzelnen Artefakte vorgenommen. Dabei muss beachtet werden, dass in den Beispielen in dieser Arbeit die AST-Knoten für `EXPAND` Anweisungen von der Adaption betroffen sein können.

Für die Templates Q und T ist in diesem Beispiel keine Adaption erforderlich, da beide Templates keine EXPAND Anweisungen enthalten. Die Templates B und I enthalten hingegen zwar EXPAND Anweisungen, die aufgerufenen DEFINE Blöcke sind jedoch nicht von Ersetzungsoperationen betroffen. Dies ist daran ersichtlich, dass die Namen der aufgerufenen DEFINE Blöcke nicht in der Datenstruktur `vrReplacementMap` enthalten sind.

Anders verhält es sich für das Template F. Dieses verweist in dem DEFINE Block G auf den DEFINE Block `/A/B : D`. Da die Datenstruktur `adaptedReplacementMap` einen Schlüssel mit diesem Namen enthält (siehe Tabelle 7.31), ist der aufgerufene DEFINE Block von einer Ersetzung betroffen. Aus diesem Grund wird als Erstes ermittelt, durch welche GRV die referenzierte VR ersetzt wurde. In `adaptedReplacementMap` ist zum Schlüssel `/A/B : D` der Wert `/H/I : L` und zu diesem Schlüssel der Wert `/S/T : U` hinterlegt. Die VR `/A/B : D` wird also durch die GRV `/S/T : U` ersetzt. Der referenzierte DEFINE Block D wurde somit im Kompositionsergebnis durch den DEFINE Block U ersetzt. Im Anschluss daran wird geprüft, ob `/A/B : D` eine VR ersetzt hat. Diese Bedingung ist allerdings nicht erfüllt, da `adaptedReplacementMap` keinen Eintrag mit diesem Wert besitzt. Auf Grundlage dieser Erkenntnisse kann der neue Name bestimmt werden. Als neuer Name wird der Pfad des Ausgangsartefakts beibehalten, da `/A/B : D` keine andere VR ersetzt, und lediglich der Name der GRV ersetzt. Der Pfad des Ausgangsartefakts ist `/A/B`, der Name der GRV ist U. Daraus ergibt sich der neue Name `/A/B : U`, so dass der alte Name `/A/B : D` im EXPAND Knoten durch diesen ersetzt wird.

Analog wird für das Template N verfahren. Gemäß `adaptedReplacementMap` wird der referenzierte Name `/H/I : L` durch den Namen `/S/T : U` ersetzt. Folglich wurde im Kompositionsschritt der DEFINE Block L durch den DEFINE Block U ersetzt. Ferner ist ersichtlich, dass es sich bei dem referenzierten Namen `/H/I : L` um eine GRV handelt, da gemäß `adaptedReplacementMap` die VR `/A/B : D` durch `/H/I : L` ersetzt wird. Aus diesen beiden Informationen wird wiederum der neue Name abgeleitet. Der Pfad zum Ausgangsartefakt (`/A/B`) wird dazu mit dem Namen der GRV (U) kombiniert, so dass der neue Name ebenfalls `/A/B : U` lautet. In beiden Artefakten wird der Name innerhalb der EXPAND Anweisung durch den gleichen Namen ersetzt, da die referenzierten Namen auf Artefakte bezogen waren, die durch die gleiche GRV ersetzt wurden.

Abbildung 7.32 zeigt den Inhalt der Artefakte nach dem Adaptionsschritt. Hervorzuheben hierbei sind die beiden angepassten EXPAND Anweisungen in den Templates F und N.

7.7 Diskussion

7.7.1 Allgemeine Diskussion

Eine essentielle Herausforderung bei der Einführung von Variabilität in Generatoren betrifft die Granularität von Generatorartefakten wie Templates. Kleine Templates sowie Templates mit einer Vielzahl von GRs ermöglichen es, kleine Teile eines Generators gezielt zu adaptieren. Eine Folge davon ist, dass es möglich ist erforderliche Änderungen durch geringfügige Anpassungen durchführen zu können, da feingranulare Anpassungen vollzogen werden können. Der Nachteil daran ist, dass die Erzeugung einer Vielzahl von Templates bzw. von Templates mit vie-

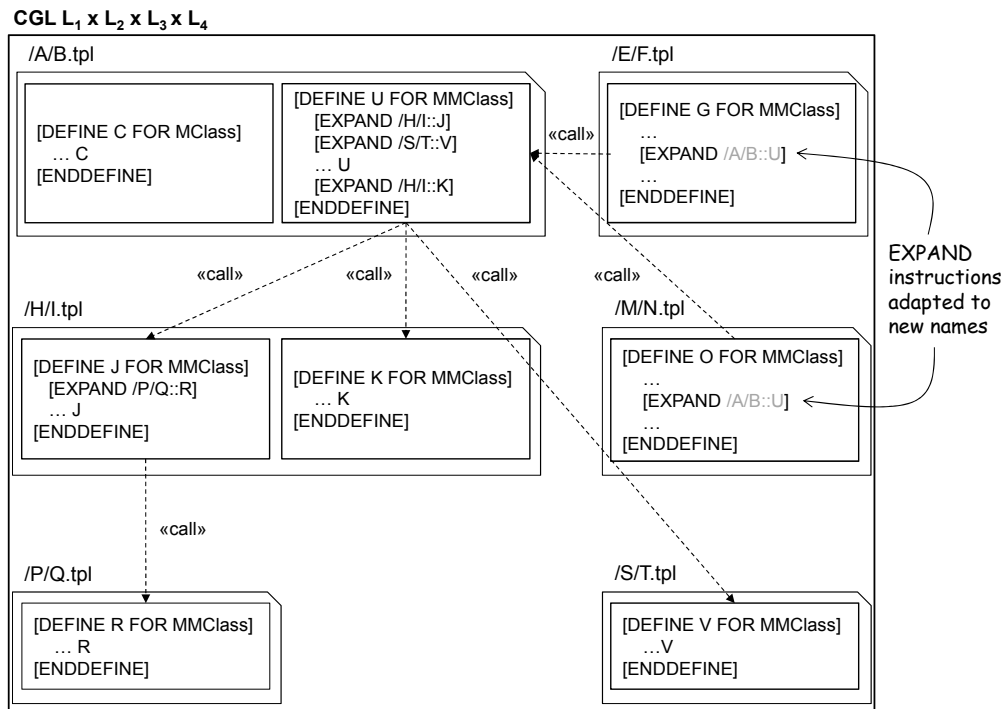


Abbildung 7.32: Inhalt der Generatorartefakte nach der Adaption der Artefakte aus Abbildung 7.29.

len kleinen GRs die Lesbarkeit und Verständlichkeit eines Generators beeinträchtigen können. Demgegenüber erlauben es grobgranulare Template besser einen schnellen Überblick über den Generator zu bekommen und die zusammenhängenden Teile zu verstehen. Andererseits führen grobgranulare Templates dazu, dass große Teile von Templates adaptiert werden müssen und dabei gegebenenfalls größere Teile bestehender Templates wiederholt werden müssen, um gezielt kleinere Änderungen in Varianten einzuführen. So ist beispielsweise in dem in Abschnitt 7.2 eingeführten Beispiel die Generierung des Methodenkörpers einer `set`-Methode ausgelagert. Dadurch ist eine Anpassung des Methodenkörpers der `set`-Methoden auf direkte Weise möglich. Wäre die gesamte Funktionalität für die Methodengenerierung beispielsweise in einem einzelnen `DEFINE` Block gekapselt, müsste der gesamte Block wiederholt werden, um eine Anpassung durchführen zu können. Bei der Modellierung der Generatorvariabilität müssen die Entwickler sich über diesen Sachverhalt bewusst sein und es muss situationsabhängig entschieden werden, welche Granularität gewählt wird. Der Grund dafür ist, dass es keine allgemeingültige Regel dafür gibt, wie Templates zu strukturieren sind. Stattdessen sollten projektspezifisch Guidelines eingeführt werden, an denen sich die Entwickler orientieren können.

Der vorgestellte Ansatz sieht zudem vor, dass für jede VR verschiedene Kompositionsoperationen durchgeführt werden können: `replace`, `extends` als Komfortoperation, `before` sowie `after`. Dessen ungeachtet ist die Anwendung dieser verschiedenen Kompositionsoperationen nicht für sämtliche Artefakte sinnvoll. Dies betrifft insbesondere die beiden Kompositi-

operations `before` und `after`. Handelt es sich bei einer VR um eine Methode in einer objektorientierten Programmiersprache, so bedeuten die Kompositionsoperationen `before` und `after`, dass vor oder nach der Methode eine andere Methode ausgeführt werden soll. Die Ausführung einer Methode vor der eigentlichen Hilfsmethode ist jedoch nur dann sinnvoll, wenn die Methode die Attribute der Hilfsklasse ändert, so dass die Ausführung der Methode einen Einfluss auf die eigentliche Hilfsmethode hat. Ansonsten bleibt die Kompositionsoperation ohne Auswirkungen. Darüber hinaus hat die Ausführung einer Methode nach einer Hilfsmethode keinen Einfluss auf die eigentliche Hilfsmethode. Eine derartige Vorgehensweise macht normalerweise keinen Sinn. Aufgrund dieser Probleme wird in dieser Arbeit vorgeschlagen `before` und `after` Kompositionsoperationen für Hilfsklassen nicht einzusetzen.

7.7.2 Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen

Der in diesem Kapitel vorgestellte Ansatz ist nicht auf ein dediziertes Generatorframework zur Entwicklung Template-basierter Generatoren ausgelegt. Bei der Integration in den Softwareentwicklungsprozess eines produktiv eingesetzten Softwaresystems wird die Wahl des Generatorframeworks folglich nicht eingeschränkt.

Wie in Abschnitt 7.8 noch dargelegt wird, existieren Ansätze zur Implementierung von Generatorvarianten, in denen nur komplette Templates ersetzt werden können. Der Ansatz dieser Arbeit bietet hingegen die Möglichkeit, auch Teile von Templates zu ersetzen. Weiterhin können durch den Ansatz Inhalte vor oder hinter bestehende Inhalte eingefügt werden. Durch diese Operationen lassen sich auf einfache Weise neue Generatorvarianten definieren. Insbesondere die Verwendung der `extends` Operation kann dabei den Modellierungsaufwand reduzieren, wie in der Fallstudie in Abschnitt 7.9 noch gezeigt wird. Dies ist ein wichtiger Faktor für den praktischen Einsatz des Ansatzes. In Abschnitt 7.9 werden darüber hinaus weitere Möglichkeiten vorgestellt, um den Modellierungsaufwand weiter zu reduzieren.

Eine Herausforderung bei der Anwendung des Ansatzes in produktiv eingesetzten Softwaresystemen ist die Evolution der Generatorvarianten. Wenn ein Generatorartefakt verändert wird, kann das die Kompositionsoperationen beeinflussen, die in dem Schichtdefinitionsmodell definiert sind. Diese Situation liegt beispielsweise vor, wenn ein `DEFINE` Block umbenannt wird, der von einer Kompositionsoperation referenziert wird. Überdies können aber auch komplexere Änderungen in einer CGL dazu führen, dass Kompositionsoperationen bzw. adaptierende Generatorartefakte angepasst werden müssen. Ein Beispiel dafür ist, dass ein `DEFINE` Block aus einem CGL in mehrere Blöcke aufgeteilt wird und in adaptierenden CGLs wurde auf diesen Block verwiesen. Um Entwickler in solchen Situationen gezielt zu unterstützen, sollte der Ansatz um Aspekte zur Auswirkungsanalyse [Boh95, Arn96] erweitert werden.

Darüber hinaus gilt es zu beachten, dass bei der Anwendung des Ansatzes vorausgesetzt wird, dass ein Entwickler die passenden GRs kennt, die zu verändern sind. Besitzt ein Entwickler jedoch keine genauen Kenntnisse eines Generators, so kann es sehr zeitaufwendig sein, die relevanten GRs zu identifizieren. Dies gilt vor allem bei komplexen Generatoren. Eine Möglichkeit, dieses Problem abzuschwächen besteht in der Dokumentation von GRs. Im Rahmen einer solchen Dokumentation sollte dargelegt werden, welchen Zweck eine spezifische GR hat und was

von Entwicklern beachtet werden muss, die diese GR verändern wollen. Auf diese Weise kann aus den Generatorartefakten eine Dokumentation generiert werden, die an die Entwickler ausgehändigt werden kann. Auf Basis einer derartigen Dokumentation können die Entwickler auf einfachere Weise Veränderungen an einem Generator vornehmen.

7.8 Verwandte Arbeiten

Zur Modellierung von Variabilität existieren unter anderem annotative Ansätze, kompositionale Ansätze sowie Ansätze unter Nutzung von Transformationen [SRC⁺12a]. Bei annotativen Ansätzen werden alle Varianten in einem Modell, auch als 150% Modell bezeichnet, spezifiziert [ZHJ04, Gom04, CA05, LSGF08]. Kompositionale Ansätze kombinieren verschiedene Artefaktfragmente, um spezifische Varianten abzuleiten [VG07b, HW07, NK08, AJTK09]. Transformationale Ansätze [HvdH07, HMPO⁺08, TM10] wie Delta-Modellierung [HRRS11, HHK⁺13, HHK⁺15] wenden Transformationen auf ein Kernartefakt an, um so erweiterte bzw. veränderte Artefakte zu erhalten. Im Kontext der Variabilität von Generatoren wurden bisher primär kompositionale und transformative Ansätze eingesetzt.

Im Folgenden werden verschiedene Ansätze zur Adressierung von Variabilität in Generatoren vorgestellt. Dabei werden auch existierende Generatorframeworks vorgestellt, die Mechanismen zur Handhabung von Variabilität mitliefern. Der in dieser Arbeit vorgestellte Ansatz ist konzeptuell unabhängig von einem spezifischen Generatorframework und kann auf viele Generatorframeworks angewendet werden. Ein weiterer Unterschied zu den meisten existierenden Techniken besteht darin, dass in diesem Ansatz beliebige Regionen als variable Regionen markiert und verändert werden können.

Im Framework Genesys [JMS08, Jör13b] werden Generatoren durch die Komposition von wiederverwendbaren Bausteinen, genannt Service Independent Building Blocks (SIBs), erzeugt. Ein SIB kann sowohl atomare Services als auch komplexe Funktionalität kapseln. Beispielsweise existiert ein SIB zum Schreiben von Text in eine Zeichenkette und ein SIB zur Ausführung einer Template Engine. Letztgenannter SIB ist ein elementarer SIB zur Erzeugung von Generatoren. Konkrete Modelle in Genesys sind Graphen, in denen die Knoten Instanzen von SIBs darstellen. Varianten von Generatoren können auf verschiedene Weisen erstellt werden. Die einfachste Möglichkeit besteht in dem Kopieren eines Genesys-Modells und dem Verändern der Kopie. Darüber hinaus können SIBs erstellt werden, welche die wiederverwendbaren Teile eines Generators kapseln. Muss eine Generatorvariante das Verhalten eines SIBs anpassen, so kann in der zugehörigen Instanz des SIBs der Template Code, der mit diesem SIB verbunden ist, neu gesetzt werden. Dies ist vergleichbar zu dem Ersetzen eines kompletten Templates in dieser Arbeit. Demgegenüber können in dieser Arbeit beliebige Bereiche eines Templates modifiziert werden. Darüber hinaus ist nicht nur eine komplette Ersetzung möglich, es können auch dedizierte Inhalte vor oder hinter Bereiche in Templates eingeführt werden.

Acceleo [Acc] ermöglicht es, Generatoren über das Konzept des dynamischen Überschreibens anzupassen. Um ein Template dynamisch zu überschreiben, muss ein Modul, welches eine Sammlung von Templates darstellt, ein anderes Modul eines existierenden Generators erweitern. Die erweiternden Module werden mit höherer Priorität behandelt als die überschriebenen Module. Dadurch wird ein Template, welches ein anderes Template überschreibt, anstelle des über-

schriebenen Templates aufgerufen. Dies entspricht einer Komposition zur Generierungszeit, wie in Abschnitt 7.6 erläutert. Das dynamische Überschreiben von Templates ist vergleichbar zur Ersetzung eines kompletten Templates durch ein anderes Template. Wie zuvor erläutert, können in dieser Arbeit Templates hingegen nicht nur als Ganzes ausgetauscht werden.

Die Templatesprache Xpand [XPa] ermöglicht in der aktuellsten Version die Anpassung von Generatoren über AOP-Mittel [VG07a]. So können Aspekte bereitgestellt werden, die `DEFINE` Blöcke enthalten, die vor, nach oder anstelle von anderen `DEFINE` Blöcken ausgeführt werden. Die Idee zur Einführung der Kompositionsoperationen wurde maßgeblich durch diesen Ansatz in Xpand motiviert, wie im Folgenden erklärt wird. So wurden aus der Möglichkeit, Templatecode vor, nach oder anstelle von Templatecode in einem spezifischen `DEFINE` Block auszuführen, die Kompositionsoperationen `replace`, `extends`, `before` und `after` abgeleitet. Im Gegensatz zu Xpand handelt es sich bei dem Ansatz in dieser Arbeit aber nicht um eine Komposition zur Generierungszeit. Stattdessen erfolgt eine Komposition über die Erzeugung neuer Artefakte. Der Vorteil davon ist, dass das Generatorframework keine eingebaute Unterstützung für die Kompositionsoperationen mitliefern muss. Ferner hat die Idee der Strukturierung von Templates in verschiedene `DEFINE` Blöcke die Idee von GRs motiviert, so dass ausgewählte Teile von Templates adaptiert werden können.

In Xtend [Xtea], dem Nachfolgeframework zu Xpand, bestehen Generatoren aus Erweiterungsmethoden. Zur Anpassung eines Generators können gezielt Erweiterungsmethoden über Dependency Injection ausgetauscht werden. Wie bei Xpand sind diese Konzepte aber an das spezifische Generatorframework und an die Templatesprache gebunden.

Die Arbeit in [VG07b] stellt verschiedene Möglichkeiten zur Umsetzung von Variabilität in generierten Systemen vor. Zum einen wird der Einsatz von AOP auf Template-Ebene angeregt, analog zu dem bereits vorgestellten Xpand Ansatz [VG07a]. Darüber hinaus wird die Möglichkeit diskutiert AOP auf Ebene des generierten Codes einzusetzen. In diesem Fall wird die Variabilität nicht in den Generator integriert, sondern über AOP auf den generierten Code abgebildet. Als weitere Alternative wird ein transformativer Ansatz angeregt. In diesem Fall erzeugt der Generator den zu generierenden Code nicht direkt, sondern baut einen AST auf, der diesen Code repräsentiert. Anschließend können Varianten des generierten Codes über die Durchführung von Transformationen auf diesem AST umgesetzt werden. Der transformierte Code kann schließlich über einen Pretty Printer ausgegeben werden. Ein vergleichbarer Ansatz wird in dieser Arbeit nicht verfolgt, da die Transformationen in diesem Kontext sehr umfangreich und komplex werden können.

7.9 Fallstudie in der Industrie

7.9.1 Ziele und Forschungsfragen

Das Ziel dieser Fallstudie ist es, die Anwendbarkeit des Ansatzes auf einen realen Generator zu demonstrieren. Dabei soll der Ansatz mit der informellen Wiederverwendung [Jör13a] über das Kopieren von Generatoren und der Veränderung dieser Kopien verglichen werden. Zu diesem Zweck wurden die folgenden Forschungsfragen abgeleitet:

- **RQ1:** Ist es praktikabel den Ansatz auf reale Generatoren mit realen Generatorvarianten

anzuwenden?

- **RQ2:** Ist die Anwendung des Ansatzes mit Hinblick auf die Komplexität der resultierenden Artefakte der informellen Wiederverwendung über Kopieren und Verändern der kopierten Generatorvarianten überlegen?

7.9.2 Aufbau und Gegenstand der Studie

Der Ansatz wird in dieser Studie auf den Java-Generator angewendet, der die Entitätsklassen für Authoring generiert. Dieser Generator verarbeitet Klassendiagramme als Eingabe und erzeugt für die enthaltenen Klassen unter anderem Java-Klassen mit Methoden zum Setzen und Erfragen von Attributwerten. Darüber hinaus enthalten die generierten Klassen innere Klassen und Zugriffsmethoden, welche die Daten der Klasse auf spezielle Weise verarbeiten. Dieser Generator wird im Folgenden über den Namen *OVI* referenziert. Neben diesem bestehenden Generator war eine Variante *OV2* dieses Generators von DSA gewünscht, die:

- nicht die zusätzlichen inneren Klassen und speziellen Zugriffsmethoden generiert.
- kein normales Java-Attribut für alle Assoziationen der zugehörigen UML-Klasse generiert. Stattdessen soll ein Java-Attribut mit einem speziellen Typ generiert werden, wenn die Zielklasse der Assoziation mit einem bestimmten Stereotyp gekennzeichnet ist. Ist die Zielklasse hingegen nicht mit diesem Stereotyp gekennzeichnet, soll das Attribut wie üblich generiert werden.
- die resultierenden Klassen so benennt wie die ursprünglichen Klassen, aber zusätzlich noch ein Suffix anhängt, um die neuen Varianten der Klassen von den anderen Varianten einer Klasse direkt unterscheiden zu können.

Zur Beantwortung der ersten Forschungsfrage wurde der Ansatz dieser Arbeit angewendet, um die beiden Generatorvarianten zu erzeugen. Zu diesem Zweck wurden drei CGLs eingeführt:

- *NV1* zur Kapselung der gemeinsamen Anteile von *OVI* und *OV2*.
- *NV2*, der mit *NV1* so komponiert wird, dass der Generator, der aus dieser Komposition entsteht, den Code der Generatorvariante *OVI* generiert.
- *NV3*, der mit *NV1* so komponiert wird, dass der Generator, der aus dieser Komposition entsteht, den Code der Generatorvariante *OV2* generiert und *NV2* nicht nutzt.

Die zweite Forschungsfrage wird beantwortet, indem die Generatorvariante *OV2* durch das Kopieren der Variante *OVI* und die anschließende Anpassung dieser Kopie implementiert wurde. Darauf folgend wurden die Generatorvarianten *OVI* und *OV2* mit den Generatorvarianten *NV1*, *NV2* und *NV3* verglichen. Zur Durchführung des Vergleichs wurden verschiedene Messungen durchgeführt. So wurde für jede Variante die Anzahl an Zeilen von Template-Code (TLOC) und die Anzahl von Zeilen von Hilfsklassen-Code (HLOC) ermittelt. Aufbauend darauf wurden die TLOC und HLOC Zahlen der Generatorvarianten *OVI* und *OV2* sowie der Generatorvarianten *NV1*, *NV2* und *NV3* aggregiert und verglichen.

7.9.3 Ergebnisse und Diskussion

Forschungsfrage RQ1

RQ1: Ist es praktikabel den Ansatz auf reale Generatoren mit realen Generatorvarianten anzuwenden?

Mit Hilfe des in dieser Arbeit vorgestellten Ansatzes konnten beide Varianten des Generators implementiert werden. Konkrete Zahlen bezüglich der Komplexität der erzeugten Artefakte zur Umsetzung der beiden Generatorvarianten werden im nächsten Unterabschnitt vorgestellt. Tabelle 7.33 legt dar, wie viele Kompositionsoperationen von welchem Typ eingesetzt wurden.

	Für Templates	Für Hilfsklassen
Anzahl <code>replace</code> Kompositionsoperationen	0	0
Anzahl <code>extends</code> Kompositionsoperationen	32	8
Anzahl durch <code>extends</code> eingesparter <code>replace</code> Kompositionsoperationen	77	16
Anzahl <code>before</code> Kompositionsoperationen	0	0
Anzahl <code>after</code> Kompositionsoperationen	42	0

Tabelle 7.33: Überblick über die Anzahl der Kompositionsoperationen zur Umsetzung der beiden Generatorvarianten.

Diese Zahlen zeigen, dass ausschließlich `extends` Kompositionsoperationen und keine `replace` Operationen definiert wurden. Die adaptierenden Templates aus NV2 und NV3 wurden dazu so implementiert, dass die Signatur der adaptierenden GRVs konform ist zu der Signatur der adaptierten VRs. Dadurch konnte der Modellierungsaufwand deutlich reduziert werden, da ohne `extends` Kompositionsoperationen 77 `replace` Kompositionsoperationen hätten definiert werden müssen. Für die Hilfsklassen ist die Reduktion etwas geringer, da anstelle von 8 `extends` Kompositionsoperationen 16 `replace` Kompositionsoperationen erforderlich gewesen wären.

Darüber hinaus wurden in der Fallstudie viele `after` Kompositionsoperationen definiert. Durch eine Erweiterung des Ansatzes ist es möglich, diesen Modellierungsaufwand noch deutlich zu reduzieren. Verschiedene Möglichkeiten dazu werden nachfolgend dargelegt:

- Die Kompositionsoperation `after` erhält, angewendet auf ein komplettes Artefakt, eine andere Bedeutung. Angenommen es gilt: `X after Y` und `X` sowie `Y` stellen Artefakte dar. Dann kann die Semantik so geändert werden, dass eine GR `X_i` aus `X` hinter die VR `X_i` aus `Y` eingefügt wird. Folglich muss eine adaptierende GR die gleiche Signatur wie die adaptierte VR besitzen. Dies führt dann zu einer Reduktion des Modellierungsaufwands, wenn in einem Artefakt viele GRs enthalten sind, die mit den VRs aus dem anderen Artefakt komponiert werden können.
- Es wird eine neue Kompositionsoperation eingefügt, welche zu dem in dem ersten Auf-listungspunkt beschriebenen Verhalten führt. Dadurch muss die ursprüngliche Bedeutung

der Kompositionsoperation nicht geändert werden.

- Die *after* Kompositionsoperationen, die ausdrücken, dass GRs aus einem Artefakt *X* hinter VRs eines Artefakts *Y* eingefügt werden sollen, werden durch eine *extends* Kompositionsoperation ersetzt. Dazu muss es möglich sein in einer GRV zu definieren, dass an einer bestimmten Stelle der Inhalt der ersetzten VR eingefügt werden soll. Listing 7.14 zeigt ein Beispiel dafür, wie dies in einem konkreten Template aussehen kann. Die adaptierende GRV muss in diesem Fall die gleiche Signatur wie die adaptierte VR besitzen, da über *extends* andernfalls keine Komposition durchgeführt wird.

```

1 [DEFINE GRVBlock FOR MMClass]
2   [SUPER]
3   [REM] Usual implementation of DEFINE block [ENDREM]
4 [ENDDEFINE]
```

XPand

Listing 7.14: Erweiterung der Templates um eine Anweisung zur Inkludierung des Inhalts der ersetzten VR.

Die vorherigen Ausführungen gelten analog für die *before* Kompositionsoperation, die in der Fallstudie nicht angewendet wurde.

Die Ergebnisse zu dieser Forschungsfrage legen nahe, dass der Ansatz für einfache Generatorvarianten praktikabel eingesetzt werden kann. Insbesondere die Verwendung der *extends* Operation kann den Modellierungsaufwand deutlich reduzieren, was den praktischen Einsatz des Ansatzes sehr vereinfacht. Ferner hat sich gezeigt, dass die zur Verfügung stehenden Kompositionsoperationen ausgereicht haben, um die gewünschten Generatorvarianten zu entwickeln. Zusätzlich dazu sollte allerdings einer der vorgestellten Ansätze zur Reduktion des Modellierungsaufwands bezüglich *after* und *before* Operationen umgesetzt werden. Andernfalls kann der Modellierungsaufwand bei komplexeren Generatorvarianten relativ hoch werden, da die Kompositionsoperationen einzeln modelliert werden müssen. Weiterhin sollte das Tooling für einen produktiven Einsatz noch erweitert werden, in dem beispielsweise Unterstützung zur Evolution der Generatorvarianten implementiert wird, wie in Abschnitt 7.7 diskutiert. Dies ist insbesondere hilfreich, falls zahlreiche Generatorvarianten entwickelt werden und bestimmte CGLs mit vielen CGLs komponiert werden.

Forschungsfrage RQ2

RQ2: Ist die Anwendung des Ansatzes mit Hinblick auf die Komplexität der resultierenden Artefakte der informellen Wiederverwendung über Kopieren und Verändern der kopierten Generatorvarianten überlegen?

Tabelle 7.35 gibt einen Überblick über die gemessenen Werte zu der ursprünglichen Generatorvariante *OVI* und den Werten zu der Variante *OV2*, die durch das Kopieren und Ändern der Variante *OVI* entstanden ist. Überdies zeigt die Tabelle die gemessenen Werte zu den Generatorvarianten *NV1*, *NV2* sowie *NV3*, die durch die Anwendung des Ansatzes dieser Arbeit entstanden

sind. Für die Anzahl der Zeilen in den Hilfsklassen (HLOC) unterscheiden sich diese Angaben von den Zahlen in [GMR⁺16]. Dies ist in einer Umstrukturierung der Hilfsklassen begründet. Qualitativ änderte sich dadurch aber nichts an der Einsparung durch den Ansatz oder an den Schlussfolgerungen.

	<i>OV1</i>	<i>OV2</i>	Σ_O	<i>NV1</i>	<i>NV2</i>	<i>NV3</i>	Σ_N
TLOC	5563	2260	7823	1724	4147	347	6218
Anzahl <code>DEFINE</code> Blöcke	327	146	473	175	285	37	497
HLOC	9386	9386	18772	9430	56	56	9542
Anzahl Hilfsmethoden	571	571	1142	579	8	8	595

Tabelle 7.35: Ergebnisse der Fallstudie: TLOC und HLOC für die verschiedenen Varianten.

Wie in Tabelle 7.35 zu sehen, kann die Anzahl der Templatezeilen (TLOC) auf 79 Prozent der Anzahl der Templatezeilen für den Kopier-basierten Ansatz reduziert werden. Der Reduktionsgrad fällt nicht größer aus, da die ursprüngliche Generatorvariante *OV1* zahlreiche weitere Templates beinhaltet. Diese mussten nicht von beiden Generatorvarianten wiederverwendet werden. Hätte die Generatorvariante diese Anteile nicht enthalten und der Fokus hätte auf der reinen Generierung von Entitätsklassen gelegen, wäre der Reduktionsgrad deutlich höher gewesen. Ferner wurden in der Fallstudie lediglich zwei Generatorvarianten umgesetzt. Müssen weitere Varianten entwickelt werden, besteht ein noch höheres Potential zur Wiederverwendung. Bezüglich der Anzahl der `DEFINE` Blöcke lässt sich keine Reduktion feststellen, in dem neuen Ansatz ist die Anzahl der `DEFINE` Blöcke sogar geringfügig höher. Zwar können einerseits `DEFINE` Blöcke durch beide Varianten wiederverwendet werden, allerdings werden zur Definition der Kompositionsoptionen jeweils neue `DEFINE` Blöcke eingeführt, so dass die Gesamtzahl an `DEFINE` Blöcken wiederum steigt. Desto geringer die Unterschiede zwischen den Generatorvarianten sind, desto deutlicher kommt jedoch die Wiederverwendung der Generatoranteile zum Tragen, so dass in anderen Fällen eine Reduktion der Anzahl der `DEFINE` Blöcke wahrscheinlich ist.

Darüber hinaus zeigt Tabelle 7.35, dass die Anzahl der Zeilen in den Hilfsklassen (HLOC) auf 51 Prozent reduziert wurde. Dies ist darin begründet, dass nur wenige Hilfsmethoden punktuell adaptiert werden mussten und die meisten Hilfsklassen unverändert übernommen werden konnten. Dies ist neben der Anzahl der Zeilen in den Hilfsklassen ebenfalls an der Anzahl der Hilfsmethoden erkennbar, die auf 52 Prozent reduziert werden konnte.

Diese Ergebnisse zeigen, dass durch den Ansatz aus dieser Arbeit der Umfang des zu implementierenden Codes für die Templates und die Hilfsklassen deutlich reduziert werden kann. Desto geringer die Unterschiede in den zu entwickelnden Generatorvarianten sind, desto deutlicher wird der Unterschied zur Erzeugung von Generatorvarianten über die informelle Wiederverwendung ausfallen. In diesen Fällen bietet sich die Verwendung des Ansatzes besonders an, da dann lediglich vereinzelte Kompositionsoptionen definiert werden müssen. Desto größer die Unterschiede zwischen den Generatorvarianten werden, desto höher wird hingegen der Modellierungsaufwand für die Generatorvarianten. In solchen Fällen bietet es sich unter Umständen an lediglich einen Teil der Varianten wiederzuverwenden und die spezifischen Teile unabhängig voneinander zu implementieren, anstatt sehr umfangreiche Kompositionsoptionen zu definieren.

7.9.4 Gültigkeit der Fallstudie

Analog zu den vorherigen Fallstudien besteht die Hauptgefahr für die Gültigkeit dieser Fallstudie darin, dass der Ansatz lediglich auf ein reales Szenario in einem einzelnen Softwaresystem angewendet wurde. Die genannten Zahlen zu den Einsparungen durch den neuen Ansatz können dabei nicht unmittelbar auf andere Softwaresysteme übertragen werden, da diese Zahlen sehr stark von dem betrachteten Anwendungsfall abhängen. Dessen ungeachtet deckt das untersuchte Szenario verschiedene Aspekte ab, die nahe legen, dass der Ansatz auch in anderen Softwaresystemen eingesetzt werden kann. Zum einen sollte eine Generatorvariante zu einem bestehenden Generator entwickelt werden, dessen Generatorframework keine dedizierten Variabilitätsmechanismen bietet. Dies ist kein Einzelfall, sondern gilt für zahlreiche Generatorframeworks. Der bestehende Generator wurde zudem unabhängig von dem in dieser Arbeit entwickelten Ansatz implementiert. Somit wurde der Generator nicht bewusst so implementiert, dass er besonders gut für eine Anwendung des Ansatzes geeignet ist. Weiterhin sollten in der neuen Generatorvariante sowohl neue Aspekte hinzugefügt als auch bestehende Aspekte des Generators angepasst werden. Beide Sachverhalte werden voraussichtlich auch für andere Generatorvarianten relevant sein und konnten über die bestehenden Kompositionsoperationen adäquat abgebildet werden.

Kapitel 8

Modellbasierte Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen

Die Gesamtfunktionalität eines Plugin-basierten Softwaresystems kann auf eine Vielzahl von Plugins verteilt sein. Darüber hinaus wird in dieser Arbeit vorausgesetzt, dass ein Plugin-basiertes Softwaresystem Features enthalten kann, welche logisch zusammenhängende Plugins kapseln. Die Prüfung der Architekturkonformität in solchen Systemen umfasst insbesondere die Prüfung der Abhängigkeiten zwischen den Plugins und Features des Softwaresystems. Zur Beschreibung von Restriktionen, welche diese Abhängigkeiten betreffen, im Folgenden als Abhängigkeitsrestriktionen (siehe Definition 23) bezeichnet, wird in dieser Arbeit die DSL Dependency Constraint Language (DepCoL) eingeführt. Die Grundlage für dieses Kapitel bildet die Publikation [GMR15].

Definition 23 (Abhängigkeitsrestriktion)

Eine Abhängigkeitsrestriktion drückt aus, dass eine Abhängigkeit zwischen spezifischen Plugins, Features, Gruppen von Plugins oder Gruppen von Features verboten oder erlaubt ist oder lediglich toleriert wird, aber nicht gewünscht ist.

Die konkreten Abhängigkeitsrestriktionen zu einem Plugin-basierten Softwaresystem werden gebündelt in einem Abhängigkeitsmodell (siehe Definition 24) zusammengefasst.

Definition 24 (Abhängigkeitsmodell)

Ein Abhängigkeitsmodell fasst sämtliche Abhängigkeitsrestriktionen zusammen, die ein Plugin-basiertes Softwaresystem betreffen. Darüber hinaus kann ein Abhängigkeitsmodell die Definition von Gruppen von Features und Plugins umfassen.

Ein Abhängigkeitsmodell wird automatisch von einer Prüfkomponente verarbeitet. Diese ist als Eclipse-Plugin realisiert und kann in die Entwicklungsumgebung integriert werden, in der das zu prüfende Plugin-basierte Softwaresystem entwickelt wird. Auf diese Weise kann die Prüfkomponente innerhalb der Entwicklungsumgebung Fehler oder Warnungen melden, wenn im Rahmen des Prüfvorgangs festgestellt wird, dass die Implementierung des Softwaresystems gegen Abhängigkeitsrestriktionen des Abhängigkeitsmodells verstößt. Dessen ungeachtet können die zugrundeliegenden Konzepte analog auf Plugin-basierte Softwaresysteme angewendet werden, die mit einer anderen Werkzeuginfrastruktur entwickelt werden. Die tatsächliche Konformitätsprüfung findet automatisch begleitend zur normalen Weiterentwicklung des Softwaresys-

tems statt. Die Entwickler können dadurch umgehend innerhalb ihrer Entwicklungsumgebung auf Verstöße gegen Abhängigkeitsrestriktionen hingewiesen werden. Auf diese Weise haben die Entwickler die Möglichkeit, die Änderungen, die zu Verstößen geführt haben, wieder rückgängig zu machen bzw. ordnungsgemäß anzupassen.

In Abschnitt 8.1 werden die Grundlagen der Arbeit eingeführt, in denen einerseits die wichtigsten Begriffe und andererseits die Anforderungen an DepCoL dargelegt werden. Darauf folgt ein Überblick über den Ansatz in Abschnitt 8.2. Danach wird in Abschnitt 8.3 detailliert DepCoL vorgestellt, gefolgt von einer Erklärung zur Semantik eines Abhängigkeitsmodells in Abschnitt 8.4 und einer Beschreibung der Vorgehensweise zur Prüfung der Architekturkonformität in Abschnitt 8.5. Anschließend wird die Methodik zur Anwendung des Ansatzes in Abschnitt 8.6 beschrieben und in Abschnitt 8.7 wird der Ansatz diskutiert. Abgeschlossen wird das Kapitel mit einer Darlegung verwandter Arbeiten in Abschnitt 8.8 und mit der Vorstellung der Ergebnisse einer Fallstudie in Abschnitt 8.9.

8.1 Grundlagen

8.1.1 Begriffsdefinitionen

Der Fokus dieser Arbeit liegt auf Plugin-basierten Softwaresystemen (siehe Definition 25), die aus Features (siehe Definition 26) und Plugins (siehe Definition 4 auf Seite 5) aufgebaut sein können.

Definition 25 (Plugin-basiertes Softwaresystem)

Ein Plugin-basiertes Softwaresystem wird über eine Liste von Features und/oder eine Liste von Plugins festgelegt.

Definition 26 (Feature)

Ein Feature besteht aus beliebig vielen Plugins und fasst diese zu einer Einheit zusammen.

Plugins und Features sind somit feste Bestandteile eines Plugin-basierten Softwaresystems. In einem Abhängigkeitsmodell soll jedoch nicht ausschließlich auf einzelne Plugins und Features verwiesen werden können, sondern auch auf Gruppen von Plugins und Features. Die Motivation hierfür besteht darin, dass auf diese Weise Abhängigkeitsrestriktionen, die für mehrere Plugins und Features gelten sollen, kompakter definiert werden können. Die dazugehörigen Definitionen sind nachfolgend angegeben.

Definition 27 (Plugin-Gruppe)

Eine Plugin-Gruppe umfasst beliebig viele Plugins. Sie ist nicht Bestandteil des Plugin-basierten Softwaresystems selbst, sondern ist eine logische, zur Entwicklung und damit innerhalb eines Abhängigkeitsmodells genutzte Struktur.

Definition 28 (Feature-Gruppe)

Eine Feature-Gruppe umfasst beliebig viele Features. Sie ist, analog zu einer Plugin-Gruppe, nicht Bestandteil des Plugin-basierten Softwaresystems selbst, sondern ist eine logische, zur Entwicklung und damit innerhalb eines Abhängigkeitsmodells genutzte Struktur.

Auf Grundlage dieser Definitionen umfassen sowohl Features als auch Plugin-Gruppen eine Menge von Plugins. Beide Begriffe sind dennoch zu unterscheiden, da ein Feature ein fester Bestandteil eines Plugin-basierten Softwaresystems ist und unabhängig von einem Abhängigkeitsmodell existiert. Demgegenüber wird eine Plugin-Gruppe ausschließlich innerhalb eines Abhängigkeitsmodells definiert, um eine Menge von Plugins zu gruppieren. In dieser Arbeit wird daher bewusst zwischen diesen beiden Begriffen unterschieden.

Abbildung 8.1 zeigt die Zusammenhänge zwischen den definierten Begriffen, wobei die illustrierten Kardinalitäten darstellen, wie die Begriffe in dieser Arbeit verwendet werden. Im Anschluss an die Abbildung werden die Gründe für die gezeigten Zusammenhänge und Kardinalitäten erläutert, falls dies nicht bereits aus den Definitionen hervorgeht.

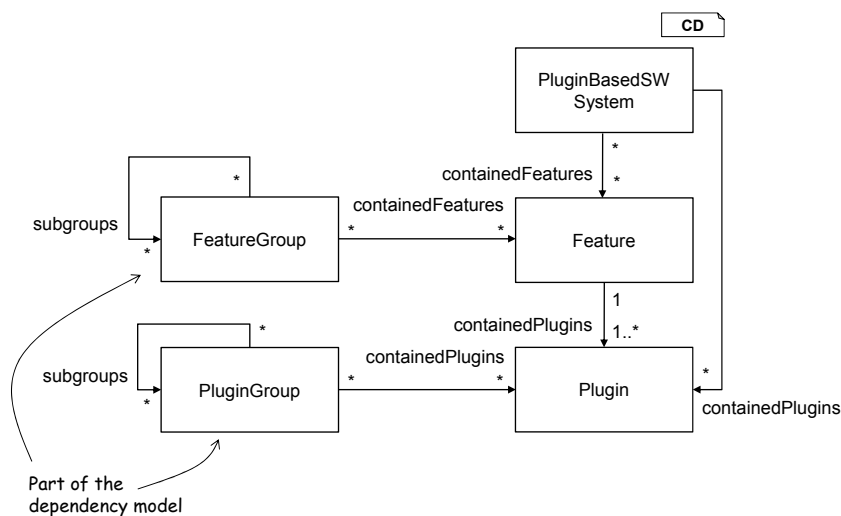


Abbildung 8.1: Überblick über den Zusammenhang zwischen Plugin-basierten Softwaresystemen, Features, Plugins, Feature-Gruppen sowie Plugin-Gruppen.

Zur Vereinfachung der Beschreibung wird in dieser Arbeit angenommen, dass jedes Plugin in genau einem Feature enthalten ist. Theoretisch ist es zwar denkbar, dass das gleiche Plugin in verschiedenen Features enthalten ist, davon wird in dieser Arbeit aber abstrahiert.

Ein Feature dient lediglich der Gruppierung von zusammengehörigen Plugins. Prinzipiell kann ein Feature zwar auch ohne Plugins existieren, in diesem Fall kann aber auf die Verwendung des Features verzichtet werden. Aus diesem Grund wird vorausgesetzt, dass jedes Feature mindestens ein Plugin enthält.

Ein Plugin kann in mehreren Plugin-Gruppen und ein Feature in mehreren Feature-Gruppen enthalten sein. Eine Beschränkung in der Art, dass ein Plugin oder ein Feature in maximal einer Gruppe enthalten sein darf, ist in manchen Situationen zu restriktiv. So können Plugin- und Feature-Gruppen mit Hinblick auf verschiedene Aspekte geschnitten sein, die sich teilweise überlappen können. Beispielsweise kann eine Plugin-Gruppe UI relevante Plugins kapseln und eine andere Plugin-Gruppe legt den Fokus auf UI relevante Plugins aus einem ganz bestimmten Bereich. Beide Plugin-Gruppen sind valide Gruppen und können das gleiche Plugin umfassen.

Analog verhält es sich für Features. Aus den gleichen Gründen kann eine Plugin-Gruppe in mehreren Plugin-Gruppen und eine Feature-Gruppe in mehreren Feature-Gruppen enthalten sein.

Ein Plugin ist in der Regel nicht unabhängig von anderen Plugins, sondern hängt oftmals von anderen Plugins ab, um deren Funktionalität wiederverwenden zu können. Die Folge davon ist, dass das Plugin nur ausgeführt werden kann, wenn diese Plugins ebenfalls verfügbar sind. Der Begriff der Abhängigkeit zwischen zwei Plugins wird in Definition 29 konkretisiert.

Definition 29 (Abhängigkeit zwischen Plugins)

Eine Abhängigkeit zwischen zwei Plugins besteht genau dann, wenn das eine Plugin als erforderliches Plugin in der Manifest-Datei des anderen Plugins aufgeführt ist. Die Folge davon ist, dass das Plugin auf den Code des anderen Plugins zugreifen und diesen verwenden kann.

Aus den Abhängigkeiten zwischen Plugins lassen sich weitere indirekte Abhängigkeiten ableiten, beispielsweise von einem Plugin zu einem Feature oder von einem Feature zu einem anderen Feature. Eine wichtige Grundlage hierfür ist, dass sich sowohl Plugins, Features, Plugin-Gruppen als auch Feature-Gruppen jeweils als Menge von Plugins darstellen lassen. Für ein Feature und eine Plugin-Gruppe ist das auf Basis der zuvor eingeführten Definitionen offensichtlich. Darüber hinaus kann ein Plugin als Menge von Plugins bestehend aus dem Plugin selbst dargestellt werden. Für eine Feature-Gruppe lässt sich die Menge von Plugins konstruieren, indem die Plugins sämtlicher Features der Feature-Gruppe ermittelt werden.

Seien x und y jeweils ein Plugin, ein Feature, eine Plugin-Gruppe oder eine Feature-Gruppe. Dann besteht eine Abhängigkeit zwischen x und y , wenn ein Plugin aus der Menge der Plugins zu x eine Abhängigkeit zu einem Plugin aus der Menge der Plugins zu y besitzt. Dies wird im Folgenden anhand des Beispiels aus Abbildung 8.2 erklärt. Dieses Beispiel zeigt ein Objektdiagramm mit zwei Features, vier Plugins und einer Feature-Gruppe. In diesem Beispiel hängt das Feature a von der Feature-Gruppe d ab, da das Plugin b des Features von dem Plugin f abhängt, welches im Feature e enthalten ist, welches Teil der Feature-Gruppe d ist. Gleichmaßen hängt Feature a von Feature e und von Plugin f ab, sowie Plugin b von Feature e und Feature-Gruppe d . Demgegenüber hängt beispielsweise Plugin c nicht von diesen ab und Feature e hängt nicht von Feature a ab.

8.1.2 Motivation im Projektkontext und Anforderungen

Insgesamt bestehen die Plugin-basierten Softwaresysteme bei DSA aus über 100 Features und über 1000 Plugins, wobei verschiedene Features und Plugins in mehreren Plugin-basierten Softwaresystemen zum Einsatz kommen. Zwischen den Features existieren insgesamt über 1000 Abhängigkeiten. Durchschnittlich hängt also jedes Feature von zehn anderen Features ab. Darüber hinaus liegen auf Plugin-Ebene mehr als 10000 Abhängigkeiten vor, so dass analog zum zuvor genannten Fall jedes Plugin durchschnittlich von zehn anderen Plugins abhängt. Diese Größenordnung der Abhängigkeiten macht eine manuelle Konformitätsprüfung äußerst aufwändig.

Diese Komplexität war die Hauptmotivation für die Entwicklung von DepCoL und der zugehörigen Prüfkomponente. Nachfolgend werden die wichtigsten Anforderungen vorgestellt, die für DepCoL und die Prüfkomponente vorab erhoben wurden:

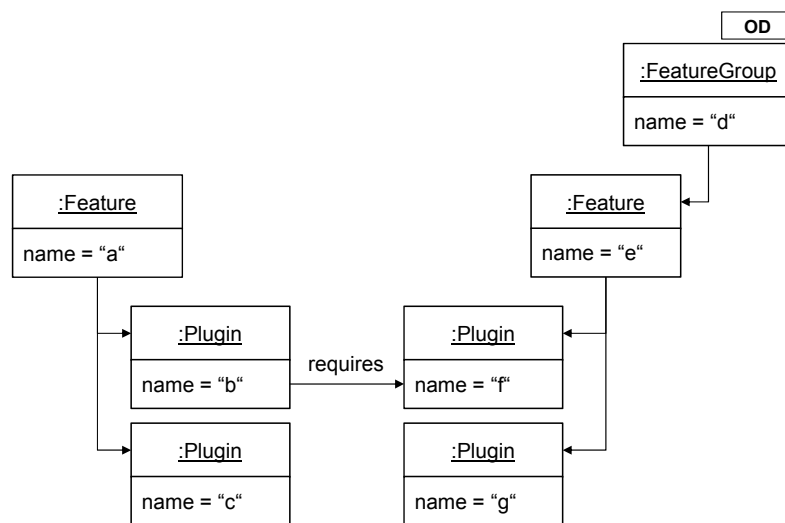


Abbildung 8.2: Objektdiagramm zur Verdeutlichung von Abhängigkeiten zwischen Features, Plugins und Feature-Gruppen.

- Die Spezifikation von Abhängigkeitsrestriktionen muss auf verschiedenen Granularitätsstufen stattfinden können. So muss es nicht nur möglich sein, Abhängigkeitsrestriktionen für einzelne Plugins und Features, sondern auch für Plugin- und Feature-Gruppen anzugeben. Andernfalls ist die Definition von Abhängigkeitsrestriktionen zu aufwändig und nicht praktikabel.
- Die Benachrichtigung über Verstöße gegen die modellierten Abhängigkeitsrestriktionen muss innerhalb der Entwicklungsumgebung erfolgen, in der das Plugin-basierte Softwaresystem implementiert wird. Dabei sollen Entwickler möglichst schnell nach Erkennung eines Verstoßes benachrichtigt werden.
- Es muss möglich sein Abhängigkeitsrestriktionen zu überschreiben. Werden beispielsweise zunächst Abhängigkeitsrestriktionen für eine Plugin-Gruppe definiert, so muss es möglich sein für einen Teil dieser Plugins andere Restriktionen anzugeben. Auf diese Weise können zunächst allgemeine Grobrichtlinien entworfen werden, die anschließend für spezifische Teile angepasst werden.
- Der Ansatz muss dafür geeignet sein ein Abhängigkeitsmodell iterativ zu entwickeln. Auf diese Weise soll sichergestellt werden, dass Abhängigkeitsrestriktionen schrittweise eingeführt werden können, anstatt direkt eine umfangreiche Gesamtarchitektur entwickeln zu müssen. Aufgrund der hohen Feature- und Plugin-Anzahl und der Anzahl der Abhängigkeiten ist diese Herangehensweise die einzige praktikable Lösung.

8.2 Überblick über den Ansatz zur Prüfung der Architekturkonformität

Die Kernidee des Ansatzes zur Prüfung der Architekturkonformität ist, dass mittels DepCoL ein Abhängigkeitsmodell definiert wird und dass eine Prüfkomponente die Konformität der Implementierung eines Softwaresystems zu diesem Abhängigkeitsmodell prüft. Abbildung 8.3 gibt einen Überblick über die wesentlichen Akteure und den Ablauf bei der Definition und Prüfung der Konformität. Dieser Ablauf wird nachfolgend genauer vorgestellt. Die Nummerierung dieser Schritte entspricht der Nummerierung in der Abbildung.

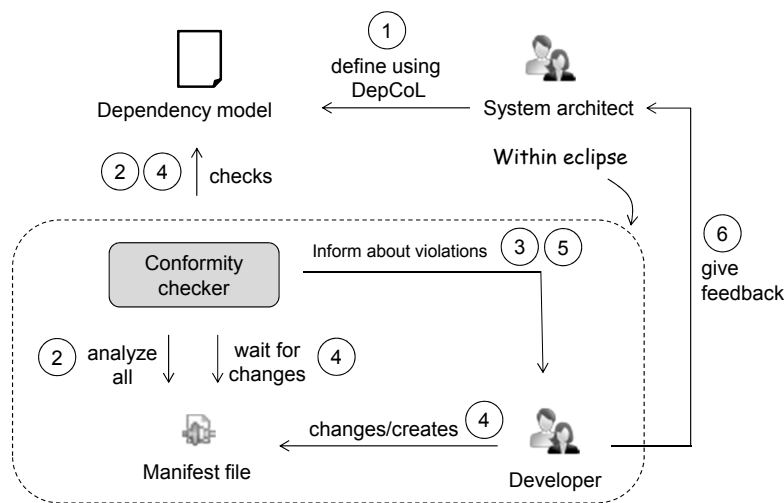
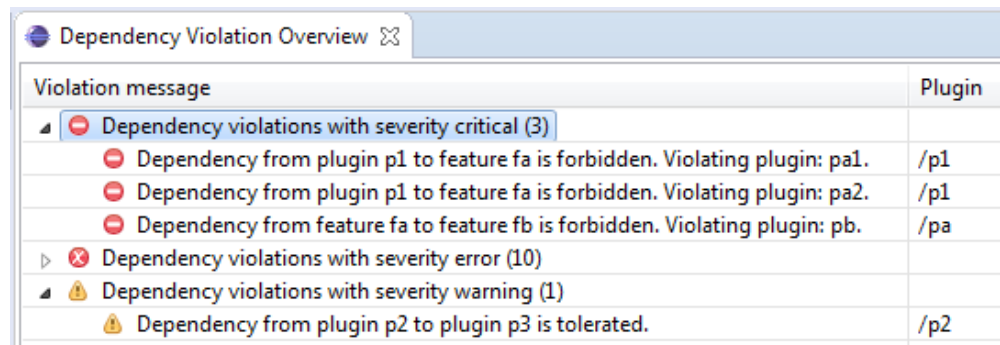


Abbildung 8.3: Überblick über den Prozess zur Prüfung der Architekturkonformität.

1. Als Erstes definieren die Systemarchitekten des Plugin-basierten Softwaresystems das Abhängigkeitsmodell unter Nutzung der DSL DepCoL. Dieses Abhängigkeitsmodell kapselt die Abhängigkeitsrestriktionen, gegen welche die Implementierung des zu untersuchenden Softwaresystems nicht verstoßen darf.
2. Nachdem ein Abhängigkeitsmodell definiert wurde, kann die Prüfkomponente die Implementierung des Softwaresystems automatisch auf Verstöße gegen das Abhängigkeitsmodell untersuchen. Bei dem Start der Entwicklungsumgebung, in welcher das Plugin-basierte Softwaresystem implementiert wird, kann dazu automatisch der Aufruf der Prüfkomponente initiiert werden. Dabei werden sämtliche bestehenden Plugin-Abhängigkeiten des Softwaresystems extrahiert. Da diese Arbeit auf die Analyse von Plugin-basierten Softwaresystemen abzielt, die mittels Eclipse implementiert wurden, erfolgt diese Extraktion mit Hilfe einer Analyse der Eclipse Manifest-Dateien des Softwaresystems. Im Anschluss daran validiert die Prüfkomponente die Abhängigkeiten sämtlicher Plugins gegen das Abhängigkeitsmodell.

3. Identifiziert die Prüfkomponte Verstöße gegen das Abhängigkeitsmodell, werden diese an die Entwickler gemeldet. In Eclipse werden dazu Fehler oder Warnungen in einer separaten Ansicht angezeigt, wie in Abbildung 8.4 veranschaulicht. Darüber hinaus werden Fehlermarkierungen zu der betroffenen Manifest-Datei hinzugefügt. Diese markieren die Zeilen, in denen unerwünschte Abhängigkeiten in der Manifest-Datei vermerkt sind. Die Entwickler können daraufhin Anpassungen vornehmen, um die Verstöße ordnungsgemäß zu beseitigen.
4. Sobald ein Entwickler die Manifest-Datei eines Plugins ändert, wird die Prüfkomponte automatisch darüber benachrichtigt. Dies ist darüber umgesetzt, dass die Prüfkomponte als Listener in der Entwicklungsumgebung angemeldet ist und sich über Änderungen von Dateien informieren lässt. Infolgedessen werden die Abhängigkeiten des betroffenen Plugins neu bestimmt und anschließend erfolgt die Prüfung dieser Abhängigkeiten gegen das Abhängigkeitsmodell. Der Grund hierfür ist, dass durch die Änderung der Manifest-Datei die Abhängigkeiten des betroffenen Plugins verändert worden sein konnten. Daher muss das Plugin erneut gegen das Abhängigkeitsmodell validiert werden.
5. Werden im vorherigen Schritt neue Verstöße gegen das Abhängigkeitsmodell identifiziert, werden die Entwickler umgehend darüber informiert. Dazu werden die Verstöße in der Manifest-Datei, wie zuvor beschrieben, mit einer Fehlermarkierung annotiert und neue Einträge in die Eclipse-Ansicht hinzugefügt. Aufgrund der Tatsache, dass die Entwickler unmittelbar über Verstöße informiert werden, können sie davon abgehalten werden im Anschluss Quellcode zu implementieren, der auf neu hinzugefügten invaliden Abhängigkeiten basiert. Auf diese Weise hilft die Prüfkomponte Entwicklern aktiv dabei, Architektur-Erosion entgegenzuwirken.
6. Die Entwickler können Rückmeldung an die Systemarchitekten geben. So können sie die Systemarchitekten beispielsweise informieren, wenn sie der Meinung sind, dass spezifische Abhängigkeiten erlaubt oder verboten werden sollten. Die Systemarchitekten können daraufhin das Abhängigkeitsmodell anpassen, so dass die Prüfkomponte anschließend gegen das aktualisierte Abhängigkeitsmodell arbeitet.

Abbildung 8.4 zeigt einen Screenshot der Eclipse-Ansicht, in der Fehler und Warnungen der Prüfkomponte angezeigt werden. In dieser Ansicht erscheinen die Verstöße gegen das Abhängigkeitsmodell gruppiert gemäß deren Kritikalität. Wie in Abschnitt 8.3.3 noch genauer beschrieben, wird zwischen kritischen Fehlern, Fehlern und Warnungen unterschieden. Die Motivation für diese Gruppierung ist, dass die Entwickler die Behebung der Verstöße leichter auf Basis der Kritikalität priorisieren können. Für jeden gemeldeten Verstoß in dieser Ansicht ist unter anderem vermerkt, welches Plugin gegen eine Abhängigkeitsrestriktion verstößt, z.B. Plugin `p1` in der ersten kritischen Fehlermeldung in Abbildung 8.4. Falls gemäß des Abhängigkeitsmodells ein Verstoß vorliegt, der in einer Abhängigkeit zu einem Feature oder einer Feature- oder Plugin-Gruppe begründet ist, gibt die Meldung zudem Auskunft darüber, welche konkrete Abhängigkeit zu dem Verstoß geführt hat. In Abbildung 8.4 ist beispielsweise für die ersten beiden gemeldeten Verstöße dargelegt, dass das Plugin `p1` von den Plugins `pa1` und `pa2` des Features `fa` abhängt und dass diese Abhängigkeiten verboten sind.



Violation message	Plugin
Dependency violations with severity critical (3)	
Dependency from plugin p1 to feature fa is forbidden. Violating plugin: pa1.	/p1
Dependency from plugin p1 to feature fa is forbidden. Violating plugin: pa2.	/p1
Dependency from feature fa to feature fb is forbidden. Violating plugin: pb.	/pa
Dependency violations with severity error (10)	
Dependency violations with severity warning (1)	
Dependency from plugin p2 to plugin p3 is tolerated.	/p2

Abbildung 8.4: Eclipse-Ansicht zur Darstellung der Ergebnisse der Architekturkonformitätsprüfung.

Zusätzlich zu dem bisher beschriebenen Einsatzszenario wurde eine Komponente entwickelt, welche es erlaubt die Prüfkomponeute in ein System zur kontinuierlichen Integration (engl. continuous integration, CI) einzubetten. Der wesentliche Unterschied zum vorherigen Szenario ist, dass die Prüfkomponeute in diesem Fall stets alle Abhängigkeiten prüft und dass die Entwickler sowie die Systemarchitekten per E-Mail über die Verstöße informiert werden. Der Vorteil der Ausführung in einem CI-System ist, dass die Prüfung der Architekturkonformität regelmäßig zu festen Zeitpunkten durchgeführt werden kann. Weiterhin können auf diese Weise auch Systemarchitekten oder andere Personen, die nur selten den Quellcode des Plugin-basierten Softwaresystems selbst modifizieren, über den aktuellen Stand der Architekturkonformität informiert werden. Darüber hinaus kann nicht garantiert werden, dass alle Entwickler eine Entwicklungsumgebung nutzen, in der die Prüfkomponeute integriert und aktiv ist. Das ist darin begründet, dass die Entwickler die Prüfkomponeute, wie jedes andere Plugin, deinstallieren können. Falls solche Entwickler Verstöße gegen die Abhängigkeitsrestriktion einführen, fallen diese Verstöße ohne eine separate Ausführung der Prüfkomponeute eventuell erst später auf.

8.3 Domänenspezifische Sprache zur Definition von Abhängigkeitsmodellen

Die DSL DepCoL ermöglicht die Definition eines Abhängigkeitsmodells, welches Abhängigkeitsrestriktionen zu einem Plugin-basierten Softwaresystem umfasst. Im Laufe dieses Abschnitts werden an verschiedenen Stellen unterschiedliche Auszüge aus der Grammatik zu DepCoL vorgestellt und mit Beispielen verdeutlicht. Die komplette Grammatik von DepCoL befindet sich zusammengefasst im Anhang in Listing E.5.

Listing 8.1 zeigt den Ausschnitt der Grammatik, der den Aufbau eines Abhängigkeitsmodells definiert. Wie aus diesem Listing hervorgeht, kann ein Abhängigkeitsmodell unter anderem die Definition einer Feature- (Zeile 2) und Plugin-Basis (Zeile 3) umfassen. Diese beiden Konzepte werden in Abschnitt 8.3.1 genauer erläutert. Darüber hinaus umfasst DepCoL im Wesentlichen zwei verschiedene Typen von Sprachkonstrukten: 1. solche, die es ermöglichen, Feature-

Gruppen oder Plugin-Gruppen zu definieren (Zeile 8), und 2. solche, die eine Definition von Abhängigkeitsrestriktionen ermöglichen (Zeile 9). Die Definition von Feature- und Plugin-Gruppen wird in Abschnitt 8.3.2 und die Definition von Abhängigkeitsrestriktionen wird darauffolgend in Abschnitt 8.3.3 dargelegt.

```
1 DependencyModel :  
2   featureBase = FeatureBase?  
3   pluginBase = PluginBase?  
4   elements += DependencyModelElement*  
5 ;  
6  
7 DependencyModelElement :  
8   GroupDeclarationElement |  
9   DependencySpecificationElement  
10 ;
```

Listing 8.1: Grammatik der DSL DepCoL. Ein Abhängigkeitsmodell kann eine Feature- und Plugin-Basis, Deklarationen von Feature- und Plugin-Gruppen sowie Definitionen von Abhängigkeitsrestriktionen enthalten.

8.3.1 Definition einer Feature und Plugin-Basis

Ein Abhängigkeitsmodell verweist üblicherweise explizit auf mehrere Feature- und Plugin-Namen. Eine mögliche Fehlerquelle in diesem Zusammenhang ist, dass Feature- und Plugin-Namen an verschiedenen Stellen unbeabsichtigt unterschiedlich geschrieben werden. Um diesem Problem entgegenzuwirken, umfasst DepCoL das Konzept der Feature- und Plugin-Basis. Diese kapseln die Namen der explizit referenzierbaren Features und Plugins. Die Spezifikation einer Feature- und Plugin-Basis ist optional, zu sehen in Zeile 8 und 9 in Listing 8.1. Der Grund dafür ist, dass in einem Abhängigkeitsmodell nicht explizit auf einzelne Features oder Plugins verwiesen werden muss, wie im weiteren Verlauf des Kapitels noch gezeigt wird. In einem solchen Fall ist die Definition einer Feature- und Plugin-Basis nicht erforderlich und kann somit entfallen. Listing 8.2 zeigt den Teil der DepCoL Grammatik, der die Definition einer Feature- und Plugin-Basis ausdrückt.

Eine Feature-Basis (Zeile 1–5) enthält die Namen aller Features, die im Abhängigkeitsmodell explizit referenziert werden können (Zeile 3). Dies wird darüber erreicht, dass die Feature-Basis über `availableFeatures` auf das Nichtterminalsymbol `FeatureName` verweist (Zeile 3). Die weiteren Teile der Grammatik, die explizit Feature-Namen referenzieren, nutzen daher Querverweise auf das Nichtterminalsymbol `FeatureName`. Dadurch wird sichergestellt, dass Parse-Fehler gemeldet werden, wenn an diesen Stellen ein referenzierter Feature-Name nicht in der Feature-Basis eingeführt wurde.

Analog zu der Definition der validen Feature-Namen in einer Feature-Basis erfolgt die Definition der gültigen Plugin-Namen in einer Plugin-Basis (Zeile 11 – 15). Für diesen Teil wurde das Nichtterminalsymbol `PluginName` eingeführt, so dass über dieses auf sämtliche in der Plugin-Basis definierten Plugin-Namen verwiesen werden kann.

Der Aufbau von Feature- und Plugin-Namen (Zeile 8, 18) wird über die Nichtterminalsymbole

```

1 FeatureBase:
2   'featurebase {'
3     (availableFeatures += FeatureName ';'')+
4   '}'
5 ;
6
7 FeatureName:
8   name = QualifiedName
9 ;
10
11 PluginBase:
12   'pluginbase {'
13     (availablePlugins += PluginName ';'')+
14   '}'
15 ;
16
17 PluginName:
18   name = QualifiedName
19 ;
20
21 QualifiedName:
22   Name+ ('.' Name)*
23 ;
24
25 Name:
26   (ID | '0..9')
27 ;

```

Listing 8.2: Grammatik der DSL DepCoL zur Definition einer Feature und Plugin-Basis.

QualifiedName (Zeile 21–23) und Name (Zeile 25–27) definiert. In beiden Fällen muss der Name mit einem Buchstaben oder einer Zahl beginnen. Darüber hinaus kann der Name einen Punkt enthalten, muss dann aber von mindestens einem Buchstaben oder einer Zahl gefolgt werden.

Listing 8.3 zeigt einen Ausschnitt aus einer einfachen Feature-Basis Spezifikation. In diesem Beispiel wird festgelegt, dass die Features `f1`, `f2` und `f3` in der Feature-Basis enthalten sind und somit im Abhängigkeitsmodell explizit referenziert werden können. Die Definition einer Plugin-Basis erfolgt analog.

```

1 declare featurebase {
2   f1;
3   f2;
4   f3;
5 }

```

Listing 8.3: Definition einer Feature-Basis welche die Features `f1`, `f2` und `f3` enthält.

Ein komplexes Plugin-basiertes Softwaresystem kann eine Vielzahl von Features und Plugins umfassen. Die manuelle Spezifikation sämtlicher Feature- und Plugin-Namen kann in sol-

chen Systemen sehr aufwändig sein. Aus diesem Grund bietet die bereitgestellte Prüfkompone-
nente zusätzlich die Möglichkeit, die Feature- und Plugin-Basis auf Grundlage einer Analyse
der Manifest-Dateien des zu prüfenden Plugin-basierten Systems automatisch zu generieren. Im
Rahmen dieser Generierung wird für jedes Feature des Plugin-basierten Systems ein Eintrag
zu der Feature-Basis und für jedes Plugin ein Eintrag zu der Plugin-Basis hinzugefügt. Eine
manuelle Spezifikation der Plugin- und Feature-Basis ist daher nicht erforderlich.

8.3.2 Definition von Feature und Plugin-Gruppen

Zur Reduktion des Modellierungsaufwands für Abhängigkeitsrestriktionen für mehrere Features
und Plugins unterstützt DepCoL das Konzept der Feature- und Plugin-Gruppen. Listing 8.4 zeigt
den Teil der DepCoL Grammatik, der die Definition von Feature- und Plugin-Gruppen abbildet.

```

1 GroupDeclarationElement:
2   'declare ' FeatureGroup |
3   'declare ' PluginGroup
4 ;
5
6 FeatureGroup:
7   'featuregroup' name = QualifiedName ' {'
8     (features += [FeatureName | QualifiedName] ';' ) *
9     ('regexp' exp += NameExp ';' ) *
10    ('featuregroup' featureGroup +=
11      [FeatureGroup | QualifiedName] ';' ) *
12    '}'
13 ;
14
15 PluginGroup:
16   'pluginGroup' name = QualifiedName ' {'
17     (plugins += [PluginName | QualifiedName] ';' ) *
18     ('regexp' exp += NameExp ';' ) *
19     ('pluginGroup' pluginGroup +=
20       [PluginGroup | QualifiedName] ';' ) *
21     '}'
22 ;
23
24 NameExp :
25   //Use * as placeholder for arbitrary string
26   (Name | '*' ) (Name | '*' | '.' ) *
27 ;

```

Listing 8.4: Grammatik der DSL DepCoL zur Definition von Feature- und Plugin-Gruppen.

Welche Features in einer Feature-Gruppe enthalten sind, kann auf verschiedene Weisen fest-
gelegt werden. Zunächst einmal können die Namen der enthaltenen Features explizit angegeben
werden (Zeile 8). Zusätzlich dazu kann eine Feature-Gruppe reguläre Ausdrücke für Feature-
Namen beinhalten, ausgedrückt über den Verweis auf das Nichtterminalsymbol NameExp (Zeile 9). Die Bedeutung eines regulären Ausdrucks innerhalb einer Feature-Gruppe ist, dass die

Feature-Gruppe sämtliche Features enthält, deren Namen konform zu dem regulären Ausdruck sind. In den regulären Ausdrücken kann das Zeichen „*“ als Platzhalterzeichen für beliebige Zeichenketten eingesetzt werden. Dieses Platzhalterzeichen kann an beliebigen Positionen beliebig oft verwendet werden. Wie in der Grammatik zu sehen (Zeile 24–27), bietet DepCoL momentan nicht die Möglichkeit, komplexere reguläre Ausdrücke unter Nutzung von z.B. Klassen von Zeichen oder dem Ausschluss spezifischer Zeichen zu definieren. Stattdessen kann ausschließlich „*“ als Platzhalter für beliebige Zeichenketten genutzt werden. Überdies kann eine Feature-Gruppe andere Feature-Gruppen beinhalten (Zeile 10 – 11). Enthält eine Feature-Gruppe f eine Feature-Gruppe g , so umfasst f implizit alle Features, die in g enthalten sind.

Auf der Ebene der Grammatik wird bewusst zwischen explizit referenzierten Feature-Namen (Zeile 8), regulären Ausdrücken für Feature-Namen (Zeile 9) und Feature-Gruppen (Zeile 10 – 11) unterschieden. Der Grund dafür ist, dass auf diese Weise Querverweise in der Grammatik genutzt werden können, wie in Kapitel 3 erläutert. Durch die Nutzung dieser Querverweise wird automatisch geprüft, ob der angegebene Feature-Name auf eine existierende `FeatureName` Instanz oder der angegebene Name einer Feature-Gruppe auf eine existierende `FeatureGroup` Instanz verweist. Diese können nur vorliegen, wenn das Feature in der Feature-Basis aufgeführt ist und die Feature-Gruppe deklariert wurde.

Alternativ dazu könnten alle drei Varianten über einen Verweis auf eine Zeichenkette abgebildet werden. Um die drei Varianten unterscheiden zu können, müsste hierbei geprüft werden, ob die Zeichenkette dem Namen eines Features aus der Feature-Basis oder dem Namen einer Feature-Gruppe entspricht. Andernfalls würde die Zeichenkette als regulärer Ausdruck interpretiert werden. Dies kann jedoch zu Problemen führen, denn auf diese Weise ist nicht direkt ersichtlich, wenn die Modellierer fälschlicherweise einen Feature-Namen oder den Namen einer Feature-Gruppe falsch geschrieben haben. Durch die Unterscheidung der verschiedenen Fälle auf Grammatik-Ebene werden hingegen Parse-Fehler an die Entwickler gemeldet, wenn ein explizit angegebener Feature-Name oder Name einer Feature-Gruppe falsch geschrieben ist.

Zusammengefasst bedeutet das, dass ein Feature f genau dann in einer Feature-Gruppe fg enthalten ist, wenn eine der folgenden Bedingungen zutrifft:

- f ist explizit als Element von Feature-Gruppe fg aufgeführt.
- f wird durch einen regulären Ausdruck beschrieben, der in der Feature-Gruppe fg enthalten ist.
- f ist in einer Feature-Gruppe $subfg$ enthalten, die Teil der Feature-Gruppe fg ist.

Da jeder explizit genannte Feature-Name innerhalb einer Feature-Gruppe als einfacher regulärer Ausdruck angesehen werden kann und Feature-Gruppen wiederum andere reguläre Ausdrücke enthalten können, kann jede Feature-Gruppe ebenso als regulärer Ausdruck beschrieben werden. Dieser reguläre Ausdruck würde viele Alternativen beinhalten und würde eine Teilmenge der existierenden Features beschreiben. Analog dazu kann auch eine Plugin-Gruppe als regulärer Ausdruck beschrieben werden. In DepCoL wird jedoch bewusst nicht von den Modellierern verlangt Feature- und Plugin-Gruppen komplett als reguläre Ausdrücke anzugeben, da die entsprechenden regulären Ausdrücke sehr umfangreich und schwer verständlich werden können.

Analog zu der Definition von Feature-Gruppen gestaltet sich in Listing 8.4 die Definition von Plugin-Gruppen, in denen ebenfalls einzelne Plugin-Namen explizit (Zeile 17) und reguläre Ausdrücke für Plugin-Namen (Zeile 18) angegeben sein können und die analog ebenfalls Plugin-Gruppen enthalten können (Zeile 19 – 20).

Sowohl bei der Definition einer Feature- als auch bei der Definition einer Plugin-Gruppe wird die Definition der Gruppe über das Schlüsselwort `declare` eingeleitet (Zeile 2 – 3). Dieses Schlüsselwort wird explizit angegeben, um die Deklaration einer Feature- oder Plugin-Gruppe zu unterscheiden von der Spezifikation von Abhängigkeitsrestriktionen für Feature- oder Plugin-Gruppen. Dieser Aspekt wird in Abschnitt 8.3.3 genauer erläutert.

Listing 8.5 zeigt verschiedene Varianten für die Definition von Feature-Gruppen. Die Definition der Feature-Gruppe `fg1` (Zeile 1 – 4) verdeutlicht die explizite Auflistung von Feature-Namen in einer Feature-Gruppe. In dem Beispiel enthält die Feature-Gruppe ausschließlich die Features `f1` (Zeile 2) und `f2` (Zeile 3). Demgegenüber veranschaulicht die Definition der Feature-Gruppe `fg2` (Zeile 6 – 9) die Verwendung von regulären Ausdrücken. In dem gezeigten Beispiel enthält `fg2` sämtliche Features deren Namen mit `f.x` (Zeile 6) oder mit `f.y` (Zeile 7) anfangen. Abschließend wird gezeigt, wie definiert werden kann, dass eine Feature-Gruppe in einer anderen Feature-Gruppe enthalten ist (Zeile 14). Ferner werden im letzten Beispiel die zuvor gezeigten Notationen kombiniert. Dies führt dazu, dass die Feature-Gruppe `fg3` (Zeile 11 – 15) das Feature `f3` (Zeile 12), die Features deren Name mit `f.z` beginnt (Zeile 13) sowie die Features aus der Feature-Gruppe `fg1` (Zeile 14) enthält. Die Definition von Plugin-Gruppen erfolgt auf analoge Weise.

```
1 declare featuregroup fg1 {  
2     f1;  
3     f2;  
4 }  
5  
6 declare featuregroup fg2 {  
7     regexp f.x*;  
8     regexp f.y*;  
9 }  
10  
11 declare featuregroup fg3 {  
12     f3;  
13     regexp f.z*;  
14     featuregroup fg1;  
15 }
```

Listing 8.5: Beispiele für Feature-Gruppen Definitionen.

Ein konkreter Anwendungsfall für die Verwendung von Feature- und Plugin-Gruppen besteht in der Umsetzung spezifischer Architekturmuster [BMR⁺96], wie der Modellierung von geschichteten Softwarearchitekturen [BBC⁺00]. Für jede Architekturschicht kann eine Feature- oder Plugin-Gruppe deklariert werden, die genau die Features oder Plugins enthält, die der Architekturschicht zugeordnet sein sollen. Darüber hinaus können für jede derartig deklarierte Feature- oder Plugin-Gruppe Abhängigkeitsrestriktionen definiert werden, die abbilden, zwi-

schen welchen Architekturschichten Abhängigkeiten bestehen dürfen. Die Definition von Abhängigkeitsrestriktionen wird im nächsten Abschnitt genauer vorgestellt.

8.3.3 Definition von Abhängigkeitsrestriktionen

Abhängigkeitsrestriktionen können für alle Features und Plugins aus der Feature- und Plugin-Basis und für alle deklarierten Feature- und Plugin-Gruppen definiert werden. Der Teil der Grammatik, der diesen Sachverhalt abbildet, ist in Listing 8.6 dargestellt.

```

1 DependencySpecificationElement:
2   ConcreteDependencySpecificationElement ' {'
3       dependencySpecification += DependencySpecification+
4   ' }'
5 ;
6
7 ConcreteDependencySpecificationElement:
8   FeatureDependencySpecification |
9   PluginDependencySpecification |
10  FeatureGroupDependencySpecification |
11  PluginGroupDependencySpecification
12 ;
13
14 FeatureDependencySpecification:
15   'feature' name = [FeatureName | QualifiedName]
16 ;
17
18 PluginDependencySpecification:
19   'plugin' name = [PluginName | QualifiedName]
20 ;
21
22 FeatureGroupDependencySpecification:
23   'featuregroup' name = [FeatureGroup | QualifiedName]
24 ;
25
26 PluginGroupDependencySpecification:
27   'plugingroup' name = [PluginGroup | QualifiedName]
28 ;

```

Listing 8.6: Grammatik der DSL DepCoL, die zeigt, dass Abhängigkeitsrestriktionen für Features, Plugins, Feature-Gruppen und Plugin-Gruppen definiert werden können.

Die Definition von Abhängigkeitsrestriktionen wird über das Nichtterminalsymbol `DependencySpecificationElement` ausgedrückt (Zeile 1 – 5). Eingeleitet wird die Definition durch die Festlegung des Elements, zu dem die Abhängigkeitsrestriktionen definiert werden. Hierzu enthält das Nichtterminalsymbol `ConcreteDependencySpecificationElement` eine Alternative für jeden Typ von Element, für den Abhängigkeitsrestriktionen modelliert werden können: Features (Zeile 8, 14 – 16), Plugins (Zeile 9, 18 – 20), Feature-Gruppen

(Zeile 10, 22 – 24) und Plugin-Gruppen (Zeile 11, 26 – 28). Hervorzuheben ist hierbei, dass die einzelnen Nichtterminalsymbole Querverweise auf die Nichtterminalsymbole `FeatureName` (Zeile 15), `PluginName` (Zeile 19), `FeatureGroup` (Zeile 23) sowie `PluginGroup` (Zeile 27) nutzen. Dadurch wird bereits auf Grammatik-Ebene sichergestellt, dass die Modellierer Parse-Fehler erhalten, wenn die referenzierten Namen ungültig sind, also z.B. ein Feature-Name angegeben ist, der nicht in der Feature-Basis deklariert wurde.

Weiterhin drückt die Grammatik aus, dass mindestens eine Abhängigkeitsrestriktion angegeben werden muss, wenn sich ein Modellierer dafür entscheidet zu einem spezifischen Element Abhängigkeitsrestriktionen festzulegen (Zeile 3).

Eine konkrete Abhängigkeitsrestriktion für Features, Plugins, Feature-Gruppen und Plugin-Gruppen wird über das Nichtterminalsymbol `DependencySpecification` gekapselt. Der Teil der Grammatik, der die Definition dieses Nichtterminalsymbols zeigt, ist in Listing 8.7 dargelegt. Für jedes dieser genannten Elemente kann festgelegt werden, ob Abhängigkeiten zu spezifischen Features, Plugins, Feature-Gruppen oder Plugin-Gruppen als erlaubt, verboten oder toleriert angesehen werden (Zeile 2). Diese verschiedenen Typen von Abhängigkeitsrestriktionen werden durch das Nichtterminalsymbol `DependencyType` ausgedrückt (Zeile 10 – 12). Standardmäßig gelten alle Abhängigkeiten als erlaubt. Um eine Abhängigkeit als verboten oder nur toleriert zu markieren, muss explizit eine zugehörige Abhängigkeitsrestriktion im Abhängigkeitsmodell definiert werden. Dessen ungeachtet kann es in manchen Situationen erforderlich sein Abhängigkeiten explizit zu erlauben. Dieser Aspekt wird im weiteren Verlauf dieses Kapitels noch genauer dargelegt.

Für die Deklaration einer Abhängigkeit als toleriert kann es verschiedene Gründe geben. Zum Beispiel kann eine tolerierte Abhängigkeit kennzeichnen, dass die Abhängigkeit zwar einerseits nicht erwünscht ist, andererseits aber zumindest temporär akzeptiert wird. In einem solchen Fall kann die Markierung als toleriert somit als Indiz dafür dienen, dass die Abhängigkeit zukünftig unter Umständen verboten wird. Überdies kann eine Abhängigkeit als toleriert deklariert werden, wenn zu dem Zeitpunkt der Spezifikation der Restriktion noch unklar ist, ob die Abhängigkeit erlaubt oder verboten werden soll. Findet die Prüfkompone eine Abhängigkeit vor, die als toleriert markiert ist, meldet die Prüfkompone eine Warnung. Demgegenüber erhalten die Entwickler Fehlermeldungen, wenn verbotene Abhängigkeiten identifiziert werden.

Bei der Deklaration einer Abhängigkeitsrestriktion ist es darüber hinaus möglich anzugeben, wie schwerwiegend ein Verstoß gegen die Restriktion ist. Ein solcher Schweregrad kann allerdings nur für verbotene Abhängigkeiten definiert werden. Dies wird über eine Kontextbedingung sichergestellt, wie in Anhang F.4 erklärt wird. Das Nichtterminalsymbol `DependencyViolationSeverity` kapselt das Wissen über diesen Schweregrad (Zeile 2, 6 – 8). Ein solcher Schweregrad impliziert, wie wichtig es ist, einen Verstoß gegen die Restriktion zu beheben. Als mögliche Werte für den Schweregrad können „critical“, „error“ und „warning“ angegeben werden (Zeile 7), wobei der Schweregrad „error“ der Standardwert ist, falls der Schweregrad nicht explizit angegeben wird. Wie in Abbildung 8.4 auf Seite 232 dargestellt, erscheinen in der Ergebnisübersicht die Meldungen bezüglich der Verstöße gruppiert nach Schweregrad der Verstöße. Auf diese Weise können die Entwickler die Verstöße auf Basis des Schweregrads abarbeiten und z.B. erst kritische Verstöße beheben.

Aus Sicht der Prüfkompone werden verbotene Abhängigkeiten mit dem Schweregrad „war-

```

1 DependencySpecification:
2     severity = DependencyViolationSeverity? type = DependencyType
3     dependency = ConcreteDependency ';'
4 ;
5
6 DependencyViolationSeverity:
7     '[' severity = ('critical' | 'error' | 'warning' ) ']'
8 ;
9
10 DependencyType:
11     AllowedDependency | ForbiddenDependency | TolerateDependency
12 ;
13
14 AllowedDependency:
15     typeStr = 'allow dependency to'
16 ;
17
18 ForbiddenDependency:
19     typeStr = 'forbid dependency to'
20 ;
21
22 TolerateDependency:
23     typeStr = 'tolerate dependency to'
24 ;
25
26 ConcreteDependency:
27     DependencyToSinglePlugin | DependencyToPluginPattern |
28     DependencytoPluginGroup |
29     DependencyToFeature | DependencyToFeaturePattern |
30     DependencytoFeatureGroup
31 ;
32
33 DependencyToSinglePlugin:
34     'plugin' plugin = [PluginName | QualifiedName]
35 ;
36
37 DependencytoPluginGroup:
38     'pluginGroup' pluginGroup = [PluginGroup | QualifiedName]
39 ;
40
41 DependencyToFeature:
42     'feature' feature = [FeatureName | QualifiedName]
43 ;
44
45 DependencytoFeatureGroup:
46     'featureGroup' featureGroup = [FeatureGroup | QualifiedName]
47 ;

```

Listing 8.7: Grammatik der DSL DepCoL zur Definition von Abhängigkeitsrestriktionen. Es können Abhängigkeitsrestriktionen hin zu Features, Feature-Gruppen, Plugins sowie Plugin-Gruppen definiert werden.

ning“ und tolerierte Abhängigkeiten gleich behandelt, da Verstöße gegen beide Restriktionen zu Warnungen führen. Dessen ungeachtet gibt es für einen Modellierer von Abhängigkeitsrestriktionen einen Unterschied zwischen diesen beiden Typen von Restriktionen. Wird eine Abhängigkeit als verboten definiert und angegeben, dass Verstöße lediglich zu Warnungen führen sollen, so wird dadurch ausgedrückt, dass die Abhängigkeit nicht bestehen darf. Aufgrund des Schweregrads „warning“ handelt es sich allerdings um keinen schwerwiegenden Verstoß. Demgegenüber drückt eine tolerierte Abhängigkeit aus, dass die Abhängigkeit nicht verboten, aber auch nicht erwünscht ist. Für einen Modellierer von Abhängigkeitsrestriktionen sind diese beiden Fälle also zu unterscheiden.

Ob eine Abhängigkeitsrestriktion die Abhängigkeit zu einem Feature, Plugin, einer Feature-Gruppe oder einer Plugin-Gruppe betrifft, wird durch das Nichtterminalsymbol `ConcreteDependency` ausgedrückt (Zeile 3, 26 – 31):

- `DependencyToSinglePlugin` repräsentiert eine Abhängigkeitsrestriktion, die eine Abhängigkeit hin zu einem einzelnen Plugin betrifft (Zeile 27, 33 – 35). Der Querverweis auf `PluginName` stellt sicher, dass ein Fehler gemeldet wird, wenn der referenzierte Plugin-Name nicht in der Plugin-Basis definiert wurde.
- `DependencyToPluginGroup` kapselt eine Abhängigkeitsrestriktion, die eine Abhängigkeit hin zu einer Plugin-Gruppe betrifft (Zeile 28, 37 – 39). Der Querverweis auf `PluginGroup` stellt sicher, dass ein Fehler gemeldet wird, wenn die referenzierte Plugin-Gruppe nicht deklariert wurde.
- `DependencyToSingleFeature` repräsentiert eine Abhängigkeitsrestriktion, die eine Abhängigkeit hin zu einem einzelnen Feature betrifft (Zeile 29, 41 – 43). Analog zum Plugin-Fall muss der Name des genannten Features in der Feature-Basis definiert worden sein.
- `DependencyToFeatureGroup` drückt eine Abhängigkeitsrestriktion aus, die eine Abhängigkeit hin zu einer Feature-Gruppe betrifft (Zeile 30, 45 – 47). Analog zu der Situation bei Plugin-Gruppen, muss auf eine Feature-Gruppe verwiesen werden, die im Abhängigkeitsmodell deklariert wurde.
- Die Nichtterminalsymbole `DependencyToPluginPattern` (Zeile 27) und `DependencyToFeaturePattern` (Zeile 29) werden später erläutert.

Listing 8.8 zeigt konkrete Beispiele für die Definition von Abhängigkeitsrestriktionen. Gemäß dieser Beispiele sind Abhängigkeiten von Plugin `p1` zu Feature `f2` und zu allen Features aus der Feature-Gruppe `fg2` verboten. Verstöße gegen die erste Restriktion werden als kritische Fehler angesehen, während Verstöße gegen die zweite Restriktion lediglich zu Warnungen führen. Weiterhin sagen die Beispiele aus Listing 8.8 aus, dass Abhängigkeiten von der Feature-Gruppe `fg1` zu allen Features, die in der Feature-Gruppe `fg3` enthalten sind, nur toleriert werden.

Soll eine Abhängigkeitsrestriktion zu einer Menge von Features oder Plugins definiert werden, muss auf Basis der bisher vorgestellten Sprachkonstrukte für diese Menge eine Feature- oder Plugin-Gruppe deklariert werden. Zusätzlich hierzu bietet `DepCoL` die Möglichkeit, in Abhängigkeitsrestriktionen die Abhängigkeiten zu Feature- oder Plugin-Mustern zu definieren, die

```

1 plugin p1 {
2   [critical] forbid dependency to feature f2;
3   [warning] forbid dependency to featuregroup fg2;
4 }
5
6 featuregroup fg1 {
7   tolerate dependency to featuregroup fg3;
8 }

```

DepCoL

Listing 8.8: Beispielhafte Abhängigkeitsrestriktionen für verbotene und tolerierte Abhängigkeiten.

direkt durch einen einfachen regulären Ausdruck beschrieben werden können. Listing 8.7 nutzt dazu die Nichtterminalsymbole `DependencyToPluginPattern` (Zeile 27) sowie `DependencyToFeaturePattern` (Zeile 29). Die Grammatik für diese beiden Nichtterminalsymbole ist in Listing 8.9 aufgeführt.

```

1 DependencyToPluginPattern:
2   'pluginpattern' pluginPtn = NameExp
3   ;
4
5 DependencyToFeaturePattern:
6   'featurepattern' featurePtn = NameExp
7   ;

```

Xtext

Listing 8.9: Grammatik der DSL DepCoL zur Spezifikation von Abhängigkeitsrestriktionen zu Feature- oder Plugin-Mustern.

In beiden Fällen wird der reguläre Ausdruck durch das Nichtterminalsymbol `NameExp` beschrieben (Zeile 2, 6). Der Vorteil von Feature- und Plugin-Mustern gegenüber Feature- und Plugin-Gruppen ist, dass eine Menge von Features und Plugins referenziert werden kann, die nicht zuvor deklariert werden muss. Bezüglich der Ausdrucksmächtigkeit unterscheiden sich die beiden Sprachkonstrukte hingegen nicht. Der Vorteil von Feature- und Plugin-Gruppen ist demgegenüber, dass die Gruppen an verschiedenen Stellen referenziert werden können, ohne die regulären Ausdrücke erneut definieren zu müssen. Listing 8.10 zeigt beispielhaft die Verwendung von Feature- und Plugin-Mustern. Durch die erste Restriktion in Zeile 2 wird ausgedrückt, dass das Plugin `p1` keine Abhängigkeiten zu Features haben darf, deren Namen mit `f1` beginnen. Ferner drückt die zweite Restriktion aus, dass das Plugin `p1` keine Abhängigkeiten zu Plugins haben darf, deren Namen mit `ab` enden.

```

1 plugin p1 {
2   forbid dependency to featurepattern f1*;
3   forbid dependency to pluginpattern *ab;
4 }

```

DepCoL

Listing 8.10: Beispielhafte Spezifikation von Abhängigkeitsrestriktionen unter Nutzung von Feature-Mustern und Plugin-Mustern.

Eine wichtige Eigenschaft von DepCoL ist, dass Abhängigkeitsrestriktionen zuvor definierte Abhängigkeitsrestriktionen überschreiben können. Beispielsweise kann zunächst ausgedrückt werden, dass die Plugins einer Plugin-Gruppe keine Abhängigkeiten zu Plugins einer anderen Plugin-Gruppe haben dürfen. Anschließend kann definiert werden, dass ausgewählte Plugins doch Abhängigkeiten zu dedizierten Plugins der anderen Plugin-Gruppe haben dürfen. Das Überschreiben von Abhängigkeitsrestriktionen wird detaillierter in Abschnitt 8.4 erläutert.

8.4 Semantik eines Abhängigkeitsmodells

Sei DM die Menge aller Abhängigkeitsmodelle, dann lässt sich ein Abhängigkeitsmodell $d \in DM$ wie folgt darstellen: $d = (P, F, PG, FG, C)$, wobei P die Menge aller Plugins, F die Menge aller Features, PG die Menge aller Plugin-Gruppen, FG die Menge aller Feature-Gruppen und C die Menge aller Abhängigkeitsrestriktionen des Abhängigkeitsmodells darstellt. Im Folgenden wird zunächst präzisiert, wie sich Features, Feature-Gruppen und Plugin-Gruppen ausdrücken lassen.

- Der Einfachheit halber identifizieren wir Features als Menge ihrer Plugins: $F \subseteq \mathcal{P}(P)$.
- Feature-Gruppen sind Paare bestehend aus anderen Feature-Gruppen und Features: $FG = \{(g, f) \mid g \subseteq FG, f \subseteq F\}$.
- Plugin-Gruppen sind analog dazu Paare bestehend aus anderen Plugin-Gruppen und Plugins: $PG = \{(g, p) \mid g \subseteq PG, p \subseteq P\}$.

Ferner sei $A = P \cup F \cup PG \cup FG$ die Menge aller Plugins und Features eines Plugin-basierten Softwaresystems und aller Plugin-Gruppen sowie Feature-Gruppen eines Abhängigkeitsmodells. Dann kann für ein Abhängigkeitsmodell $d = (P, F, PG, FG, C) \in DM$ eine Funktion $PS_d: A \rightarrow \mathcal{P}(P)$ definiert werden, die zu einem Plugin, Feature, einer Plugin-Gruppe oder einer Feature-Gruppe die darüber repräsentierte Menge von Plugins zurückliefert:

$$PS_d(x) = \begin{cases} \{x\} & \text{falls } x \in P \\ x & \text{falls } x \in F \subseteq P \\ \bigcup_{y \in g} PS_d(y) \cup p & \text{falls } x = (g, p) \in PG \\ \bigcup_{y \in g} PS_d(y) \cup \bigcup_{z \in f} PS_d(z) & \text{falls } x = (g, f) \in FG \end{cases}$$

Die Abbildung PS_d ist wohldefiniert, wenn weder Feature- noch Plugin-Gruppen Zyklen bilden.

Wie zuvor bereits eingeführt, enthält die Menge C sämtliche Abhängigkeitsrestriktionen eines Abhängigkeitsmodells. Eine konkrete Abhängigkeitsrestriktion $c \in C$ lässt sich wie folgt als Tupel darstellen: $c = (s, t, k, i)$ mit $s, t \in A, i \in \mathbb{N}, k \in K$ wobei die Menge K wie folgt definiert ist:

$$K = \{critical, error, warning, allowed\}$$

Gemäß dieser Definition gibt k an, ob die Abhängigkeit zwischen s und t erlaubt ist oder zu einem kritischen Fehler, einem Fehler oder einer Warnung führt. Ferner gibt der Index i die eindeutige Position der Abhängigkeitsrestriktion innerhalb des Abhängigkeitsmodells an. Je niedriger dieser Index ist, desto weiter vorne im Abhängigkeitsmodell ist die Abhängigkeitsrestriktion definiert. Der Index einer Abhängigkeitsrestriktion wird später für die Definition des Überschreibens von Abhängigkeitsrestriktionen benötigt. Auf Basis der vorherigen Ausführungen lässt sich eine Abhängigkeitsrestriktion $c \in C$ eines Abhängigkeitsmodells $d \in DM$ über eine Menge von Plugin-Paaren ausdrücken, wie nachfolgend durch die Funktion $PP_d: C \rightarrow P \times P$ gezeigt für $c = (s, t, k, i)$:

$$PP_d(c) = \{(x, y) \mid x \in PS_d(s), y \in PS_d(t)\}$$

Aufbauend auf dieser Funktion PP_d wird nachfolgend das Konzept des Überschreibens von Abhängigkeitsrestriktionen eingeführt (siehe Definition 30), welches für das Verständnis der Semantik eines Abhängigkeitsmodells wichtig ist.

Definition 30 (Überschreiben von Abhängigkeitsrestriktionen)

Seien $b, c \in C$ mit $b = (s, t, k, i)$ und $c = (l, m, k', j)$ zwei Abhängigkeitsrestriktionen des Abhängigkeitsmodells $d \in DM$. Unter der Voraussetzung, dass b im Abhängigkeitsmodell vor c definiert ist, also $i < j$, überschreibt c die Abhängigkeitsrestriktion b genau dann, wenn gilt: $PP_d(b) \cap PP_d(c) \neq \emptyset$. Eine Abhängigkeitsrestriktion überschreibt eine andere Abhängigkeitsrestriktion also genau dann, wenn beide Restriktionen nicht disjunkte Mengen von Plugin-Paaren beschreiben.

Für zwei Abhängigkeitsrestriktionen $b, c \in C$ enthält der Durchschnitt zwischen $PP_d(b)$ und $PP_d(c)$ genau die Menge der überschriebenen Plugin-Paare. Das Überschreiben von Abhängigkeitsrestriktionen ist für die Semantik eines Abhängigkeitsmodells insofern relevant, als dass dadurch zu einem Plugin-Paar mehrere Abhängigkeitsrestriktionen existieren können, die Restriktionen bezüglich des Plugin-Paars ausdrücken.

Aufbauend auf den zuvor definierten Funktionen lässt sich die Semantik eines Abhängigkeitsmodells $d \in DM$ über eine Zuordnung einer Eigenschaft aus der Menge K zu Plugin-Paaren beschreiben, wie nachfolgend durch die Funktion $SD_d: P \times P \rightarrow K$ gezeigt:

$$SD_d(x, y) = \begin{cases} k & \text{falls } \exists (s, t, k, i) \in C : x \in PS_d(s), y \in PS_d(t) \text{ und} \\ & \nexists (s', t', k', i') \in C : x \in PS_d(s'), y \in PS_d(t') \text{ und } i' > i \\ \text{allowed} & \text{sonst} \end{cases}$$

Die Fallunterscheidung in dieser Funktion wird nachstehend genauer erläutert:

- Es liegt mindestens eine Abhängigkeitsrestriktion vor, die das Plugin-Paar (x, y) beschreibt. Unter diesen Abhängigkeitsrestriktionen entscheidet die Abhängigkeitsrestriktion, die als Letztes im Abhängigkeitsmodell definiert ist, welche Eigenschaft dem Plugin-Paar zugeordnet wird. Alle anderen Abhängigkeitsrestriktionen werden von dieser Restriktion überschrieben, so dass nicht betrachtet wird, welche Eigenschaft sie dem Plugin-Paar zuordnen.

- Andernfalls liegt keine Abhängigkeitsrestriktion vor, die das Plugin-Paar (x, y) beschreibt. In diesem Fall wird eine Abhängigkeit zwischen $x \in P$ und $y \in P$ standardmäßig als erlaubt betrachtet.

Zusammengefasst bedeutet das, dass ein Abhängigkeitsmodell eine Abbildung von Plugin-Paaren auf eine Eigenschaft beschreibt, die ausdrückt, ob eine Abhängigkeit zwischen den entsprechenden Plugins erlaubt ist oder zu kritischen Fehlern, einem Fehler oder einer Warnung führt.

8.5 Vorgehensweise zur Prüfung der Architekturkonformität

8.5.1 Implementierung

Die Prüfkomponeute kann auf drei verschiedene Weisen damit beauftragt werden, die Architekturkonformität zu prüfen:

- Nach dem Start der Entwicklungsumgebung, in der das Plugin-basierte Softwaresystem entwickelt wird, kann die Prüfkomponeute automatisch die Architekturkonformität sämtlicher Features und Plugins des Plugin-basierten Softwaresystems prüfen. Ob diese initiale Prüfung nach dem Start der Entwicklungsumgebung erfolgen soll oder nicht kann konfiguriert werden. Standardmäßig ist diese Prüfung aktiviert.
- Ein Entwickler stößt manuell die Prüfung der Architekturkonformität sämtlicher Plugins des Plugin-basierten Softwaresystems an. Die Entwickler können die Prüfung zu jeder Zeit manuell initiieren.
- Die Prüfkomponeute ist als Listener in der Entwicklungsumgebung angemeldet und wird über Änderungen von Dateien benachrichtigt. Falls die geänderte Datei eine Manifest-Datei ist, wird die Prüfung der Architekturkonformität des betroffenen Plugins angestoßen. Der Grund dafür ist, dass durch die Änderung der Manifest-Datei die Abhängigkeiten des Plugins geändert worden sein könnten und die Abhängigkeiten daher erneut gegen das Abhängigkeitsmodell geprüft werden müssen. Die Ausführung der Prüfkomponeute nach Änderungen von Manifest-Dateien kann ebenfalls konfiguriert werden. Standardmäßig ist diese Prüfung aktiviert.

Der Pseudocode der Prüfkomponeute zur automatischen Prüfung der Architekturkonformität ist in Algorithmus 8.1 dargelegt. Dieser Pseudocode prüft, ob ein spezifisches Plugin gegen die Abhängigkeitsrestriktionen aus dem Abhängigkeitsmodell verstößt. Der Anwendungsfall, dass die Architekturkonformität aller Plugins eines Softwaresystems geprüft wird, lässt sich über die wiederholte Ausführung der genannten Schritte für jedes einzelne Plugin abbilden.

Der erste Schritt der Konformitätsprüfung besteht in der Prüfung, ob die Abhängigkeiten des zu prüfenden Plugins extrahiert werden müssen (Zeile 1). Diese Extraktion (Zeile 2) ist in genau zwei Fällen erforderlich: 1. die Abhängigkeiten des Plugins wurden zuvor noch nicht extrahiert oder 2. die Prüfkomponeute wurde benachrichtigt, dass die Manifest-Datei des Plugins geändert wurde. Zur Abbildung des zweiten Falls wird an die Prüfkomponeute ein boolescher Wert

Algorithmus 8.1 : Function checkArchitectureConformance()

```

input  : plugin: Plugin whose architecture conformity is validated,
          changedPlugin: Boolean value indicating whether conformity check
          was initiated by plugin
output : Architecture violations caused by checked plugin

1 if dependencyExtractionRequired(changedPlugin) then
2   | extractPluginDependencies(plugin);
3 if dependency model has changed since last check then
4   | loadDependencyModel();
   /* Traverse from end to beginning */
5 for each depModelElem : getDepModelElements() do
6   | if depModelElem contains DRs which concern plugin or feature of plugin then
   /* Traverse from end to beginning */
7     | for each depRestriction : getDRs(depModelElem) do
8       | reqPlugins ← getRequiredPluginsDRRefersTo(depRestriction);
9       | for each reqPlugin : reqPlugins do
10        | storedDepRestriction ← getDRForPluginTuple(
11          | plugin, reqPlugin);
12        | if storedDepRestriction does not exist then
13          | | addEntryToR(plugin, reqPlugin, depRestriction);
14        | reqFeatures = getRequiredFeaturesDRRefersTo(depRestriction);
15        | for each reqFeature : reqFeatures do
16          | reqPlugins = getRequiredPluginsFromFeature(reqFeature);
17          | for each pluginFromReqFeature : reqPlugins do
18            | storedDepRestriction ← getDRForPluginTuple(
19              | plugin, reqPlugin);
20            | if storedDepRestriction does not exist then
21              | | addEntryToR(plugin, reqPlugin, depRestriction);

22 return determineViolationResults(R);

```

`changedPlugin` übergeben. Dieser drückt aus, ob der Aufruf der Prüfkomponeute daraus resultiert, dass die Manifest-Datei geändert wurde. Sind beide Bedingungen nicht erfüllt, ist eine erneute Extraktion der Plugin-Abhängigkeiten nicht erforderlich und die Plugin-Abhängigkeiten werden für die Prüfung aus einem Cache geladen. Auf diese Weise wird verhindert, dass die Abhängigkeiten bei jeder Ausführung der Prüfkomponeute extrahiert werden.

Im Anschluss daran wird das Abhängigkeitsmodell geladen, sofern dies erforderlich ist (Zeile 3). Dieser Ladevorgang wird nur dann durchgeführt, wenn das Abhängigkeitsmodell noch nicht eingelesen wurde oder wenn sich das Abhängigkeitsmodell seit der letzten Prüfung der Architekturkonformität geändert hat. Andernfalls wurde das Abhängigkeitsmodell zuvor verarbeitet und die Informationen dazu liegen in der Prüfkomponeute noch vor.

Als Nächstes werden die Elemente des Abhängigkeitsmodells durchlaufen (Zeile 5 – 21). Zur Prüfung der Architekturkonformität des zu untersuchenden Plugins werden sämtliche Strukturen bestimmt, die Abhängigkeitsrestriktionen beinhalten, die für das zu untersuchende Plugin oder das zugehörige Feature relevant sind. Unter Verwendung der in Abschnitt 8.4 eingeführten Mengen und Funktionen, trifft dies genau dann zu, wenn eine Struktur $a \in A$ vorliegt, so dass für ein zu prüfendes Plugin x gilt: $x \in PS(a)$. Das zu untersuchende Plugin muss also ein Element der Menge sein, die durch die Struktur a beschrieben wird. Dies trifft genau dann zu, wenn eine Struktur:

- für das zu prüfende Plugin x definiert ist und somit das Plugin direkt betrifft.
- für das Feature definiert ist, welches das Plugin x umfasst.
- für eine Plugin-Gruppe definiert ist, welche das Plugin x enthält.
- für eine Feature-Gruppe definiert ist, welche das Feature umfasst, welches das Plugin x kapselt.

Die Prüfung, ob das aktuell betrachtete Element eine solche Struktur darstellt, erfolgt in Zeile 6. Verläuft diese Prüfung positiv, werden darauffolgend die Abhängigkeitsrestriktionen verarbeitet, die in dieser Struktur enthalten sind (Zeile 7 – 21). Ein wichtiger Aspekt der von der Prüfkomponeute beachtet werden muss ist, dass Abhängigkeitsrestriktionen innerhalb eines Abhängigkeitsmodells überschrieben werden können. Dieser Sachverhalt wird umgesetzt, indem das Abhängigkeitsmodell von hinten nach vorne verarbeitet wird. Des Weiteren werden innerhalb einer Struktur, die Abhängigkeitsrestriktionen enthält, die Restriktionen analog von hinten nach vorne verarbeitet. Auf diese Weise werden als Erstes die Abhängigkeitsrestriktionen eingelesen und beachtet, die andere Abhängigkeitsrestriktionen überschreiben können. Diese werden folglich mit höherer Priorität behandelt.

Für jede ermittelte Abhängigkeitsrestriktion wird als Erstes untersucht, ob sie auf Plugins verweist und falls ja, auf welche (Zeile 8). In diesem Kontext sind allerdings nur die Plugins relevant, von denen das zu prüfende Plugin abhängt. Dieser Schritt richtet sich nach der Art der Abhängigkeitsrestriktion:

- Für eine Abhängigkeitsrestriktion die auf ein einzelnes Plugin verweist, wird das referenzierte Plugin nur dann zurückgeliefert, wenn das zu prüfende Plugin von diesem abhängt.

- Für eine Abhängigkeitsrestriktion die auf eine Plugin-Gruppe verweist, werden sämtliche Plugins dieser Gruppe zurückgeliefert, von denen das zu prüfende Plugin abhängt.
- Für eine Abhängigkeitsrestriktion die auf ein Plugin-Muster verweist, werden analog sämtliche Plugins zurückgeliefert, auf die das Plugin-Muster zutrifft und von denen das zu prüfende Plugin abhängt.

Während der Abarbeitung der verschiedenen relevanten Abhängigkeitsrestriktionen erzeugt und verwaltet die Prüfkomponente eine Relation $R \subseteq P \times P \times C$ für die Paare von Plugins, zu denen gemäß Abhängigkeitsmodell Abhängigkeitsrestriktionen definiert sind. Konkret kapselt die Relation zu einem Plugin-Paar die relevante Abhängigkeitsrestriktion, welche eine Aussage über die Art der Abhängigkeit zwischen den beiden Plugins macht. Existieren mehrere Abhängigkeitsrestriktionen bezüglich eines Plugin-Paars, enthält R nur die Abhängigkeitsrestriktion, welche die anderen überschreibt. Auf diese Weise kann aus der Relation R abgeleitet werden, zwischen welchen Paaren von Plugins gemäß Abhängigkeitsmodell Abhängigkeiten erlaubt sind oder bei denen Abhängigkeiten zu kritischen Fehlern, Fehlern oder Warnungen führen.

Für jedes zurückgelieferte Plugin (Zeile 9 – 13) wird geprüft, ob bereits eine Abhängigkeitsrestriktion zu dem zu prüfenden Plugin und dem zurückgelieferten Plugin in R hinterlegt ist (Zeile 10 – 11). Existiert eine solche Abhängigkeitsrestriktion nicht, wird R um einen Eintrag zu dem Plugin-Paar und der aktuellen Abhängigkeitsrestriktion erweitert (Zeile 12 – 13). Liegt bereits eine Abhängigkeitsrestriktion in R vor, wurde zuvor bereits eine Abhängigkeitsrestriktion verarbeitet, welche die aktuell betrachtete Abhängigkeitsrestriktion überschrieben hat. Dies ist in der Traversierung des Abhängigkeitsmodells von hinten nach vorne (Zeile 5 – 21) und in der Verarbeitung der Abhängigkeitsrestriktionen von hinten nach vorne (Zeile 7 – 21) begründet.

Zwischen der Relation R und der zuvor erläuterten Abbildung von Plugin-Paaren auf eine Eigenschaft, gibt es zwei wichtige Unterschiede. Zum einen werden zu jeder Abhängigkeitsrestriktion nur solche Plugin-Tupel (x, y) betrachtet, bei denen y ein Plugin ist, zu dem Plugin x eine Abhängigkeit besitzt. Folglich beinhaltet R ausschließlich Informationen zu dem analysierten Plugin und dessen benötigten Plugins. Der zweite wichtige Unterschied betrifft die Tatsache, dass in R nur Plugin-Paare verwaltet werden, die durch die explizit definierten Abhängigkeitsrestriktionen erzeugt wurden. Falls es keine Restriktion zu der Abhängigkeit zwischen dem zu prüfenden Plugin x und einem anderen Plugin y gibt, ist in R zu dem Plugin-Paar (x, y) keine Abhängigkeitsrestriktion hinterlegt.

Als Nächstes wird der Fall betrachtet, dass eine Abhängigkeitsrestriktion auf Features verweist. In diesem Fall muss geprüft werden, ob das Plugin eine Abhängigkeit zu einem Plugin dieser Features besitzt. Dazu wird zunächst ermittelt, auf welche Features die jeweilige Abhängigkeitsrestriktion verweist (Zeile 14). Dabei werden allerdings nur solche Features zurückgeliefert, die Plugins enthalten, von denen das zu prüfende Plugin abhängt. Dieser Schritt richtet sich wiederum nach der Art der vorliegenden Abhängigkeitsrestriktion:

- Für eine Abhängigkeitsrestriktion die auf ein einzelnes Feature verweist, wird das referenzierte Feature nur dann zurückgeliefert, falls das zu prüfende Plugin von einem Plugin des Features abhängt.

- Für eine Abhängigkeitsrestriktion die auf eine Feature-Gruppe verweist, werden sämtliche Features zurückgeliefert, bei denen das Plugin von einem Plugin des jeweiligen Features abhängt.
- Für eine Abhängigkeitsrestriktion die auf ein Feature-Muster verweist, werden analog sämtliche Features zurückgeliefert, auf die das Feature-Muster zutrifft und bei denen das zu prüfende Plugin von einem Plugin des Features abhängt.

Über die resultierenden Features wird schließlich iteriert (Zeile 15 – 21). Für jedes Feature werden sämtliche Plugins des Features ermittelt, von denen das zu prüfende Plugin abhängt (Zeile 16). Für diese Plugins wird wieder so verfahren (Zeile 17 – 21), wie zuvor für Abhängigkeitsrestriktionen beschrieben, die auf Plugins verweisen (Zeile 9 – 13).

Abschließend werden auf Basis der Einträge in R die Architekturverletzungen des geprüften Plugins ermittelt und zurückgegeben (Zeile 22). Die Vorgehensweise dazu ist nachfolgend dargestellt. Dabei müssen nur für solche Einträge $(p, q, c) \in R$ Warnungen oder Fehler gemeldet werden, bei denen die Abhängigkeitsrestriktion c keine erlaubte Abhängigkeit betrifft.

- Falls c eine Abhängigkeitsrestriktion darstellt, welche die Abhängigkeit zu einem Plugin, einer Plugin-Gruppe oder einem Plugin-Muster beschreibt, wird eine Fehlermeldung oder Warnung erzeugt, die ausdrückt, dass die Abhängigkeit von p zu q verboten oder toleriert ist. Wie schwerwiegend der Verstoß ist und ob eine Fehlermeldung oder eine Warnung erzeugt werden muss, geht aus der Abhängigkeitsrestriktion c hervor.
- Falls c eine Abhängigkeitsrestriktion darstellt, welche die Abhängigkeit zu einem Feature, einer Feature-Gruppe oder einem Feature-Muster beschreibt, wird eine Fehlermeldung oder Warnung erzeugt, die ausdrückt, dass die Abhängigkeit von p zu f verboten ist, wobei f das Feature ist, welches das Plugin q enthält. Ferner gibt die Fehlermeldung oder Warnung Aufschluss darüber, dass die Abhängigkeit zu Plugin q aus Feature f diese Meldung verursacht hat. Die resultierenden Fehlermeldungen und Warnungen sind dadurch näher an dem tatsächlichen Abhängigkeitsmodell. Wie schwerwiegend der Verstoß ist und ob eine Fehlermeldung oder eine Warnung erzeugt werden muss, geht wiederum aus der Abhängigkeitsrestriktion c hervor.

Zur Vereinfachung des Pseudocodes wurde davon abstrahiert, dass bei der Konformitätsprüfung durch die Prüfkomponekte verwaltet werden kann, welche Abhängigkeitsrestriktion welche andere Abhängigkeitsrestriktion überschreibt. Die Prüfkomponekte erkennt dies bei der Prüfung, ob zu einem Plugin-Tupel (p, q) bereits eine Abhängigkeitsrestriktion hinterlegt ist. Liegt eine solche Abhängigkeitsrestriktion vor, kann vermerkt werden, dass die Abhängigkeitsrestriktion, die dem Plugin-Tupel zugeordnet ist, die aktuell betrachtete Abhängigkeitsrestriktion überschreibt. Am Ende der Verarbeitung eines Abhängigkeitsmodells können die auf diese Weise gesammelten Informationen an die Entwickler gemeldet werden. Diese können daraufhin prüfen, ob die gemeldeten überschriebenen Abhängigkeitsrestriktionen gewünscht sind oder nicht. Die Motivation für diesen Anwendungsfall ist, dass gerade in großen und komplexen Abhängigkeitsmodellen die Modellierer von Abhängigkeitsrestriktionen möglicherweise nicht immer

wissen, dass eine neu hinzugefügte Abhängigkeitsrestriktion eine vorherige Abhängigkeitsrestriktion überschreibt. Ohne zusätzliche Werkzeugunterstützung kann es daher leicht dazu kommen, dass eine Abhängigkeitsrestriktion unbeabsichtigt überschrieben wird. Dieser Sachverhalt ist in dem Pseudocode nicht explizit aufgeführt, da der Pseudocode den Fokus auf die Konformitätsprüfung legt.

8.5.2 Beispiel

Die Prüfung der Architekturkonformität wird im Folgenden für ein fiktives Plugin-basiertes Softwaresystem verdeutlicht. Die Feature- und Plugin-Struktur dieses Systems ist in Abbildung 8.15 dargestellt. Wie dort zu sehen, besteht das System aus den Features *fa*, *fb* und *fc*. Das Feature *fa* besteht aus den Plugins *pla1*, *pla2*, *pla3* und sämtlichen Plugins, die mit den Namen *pa* beginnen. Letzteres wird ausgedrückt durch den regulären Ausdruck *pa**. In Abbildung 8.15 ist mit dem Plugin *pa1* ein beispielhaftes Plugin angegeben, auf das dieser reguläre Ausdruck zutrifft. Ferner enthält das Feature *fb* die Plugins *fb1* und *fb2* und das Feature *fc* enthält die Plugins *fc1* sowie *fc2*. Überdies zeigt Abbildung 8.15 die Abhängigkeiten zwischen den Plugins die für das Beispiel angenommen werden.

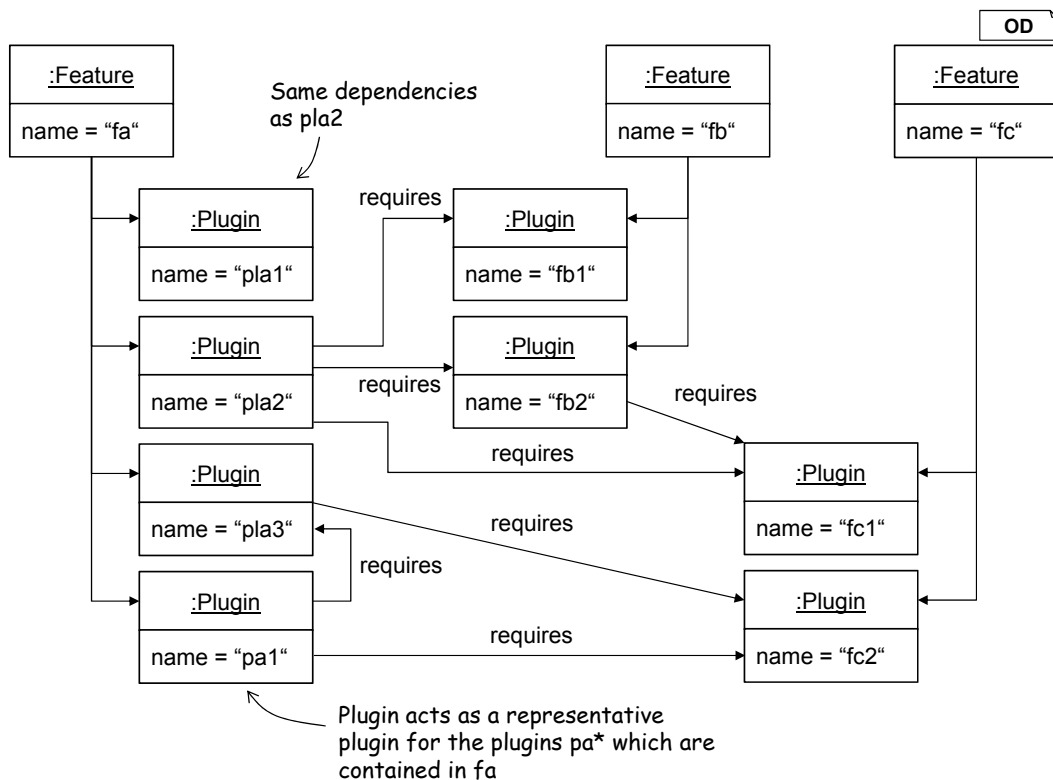


Abbildung 8.15: Überblick über die Features und Plugins eines fiktiven Plugin-basierten Softwaresystems.

Als zu untersuchendes Abhängigkeitsmodell wird das Abhängigkeitsmodell aus Listing 8.11 angenommen.

```

1 declare featuregroup fg {
2     fa;
3     fb;
4 }
5
6 declare pluginingroup pag {
7     pla1;
8     pla2;
9     pla3;
10    regexp pa*;
11 }
12
13 //Definition of constraints:
14
15 featuregroup fg {
16     forbid dependency to feature fc;
17 }
18
19 feature fa {
20     forbid dependency to pluginpattern *;
21     allow dependency to feature fa;
22 }
23
24 pluginingroup pag {
25     tolerate dependency to pluginpattern fb*;
26 }
27
28 plugin pla1 {
29     allow dependency to plugin fb1;
30 }
31
32 plugin pla2 {
33     allow dependency to pluginpattern *;
34     forbid dependency to fb2;
35 }

```

Listing 8.11: Beispielhaftes fiktives Abhängigkeitsmodell mit überschriebenen Abhängigkeitsrestriktionen.

In der folgenden Beschreibung des Beispiels wird die Relation R in die vier Relationen $R_{allowed}$, $R_{critical}$, R_{error} und $R_{warning}$ aufgeteilt. Auf diese Weise kann einfacher hervorgehoben werden, ob eine Abhängigkeit zwischen einem bestimmten Plugin-Paar erlaubt ist oder zu einem Fehler, einer Warnung oder einem kritischen Fehler führt. Wird in der Beschreibung nur auf R verwiesen, bezieht sich das auf die Vereinigung dieser vier Relationen. Ferner wird zur Vereinfachung der Beschreibung davon ausgegangen, dass diese vier Relationen nur Plugin-Paare kapseln und nicht zusätzlich noch die zugehörige Abhängigkeitsrestriktion.

Validierung des Plugins `p1a1`

Als Erstes wird im Folgenden der Fall betrachtet, dass die Architekturkonformität des Plugins `p1a1` geprüft wird. In der Beispielbeschreibung verweist *PB* auf die Plugin-Basis, also auf die Menge aller zulässigen Plugin-Namen.

Da als Erstes das Plugin `p1a1` validiert wird, werden nur die Abhängigkeitsrestriktionen in die Konformitätsprüfung einbezogen, welche das Plugin `p1a1` oder das zugehörige Feature `fa` betreffen. Bei dem vorliegenden Abhängigkeitsmodell werden daher die Abhängigkeitsrestriktionen betrachtet, die für das Plugin selbst (Zeile 28 – 30), die Plugin-Gruppe `pag` (Zeile 24 – 26), das Feature `fa` (Zeile 19 – 22) und die Feature-Gruppe `fg` (Zeile 15 – 17) definiert wurden. Demgegenüber sind die Abhängigkeitsrestriktionen, die für Plugin `p1a2` (Zeile 32 – 35) angegeben sind, nicht relevant für `p1a1`.

Im ersten Schritt wird aufgrund der Abhängigkeitsrestriktion aus Zeile 29 das Plugin-Tupel $(p1a1, fb1)$ zu $R_{allowed}$ hinzugefügt, da R dieses Plugin noch nicht enthält und `p1a1` eine Abhängigkeit zu `fb1` besitzt.

Im Anschluss daran fährt die Prüfkompone mit den Abhängigkeitsrestriktionen zur Plugin-Gruppe `pag` fort. Bei der Verarbeitung der Abhängigkeitsrestriktion in Zeile 25 wird geprüft, ob die Plugin-Tupel $(p1a1, y)$ zu R hinzugefügt werden können, wobei y ein Plugin sein muss, dessen Name mit der Zeichenkette `fb` beginnt. Übertragen auf das vorliegende Plugin-basierte Softwaresystem wird also geprüft, ob die Tupel $(p1a1, fb1)$ und $(p1a1, fb2)$ zu R hinzugefügt werden können. Zu diesem Zeitpunkt enthält $R_{allowed}$ bereits $(p1a1, fb1)$, so dass nur das Tupel $(p1a1, fb2)$ zu $R_{warning}$ hinzugefügt wird. Das Tupel wird zu $R_{warning}$ hinzugefügt, da die Abhängigkeitsrestriktion ausdrückt, dass die Abhängigkeit zum Plugin-Muster `fb*` toleriert wird.

Danach prüft die Prüfkompone die Abhängigkeitsrestriktionen, die für Feature `fa` angegeben sind, beginnend mit der Abhängigkeitsrestriktion in Zeile 21. Da das Plugin `p1a1` keine Abhängigkeit zu einem Plugin des Features `fa` besitzt, wird kein Tupel zu R hinzugefügt. Die Prüfkompone fährt mit der Abhängigkeitsrestriktion aus Zeile 20 fort. Diese führt dazu, dass die Tupel $(p1a1, y)$ mit $y \in PB$ zu R_{error} hinzugefügt werden, falls R diese Tupel nicht bereits enthält. Das einzige solche Tupel ist $(p1a1, fc1)$. Zusammengefasst bedeutet das, dass $R_{allowed}$ das Plugin-Tupel $(p1a1, fb1)$, $R_{warning}$ das Plugin-Tupel $(p1a1, fb2)$ und R_{error} das Plugin-Tupel $(p1a1, fc1)$ enthält. Folglich wird eine Fehlermeldung gemeldet, die auf die verbotene Abhängigkeit zwischen `p1a1` und `fc1` hinweist und eine Warnung erzeugt, die auf die lediglich tolerierte Abhängigkeit zwischen `p1a1` und `fb2` hinweist. Eine Übersicht über den Inhalt der verschiedenen R -Relationen gibt Tabelle 8.17 auf Seite 254.

Validierung des Plugins `p1a2`

Als Nächstes wird der Fall betrachtet, dass das Plugin `p1a2` validiert wird. Die ersten relevanten Abhängigkeitsrestriktionen betreffen das Plugin `p1a2` selbst (Zeile 32 – 35). Da `p1a2` eine Abhängigkeit zum Plugin `fb2` besitzt und die Abhängigkeitsrestriktion in Zeile 34 diese Abhängigkeit verbietet, wird das Tupel $(p1a2, fb2)$ zu R_{error} hinzugefügt. Im Anschluss daran werden aufgrund der Abhängigkeitsrestriktion in Zeile 33 die Tupel $(p1a2, y)$ zu $R_{allowed}$ hinzugefügt, falls R diese Tupel noch nicht enthält. Hierbei stellen $y \in PB$ Plugins dar, zu denen

p_{la2} eine Abhängigkeit besitzen muss. Konkret wird also geprüft, ob folgende Plugin-Tupel zu R hinzugefügt werden können: $(p_{la2}, fb1)$, $(p_{la2}, fb2)$ und $(p_{la2}, fc1)$. Von diesen Tupeln ist lediglich das Tupel $(p_{la2}, fb2)$ in R enthalten, so dass die anderen beiden Tupel zu $R_{allowed}$ hinzugefügt werden.

Daraufhin wird die Abhängigkeitsrestriktion analysiert, die für die Plugin-Gruppe pag definiert ist (Zeile 24 – 26). Diese hat aber keinen Einfluss, da bereits sämtliche relevanten Plugin-Tupel bezüglich p_{la2} in R enthalten sind. Aus dem gleichen Grund sind auch die Abhängigkeitsrestriktionen, die für Feature fa (Zeile 19 – 22) und für die Feature-Gruppe fg (Zeile 15 – 17) definiert sind, ohne Folgen. Dementsprechend wird als Ergebnis gemeldet, dass das Plugin p_{la2} eine verbotene Abhängigkeit zu Plugin $fb2$ besitzt.

Validierung des Plugins p_{la3}

Bei den bisherigen beiden Beispielen wurden stets Plugins validiert, zu denen selbst explizit Abhängigkeitsrestriktionen angegeben waren. Wird demgegenüber beispielsweise das Plugin p_{la3} validiert, so werden ausschließlich die Abhängigkeitsrestriktionen analysiert, die für die Plugin-Gruppe pag (Zeile 24 – 26), das Feature fa (Zeile 19 – 22) und die Feature-Gruppe fg (Zeile 15 – 17) angegeben sind. Die Abhängigkeitsrestriktion, die für die Plugin-Gruppe pag definiert ist, hat in dem vorliegenden Fall keine Folgen, da es kein Plugin gibt, zu dem das Plugin p_{la3} eine Abhängigkeit besitzt und auf welches der reguläre Ausdruck fb^* zutrifft. Ferner besitzt p_{la3} keine Abhängigkeiten zu einem Plugin des Features fa , so dass auch die Abhängigkeitsrestriktion aus Zeile 21 keine Auswirkungen hat. Erst durch die Abhängigkeitsrestriktion aus Zeile 20 wird das Plugin-Tupel $(p_{la3}, fc2)$ zu R_{error} hinzugefügt, da p_{la3} eine Abhängigkeit zu $fc2$ besitzt.

Validierung des Plugins $pa1$

Auf ähnliche Weise führen erst die Abhängigkeitsrestriktionen zu dem Feature fa (Zeile 19–22) bei der Validierung des Plugins $pa1$ dazu, dass R mit Tupeln befüllt wird. Konkret wird hier das Tupel $(pa1, p_{la3})$ zu $R_{allowed}$ hinzugefügt, aufgrund der Abhängigkeitsrestriktion aus Zeile 21. Darauffolgend wird das Tupel $(pa1, fc2)$ zu R_{error} hinzugefügt, bedingt durch die Abhängigkeitsrestriktion aus Zeile 20.

Validierung des Plugins $fb2$

Die vorherigen Beispiele betrafen allesamt Plugins des Features fa . Wird hingegen ein Plugin des Features fb validiert, z.B. $fb2$, so werden ausschließlich die Abhängigkeitsrestriktionen untersucht, die für die Feature-Gruppe fg (Zeile 15 – 17) angegeben sind. In diesem Fall drückt die Restriktion in Zeile 16 aus, dass Abhängigkeiten zu Plugins des Features fc verboten sind, mit der Folge, dass $(fb2, fc1)$ zu R_{error} hinzugefügt wird.

Zusammenfassung

Wie zuvor bereits beschrieben, fasst Tabelle 8.17 die Ergebnisse für die beschriebenen Fälle zusammen.

Validiertes Plugin	$R_{allowed}$	$R_{critical}$	R_{error}	$R_{warning}$
pla1	$\{(pla1,fb1)\}$	$\emptyset$	$\{(pla1,fc1)\}$	$\{(pla1,fb2)\}$
pla2	$\{(pla1,fb1), (pla1,fc1)\}$	$\emptyset$	$\{(pla2,fb2)\}$	$\emptyset$
pla3	$\emptyset$	$\emptyset$	$\{(pla3,fc2)\}$	$\emptyset$
pa1	$\{(pa1,pla3)\}$	$\emptyset$	$\{(pa1,fc2)\}$	$\emptyset$
fb2	$\emptyset$	$\emptyset$	$\{(fb2,fc1)\}$	$\emptyset$
fc1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tabelle 8.17: Übersicht über die Ergebnisse zur Prüfung der Architekturkonformität.

Vollständigkeitshalber sei noch angemerkt, dass die R -Relationen nicht befüllt werden, wenn ein Plugin validiert wird, zu dem keine relevanten Abhängigkeitsrestriktionen vorliegen. So wird bei der Validierung des Plugins $fc1$ kein Tupel zu R hinzugefügt, da keine Abhängigkeitsrestriktion existiert, welche für dieses Plugin relevant ist.

8.6 Anwendungsmethodik

Der in dieser Arbeit vorgestellte Ansatz erlaubt die inkrementelle Einführung einer automatischen Architekturkonformitätsprüfung für Plugin-basierte Softwaresysteme. Im Folgenden wird eine Methodik vorgestellt, die beschreibt, wie der Ansatz in realen Projekten eingesetzt werden kann. Die Ausführungen sind angelehnt an die Prozessunterstützung aus der Arbeit [DHHJ10].

1. Der erste Schritt zur Integration der Architekturkonformitätsprüfung besteht in der Erstellung einer initialen Version eines Abhängigkeitsmodells für das Softwaresystem. Dieser Schritt kann durch die Erfahrungswerte der Systemarchitekten, aber auch durch externe Dokumentation oder durch die Dokumentation des Quellcodes getrieben werden. Das Hauptaugenmerk bei der initialen Version sollte auf der Definition der wichtigsten Feature- und Plugin-Gruppen liegen. Bei geschichteten Softwaresystemen [BMR⁺96] bietet es sich beispielsweise an Gruppen anzulegen, welche die verschiedenen Schichten repräsentieren. Zusätzlich zu der Definition der wesentlichen Gruppen sollten auch bereits erste Abhängigkeitsrestriktionen zwischen den Gruppen definiert werden.
2. Sobald eine erste Initialversion vorliegt, sollte diese Version einer Bewertung unterzogen werden. Falls möglich, sollte eine Plausibilitätsprüfung der Initialversion durch andere Systemarchitekten oder erfahrene Entwickler erfolgen. Darüber hinaus kann die Prüfkomponente mit der Initialversion ausgeführt werden. Die gemeldeten Verstöße können wiederum von Systemarchitekten und erfahrenen Entwicklern diskutiert werden.
3. Die Initialversion des Abhängigkeitsmodells kann darauffolgend sukzessive verfeinert werden. Im Zuge dessen werden üblicherweise feingranularere Abhängigkeitsrestriktionen definiert. Überdies können bestehende Feature- und Plugin-Gruppen verfeinert oder neue, normalerweise kleinere, Feature- und Plugin-Gruppen eingeführt werden. Liegen

Abhängigkeiten vor, für die nicht eindeutig feststeht, ob sie erlaubt oder verboten werden sollten, können diese Abhängigkeiten gegebenenfalls als tolerierte Abhängigkeiten definiert werden. Alternativ können tolerierte Abhängigkeiten eingeführt werden, wenn die Abhängigkeiten zum jeweiligen Zeitpunkt noch akzeptiert werden, aber ausgedrückt werden soll, dass die Abhängigkeiten zukünftig unter Umständen als verboten erachtet werden.

4. Das Abhängigkeitsmodell sollte darauffolgend in regelmäßigen Intervallen geprüft und bewertet werden, zumindest falls erlaubte Abhängigkeiten zum Abhängigkeitsmodell hinzugefügt wurden. Durch regelmäßige Prüfungen kann verhindert werden, dass ausgewählte Abhängigkeiten von einer einzelnen Person lediglich als erlaubt modelliert werden, um Fehlermeldungen zu vermeiden.

Zusätzlich zu dem zuvor skizzierten Anwendungsszenario ist es möglich, den Ansatz ausgehend von dem Ist-Modell der Abhängigkeiten eines Plugin-basierten Softwaresystems anzuwenden. Die Werkzeuginfrastruktur des Ansatzes erlaubt es, das Ist-Modell eines konkreten Plugin-basierten Softwaresystems als Abhängigkeitsmodell zu exportieren. In diesem exportierten Abhängigkeitsmodell sind alle vorhandenen Abhängigkeiten als erlaubt und alle anderen Abhängigkeiten als verboten deklariert. Verstöße gegen das Abhängigkeitsmodell können in diesem Szenario nur durch das Hinzufügen neuer Abhängigkeiten auftreten. Das exportierte Abhängigkeitsmodell muss aber nicht zwangsläufig unverändert als Referenzarchitektur verwendet werden. Stattdessen können die Systemarchitekten ausgehend von dem Ist-Modell die erforderlichen Änderungen vornehmen. Dieses Szenario bietet sich im Wesentlichen dann an, wenn eine Version eines Softwaresystems vorliegt, dessen Architektur sauber ist, z.B. weil zuvor Umbauarbeiten an dem Softwaresystem vorgenommen wurden und das Ist-Modell einer Referenzarchitektur sehr nahe kommt.

8.7 Diskussion

8.7.1 Allgemeine Diskussion

Die im Rahmen dieser Arbeit entwickelte Implementierung der Prüfkompone nte kann unmittelbar nur für Eclipse-basierte Softwaresysteme eingesetzt werden, da die Prüfkompone nte als Eclipse-Plugin umgesetzt ist und Eclipse-Abhängigkeiten untersucht. Dessen ungeachtet können die zugrundeliegenden Konzepte analog auf Plugin-basierte Softwaresysteme angewendet werden, die mit einer anderen Werkzeuginfrastruktur entwickelt werden. In diesem Fall muss die Logik zur Extraktion der Abhängigkeiten zwischen Plugins adaptiert werden. Unter der Voraussetzung, dass die Prüfkompone nte weiterhin auf Änderungen in der Entwicklungsumgebung reagieren können soll, muss zudem der Listener-Mechanismus passend für die entsprechende Werkzeuginfrastruktur implementiert werden. Demgegenüber kann DepCoL auch zur Spezifikation von Abhängigkeitsmodellen für andersartige Plugin-basierte Softwaresysteme eingesetzt werden, da dort ebenfalls Aussagen über die Abhängigkeiten zwischen Plugins getroffen werden müssen.

Weiterhin gilt es bei dem Einsatz von DepCoL zu beachten, dass lediglich Aussagen über Abhängigkeiten auf der Ebene von Features, Plugins oder Feature- und Plugin-Gruppen gemacht werden können. Es ist nicht möglich feingranulare Abhängigkeiten auf der Ebene von Klassen oder Methoden zu spezifizieren. Für diesen Fall müssen andere existierende Ansätze eingesetzt werden (siehe Abschnitt 8.8). Der in dieser Arbeit vorgestellte Ansatz ist hingegen ein ergänzender Ansatz, der eingesetzt werden kann, um Abhängigkeiten auf der Ebene von Features und Plugins einzuschränken.

Ein Problem bei großen und komplexen Abhängigkeitsmodellen ist, dass es schwierig sein kann, das Abhängigkeitsmodell zu verstehen und weiterzuentwickeln. Dieses Problem kann etwas abgeschwächt werden, indem das Abhängigkeitsmodell in mehrere Teilmodelle aufgeteilt wird. Auf diese Weise können Abhängigkeitsrestriktionen z.B. für verschiedene Systembereiche in verschiedenen Teilmodellen definiert werden. Darüber hinaus können beispielsweise die Feature- und die Plugin-Basis in eine separate Datei ausgelagert werden. Falls eine solche Aufteilung auf mehrere Dateien durchgeführt wird, muss allerdings spezifiziert werden, in welcher Reihenfolge die Teilmodelle verarbeitet werden sollen. Eine Verarbeitung in zufälliger Reihenfolge kann zu Fehlern in der Analyse der Abhängigkeitsmodelle führen, da unterschiedliche Reihenfolgen Auswirkungen auf die Verarbeitung von überschriebenen Abhängigkeitsrestriktionen haben.

Ein mögliches Problem bei sämtlichen Ansätzen zur Prüfung der Architekturkonformität ist, dass die Referenzarchitektur unter Umständen bewusst angepasst wird, um Fehlermeldungen zu unterdrücken. Übertragen auf den Ansatz dieser Arbeit bedeutet das, dass bewusst ausgewählte Abhängigkeiten erlaubt werden, um zu verhindern, dass eine Fehlermeldung gemeldet wird. Ein Mittel um diesem Phänomen zumindest zum Teil entgegenzuwirken ist, eine solche Abhängigkeit als toleriert zu markieren. Auf diese Weise wird ausgedrückt, dass eine Abhängigkeit temporär akzeptiert wird, dass sie zukünftig aber verboten sein kann. Darüber hinaus sollten regelmäßige Reviews des Abhängigkeitsmodells im Entwicklungsprozess verankert sein. Durch diese steigt die Wahrscheinlichkeit, dass auffällt, falls einzelne Architekten die Referenzarchitektur lediglich aus Gründen der Vermeidung von Fehlermeldungen angepasst haben. Ferner können in solchen Reviewrunden notwendige Anpassungen der Referenzarchitektur beschlossen werden.

Bei der Anwendung des Ansatzes ist zudem zu beachten, dass nicht proaktiv verhindert wird, dass verbotene Abhängigkeiten in ein Softwaresystem eingeführt werden. Der Ansatz informiert lediglich nach den Änderungen darüber, ob verbotene Abhängigkeiten eingeführt wurden. Eine Möglichkeit das zu verhindern, besteht in der Umsetzung eines Hooks in dem verwendeten Versionsverwaltungssystem. Ein solcher Hook kann vor dem Einchecken der Manifest-Dateien sicherstellen, dass die Abhängigkeiten der Plugins konform zu dem Abhängigkeitsmodell sind. Eine weniger restriktive Herangehensweise wäre es den Eclipse-Dialog zum Hinzufügen von Abhängigkeiten zu erweitern, so dass nur Abhängigkeiten hinzugefügt werden können, die erlaubt sind. In manchen Situationen liegt die höchste Priorität jedoch darauf eine spezifische Funktionalität möglichst schnell umzusetzen, selbst wenn diese gegen Abhängigkeitsrestriktionen verstößt. Dies gilt vor allem in Industrieprojekten mit festen zeitlichen Vorgaben. Aus diesem Grund wurden die beiden genannten Varianten zum Verhindern der Einführung von verbotenen Abhängigkeiten nicht umgesetzt.

Wie in Abschnitt 8.3.3 dargelegt, werden standardmäßig alle Abhängigkeiten zwischen den Plugins und Features erlaubt. Ein Vorteil davon ist, dass dadurch die schrittweise Integration des Ansatzes in den Entwicklungsprozess vereinfacht wird, wie im nächsten Abschnitt noch genauer dargelegt wird. Systemarchitekten können dieses Standardverhalten jedoch auf einfache Weise anpassen, wie in Listing 8.12 verdeutlicht. So können sie zu Beginn eines Abhängigkeitsmodells eine Plugin-Gruppe deklarieren, die sämtliche Plugins umfasst. Daraufhin können sie für diese Plugin-Gruppe eine Abhängigkeitsrestriktion definieren, wonach Abhängigkeiten zu sämtlichen Plugins verboten oder toleriert sind. Darauffolgend können sie die restlichen Abhängigkeitsrestriktionen wie üblich modellieren. Die Folge davon ist, dass die restlichen Abhängigkeitsrestriktionen die erste Abhängigkeitsrestriktion überschreiben. Liegt hingegen ein Plugin-Paar vor, für das keine weitere Abhängigkeitsrestriktion definiert ist, greift die erste Abhängigkeitsrestriktion. Auf diese Weise kann die Standardvorgehensweise zur Behandlung von Abhängigkeiten leicht adaptiert werden.

```
1 declare plugingroup ALLPLUGINS {  
2   regex *;  
3 }  
4  
5 plugingroup ALLPLUGINS {  
6   forbid dependency to pluginpattern *;  
7 }  
8  
9 //Further dependency restrictions
```

DepCoL

Listing 8.12: Abhängigkeitsmodell durch das sämtliche Abhängigkeiten standardmäßig verboten werden.

8.7.2 Diskussion zur Anwendbarkeit des Ansatzes in produktiv eingesetzten Softwaresystemen

Die entwickelte Prüfkomponekte kann nahtlos in die Entwicklungsumgebung integriert werden, in der das jeweilige Softwaresystem implementiert wird. Dieser Faktor erleichtert die Integration des Ansatzes in den Softwareentwicklungsprozess eines produktiv eingesetzten Softwaresystems wesentlich. Auf diese Weise kann die Prüfung auf die Verletzung von Abhängigkeitsrestriktionen nach jeder Änderung der Entwickler automatisch durchgeführt und die Entwickler unmittelbar auf Verstöße hingewiesen werden. Für die Entwickler entsteht dadurch kein Mehraufwand. Zusätzlich ermöglicht die Integration in ein CI-System die Prüfung zu festdefinierten Zeitpunkten, unabhängig von der Integration in eine Entwicklungsumgebung.

Aus Sicht der Systemarchitekten wird die Integration des Ansatzes in den Softwareentwicklungsprozess darüber hinaus dadurch stark vereinfacht, dass standardmäßig alle Abhängigkeiten zwischen den Plugins und Features als erlaubt betrachtet werden. Würden sämtliche Abhängigkeiten initial als verboten erachtet werden, müssten die Systemarchitekten alle erlaubten Abhängigkeiten explizit definieren. Dies kann gerade bei komplexen Softwaresystemen sehr aufwändig sein. Würden die Systemarchitekten hingegen nur einen Teil der erlaubten Abhängigkeiten definieren, würden die Entwickler zu Beginn unter Umständen sehr viele, zum Teil unnötige,

Fehlermeldungen erhalten. Letzteres könnte allerdings zu Akzeptanzproblemen des Ansatzes unter den Entwicklern führen. Dadurch, dass sämtliche Abhängigkeiten standardmäßig erlaubt sind, können die Systemarchitekten hingegen für die gewünschten Systemteile schrittweise Abhängigkeitsrestriktionen hinzufügen und das Abhängigkeitsmodell iterativ aufbauen, ohne dass die Entwickler eine Vielzahl unnötiger Fehlermeldungen erhalten.

Überdies kann auch das Überschreiben von Abhängigkeitsrestriktionen die Nutzung des Ansatzes für Systemarchitekten vereinfachen. Dies ermöglicht es Systemarchitekten, zunächst eine allgemeine Grobrichtlinie für die Abhängigkeiten zu entwerfen und diese anschließend für spezifische Plugins und Features anzupassen. Dies kann in einer Reduktion des Modellierungsaufwands resultieren, da andernfalls unter Umständen eine höhere Anzahl von einzelnen Abhängigkeitsrestriktionen definiert werden muss, um die verschiedenen Konstellationen passend zu beschreiben.

Ferner kann die Möglichkeit zur Definition von tolerierten Abhängigkeiten in produktiv eingesetzten Softwaresystemen hilfreich sein. Über diese können Abhängigkeiten markiert werden, die zwar nicht verboten sind, die aber unter Umständen nicht bestehen sollten. Weiterhin kann darüber ausgedrückt werden, dass eine Abhängigkeit zum aktuellen Zeitpunkt temporär toleriert wird, zukünftig aber aus dem System entfernt werden sollte. Letztgenannter Fall wurde in der Fallstudie beobachtet, die in Abschnitt 8.9 vorgestellt wird.

8.8 Verwandte Arbeiten

In diesem Abschnitt werden verwandte Arbeiten aus dem Umfeld der statischen Prüfung der Architekturkonformität vorgestellt. Der wichtigste Unterschied zwischen dem Ansatz dieser Arbeit und den existierenden Ansätzen ist, dass der Ansatz aus dieser Arbeit dediziert auf Plugin-basierte Softwaresysteme zugeschnitten ist. Auf diese Weise können Abhängigkeiten auf der Ebene von Features und Plugins eingeschränkt werden. Demgegenüber zielen die meisten existierenden Ansätze auf feingranulare Abhängigkeiten zwischen Quellcode-Entitäten ab.

In [TV08, TV09] wird die statische, deklarative Sprache DCL (Dependency Constraint Language) eingeführt, welche es ermöglicht Restriktionen zu Abhängigkeiten in einem Softwaresystem anzugeben. In DCL können Restriktionen zwischen Modulen spezifiziert werden, die jeweils eine Menge von Klassen repräsentieren. Die Festlegung, welche Klassen in welchem Modul enthalten sind, erfolgt über eine explizite Festlegung der vollqualifizierten Namen der Klassen oder über die Nutzung von regulären Ausdrücken. Aufbauend auf der Definition der Module können in DCL die Restriktionen formuliert werden, z.B. dass nur Klassen aus Modul A auf Klassen aus Modul B zugreifen können. Die Definition von Feature- und Plugin-Gruppen in dieser Arbeit ist durch die Definition von Modulen in [TV08, TV09] motiviert.

Die Arbeit in [MNS95, MN97, MNS01] führt Software-Reflexionsmodelle zur Prüfung der Architekturkonformität ein. Software-Reflexionsmodelle sehen vor, dass Entwickler ein abstraktes Referenzmodell definieren, welches sich aus Entitäten und Beziehungen zwischen diesen Entitäten zusammensetzt. Darüber hinaus müssen Entwickler ein Quellmodell aus dem Quellcode extrahieren, z.B. einen Graphen, der die Aufrufbeziehungen zwischen Klassen im Quellcode widerspiegelt. Abschließend muss ein Mapping zwischen dem Referenzmodell und dem Quellmodell durchgeführt werden. In diesem Mapping müssen Entitäten aus dem Referenz-

modell Elementen aus dem Quellmodell zugeordnet werden. Basierend auf diesem Mapping, dem Referenzmodell und dem Quellmodell wird schließlich das Software-Reflexionsmodell berechnet. Dieses gibt Aufschluss darüber, wo das Referenzmodell (nicht) mit dem Quellmodell übereinstimmt. Falls das Referenzmodell Entitäten und erlaubte oder verbotene Abhängigkeiten zwischen den Entitäten enthält, können Software-Reflexionsmodelle dazu eingesetzt werden Architektur-Erosion zu erkennen. Eine Gemeinsamkeit von Software-Reflexionsmodellen und dem Ansatz aus dieser Arbeit besteht in der Notwendigkeit der Definition eines Referenzmodells. Ein in DepCoL modelliertes Abhängigkeitsmodell stellt in dieser Arbeit ein Referenzmodell dar. Die Deklaration von Feature- und Plugin-Gruppen ist zudem vergleichbar zu der Bereitstellung eines Mappings. Da allerdings keine Feature- und Plugin-Gruppen definiert werden müssen, ist die Definition eines Mappings in dieser Arbeit optional. Im Gegensatz zu Software-Reflexionsmodellen müssen Systemarchitekten bei der Nutzung von DepCoL kein Quellmodell extrahieren. Stattdessen werden die bestehenden Abhängigkeiten zwischen den Plugins, die zu dem Quellmodell korrespondieren, automatisch durch die Prüfkomponente ermittelt. Darüber hinaus ist es in DepCoL auf einfache Weise möglich, Abhängigkeitsrestriktionen zu überschreiben und zwischen erlaubten, tolerierten und verbotenen Abhängigkeiten zu unterscheiden.

Eine weitere Technik, die auf der Extraktion von Abhängigkeiten aus dem Quellcode mittels statischer Code-Analyse basiert, wird in [SJSJ05] vorgestellt. Die extrahierten Abhängigkeiten werden in tabellarischer Form dargestellt - dependency structure matrix (DSM) genannt. Über Design-Regeln kann definiert werden, welche Arten von Abhängigkeiten in der DSM erlaubt oder verboten sind. Angewandt auf eine konkrete DSM können die Design-Regeln die Abhängigkeiten identifizieren, die gegen die Referenzarchitektur verstoßen. Da Design-Regeln standardmäßig vererbt werden, gelten alle Restriktionen, die für ein Subsystem definiert wurden, implizit für alle Elemente des Subsystems. Überdies können Design-Regeln für spezifische Elemente der DSM überschrieben werden. Die Idee für das Überschreiben von Abhängigkeitsrestriktionen in DepCoL ist durch das Überschreiben von Design-Regeln motiviert. Darüber hinaus werden Abhängigkeitsrestriktionen ebenfalls vererbt, d.h. falls eine Abhängigkeitsrestriktion für eine Feature-Gruppe definiert ist, erben standardmäßig alle Plugins dieser Feature-Gruppe diese Abhängigkeitsrestriktion. Zusätzlich dazu können in DepCoL allerdings tolerierte Abhängigkeiten spezifiziert werden.

In [VGG08] wird als Alternative zu den bisherigen Ansätzen die Verwendung von Query-Sprachen zur Erkennung von Architektur-Erosion vorgeschlagen. Queries können auf dem Quellcode oder anderen relevanten Artefakten ausgeführt werden, z.B. um alle Klassen zu identifizieren, die von bestimmten anderen Klassen oder Packages abhängen [VEBHH11]. Die Architekturkonformität kann folglich geprüft werden, indem Queries formuliert werden, die verbotene Abhängigkeiten auffinden.

Basierend auf dieser Idee wird in [DDHR09] ein regelbasierter Ansatz zur Prüfung der Architekturkonformität vorgestellt. Bei diesem Ansatz werden die vorgesehene Softwarearchitektur, das Design bzw. die Implementierung des Softwaresystems sowie das Mapping dazwischen über formale Notationen repräsentiert. Jeder dieser Teile wird in eine logische Wissensbasis überführt, die Fakten enthält, welche die Elemente der Beschreibungen repräsentieren. Die Architekturkonformität wird dabei durch die Ausführung von Architekturregeln auf der Vereinigung der Wissensbasen durchgeführt. Dabei werden die Architekturregeln als Queries ausgeführt, deren

Ergebnisse ausgewertet werden. Eine Besonderheit der Architekturregeln ist, dass sie nicht nur Verstöße melden können, sondern auch die Elemente erkennen und melden, die zu den Verstößen führen. Das zugehörige prototypische Werkzeug ist als Eclipse-Plugin verfügbar, welches unter anderem in der Lage ist Fakten automatisch zu generieren, z.B. aus einem gegebenen UML-Modell. In [HR13] wird diese Idee noch verfeinert. In dieser Arbeit sind Architekturregeln als logische Formeln über einer gemeinsamen und erweiterbaren Ontologie definiert und Modelle werden auf Instanzen dieser Ontologie abgebildet. Zur Repräsentation von Modellen in dieser Form müssen Transformationen definiert werden, die ausdrücken, wie Modellelemente konkret dargestellt werden können. Auf diese Weise können Modelle zur Definition von Architekturregeln verwendet werden. Infolgedessen müssen lediglich die Entwickler der Transformationen Kenntnisse über logische Programmierung besitzen. Im Gegensatz zu diesen Arbeiten erfordert der Ansatz in dieser Arbeit keinerlei Kenntnisse über logische Programmierung. Auf der anderen Seite ist der Ansatz weniger flexibel als die beiden genannten Ansätze. Diese Flexibilität wird jedoch in dem Anwendungskontext nicht benötigt. Der Fokus auf Plugin-basierte Softwaresysteme erlaubt es hingegen den Softwarearchitekten direkt die Begrifflichkeiten wiederzuverwenden, mit denen sie bereits vertraut sind.

Weitere Ansätze und Werkzeuge zur (statischen) Überprüfung der Architekturkonformität werden in [PTV⁺10] und [KP07] vorgestellt. Einen umfassenden Überblick über Architektur-Erosion und Ansätze zur Minimierung, Prävention und Reparatur von Architektur-Erosion gibt die Arbeit [SB12].

Abhängigkeiten zwischen Plugins lassen sich normalerweise darauf zurückführen, dass Klassen dieser Plugins auf irgendeine Weise voneinander abhängen. Eine Alternative zu der in Abschnitt 8.5 beschriebenen Implementierung der Prüfkomponente ist es daher ein Abhängigkeitsmodell in ein Modell zu transformieren, welches von einem der zuvor beschriebenen bestehenden Klassen-basierten Ansätzen verarbeitet werden kann. Der Vorteil dieser Vorgehensweise ist, dass bestehende Werkzeuge direkt wiederverwendet werden können. Dieses Szenario hat jedoch auch einige Nachteile. Ein Problem in diesem Kontext ist, dass es Abhängigkeiten zwischen Plugins gibt, die sich nicht auf Abhängigkeiten zwischen Klassen abbilden lassen. Ein Klassen-basierter Ansatz kann solche Abhängigkeiten nicht als verboten identifizieren. Dieses Problem tritt unter anderem dann auf, wenn eine Manifest-Datei nicht verwendete Abhängigkeiten enthält. Folglich müsste sichergestellt werden, dass keine Manifest-Datei nicht verwendete Abhängigkeiten enthält. Ein weiteres Problem ist, dass mehrere Plugins eine Klasse mit dem gleichen qualifizierten Namen enthalten können. Ein Klassen-basierter Ansatz kann nicht direkt zwischen diesen Klassen unterscheiden. Diese Probleme und die Anforderungen bezüglich der Spezifikation von tolerierten Abhängigkeiten, der Möglichkeit Abhängigkeitsrestriktionen überschreiben zu können und die Eclipse-Integration führten zu der Entscheidung eine eigenständige Prüfkomponente zu implementieren.

8.9 Fallstudie in der Industrie

8.9.1 Ziele und Forschungsfragen

Das Ziel dieser Fallstudie ist es zu validieren, dass der vorgestellte Ansatz in einem realen Plugin-basierten Softwaresystem genutzt werden kann, um Entwickler dabei zu unterstützen, die Architekturkonformität des Softwaresystems sicherzustellen. Zu diesem Zweck wurden die folgenden Forschungsfragen abgeleitet:

- **RQ1:** Ist die entwickelte DSL DepCoL ausdrucksstark genug, um Abhängigkeitsrestriktionen für ein reales Plugin-basiertes Softwaresystem zu definieren, welches sich stetig weiterentwickelt?
- **RQ2:** Führen die Entwickler auf Basis erkannter Verstöße Änderungen durch, um die Architekturkonformität zu erhöhen?
- **RQ3:** Finden die Entwickler den Ansatz nützlich, um die Architekturkonformität sicherzustellen bzw. zu erhöhen?

Diese Forschungsfragen sind durch Forschungsfragen aus anderen Fallstudien im Kontext der Architekturkonformitätsprüfung motiviert. So ist die erste Forschungsfrage aus [ACN02], die zweite aus [RGBB08] und die dritte aus [ARB12] motiviert.

8.9.2 Aufbau und Gegenstand der Studie

Der Ansatz wird in dieser Studie auf die Features und Plugins von Authoring, Automation, RTS und WTS angewendet. Ein Überblick über diese Softwaresysteme wurde einleitend in Abschnitt 2.1 gegeben. Einige Features und Plugins dieser Softwaresysteme werden von mehreren der genannten Softwaresysteme gemeinsam genutzt, während manche Features und Plugins spezifisch für die jeweiligen Softwaresysteme sind.

Zur Beantwortung der ersten Forschungsfrage wurde in Treffen mit Systemarchitekten ein Abhängigkeitsmodell für die betrachteten Softwaresysteme entwickelt. In diesen Treffen wurde das Abhängigkeitsmodell mittels DepCoL iterativ entwickelt, bis es sämtliche Vorgaben der Systemarchitekten umfasste. Ferner wurde in diesen Treffen protokolliert, ob es Vorgaben gab, die sich nicht oder nur umständlich mit DepCoL umsetzen ließen. Dieses Protokoll wurde abschließend untersucht, um Unzulänglichkeiten von DepCoL zu identifizieren.

Zur Beantwortung der zweiten Forschungsfrage wurde die Konformitätsprüfung für jedes Plugin initiiert, dessen Abhängigkeiten durch einen Commit eines Entwicklers verändert wurden. Zu Beginn der Fallstudie wurde die Konformitätsprüfung allerdings nur angewendet, ohne die Entwickler und Systemarchitekten über die Ergebnisse zu informieren. Auf diese Weise wurde ermittelt, wie viele neue Verstöße ohne den Ansatz hinzugefügt wurden und wie viele Verstöße ohne den Ansatz beseitigt wurden. Diese Beobachtungsphase lief insgesamt sechs Wochen. Anschließend lief die Fallstudie weitere 12 Wochen in der Anwendungsphase, in der die Entwickler und Systemarchitekten über die Verstöße informiert wurden. Im Zuge der Ermittlungen der Verstöße gegen das Abhängigkeitsmodell wurde das jeweils aktuelle Abhängigkeitsmodell gegen

die aktuelle Implementierung ausgeführt. Neue Verstöße können allerdings nicht nur durch Änderungen in der Implementierung auftreten, sondern auch durch eine Änderung des Abhängigkeitsmodells. Aus diesem Grund wurde begleitend zur Fallstudie nicht nur die Gesamtzahl der Verstöße festgehalten, sondern auch untersucht, wie viele Verstöße aufgrund von Änderungen im Abhängigkeitsmodell hinzukamen oder entfallen sind.

Die dritte Forschungsfrage wurde beantwortet, indem am Ende der Fallstudie Fragebögen an 26 Entwickler verteilt wurden. Diese Fragebögen umfassten die nachfolgend aufgeführten Fragen. Zudem konnten die Entwickler einen Freitext als weiteres Feedback niederschreiben.

- **DQ1:** Wie bewerten Sie die aktuelle Form der Benachrichtigung über Verstöße gegen Abhängigkeitsrestriktionen?
- **DQ2:** Wie hilfreich finden Sie die unmittelbare Prüfung einer Manifest-Datei nach dem Speichern einer Manifest-Datei und die unmittelbare Rückmeldung zu Verstößen?
- **DQ3:** Haben Sie aufgrund der Rückmeldung durch die Prüfkomponeente eine Änderung nicht eingeecheckt, da die Prüfkomponeente eine Fehlermeldung bzw. eine Warnung diesbezüglich gemeldet hat? Falls ja: Wie häufig ist das vorgekommen (circa) und was haben Sie daraufhin unternommen?

Zu den ersten beiden Fragen konnten die Entwickler sowohl einen Freitext eingeben als auch eine Bewertung auf einer Skala angeben. Die möglichen Auswahloptionen dieser Skala sind nachfolgend dargelegt:

- ++: Sehr gut/Sehr hilfreich
- +: Gut/Hilfreich
- -: Weniger gut/Weniger hilfreich
- --: Sehr schlecht/Gar nicht hilfreich

Die vollständigen Antworten der Entwickler zu den Fragen sind in Abschnitt G.2 aufgeführt.

8.9.3 Ergebnisse und Diskussion

Forschungsfrage RQ1

RQ1: Ist die entwickelte DSL DepCoL ausdrucksstark genug, um Abhängigkeitsrestriktionen für ein reales Plugin-basiertes Softwaresystem zu definieren, welches sich stetig weiterentwickelt?

Im Zuge der Fallstudie wurden insgesamt fünf Versionen des Abhängigkeitsmodells entwickelt. Die ersten vier Versionen davon wurden in der Beobachtungsphase und die letzte Version in der Anwendungsphase der Fallstudie entwickelt. Tabelle 8.19 gibt einen Überblick über die

Komplexität dieser fünf Versionen. Die erste Version des Abhängigkeitsmodells fokussierte sich auf die elementaren Feature-Gruppen, die für die betrachteten Softwaresysteme relevant sind. Ferner wurden in der ersten Version ausschließlich Abhängigkeitsrestriktionen für diese Feature-Gruppen definiert, um die wesentlichen Architekturvorgaben auszudrücken. In den darauffolgenden Versionen wurden weitere kleinere Feature-Gruppen hinzugefügt und die Struktur der Feature-Gruppen überarbeitet. Die grundlegende Struktur der initial definierten Feature-Gruppen blieb jedoch in sämtlichen Versionen erhalten. Darüber hinaus wurden in den weiteren Versionen Abhängigkeitsrestriktionen hinzugefügt, die ebenfalls einzelne Features sowie kleinere Feature-Gruppen betroffen haben. Erst die letzte entwickelte Version des Abhängigkeitsmodells umfasste zusätzlich eine Abhängigkeitsrestriktion hin zu einer Plugin-Gruppe. Wie in Tabelle 8.19 zu sehen, wurden insbesondere von der ersten auf die zweite Version des Abhängigkeitsmodells zahlreiche Abhängigkeitsrestriktionen hinzugefügt. Der Hauptgrund hierfür ist, dass auf Basis der ersten Version umfangreiche Diskussionen geführt wurden, so dass bereits für die zweite Version viele benötigte Abhängigkeitsrestriktionen hinzugefügt werden konnten. Insgesamt konnte jede Architekturvorgabe der Systemarchitekten problemlos durch DepCoL ausgedrückt werden.

	V1	V2	V3	V4	V5
Anzahl deklarierter Feature-Gruppen	10	13	14	15	15
Anzahl deklarierter Plugin-Gruppen	0	0	0	0	1
Anzahl Abhängigkeitsrestriktionen	14	40	44	44	46
Anzahl Abhängigkeitsrestriktionen hin zu Feature-Gruppen	14	36	40	40	40
Anzahl Abhängigkeitsrestriktionen hin zu Plugin-Gruppen	0	0	0	0	1
Anzahl Abhängigkeitsrestriktionen hin zu Features	0	4	4	4	5
Anzahl Abhängigkeitsrestriktionen hin zu Plugins	0	0	0	0	0
Anzahl erlaubter Abhängigkeitsrestriktionen	0	2	4	4	5
Anzahl tolerierter Abhängigkeitsrestriktionen	0	11	12	12	13
Anzahl verbotener Abhängigkeitsrestriktionen	14	27	28	28	28
Anzahl überschreibender Abhängigkeitsrestriktionen	0	4	7	7	8

Tabelle 8.19: Überblick über die Komplexität der verschiedenen Versionen des Abhängigkeitsmodells.

Die Möglichkeit zur Deklaration von Feature-Gruppen stellte sich bei der Definition des Abhängigkeitsmodells als besonders hilfreich heraus, da die meisten Architekturvorgaben auf dieser Ebene gemacht wurden, bevor die Systemarchitekten die zur Verfügung stehenden Sprachmittel in DepCoL überhaupt kannten. So betreffen in der fünften Version des Abhängigkeitsmodells circa 90 Prozent der Abhängigkeitsrestriktionen Abhängigkeiten hin zu Feature-Gruppen, wie in Tabelle 8.19 zu sehen. Demgegenüber wurde lediglich eine Abhängigkeitsrestriktion hin zu einer Plugin-Gruppe und keine hin zu einem einzelnen Plugin definiert. Der Hauptgrund hierfür ist, dass der Fokus der Abhängigkeitsmodelle auf der groben Architektur liegen sollte und noch nicht jedes Detail der Architektur genau abdecken sollte.

Weiterhin zeigte sich, dass die Unterscheidung zwischen tolerierten und verbotenen Abhängigkeiten von den Systemarchitekten als sinnvoll erachtet wurde. Wie in Tabelle 8.19 zusam-

mengefasst, betreffen fast 30 Prozent der Abhängigkeitsrestriktionen in der fünften Version des Abhängigkeitsmodells tolerierte Abhängigkeiten, nur circa 10 Prozent erlaubte Abhängigkeiten und die restlichen 60 Prozent verbotene Abhängigkeiten. Der Anteil der explizit erlaubten Abhängigkeiten ist relativ gering, da standardmäßig jede Abhängigkeit erlaubt ist. Dessen ungeachtet wurden einige Abhängigkeitsrestriktionen eingefügt, die zuvor definierte Abhängigkeitsrestriktionen überschreiben, um ausgewählte Abhängigkeiten zu erlauben.

Insgesamt liegt der Anteil der Abhängigkeitsrestriktionen, die eine andere Abhängigkeitsrestriktion überschreiben, bei 17 Prozent. In diesen Fällen zeigte sich, dass das Überschreiben von Abhängigkeitsrestriktionen die praktische Nutzung des Ansatzes deutlich vereinfachen kann. Der Grund dafür ist, dass dadurch der Modellierungsaufwand reduziert werden kann. Konkret konnten im Abhängigkeitsmodell Abhängigkeiten von Feature-Gruppen zu anderen Feature-Gruppen verboten werden und anschließend konnte definiert werden, dass ausgewählte Sonderfälle anders gehandhabt werden sollen. Dies machte es überflüssig alle verbotenen Abhängigkeiten explizit auflisten zu müssen.

Um den praktischen Einsatz der Prüfkomponekte und von DepCoL weiter zu vereinfachen, wurde im Zuge der Fallstudie das bestehende Tooling erweitert. Konkret wurde das Tooling um eine Analyse ergänzt, die prüft, ob neue Features und Plugins zu den Softwaresystemen hinzugefügt wurden, die von dem bisherigen Abhängigkeitsmodell nicht beachtet wurden. Diese Analyse kann automatisch ausgeführt werden, so dass die Systemarchitekten, falls erforderlich, das Abhängigkeitsmodell anpassen können.

Als Fazit lässt sich festhalten, dass DepCoL sehr gut dafür geeignet war, die Architekturvorgaben der Systemarchitekten umzusetzen. Während der Fallstudie wurden einzelne neue Features in die Systeme eingeführt und die bestehenden Features und Plugins teilweise umstrukturiert. Diese Änderungen der Softwaresysteme konnten problemlos in das Abhängigkeitsmodell aufgenommen werden. So wurden die neuen Features beispielsweise den bestehenden Feature-Gruppen zugeordnet, so dass die Prüfkomponekte deren Abhängigkeiten ebenfalls prüfen konnte. Die weiteren Umstrukturierungen der Softwaresysteme konnten ebenfalls durch Anpassungen der Feature-Gruppen abgebildet werden.

Forschungsfrage RQ2

RQ2: Führen die Entwickler auf Basis erkannter Verstöße Änderungen durch, um die Architekturkonformität zu erhöhen?

Im Folgenden werden als Erstes die Ergebnisse der Beobachtungsphase vorgestellt, in der die Ergebnisse der Konformitätsprüfung ermittelt, aber nicht an die Entwickler und Systemarchitekten gemeldet wurden. Tabelle 8.20 fasst die Ergebnisse zusammen. Die Bedeutung der verschiedenen Werte in der Tabelle ist nachstehend erläutert, wobei die genannten Tage Wochentage sind und keine Arbeitstage.

- T1 bis T5 stellen fünf verschiedene Zeitpunkte in der Beobachtungsphase dar:

- T1: Zeitpunkt zu dem die erste Version des Abhängigkeitsmodells fertiggestellt wurde. T1 markiert den Beginn der Beobachtungsphase.
 - T2: Zeitpunkt zu dem die zweite Version des Abhängigkeitsmodells fertiggestellt wurde. Diese zweite Version lag circa drei Wochen später als T1 vor.
 - T3: Zeitpunkt zu dem die dritte Version des Abhängigkeitsmodells fertiggestellt wurde. Diese lag circa eine Woche nach T2 vor.
 - T4: Zeitpunkt zu dem die vierte Version des Abhängigkeitsmodells fertiggestellt wurde. Diese lag etwas weniger als eine Woche später als T3 vor.
 - T5: Zeitpunkt eines Commits, der zu Änderungen in den gemeldeten Verstößen geführt hat. T5 markiert das Ende der sechswöchigen Beobachtungsphase, die somit nach etwas mehr als einer Woche nach T4 beendet war.
- W verweist auf die Gesamtanzahl der zu diesem Zeitpunkt vorliegenden Warnungen.
 - Δ_{WA} verweist auf die Anzahl der Warnungen, die im Vergleich zur vorherigen Version aufgrund von Änderungen des Abhängigkeitsmodells hinzukamen oder entfernt wurden.
 - Δ_{WI} verweist auf die Anzahl der Warnungen, die im Vergleich zur vorherigen Version aufgrund von Änderungen der Implementierung hinzukamen oder entfernt wurden.
 - F verweist auf die Gesamtanzahl der zu diesem Zeitpunkt vorliegenden Fehler.
 - Δ_{FA} verweist auf die Anzahl der Fehler, die im Vergleich zur vorherigen Version aufgrund von Änderungen des Abhängigkeitsmodells hinzukamen oder entfernt wurden.
 - Δ_{FI} verweist auf die Anzahl der Fehler, die im Vergleich zur vorherigen Version aufgrund von Änderungen der Implementierung hinzukamen oder entfernt wurden.

	W	Δ_{WA}	Δ_{WI}	F	Δ_{FA}	Δ_{FI}
T1	0	0	0	0	0	0
T2	14	+14	0	8	+8	0
T3	16	+2	0	8	0	0
T4	16	0	0	8	0	0
T5	16	0	0	9	0	+1

Tabelle 8.20: Übersicht über die gemeldeten Verstöße in der Beobachtungsphase.

Wie in Tabelle 8.20 zu sehen, kam es während der Beobachtungsphase bis zum Zeitpunkt T4 ausschließlich zu neuen Verstößen aufgrund der Änderungen des Abhängigkeitsmodells. Sämtliche neuen Verstöße in diesem Zeitraum waren somit darin begründet, dass neue Abhängigkeitsrestriktionen hinzugefügt wurden, so dass weitere Verstöße identifiziert wurden. Erst zum Zeitpunkt T5 hat eine Änderung eines Entwicklers zu einem neuen Verstoß geführt. Diese Zahlen zeigen überdies, dass in der Beobachtungsphase kein Verstoß ohne den Ansatz behoben wurde. Insgesamt blieb die Anzahl der Verstöße aufgrund von Änderungen in der Implementierung folglich nahezu konstant.

In der anschließenden Anwendungsphase wurden die Verstöße an die Entwickler und Systemarchitekten gemeldet. Tabelle 8.21 zeigt die Ergebnisse dieser Anwendungsphase. In dieser Tabelle werden verschiedene Zeitpunkte referenziert. Nachfolgend wird vorab zunächst aufgeführt, welche Zeiträume zwischen den Zeitpunkten lagen, wobei die Anzahl der Tage wiederum Wochentage darstellen und keine Arbeitstage. Die ersten sechs Zeitpunkte stellen dabei Zeitpunkte von Commits dar, durch die Verstöße hinzukamen oder beseitigt wurden:

- T6 war vier Tage nach T5.
- T7 war sechs Tage nach T6.
- T8 war vier Tage nach T7.
- T9 war vier Tage nach T8.
- T10 war acht Tage nach T9.
- T11 war eine Woche nach T10.
- T12 war sechs Wochen nach T11. Dies entsprach dem Zeitpunkt, zu dem die fünfte Version des Abhängigkeitsmodells fertiggestellt wurde.
- T13 war zwei Wochen nach T12. Dies ist das Ende der Anwendungsphase.

	W	Δ_{WA}	Δ_{WI}	F	Δ_{FA}	Δ_{FI}
T6	16	0	0	8	0	-1
T7	0	0	0	5	0	-3
T8	17	0	+1	5	0	0
T9	16	0	-1	5	0	0
T10	16	0	0	0	0	-5
T11	2	0	-14	0	0	0
T12	2	0	0	0	0	0
T13	2	0	0	0	0	0

Tabelle 8.21: Übersicht über die gemeldeten Verstöße in der Anwendungsphase.

Die Tabelle zeigt, dass bereits zu Beginn der Anwendungsphase ein Fehler beseitigt wurde. Dabei handelte es sich um den Fehler, der in T5 in die Systeme eingeführt wurde. Eine genauere Analyse zeigte, dass es in diesem Bereich weitere bestehende Verstöße gab. Aus diesem Grund hat der Entwickler die Änderung nicht unmittelbar rückgängig gemacht, sondern erst weitere Schritte unternommen, um die weiteren Verstöße später einfacher entfernen zu können. Zum Zeitpunkt T7 wurden diese weiteren Verstöße schließlich ebenfalls beseitigt. Die Zeitpunkte T8 und T9 sind insofern interessant, als dass zum Zeitpunkt T9 der Verstoß, der durch T8 eingeführt wurde, wieder behoben wurde. Daraufhin wurden in T10 weitere Fehler behoben. Weiterhin wurden in T11 mehrere Warnungen beseitigt. Hierbei konnten zahlreiche Warnungen auf einmal beseitigt werden, da die beseitigten Warnungen den gleichen Themenkomplex betroffen haben

und dieser Bereich relativ unabhängig von den anderen Teilen der Softwaresysteme war. Aus der Einführung der fünften Version des Abhängigkeitsmodells folgten hingegen bis zum Ende der Fallstudie keine neuen Verstöße. Zudem wurden bis zum Ende der Fallstudie keine weiteren Verstöße beseitigt.

Am Ende der Anwendungsphase enthielt die Implementierung noch zwei Verstöße gegen das Abhängigkeitsmodell. In beiden Fällen handelte es sich um marginale Verstöße, die aufgrund von zeitlichen Restriktionen noch nicht behoben wurden.

Die geringe Anzahl an Verstößen lässt sich durch verschiedene Faktoren erklären. Ein wesentlicher Gesichtspunkt der hierbei betrachtet werden muss ist, dass der Fokus darauf lag ein Abhängigkeitsmodell zu entwickeln, welches vor allem zentrale Architekturvorgaben widerspiegelt. Das Ziel war es, aufgrund der Größe der Systeme, hingegen nicht, ein vollständiges Abhängigkeitsmodell zu entwickeln. Aus diesem Grund existieren unter Umständen noch weitere Verstöße, die in der aktuellen Version aber nicht beachtet wurden. Darüber hinaus wurden wenige Monate vor Beginn der Fallstudie bereits umfangreiche Restrukturierungen vorgenommen. Diese haben dazu geführt, dass es keine fundamentalen Abweichungen von der Architektur gegeben hat, die durch die entwickelten Abhängigkeitsmodelle identifiziert worden wären.

Als Fazit lässt sich festhalten, dass der Ansatz das Bewusstsein für die Verstöße geschärft hat und somit wesentlich dazu beigetragen hat, dass die Verstöße zu großen Teilen entfernt wurden. Die Verstöße, die zu den Zeitpunkten T5 und T8 eingeführt wurden, wurden nach kurzer Zeit wieder behoben. Konkret lagen zwischen der Einführung der Verstöße und dem Beheben der Verstöße jeweils nur zwei Arbeitstage.

Forschungsfrage RQ3

RQ3: Finden die Entwickler den Ansatz nützlich, um die Architekturkonformität sicherzustellen bzw. zu erhöhen?

- **DQ1:** Wie bewerten Sie die aktuelle Form der Benachrichtigung über Verstöße gegen Abhängigkeitsrestriktionen?
- **DQ2:** Wie hilfreich finden Sie die unmittelbare Prüfung einer Manifest-Datei nach dem Speichern einer Manifest-Datei und die unmittelbare Rückmeldung zu Verstößen?
- **DQ3:** Haben Sie aufgrund der Rückmeldung durch die Prüfkomponeente eine Änderung nicht eingeecheckt, da die Prüfkomponeente eine Fehlermeldung bzw. eine Warnung diesbezüglich gemeldet hat? Falls ja: Wie häufig ist das vorgekommen (circa) und was haben Sie daraufhin unternommen?

Von den 26 Entwicklern, an die ein Fragebogen verteilt wurde, beantworteten insgesamt zehn Entwickler den Fragebogen. Dies entspricht einer Rücklaufquote von circa 40 Prozent. Die vollständigen Antworten zu den Fragebögen sind in Abschnitt G.2 dargelegt. Nachfolgend werden

lediglich einige dieser Antworten vorgestellt. Der Fokus liegt darauf die grundsätzliche Rückmeldung der Entwickler zu den Fragen vorzustellen.

Tabelle 8.22 gibt einen Überblick über die Bewertung der Entwickler zu den Fragen DQ1 und DQ2. Wie dort zu sehen, haben sämtliche Entwickler die aktuelle Form der Benachrichtigung über Verstöße entweder als sehr hilfreich oder als hilfreich erachtet, abgesehen von einer Enthaltung. Ein ähnliches Ergebnis zeigt sich bei der Bewertung der unmittelbaren Ausführung der Prüfkomponente nach dem Ändern der Abhängigkeiten.

	++	+	-	--	Keine Angabe
DQ1	5	4	0	0	1
DQ2	6	4	0	0	0

Tabelle 8.22: Übersicht über die Bewertung der Entwickler zu den Fragen DQ1 und DQ2.

In dem Freitext zur ersten Frage gab ein Entwickler an, dass die Nutzung von Fehler-Markern hilfreich ist, da dies dem aus Eclipse bekannten Schema bei z.B. Syntaxfehlern entspricht.

Bei dem Freitext zur zweiten Frage hob ein Entwickler hervor, dass die unmittelbare Prüfung nach dem Speichern sinnvoll ist, da auf diese Weise verhindert werden kann, dass man viel Arbeit und Zeit in eine fehlerhafte Entwicklung steckt. Als Verbesserungsvorschlag merkte überdies ein Entwickler an: "Besonders hilfreich wäre es, wenn man schon vor dem Speichern des Manifests erkennen könnte, dass die aktuelle Änderung zu einem Verstoß führen wird." Dieser Aspekt wurde bereits in Abschnitt 8.7 diskutiert. Für die Zukunft ist es denkbar, dass die Entwickler bereits im Dialog zum Hinzufügen von Abhängigkeiten darauf hingewiesen werden, dass eine bestimmte Abhängigkeit zu einem Verstoß führen wird. Von einem strikten Verbot einer Änderung einer Manifest-Datei wird jedoch abgesehen, da es in manchen Situationen zumindest temporär wichtiger ist, ein lauffähiges System zu haben, als dass ein System keine Verstöße gegen die Architekturvorgaben beinhaltet.

Bei der dritten Frage gaben neun der zehn Entwickler an noch keine Änderung vorgenommen zu haben, die gegen das Abhängigkeitsmodell verstoßen hat. Dies deckt sich mit den Ergebnissen zu den ersten beiden Forschungsfragen. Der Entwickler, dessen Änderung zu einem Verstoß führte, hat diesen Verstoß durch eine Änderung seiner Implementierung wieder behoben.

Im Bereich zur Angabe von offenem Feedback gab ein Entwickler an, dass Eclipse durch die initiale Prüfung aller Manifest-Dateien etwas langsamer startet, dies aber kaum auffällt. Ein weiterer Entwickler merkte an, dass in manchen Fällen die Fehlermeldungen ein bisschen anders formuliert sein sollten, dass die Prüfkomponente dessen ungeachtet aber sehr gut arbeitet. Zudem regte ein Entwickler an das Abhängigkeitsmodell noch weiter auszubauen und weitere Architekturvorgaben hinzuzufügen, damit die Plugin-Struktur auch zukünftig sauber bleibt.

Als Fazit lässt sich festhalten, dass die Entwickler den Ansatz als sinnvolle Erweiterung ansehen. Kombiniert mit den Ergebnissen zu den ersten beiden Forschungsfragen zeigt sich, dass der Ansatz vor allem dabei hilft das Bewusstsein über Verstöße zu schärfen. Selbst wenn die Entwickler auch ohne Ansatz nur wenige neue Verstöße einführen, hilft der Ansatz dabei die bestehenden Verstöße zu entfernen und zu verhindern, dass die Verstöße nach Aufräumarbeiten wieder eingeführt werden.

8.9.4 Gültigkeit der Fallstudie

Die Hauptgefahr für die Gültigkeit der Fallstudie besteht darin, dass sich die Ergebnisse der Fallstudie nicht auf andere Softwaresysteme übertragen lassen, da der Ansatz lediglich auf vier Softwaresysteme angewendet wurde, die innerhalb des gleichen Unternehmens entwickelt wurden. So können in anderen Softwaresystemen beispielsweise Architekturvorgaben existieren, die nicht passend durch DepCoL modelliert werden können. Dadurch, dass die betrachteten Softwaresysteme allerdings komplexe Softwaresysteme mit einer hohen Anzahl von Plugins, Features und Abhängigkeit waren, ist nicht damit zu rechnen, dass DepCoL in vielen anderen Softwaresystemen die erforderlichen Architekturvorgaben nicht umsetzen kann. Ferner würde es kein fundamentales Problem darstellen, wenn spezifische Architekturvorgaben nicht direkt umgesetzt werden können, da DepCoL auf einfache Weise um Sprachkonstrukte erweitert werden kann, um ebenfalls andersartige Architekturvorgaben ausdrücken zu können.

Eine weitere mögliche Gefahr für die Gültigkeit der Fallstudie ist, dass nicht gesichert ist, dass die Entwickler die Verstöße gegen die Architekturvorgaben aufgrund der Prüfkomponte behoben haben. Zwar kann nicht ausgeschlossen werden, dass die Entwickler einen Teil der Verstöße auch ohne die Prüfkomponte entfernt hätten, allerdings liefern die Ergebnisse der Beobachtungsphase und der Anwendungsphase der Fallstudie starke Indizien dafür, dass die Änderungen im Wesentlichen in dem Einsatz der Prüfkomponte begründet sind. Im Rahmen der Beobachtungsphase wurden die Architekturverstöße zwar ermittelt, nicht aber an die Systemarchitekten und Entwickler gemeldet. In dieser Phase wurde kein Architekturverstoß behoben. In der anschließenden Anwendungsphase, in der die Systemarchitekten und Entwickler über die Verstöße informiert wurden, wurden hingegen zahlreiche Verstöße behoben. Dies lässt darauf schließen, dass die Behebung der Verstöße insbesondere in dem Einsatz der Prüfkomponte begründet war, da diese die Systemarchitekten und Entwickler in der Anwendungsphase auf die Verstöße aufmerksam gemacht hat.

Aus den Erkenntnissen zu der zweiten Forschungsfrage kann indes nicht allgemeingültig geschlossen werden, dass auch in anderen Softwaresystemen die gemeldeten Architekturverstöße behoben werden. Hierbei gilt es zu beachten, dass der Einsatz des in dieser Arbeit vorgestellten Ansatzes nicht zwangsläufig zu einer höheren Architekturkonformität führt. Eine wesentliche Voraussetzung für die Erhöhung der Architekturkonformität ist, dass die Entwicklungsteams und Systemarchitekten verinnerlichen, welche Vorteile eine höhere Architekturkonformität für die Softwaresysteme und den Entwicklungsprozess hat. Dies gilt für jeden Ansatz im Kontext der Prüfung der Architekturkonformität. Der Ansatz dieser Arbeit bietet den Entwicklern und Systemarchitekten aber die Mittel dazu, die Architekturkonformität auf einfachere Weise prüfen und erhöhen zu können.

Kapitel 9

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde für verschiedene Gebiete aus dem Bereich der Software Evolution verdeutlicht, wie die Software Evolution im industriellen Kontext über Modellierungs- und Generierungstechniken unterstützt werden kann. Der Fokus dieser Arbeit lag dabei auf Softwaresystemen, die modellgetrieben entwickelt werden. Sämtliche entwickelten Ansätze wurden im Unternehmen DSA GmbH in dort in Entwicklung bzw. Wartung und Evolution befindlichen Produkten evaluiert.

Im Folgenden werden zunächst die wesentlichen Ergebnisse dieser Arbeit zusammengefasst. Im Anschluss daran werden mögliche Erweiterungen dargelegt. Diese betreffen einerseits Erweiterungen der vorgestellten Ansätze, die sich während der Anfertigung der Arbeit gezeigt haben, sowie Erweiterungen mit Hinblick auf andere Themengebiete der Software Evolution.

9.1 Zusammenfassung

In Kapitel 1 wurde die Arbeit einleitend motiviert. Darüber hinaus wurden die zentralen Forschungsfragen definiert, welche dieser Arbeit zugrunde liegen.

Darauffolgend wurde in Kapitel 2 der Kontext der Arbeit im Detail erläutert. Dazu wurde zu jedem Teilgebiet der Software Evolution, welches in dieser Arbeit vorgestellt wurde, eine allgemeine Problembeschreibung dargelegt. Diese zeigte, dass die in dieser Arbeit behandelten Themen nicht spezifisch für die Softwaresysteme bei DSA sind. Vielmehr sind die Themen auf viele Softwaresysteme übertragbar, die generativ entwickelt werden. Ferner wurde dargelegt, wie die allgemeine Problembeschreibung im Kontext dieser Arbeit ausgeprägt ist und wie die Forschungsfrage, die jedem Teilgebiet zugrunde liegt, in dieser Arbeit beantwortet wird.

Im Anschluss daran wurden in Kapitel 3 die Grundlagen der DSL-Entwicklung vorgestellt. Diese umfassen die wesentlichen Bestandteile der Grammatiken, die in der Arbeit an verschiedenen Stellen zur Definition der entwickelten textuellen DSLs genutzt werden.

Als Nächstes wurde in Kapitel 4 erklärt, wie in dieser Arbeit vorgegangen wird, um die Modellunterschiede zwischen zwei Modellversionen zu bestimmen. Dieser Schritt ist für die modellgetriebene Softwareentwicklung von zentraler Bedeutung, da dort Modelle die primären Entwicklungsartefakte sind. Zudem müssen Modelle, wie alle Artefakte im Softwareentwicklungsprozess, im Zuge der Software Evolution kontinuierlich angepasst werden [Leh79]. In dieser Arbeit wird ein zustandsbasierter Ansatz zur Unterschiedsberechnung verfolgt. Dieser Ansatz beinhaltet, dass die Unterschiedsberechnung in zwei Phasen erfolgt: der Matchingphase und der Differenzierungsphase. In der Matchingphase werden die zueinander korrespondierenden Mo-

dellelemente bestimmt. Aufbauend darauf wird in der Differenzierungsphase abgeleitet, welche Modelländerungen vorgenommen wurden.

Ein elementares Problem bei der automatischen Bestimmung von Modellunterschieden ist, dass nicht garantiert werden kann, dass die korrekten Modellunterschiede ermittelt werden. Der Hauptgrund hierfür ist, dass es normalerweise verschiedene Interpretationen dazu geben kann, welche Modelländerungen vorgenommen wurden. Problematisch ist dieser Sachverhalt dann, wenn auf Basis der ermittelten Modellunterschiede automatisch weitere Aktivitäten ausgeführt werden. Um zu verhindern, dass diese Aktivitäten auf Basis von fehlerhaften Modellunterschieden ausgeführt werden, wurde in dieser Arbeit das Konzept der Delta-Benutzervorgaben eingeführt. Diese ermöglichen es, für ein Modellelement festzulegen, auf welches andere Modellelement das Modellelement in der Matchingphase abgebildet wird. Folglich können Delta-Benutzervorgaben für die Modellelemente definiert werden, für die andernfalls fehlerhafte Modellunterschiede ermittelt werden würden.

Anschließend wurde in Kapitel 5 ein Ansatz vorgestellt, der es ermöglicht, die Auswirkungen von Modelländerungen explizit über Auswirkungsregeln zu modellieren. Auf diese Weise können erfahrene Entwickler das Wissen über bekannte Abhängigkeiten und Charakteristiken eines Softwaresystems in Auswirkungsregeln festhalten. Auf Basis der modellierten Auswirkungsregeln erzeugt ein Generator Implementierungen der Auswirkungsregeln. Mit Hilfe dieser Implementierungen kann für eine Menge von Modelländerungen automatisch geprüft werden, ob eine spezifische Modelländerung Auswirkungen hat und falls ja, welche. Als Ergebnis des Prozesses liegt eine Checkliste vor, die aufführt, welche Tätigkeiten aufgrund der Modelländerungen durchgeführt werden müssen. Eine derart generierte Checkliste unterstützt die Entwickler auf verschiedene Weisen bei der Software Evolution:

- Die Checkliste führt explizit auf, welche Tätigkeiten aufgrund der Modelländerungen durchzuführen sind. Dadurch wird die Wahrscheinlichkeit, dass die Entwickler vergessen diese Tätigkeiten durchzuführen, stark reduziert.
- Die Prüfung der Auswirkungen erfolgt jedes Mal nach dem gleichen Schema. Bei der manuellen Prüfung ist es hingegen abhängig davon, wie erfahren der Entwickler ist und ob er alle erforderlichen Fälle bedenkt oder nicht.
- Die Checkliste kann auf Artefakte verweisen, die angepasst werden müssen, so dass die Entwickler diese Artefakte nicht selbst ermitteln müssen.

In einer Fallstudie hat sich gezeigt, dass der Ansatz in einem realen Softwaresystem verwendet werden kann, um die Auswirkungen von Modelländerungen zu modellieren. Darüber hinaus verdeutlichte die Fallstudie, dass einige der entwickelten Auswirkungsregeln auf erforderliche Änderungen hingewiesen haben, die in der Vergangenheit teilweise vergessen wurden durchzuführen. Auf der anderen Seite zeigte die Fallstudie, dass es in komplexen Szenarien nicht immer praktikabel ist eine Auswirkungsregel vollumfänglich zu modellieren. In derartigen Fällen bietet sich die partielle Implementierung der Auswirkungsregel in Java an.

Darauffolgend wurde in Kapitel 6 erläutert, wie für verschiedene Arten von Modelländerungen vorgegangen werden muss, um bestehende Daten konform zu Modelländerungen zu migrieren. In dem betrachteten Anwendungsfall werden Klassendiagramme zur Modellierung der Datenmodelle eingesetzt und als zu migrierende Daten liegen JSON-Dokumente vor. Dabei können

nur JSON-Dokumente zu Instanzen spezifischer Klassen, genannt primäre Klassen, existieren. Aus diesem Grund werden in einem ersten Schritt die Pfade von den primären Klassen ausgehend zu den von den Modelländerungen betroffenen Modellelementen bestimmt. Aus diesen Pfaden kann abgeleitet werden, wie in einem JSON-Dokument zu den Daten navigiert werden kann, die von den Änderungen betroffen sind. Aus diesen Pfaden werden anschließend Graphen konstruiert, um Zyklen in den Assoziationsketten der Klassendiagramme einfacher handhaben zu können. Auf Basis dieser Graphen erzeugt ein Generator die Migrationslogik zur Migration der zugehörigen JSON-Dokumente. Zudem wurde dargelegt, dass in manchen Fällen eine Minimierung der Graphen durchgeführt werden kann. Dieser Fall tritt auf, wenn durch eine Modelländerung in einem JSON-Dokument ein Teil des Dokuments entfernt werden würde und andere Modelländerungen betreffen eine Region des JSON-Dokuments, die von dem Löschen betroffen ist. Der vorgestellte Ansatz wurde in einer Fallstudie erfolgreich dafür genutzt, Migrationsklassen für eine Versionshistorie eines realen Softwaresystems zu generieren. Ferner wurde in der Fallstudie demonstriert, dass über die generierten Migrationsklassen reale Datenbestände korrekt migriert werden konnten. Eine Minimierung der Graphen war im Rahmen der Fallstudie hingegen lediglich in einem einzigen Fall möglich.

Im Anschluss daran wurde in Kapitel 7 ein Ansatz zur Modellierung von Generatorvariabilität vorgestellt, über den auf einfache Weise Generatorvarianten entwickelt werden können. Die Kernidee dieses Ansatzes ist es, dass über eine DSL Operationen modelliert werden können, die ausdrücken, dass eine dedizierte Region eines Generatorartefakts durch anderen Inhalt ersetzt werden soll oder dass Inhalt vor oder hinter eine dedizierte Region eines Generatorartefakts eingefügt werden soll. Auf diese Weise kann der Inhalt beliebiger Regionen von Generatorartefakten gezielt verändert werden, um neue Generatorvarianten zu modellieren. Darüber hinaus wird ausgenutzt, dass die Generatorartefakte in wiederverwendbaren Schichten organisiert werden. Eine konkrete Generatorvariante kann durch die Auswahl einer spezifischen Schicht festgelegt werden. Die Folge davon ist, dass die Generatorartefakte der ausgewählten Schicht sowie die Generatorartefakte der weiteren benötigten Schichten in einem Kompositionsschritt zu neuen Generatorartefakten komponiert werden. Im Rahmen einer Fallstudie wurde zudem belegt, dass der Ansatz dazu geeignet ist Generatorvarianten in einem realen Kontext zu entwickeln. Bei dieser Anwendung des Ansatzes zeigte sich insbesondere, dass der Umfang der zu entwickelnden Templates und Hilfsklassen zum Teil deutlich reduziert werden kann, im Vergleich zur informellen Wiederverwendung über das Kopieren eines Generators und dem Ändern dieser Kopie. Desto geringer die Unterschiede zwischen den zu entwickelnden Generatorvarianten sind, desto geringer ist auch der Modellierungsaufwand zur Spezifikation der erforderlichen Änderungsoperationen.

Abschließend wurde in Kapitel 8 ein Ansatz zur automatischen Prüfung der Architekturkonformität in Plugin-basierten Softwaresystemen eingeführt. Dieser sieht vor, dass Entwickler über eine DSL festhalten, welche Elemente eines Plugin-basierten Softwaresystems (nicht) von welchen anderen Elementen des Plugin-basierten Softwaresystems abhängen dürfen. Diese DSL ermöglicht es Restriktionen zu Abhängigkeiten zwischen Features, Plugins, frei definierbaren Feature-Gruppen sowie frei definierbaren Plugin-Gruppen zu definieren. Insbesondere kann ausgedrückt werden, dass spezifische Abhängigkeiten erlaubt oder verboten sind oder lediglich toleriert werden. Ferner wurde eine Prüfkomponente entwickelt, welche in die Entwicklungsumge-

bung des Softwaresystems integriert werden kann. Diese prüft begleitend zur Weiterentwicklung des Systems, ob Änderungen vorgenommen wurden, die gegen modellierte Abhängigkeitsrestriktionen verstoßen. Auf diese Weise können Entwickler umgehend nach Änderungen über Verstöße informiert werden. Durch eine Fallstudie wurde verdeutlicht, dass der Ansatz dafür geeignet ist, die Architekturkonformität in komplexen Softwaresystemen zu erhöhen. So konnten in der Fallstudie sämtliche Vorgaben der Systemarchitekten problemlos mit der DSL abgebildet werden. Zudem wurden unter Nutzung der Prüfkomponeente zahlreiche bestehende Verstöße gegen Abhängigkeitsrestriktionen beseitigt.

9.2 Ausblick

Der in Kapitel 5 vorgestellte Ansatz zur Modellierung der Auswirkungen von Modelländerungen wurde in dieser Arbeit nur auf Klassendiagramme angewendet. Das Konzept ist allerdings nicht auf Klassendiagramme beschränkt, sondern kann auch auf andere Modellierungstechniken übertragen werden. Dies könnte in zukünftigen Arbeiten für weitere Modellierungstechniken demonstriert werden. In der Fallstudie hat sich zudem gezeigt, dass unterschiedlich erfahrene Entwickler unterschiedliche Anforderungen an den Detailgrad einer Checkliste haben. Aus diesem Grund könnte der Ansatz erweitert werden, so dass die Auswirkungen auf verschiedenen Detailebenen definiert werden können. Der Checklistengenerator könnte infolgedessen verschiedene Varianten der Checklisten erzeugen. Darüber hinaus könnte der Ansatz in der Art erweitert werden, dass vor der Erzeugung einer Checkliste geprüft wird, ob die Entwickler bereits Änderungen vorgenommen haben, auf welche die Checkliste hinweisen würde. In diesen Fällen müssten die Hinweise zu den entsprechenden Änderungen nicht mehr mit in die Checkliste aufgenommen werden und die Entwickler würden nur noch die relevanten Checklistenhinweise sehen.

Der Ansatz zur Generierung der Datenmigrationsfunktionalität zur Daten Evolution, siehe Kapitel 6, ermittelt bisher automatisch die Modelländerungen, die zwischen zwei Versionen eines Datenmodells vorgenommen wurden. Alternativ dazu wäre es denkbar, eine DSL bereitzustellen, durch welche die Änderungsoperationen a-priori modelliert werden können. Bei DSA wurde von dieser Möglichkeit abgesehen, da die Modellierer die Klassendiagramme direkt modifizieren möchten, so dass die Modelländerungen danach abgeleitet werden müssen.

Wird die bestehende Vorgehensweise beibehalten, könnte eine sinnvolle Erweiterung darin bestehen, die vorliegenden Modelländerungen noch genauer zu untersuchen und diese auf eine andere Abstraktionsstufe anzuheben [KKT11, KKOS12]. Beispielsweise kann bereits auf der Ebene der Modelländerungen ausgedrückt werden, dass ein Attribut in der Vererbungshierarchie nach oben gezogen wurde - bekannt als *Pull Up Refactoring* [Fow99]. Dadurch kann die Implementierung der Generierungskomponente für die Datenmigrationen vereinfacht werden.

Im Kontext des Konzepts zur Modellierung von Generatorvariabilität, welches in Kapitel 7 vorgestellt wurde, besteht Erweiterungspotential mit Hinblick auf die Unterstützung der Evolution der Generatorvarianten. Werden die Generatorartefakte verändert, auf Basis derer die Generatorvarianten erzeugt werden, kann dies dazu führen, dass für die Generatorvarianten weitere Änderungen vorgenommen werden müssen. Ist für eine Generatorvariante beispielsweise definiert, dass eine bestimmte Region eines Artefakts A durch den Inhalt aus Artefakt B ersetzt

werden soll und die betroffene Region des Artefakts A wird geändert, kann es erforderlich sein Teile von B ebenfalls anpassen zu müssen. Entwickler müssen dies derzeit selbstständig erkennen und die Artefakte entsprechend anpassen. Zur Unterstützung der Entwickler sollten Konzepte zur Auswirkungsanalyse [Arn96] integriert werden, damit die Entwickler besser abschätzen können, welche Teile der Generatorvarianten angepasst werden müssen.

Die Prüfkomponte zur Prüfung der Architekturkonformität meldet, wie in Kapitel 8 beschrieben, Verstöße gegen die modellierten Abhängigkeitsrestriktionen. In der praktischen Anwendung der Prüfkomponte zeigte sich jedoch, dass es in manchen Fällen nicht nur hilfreich ist zu wissen, dass Verstöße vorliegen, sondern auch, welche konkreten Code-Zeilen zu den Verstößen führen. Wird eine verbotene Abhängigkeit zwischen zwei Plugins gemeldet, ist eine einfache Vorgehensweise zur Identifikation des verursachenden Codes die Abhängigkeit aus der Manifest-Datei zu entfernen. Anschließend weist die Entwicklungsumgebung über Syntaxfehler darauf hin, falls Code vorliegt, der Code des anderen Plugins verwendet hat. In einer Arbeitsumgebung mit vielen Plugins kann diese einfache Herangehensweise jedoch zu sehr langen Wartezeiten führen. Alternativ könnte das Tooling erweitert werden, so dass das Tooling die Code-Bereiche, die zu Verstößen gegen die Vorgaben führen, identifiziert und die Entwickler darüber informiert.

Im Rahmen dieser Arbeit konnten aufgrund der Vielzahl an Themen im Gebiet der Software Evolution nur einige ausgewählte Teilgebiete untersucht werden. Darüber hinaus gibt es aber noch viele weitere Teilgebiete, die zukünftig noch intensiver untersucht werden könnten, um die Software Evolution über Modellierungs- und Generierungstechniken zu verbessern. Zu diesen zählen beispielsweise:

- die Modellierung und Generierung von Testfällen und Testdaten. In generativen Softwareentwicklungsprozessen könnte insbesondere das Testen von Generatoren [KLM⁺16, SWC05] intensiviert werden, da Generatoren essentielle Teile der Softwaresysteme generieren.
- die Analyse der verschiedenen Artefakte eines Softwaresystems. Sowohl in generativen als auch in nicht-generativen Softwareentwicklungsprozessen existieren eine Vielzahl handgeschriebener Artefakte. Mittels einer tiefergehenden Analyse dieser Artefakte könnte es möglich sein Artefakte automatisch zu identifizieren, die fortan generiert werden können [MSNR15]. Dadurch kann die Anzahl handgeschriebener Artefakte reduziert werden. Der Vorteil davon ist, dass Änderungen zentral im Generator durchgeführt werden können, anstatt daran denken zu müssen, die notwendigen Informationen in verschiedenen Artefakten korrekt zu pflegen.
- die Analyse und Verbesserung von Modelltransformationen in generativen Softwareentwicklungsprozessen. Hierbei könnte intensiver untersucht werden, wie die Nutzung von Modelltransformationen im industriellen Kontext gefördert werden kann und wie die Definition von Modelltransformationen vereinfacht werden kann. Beispielsweise können domänenspezifische Transformationssprachen generiert werden, welche die konkrete Syntax einer Sprache nutzen [Wei12, HRW15].

Literaturverzeichnis

- [ABK⁺09] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why Model Versioning Research is Needed!? An Experience Report. In *Proceedings of the Joint MoDSE-MCCM 2009 Workshop*, 2009.
- [ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, October 2013.
- [Acc] Acceleo Homepage. <http://www.eclipse.org/acceleo/>. (Stand: 24.06.2016).
- [ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the International Conference on Software Engineering, ICSE '02*, pages 187–197, New York, NY, USA, 2002. ACM.
- [AJTK09] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. Model Superimposition in Software Product Lines. In *Proceedings of the International Conference on Theory and Practice of Model Transformations, ICMT '09*, pages 4–19, Berlin, Heidelberg, 2009. Springer-Verlag.
- [ALP91] José Andany, Michel Léonard, and Carole Palisser. Management Of Schema Evolution In Databases. In *Proceedings of the International Conference on Very Large Data Bases, VLDB '91*, pages 161–170, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
- [ALS05] Sven Apel, Thomas Leich, and Gunter Saake. Aspect Refinement and Bounding Quantification in Incremental Designs. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference, APSEC '05*, pages 796–804, Washington, DC, USA, 2005. IEEE Computer Society.
- [ALS06] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering, ICSE '06*, pages 122–131, New York, NY, USA, 2006. ACM.
- [AP03] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2003.

- [ARB12] Nour Ali, Jacek Rosik, and Jim Buckley. Characterizing Real-time Reflexion-based Architecture Recovery: An In-vivo Multi-case Study. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '12, pages 23–32, New York, NY, USA, 2012. ACM.
- [Arn96] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
- [Art88] Lowell Jay Arthur. *Software Evolution: The Software Maintenance Challenge*. Wiley-Interscience, New York, NY, USA, 1988.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [BBC⁺00] Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord, and Reed Little. Software Architecture Documentation in Practice: Documenting Architectural Layers. Technical Report CMU/SEI-2000-SR-004, Software Engineering Institute, Carnegie Mellon University, 2000.
- [BBE⁺95] Serge Barros, Thierry Bodhuin, Anne Escudie, Jean-Pierre Queille, and Jean-Francois Voidrot. Supporting Impact Analysis: A Semi-automated Technique and associated Tool. In *Proceedings of the International Conference on Software Maintenance*, ICSM '95, pages 42–51, Washington, DC, USA, 1995. IEEE Computer Society.
- [BCE⁺06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A Manifesto for Model Merging. In *Proceedings of the International Workshop on Global Integrated Model Management*, GaMMa '06, pages 5–12, New York, NY, USA, 2006. ACM Press.
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012.
- [Ben87] Salah Bendifallah. Understanding Software Maintenance Work. *IEEE Transactions on Software Engineering*, SE-13(3):311–323, 1987.
- [Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Springer-Verlag, 2013.
- [Bib] Bibliography on Comparison and Versioning of Software Models. <http://pi.informatik.uni-siegen.de/CVSM/>. (Stand: 24.06.2016).
- [BJN⁺06] Manfred Broy, Matthias Jarke, Manfred Nagl, Hans D. Rombach, Armin B. Cremers, Jürgen Ebert, Sabine Glesner, Martin Glinz, Michael Goedicke, Gerhard Goos, Volker Gruhn, Wilhelm Hasselbring, Stefan Jähnichen, Stefan Kowalewski, Bernd J. Krämer, Stefan Leue, Claus Lewerentz, Peter Liggesmeyer, Christoph Lüth, Barbara Paech, Helmut A. Partsch, Ilka Philippow, Lutz Prechelt,

- Andreas Rausch, Willem-Paul de Roever, Bernhard Rumpe, Gudula Rünger, Wilhelm Schäfer, Kurt Schneider, Andy Schürr, Walter F. Tichy, Bernhard Westfechtel, Wolf Zimmermann, and Albert Zündorf. Dagstuhl-manifest zur strategischen bedeutung des software engineering in deutschland. In Manfred Broy, Manfred Nagl, Hans Dieter Rombach, and Matthias Jarke, editors, *Perspectives Workshop*, number 05402 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- [BKL⁺12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An Introduction to Model Versioning. In *Proceedings of the International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*, SFM '12, pages 336–398, Berlin, Heidelberg, 2012. Springer-Verlag.
- [BKPS07] Manfred Broy, Ingolf H. Krüger, Alexander Pretschner, and Christian Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007.
- [BL10] David Binkley and Dawn Lawrie. Information Retrieval Applications in Software Maintenance and Evolution. In *Encyclopedia of Software Engineering*, pages 454–463. 2010.
- [BLO03a] Lionel C. Briand, Yvan Labiche, and Leeshawn O’Sullivan. Impact Analysis and Change Management of UML Models. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 256–265. IEEE Computer Society, 2003.
- [BLO03b] Lionel C. Briand, Yvan Labiche, and Leeshawn O’Sullivan. Impact Analysis and Change Management of UML Models. Technical Report SCE-03-01, Carleton University, 2003.
- [BLOS06] Lionel C. Briand, Yvan Labiche, Leeshawn O’Sullivan, and Michal M. Sówka. Automated Impact Analysis of UML Models. *Journal of Systems and Software*, 79(3):339–352, March 2006.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. John Wiley & Sons, 1996.
- [Boh95] Shawn A. Bohner. *A Graph Traceability Approach for Software Change Impact Analysis*. PhD thesis, George Mason University, Fairfax, VA, USA, 1995.
- [BP08] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modeling Framework. *The European Journal for the Informatics Professional*, pages 29–34, 2008.

- [BR00] Keith H. Bennett and Václav T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007.
- [Bra14] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, RFC Editor, March 2014.
- [Bro06] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceedings of the International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.
- [BSR03] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-wise Refinement. In *Proceedings of the International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.
- [CA05] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE '05, pages 422–437, Berlin, Heidelberg, 2005. Springer-Verlag.
- [CC05] Gerardo Canfora and Luigi Cerulo. Impact Analysis by Mining Software and Change Request Repositories. In *Proceedings of the IEEE International Software Metrics Symposium*, METRICS '05, pages 29–36, Washington, DC, USA, 2005. IEEE Computer Society.
- [CC06] Gerardo Canfora and Luigi Cerulo. Fine Grained Indexing of Software Repositories to Support Impact Analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR '06, pages 105–111, New York, NY, USA, 2006. ACM.
- [CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003.
- [CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006.
- [Che75] Peter Pin-Shan Chen. The Entity-relationship Model: Toward a Unified View of Data. *SIGIR Forum*, 10(3):9–9, December 1975.
- [Che02] Peter Chen. Software pioneers. chapter Entity-relationship Modeling: Historical Events, Future Trends, and Lessons Learned, pages 296–310. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

- [CKKL02] Mamade A. Chaumon, Hind Kabaili, Rudolf K. Keller, and François Lustman. A Change Impact Model for Changeability Assessment in Object-oriented Software Systems. *Science of Computer Programming - Software Maintenance and Reengineering*, 45:155–174, November 2002.
- [CMDZ13] Carlo Curino, Hyun J. Moon, Alin Deutsch, and Carlo Zaniolo. Automating the Database Schema Evolution Process. *The VLDB Journal – The International Journal on Very Large Data Bases*, 22(1):73–98, February 2013.
- [CMZ08] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful Database Schema Evolution: The PRISM Workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772, August 2008.
- [CMZ09] Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. Automating Database Schema Evolution in Information System Upgrades. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM.
- [CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.
- [CRF03] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, August 2003.
- [DDHR09] Constanze Deiters, Patrick Dohrmann, Sebastian Herold, and Andreas Rausch. Rule-Based Architectural Compliance Checks for Enterprise Architecture Management. In *Proceedings of the International Enterprise Distributed Object Computing Conference*, EDOC '09, pages 158–167, Piscataway, NJ, USA, Sept 2009. IEEE Press.
- [DHHJ10] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Juergens. Flexible Architecture Conformance Assessment with ConQAT. In *Proceedings of the International Conference on Software Engineering*, volume 2 of ICSE '10, pages 247–250, New York, NY, USA, May 2010. ACM.
- [DLFO08] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. Traceability Management for Impact Analysis. In *Frontiers of Software Maintenance (FoSM'08)*, pages 21–30, October 2008.
- [DLRZ05] Eladio Domínguez, Jorge Lloret, Ángel L. Rubio, and María A. Zapata. Evolving XML Schemas and Documents Using UML Class Diagrams. In *Proceedings of the International Conference on Database and Expert Systems Applications*, DEXA '05, pages 343–352, Berlin, Heidelberg, 2005. Springer-Verlag.

- [DLZ03] Eladio Domínguez, Jorge Lloret, and María A. Zapata. *Advanced Conceptual Modeling Techniques – ER 2002 Workshops*, chapter An Architecture for Managing Database Evolution, pages 63–74. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
- [DSAA] DSA Homepage. <http://www.dsa.de/>. (Stand: 24.06.2016).
- [DSAb] DSA Homepage. Autorenwerkzeug PRODIS.Authoring. <http://www.dsa.de/de/loesungen/produkte/autorenwerkzeug-prodisauthoring/>. (Stand: 24.06.2016).
- [DSAc] DSA Homepage. Autorenwerkzeug PRODIS.Automation. <http://www.dsa.de/de/loesungen/produkte/autorenwerkzeug-prodisautomation/>. (Stand: 24.06.2016).
- [DSAd] DSA Homepage. Über DSA. <http://www.dsa.de/de/ueber-dsa/>. (Stand: 24.06.2016).
- [DS Ae] DSA Homepage. Laufzeitsystem PRODIS.RTS. <http://www.dsa.de/de/loesungen/produkte/laufzeitsystem-prodisrts/>. (Stand: 24.06.2016).
- [DS Af] DSA Homepage. Laufzeitsystem PRODIS.WTS. <http://www.dsa.de/de/loesungen/produkte/laufzeitsystem-prodiswts/>. (Stand: 24.06.2016).
- [Ecl] Eclipse Modeling Framework Homepage. <https://eclipse.org/modeling/emf/>. (Stand: 24.06.2016).
- [EMFa] EMF Compare Homepage. <http://www.eclipse.org/emf/compare/>. (Stand: 24.06.2016).
- [EMFb] EMF Store Homepage. <http://www.eclipse.org/emfstore/>. (Stand: 24.06.2016).
- [EPK06] Klaus-Dietrich Engel, Richard F. Paige, and Dimitrios S. Kolovos. Using a Model Merging Language for Reconciling Model Versions. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 143–157. Springer Berlin Heidelberg, 2006.
- [EvdSV⁺13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. *The State of the Art in Language Workbenches*, pages 197–217. Springer International Publishing, Cham, 2013.

- [FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
- [Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.
- [FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Proceedings Future of Software Engineering, FUSE '07*, pages 37–54, 2007.
- [GG08] Michael W. Godfrey and Daniel M. German. The Past, Present and Future of Software Architecture. In *Frontiers of Software Maintenance, FoSM '08*, pages 129–138, 2008.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [GHK⁺15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram M. S. Nazari, Klaus Müller, Antonio N. Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, pages 74–85. Scitepress, 2015.
- [GHK⁺15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram M. S. Nazari, Klaus Müller, Antonio N. Pérez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In P. Desfray, J. Filipe, S. Hammoudi, and L.F. Pires, editors, *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112 – 132. Springer, 2015.
- [Gmb10] Generative Software GmbH. Umfrage zu Verbreitung und Einsatz modellgetriebener Softwareentwicklung - Abschlussbericht. Technical report, FZI (Forschungszentrum Informatik, Karlsruhe), 2010. <http://www.mdsd-umfrage.de/mdsd-report-2010.pdf>.
- [GMR15] Timo Greifenberg, Klaus Müller, and Bernhard Rumpe. Architectural Consistency Checking in Plugin-Based Software Systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW '15*, pages 58:1–58:7, New York, NY, USA, 2015. ACM.

- [GMR⁺16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016*, pages 141–156. Bonner Köllen Verlag, 2016.
- [Gom04] Hassan Gomaa. *Designing Software Product Lines with UML*. Addison-Wesley Boston, 2004.
- [Goo] Google Cloud Platform Homepage. <https://cloud.google.com/>. (Stand: 24.06.2016).
- [GS94] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical report, Pittsburgh, PA, USA, 1994.
- [HB99] Terry Halpin and Anthony Bloesch. Data modeling in UML and ORM: a comparison. *Journal of Database Management*, 10(4):4–13, 1999.
- [Her11] Markus Herrmannsdoerfer. COPE – A Workbench for the Coupled Evolution of Metamodels and Models. In *Proceedings of the International Conference on Software Language Engineering, SLE '11*, pages 286–295, Berlin, Heidelberg, 2011. Springer-Verlag.
- [HHK⁺13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the International Software Product Line Conference, SPLC'13*, pages 22–31, Tokyo, Japan, September 2013. ACM.
- [HHK⁺15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.
- [HK10] Markus Herrmannsdoerfer and Maximilian Koegel. Towards a Generic Operation Recorder for Model Evolution. In *Proceedings of the International Workshop on Model Comparison in Practice, IWMCP'10*, pages 76–81, New York, NY, USA, 2010. ACM.
- [HMPO⁺08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the International Software Product Line Conference, SPLC '08*, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.
- [HR13] Sebastian Herold and Andreas Rausch. Complementing Model-Driven Development for the Detection of Software Architecture Erosion. In *International Workshop on Modeling in Software Engineering, MiSE '13*, pages 24–30, 2013.

- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 1 – 10, Munich, Germany, February 2011. fortiss GmbH.
- [HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems, MODELS'15*, pages 136–145. ACM/IEEE, 2015.
- [HvdH07] Scott A. Hendrickson and Andre van der Hoek. Modeling Product Line Architectures through Change Sets and Relationships. In *Proceedings of the International Conference on Software Engineering, ICSE '07*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.
- [HVEK07] Arno Haase, Markus Völter, Sven Efftinge, and Bernd Kolb. Introduction to open-ArchitectureWare 4.1.2. *Model-Driven Development Tool Implementers Forum (MDD-TIF'07)*, 2007.
- [HW07] Florian Heidenreich and Christian Wende. Bridging the Gap Between Features and Models. *Aspect-Oriented Product Line Engineering (AOPLE '07)*, 2007.
- [HWR14] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that lead to success or failure. *Science of Computer Programming*, 89, Part B:144 – 161, 2014. Special issue on Success Stories in Model Driven Engineering.
- [HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the International Conference on Software Engineering, ICSE '11*, pages 471–480, New York, NY, USA, 2011. ACM.
- [Jac] Jackson Homepage. <https://github.com/FasterXML/jackson>. (Stand: 24.06.2016).
- [JMS08] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. Genesys: Service-Oriented Construction of Property Conform Code Generators. *Innovations in Systems and Software Engineering*, 4(4):361–384, 2008.
- [Jör13a] Sven Jörges. *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*, volume 7747 of *Lecture Notes in Computer Science*. Springer, 2013.
- [Jör13b] Sven Jörges. *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*, chapter Case Study: Service-Oriented Combination of Code Generation Frameworks, pages 193–203. Springer, Berlin, Heidelberg, 2013.

- [KCM05] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Towards a Taxonomy of Approaches for Mining of Source Code Repositories. *SIGSOFT Software Engineering Notes*, 30(4):1–5, May 2005.
- [KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, March 2007.
- [KDRPP09] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.
- [KGFE08] Jochen M. Küster, Christian Gerth, Alexander Förster, and Gregor Engels. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008.
- [KHWH10] Maximilian Koegel, Markus Herrmannsdoerfer, Otto von Wesendonk, and Jonas Helming. Operation-based Conflict Detection. In *Proceedings of the International Workshop on Model Comparison in Practice*, IWMCP'10, pages 21–30, New York, NY, USA, 2010. ACM.
- [KKK13] Udo Kelter, Timo Kehrer, and Dennis Koch. Patchen von Modellen. In *Software Engineering*, volume 213 of *LNI*, pages 171–184. GI, 2013.
- [KKOS12] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding Model Evolution Through Semantically Lifting Model Differences with SiLift. In *Proceedings of the International Conference on Software Maintenance*, ICSM'12, pages 638–641. IEEE Computer Society, 2012.
- [KKPS12a] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of Model Comparison Tools. In *Proceedings of the International Conference on Automated Software Engineering*, ASE'12, pages 306–309, New York, NY, USA, 2012. ACM.
- [KKPS12b] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Operation-based Model Differencing meets State-based Model Comparison. *Softwaretechnik-Trends*, 32(4), 2012.
- [KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A Rule-based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 163–172, Washington, DC, USA, 2011. IEEE Computer Society.

- [Kle07] Meike Klettke. Conceptual XML Schema Evolution - the CoDEX Approach for Design and Redesign. In *Proceedings Datenbanksysteme in Business, Technologie und Web*, BTW '07, pages 53–63, Aachen, Germany, 2007.
- [KLM⁺16] Carsten Kolassa, Markus Look, Klaus Müller, Alexander Roth, Dirk Reiß, and Bernhard Rumpe. TUnit - Unit Testing For Template-based Code Generators. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 221–236. Bonner Köllen Verlag, 2016.
- [KP07] Jens Knodel and Daniel Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *The Working IEEE/IFIP Conference on Software Architecture*, WICSA '07, pages 12–12, 2007.
- [KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In *Proceedings of the International Conference on Model Driven Engineering Languages*, MODELS'06, pages 215–229, Berlin, Heidelberg, 2006. Springer-Verlag.
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. PhD thesis, RWTH Aachen University, 2010.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315, Berlin, Heidelberg, 2008. Springer-Verlag Berlin-Heidelberg.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Steven 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.
- [KSW11] Christian Kehrt, Peter Schüßler, and Marc-Denis Weitze. *Neue Technologien in der Gesellschaft: Akteure, Erwartungen, Kontroversen und Konjunktoren*. transcript, 1st edition, 2011.
- [KV10] Lennart C.L. Kats and Eelco Visser. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.
- [KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, pages 105–116. GI, 2005.
- [Lad09] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications, 2nd Edition*. Manning Publications, 2009.

- [Läm04] Ralf Lämmel. Coupled Software Transformations. In *Proceedings of the International Workshop on Software Evolution Transformations*, SETra '04, pages 31–35, 2004.
- [LDLL10] Wen-Tin Lee, Whan-Yo Deng, Jonathan Lee, and Shin-Jie Lee. Change Impact Analysis with a Goal-driven Traceability-based Approach. *International Journal of Intelligent Systems*, 25(8):878–908, August 2010.
- [Leh79] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213 – 221, 1979.
- [Leh11] Steffen Lehnert. A Review of Software Change Impact Analysis. Technical report, Department of Software Systems / Process Informatics, Ilmenau University of Technology, December 2011.
- [LFR13] Steffen Lehnert, Qurat-UI-Ann Farooq, and Matthias Reibisch. Rule-based Impact Analysis for Heterogeneous Software Artifacts. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 209–218, Washington, DC, USA, 2013. IEEE Computer Society.
- [LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: A Differentiation Tool for Domain-specific Models. *European Journal of Information Systems*, 16:349–361, 2007.
- [Lie83] Bennet P. Lientz. Issues in Software Maintenance. *ACM Computing Surveys*, 15(3):271–278, September 1983.
- [LO92] Ernst Lippe and Norbert van Oosterom. Operation-based Merging. In *Proceedings of the ACM SIGSOFT Symposium on Software Development Environments*, SDE'92, pages 78–87, New York, NY, USA, 1992. ACM.
- [Loo17] Markus Look. *Unterstützung modellgetriebener, agiler Entwicklung mehrbenutzerfähiger, ubiquitärer Enterprise Applikationen durch Generatoren*. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, 2017.
- [LRSS07] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Proceedings International Dagstuhl conference on Model-based engineering of embedded real-time systems (MBEERTS'07)*, pages 241–270, 2007.
- [LSGF08] Neil Loughran, Pablo Sánchez, Alessandro Garcia, and Lidia Fuentes. Language Support for Managing Variability in Architectural Models. In *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*. 2008.
- [LSLZ13] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A Survey of Code-based Change Impact Analysis Techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

- [MD08] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition, 2008.
- [Men02] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.
- [Mer10] Bernhard Merkle. Textual Modeling Tools: Overview and Comparison of Language Workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 139–148, New York, NY, USA, 2010. ACM.
- [MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proceedings of the International Conference on Data Engineering*, ICDE '02, pages 117–128, Washington, DC, USA, 2002. IEEE Computer Society.
- [MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, Jun 1995.
- [MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer Society*, 43(5):42–48, 2010.
- [MN97] Gail C. Murphy and David Notkin. Reengineering With Reflexion Models: A Case Study. *Computer*, 30(8):29–36, Aug 1997.
- [MNS95] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.
- [MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering.*, 27(4):364–380, 2001.
- [MR14a] Klaus Müller and Bernhard Rumpe. A Model-Based Approach to Impact Analysis Using Model Differencing. In *Proceedings of the International Workshop on Software Quality and Maintainability*, ECEASST Journal, volume 65 of *SQM'14*, 2014.
- [MR14b] Klaus Müller and Bernhard Rumpe. User-Driven Adaptation of Model Differencing Results. *International Workshop on Comparison and Versioning of Software Models*, *GI Softwaretechnik-Trends*, 34(2):25–29, May 2014.
- [MR15] Klaus Müller and Bernhard Rumpe. A Methodology for Impact Analysis Based on Model Differencing. *Workshop Software-Reengineering und -Evolution*, *GI Softwaretechnik-Trends*, 35(2):13–14, May 2015.

- [MRR11a] Shahar Maoz, Jan O. Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings of the International Conference on Models in Software Engineering*, MODELS '11, pages 194–203, Berlin, Heidelberg, 2011. Springer-Verlag.
- [MRR11b] Shahar Maoz, Jan O. Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proceedings of the European Conference on Foundations of Software Engineering*, ESEC/FSE'11, pages 179–189, New York, NY, USA, 2011. ACM.
- [MRR11c] Shahar Maoz, Jan O. Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In *Proceedings of the European Conference on Object-oriented Programming*, ECOOP'11, pages 230–254, Berlin, Heidelberg, 2011. Springer-Verlag.
- [MRR12] Shahar Maoz, Jan O. Ringert, and Bernhard Rumpe. An Interim Summary on Semantic Model Differencing. *Softwaretechnik-Trends*, 32(4), 2012.
- [MSNR15] Pedram M. S. Nazari and Bernhard Rumpe. Identifying Code Generation Candidates Using Software Categories. In *Model-Driven Engineering and Software Development Conference*, volume 580 of *MODELSWARD'15*, pages 356–372. Springer, 2015.
- [MV11] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223 – 1246, 2011. Special Issue on Software Evolution, Adaptability and Variability.
- [MWD⁺05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayer. Challenges in Software Evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '05, pages 13–22, 2005.
- [NK08] Natsuko Noda and Tomoji Kishi. Aspect-Oriented Modeling for Variability Management. In *Proceedings of the International Software Product Line Conference*, SPLC '08, pages 213–222, Washington, DC, USA, 2008. IEEE Computer Society.
- [NSC⁺07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and Merging of Statecharts Specifications. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 54–64, Washington, DC, USA, 2007. IEEE Computer Society.
- [OMG10a] Object Management Group. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01), 2010.
<http://www.omg.org/spec/OCL/2.2/PDF>.
- [OMG10b] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03), May 2010.
<http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/>.

- [OMG10c] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>.
- [OMG15] Object Management Group. OMG XML Metadata Interchange (XMI), June 2015. <http://www.omg.org/spec/XMI/>.
- [Pan07] Marc Pantel. The TOPCASED project: a Toolkit in OPen source for Critical Applications & SystEms Design. In *Model-Driven Development Tool Implementers Forum (MDD-TIF)*, Zurich, 24/06/2007, page (on line), <http://www.dsmforum.org/events/MDD-TIF07/>, 2007. Domain Specific Modeling Forum (DSMF).
- [Par72] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger, and Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. In *2007 Future of Software Engineering, FOSE '07*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.
- [PJ07a] Markus Pizka and Elmar Jürgens. Automating Language Evolution. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE'07*, pages 305–315, 2007.
- [PJ07b] Markus Pizka and Elmar Jürgens. Tool-supported Multi-Level Language Evolution. In *Proceedings of Software and Services Variability Management Workshop - Concepts, Models and Tools*, pages 48–67, 2007.
- [PMR13] Pit Pietsch, Klaus Müller, and Bernhard Rumpe. Model Matching Challenge: Benchmarks for Ecore and BPMN Diagrams. *Softwaretechnik-Trends*, 33(2), May 2013.
- [PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(K) Parser Generator. *Software Practice & Experience*, 25(7):789–810, July 1995.
- [PTV⁺10] Leonardo Passos, Ricardo Terra, Marco T. Valente, Renato Diniz, and Nabor Mendonca. Static Architecture Conformance Checking: An Illustrative Overview. *IEEE Software*, 27(5):82–89, 2010.
- [PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
- [QVWM94] Jean-Pierre Queille, Jean-Francois Voidrot, Norman Wilde, and Malcolm Munro. The Impact Analysis Task in Software Maintenance: A Model and a Case Study. In *Proceedings of the International Conference on Software Maintenance, ICSM '94*, pages 234–242, 1994.

- [RFG⁺05] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. Model Composition - A Signature-Based Approach. In *Proceedings of the Aspect Oriented Modeling Workshop*, October 2005.
- [RGBB08] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad A. Babar. An Industrial Case Study of Architecture Conformance. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, pages 80–89, New York, NY, USA, 2008. ACM.
- [Rie13] Leanna Rierison. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. Taylor & Francis Inc, 2013.
- [Rod92] John F. Roddick. Schema Evolution in Database Systems: An Annotated Bibliography. *ACM SIGMOD Record*, 21(4):35–40, December 1992.
- [Rod95] John F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383 – 393, 1995.
- [Roy70] Walker W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. *Proceedings IEEE WESTCON*, pages 1–9, August 1970. Reprinted in *Proceedings of the Ninth International Conference on Software Engineering*, March 1987, pp. 328–338.
- [RR15] Alexander Roth and Bernhard Rumpe. Towards Product Lining Model-Driven Development Code Generators. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, MODELSWARD '15*, pages 539–545, Angers, France, 2015. SciTePress.
- [Rum11] Bernhard Rumpe. *Modellierung mit UML*. Xpert.press. Springer Berlin, 2nd edition edition, September 2011.
- [Rum12] Bernhard Rumpe. *Agile Modellierung mit UML*. Springer Berlin, 2nd edition, March 2012.
- [RY98] Eric S. Ristad and Peter N. Yianilos. Learning String-Edit Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, May 1998.
- [SB02] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.
- [SB12] Lakshitha de Silva and Dharini Balasubramaniam. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software*, 85(1):132–151, January 2012.

- [SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P*. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, 2012.
- [SiD] SiDiff Homepage. <http://pi.informatik.uni-siegen.de/Projekte/sidiff>. (Stand: 24.06.2016).
- [SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, pages 167–176, New York, NY, USA, 2005. ACM.
- [SKS13] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proceedings of the International Symposium on Database Programming Languages, DBPL '13*, 2013.
- [SKS15] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Cleager: Eager Schema Evolution in NoSQL Document Stores. In *Proceedings Datenbanksysteme für Business, Technologie und Web, BTW '15*, pages 659–662, 2015.
- [SLT⁺10] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. Change Impact Analysis Based on a Taxonomy of Change Types. In *Proceedings Annual Computer Software and Applications Conference, COMPSAC '10*, pages 373–382, 2010.
- [Smi15] Ben Smith. *Beginning JSON*. Apress, February 2015.
- [Spo] Spoofox Homepage. <http://www.springsource.org>. (Stand: 24.06.2016).
- [SRC⁺12a] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *STTT*, 2012.
- [SRC12b] Matthew Stephan and James R. Cordy. A Survey of Methods and Applications of Model Comparison. Technical Report 2011-582 Rev. 3, School of Computing, Queen's University, June 2012.
- [SRC13] Matthew Stephan and James R. Cordy. A Survey of Model Comparison Approaches and Applications. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development, Modelsward '13*, pages 265–277, Barcelona, Spain, 2013. Scitepress.

- [SVC06] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
- [SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt, 2 edition, 2007.
- [SWC05] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of Existing Safeguarding Techniques for Automatically Generated Code. In *Proceedings of the International Workshop on Software Engineering for Automotive Systems*, SEAS '05, pages 1–6, New York, NY, USA, 2005. ACM.
- [SZ10] Jörg Schäuuffele and Thomas Zurawka. *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Vieweg+Teubner Verlag, 4 edition, 2010.
- [SZN04] Christian Schneider, Albert Zündorf, and Jörg Niere. CoObRA - A Small Step for Development Tools to Collaborative Environments. In *Workshop on Directions in Software Engineering Environments*, 2004.
- [TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference Computation of Large Models. In *Proceedings of the Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium On The Foundations Of Software Engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM.
- [TM10] Dalila Tamzalit and Tom Mens. Guiding Architectural Restructuring through Architectural Styles. In *International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 69–78. IEEE, 2010.
- [TTR⁺11] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Preliminary Findings from a Survey on the MD State of the Practice. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, pages 372–375, Sept 2011.
- [TTT⁺12] Federico Tomassetti, Marco Torchiano, Alessandro Tiso, Filippo Ricca Ricca, and Gianna Reggio. Maturity of Software Modelling and Model Driven Engineering: A Survey in The Italian Industry. In *Proceedings of the International Conference on Evaluation Assessment in Software Engineering*, EASE '12, pages 91–100. IET, May 2012.
- [TV08] Ricardo Terra and Marco T. Valente. Towards a Dependency Constraint Language to Manage Software Architectures. In *Proceedings of the European conference on Software Architecture*, ECSA '08, pages 256–263, Berlin, Heidelberg, 2008. Springer-Verlag.

- [TV09] Ricardo Terra and Marco T. Valente. A Dependency Constraint Language to Manage Object-oriented Software Architectures. *Software — Practice & Experience*, 39(12):1073–1094, August 2009.
- [TVCB12] Ricardo Terra, Marco T. Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. Recommending Refactorings to Reverse Software Architecture Erosion. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 335–340, Washington, DC, USA, 2012. IEEE Computer Society.
- [VEBHH11] Jo Van Eyck, Nelis Boucké, Alexander Helleboogh, and Tom Holvoet. Using Code Analysis Tools for Architectural Conformance Checking. In *Proceedings of the International Workshop on SHaring and Reusing Architectural Knowledge*, SHARK '11, pages 53–54, New York, NY, USA, 2011. ACM.
- [VG07a] Markus Voelter and Iris Groher. Handling Variability in Model Transformations and Generators. In *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling*, DSM '07, 2007.
- [VG07b] Markus Voelter and Iris Groher. Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the International Software Product Line Conference*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.
- [VGG08] Mathieu Verbaere, Michael W. Godfrey, and Tudor Gîrba. Query Technologies and Applications for Program Comprehension. In *Proceedings of the International Conference on Program Comprehension*, ICPC '08, pages 285–288. IEEE, June 2008.
- [vKG03] Antje von Knethen and Mathias Grund. QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 246–255. IEEE, Sept 2003.
- [Vog15] Lars Vogel. *Eclipse Rich Client Platform*. Lars Vogel, 3rd edition, 2015.
- [VV08] Sander D. Vermolen and Eelco Visser. Heterogeneous Coupled Evolution of Software Languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS '08, pages 630–644, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [VWV11] Sander D. Vermolen, Guido Wachsmuth, and Eelco Visser. Generating Database Migrations for Evolving Web Applications. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 83–92, New York, NY, USA, 2011. ACM.

- [Wac07] Guido Wachsmuth. *European Conference on Object-Oriented Programming*, chapter Metamodel Adaptation and Model Co-adaptation, pages 600–624. ECOOP '07. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik Berichte, Software Engineering. Shaker Verlag, 2012.
- [WHR14] John Whittle, Jon Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, May 2014.
- [Wil03] David S. Wile. Lessons Learned from Real DSL Experiments. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, volume 9, page 325.2, Washington, DC, USA, 2003. IEEE Computer Society.
- [WK08] Sven Wenzel and Udo Kelter. Analyzing Model Evolution. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 831–834, New York, NY, USA, 2008. ACM.
- [WKS⁺10] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. On using Inplace Transformations for Model Co-evolution. In *Proceedings of the International Workshop on Model Transformation with ATL, MtATL'10*, pages 65–78, 2010.
- [Wor] WordNet Lexical Database Homepage. <http://wordnet.princeton.edu/>. (Stand: 24.06.2016).
- [XCN09] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software. In *Proceedings of the International Conference on Software Maintenance*, ICSM '09, pages 51–60. IEEE, 2009.
- [XPa] XPand Homepage. <https://eclipse.org/modeling/m2t/?project=xpand>. (Stand: 24.06.2016).
- [XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.
- [Xtea] Xtend Homepage. <http://www.eclipse.org/xtend/>. (Stand: 24.06.2016).
- [Xteb] Xtext Homepage. <http://www.eclipse.org/Xtext>. (Stand: 24.06.2016).

- [ZHJ04] Tewfik Ziadi, Loïc H  lou  t, and Jean-Marc Jezequel. Towards a UML Profile for Software Product Lines. In Frank J. Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer Berlin Heidelberg, 2004.
- [ZS14] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. Vieweg, 5 edition, 2014.

Abbildungsverzeichnis

2.1	Zusammenhang zwischen Authoring, WTS, Diagnosegeräten und dem Fahrzeug in der Werkstatt.	10
2.2	Klassisches Generation Gap Muster [GHK ⁺ 15a, GHK ⁺ 15b] zur Integration von handgeschriebenem und generierten Code.	13
2.3	Klassisches Generation Gap Muster [GHK ⁺ 15a, GHK ⁺ 15b] zur Integration von handgeschriebenem und generierten Code. Die Klasse <code>XImpl</code> hängt von dem generierten Datenbankskript <code>tableX.sql</code> ab.	14
2.4	Kommunikation zwischen Client (Version v_a) und Server (Version v_n). Variante 1 (Soll-Situation): Server erkennt Umbenennung und übernimmt Daten aus umbenanntem Attribut. Variante 2 (Ist-Situation): Server erkennt Umbenennung nicht und verliert Daten zu Attribut-Wert.	17
3.1	Modellverarbeitung durch Lexer und Parser.	23
4.1	Vereinfachter Ausschnitt aus dem ECore-Metamodell [Ecl].	28
4.2	Überblick über die Schritte zur Identifikation von Modellunterschieden für zwei gegebene Modelle.	32
4.3	Auszug aus einem Klassendiagramm für ein Bibliothekssystem.	36
4.4	Ausschnitt aus dem ECore-Modell zum Klassendiagramm in Abbildung 4.3. . .	37
4.5	Metamodell für die Matchmodelle dieser Arbeit.	37
4.6	Klassendiagramm für ein Bibliothekssystem (ältere Version v_a).	42
4.7	Klassendiagramm für ein Bibliothekssystem (neuere Version v_n).	42
4.8	Matchmodell für die Modelle aus Abbildung 4.6 und Abbildung 4.7.	43
4.12	Metamodell für die Differenzmodelle dieser Arbeit.	52
4.13	Differenzmodell für die Modelle aus Abbildung 4.6 und Abbildung 4.7 (auf Seite 42.	54
5.2	Überblick über den Ansatz zur Generierung von Checklisten.	61
5.5	Überblick über die Basisklasse für alle generierten Implementierungen von Auswirkungsregeln.	70
5.16	Beispiel für ein Szenario, in dem eine Klasse durch eine Modelländerung über verschiedene Pfade von einer anderen Klasse aus erreichbar ist.	87
6.1	Beispielhaftes Datenmodell (linke Seite), beispielhaftes Objektdiagramm zu dem Datenmodell (Mitte) und JSON-Dokument konform zu diesem Objektdiagramm (rechte Seite).	103
6.2	Auswirkungen der Umbenennung des Attributs h der Klasse $x.y.G$ in $hNew$. .	105

6.3	Verwendung der Datenmigrationsfunktionalität in der Client/Server-Kommunikation. Clients in verschiedenen Versionen können mit dem gleichen Server kommunizieren.	106
6.4	Beispielhaftes Klassendiagramm zur Bestimmung von Pfaden zur Klasse D. . .	111
6.5	Beispielhaftes Klassendiagramm zur Bestimmung von Pfaden zur Klasse D - unter Beachtung von Zyklen.	112
6.6	Beispielhaftes Klassendiagramm zur Bestimmung von Pfaden zur Klasse E unter Betrachtung von Vererbung.	114
6.7	Graph der zu den Pfaden aus Abbildung 6.6 erzeugt wurde.	116
6.8	Klassendiagramm für ein Bibliothekssystem (ältere Version a).	117
6.9	Klassendiagramm für ein Bibliothekssystem (neuere Version n).	117
6.10	Verschiebung eines Attributs in eine Oberklasse der Quellklasse.	118
6.11	Verschiebung eines Attributs in eine Zielklasse, die unabhängig von der Quellklasse ist.	119
6.12	Änderung der Quellklasse einer Assoziation, so dass die neue Quellklasse eine Oberklasse der alten Quellklasse ist.	120
6.13	Änderung der Quellklasse einer Assoziation, so dass die neue Quellklasse in keiner Beziehung zu der alten Quellklasse steht.	121
6.14	Änderung der Zielklasse einer Assoziation, so dass die neue Zielklasse eine Oberklasse der alten Zielklasse ist.	121
6.15	Änderung der Zielklasse einer Assoziation, so dass die neue Zielklasse keine Oberklasse oder Unterklasse der alten Zielklasse ist.	122
6.16	Verschiebung eines Literals aus einer Enumeration.	122
6.17	Hinzufügen einer Vererbungsbeziehung.	125
6.18	Datenmigration nach dem Hinzufügen eines Attributs.	128
6.19	Datenmigration nach der Umbenennung eines Attributs.	130
6.20	Datenmigration nach der Verschiebung eines Attributs. Die Zielklasse der Verschiebung ist von der Quellklasse der Verschiebung ausgehend erreichbar. . . .	131
6.21	Datenmigration nach der Verschiebung eines Attributs. Die Quellklasse der Verschiebung ist von der Zielklasse der Verschiebung ausgehend erreichbar. . . .	132
6.22	Datenmigration nach der Veränderung der Quellklasse einer Assoziation. Die alte Quellklasse ist von der neuen Quellklasse ausgehend erreichbar.	134
6.23	Datenmigration nach der Veränderung der Quellklasse einer Assoziation. Die neue Quellklasse ist von der alten Quellklasse ausgehend erreichbar.	136
6.24	Veränderung der Quellklasse einer Assoziation. Die Zielklasse, der von der Änderung betroffenen Assoziation, liegt auf dem Pfad von der alten Quellklasse zu der neuen Quellklasse.	136
6.25	Datenmigration nach der Veränderung der Zielklasse einer Assoziation. . . .	138
6.26	Datenmigration nach der Veränderung der Kardinalität eines Assoziationsendes von einwertig zu mehrwertig.	139
6.27	Datenmigration nach dem Löschen eines Literals einer Enumeration.	141
6.28	Datenmigration nach der Umbenennung eines Literals einer Enumeration. . . .	142
6.29	Datenmigration nach dem Löschen einer Klasse.	144

6.30	Datenmigration nach der Umbenennung einer Klasse.	145
6.31	Datenmigration nach dem Ändern einer abstrakten Klasse hin zu einer nicht-abstrakten Klasse.	147
6.32	Datenmigration nach dem Löschen einer Vererbungsbeziehung.	148
6.33	Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Unterklasse von neuer Oberklasse).	150
6.34	Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Oberklasse von neuer Oberklasse).	150
6.35	Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Unterklasse von neuer Oberklasse).	150
6.36	Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Oberklasse von neuer Oberklasse).	150
6.37	Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Unterklasse von neuer Oberklasse).	151
6.38	Veränderung der Oberklasse einer Klasse (alte Oberklasse ist Oberklasse von neuer Oberklasse).	151
6.39	Beispiel zur Minimierung von Graphen. Die Knoten B und C können aufgrund des Löschens der Kante b ebenfalls entfernt werden.	154
6.40	Minimierung von Graphen. Der auf den gelöschten Knoten folgende Knoten kann nicht gelöscht werden.	155
6.43	Graph in dem die Migrationslogik für die zu Knoten C hinterlegten Modelländerungen nicht in beliebiger Reihenfolge ausgeführt werden können.	159
6.44	Graph für die Verschiebung des Attributs name von Klasse B in die Klasse D. .	163
6.47	Graph wird angepasst, da die Quellklasse der Assoziation e von der Klasse C in die Klasse A geändert wurde.	165
6.48	Generierte Migrationsklasse verwendet handgeschriebene Erweiterungsklasse zur Durchführung der Datenmigration.	166
7.1	Beispiel für drei FOP-Schichten L_1 bis L_3 mit vier Kompositionsoperationen. .	176
7.9	Auswirkungen der Kompositionsoperationen <i>before</i> , <i>replace</i> und <i>after</i>	186
7.10	GRV C wird samt der GRs die vor und hinter die GRV eingefügt wurden (B und A) zu neuem Artefakt hinzugefügt.	187
7.12	Mehrfache Anwendung der Kompositionsoperation <i>replace</i> für mehrere GRVs mit der gleichen Signatur wie die ersetzte VR.	189
7.18	Manuelle Selektion zweier CGLs. Der Knoten C_3^1 enthält zwei eingehende Kanten, so dass die Validierung einen Fehler meldet.	195
7.19	Manuelle Selektion eines CGLs. Der resultierende Graph enthält einen Zyklus, da ein Pfad (C_1^3, C_1^1) und ein Pfad (C_1^1, C_1^3) existiert.	196
7.20	Komposition zur Laufzeit des Generators, ohne dass neue Generatorartefakte hierfür erzeugt werden.	197
7.21	Komposition durch die Erzeugung neuer Artefakte. Die ursprüngliche VR wird in einem neuen Artefakt durch eine andere GRV ersetzt.	197

7.22	Komposition durch die Erzeugung neuer Artefakte. Eine GR wird vor einer VR aufgerufen, um die <i>before</i> Kompositionsoperation abzubilden.	198
7.23	Grafische Repräsentation eines Schichtdefinitionsmodells, welches vier CGLs umfasst, in denen Kompositionsoperationen definiert sind.	199
7.24	Graph zu dem Schichtdefinitionsmodell aus Abbildung 7.23 auf Seite 199.	204
7.25	Graph nach Betrachtung des Knotens <i>A.B:D</i>	204
7.26	Graph nach Betrachtung des Knotens <i>H.I:J</i>	205
7.27	Graph nach Betrachtung des Knotens <i>H.I:L</i>	205
7.28	Handhabung von <i>super</i> Methodenaufrufen bei der Komposition von Java Hilfsklassen.	209
7.29	Inhalt der Generatorartefakte für die Generatorvariante, die auf Basis des Schichtdefinitionsmodells aus Abbildung 7.23 erzeugt wurde.	211
7.30	Inhalt der Datenstruktur <i>adaptedReplacementMap</i> vor Anwendung der Ableitungsvorschrift.	214
7.31	Inhalt der Datenstruktur <i>adaptedReplacementMap</i> nach Anwendung der Ableitungsvorschrift.	214
7.32	Inhalt der Generatorartefakte nach der Adaption der Artefakte aus Abbildung 7.29.216	
8.1	Überblick über den Zusammenhang zwischen Plugin-basierten Softwaresystemen, Features, Plugins, Feature-Gruppen sowie Plugin-Gruppen.	227
8.2	Objektdiagramm zur Verdeutlichung von Abhängigkeiten zwischen Features, Plugins und Feature-Gruppen.	229
8.3	Überblick über den Prozess zur Prüfung der Architekturkonformität.	230
8.4	Eclipse-Ansicht zur Darstellung der Ergebnisse der Architekturkonformitätsprüfung.	232
8.15	Überblick über die Features und Plugins eines fiktiven Plugin-basierten Softwaresystems.	250

Listings

3.1	Beispiel für eine Xtext-Grammatik.	24
3.2	Lexerregel ID der Xtext-Grammatik <code>org.eclipse.xtext.common.Terminals</code>	25
3.3	Beispielhaftes Modell, welches konform zu der Grammatik aus Listing 3.1 ist. .	26
4.1	Delta-Benutzervorgaben in Form einer Abbildungsvorschrift.	45
4.2	Grammatik der DSL DUPS.	47
4.3	Spezifikation von Modelländerungen für Delta-Benutzervorgaben.	48
4.4	Spezifikation einer Delta-Operation zur Umbenennung eines Attributs.	49
5.1	Vereinfachte Modellierung der Auswirkungsregel ORM-AR.	58
5.2	Beispielhafte Checkliste unter Nutzung der Auswirkungsregel ORM-AR. . . .	60
5.3	Teil der Grammatik der DSL IRS, in dem eine Auswirkungsregel allgemein beschrieben wird.	64
5.4	Teil der Grammatik der DSL IRS, über den die Definition der Bedingungen einer Auswirkungsregel erfolgt.	65
5.5	Modellierung der Auswirkungsregel XMLMig-AR.	74
5.6	Generierte Implementierung der Auswirkungsregel XMLMig-AR (siehe Listing 5.5 auf Seite 74) mit Methoden zu allgemeinen Informationen, Platzhaltern sowie benutzerdefinierten Bedingungen.	76
5.7	Generierte Implementierung der Auswirkungsregel XMLMig-AR (siehe Listing 5.5 auf Seite 74) zur Prüfung der Bedingungen der Auswirkungsregel. . . .	77
5.8	Implementierung der Methode <code>createOutputForGroup()</code>	78
5.9	Modellierung der Auswirkungsregel ORM-AR.	80
5.10	Modellierung der Auswirkungsregel DBMig-AR.	82
5.11	Modellierung der allgemeinen Informationen zur Auswirkungsregel Prop-AR und Definition der Auswirkungen von Änderungen, die Klassen betreffen. . . .	83
5.12	Modellierung der Auswirkungsregel Prop-AR (siehe Listing 5.11). Definition der Auswirkungen von Änderungen, die Attribute betreffen.	84
5.13	Modellierung der Auswirkungsregel Prop-AR (siehe Listing 5.11). Definition der Auswirkungen von Änderungen, die Assoziationen betreffen.	85
5.14	Modellierung der Auswirkungsregel BR-AR.	86
5.15	Auszug aus einem fiktiven Changelog. Der Changelog weist auf drei Modelländerungen hin, die von einem Modellierer durchgeführt wurden.	87
5.16	Modellierung der Auswirkungsregel Changelog-AR.	88
6.1	Generat mit Traversierungslogik zu Abbildung 6.6 bzw. Abbildung 6.7.	157

6.2	Generat mit Datenmigrationslogik zur Handhabung der Umbenennung des Attributs <code>name</code> nach <code>newName</code> für den Knoten E.	158
6.3	Generierte Datenmigrationslogik für die Verschiebung des Attributs von der primären Klasse B in die Zielklasse D.	163
6.4	Generierte Datenmigrationslogik für die Verschiebung des Attributs von der Quellklasse D in die Zielklasse B.	164
7.1	Template <code>Class.tpl</code> zur Generierung des Grundgerüsts einer Java-Klasse.	178
7.2	Basisvariante der Java-Klasse <code>Library</code> mit einem Attribut, einem öffentlichen Konstruktor sowie einer <code>set</code> -Zugriffsmethode.	179
7.3	Die Metamodellklasse <code>MMClass.java</code> kapselt Hilfsfunktionalität, die innerhalb von Templates aufgerufen werden kann.	180
7.4	Template <code>Attribute.tpl</code> zur Generierung der Attribute für eine Klasse.	180
7.5	Template <code>Method.tpl</code> zur Generierung der Methoden für eine Klasse.	181
7.6	Erweiterte Variante der Java-Klasse mit einem Attribut, einer <code>set</code> -Zugriffsmethode sowie einer statischen Instanziierungsmethode.	182
7.7	Alternative Template-Implementierung für die Generierung einer Methode mit Methodenkörper unter Nutzung von Kommentaren zur Kennzeichnung von GRs.	185
7.8	Grammatik zur DSL LDL.	188
7.9	Das Template <code>Methods</code> für die angepasste Generierung von <code>set</code> -Methoden und die Generierung einer statischen Instanziierungsmethode.	190
7.10	Die Hilfsklasse <code>MMClassFactoryVar</code> , welche das Klassen-Suffix <code>Impl</code> und eine <code>protected</code> Sichtbarkeit für Konstruktoren festlegt.	190
7.11	Schichtdefinitionsmodell für das Beispiel aus Abschnitt 7.2.	191
7.12	Grammatik zur DSL PCL.	192
7.13	Produktkonfigurationsmodell durch das der CGL <code>factoryVariant</code> ausgewählt wird.	192
7.14	Erweiterung der Templates um eine Anweisung zur Inkludierung des Inhalts der ersetzten VR.	222
8.1	Grammatik der DSL <code>DepCoL</code> . Ein Abhängigkeitsmodell kann eine Feature- und Plugin-Basis, Deklarationen von Feature- und Plugin-Gruppen sowie Definitionen von Abhängigkeitsrestriktionen enthalten.	233
8.2	Grammatik der DSL <code>DepCoL</code> zur Definition einer Feature und Plugin-Basis.	234
8.3	Definition einer Feature-Basis welche die Features <code>f1</code> , <code>f2</code> und <code>f3</code> enthält.	234
8.4	Grammatik der DSL <code>DepCoL</code> zur Definition von Feature- und Plugin-Gruppen.	235
8.5	Beispiele für Feature-Gruppen Definitionen.	237
8.6	Grammatik der DSL <code>DepCoL</code> , die zeigt, dass Abhängigkeitsrestriktionen für Features, Plugins, Feature-Gruppen und Plugin-Gruppen definiert werden können.	238
8.7	Grammatik der DSL <code>DepCoL</code> zur Definition von Abhängigkeitsrestriktionen. Es können Abhängigkeitsrestriktionen hin zu Features, Feature-Gruppen, Plugins sowie Plugin-Gruppen definiert werden.	240
8.8	Beispielhafte Abhängigkeitsrestriktionen für verbotene und tolerierte Abhängigkeiten.	242

8.9	Grammatik der DSL DepCoL zur Spezifikation von Abhängigkeitsrestriktionen zu Feature- oder Plugin-Mustern.	242
8.10	Beispielhafte Spezifikation von Abhängigkeitsrestriktionen unter Nutzung von Feature-Mustern und Plugin-Mustern.	242
8.11	Beispielhaftes fiktives Abhängigkeitsmodell mit überschriebenen Abhängigkeitsrestriktionen.	251
8.12	Abhängigkeitsmodell durch das sämtliche Abhängigkeiten standardmäßig verboten werden.	257
C.1	Beispielhafte Checkliste unter Nutzung der Auswirkungsregeln XMLMig-AR, ORM-AR, Prop-AR, DBMig-AR, BR-AR und Changelog-AR.	315
E.1	Grammatik der DSL DUPS.	323
E.2	Grammatik der DSL IRS.	324
E.3	Grammatik der DSL LDL.	325
E.4	Grammatik der DSL PCL.	326
E.5	Grammatik der DSL DepCoL.	326
G.1	Antworten des ersten Entwicklers zum Fragebogen aus Abschnitt 5.7.	337
G.2	Antworten des zweiten Entwicklers zum Fragebogen aus Abschnitt 5.7.	337
G.3	Antworten des dritten Entwicklers zum Fragebogen aus Abschnitt 5.7.	338
G.4	Antworten des vierten Entwicklers zum Fragebogen aus Abschnitt 5.7.	338
G.5	Antworten des fünften Entwicklers zum Fragebogen aus Abschnitt 5.7.	338
G.6	Antworten zu der ersten Frage der Fallstudie aus Abschnitt 8.9.3.	339
G.7	Antworten zu der zweiten Frage der Fallstudie aus Abschnitt 8.9.3.	339
G.8	Antworten zu der dritten Frage der Fallstudie aus Abschnitt 8.9.3.	340
G.9	Offenes Feedback zur Fallstudie aus Abschnitt 8.9.3.	340

Tabellenverzeichnis

6.49	Übersicht über die Ausführungszeit der Datenmigrationen mit und ohne Steuerung der Traversierung.	173
6.50	Übersicht über die Anzahl der Traversierungsmethoden und Migrationsmethoden mit und ohne Minimierung der Graphen.	173
7.33	Überblick über die Anzahl der Kompositionsoperationen zur Umsetzung der beiden Generatorvarianten.	221
7.35	Ergebnisse der Fallstudie: TLOC und HLOC für die verschiedenen Varianten. .	223
8.17	Übersicht über die Ergebnisse zur Prüfung der Architekturkonformität.	254
8.19	Überblick über die Komplexität der verschiedenen Versionen des Abhängigkeitsmodells.	263
8.20	Übersicht über die gemeldeten Verstöße in der Beobachtungsphase.	265
8.21	Übersicht über die gemeldeten Verstöße in der Anwendungsphase.	266
8.22	Übersicht über die Bewertung der Entwickler zu den Fragen DQ1 und DQ2. . .	268

Liste der Algorithmen

4.1	Function matchModels()	38
4.2	Function mapLists()	39
4.3	Function calculateSimilarity()	40
4.4	Function recursiveMappings()	41
4.5	Function matchModels() - Um Delta-Benutzervorgaben erweitert	50
4.6	Function mapLists() - Um Delta-Benutzervorgaben erweitert	51
5.1	Function generateChecklist()	63
6.1	Function generateMigrationFunctionalityVersions()	108
6.2	Function findPathsToPrimaryClasses()	109
6.3	Function traverseAssociations()	110
6.4	Function traverseAssociations() - Second part	113
6.5	Function createPathEntry()	113
7.1	Function createCompositionGraph()	194
7.2	Function determineCompositionOperationConsequences()	201
7.3	Function traverseEdgesOfNode()	202
7.4	Function handleSuperInvocations()	208
7.5	Function replaceReferencedNamesByNewNames()	213
8.1	Function checkArchitectureConformance()	246

Anhang A

Abkürzungsverzeichnis

AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
CGL	Code Generator Layer
CI	Continuous Integration
DAO	Data Access Object
DCL	Dependency Constraint Language
DepCoL	Dependency Constraint Language
DSL	Domain Specific Language
DSM	Dependency Structure Matrix
DUPS	Delta User Presetting Specification
EMF	Eclipse Modeling Framework
FOP	Feature-Oriented Programming
GPL	General Purpose Language
GR	Generator Region
GRV	Generator Region Variant
IRS	Impact Rule Specification
JSON	Java Script Object Notation
MDI	Mobile Diagnostic Interface
LDL	Layer Definition Language
OCL	Object Constraint Language
OMG	Object Management Group
ORM	Object Relational Mapping
PCL	Product Configuration Language
RCP	Rich Client Platform
SIB	Service Independent Building Block
SPL	Software Product Line
UI	User Interface
UML	Unified Modeling Language
VR	Variability Region
WDI	Wireless Diagnostic Interface
WLAN	Wireless Local Area Network
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Anhang B

Notationen

Im Folgenden sind die Notationen dargelegt, die in Abbildungen und Listings zur Kennzeichnung der zugrundeliegenden Sprache oder für Anmerkungen verwendet werden.

	Klassendiagramm
	Delta-Modell
	Modell der DSL DepCoL
	Modell der DSL DUPS
	Modell der DSL IRS
	Java-Datei
	JSON-Datei
	Modell der DSL LDL
	Objektdiagramm
	Modell der DSL PCL
	Xpand-Template
	Xtext-Grammatik
	Kennzeichnet einen generierten Teil eines Systems
	Kennzeichnet einen handgeschriebenen Teil eines Systems
	Kommentar mit Verweis auf das kommentierte Element
	Erkennungsmerkmal einer unvollständigen Abbildung oder eines unvollständigen Listings

Anhang C

Checkliste

```
1 Creation time: 06.12.2016, 14:53
2 Old model: library, revision 4
3 New model: library, revision 5
4
5 Found model differences:
6 -----
7 - Deleted attribute 'core.Library#name' (type: String)
8 - Renamed attribute 'core.Book#title' (type: String) to 'fullTitle'
9 - Added attribute 'core.Book#year' (type: int)
10 - Moved attribute 'core.BookExemplar#isbn' to class 'core.Book'
11 - Deleted class 'ext.Address'
12
13 Used impact rules:
14 -----
15 - XML Migration Analysis
16 - ORM File Analysis
17 - Database Data Migration Analysis
18 - Property File Analysis
19 - Business Rule Analysis
20 - Changelog Analysis
21
22 XML Migration Analysis:
23 =====
24 This rule checks whether it is necessary to implement XML
25 migration classes.
26 Severity: critical
27 Probability: high
28
29 - Implement XML migration class for class 'core.Library':
30   - Write migration to delete XML element 'name'. (Reason: Deleted
31     attribute 'core.Library#name' (type: String)
32   - Write migration to rename XML element 'title' to 'fullTitle'
33     (Reason: Renamed attribute 'core.Book#title' (type: String) to
34     'fullTitle')
35   - Write migration to move XML element 'BookExemplar.isbn' to
36     'Book'. (Reason: Moved attribute 'core.BookExemplar#isbn'
37     to class 'core.Book')
38   - Write migration to delete XML element 'Address'.
```

```

39   (Reason: Deleted class 'ext.Address')
40
41   Proposed XML migration class:
42   public class ConvertLibrary implements IConverter {
43       public void convert(Document pDocument) {
44           //Implementation not shown here
45       }
46   }
47
48   ORM File Analysis:
49   =====
50   This rule checks whether it is necessary to change an object
51   relational mapping file, by adding entries, changing existing
52   entries or deleting existing entries.
53   Severity: critical
54   Probability: high
55
56   Entries concerning class 'core.Library':
57   - Delete entry from mapping file. (Reason: Deleted attribute
58     'core.Library#name' (type: String))
59
60   Entries concerning class 'core.Book':
61   - Rename old entry 'title' to 'fullTitle' in mapping file.
62     (Reason: Renamed attribute 'core.Book#title' (type: String)
63     to 'fullTitle')
64   - Add new entry to mapping file. (Reason: Added attribute
65     'core.Book#year' (type: int))
66
67   Entries concerning class 'core.BookExemplar':
68   - Move entry 'isbn' to new location 'core.Book' in mapping file.
69     (Reason: Moved attribute 'core.BookExemplar#isbn' to class
70     'core.Book')
71
72   Entries concerning class 'ext.Address':
73   - Delete entry from mapping file. (Reason: Deleted class
74     'ext.Address')
75
76   Database Data Migration Analysis:
77   =====
78   This rule checks whether it is necessary to implement data
79   migration scripts for the database.
80   Severity: critical
81   Probability: high
82
83   Migrations concerning table 'TBook':
84   - Write data migration to rename column 'title' to 'fullTitle'.
85     (Reason: Renamed attribute 'core.Book#title' (type: String) to
86     'fullTitle')
87

```

```

88 Migrations concerning table 'TBookExemplar':
89   - Write data migration to move column content 'isbn' from table
90   'TBookExemplar' to table 'TBook'. (Reason: Moved attribute
91   'core.BookExemplar#isbn' to class 'core.Book')
92
93 Property File Analysis:
94 =====
95 This rule checks in which way property files have to be changed.
96 This can comprise adding new entries, changing existing entries or
97 deleting existing entries.
98 Severity: minor
99 Probability: high
100
101 - Changes in property file core.properties:
102   - Remove field.Library.name. (Reason: Deleted attribute
103   'core.Library#name' (type: String))
104   - Add field.Book.fullTitle. Check whether entry field.Book.title
105   already exists. You might take the associated value for the new
106   entry. (Reason: Renamed attribute 'core.Book#title' (type: String)
107   to 'fullTitle')
108   - Add field.Book.year. (Reason: Added attribute
109   'core.Book#year' (type: int))
110   - Add field.BookExemplar.isbn. Check whether entry field.
111   Book.isbn already exists. You might take
112   the associated value for the new entry.
113   (Reason: Moved attribute 'core.BookExemplar#isbn' to class
114   'core.Book')
115   - Remove entity.typeName.singular.Address and entity.typeName.
116   plural.Address. (Reason: Deleted class 'ext.Address')
117
118 Business Rule Analysis:
119 =====
120 This rule checks whether it might be required to add new or adapt
121 existing BR's.
122 Probability: low
123 Relevant for: ab
124
125 - Check whether BR should be added which checks whether values
126 of added attribute should be restricted. (Added attribute
127 'core.Book#year' (type: int))
128 - Check whether BR exists concerning moved attribute. In this case,
129 the BR needs to be adapted to new class of the attribute.
130 (Reason: Moved attribute 'core.BookExemplar#isbn' to class
131 'core.Book')
132
133 Changelog Analysis:
134 =====
135 This rule checks whether new entries have to be added to the
136 changelog file. Take the following textual summary of the

```

```
137 identified model differences and copy it to the changelog file.  
138 Severity: minor  
139 Probability: high  
140  
141 - Deleted attribute 'core.Library#name' (type: String)  
142 - Renamed attribute 'core.Book#title' (type: String) to 'fullTitle'  
143 - Added attribute 'core.Book#year' (type: int)  
144 - Moved attribute 'core.BookExemplar#isbn' to class 'core.Book'  
145 - Deleted class 'ext.Address'
```

Listing C.1: Beispielhafte Checkliste unter Nutzung der Auswirkungsregeln XMLMig-AR, ORM-AR, Prop-AR, DBMig-AR, BR-AR und Changelog-AR.

Anhang D

Vordefinierte Bedingungen der DSL IRS

- Prüfung auf Modelländerungen bezüglich von Modellelementen allgemein (Pakete, Klassen, Assoziationen oder Attributen):
 - `addedModelElement()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen eines Modellelements repräsentiert.
 - `deletedModelElement()`: Liefert `true`, wenn die Modelländerung das Löschen eines Modellelements darstellt.
 - `deletedStereotype()`: Gibt `true` zurück, insofern die Modelländerung das Löschen eines Stereotyps repräsentiert.
 - `addedStereotype()`: Liefert `true`, falls die Modelländerung das Hinzufügen eines Stereotyps ausdrückt.
 - `changedStereotype()`: Gibt `true` zurück, wenn die Modelländerung die Veränderung eines Stereotyps repräsentiert.
 - `deletedTaggedValue()`: Liefert `true`, falls die Modelländerung das Löschen eines Tagged Values darstellt.
 - `addedTaggedValue()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen eines Tagged Values ausdrückt.
 - `changedTaggedValue()`: Liefert `true`, falls die Modelländerung die Veränderung eines Tagged Values repräsentiert.
- Prüfung auf Modelländerungen bezüglich von Paketen:
 - `addedPackage()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen eines Pakets repräsentiert.
 - `deletedPackage()`: Gibt `true` zurück, falls die Modelländerung das Löschen eines Pakets darstellt.
 - `renamedPackage()`: Liefert `true`, falls die Modelländerung das Umbenennen eines Pakets repräsentiert.
 - `movedPackage()`: Gibt `true` zurück, falls die Modelländerung das Verschieben eines Pakets darstellt.
- Prüfung auf Modelländerungen bezüglich von Klassen:

- `addedClass()`: Gibt `true` zurück, falls die Modelländerung das Hinzufügen einer Klasse repräsentiert.
 - `deletedClass()`: Gibt `true` zurück, falls die Modelländerung das Löschen einer Klasse darstellt.
 - `renamedClass()`: Liefert `true`, falls die Modelländerung das Umbenennen einer Klasse repräsentiert.
 - `movedClass()`: Gibt `true` zurück, falls die Modelländerung das Verschieben einer Klasse darstellt.
 - `changedToAbstractClass()`: Gibt `true` zurück, falls die Modelländerung ausdrückt, dass die Klasse von einer nicht-abstrakten zu einer abstrakten Klasse geändert wurde.
 - `changedToNonAbstractClass()`: Gibt `true` zurück, falls die Modelländerung ausdrückt, dass die Klasse von einer abstrakten zu einer nicht-abstrakten Klasse geändert wurde.
 - `deletedSuperType()`: Liefert `true`, falls die Modelländerung ausdrückt, dass die Oberklasse der Klasse entfernt wurde.
 - `addedSuperType()`: Gibt `true` zurück, falls die Modelländerung ausdrückt, dass eine Oberklasse zur Klasse hinzugefügt wurde.
 - `changedSuperTypeToNewSuperType()`: Liefert `true`, falls die Modelländerung ausdrückt, dass die Oberklasse der Klasse von einer Klasse zu einer anderen Klasse geändert wurde.
- Prüfung auf Modelländerungen bezüglich von Assoziationen:
 - `addedAssociation()`: Gibt `true` zurück, insofern die Modelländerung das Hinzufügen einer Assoziation repräsentiert.
 - `deletedAssociation()`: Liefert `true`, falls die Modelländerung das Löschen einer Assoziation repräsentiert.
 - `renamedTgtAssociationEnd()`: Liefert `true`, falls die Modelländerung das Umbenennen des Rollenamens des Zielassoziationsendes der Assoziation darstellt.
 - `changedTgtAssociationEndCardinality()`: Liefert `true`, insofern die Modelländerung die Veränderung der Kardinalität des Zielassoziationsendes der Assoziation darstellt.
 - `changedTgtAssociationEndCardinalityFromOneToStar()`: Liefert `true`, wenn die Modelländerung die Veränderung der Kardinalität des Zielassoziationsendes von einer einwertigen zu einer mehrwertigen Assoziation repräsentiert.
 - `changedTgtAssociationEndCardinalityFromStarToOne()`: Liefert `true`, wenn die Modelländerung die Veränderung der Kardinalität des Zielassoziationsendes von einer mehrwertigen zu einer einwertigen Assoziation ausdrückt.

- `changedSourceClassOfAssociation()`: Gibt `true` zurück, falls die Modelländerung die Änderung der Quellklasse der Assoziation ausdrückt.
- `changedTgtClassOfAssociation()`: Liefert `true`, wenn die Modelländerung die Änderung der Zielklasse der Assoziation ausdrückt.
- Prüfung auf Modelländerungen bezüglich von Attributen:
 - `addedAttribute()`: Gibt `true` zurück, insofern die Modelländerung das Hinzufügen eines Attributs repräsentiert.
 - `deletedAttribute()`: Liefert `true`, wenn die Modelländerung das Löschen eines Attributs repräsentiert.
 - `renamedAttribute()`: Gibt `true` zurück, insofern die Modelländerung das Umbenennen eines Attributs repräsentiert.
 - `movedAttribute()`: Liefert `true`, falls die aktuell untersuchte Modelländerung die Verschiebung eines Attributs darstellt.
 - `changedAttributeCardinality()`: Gibt `true` zurück, falls die Modelländerung das Ändern der Kardinalität eines Attributs ausdrückt.
 - `changedAttributeCardinalityFromStarToOne()`: Liefert `true`, falls die Modelländerung das Ändern der Kardinalität eines Attributs von einem mehrwertigen zu einem einwertigen Attribut ausdrückt.
 - `changedAttributeCardinalityFromOneToStar()`: Gibt `true` zurück, falls die Modelländerung das Ändern der Kardinalität eines Attributs von einem einwertigen zu einem mehrwertigen Attribut ausdrückt.
- Prüfung auf Modelländerungen bezüglich von Enumerationen:
 - `addedEnumeration()`: Gibt `true` zurück, insofern die Modelländerung das Hinzufügen einer Enumeration repräsentiert.
 - `deletedEnumeration()`: Liefert `true`, wenn die Modelländerung das Löschen einer Enumeration repräsentiert.
 - `renamedEnumeration()`: Gibt `true` zurück, insofern die Modelländerung das Umbenennen einer Enumeration repräsentiert.
 - `movedEnumeration()`: Liefert `true`, falls die aktuell untersuchte Modelländerung die Verschiebung einer Enumeration darstellt.
- Prüfung auf Modelländerungen bezüglich von Literalen von Enumerationen:
 - `addedEnumerationLiteral()`: Gibt `true` zurück, insofern die Modelländerung das Hinzufügen eines Literals zu einer Enumeration repräsentiert.
 - `deletedEnumerationLiteral()`: Liefert `true`, wenn die Modelländerung das Löschen eines Literals aus einer Enumeration repräsentiert.
 - `renamedEnumerationLiteral()`: Gibt `true` zurück, insofern die Modelländerung das Umbenennen eines Literals einer Enumeration repräsentiert.

- `movedEnumerationLiteral()`: Liefert `true`, falls die aktuell untersuchte Modelländerung die Verschiebung eines Literals von einer Enumeration in eine andere Enumeration darstellt.

Anhang E

Grammatiken

E.1 Grammatik der DSL DUPS

```
1 grammar rwth.DUPS with org.eclipse.xtext.common.Terminals
2 generate dups "http://www.se.de/rwth/DUPS"
3
4 DeltaUserPresettings:
5     presettings += DeltaUserPresettingSpecification+;
6
7 DeltaUserPresettingSpecification:
8     DeltaUserPresettingAdd | DeltaUserPresettingDelete |
9     DeltaUserPresettingMove | DeltaUserPresettingRename |
10    DeltaUserPresettingMoveAndRename
11 ;
12
13 DeltaUserPresettingAdd:
14     'added' addedElem = ModelElemName';'
15 ;
16
17 DeltaUserPresettingDelete:
18     'deleted' deletedElem = ModelElemName';'
19 ;
20
21 DeltaUserPresettingMove:
22     'moved' movedSrc = ModelElemName ' to '
23     movedTgt = ModelElemName';'
24 ;
25
26 DeltaUserPresettingRename:
27     'renamed' renamedOrg = ModelElemName ' to '
28     renamedNew = ModelElemName';'
29 ;
30
31 DeltaUserPresettingMoveAndRename:
32     'moved and renamed' orgElem = ModelElemName ' to '
33     newElem = ModelElemName';'
34 ;
35
```

Xtext

```

36 ModelElemName:
37   Name* ( '.' Name)* ( '#' Name)?
38 ;
39
40 Name :
41   (ID | '0..9')
42 ;

```

Listing E.1: Grammatik der DSL DUPS.

E.2 Grammatik der DSL IRS

```

1 grammar rwth.IRS with org.eclipse.xtext.common.Terminals
2 generate iRS "http://www.se.de/rwth/IRS"
3
4 ImpactRule:
5   'impactRule ' name = STRING '{'
6   'description ' description = STRING
7   ('severity = ' severity = ('minor' | 'normal' | 'critical'))?
8   ('probability = ' probability = ('low' | 'medium' | 'high'))?
9   ('relevantFor = ' relevantFor = STRING)?
10
11   (impactSpec = ImpactSpecificationBlock)?
12   '}'
13 ;
14
15 ImpactSpecificationBlock:
16   'impact' (incomplete ?= '[incomplete]')? '{'
17   (groupSpec += GroupSpecification)*
18   (impactSpec += ImpactSpecification)+
19   '}'
20 ;
21
22 GroupSpecification:
23   'group ' name = ID STRING ';'
24 ;
25
26 ImpactSpecification:
27   condition = ImpactRuleCondition '=>'
28   ('group ' (abbrGroup = [GroupSpecification | ID] |
29   group = STRING)', hint')? impactDesc = STRING
30 ;
31
32 ImpactRuleCondition:
33   BooleanExpr
34 ;
35
36 BooleanExpr returns ImpactRuleCondition:

```

Xtext

```

37   PrefixedExpr
38   (({AndOrExpr.left=current} op=("|" | "&&")) right=PrefixedExpr)*
39   ;
40
41   PrefixedExpr returns ImpactRuleCondition:
42   {BooleanNegation} =>"!" expr=Atomic | Atomic
43   ;
44
45   Atomic returns ImpactRuleCondition:
46   {ParenthesizedExpr} '(' expr=ImpactRuleCondition ')' |
47   {MethodNameLiteral} value=MethodName
48   ;
49
50   MethodName returns ImpactRuleCondition:
51   ID'()' | 'pc.' ID'()'
52   ;

```

Listing E.2: Grammatik der DSL IRS.

E.3 Grammatik der DSL LDL

```

1  grammar rwth.LDL with org.eclipse.xtext.common.Terminals
2  generate LDL "http://www.se.de/rwth/LDL"
3
4  LayerDefinitionModel:
5    cgl = CGLSpecification+
6    ;
7
8  CGLSpecification:
9    'compose layer ' adaptingLayerName = ID ' with layer '
10     adaptedLayerName = [ID | CGLSpecification] '{'
11     vrCompositions = CompositionOperation+
12     '}'
13    ;
14
15  CompositionOperation:
16    adaptingGR = GRSignature compositionType = CompositionType
17    adaptedVR = GRSignature ';'
18    ;
19
20  CompositionType:
21    replace = 'replace'
22    | extends = 'extends'
23    | before = 'before'
24    | after = 'after'
25    ;
26
27  GRSignature:

```

Xtext

```

28 signature = Name
29 ;
30
31 Name :
32 ID* (('.' | '-' | ':') ID)*
33 ;

```

Listing E.3: Grammatik der DSL LDL.

E.4 Grammatik der DSL PCL

```

1 grammar rwth.PCL with org.eclipse.xtext.common.Terminals
2 generate pCL "http://www.se.de/rwth/PCL"
3
4 productConfigurationModel:
5   'generator ' variantName = ID '{'
6     'layer = ' layer = ID (',' furtherLayers += ID)* ';'
7     ('output = ' output = STRING ';' )?
8   '}'
9 ;

```

Listing E.4: Grammatik der DSL PCL.

E.5 Grammatik der DSL DepCoL

```

1 grammar rwth.depcol with org.eclipse.xtext.xbase.Xbase
2
3 generate depModelDSL
4   "http://www.dsa.de/devsupport/dependencymodeling/DepModelDSL"
5
6 DependencyModel:
7   featureBase = FeatureBase?
8   pluginBase = PluginBase?
9   elements += DependencyModelElement*
10 ;
11
12 FeatureBase:
13   'featurebase {'
14     (availableFeatures += FeatureName ';' )+
15   '}'
16 ;
17
18 FeatureName:
19   name = QualifiedName
20 ;
21
22 PluginBase:
23   'pluginbase {'

```

```

24         (availablePlugins += PluginName ';'')+
25     '}'
26 ;
27
28 PluginName:
29     name = QualifiedName
30 ;
31
32 QualifiedName:
33     Name+ ('.' Name) *
34 ;
35
36 Name:
37     (ID | '0..9')
38 ;
39
40 DependencyModelElement:
41     DependencySpecificationStructure |
42     DependencySpecificationElement
43 ;
44
45 DependencySpecificationStructure:
46     'declare ' FeatureGroup |
47     'declare ' PluginGroup
48 ;
49
50 FeatureGroup:
51     'featuregroup' name = QualifiedName ' {'
52     (features += [FeatureName | QualifiedName] ';') *
53     ('regexp' exp += NameExp ';') *
54     ('featuregroup' featureGroup +=
55     [FeatureGroup | QualifiedName] ';') *
56     '}'
57 ;
58
59 PluginGroup:
60     'pluginGroup' name = QualifiedName ' {'
61     (plugins += [PluginName | QualifiedName] ';') *
62     ('regexp' exp += NameExp ';') *
63     ('pluginGroup' pluginGroup +=
64     [PluginGroup | QualifiedName] ';') *
65     '}'
66 ;
67
68 NameExp:
69     (Name | '*' ) (Name | '*' | '.') *
70 ;
71
72 DependencySpecificationElement:

```

```

73 ConcreteDependencySpecificationElement ' {'
74     dependencySpecification += DependencySpecification+
75     '}'
76 ;
77
78 ConcreteDependencySpecificationElement:
79     FeatureDependencySpecification |
80     PluginDependencySpecification |
81     FeatureGroupDependencySpecification |
82     PluginGroupDependencySpecification
83 ;
84
85 FeatureDependencySpecification:
86     'feature' name = [FeatureName | QualifiedName]
87 ;
88
89 PluginDependencySpecification:
90     'plugin' name = [PluginName | QualifiedName]
91 ;
92
93 FeatureGroupDependencySpecification:
94     'featuregroup' name = [FeatureGroup | QualifiedName]
95 ;
96
97 PluginGroupDependencySpecification:
98     'pluggingroup' name = [PluginGroup | QualifiedName]
99 ;
100
101 DependencySpecification:
102     severity = DependencyViolationSeverity? type = DependencyType
103     dependency = ConcreteDependency ';'
104 ;
105
106 DependencyViolationSeverity:
107     '[' severity = ('critical' | 'error' | 'warning' ) ']'
108 ;
109
110 DependencyType:
111     AllowedDependency | ForbiddenDependency | TolerateDependency
112 ;
113
114 AllowedDependency:
115     typeStr = 'allow dependency to'
116 ;
117
118 ForbiddenDependency:
119     typeStr = 'forbid dependency to'
120 ;
121

```



```

122 TolerateDependency:
123     typeStr = 'tolerate dependency to'
124 ;
125
126 ConcreteDependency:
127     ConcretePluginDependency | ConcreteFeatureDependency
128 ;
129
130 ConcretePluginDependency:
131     DependencyToSinglePlugin |
132     DependencyToPluginPattern |
133     DependencytoPluginGroup
134 ;
135
136 DependencyToSinglePlugin:
137     'plugin' plugin = [PluginName | QualifiedName]
138 ;
139
140 DependencyToPluginPattern:
141     'pluginpattern' pluginPtn = RegExp
142 ;
143
144 DependencytoPluginGroup:
145     'pluggingroup' pluginGroup = [PluginGroup | QualifiedName]
146 ;
147
148 ConcreteFeatureDependency:
149     DependencyToFeature |
150     DependencyToFeaturePattern |
151     DependencytoFeatureGroup
152 ;
153
154 DependencyToFeature:
155     'feature' feature = [FeatureName | QualifiedName]
156 ;
157
158 DependencyToFeaturePattern:
159     'featurepattern' featurePtn = RegExp
160 ;
161
162 DependencytoFeatureGroup:
163     'featuregroup' featureGroup = [FeatureGroup | QualifiedName]
164 ;

```

Listing E.5: Grammatik der DSL DepCoL.

Anhang F

Kontextbedingungen

In diesem Abschnitt werden die Kontextbedingungen vorgestellt, die für die DSLs implementiert wurden, die in dieser Arbeit entwickelt wurden. Diese Kontextbedingungen sind syntaktisch prüfbare Regeln, die sich nicht alleine aus der Grammatik der Sprache ergeben und welche die gültigen Ausdrücke einer Sprache einschränken [Sch12].

In den nachfolgenden Beschreibungen ist jeder Kontextbedingung ein eindeutiger Identifizierer zugeordnet. In der Entwicklungsumgebung in der konkrete Modelle der zugehörigen Sprache modelliert werden, werden diese Identifizierer mit angegeben, falls ein Verstoß gegen eine Kontextbedingung identifiziert wird. Auf diese Weise haben die Modellierer die Möglichkeit gegebenenfalls eine ausführlichere Erläuterung zu der jeweiligen Kontextbedingung nachzulesen, insofern sie dies für notwendig erachten.

Für die DSL DUPS, die in Abschnitt 4.2.3 eingeführt wurde, sind keine Kontextbedingungen angegeben. Der Grund hierfür ist, dass in der erweiterten Matching-Komponente eine Validitätsprüfung der modellierten Delta-Benutzervogaben stattfindet. Über diese Validitätsprüfungen hinaus werden keine weiteren Bedingungen für DUPS Modelle geprüft.

F.1 Kontextbedingungen für IRS

Dieser Abschnitt stellt die Kontextbedingungen für die DSL IRS vor. Weitere Erläuterungen zu dieser Sprache können Abschnitt 5.4.1 entnommen werden.

N-01:	Der Name einer Auswirkungsregel muss eindeutig sein.
Schweregrad:	Fehler
Bemerkung:	Verschiedene Auswirkungsregeln müssen unterschiedliche Namen besitzen. Andernfalls kann nicht unmittelbar zwischen den Checklistenhinweisen der Auswirkungsregeln unterschieden werden, was zu Verwirrung bei den Entwicklern führen könnte, welche die Inhalte einer Checkliste abarbeiten.

N-02:	Der Name eines Gruppenkürzels muss eindeutig sein.
Schweregrad:	Fehler

Bemerkung:	In einer Auswirkungsregel können in dem Auswirkungsblock Gruppenkürzel und dazugehörige Beschreibungen angegeben werden. Innerhalb einer Auswirkungsregel darf ein Gruppenkürzel jedoch nur einmal deklariert werden. Andernfalls ist nicht ersichtlich, welcher Gruppe ein Checklistenhinweis, der auf das entsprechende Gruppenkürzel verweist, zugeordnet werden soll.

C-CH-01:	Die Beschreibung einer Auswirkung darf nicht leer sein.
Schweregrad:	Fehler
Bemerkung:	Die Beschreibung einer Auswirkung wird in einer Checkliste an die Entwickler gemeldet. Eine leere Beschreibung ist jedoch für Entwickler unbrauchbar, da dadurch nicht ausgedrückt wird, welche Entwicklungstätigkeit aufgrund der betroffenen Modelländerung durchgeführt werden muss.

C-CH-02:	Die Zeichenkette zu den Personen bzw. Rollen, für welche die Checklistenhinweise einer Auswirkungsregel relevant sind, darf nicht leer sein.
Schweregrad:	Warnung
Bemerkung:	Die Angabe zu den Personen bzw. Rollen, für die Checklistenhinweise einer Auswirkungsregel relevant sind, ist optional. Aus diesem Grund ist es unnötig eine leere Zeichenkette anzugeben. Dies verkompliziert eine Auswirkungsregel nur unnötig und resultiert daher in einer Warnung.

C-CH-03:	Die Beschreibung eines Gruppenkürzels darf nicht leer sein.
Schweregrad:	Fehler
Bemerkung:	Die Spezifikation einer Gruppe zu einem Checklistenhinweis ist optional. Sämtlichen Checklistenhinweisen, denen keine Gruppe zugeordnet ist, werden automatisch der Standardgruppe zugeordnet, zu der keine Beschreibung existiert. Die Zuordnung einer leeren Gruppenbeschreibung zu einem Checklistenhinweis ist dadurch unnötig und führt potentiell lediglich zu Verwirrung bei den Entwicklern.

F.2 Kontextbedingungen für LDL

Dieser Abschnitt stellt die Kontextbedingungen für die DSL LDL vor. Weitere Erläuterungen zu dieser Sprache können Abschnitt 7.5.1 entnommen werden.

CGL-01:	Innerhalb einer Schichtenspezifikation darf dieselbe VR nicht mehrfach ersetzt werden.

Schweregrad:	Fehler
Bemerkung:	Der Kompositionsschritt verarbeitet sämtliche Kompositionsoperationen des Schichtdefinitionsmodells. Im Rahmen dieser Komposition werden VRs konform zum Schichtdefinitionsmodell durch die entsprechenden GRVs ersetzt. Wird gemäß Schichtdefinitionsmodell eine VR durch mehrere GRVs ersetzt, kann nicht automatisch entschieden werden, durch welche GRV die VR in der Komposition ersetzt werden soll. Abschnitt 7.5.3 beschreibt weitere Validitätsprüfungen, für die zusätzlich ein konkretes Produktkonfigurationsmodell, welches in Abschnitt 7.5.2 eingeführt wird, benötigt wird.

F.3 Kontextbedingungen für PCL

Dieser Abschnitt stellt die Kontextbedingungen für die DSL PCL vor. Weitere Erläuterungen zu dieser Sprache können Abschnitt 7.5.2 entnommen werden.

N-01:	Der Name der Variante muss eindeutig sein.
Schweregrad:	Fehler
Bemerkung:	Wird der gleiche Name für verschiedene Generatorvarianten eingesetzt, kann nicht unmittelbar zwischen diesen Generatorvarianten unterschieden werden. Aus diesem Grund muss der Name der Variante eindeutig sein.

N-02:	Ein Produktkonfigurationsmodell darf nur auf existierende CGLs verweisen.
Schweregrad:	Fehler
Bemerkung:	Die Komposition einer Generatorvariante setzt voraus, dass die CGLs bekannt sind, die komponiert werden müssen. Verweist ein Produktkonfigurationsmodell auf einen nicht-existierenden CGL, kann die Komposition nicht ordnungsgemäß durchgeführt werden.

F.4 Kontextbedingungen für DepCoL

Dieser Abschnitt stellt die Kontextbedingungen für die DSL DepCoL vor. Weitere Erläuterungen zu dieser Sprache können Abschnitt 8.3 entnommen werden.

N-FPG-01:	Die Namen von Feature-Gruppen dürfen nicht mit Namen von Plugin-Gruppen übereinstimmen.
Schweregrad:	Fehler

Bemerkung:	Auch wenn syntaktisch zwischen Abhängigkeiten zu Feature- und Plugin-Gruppen unterschieden werden kann, sollten sämtliche modellierten Gruppen mit Hinblick auf eine bessere Lesbarkeit des Abhängigkeitsmodells unterschiedliche Namen besitzen.

N-FPG-02:	Die Namen von Feature- und Plugin-Gruppen dürfen nicht den Namen von existierenden Features und Plugins entsprechen.
Schweregrad:	Fehler
Bemerkung:	Die Verarbeitung eines Abhängigkeitsmodells ist problemlos möglich, selbst wenn eine Feature-Gruppe den Namen eines Features oder eine Plugin-Gruppe den Namen eines Plugins besitzt. Dessen ungeachtet beeinträchtigt dies unter Umständen die Verständlichkeit eines Abhängigkeitsmodells und ist daher verboten.

R-E-01:	Zu einem spezifischen Feature oder Plugin oder einer spezifischen Feature- oder Plugin-Gruppe dürfen nur einmal Abhängigkeitsrestriktionen in einem Abhängigkeitsmodell definiert werden. Es ist somit nicht zulässig mehrfach Abhängigkeitsrestriktionen zu demselben Feature, Plugin oder derselben Feature- oder Plugin-Gruppe zu definieren.
Schweregrad:	Fehler
Bemerkung:	Ein Abhängigkeitsmodell darf nicht mehrmals Definitionen von Abhängigkeitsrestriktionen für das selbe Element enthalten. Eine solche verteilte Definition von Abhängigkeitsrestriktionen würde lediglich die Verständlichkeit eines Abhängigkeitsmodells negativ beeinflussen.

R-E-02:	Werden Abhängigkeitsrestriktionen für ein Feature, Plugin, eine Feature- oder eine Plugin-Gruppe definiert, darf nur einmal eine Abhängigkeitsrestriktion bezüglich desselben Elements (Feature, Plugin, Feature-Gruppe oder Plugin-Gruppe) gemacht werden.
Schweregrad:	Warnung

Bemerkung:	Wird eine Abhängigkeitsrestriktion für ein Feature, Plugin, eine Feature- oder eine Plugin-Gruppe mehrfach angegeben, stellt das zwar prinzipiell kein Problem dar, diese redundante Definition ist aber überflüssig und kann daher entfernt werden, ohne die Bedeutung des Abhängigkeitsmodells zu verändern. Darüber hinaus ist es nicht sinnvoll, verschiedene Angaben bezüglich desselben Elements zu machen. Dieser Fall würde beispielsweise eintreten, wenn ausgedrückt wird, dass eine Abhängigkeit zu einem Plugin erlaubt ist und in demselben Element wird zusätzlich angegeben, dass die Abhängigkeit zu dem gleichen Plugin verboten ist. Aufgrund des Überschreibungsmechanismus könnten derartige Angaben zwar von der Prüfkomponente verarbeitet werden, allerdings stellen solche Angaben lediglich vermeidbare Widersprüche dar, die aufgelöst werden müssen, um Missdeutungen zu vermeiden.
R-E-03:	Ausschließlich zu Abhängigkeitsrestriktionen, die verbotene Abhängigkeiten kennzeichnen, darf ein Schweregrad angegeben werden.
Schweregrad:	Warnung
Bemerkung:	Der Schweregrad einer Abhängigkeitsrestriktion drückt aus, wie schwerwiegend ein Verstoß gegen die Abhängigkeitsrestriktion ist. Für erlaubte und tolerierte Abhängigkeiten ist die Definition eines solchen Schweregrads allerdings nicht sinnvoll. Aus diesem Grund sollte sichergestellt werden, dass ein Schweregrad nur für verbotene Abhängigkeitsrestriktionen angegeben wird.
G-FPB-01:	Jedes Abhängigkeitsmodell sollte über maximal eine Feature- und Plugin-Basis verfügen.
Schweregrad:	Warnung
Bemerkung:	Die Definitionen der Features und Plugins auf die innerhalb eines Abhängigkeitsmodells explizit verwiesen werden kann sollten in einer Feature- und Plugin-Basis vorgenommen werden, damit diese Informationen gebündelt an einer Stelle definiert sind. Andernfalls wären diese Informationen über ein Abhängigkeitsmodell verteilt, worunter die Verständlichkeit und Lesbarkeit des Abhängigkeitsmodell leiden könnte.
G-FPG-01:	Jede Feature- oder Plugin-Gruppe sollte benutzt werden.
Schweregrad:	Warnung

Bemerkung:	Wird eine Feature- oder Plugin-Gruppe deklariert, sollten für die Gruppe entweder Abhängigkeitsrestriktionen angegeben werden oder andere Abhängigkeitsrestriktionen sollten diese Gruppen referenzieren. Andernfalls handelt es sich um eine überflüssige Feature- oder Plugin-Gruppen.

G-FPG-02:	Jede Feature- oder Plugin-Gruppe darf nur einmal in einem Abhängigkeitsmodell deklariert werden.
Schweregrad:	Fehler
Bemerkung:	Eine Feature- oder Plugin-Gruppe darf nur einmal in einem Abhängigkeitsmodell deklariert werden, damit keine Missverständnisse bezüglich der in diesen Gruppen enthaltenen Elemente auftreten. Andernfalls ist fraglich, wie mit mehrfachen Deklarationen umgegangen werden soll. Würden alle Deklarationen vereint werden, wäre die Deklaration einer Feature- oder Plugin-Gruppe auf mehrere Stellen verteilt und dies würde die Lesbarkeit eines Abhängigkeitsmodells beeinträchtigen. Würde eine Deklaration eine vorherige Deklaration überschreiben, könnten Missverständnisse bezüglich der enthaltenen Elemente auftreten, so dass die Verständlichkeit des Abhängigkeitsmodells beeinträchtigt werden würde. Um diesen Problemen entgegenzuwirken ist die mehrfache Deklaration einer Feature- oder Plugin-Gruppe mit dem gleichen Namen unzulässig.

Anhang G

Fragebögen zu Fallstudien

G.1 Fragebögen bezüglich der Checklistenenerzeugung

```
1 > DQ1: Wie sehr vereinfachen die Checklisten die Ausführung
2 > der weiteren Entwicklungsaktivitäten?
3 Die Informationen, die in den Checklisten enthalten sind, sind
4 sehr hilfreich.
5
6 > DQ2: Wie bewerten Sie den Aufwand zur Abarbeitung der
7 > Checklisten?
8 Ich erwarte, dass die Checklisten die Entwicklungsaktivitäten
9 beschleunigen.
10
11 > DQ3: Welche Gründe sprechen für Sie für und gegen die
12 > Nutzung der Checklisten in Zukunft?
13 Ich würde sie einsetzen, da sie dabei helfen können die
14 Entwicklungsaufgaben schneller durchzuführen. Für mich
15 gibt es keinen Grund gegen den Einsatz der Checklisten.
```

Listing G.1: Antworten des ersten Entwicklers zum Fragebogen aus Abschnitt 5.7.

```
1 > DQ1: Wie sehr vereinfachen die Checklisten die Ausführung
2 > der weiteren Entwicklungsaktivitäten?
3 Die Wahrscheinlichkeit, dass eine erforderliche
4 Entwicklungstätigkeit vergessen wird, sinkt. Die Checklisten
5 animieren dazu auf strukturierte Weise zu arbeiten.
6
7 > DQ2: Wie bewerten Sie den Aufwand zur Abarbeitung der
8 > Checklisten?
9 Die Listen sollten sich relativ schnell abarbeiten lassen.
10 Den Detailgrad der Checklisten halte ich für richtig, da in
11 einigen Fällen zusätzliche Hinweise als Unterstützung
12 hilfreich sein können.
13
14 > DQ3: Welche Gründe sprechen für Sie für und gegen die
15 > Nutzung der Checklisten in Zukunft?
16 Ich würde die Checklisten immer verwenden. Auf diese Weise
```

17 kann man nicht vergessen bestimmte Entwicklungsaktivitäten
 18 auszuführen, unabhängig davon wie groß oder klein die
 19 Änderungen sind.

Listing G.2: Antworten des zweiten Entwicklers zum Fragebogen aus Abschnitt 5.7.

1 > DQ1: Wie sehr vereinfachen die Checklisten die Ausführung
 2 > der weiteren Entwicklungsaktivitäten?
 3 Die Checklisten könnten dabei helfen die Ausführung der
 4 Schritte zu vereinfachen. Allerdings müssten sie dazu
 5 gekürzt werden, da sie zu viele Details beinhalten.
 6
 7 > DQ2: Wie bewerten Sie den Aufwand zur Abarbeitung der
 8 > Checklisten?
 9 Die Checklisten sind größer als erforderlich. Ihre Größe
 10 wirkte zu Beginn auf mich abschreckend.
 11
 12 > DQ3: Welche Gründe sprechen für Sie für und gegen die
 13 > Nutzung der Checklisten in Zukunft?
 14 Ich würde die Checklisten gerne nutzen, wenn sie kompakter
 15 wären. In der aktuellen Form sind sie mir allerdings zu
 16 umfangreich.

Listing G.3: Antworten des dritten Entwicklers zum Fragebogen aus Abschnitt 5.7.

1 > DQ1: Wie sehr vereinfachen die Checklisten die Ausführung
 2 > der weiteren Entwicklungsaktivitäten?
 3 Die Checklisten können uns bei der Weiterentwicklung des
 4 Systems helfen. Vor allem zum Hinweis auf die
 5 XML-Migrationen halte ich sie für sinnvoll.
 6
 7 > DQ2: Wie bewerten Sie den Aufwand zur Abarbeitung der
 8 > Checklisten?
 9 Der Aufwand ist in Ordnung, wenn man den Nutzen der
 10 Checklisten beachtet.
 11
 12 > DQ3: Welche Gründe sprechen für Sie für und gegen die
 13 > Nutzung der Checklisten in Zukunft?
 14 Ich würde die Checklisten einsetzen, da sie uns dabei
 15 unterstützen die erforderlichen Schritte durchzuführen.
 16
 17 > Offenes Feedback:
 18 Das aktuelle Format der Checklisten ist verbesserungsfähig. Das
 19 Abarbeiten der Hinweise in einer Textdatei ist etwas umständlich.

Listing G.4: Antworten des vierten Entwicklers zum Fragebogen aus Abschnitt 5.7.

1 > DQ1: Wie sehr vereinfachen die Checklisten die Ausführung
 2 > der weiteren Entwicklungsaktivitäten?

```
3 Die Checklisten erleichtern die Arbeit mit dem System. Sie
4 zeigen welche Modelländerungen die Änderungen verursachen und
5 das ist in manchen Fällen wirklich hilfreich.
6
7 > DQ2: Wie bewerten Sie den Aufwand zur Abarbeitung der
8 > Checklisten?
9 Der Aufwand zum Abarbeiten der Hinweise ist akzeptabel. In
10 einigen Fällen enthalten die Checklisten aber recht
11 ausführliche Beschreibungen. Die könnte man teils weglassen,
12 da ich diese zusätzlichen Beschreibungen nicht benötige.
13
14 > DQ3: Welche Gründe sprechen für Sie für und gegen die
15 > Nutzung der Checklisten in Zukunft?
16 Für den Einsatz spricht, dass sie die Arbeit mit dem System
17 erleichtern. Jede Unterstützung zur Vereinfachung der
18 Arbeit ist hilfreich.
19
20 > Offenes Feedback:
21 Vielleicht könnte man die Checklisten in ein ALM-System
22 oder etwas ähnliches integrieren, so dass jeder Entwickler
23 Tickets für die Aufgaben erhält. Das wäre besser als die
24 aktuelle Textdatei.
```

Listing G.5: Antworten des fünften Entwicklers zum Fragebogen aus Abschnitt 5.7.

G.2 Fragebögen bezüglich der Architekturkonformität

```
1 > DQ1: Wie bewerten Sie die aktuelle Form der Benachrichtigung über
2 > Verstöße gegen Abhängigkeitsrestriktionen?
3
4 - Analog zu den Compilerfehlern aus Eclipse, also genau da, wo die
5 Meldungen hinkommen sollen!
6
7 - Die Markierung der betroffenen Zeile ist hilfreich, da man dann
8 direkt sieht welche Abhängigkeit die Probleme verursacht.
```

Listing G.6: Antworten zu der ersten Frage der Fallstudie aus Abschnitt 8.9.3.

```
1 > Wie hilfreich finden Sie die unmittelbare Prüfung einer
2 > Manifest-Datei nach dem Speichern einer Manifest-Datei und
3 > die unmittelbare Rückmeldung zu Verstößen?
4
5 - Besonders hilfreich wäre es, wenn man schon vor dem Speichern
6 des Manifests erkennen könnte, dass die aktuelle Änderung zu
7 einem Verstoß führen wird.
8
9 - Es ist gut, dass man direkt auf Verstöße aufmerksam gemacht
```

```
10 wird. So gibt es nicht irgendwann ein böses Erwachen, wenn man
11 schon Arbeit und Zeit in die Entwicklung gesteckt hat.
12
13 - Für den Entwickler darf es keinen Zusatzaufwand bedeuten, das
14 muss alles im Hintergrund laufen. Das ist gut so.
```

Listing G.7: Antworten zu der zweiten Frage der Fallstudie aus Abschnitt 8.9.3.

```
1 > DQ3: Wie bewerten Sie die aktuelle Form der Benachrichtigung über
2 > Verstöße gegen Abhängigkeitsrestriktionen? (die anderen Antworten
3 > sind nicht dargestellt, da diese nur ausdrücken, dass kein
4 > Verstoß begangen wurde)
5
6 - Ich habe einmal einen Verstoß eingeführt. Dieser wurde aber kurz
7 darauf wieder entfernt.
```

Listing G.8: Antworten zu der dritten Frage der Fallstudie aus Abschnitt 8.9.3.

```
1 > Offenes Feedback
2
3 - Es ist eine sinnvolle Erweiterung.
4
5 - Die Eclipse startet etwas langsamer, aber das fällt kaum auf.
6
7 - Manche Fehlermeldungen sollten vielleicht ein bisschen anders
8 formuliert werden. Ansonsten sieht alles sehr gut aus. Die
9 Ergebnisse sollten so gut wie möglich in die Standard
10 Entwicklungsumgebung integriert werden. Das ist der Fall.
11
12 - Nachdem die Bereinigung der Plugin-Struktur beendet ist, sollten
13 die erlaubten Abhängigkeiten weiter verfeinert werden ansonsten wird
14 die Plugin-Struktur schnell wieder unaufgeräumt.
```

Listing G.9: Offenes Feedback zur Fallstudie aus Abschnitt 8.9.3.

Anhang H

Glossar

Überschreiben von Abhängigkeitsrestriktionen Seien $b, c \in C$ mit $b = (s, t, k, i)$ und $c = (l, m, k', j)$ zwei Abhängigkeitsrestriktionen des Abhängigkeitsmodells $d \in DM$. Unter der Voraussetzung, dass b im Abhängigkeitsmodell vor c definiert ist, also $i < j$, überschreibt c die Abhängigkeitsrestriktion b genau dann, wenn gilt: $PP_d(b) \cap PP_d(c) \neq \emptyset$. Eine Abhängigkeitsrestriktion überschreibt eine andere Abhängigkeitsrestriktion also genau dann, wenn beide Restriktionen nicht disjunkte Mengen von Plugin-Paaren beschreiben.

Abhängigkeitsmodell Ein Abhängigkeitsmodell fasst sämtliche Abhängigkeitsrestriktionen zusammen, die ein Plugin-basiertes Softwaresystem betreffen. Darüber hinaus kann ein Abhängigkeitsmodell die Definition von Gruppen von Features und Plugins umfassen.

Abhängigkeitsrestriktion Eine Abhängigkeitsrestriktion drückt aus, dass eine Abhängigkeit zwischen spezifischen Plugins, Features, Gruppen von Plugins oder Gruppen von Features verboten oder erlaubt ist oder lediglich toleriert wird, aber nicht gewünscht ist.

Abhängigkeit zwischen Plugins Eine Abhängigkeit zwischen zwei Plugins besteht genau dann, wenn das eine Plugin als erforderliches Plugin in der Manifest-Datei des anderen Plugins aufgeführt ist. Die Folge davon ist, dass das Plugin auf den Code des anderen Plugins zugreifen und diesen verwenden kann.

Architektur-Erosion Architektur-Erosion bezeichnet die Abweichung zwischen der Architektur eines Softwaresystems und dessen Implementierung.

Auswirkungsregel Eine Auswirkungsregel beschreibt die Auswirkungen von Modelländerungen auf ein Softwaresystem. Dazu definiert eine Auswirkungsregel Bedingungen und legt dar, welche Auswirkungen eine Modelländerung hat, wenn sie eine der Bedingungen erfüllt.

Checkliste Eine Checkliste beschreibt die Auswirkungen von Modelländerungen. Konkret kapselt eine Checkliste Hinweise zu den Entwicklungsaktivitäten, die in einem Softwaresystem aufgrund der Modelländerungen durchgeführt werden müssen.

Datenmodell Ein Datenmodell definiert die Struktur der Daten, die durch ein Softwaresystem verarbeitet werden.

Delta-Benutzervorgabe Eine Delta-Benutzervorgabe steuert, auf welches Modellelement ein spezifisches Modellelement in der Matchingphase abgebildet wird oder legt fest, dass ein Modellelement auf kein anderes Modellelement abgebildet werden soll.

Differenzmodell Ein Differenzmodell kapselt sämtliche Modellunterschiede, die bei dem Vergleich zweier Modelle (bzw. Modellversionen) ermittelt wurden. Ein Differenzmodell wird aufbauend auf einem Matchmodell berechnet.

Feature (Kontext Feature-orientierte Programmierung) Ein Feature repräsentiert eine konfigurierbare Einheit eines Softwaresystems, die dazu beiträgt, dass eine Anforderung an das Softwaresystem umgesetzt wird [ALS06, ABKS13].

Feature-Gruppe Eine Feature-Gruppe umfasst beliebig viele Features. Sie ist, analog zu einer Plugin-Gruppe, nicht Bestandteil des Plugin-basierten Softwaresystems selbst, sondern ist eine logische, zur Entwicklung und damit innerhalb eines Abhängigkeitsmodells genutzte Struktur.

Feature (Kontext Plugin-basierte Softwaresysteme) Ein Feature besteht aus beliebig vielen Plugins und fasst diese zu einer Einheit zusammen.

Generatorregion Eine Generatorregion (engl. generator region, GR) ist eine dedizierte Region innerhalb eines Generatorartefakts, die eindeutig referenziert werden kann. Eine GR kann entweder einen Bereich innerhalb eines Generatorartefakts markieren, der Text enthält oder eine leere Region kennzeichnen, die keinen Text enthält.

Generatorvariante Eine Generatorvariante ist ein konkreter Generator, der Teile bestehender Generatoren wiederverwendet, erweitert und/oder anpasst.

Kompositionsoperation Eine Kompositionsoperation drückt aus, dass auf Basis zweier Artefakte A und B ein neues Artefakt C erzeugt werden soll, indem eine bestimmte Region innerhalb des Artefakts A durch Inhalt aus dem Artefakt B ersetzt oder Inhalt aus B zum Artefakt A hinzugefügt wird.

Komposition von Artefakten Komposition bezeichnet eine Operation, die zwei oder mehrere Artefakte zu einem neuen Artefakt zusammenfasst [Kra10].

Matchmodell Ein Matchmodell zeigt auf, welches Modellelement in einem Modell (bzw. in einer Modellversion) zu welchem Modellelement in einem anderen Modell (bzw. einer anderen Modellversion) korrespondiert. Überdies kann ein Matchmodell Modellelemente umfassen, die in der Matchingphase keinem anderen Modellelement zugeordnet werden konnten.

Metaassoziation Eine Metaassoziation verweist auf eine Assoziation einer Metaklasse, die keine Komposition darstellt, z.B. `eSuperTypes` der Metaklasse `EClass`.

Metaattribut Ein Metaattribut bezeichnet ein Attribut einer Metaklasse, z.B. `name` der Metaklasse `ENamedElement`.

Modell Evolution Modell Evolution bezeichnet einen kontinuierlichen Prozess im Lebenszyklus eines Modells, in dem ein Modell über eine Folge von Änderungsoperationen angepasst wird. Die durchzuführenden Änderungen sind primär in neuen oder sich ändernden Anforderungen oder in Änderungen der Ausführungsumgebung des modellierten Softwaresystems begründet..

Plugin Ein Plugin stellt die kleinste betrachtete Komponente eines Plugin-basierten Systems dar, die im Wesentlichen aus einer Menge von Klassen und benötigten Bibliotheken besteht.

Plugin-Gruppe Eine Plugin-Gruppe umfasst beliebig viele Plugins. Sie ist nicht Bestandteil des Plugin-basierten Softwaresystems selbst, sondern ist eine logische, zur Entwicklung und damit innerhalb eines Abhängigkeitsmodells genutzte Struktur.

Plugin-basiertes Softwaresystem Ein Plugin-basiertes Softwaresystem wird über eine Liste von Features und/oder eine Liste von Plugins festgelegt.

Primäre Klasse Eine primäre Klasse eines Klassendiagramms zeichnet sich dadurch aus, dass zu ihr eigenständige Daten vorliegen können. Sämtliche zu migrierenden Daten sind stets Daten, die zu primären Klassen erzeugt wurden.

Produktkonfigurationsmodell Über ein Produktkonfigurationsmodell wird eine konkrete Generatorvariante konfiguriert. Der wichtigste Bestandteil dieser Konfiguration ist die Festlegung der CGLs, die für die Generatorvariante ausgewählt werden. Aus dieser Selektion wird abgeleitet, aus welchen CGLs die Generatorvariante komponiert werden muss.

Schichtendefinitionsmodell Ein Schichtendefinitionsmodell kapselt für sämtliche CGLs, welche GRVs welche VRs ersetzen und welche GRs vor und hinter welche VRs eingefügt werden.

Sekundäre Klasse Eine sekundäre Klasse eines Klassendiagramms ist dadurch gekennzeichnet, dass zu ihr keine eigenständigen Daten vorliegen. Stattdessen sind die Daten zu sekundären Klassen stets in den Daten zu primären Klassen eingebettet.

Signatur einer Generatorregion Eine Signatur ist eine Zeichenkette, über die eine GR innerhalb eines Artefakts eindeutig identifiziert werden kann.

Software Evolution Software Evolution ist ein kontinuierlicher Prozess im Lebenszyklus eines Softwaresystems [Leh79], in dem ein Softwaresystem über eine Folge von Änderungsoperationen angepasst wird. Die durchzuführenden Änderungen sind primär in neuen oder sich ändernden Anforderungen oder in Änderungen der Ausführungsumgebung

eines Softwaresystems begründet [BR00]. Der Fokus der Änderungen liegt auf einer stetigen funktionalen Verbesserung des Softwaresystems, kann aber auch die Behebung von Fehlern betreffen [Art88].

Softwareproduktlinie Eine Softwareproduktlinie (engl. software product line, SPL) besteht aus einer Menge von Komponenten. Ein konkretes Produkt einer SPL wird durch die Auswahl einer Teilmenge dieser Komponenten erzeugt [RR15, GMR⁺16].

Variabilitätsregion Eine Variabilitätsregion (engl. variability region, VR) ist eine GR, für die über mindestens eine Kompositionsoption definiert ist, dass ihr Inhalt verändert werden soll.

Variante einer Generatorregion Eine Variante einer Generatorregion (engl. generator region variant, GRV) ist eine GR, die eine VR ersetzt.

Anhang I

Curriculum Vitae

Name	Müller
Vorname	Klaus
Geburtstag	25.06.1985
Geburtsort	Wittlich
Staatsangehörigkeit	deutsch

10/2011 - 07/2017	RWTH Aachen
	Promotion am Lehrstuhl für Software Engineering

10/2007 - 10/2010	Fachhochschule Trier
	Master of Science in Informatik

10/2004 - 08/2007	Fachhochschule Trier
	Bachelor of Science in Informatik

08/2001 - 03/2004	Cusanus Gymnasium Wittlich
	Abitur

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: “Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.” Modeling will be used in development projects much more, if the benefits become evident early, e.g. with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR⁺06, GKR⁺08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR⁺09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG⁺14] we discuss how to improve reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR⁺06, GKR⁺08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP⁺98] and describe UML semantics using the “System Model” [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH⁺98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR⁺06, KRV10, Kra10, GKR⁺08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR⁺07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK⁺11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK⁺07], guidelines to define DSLs [KKP⁺09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF⁺15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK⁺15a, HHK⁺13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR⁺11] using deltas [HRRS11, ?] and evolution on deltas [HRRS12]. [GHK⁺07] and [GHK⁺08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR⁺09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR⁺07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design

guidelines incorporates reuse through this form of composition [KKP⁺09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF⁺15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the “globalized” use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory in [RKB95, BHP⁺98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH⁺97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH⁺98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe

variability in a top down fashion, e.g., in the automotive domain [GHK⁺08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR⁺11, HRR⁺11] and to Delta-Simulink [HKM⁺13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK⁺13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK⁺11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP⁺11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR⁺13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK⁺07, GHK⁺08]. [HKM⁺13] describes a tool for delta modeling for Simulink [HKM⁺13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW⁺15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO₂ emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP⁺11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK⁺14, HHK⁺15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSElab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

- [BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.
- [BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.
- [BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.
- [BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.
- [BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.
- [BGH⁺97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.
- [BGH⁺98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.
- [BHP⁺98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.
- [BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.
- [BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.
- [CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.
- [CCF⁺15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

- [CEG⁺14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.
- [CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.
- [CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.
- [EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluwer Academic Publisher, 1999.
- [FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.
- [FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.
- [FLP⁺11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.
- [FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.
- [GHK⁺07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.
- [GHK⁺08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.
- [GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.
- [GKR⁺06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKR⁺07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

- [GKR⁺08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.
- [GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.
- [GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS’15)*, pages 34–43. ACM/IEEE, 2015.
- [GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.
- [GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.
- [GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ’12)*, 2012.
- [GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS’10)*, LNCS 6394, pages 331–345. Springer, 2010.
- [HHK⁺13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC’13)*, pages 22–31. ACM, 2013.
- [HHK⁺14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud’14)*. IEEE, 2014.
- [HHK⁺15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.
- [HHK⁺15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.
- [HKM⁺13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS’13)*, pages 11–18. ACM, 2013.
- [HKR⁺07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA’07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

- [HKR⁺09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineering in Research and Practice (SERP'09)*, pages 172–176, July 2009.
- [HKR⁺11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.
- [HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.
- [HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.
- [HRR⁺11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.
- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 1 – 10. fortiss GmbH, 2011.
- [HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.
- [HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.
- [HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.
- [KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.
- [KKP⁺09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.
- [KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

- [KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.
- [KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.
- [KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapman & 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'07)*, LNCS 4735, pages 286–300. Springer, 2007.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.
- [LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.
- [MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.
- [MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.
- [MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.
- [MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.
- [MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.
- [MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.
- [MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.
- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.
- [PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.
- [PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.
- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.
- [PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.
- [PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.
- [Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

- [RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.
- [RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.
- [RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.
- [RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.
- [RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.
- [RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.
- [RSW⁺15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.
- [Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations*, Seattle, pages 697–701. Idea Group Publishing, London, 2002.
- [Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.
- [Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.
- [Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.
- [Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.
- [Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

- [SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.
- [THR⁺13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.
- [Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.
- [Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.
- [ZPK⁺11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.