Modelle und Werkzeuge für modellgetriebene Softwareproduktlinien am Beispiel von Softwarekonfigurations-verwaltungssystemen

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Thomas Buchmann

geboren in Weiden i.d. Opf.

1. Gutachter: Prof. Dr. Bernhard Westfechtel

2. Gutachter: Prof. Dr. Albert Zündorf

Tag der Einreichung: 18. Juni 2010

Tag des Kolloquiums: 17. September 2010

Danksagung

An dieser Stelle möchte ich allen, die mich während der Entstehung dieser Arbeit begleitet haben, meinen herzlichsten Dank aussprechen.

Allen voran natürlich Prof. Dr. Bernhard Westfechtel, der mir durch die Anstellung als wissenschaftlicher Mitarbeiter an seinem Lehrstuhl überhaupt erst ermöglicht hat, ein Promotionsvorhaben anzugehen. Bei der Wahl des Promotionsthemas lies mir mein Doktorvater zunächst die Freiheit, mich ausgiebig mit verwandten Themen zu beschäftigen. Herr Westfechtel hatte dabei immer ein offenes Ohr für Probleme und Fragen und nahm sich immer die Zeit, um über Hindernisse oder neue Aspekte zu diskutieren. Durch das harmonische Arbeitsklima, das am Lehrstuhl von Prof. Westfechtel herrscht, konnte ich mich voll und ganz auf meine Arbeit konzentrieren.
Bei Prof. Dr. Albert Zündorf möchte ich mich herzlich für die Übernahme des Zweitgutachtens und die konstruktiven Ratschläge bedanken.

Einen großen Teil zu dem fruchtbaren Arbeitsklima am Lehrstuhl von Prof. Dr. Bernhard Westfechtel trugen auch meine Kollegen während dieser Zeit bei. Allen voran Alexander Dotor, mit dem ich mir ein Büro teilen durfte, und der auch immer für kritische Diskussionen bereit stand. Desweiteren möchte ich auch meinem ehemaligen Kollegen Dr. Bernhard Daubner, sowie Sabrina Uhrig und Sabine Winetzhammer danken. Weiterhin möchte ich den Lehrstuhlmitarbeitern Monika Glaser und Bernd Schlesier danken, die durch ihre Tätigkeiten einen reibungslosen Ablauf am Lehrstuhl sicherstellen.

Desweiteren möchte ich meinen Kollegen an den Standorten Kassel, Paderborn und Darmstadt, die für die Entwicklung von Fujaba verantwortlich sind, für die wertvollen Tipps und Hinweise danken. Allen voran Leif Geiger (Uni Kassel / Yatta Solutions), Dietrich Travkin (Uni Paderborn), Felix Klar (Uni Darmstadt) und Dr. Christian Schneider (Uni Kassel / Yatta Solutions).

Ein besonderer Dank gebührt meinen Eltern, die mir das Studium und somit indirekt auch die Promotion ermöglicht haben. Meinem Vater möchte ich nochmals für die Übernahme der nicht beneidenswerten Aufgabe danken, diese Arbeit auf typographische Fehler durchzusehen.

Bei meiner Frau Aisha möchte ich mich ganz besonders für die Unterstützung während der letzten Jahre bedanken, und vor allem dafür, dass sie mich ermutigt hat, dieses Promotionsvorhaben auch wirklich anzugehen.

Zusammenfassung

Die Entwicklung von Softwareproduktlinien basiert auf der pro-aktiven Wiederverwendung von Softwareartefakten (Anforderungen, Komponenten, Code, Testfällen, etc.). In einer Softwareproduktlinie werden die invarianten Eigenschaften der Softwareprodukte durch eine gemeinsame Basis realisiert. Durch Integration von Variabilität in diese gemeinsame Plattform wird die effiziente Entwicklung von kunden- bzw. marktspezifischen Softwarelösungen ermöglicht. Der hohe Anteil der Wiederverwendung führt dabei im Vergleich zur Entwicklung von Einzel-Softwaresystemen zu einer Reduktion von Entwicklungszeit und Entwicklungskosten bei einer gleichzeitigen Steigerung der Softwarequalität.

Modellgetriebene Entwicklung hingegen beschreibt die Verwendung von formalen Modellen zur Beschreibung eines Softwaresystems anstelle der Erstellung von Programmcode. Geeignete Werkzeuge sind in der Lage, diese Modelle in Programmcode zu übersetzen. Diese formalen Modelle beschreiben das zu erstellende System auf einer höheren Abstraktionsebene als der Programmcode. Ebenso wird die Wiederverwendbarkeit für unterschiedliche Zielplattformen erhöht, indem spezielle Codegeneratoren verwendet werden.

Die Kombination von modellgetriebener Softwareentwicklung und Software Produktlinien verspricht Produktivitätssteigerungen aus mehreren Gründen: (1) durch die Entwicklung von wiederverwendbaren Komponenten in einer Produktlinie und (2) durch die Erstellung von Modellen anstatt von Programmcode, (3) durch eine erleichterte Erweiterung von bestehenden Systemen, (4) durch Werkzeuge zur Automation und (5) durch Wiederverwendbarkeit des mittels Modellen beschriebenen Wissens für alle Projektmitarbeiter.

Diese Arbeit untersucht die Konzepte, Modelle und Werkzeuge, die benötigt werden, um modellgetrieben Produktlinien zu entwickeln. Als durchgängiges anwendungsbezogenes Beispiel wird die modellgetriebene Entwicklung einer Produktlinie für Softwarekonfigurationsverwaltungssysteme betrachtet. Das Beispiel Softwarekonfigurationsverwaltungssysteme wurde als nicht-trivialer Anwendungsfall gewählt, da in dieser Domäne sehr viele Systeme existieren, die sich teilweise in ihrer Funktionalität sehr ähneln, dennoch aber jeweils von Grund auf neu entwickelt wurden. Im Verlauf der Ausarbeitung werden bestehende Prozesse zur Produktlinienentwicklung vorgestellt und eine Kombination mit dem modellgetriebenen Entwicklungsansatz diskutiert. Es wird ein innovativer Ansatz eines modellgetriebenen Entwicklungsprozesses für Produktlinien vorgestellt und es werden neu entwickelte Werkzeuge präsentiert und diskutiert, die diesen spezifischen

Prozess unterstützen. Für die Entwicklung einer durchgängigen Werkzeugunterstützung wurde versucht, auf bestehende Werkzeuge zurückzugreifen. Dennoch waren umfangreiche Neuentwicklungen nötig, um eine Werkzeugunterstützung des kompletten Prozesses zu gewährleisten. Die Neuentwicklungen betrafen einerseits die Kopplung von Modellen aus der modellgetriebenen Softwareentwicklung mit Modellen aus der Produktlinienentwicklung, andererseits aber auch eine weitreichendere Unterstüzung von modellgetriebener Entwicklung im Bereich der Architekturmodellierung.

Abstract

The idea behind software product lines is based on an active and planned reuse of software artifacts. These artifacts comprise requirements, components, code and even test cases. The realization of invariant features of single products within a software product line are realized by a common platform. Integrating variability into this platform allows for efficient development of customer specific software solutions. The high amount of reuse leads to a significant reduction of both development time and costs compared to traditional software development and simultaneosly increases the quality of the resulting product.

The term model-driven software development on the other hand describes the creation of software systems by specifying formal models instead of writing code. These models are created using special CASE tools which allow transforming these models into program code. By using formal models, the systems is described on a higher level of abstraction and reuse for different platforms can easily be achieved by using different code generators.

The combination of these different techniques promises an increase of productivity for several reasons: (1) by developing reuseable components within a product line, (2) by creating formal models instead of program code, (3) by an easy adoption and extension of existing systems, (4) by providing tools for automation and (5) by resuing domain knowledge captured in models.

This thesis investigates concepts, models and tools required for a model-driven development of software product lines. The model-driven development of a product line for software configuration management (SCM) systems is used as an application-oriented example. SCM systems were chosen as a non-trivial example, because this domain is characterized by a huge number of systems, each of which realize similar functionality but have all been developed from scratch.
In this thesis, existing processes for developing product lines are introduced and a combination of them with the model-driven approach is discussed. An innovative approach for the model-driven development of software product lines is presented and new tools are introduced, which have been created to support the specific requirements of this process. In order to provide a consistent tool chain, the focus was to reuse existing tools as far as possible and to extend and integrate them with own tools to provide tool support for the whole development process. The tools developed in the context of this work comprise the mapping of models for software product line engineering to models used for model-driven software development. A second tool was created to provide a broad support for model-driven development in terms of modeling in the large.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Bereits gegen Mitte der 60er Jahre des vorigen Jahrhunderts trat der Begriff *Softwarekrise* zum ersten Mal auf. Durch einen rasanten Leistungsanstieg der Hardware entstand der Effekt, dass die Kosten für die Software erstmalig die Kosten für die Hardware überstiegen, was auch heute immer noch gilt. Bereits kurz darauf kam es zu den ersten großen gescheiterten Softwareprojekten. In der Folge wurde der Begriff des *Software Engineering*, der die ingenieurmäßige Entwicklung von Software beschreibt, als Reaktion auf die Erkenntnis geprägt, dass die bisher verwendeten Techniken mit dem zunehmenden Umfang und der Komplexität der Softwaresysteme nicht mehr Schritt halten konnten. Edsger W. Dijkstra sagte beispielsweise zu diesem Thema in seiner Dankesrede für den Touring Award [Dij72]:

> „[Die Hauptursache für die Softwarekrise liegt darin begründet,] dass die Maschinen um einige Größenordnungen mächtiger geworden sind! Um es ziemlich einfach auszudrücken: Solange es keine Maschinen gab, stellte die Programmierung kein Problem dar; als wir ein paar schwache Computer hatten, wurde Programmierung zu einem geringen Problem, und nun da wir gigantische Computer haben, ist die Programmierung ein ebenso gigantisches Problem.“

Diese nun fast 40 Jahre alten Ausführungen besitzen auch heute noch Gültigkeit, da die Softwarekrise noch nicht als beendet betrachtet werden kann. Die Komplexität von Softwaresystemen nimmt stetig zu, und damit, trotz großer Fortschritte in der Modernisierung und Strukturierung des Software-Prozesses, auch die Zahl der gescheiterten Softwareprojekte.

Es existieren einige Lösungsansätze und Konzepte, um die zunehmende Komplexität zu mildern, etwa durch objektorientierte Programmierung, aspektorientierte Programmierung und verschiedene Entwicklungsprozesse. Desweiteren versucht man verstärkt auf bereits erprobte Komponenten und Softwarebibliotheken zurückzugreifen, um einerseits effektiver zu arbeiten, andererseits aber auch die Fehlerrate zu minimieren. Ebenso wird der Einsatz von Code Generatoren und modellgetriebener Softwareentwicklung vermehrt bevorzugt.

1.2 Lösungsansätze

1.2.1 Wiederverwendung

In den 1970er Jahren erkannte man, dass sich Prozeduren leichter und besser wiederverwenden lassen, wenn sie zu Modulen zusammengefasst sind. Eine Anpassung dieser Module kann aber nur durch eine Veränderung des Modulcodes selbst erfolgen, was die Wiederverwendung darauf einschränkt, das Modul in seinem Status-Quo zu übernehmen, oder es auf die speziellen Bedürfnisse anzupassen. David L. Parnas, ein Pionier der Softwaretechnik, entwickelte das Modulkonzept [Par72], das eine wesentliche Grundlage der objektorientierten Programmierung darstellt.

Durch Einführung der objektorientierten Softwareentwicklung in den 80er Jahren des vorigen Jahrhunderts wurde eine nachträgliche Veränderung von wiederverwendetem Code durch Vererbung erreicht, was eine signifikante Verbesserung für die Softwareentwicklung darstellt. Dennoch erfolgt der Grad der Wiederverwendung auf Basis relativ kleiner Codefragmente, die als Bausteine für ein neues System dienen. Daher wurde später die objektorientierte Softwareentwicklung zur komponentenbasierten Softwareentwicklung weiterentwickelt und zum anderen wurden objektorientierte Rahmenwerke eingesetzt, die abstrakte Anwendungen für eine bestimmte Domäne darstellen. Allerdings wurde die Komplexität der Wiederverwendung unterschätzt, da Wiederverwendung auf der richtigen Abstraktionsebene geplant sein muss, und daher mehr umfasst als nur den reinen Quelltext.

Der Softwareproduktlinienansatz geht noch weiter. Eine Softwareproduktlinie enthält mehrere Softwareprodukte einer Domäne, die auf einer gemeinsamen Produktlinienplattform aufbauen und idealerweise als Produktfamilie realisiert sind [CN01, PBL05]. Die einzelnen Bestandteile sind genau auf die Produktlinie zugeschnitten und auf hohe Wiederverwendbarkeit ausgelegt. Sämtliche Teile wie Softwarekomponenten, Dokumente und Methoden können innerhalb der Produktlinie mit geringem Aufwand miteinander kombiniert werden.

1.2.2 Modellgetriebene Softwareentwicklung

Das Ziel von modellgetriebener Softwareentwicklung ist es, eine deutliche Produktionssteigerung zu erreichen, indem durch domänenspezifische Abstraktionen und deren formaler Modellierung ein hohes Automationspotenzial für die Softwareerstellung erschaffen wird [SVEH07]. Bei der Entwicklung komplexer Softwaresysteme spielen Modelle eine essentielle Rolle. Ein *Modell* ist eine Repräsentation des zu entwickelnden Systems, die auf einer höheren Abstraktionsebene liegt als der Programmcode. Hinsichtlich der Verwendung von Modellen im Softwareentwicklungsprozess unterscheidet man zwischen *modellbasierter Entwicklung*, bei der Modelle zur abstrakten Beschreibung des Zielsystems eingesetzt werden, und dem engeren Begriff der *modellgetriebenen Entwicklung*, die zusätzlich die Erstellung ausführbarer Modelle verlangt.

Zunehmend komplexe Geschäftsanforderungen lassen sich nur mit verbesserten Methoden und Werkzeugen zur Softwareentwicklung bewältigen. Modellgetriebene Entwicklung soll dazu beitragen, die Produktivität der Softwareentwicklung zu steigern, indem konventionelles Programmieren durch die Erstellung ausführbarer Modelle auf einer hohen Abstraktionsebene ersetzt wird.

Außerdem wird eine Verbesserung der Qualität und Wartbarkeit der entwickelten Softwaresysteme erreicht. Die dafür notwendigen Modelle nehmen bei diesem Ansatz eine zentrale Position ein. Daher werden für eine erfolgreiche Umsetzung sowohl domänenspezifische Sprachen, mit Hilfe derer die Modelle formuliert werden können, als auch Code Generatoren oder Interpreter, die aus diesen Modellen Programmcode erzeugen können oder die Ausführung auf einer vorhandenen Plattform erlauben, benötigt. Im Rahmen dieser Arbeit wird die modellgetriebene Entwicklung mit Werkzeugen, die auf der Modellierungssprache UML basieren und aus den dort erstellten Modellen ausführbaren Programmcode erzeugen können, betrachtet.

1.3 Software Produktlinien

Organisierte Wiederverwendung und organisierte Variabilität auf Basis einer gemeinsamen Plattform stehen bei der Produktlinienentwicklung im Mittelpunkt. Hierbei besagt der Begriff *organisierte Wiederverwendung*, dass einzelne Softwareprodukte durch Wiederverwendung aus einer speziell für die Wiederverwendung entwickelten Plattform abgeleitet werden. Dieser Ansatz stellt einen entscheidenden Umbruch im Softwareentwicklungsprozess dar, da einzelne Softwareprodukte nicht mehr reaktiv nach Kundenbedarf entwickelt werden. Vielmehr liegt der Fokus auf einer pro-aktiven Erstellung einer gemeinsamen Plattform, die für eine große Anzahl jetziger und zukünftiger Produkte verwendet werden kann [BKPS04].

1.3.1 Grundlagen

Bei der Planung und Realisierung einer Produktlinie müssen zu Beginn die wesentlichen Eigenschaften der Produkte erfasst und deren Gemeinsamkeiten und Unterschiede ermittelt werden. Bei der eigentlichen Entwicklung der Softwareproduktlinie wird zwischen den beiden Ebenen der Domänenentwicklung und der Anwendungsentwicklung unterschieden (siehe Abbildung 1.1) [PBL05].

In der Disziplin *Domänenentwicklung* findet die Entwicklung der gemeinsamen und variablen Bestandteile (auch Artefakte genannt) statt, die die Plattform bilden. Der Begriff Variabilität beschreibt hierbei die möglichen Unterschiede der einzelnen Produkte innerhalb der Produktlinie. Sie ist Voraussetzung für die gezielte Konstruktion von Produktartefakten und deren Wiederverwendung in der Anwendungsentwicklung.

Anwendungsentwicklung hingegen beschreibt den Prozess der Entwicklung einzelner Produkte aus der Produktlinie. Dabei steht die Wiederverwendung von Artefakten aus

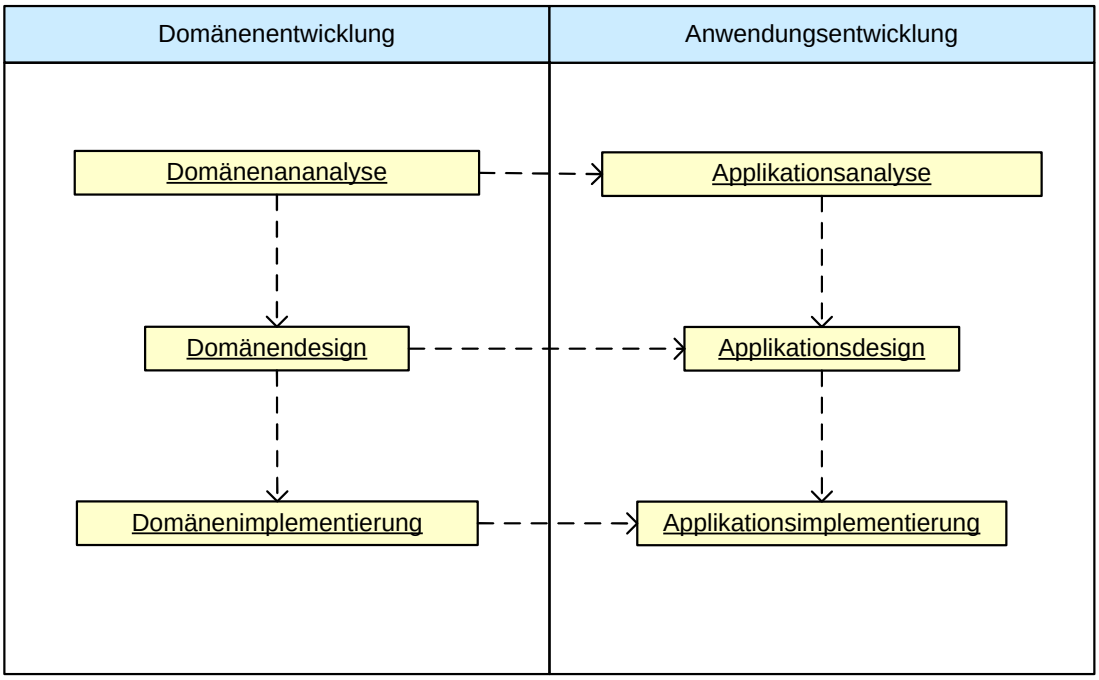

Abbildung 1.1: Referenzprozess der Softwareproduktlinienentwicklung nach [Lin02].

der gemeinsamen Plattform im Vordergrund. Diese Wiederverwendung erfolgt im Idealfall durch Konfigurierung, d.h. durch Zusammenfügen der Bestandteile mit möglichst geringen produktspezifischen Weiterentwicklungen.

Das SEI (Software Engineering Institute)[1] hat dabei drei Gruppen von Aktivitäten ermittelt, die nach Ansicht von Clements und Northrop [CN01] nötig sind, um Softwareproduktlinien erfolgreich zu entwickeln. Abbildung 1.2 zeigt diese Aktivitäten, die durch iterative Prozesse abgebildet werden:

Core Asset Development Während dieser Aktivität werden die Kernkomponenten der Softwareproduktlinie erstellt und erweitert. Hierfür sind eine Reihe von Eingaben, wie etwa die Anforderungen, Rahmenbedingungen und benötigte Qualitätsstandards erforderlich. Um in diesem Schritt bereits die Entwicklungskosten gering zu halten, wird dabei auf bereits bekannte Rahmenwerke und Entwurfsmuster zurückgegriffen. Auch werden bereits vorhandene Komponenten dahingehend untersucht, ob sie zu Produktlinienbestandteilen umgewandelt werden können und es wird eine Produktionsstrategie ausgesucht.

Product Development Hier werden aus den Kernkomponenten (Core Assets) die einzelnen Produkte erstellt. Die erforderlichen Eingaben für diesen Prozess erhält

[1] Das Software Engineering Institute ist ein Forschungs- und Entwicklungszentrum der Carnegie Mellon Universität. http://www.sei.cmu.edu/

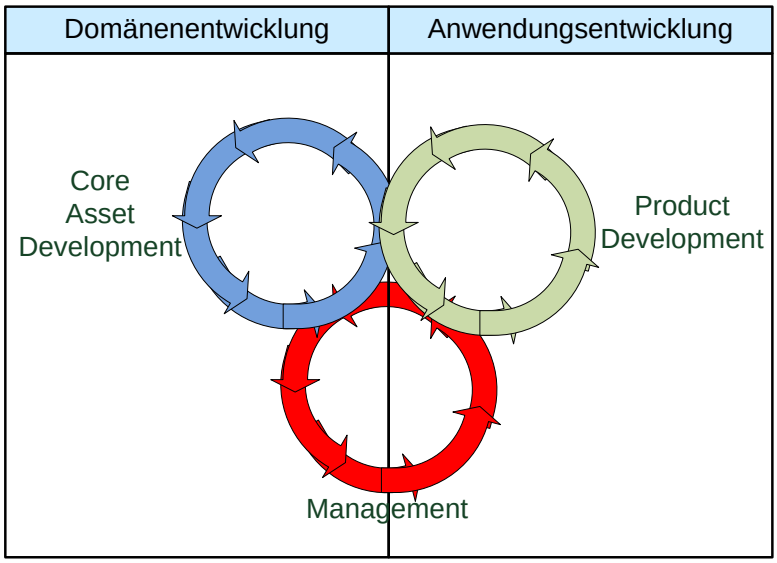

Abbildung 1.2: Die drei essenziellen Aktivitäten nach [CN01].

man aus dem Core Asset Development. Zu diesen kommen noch zusätzlich die produktspezifischen Anforderungen hinzu.

Management Das Management ist an allen Prozessen beteiligt. Hierbei wird zwischen technischem und organisatorischem Management unterschieden. Während das technische Management die Entwicklung der Kernkomponenten und Produkte überwacht, ist das organisatorische Management für den Vertrieb und die Kontaktpflege zu den Kunden verantwortlich.

Der Ansatz der Softwareproduktlinienentwicklung birgt etliche Vorteile gegenüber der herkömmlichen Softwareentwicklung:

Aufwandsreduzierung Durch die Wiederverwendung von bereits bestehenden Artefakten wird der Aufwand zur Entwicklung neuer Produkte drastisch reduziert.

Massenfertigung Durch die Verwendung von Variabilität, die die Modellierung von Individualisierung ermöglicht und durch die gemeinsamen Plattformen ist es möglich, Individualsoftware auf Basis einer Massenfertigung zu erstellen.

Steigerung der Qualität Durch die intensive Nutzung der bereits bestehenden Artefakte liegt deren Qualität über der von vergleichbaren Einzelkomponenten.

Die oben genannten Vorteile führen zu sinkenden Kosten bei der Softwareentwicklung und der Wartung, die Produktivität wird gesteigert während die Fehlerrate sinkt.

Allerdings erfordert die Arbeit mit Softwareproduktlinien auch ein gewisses Maß an Disziplin. Änderungen in den Anforderungen sollten keine unkoordinierten Änderungen einzelner Produkte bzw. deren Artefakte nach sich ziehen, um die Konsistenz der Plattform nicht zu gefährden. So muss für jede Änderung überprüft werden, ob sie in die gemeinsame Plattform übernommen wird, oder ob sie während der Anwendungsentwicklung nur für einzelne Produkte durchgeführt wird.

1.3.2 Variabilität

Das zentrale Konzept in allen Artefakten von Softwareproduktlinien stellt Variabilität dar. Die explizite Dokumentation der Variabilität in der Domänenentwicklung ermöglicht die gezielte Wiederverwendung von Artefakten in der Anwendungsentwicklung.

In der Literatur gibt es unterschiedliche Definitionen des Begriffs Variabilität. Bei Svahnberg und Bosch bedeutet Variabilität

> „... die Fähigkeit, ein System zu verändern oder anzupassen.“[SB00]

wobei der Fokus auf der Anpassung der Produkte an die Bedürfnisse der Kunden liegt. Weiss hingegen versteht unter Variabilität

> „... eine Annahme darüber, wie sich Mitglieder einer Softwarefamilie unterscheiden.“[WL99]

Nach der Definition von Weiss unterscheiden sich Produkte nur durch ihre variablen Teile voneinander, da alle gültigen Produkte der Softwareproduktlinie einen gemeinsamen unveränderlichen Kern besitzen. Hierbei ist zu beachten, dass David Weiss den Begriff „Softwarefamilie“ verwendet. Dieser Begriff ist in diesem Kontext gleichzusetzen mit dem Begriff Softwareproduktlinie.

Beiden Definitionen gemein ist das Konzept, spezifische an Kundenbedürfnisse angepasste Produkte abzuleiten. Diese spezifischen Produkte besitzen einen invarianten und einen vordefinierten variablen Teil (siehe [BKPS04], S. 13).

Bei der Planung von Softwareproduktlinien ist die rechtzeitige Erkennung und die Ermittlung des Umfangs von Variabilität entscheidend für den späteren Erfolg. Der Erfolg zeigt sich zum einen in einem hohen Grad an Wiederverwendbarkeit und zum anderen durch eine große Flexibilität der Produktlinie, die eine Vielzahl von Anforderungen abdeckt. Dies führt im Allgemeinen zu einem Konflikt, da die Unterstützung eines hohen Maßes an Flexibilität sehr kostenintensiv ist und andererseits aber alle Anforderungen erfüllt werden müssen.

In den letzten Jahren haben sich Featuremodelle und die zugehörigen Featurediagramme [BHST04] zur Modellierung der Variabilität in Softwareproduktlinien durchgesetzt

[KLD02]. Ein Featurediagramm ist dabei ein Baum, der aus einer Menge von Knoten und gerichteten Kanten besteht. Die einzelnen Knoten repräsentieren dabei einzelne Features (also Eigenschaften der Produktlinie) und die Kanten stellen die Beziehungen zwischen diesen Knoten dar. So können Knoten entweder alternativ, obligatorisch oder optional sein, sie können aber auch direkt oder indirekt untergeordnete Subfeatures darstellen.

1.4 Modellgetriebene Entwicklung

Seit nunmehr etwa 50 Jahren wird Software in industriellem Maße entwickelt. Dabei wurde im Laufe der Zeit der Abstraktionsgrad bei der Erstellung der Software stetig erhöht. Mussten zu Beginn Programme noch in der jeweiligen Maschinensprache des Zielsystems erstellt werden, folgte schließlich mit Assembler eine Sprache auf höherer Abstraktionsebene, die mittels eines „Assemblierers“ Maschinencode erzeugte. Assembler Quelltext ist für den Entwickler zwar einfacher zu lesen und zu verstehen als Maschinencode, der Abstraktionsgrad ist für die Entwicklung großer Systeme mit wiederkehrenden Problemen aber immer noch viel zu gering. Nach und nach wurden prozedurale Sprachen, wie etwa Fortran oder C eingeführt, die die Verwendung von Kontrollstrukturen zur Steuerung des Kontrollflusses ermöglichten. In den 70er Jahren des vorigen Jahrhunderts schließlich wurde erkannt, dass die Modularisierung [Par72] von Softwarekomponenten deren Wiederverwendungspotenzial erheblich steigert. Unabhängig davon wurde die objektorientierte Softwareentwicklung entwickelt, die durch eine Vielzahl von Programmiersprachen, wie C++ oder Java unterstützt wird.

1.4.1 Grundlagen

Da aber selbst bei der objektorientierten Softwareentwicklung noch Quelltext in einer Programmiersprache erstellt werden muss, erfordert dies meist einen ernormen Einarbeitungsaufwand, wenn Außenstehende bereits bestehende Programme warten oder erweitern müssen. Daher versucht der modellgetriebene Entwicklungsansatz eine erneute Steigerung des Abstraktionsniveaus durch die Beschreibung von Software durch formale Modelle anstatt von Programmcode. Diese Modelle können entweder mit einer textuellen oder einer graphischen Notation beschrieben sein, oder aber auch mit einer Kombination von beiden Techniken. Ein wesentlicher Vorteil des modellgetriebenen Ansatzes ist die Trennung des eigentlich zu lösenden Problems von den Details der plattformspezifischen Implementierung. Der Modellierer kann sich so ganz auf die Problemlösung konzentrieren und muss sich nicht um Implementierungsdetails kümmern. Diese Abstraktion hat außerdem den Vorteil, dass sie plattformunabhängig ist. Um aus den formalen Modellen letztlich ausführbare Programme zu erhalten sind entweder Generatoren zur Codeerzeugung oder Interpreter notwendig, die die Modelle ausführen.

Im Gegensatz zur *modellbasierten* Softwareentwicklung, bei der die Modelle in der Regel nur zum Zwecke der Dokumentation eingesetzt werden, sind die Modelle beim *modellgetriebenen* Entwicklungsansatz mit dem Quelltext gleichzusetzen, da sie die Grundlage für die Erzeugung von ausführbarem Quelltext darstellen. Alle Änderungen werden nur

auf dem Modell durchgeführt, eine manuelle Anpassung des generierten Quelltextes ist in der Regel nicht erforderlich. Auch bei der rein modellbasierten Entwicklung sind Modelle hilfreich, da viele Werkzeuge in der Lage sind, aus Klassendiagrammen zumindest die statische Struktur des Softwaresystems zu generieren. Lediglich die Methodenrümpfe müssen bei diesem Ansatz noch manuell implementiert werden. Die rein modellbasierte Nutzung hat aber den Nachteil, dass aufgrund der Dynamik des Enwicklungsprozesses insbesondere in den ersten Phasen des Lebenszyklus eines Softwaresystems häufig größere Änderungen erfolgen. Daher muss das Modell oft und sorgfältig angepasst werden, um Inkonsistenzen gegenüber der Implementierung zu vermeiden. Werden Modelle auf diese Art und Weise benutzt, so dienen sie nicht unmittelbar der Weiterentwicklung, da der Entwickler durch seine eigene Interpretation diese Modelle in Quelltext übersetzen muss [SVEH07].

1.4.2 Ansätze

Die Object Management Group (OMG) verfolgt mit der UML (Unified Modeling Language) das Ziel, eine einheitliche Modellierungssprache für den von der OMG propagierten *Model Driven Architecture*-Ansatz bereit zu stellen. Model Driven Architecture – kurz MDA – ist das Ergebnis eines Standardisierungsprozesses für einige Kernkonzepte der modellgetriebenen Softwareentwicklung mit dem Fokus auf Interoperabilität und Portabilität. So werden im MDA Ansatz plattformunabhängige (*platform independent model – PIM*) und plattformspezifische Modelle (*platform specific model – PSM*) unterschieden. Zur Spezifikation der plattformunabhängigen Modelle (PIMs) kommt dabei die UML zum Einsatz.

UML an sich besteht wiederum aus mehreren Teilen: (1) der sogenannten *Infrastructure* [OMG09a], die den Sprachkern für eine Metasprache definiert und so die Basis der Architektur darstellt, (2) der *Meta Object Facility (MOF)* [OMG06a], einer Metamodellierungssprache, die die in der Infrastruktur definierte abstrakte Syntax verwendet und erweitert, (3) der *Superstructure* [OMG09b], in der die unterschiedlichen von der UML angebotenen Diagrammarten definiert werden und die somit das Metamodell der UML Diagramme darstellt, (4) *XMI* (XML Metadata Interchange) [OMG07] als Austauschmechanismus von Modellen zwischen unterschiedlichen Werkzeugen und als Eingabe für Generatoren und Interpreter und (5) der *Object Constraint Language (OCL)* [OMG06b], einer formalen, textuellen Notation, die basierend auf mathematischen Konzepten eine Präzisierung von Modellen erlaubt (siehe [GPR06], S. 80f.).

Die UML ist eine standardisierte Modellierungssprache mit Möglichkeiten zur domänenspezifischen Anpassung durch die Verwendung von UML Profilen. Demgegenüber bieten die sogenannten *domänenspezifischen Sprachen (DSLs)* beliebige Freiheitsgrade bezüglich der Spezialisierung auf die Problemdomäne, allerdings bieten sie keine Standardisierung. Durch die Verwendung von Begriffen und Elementen aus der Anwendungsdomäne wird das Interessensgebiet stark eingeschränkt und somit auch die Komplexität der Sprache begrenzt. Bei der Darstellung der konkreten Syntax einer DSL werden in der Regel

Begriffe oder Symbole aus der Domäne verwendet, so dass im günstigsten Fall auch ein Domänenexperte, der selbst kein Softwareentwickler ist, ausführbare Modelle erstellen kann. Diese Modelle werden dann entweder durch Generatoren in Programmcode überführt, der anschließend übersetzt und ausgeliefert werden kann, oder aber es werden Interpreter bereitgestellt, die eine Laufzeitumgebung für die Modelle bereitstellen.

1.5 Softwarekonfigurationsverwaltung

1.5.1 Einführung

Als reales Anwendungsbeispiel wird in dieser Arbeit die Domäne Softwarekonfigurationsverwaltung betrachtet. Das MOD2-SCM Projekt [BD09c, BDW08b, BDW09, BDW10] zielt darauf ab, eine Produktlinie für Softwarekonfigurationsverwaltungssyteme modellgetrieben zu entwickeln. Die dafür notwendigen Komponenten sollen möglichst lose gekoppelt sein, um einen hohen Grad an Konfigurierbarkeit und Austauschbarkeit zu gewährleisten. Das im Rahmen dieser Arbeit verwendete Anwendungsbeispiel ist keineswegs konstruiert um die Berechtigung der Werkzeuge zu untermauern. Die Notwendigkeit zur Erstellung der Werkzeuge ist vielmehr das Resultat der Erfahrungen, die bei der Entwicklung der Produktlinie für Softwarekonfigurationsverwaltungssysteme gemacht wurden.

Unter Softwarekonfigurationsverwaltung (software configuration management - SCM) versteht man die Maßnahmen zur Kontrolle der Evolution von großen und komplexen Softwaresystemen. Es ist ein Arbeitsablauf im Softwareentwicklungsprozess [Kru03]. Gerade bei der Arbeit im Team ist es unerlässlich, einen Kontrollmechanismus für Änderungen von gemeinsam genutzten Daten zu besitzen, um typische Probleme wie Mehrfachwartung oder simultane Updates vermeiden zu können. In [IEE05] wird Konfigurationsverwaltung wie folgt definiert:

> „Konfigurationsverwaltung ist der Prozess, in dem Elemente des Systems *ermittelt* und definiert werden, Änderungen dieser Elemente während ihres gesamten Lebenszyklus *kontrolliert* werden, der *Zustand* von diesen Elementen und Änderungsanforderungen *aufgezeichnet* und gemeldet wird und die Korrektheit und Vollständigkeit der Elemente *verifiziert* wird.“

Die Kernfunktion eines jeden Softwarekonfigurationsverwaltungssystems ist *Versionskontrolle*. Viele der Versionskontrolle zu Grunde liegenden *Versionsmodelle* wurden in etlichen Forschungsprototypen, quelloffenen Produkten und kommerziellen Systemen implementiert [CW98, Dar91]. Ein Versionsmodell

> „... definiert die Elemente, die versioniert werden sollen, gemeinsame Eigenschaften aller Versionen eines Elements und die Deltas, d.h. die Unterschiede

> zwischen ihnen. Weiterhin bestimmt es die Art und Weise, wie Versionsmengen zusammengesetzt sind. Hierfür werden verschiedene Evolutionsdimensionen, wie etwa Revisionen und Varianten eingeführt, es wird festgelegt, ob eine Version durch den repräsentierten Zustand oder die Änderungen relativ zu einem festgelegten Anfangspunkt beschrieben wird, es wird eine geeignete Repräsentation für die Versionsmenge gewählt (z. B. ein Graph) und schließlich werden Operationen zur Wiederherstellung von alten und zur Erzeugung von neuen Versionen bereitgestellt.“ [CW98]

Im Laufe der Jahre wurden unzählige SCM Systeme entwickelt, die von kleinen Werkzeugen, wie RCS [Tic85] über mittelgroße Systeme wie CVS [Ves06] oder Subversion [CSFP04] bis hin zu großen, in der Industrie genutzten Systemen wie Adele [EC94] und ClearCase [Whi03] reichen.

Eine genauere Betrachtung dieser Systeme zeigt, dass wiederkehrende Konzepte, wie z.B. Revisionen, Varianten, zustands- und änderungsbasierte Versionierung immer wieder auftreten. Leider sind die Versionsmodelle für gewöhnlich implizit in den jeweiligen Systemen implementiert, was eine Wiederverwendung ausschließt. Daher wurden sämtliche Systeme mit teils erheblichen Aufwand von Grund auf erstellt. Alle in dieser Arbeit vorkommenden Beispiele sind aus dem MOD2-SCM Projekt entnommen. Dieses Projekt zielt darauf ab, eine modellgetriebene Produktlinie für Softwarekonfigurationsverwaltungssysteme zu erstellen [BDW08b, BD09c, BDW09], wobei der Schwerpunkt derzeit noch auf Versionskontrolle liegt.

In den folgenden Abschnitten wird ein grober Überblick über die am häufigsten verwendeten Konzepte in Versionskontrollsystemen [CW98] gegeben, die auch weitestgehend im MOD2-SCM Projekt implementiert wurden. Im folgenden bezeichnet der Ausdruck *versioniertes Objekt* ein beliebiges Objekt, das unter Versionskontrolle gestellt werden kann, so etwa Dateien oder Verzeichnisse oder auch Diagramme bzw. deren Bestandteile. Eine Version v repräsentiert einen Zustand eines sich entwickelnden Objekts o. Im Gegensatz dazu besitzt ein unversioniertes Objekt immer nur genau einen Zustand. Versionierung kann in beliebiger Granularität erfolgen, wie z.B. für ein komplettes Softwaresystem bis hin zu einzelnen Textzeilen. Wird ein Objekt unter Versionskontrolle gestellt, so erfordert es Mechanismen zur eindeutigen Zuordnung von Versionen, die zu diesem Objekt gehören. Im Falle von Softwareobjekten kann diese Zuordnung über einen eindeutigen Objektbezeichner erfolgen. Auch die einzelnen Versionen, die zu einem Objekt gehören, müssen durch einen eindeutigen Bezeichner unterscheidbar sein. Das Tupel $v = (ps, vs)$ beschreibt eine Version v durch einen Zustand des Produktraums (ps) und einen Punkt im Versionsraum (vs). Der Produktraum enthält alle Objekte, die unter Versionskontrolle gestellt werden und der Versionsraum legt deren Versionen in geeigneten Strukturen, etwa Graphen, ab.

1.5.2 Speicherung von Versionen

Viele Versionskontrollsysteme verwenden *Versionsgraphen* zur Darstellung des Versionsraumes. Ein solcher Graph besteht aus Knoten und Kanten, die Versionen und deren Beziehungen untereinander repräsentieren. Abbildung 1.3 zeigt eine Übersicht über die gängigsten Konzepte von Versionsgraphen.

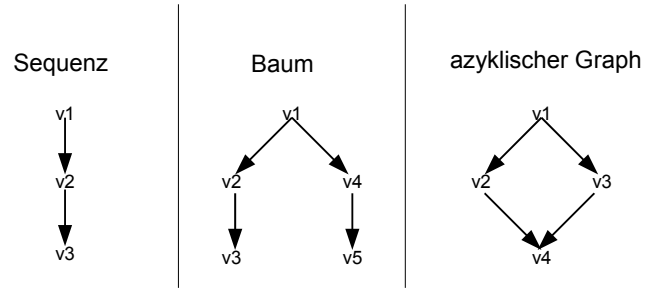

Abbildung 1.3: Versionsgraphen mit einer Ebene [CW98].

Sequenz

Im einfachsten Fall besteht ein Versionsgraph aus einer Menge von Knoten (die einzelnen Versionen, *Revisionen* genannt), die durch eine *Vorgänger-Nachfolger*-Beziehung miteinander verbunden sind. Ein solcher Graph dient zur Darstellung der Entwicklungsgeschichte eines Objekts, das unter Versionskontrolle steht. Die linke Seite von Abbildung 1.3 zeigt eine Sequenz, in der $v2$ der Nachfolger von $v1$ ist.

Baum

In einem Versionsbaum hingegen, können mehrere Nachfolger von Knoten erzeugt werden, die kein Blatt sind. Dies kann der Fall sein, wenn verschiedene Auslieferungsstände der Software gepflegt werden müssen. Im Beispiel in Abbildung 1.3 existieren zwei unterschiedliche Auslieferungsstände $v2$ und $v4$, die von der gleichen Basisversion $v1$ abgeleitet sind. Beide Versionen können nun unabhängig voneinander gepflegt werden (die Versionen $v3$ und $v5$).

Azyklischer Graph

Ein azyklischer Graph erlaubt mehrere Vorgänger für eine Version. Dies kann dazu benutzt werden, um Fehlerkorrekturen in einer älteren Version mit der Version, die aktuell entwickelt wird, zu verschmelzen.

Zweige und Verschmelzen

Ist der Graph hingegen nicht nur in einer Ebene (der Zeit) organisiert, sondern in zwei Ebenen, so besteht er aus Zweigen, von denen jeder Zweig eine Sequenz von Revisionen

enthält. In dieser Art der Versionsspeicherung können mehrere unterschiedliche *Varianten* zeitgleich existieren. Dazu werden mindestens zwei Arten von Beziehungen zwischen den Versionen benötigt: Nachfolgebeziehungen, die innerhalb eines Zweiges benutzt werden, und Verzweigungen. Wird weiterhin noch die Verschmelzung unterstützt, so ist es möglich, Änderungen auf einem Zweig in Versionen auf einem anderen Zweig zu propagieren. Dies führt wiederum zu einem gerichteten azyklischen Graphen. Die Zweige werden aber nicht vereint, sondern sie existieren weiterhin. Abbildung 1.4 zeigt einen Versionsgraphen mit vier Zweigen. Die Zweige $z2$ und $z3$ gehen aus der Version $v2$ des Zweigs $z1$ durch Abzweigung hervor. Von $v2$ des Zweigs $z2$ wird ein weiterer Zweig abgespalten, der dann nach einer Evolution wieder mit dem Zweig $z2$ verschmolzen wird $v3$ aus $z4$ mit $v4$ aus $z2$. Die Versionen $v2$ bzw. $v3$ aus dem Zweig $z3$ werden mit den Versionen $v4$ bzw. $v5$ des Ursprungszweiges $z1$ verschmolzen.

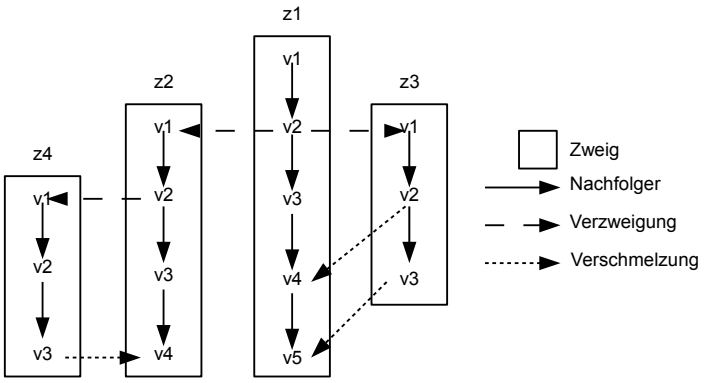

Abbildung 1.4: Versionsgraph mit Zweigen und Verschmelzungen [CW98].

1.5.3 Speicherung von Artefakten

Um ein versioniertes Objekt zu speichern, existieren ebenfalls mehrere unterschiedliche Ansätze [CW98]. Versionen unterscheiden sich durch spezielle Eigenschaften, die beispielsweise durch versionierte Attribute ausgedrückt werden können. Als *Delta* wird der Unterschied zwischen zwei Versionen bezeichnet. Prinzipiell kann man diese Unterschiede auf zwei Arten beschreiben: (1) Ein *symmetrisches Delta* zwischen zwei Versionen v_1 und v_2 besteht aus der Vereinigung der Mengen der spezifischen Eigenschaften beider Versionen, abzüglich der Schnittmenge: $\Delta(v_1, v_2) = (v_1 \cup v_2) \setminus (v_1 \cap v_2)$. (2) Ein *gerichtetes Delta*, auch *Änderung* genannt, wird durch eine Sequenz von elementaren Änderungsoperationen $op_1, \ldots, op_m$ beschrieben, die auf eine Version v_1 angewandt, zu einer Version v_2 führen.

Abbildung 1.5 zeigt auf der linken Seite einen möglichen Versionsgraphen und rechts davon die nachfolgend beschriebenen Möglichkeiten zur Speicherung der einzelnen Zustände des versionierten Objekts.

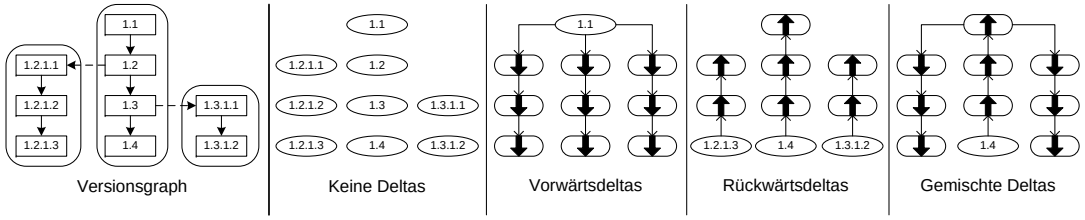

Abbildung 1.5: Logischer Aufbau des Versionsgraphen (links) und mögliche physikalische Darstellungen.

Ohne Deltas

Für jede Version v eines versionierten Objektes wird jeweils eine Kopie mit dem aktuellen Zustand abgespeichert. Sämtliche invarianten und variablen Bestandteile werden abgespeichert. Die einzelnen abgespeicherten Versionen sind voneinander unabhängig.

Vorwärtsdeltas

Für jede Version v eines versionierten Objektes, werden jeweils nur die Unterschiede zu einem festgelegten Vorgänger ausgehend vom Basisobjekt abgespeichert. Dabei wird, sobald das Objekt unter Versionskontrolle gestellt wird, die Basisversion $v_{1.1}$ abgespeichert. Alle weiteren Veränderungen an dem Objekt werden durch Änderungen zur jeweils letzten Version erfasst. Für die Rekonstruktion einer Version v_n werden ausgehend von der Basisversion v_1 alle Deltas der Versionen $v_2, \ldots, v_{n-1}$ der Reihe nach angewandt. Speziell für Objekte mit einer langen Historie ist dieses Vorgehen sehr ineffizient.

Rückwärtsdeltas

Im Gegensatz zu den im vorigen Abschnitt vorgestellten Vorwärtsdeltas wird bei den Rückwärtsdeltas die jeweils letzte Version als Basisversion v_n und die Vorgängerversionen als Deltas davon abgespeichert. Zur Rekonstruktion der Version v_i werden nun ausgehend von der letzten Version v_n alle Deltas der Versionen $v_{n-1}, \ldots, v_i$ angewandt.

Gemischte Deltas

Bei Systemen, die einen Versionsgraphen mit Zweigen und Verschmelzungen benutzen, wird oftmals die Form der gemischten Deltaspeicherung angewendet. Hierbei werden auf dem Hauptzweig Rückwärtsdeltas verwendet, auf allen Seitenzweigen hingegen Vorwärtsdeltas. Es gibt nach wie vor nur eine Basisversion, diese ist die letzte Version auf dem Hauptzweig. Um beispielsweise Version *1.2.1.3* aus dem Beispiel in Abbildung 1.5 zu erhalten, werden ausgehend von der letzten Version *1.4* bis zur Abzweigung bei Version *1.2* Rückwärtsdeltas angewendet, anschließend werden auf dem Zweig die Vorwärtsdeltas angewendet, bis die gewünschte Version erreicht ist.

1.5.4 Auswahl der Objekte

Die in Abschnitt 1.5.2 vorgestellten Konzepte zur Organisation des Versionsraums sind unabhängig vom verwendeten Produktraum. Im folgenden Abschnitt werden Möglichkeiten zur Kombination von Versionsraum und Produktraum vorgestellt. Bisher wurde nur die Versionierung einzelner Objekte betrachtet, nun aber wird ermittelt, welche Art von Objekten versioniert werden und wie die einzelnen Versionen von verschiedenen versionierten Objekten zusammenhängen.
Walter F. Tichy entwickelte das Modell der *AND/OR* Graphen, um das Zusammenspiel von Versionsraum und Produktraum mit Hilfe eines allgemeinen Modells zu beschreiben [Tic86]. Ein AND/OR Graph enthält zwei verschiedene Knotentypen, AND und OR Knoten. Von beiden Knotentypen gehen entsprechende AND und OR Kanten aus. Während ein unversionierter Produktgraph lediglich aus AND Knoten und Kanten besteht, wird die Versionierung durch die Einführung von OR Knoten modelliert.

Product first

Bei diesem Ansatz wird zuerst die gewünschte Struktur des Produkts ausgewählt. Danach erfolgt die Auswahl von Versionen der verschiedenen Komponenten. Mit diesem Ansatz kann die Struktur des Produktes nicht versioniert werden, d.h. sie ist für alle Konfigurationen gleich. Prominente Vertreter dieses Ansatzes sind beispielsweise RCS [Tic82, Tic85] und CVS [Ber90].

Version first

Der obige Ansatz wird hier invertiert. Zuerst wird die Version des Produktes selektiert, und dadurch werden eindeutig die Versionen der einzelnen Komponenten festgelegt. Mit Hilfe dieses Ansatzes können nun auch unterschiedliche Versionen von Produktkonfigurationen erfasst werden. Diese Art des Zusammenspiels zwischen Produktraum und Versionsraum ist etwa in Subversion realisiert [CSFP08].

Intertwined

Bei diesem Ansatz werden abwechselnd AND und OR Knoten ausgewählt. ClearCase [Leb95] verwendet dieses Modell, um beispielsweise Dateien und Verzeichnisse zu versionieren. Auch hier kann die Struktur des Produkts versioniert werden. Genau wie im *Version first* Ansatz werden hier nicht nur die Objekte versioniert, sondern auch die Beziehungen zwischen diesen Objekten. Der Unterschied zwischen beiden Ansätzen ist, dass hier auf jeder Ebene der Inhalt vorgegeben ist, die Version der Objekte allerdings frei gewählt werden kann. Während beim *Version first* Ansatz also die Wahl eines Verzeichnisses automatisch den Inhalt und dessen Version vorgibt, so kann beim *Intertwined* Ansatz hier die Version des Inhalts selbst bestimmt werden.

1.5.5 Art der Versionierung

Ein *versioniertes Objekt* besteht aus der Menge V aller Versionen dieses Objekts. Die Menge V kann prinzipiell auf zwei unterschiedliche Arten definiert werden.

Extensionale Versionierung

Werden alle Mengenelemente von V aufgezählt, so spricht man von *extensionaler Versionierung*:

$$V = \{v_1, ..., v_n\}.$$

Alle Versionen wurden im Laufe der Entwicklungsgeschichte eingecheckt und können auch wiederhergestellt werden. Die Identifikation einzelner Versionen erfolgt typischerweise über den eindeutigen Bezeichner.

Intensionale Versionierung

Im Gegensatz dazu wird Prädikatenlogik zur Aufzählung der Mengenelemente bei der *intensionalen Versionierung* verwendet:

$$V = \{v|c(v)\}.$$

Bei dieser Art der Versionierung können neue Versionen durch logische Ausdrücke erzeugt werden. Das Prädikat c definiert hierbei die Bedingungen, die für alle Elemente von V erfüllt sein müssen.

1.5.6 Erfassen von Änderungen

Zustandsbasiert

In Abschnitt 1.5 wurde eine Version als Zustand eines sich entwickelnden Objekts definiert. Dementsprechend werden Versionsmodelle, die Zustände von versionierten Objekten erfassen *zustandsbasiert* genannt. Die Beschreibung von Versionen geschieht hierbei mit Revisionen und Varianten (siehe Abschnitt 1.5.2).

Änderungsbasiert

Alternativ zu den Zuständen eines versionierten Objektes, können auch dessen Änderungen gespeichert werden. Eine bestimmte Version wird dann durch eine Menge von Änderungen beschrieben, die auf das Basisobjekt angewendet werden.

1.5.7 Synchronisation

Da in der Regel mehrere Benutzer an einem gemeinsamen Repository arbeiten, ist eine Synchronisation der Änderungen notwendig.

Pessimistische Synchronisation

Im Falle einer *pessimistischen Synchronisation* werden Sperren verwendet, um den Zugriff auf ein versioniertes Objekt für alle anderen Benutzer zu unterbinden. So ist sichergestellt, dass nur der Benutzer, der die Sperre angefordert hat, an dem Objekt arbeiten und anschließend seine Änderungen in das Repository übertragen kann.

Optimistische Synchronisation

Bei der *optimistischen Synchronisation* hingegen werden diese Sperren vermieden, was bedeutet, dass beliebig viele Benutzer gleichzeitig Änderungen an versionierten Objekten durchführen können. Werden diese Änderungen dann in das Repository übertragen, so muss eine Überprüfung auf etwaige Konflikte erfolgen und eine Auflösung dieser Konflikte erzwungen werden.

1.6 Beitrag der Arbeit

Diese Arbeit untersucht die Kombination der beiden in den Abschnitten 1.3 und 1.4 dargestellten Herangehensweisen moderner Softwareentwicklung. Wiederverwendung von bereits erprobten Softwarebestandteilen und modellgetriebene Softwareentwicklung sind, wie in Abschnitt 1.1 erläutert, Praktiken, die eine Erhöhung der Produktivität bei gleichzeitiger Minimierung der Fehler versprechen. Bislang werden diese Techniken weitgehend unabhängig voneinander eingesetzt. Diese Arbeit zeigt Modelle und Werkzeuge, mit deren Hilfe es möglich ist, beide Ansätze zu kombinieren, um somit die Vorteile beider Vorgehensweisen zu nutzen. Dabei werden soweit möglich bestehende und erprobte Werkzeuge eingesetzt und diese durch sinnvolle, neu entwickelte Hilfsmittel zu einer durchgängigen Werkzeugkette ergänzt. Als durchgängiges Anwendungsbeispiel wird in dieser Arbeit das MOD2-SCM Projekt verwendet, das die Erstellung einer Produktlinie für Softwarekonfigurationsverwaltungssysteme zum Ziel hat.

1.6.1 Modellierungsansatz

Der verwendete Modellierungsansatz zur modellgetriebenen Entwicklung von Softwareproduktlinien [BDW10] (siehe Abbildung 1.6) unterscheidet, wie in Softwareproduktlinien üblich, die Ebenen Domänenentwicklung und Anwendungsentwicklung [PBL05].

Domänenentwicklung In diesem Schritt wird zuerst die Domäne analysiert. Das Ergebnis der Analyse wird in einem *Featuremodell* [KCH+90] erfasst. Ein Featuremodell beschreibt variable und feste Bestandteile einer Produktlinie, alternative Ausprägungen der variablen Teile und deren Beziehung zueinander. Auf Basis des Featuremodells wird im nächsten Schritt ein konfigurierbares *Domänenmodell* entwickelt. Hierfür wird Fujaba [Zün02] benutzt. In diesem Domänenmodell werden die einzelnen Features aus dem Featuremodell realisiert. Das Domänenmodell umfasst hierbei sowohl ein Modell, das die Struktur beschreibt und durch Paket- und

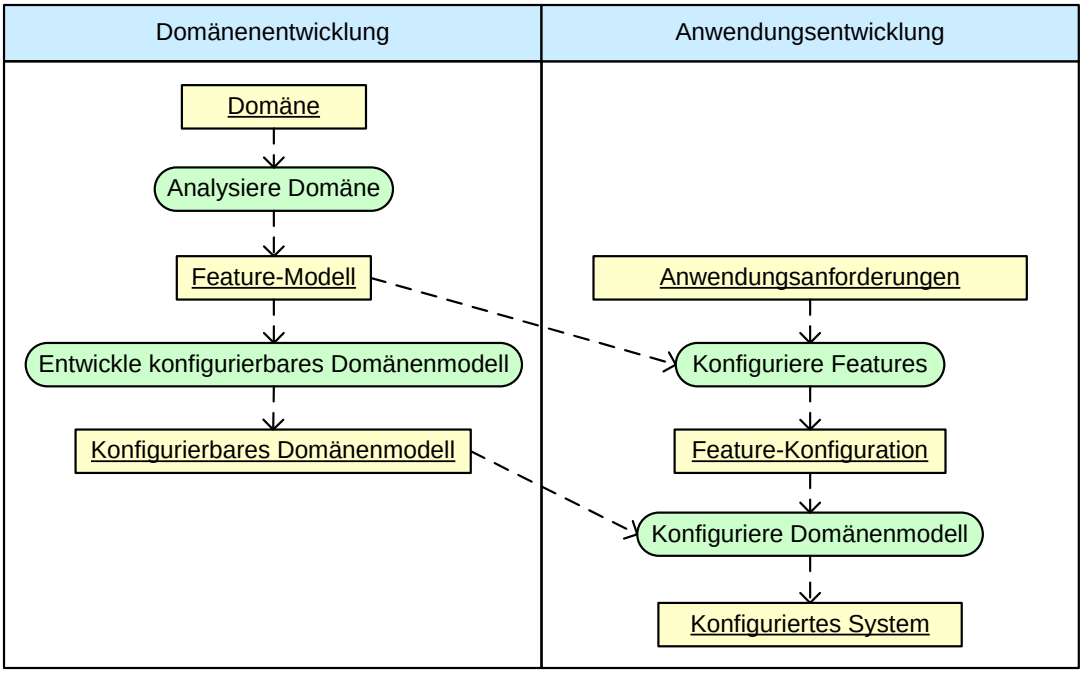

Abbildung 1.6: verwendeter Modellierungsansatz

Klassendiagramme ausgedrückt wird, als auch ein Verhaltensmodell, das durch Storydiagramme (siehe unten) beschrieben wird.

Anwendungsentwicklung Bei dem in dieser Arbeit vorgestellten Ansatz zur modellgetriebenen Entwicklung von Softwareproduktlinien wird die Anwendungsentwicklung zu einem reinen *Konfigurierungsprozess* reduziert. Zuerst wird das Featuremodell konfiguriert, indem die gewünschten Eigenschaften des zu erzeugenden Systems spezifiziert werden. Die daraus resultierende *Featurekonfiguration* wird anschließend verwendet, um das ausführbare Domänenmodell zu konfigurieren. Im letzten Schritt wird aus dem *konfigurierten Domänenmodell* schließlich ausführbarer Quelltext generiert. Lediglich der erste Schritt muss bei diesem Konfigurierungsprozess interaktiv durchgeführt werden, die weiteren Abläufe erfolgen vollständig automatisiert.

Abbildung 1.7 vermittelt einen Überblick über die zur Modellierung verwendeten Modelltypen und die zwischen ihnen bestehenden grobgranularen Beziehungen. Zur Definition der Eigenschaften der Softwareproduktlinie wird ein *globales Featuremodell* erstellt. Alle Teilmodelle des Domänenmodells nehmen auf dieses Modell Bezug. Zum *Modellieren im Großen* werden *Paketdiagramme* eingesetzt. Jedes Paket wird anschließend durch ein *Klassendiagramm* verfeinert. Das Verhaltensmodell schließlich wird durch *Storydiagramme* beschrieben [Zün02]. Jedes Storydiagramm realisiert das Verhalten einer Methode

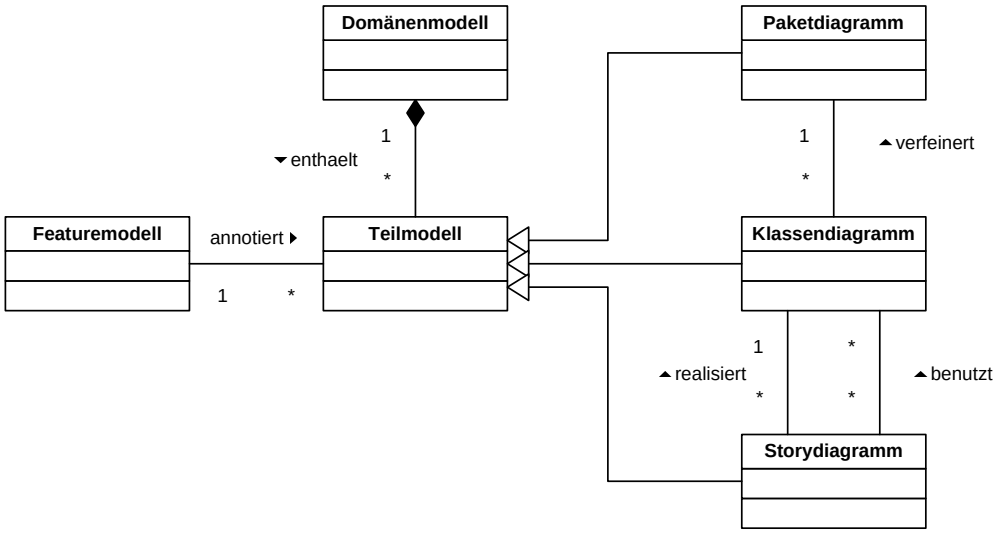

Abbildung 1.7: Modelle und deren Beziehungen

einer Klasse. Ein Storydiagramm ähnelt einem UML2-Interaktionsübersichtsdiagramm und besteht aus einem Aktivitätsdiagramm, dessen Knoten durch Storymuster (ähnlich zu Kommunikationsdiagrammen) verfeinert werden. Da Storydiagramme in ausführbaren Java-Code übersetzt werden, ist das konfigurierte Domänenmodell ausführbar.

Abbildung 1.8 zeigt beispielhaft einige Ausschnitte aus dem Modell der Produktlinie für Softwarekonfigurationsverwaltungssysteme, das im Rahmen von MOD2-SCM entstanden ist. Im Folgenden folgt eine knappe Beschreibung dieses Modells; für weitere Details sei auf [BDW09] bzw. die Doktorarbeit von Alexander Dotor [Dot10] verwiesen. Die linke Seite der Abbildung zeigt einen Ausschnitt aus einer Konfiguration eines Featuremodells, das die Eigenschaften der mit der Produktlinie erzeugbaren Versionskontrollsysteme beschreibt. Das der Konfiguration zugrunde liegende Featuremodell definiert z.B. Varianten für die Entwicklungsgeschichte (Feature **History**) und für die Abspeicherung (Feature **Storage**). Rechts oben ist ein Ausschnitt aus dem Paketdiagramm dargestellt, das die Modellstruktur im Großen definiert. Pakete, die ein spezifisches Feature des Featuremodells realisieren, werden mit Features annotiert (so wird beispielsweise dem Paket **Directed Deltas** das gleichnamige Feature zugeordnet). Rechts unten sind Ausschnitte aus einigen Klassendiagrammen dargestellt, die Pakete verfeinern, die zur Realisierung der Speicherung von Versionen dienen. Auch die Elemente der Klassendiagramme können mit Features annotiert sein (z.B. ist der Klasse **MixedDeltaStorage** das Feature **Mixed Deltas** zugeordnet). Aus Platzgründen zeigt die Abbildung kein Storydiagramm. Im bisherigen Modell sind Storydiagramme allenfalls insgesamt featurespezifisch. Von der Annotation feingranularer Elemente von Storydiagrammen wurde dagegen bisher kein Gebrauch gemacht, wenngleich das neu entwickelte Werkzeug auch dafür Unterstützung anbietet.

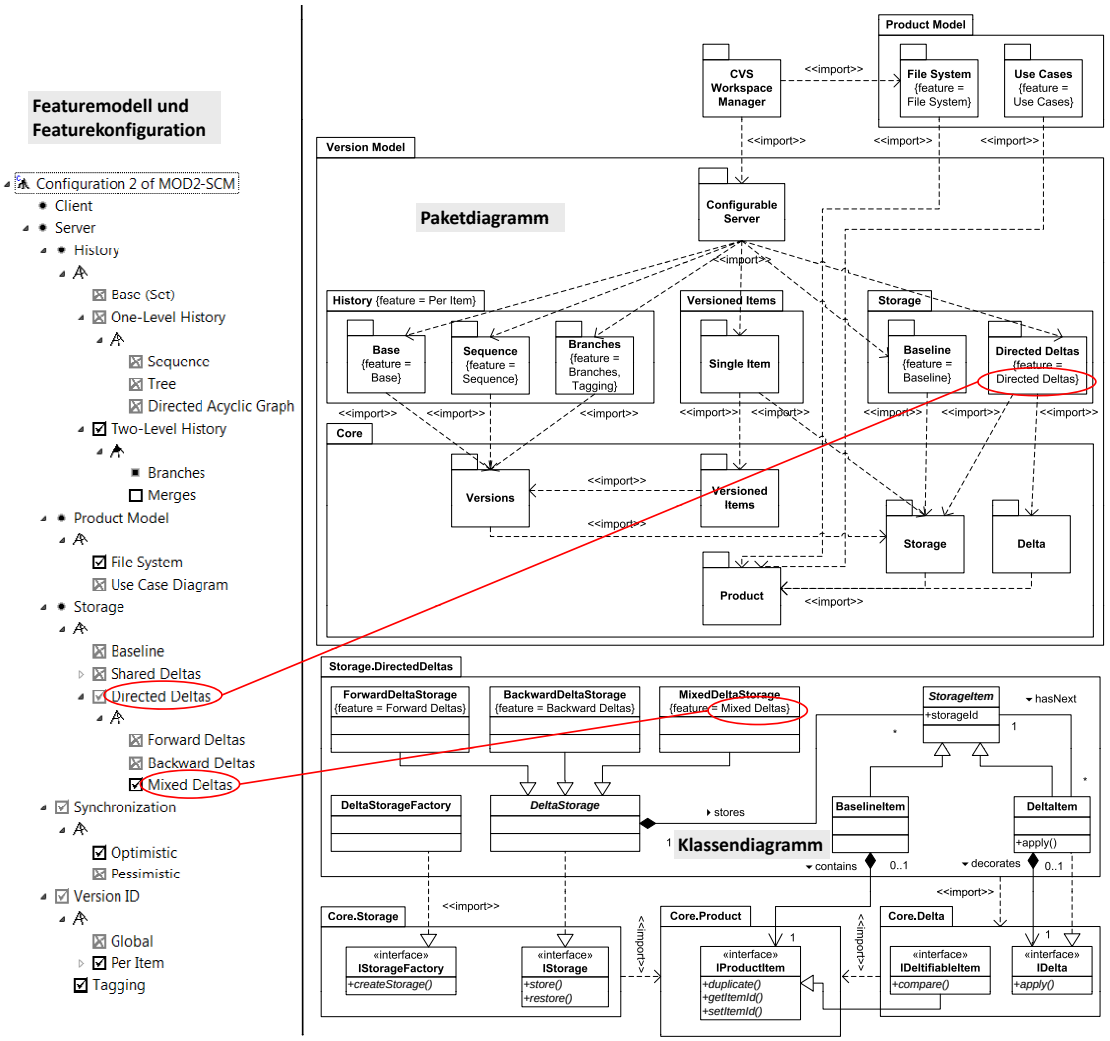

Abbildung 1.8: Modellausschnitte

1.6.2 Werkzeuge

Der Fokus dieser Arbeit liegt in der Wiederverwendung bereits bestehender Werkzeuge für die einzelnen Disziplinen im Entwicklungsprozess, der Identifikation von Lücken in der Werkzeugkette und das Schließen dieser durch die Entwicklung von eigenen Werkzeugen. Abbildung 1.9 zeigt einen Überblick über die wiederverwendeten bzw. neu entwickelten Werkzeuge und deren Interaktion.

Im Laufe des MOD2-SCM Projekts, das die Entwicklung einer Produktlinie für Softwarekonfigurationsverwaltungssysteme zum Ziel hat und in der Arbeit von Alexander Dotor [Dot10] beschrieben ist, zeigte sich jedoch, dass die zur Verfügung stehende Werkzeugunterstützung gravierende Lücken aufwies, die sich nur durch Eigenentwicklung schließen ließen. Insgesamt ist somit eine *Modellierungsumgebung* entstanden, die sich teilweise

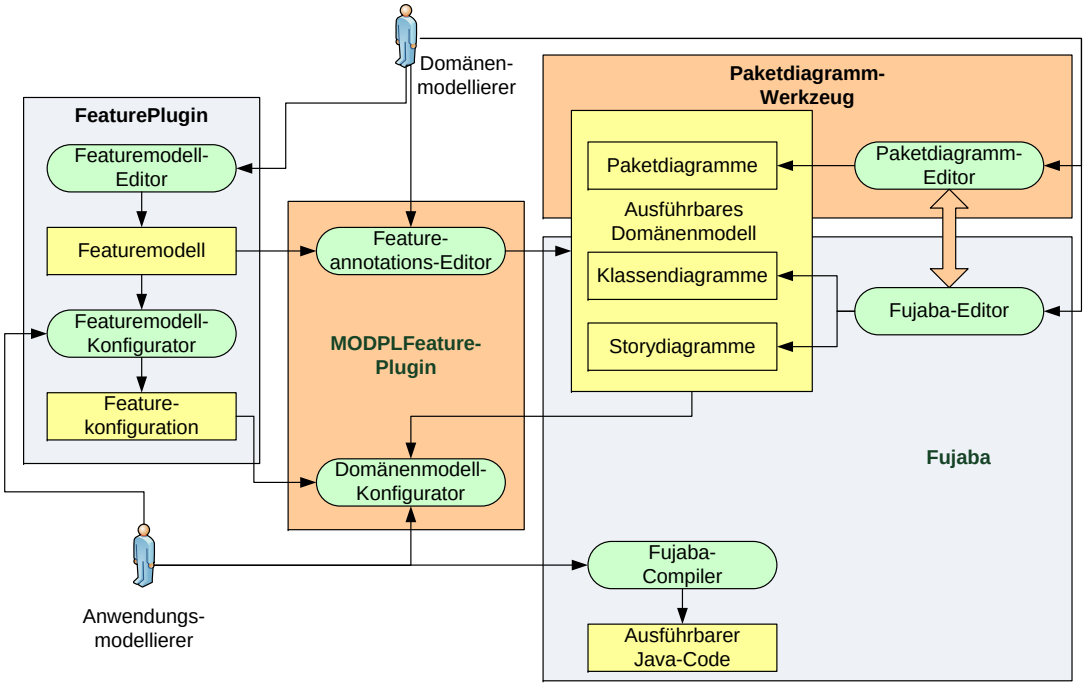

Abbildung 1.9: Überblick über die wiederverwendeten und neu entwickelten Werkzeuge und deren Interaktion.

aus bereits existierenden und teilweise aus selbstentwickelten Werkzeugen zusammensetzt (siehe Abbildung 1.9, in der die selbstentwickelten Werkzeuge orange hinterlegt sind).

Featuremodelle werden mit dem *FeaturePlugin* [AC04] erstellt und konfiguriert und das Domänenmodell wird mit *Fujaba* [Zün02] erstellt. Die im Rahmen dieser Arbeit entwickelten Werkzeuge dienen dazu, diese Werkzeuge zu integrieren — dies ist die Aufgabe des *MODPL Feature Plugin* — bzw. bei der Erstellung des Domänenmodells das Modellieren im Großen zu unterstützen — dies leistet das *Paketdiagramm-Werkzeug*. In den folgenden Unterabschnitten werden die Werkzeuge der Modellierungsumgebung kurz beschrieben. Dabei liegt der Schwerpunkt auf den selbstentwickelten Werkzeugen.

1.6.3 Featuremodellierung und -konfigurierung

Featuremodelle wurden ursprünglich in [KCH+90] eingeführt. Aus dem ursprünglichen FODA-Ansatz (**F**eature **o**rientierte **D**omänen**a**nalyse) haben sich mittlerweile eine Reihe von Varianten entwickelt, s. z.B. [BHST04]. Im Rahmen dieser Arbeit wird eine dieser Varianten benutzt, die im Werkzeug FeaturePlugin [AC04] implementiert ist.

Der linke Teil von Abbildung 1.8 ist ein Ausschnitt aus einem mit diesem Werkzeug erstellten Modell. Er zeigt eine *Featurekonfiguration*, die durch Auswahl von Features

aus dem Featuremodell für MOD2-SCM entsteht. Die Featurekonfiguration besteht aus den mit Häkchen gekennzeichneten Features und ihren übergeordneten Features im Featuremodell. Die durch Kreuze markierten Features sind Bestandteil des Featuremodells, wurden jedoch in der dargestellten Featurekonfiguration ausgeschlossen. Die Featurekonfiguration repräsentiert ein CVS-ähnliches System: Für die Versionshistorie wurde eine zweistufige Versionsgeschichte mit expliziten Zweigen ausgewählt. Als Produktmodell wird das Dateisystem festgelegt. Zur Speicherung werden gemischte Deltas verwendet (Rückwärtsdeltas auf dem Hauptzweig, Vorwärtsdeltas auf den Seitenzweigen). Die Synchronisation erfolgt optimistisch. Versionen werden lokal identifiziert, d.h. jedes versionierte Objekt vergibt seine Versionsbezeichner unabhängig von anderen versionierten Objekten. „Tagging“ bedeutet, dass Versionen durch symbolische Namen markiert werden können.

FeaturePlugin unterstützt die Erstellung von Featuremodellen und deren Konfigurierung, ohne dass jedoch Beziehungen zum Domänenmodell hergestellt werden. Diese Aufgabe übernimmt das neu entwickelte Werkzeug MODPLFeaturePlugin (Kapitel 6).

1.6.4 Fujaba

Im Zentrum der Werkzeugunterstützung steht *Fujaba* [Zün02], die mit Abstand größte Komponente der Modellierungsumgebung. Fujaba unterstützt die Erstellung von *Klassen- und Storydiagrammen*, aus denen der Fujaba-Compiler ausführbaren Code generiert. Dabei stellen die Storydiagramme [ZSW99] und die Erzeugung von ausführbarem Code einen beträchtlichen Mehrwert gegenüber anderen Modellierungswerkzeugen dar. Da die zugrunde liegende Technologie über einen langen Zeitraum gereift ist und echte modellgetriebene Entwicklung erlaubt, fiel die Wahl auf Fujaba.

Dennoch ist es mit Fujaba und FeaturePlugin alleine nicht möglich, Softwareproduktlinien rein modellgetrieben zu entwickeln. Außerdem waren damit die Herausforderungen des MOD2-SCM-Projekts nicht zu bewältigen. Zum einen wird das Modellieren im Großen durch Fujaba nicht unterstützt, zum anderen bietet Fujaba keine spezifische Unterstützung von Produktlinien. FeaturePlugin hingegen bietet nur die Möglichkeit zur Erstellung und Konfigurierung von Featuremodellen. Anbindungen an Werkzeuge zur modellgetriebenen Softwareentwicklung sind nicht vorgesehen.

1.6.5 Paketdiagramm-Werkzeug

Ein Featuremodell, wie es auf der linken Seite in Abbildung 1.8 dargestellt wurde, suggeriert, dass das zu entwickelnde System einfach ist. Dies ist aber keineswegs der Fall: Es ist nämlich insbesondere zu klären, wie das Featuremodell auf ein Domänenmodell so abgebildet werden kann, dass im Featuremodell als unabhängig deklarierte Features im Domänenmodell tatsächlich voneinander entkoppelt sind. Dies stellt große Anforderungen an die *Architekturmodellierung*.

Im MOD2-SCM-Projekt zeigte sich schon früh, dass das Domänenmodell mit den von Fujaba angebotenen Werkzeugen allein nicht strukturiert werden kann [BDW08a]. Bereits nach kurzer Zeit erreichte das Domänenmodell eine erhebliche Größe, so dass man sehr schnell den Überblick über die zahlreichen Klassen- und Storydiagramme verliert. Insbesondere beim Bau einer Produktlinie ist aber darauf zu achten, die Abhängigkeiten zwischen den Modellkomponenten präzise zu überwachen; andernfalls sind bei der Konfigurierung des Domänenmodells sehr leicht unbrauchbare Resultate die Folge. Daher wurde als zusätzliches Hilfsmittel ein *Pakdetdiagramm-Werkzeug* entwickelt, dessen Konzeption und technische Realisierung in Kapitel 5 beschrieben wird.

Den Kern des Paketdiagramm-Werkzeugs bildet ein *Paketdiagramm-Editor*, der die Erstellung von UML2-Paketdiagrammen unterstützt. Abbildung 1.10 zeigt einen Bildschirmabzug dieses Editors, der mit Hilfe von GMF [EF09] realisiert wurde. Während der Bildschirmabzug nur Pakete als Elemente des Diagramms zeigt, unterstützt der Editor auch das Einfügen von Klassen und Interfaces in ein Paket. Abgesehen von der hierarchischen Schachtelung, lassen sich mit dem Editor sowohl private als auch öffentliche (transitiv wirkende) Importe eintragen — und zwar sowohl Paket- als auch Elementimporte.

Der Paketdiagramm-Editor kann auch unabhängig von Fujaba genutzt werden. Die zweite, im Kontext des MOD2-SCM-Projekts wichtige Komponente des Paketdiagramm-Werkzeugs ist der *Integrator*, der in Abbildung 1.9 durch einen Doppelpfeil repräsentiert wird. Im Einzelnen bietet der Integrator folgende Funktionalitäten an:

Forward Engineering Aus einem Paketdiagramm wird das Gerüst eines Fujaba Modells erzeugt. Dieses Gerüst umfasst alle Pakete und die darin enthaltenen Elemente. Enthält ein Paket Klassen, so wird für dieses Paket automatisch ein Klassendiagramm mit dem Namen des Pakets angelegt. Man beachte, dass das Fujaba zugrunde liegende Metamodell die Zuordnung von Klassen und Interfaces zu Paketen zwar unterstützt, jedoch kein Fujaba-Werkzeug zur Erstellung und Visualisierung von hierarchischen Paketdiagrammen zur Verfügung steht. Die Verfeinerung dieser groben Architektur erfolgt dann ausschließlich in Fujaba mit Hilfe der Klassen- und Storydiagramme.

Reverse Engineering Ein Paketdiagramm wird aus einem zuvor entwickelten Fujaba Modell erstellt. Dazu wählt der Modellierer den gewünschten Importtyp aus (öffentliche oder private bzw. Element- oder Paketimporte). Das Paketdiagramm enthält anschließend alle im Fujaba Modell definierten Pakete und die gewünschten Importbeziehungen. Im Falle von Elementimporten enthält das Paketdiagramm zusätzlich noch die dafür benötigten Elemente, die in den jeweiligen umschließenden Paketen erzeugt werden.

Inkrementelles Roundtrip-Engineering Neben den batchartigen Prozessen des Forward und Reverse Engineering wird auch inkrementelles Roundtrip-Engineering unterstützt. Hierbei werden sowohl Änderungen am Paketdiagramm als auch Änderungen am Fujaba-Modell in die jeweilige Richtung propagiert.

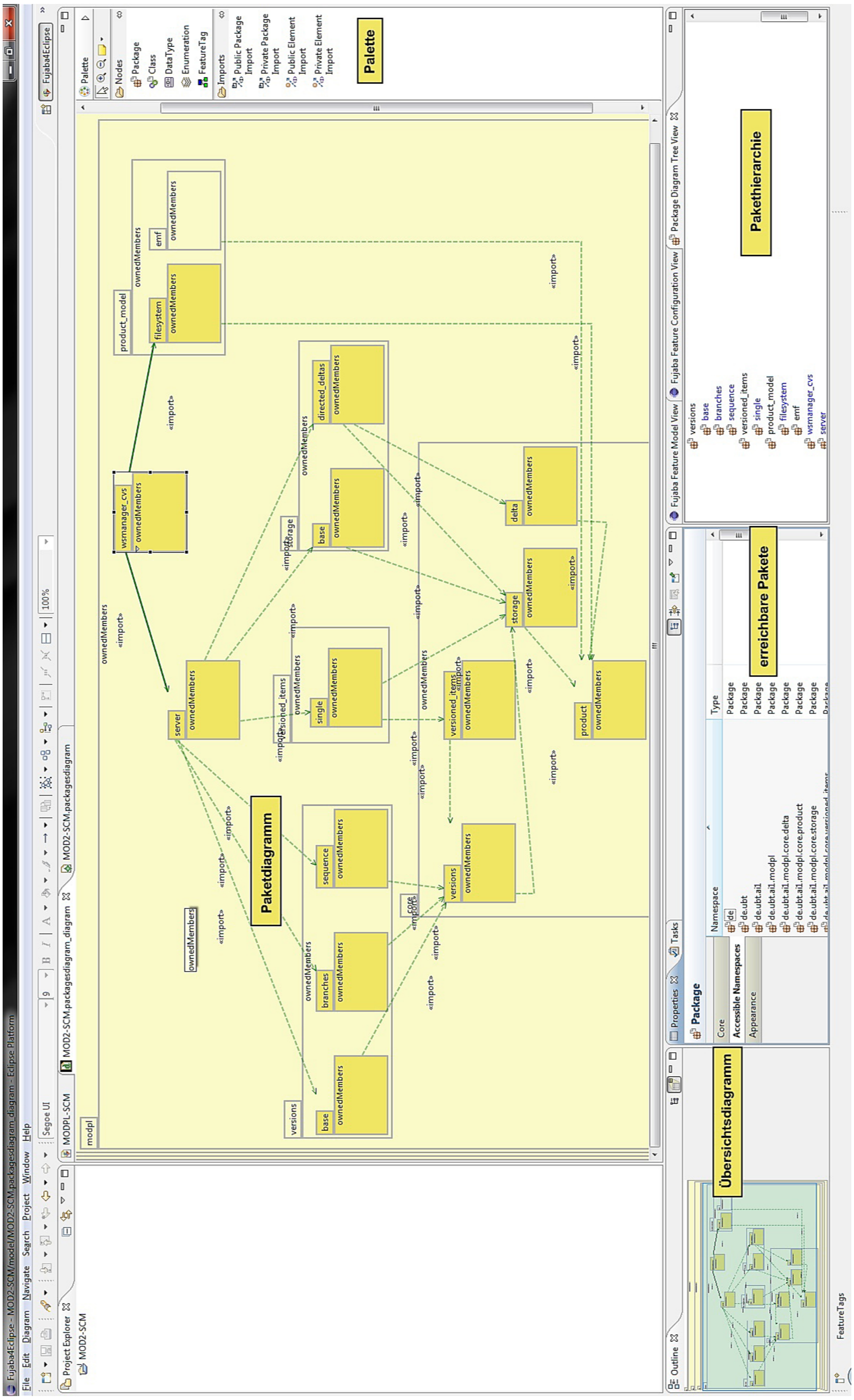

Abbildung 1.10: Paketdiagramm-Editor

Konsistenzüberprüfungen im Fujaba-Editor Wird ein Modellelement in einem Klassen- oder Storydiagramm benutzt, so wird überprüft, ob das benutzte Element gemäß dem Paketdiagramm sichtbar ist. Ist dies nicht der Fall, so wird das entsprechende Modellelement bei der Validierung farblich markiert und dem Benutzer auf diese Weise eine Verletzung der Sichtbarkeitsbedingungen signalisiert. Dem Benutzer stehen außerdem Hilfsfunktionen zur Wiederherstellung der Konsistenz durch automatisches Einfügen von entsprechenden Importbeziehungen im Paketdiagramm zur Verfügung.

Restriktiver Bearbeitungsmodus Der Fujaba-Editor kann mit Hilfe des Integrators in einen restriktiven Bearbeitungsmodus versetzt werden. In diesem Modus wird bei jeder Modelländerung geprüft, ob im Paketdiagramm spezifizierte Sichtbarkeiten durch Ausführung der Operation verletzt werden. Ist dies der Fall, so wird diese Operation angehalten und der Benutzer kann entscheiden, ob die Operation unter automatischer Änderung der Sichtbarkeiten im Paketdiagramm abgeschlossen, oder ob die Modelländerung in Fujaba abgebrochen werden soll. Als weiteres Hilfsmittel können Auswahldialoge in Fujaba dahingehend begrenzt werden, so dass darin nur sichtbare Typen zur Verfügung stehen.

1.6.6 Verbindung von Feature- und Domänenmodell

Die zweite große Lücke in der Werkzeugunterstützung betraf die *Verbindung von Feature- und Domänenmodell*. Diese Lücke wird durch das *MODPLFeaturePlugin* geschlossen, dessen Konzeption und technische Realisierung Kapitel 6 beschrieben wird.

Das MODPLFeaturePlugin besteht aus zwei Teilen: einem Editor und einem Konfigurator. Abbildung 1.11 zeigt einen Bildschirm-Schnappschuss des *Editors*. Genauer sollte man hier von einer Erweiterung des Fujaba-Editors sprechen, die die Annotation von Modellelementen mit Features unterstützt. Es sei darauf hingewiesen, dass eine analoge Erweiterung auch in den Paketdiagramm-Editor eingebaut wurde.

Um das Plugin im Editor benutzen zu können, muss zunächst ein Featuremodell geladen werden. Dies geschieht mit Hilfe eines Exports aus dem Werkzeug FeaturePlugin, das in Abschnitt 1.6.3 erläutert wurde. Das untere Fenster in Abbildung 1.11 zeigt das Ergebnis dieser Transformation in einer Baumsicht an. Nach dem Laden des Featuremodells können Modellelemente mit Features dekoriert werden. Dies geschieht in Fujaba mit Hilfe von *Annotationen*, die einer spezifischen Syntax genügen. Falls einem Modellelement mehrere Features zugeordnet werden, so gelten diese als konjunktiv verknüpft: Ein Modellelement m mit Features $f_1, \ldots, f_n$ ist nur Teil des konfigurierten Modells, wenn die entsprechende Feature-Konfiguration alle Features $f_1, \ldots, f_n$ enthält.

Der Editor bietet eine Reihe von Funktionalitäten für die *Konsistenzkontrolle zwischen Feature- und Domänenmodell* an:

Annotation nur mit deklarierten Features Es können nur Features verwendet werden,

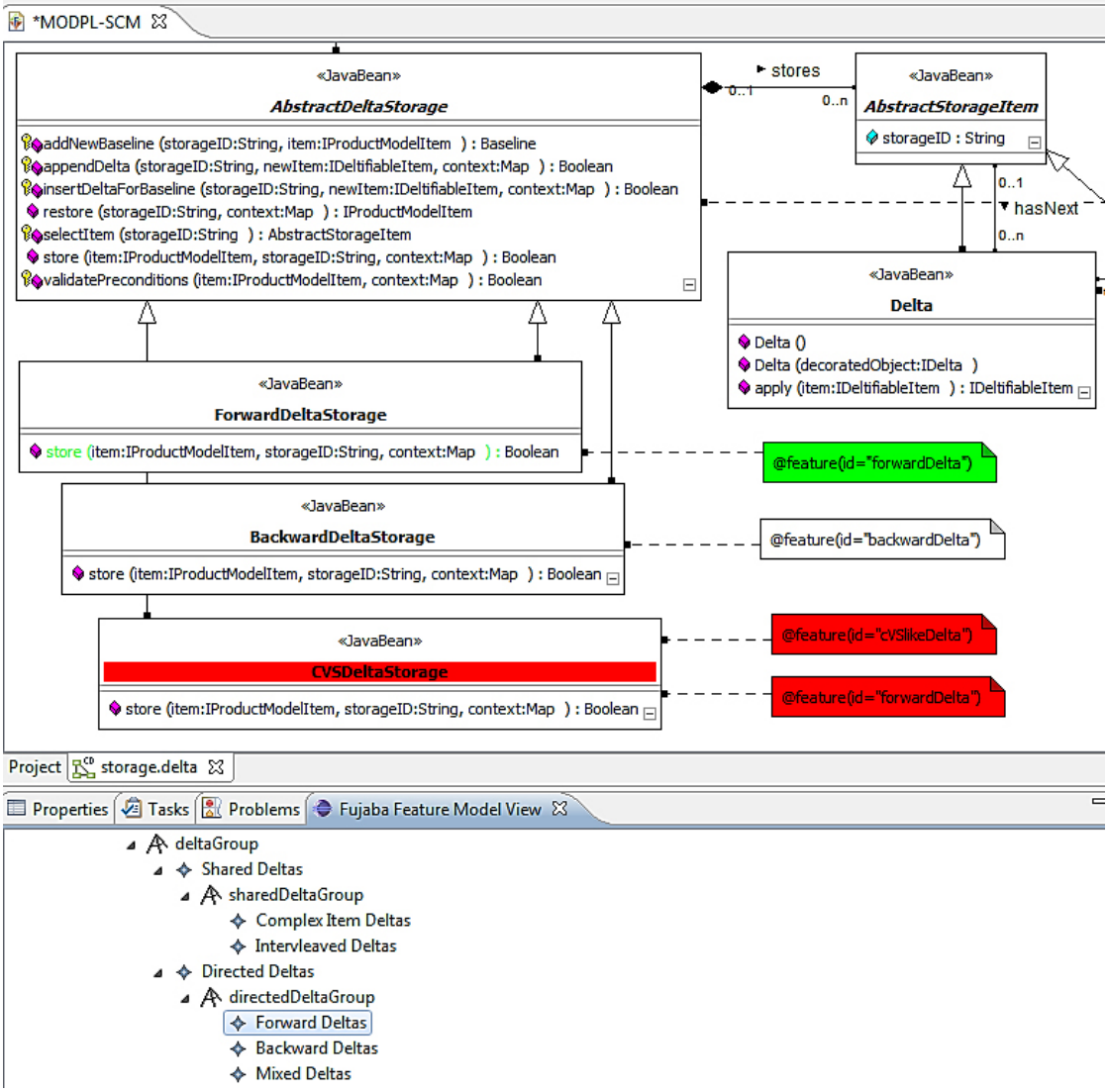

Abbildung 1.11: MODPLFeaturePlugin

die im geladenen Featuremodell auch deklariert werden. Dadurch werden z.B. Tippfehler ausgeschlossen.

Konsistenzanalysen MODPLFeaturePlugin bietet diverse Konsistenzanalysen an. Im geladenen Featuremodell werden nicht benutzte Features markiert; dadurch werden Hinweise auf möglicherweise unvollständige Domänenmodelle geliefert (falls kein Modellelement mit einem Feature f annotiert ist, ist f möglicherweise im Domänenmodell nicht realisiert). Ferner werden widersprüchliche Annotationen markiert (so ist beispielsweise die Klasse **CVSDeltaStorage** in der Mitte von Abbildung 1.11 mit Features zur Deltaspeicherung annotiert, die sich gegenseitig ausschließen). Schließlich werden mit dem Metamodell inkonsistente Annotationen erkannt (s.u.).

Reparaturaktionen Das Werkzeug bietet automatische Reparaturen von Annotationen an. So können z.B. Features auf benachbarte Modellelemente propagiert werden (s.u.), oder es können Vorschläge für Annotationen generiert werden (sind z.B. zwei Unterklassen c_1 und c_2 von c mit zwei Subfeatures f_1 und f_2 von f annotiert, so wird vorgeschlagen, der Klasse c die Annotation f zuzuordnen).

Im Folgenden wird exemplarisch auf das *Problem mit dem Metamodell inkonsistenter Annotationen* eingegangen. Eine ausführliche Diskussion des Problems und mögliche Lösungsansätze werden in Kapitel 6 dargestellt. MODPLFeaturePlugin überprüft die folgende Regel (und stellt sicher, dass sie eingehalten wird):

> Hängt ein Modellelement e_1 von einem Modellelement e_2 ab, dann darf e_1 nur selektiert werden (Teil eines konfigurierten Modells sein), wenn e_2 ebenfalls selektiert wird.

Beispielsweise darf eine Assoziation nur selektiert werden, wenn auch die Klassen an den Assoziationsenden selektiert werden. Diese Regel lässt sich wie folgt auf Feature-Annotationen übersetzen (man beachte, dass mehrere einem Element zugeordneten Features konjunktiv verknüpft sind, s.o.):

> Hängt e_1 von e_2 ab und ist e_2 mit f annotiert, dann muss auch e_1 mit f annotiert werden.

MODPLFeaturePlugin überwacht diese Regel und bietet Reparaturaktionen an, um Feature-Annotationen automatisch zu propagieren.

Die zweite Komponente von MODPLFeaturePlugin ist der *Konfigurator* für das Domänenmodell. Um ihn benutzen zu können, muss man zuvor eine Feature-Konfiguration laden. Der Domänenmodell-Konfigurator unterstützt verschiedene Arten der Abbildung einer Konfiguration. Für alle angebotenen Konfigurierungsarten gilt die oben angegebene Regel für Annotationen voneinander abhängiger Modellelemente, deren Einhaltung der Konfigurator durch Propagierung von Features sicherstellt. Die unterstützten Konfigurationsarten sind:

Visualisierung Direkt im Fujaba Editor wird dem Benutzer eine Konfiguration durch Ausblenden der nicht benötigten Modellelemente präsentiert. Somit steht dem Modellierer jederzeit ein visuelles Hilfsmittel zur Überprüfung einzelner Produktkonfigurationen zur Verfügung.

Codegenerierung Der Konfigurator stellt einen Präprozessor für den Fujaba Compiler zur Verfügung. Dabei werden dem Fujaba-Compiler nur die Modellelemente zur Codeerzeugung übergeben, die zur spezifizierten Konfiguration gehören.

Modelltransformation Der Konfigurator ist in der Lage, mittels Modelltransformation ein annotiertes Domänenmodell anhand einer spezifischen Konfiguration in ein Modell zu transformieren, das dieser Konfiguration entspricht und keine variablen Bestandteile mehr enthält.

1.6.7 Werkzeuge für domänenspezifische Sprachen

Die Entwicklung der im Rahmen dieser Arbeit entstandenen Werkzeuge erfolgte hierbei ebenfalls modellgetrieben. So wurde das Metamodell des Paketdiagrammwerkzeugs mit Hilfe von Fujaba erstellt. Da Fujaba zwar umfangreiche Möglichkeiten zur Erstellung ausführbarer Modelle besitzt, aber keine Modellierung von Benutzeroberflächen anbietet, wurde der zugehörige graphische Editor mit dem Graphical Modeling Framework (GMF) [EF09] generiert. GMF ist eng verzahnt mit dem Eclipse Modeling Framework (EMF) [ecl09], daher muss das Fujaba Modell zunächst in ein Ecore Modell transformiert werden, aus dem anschließend der Code für die statische Struktur erzeugt wird. Die Methodenimplementierungen mit Hilfe der Storydiagramme werden von Fujaba direkt in ausführbaren Java Quelltext übersetzt, der im Laufe der Codeerzeugung von EMF mit den statischen Anteilen vermischt wird.

1.7 Verwandte Arbeiten

Bisherige Arbeiten auf diesem Gebiet konzentrieren sich auf einzelne Aspekte der Entwicklung. Im Gegensatz dazu ist im Rahmen dieser Arbeit eine integrierte Umgebung zur modellgetriebenen Entwicklung von Softwareproduktlinien entstanden, wie in Abbildung 1.9 ersichtlich ist. Bisherige Ansätze und Werkzeuge behandeln entweder nur die Entwicklung von Softwareproduktlinien bei denen die Artefakte in Form von Quelltext vorliegen, so etwa das kommerzielle Produkt *pure::variants*[2]. Im Bereich der modellgetriebenen Entwicklung von Produktlinien bietet *pure::variants* lediglich eine Anbindung an das kommerzielle Werkzeug *Sparx Enterprise Architect*[3], das aber keine Unterstützung für Modellieren im Großen – so wie im Rahmen dieser Arbeit vorgestellt – bietet. Im Rahmen von UML Modellen haben Czarnecki et. al einen Ansatz vorgestellt, Features auf UML Modellelemente abzubilden [CA05]. Hier werden aber nur einfache Konsistenzbedingungen aufgestellt. Eine Unterstützung für Modellieren im Großen existiert ebensowenig wie die Möglichkeit, direkt ausführbaren Quelltext zu erzeugen. Auch Heidenreich et. al beschäftigen sich mit der Kombination von modellbasierter Entwicklung und Softwareproduktlinien. Das Werkzeug *Featuremapper* [HKW08] bildet Elemente von Featuremodellen auf Ecore kompatible Modelle ab und erlaubt daraus die Erzeugung von konfigurierten Modellen. Allerdings kann die Wohlgeformtheit bzw. syntaktische Korrektheit der erzeugten Modelle nicht sichergestellt werden, da keinerlei Konsistenzbedingungen existieren.

Verwandte Arbeiten werden in den jeweiligen Kapiteln zum Paketdiagramm-Werkzeug (Kapitel 5) und zur Verbindung von Feature- und Domänenmodell (Kapitel 6) ausführlich diskutiert. Desweiteren beschäftigt sich Kapitel 4 mit der modellgetriebenen Entwicklung von domänenspezifischen Sprachen inklusive zugehöriger Editoren, da in der Entwicklung von Softwareproduktlinien häufig auf domänenspezifische Sprachen zurückgegriffen wird. Während im Bereich der Abbildung von Elementen des Featuremodells auf Domänenmodellelemente zahlreiche verwandte Ansätze existieren, erfolgt der

[2] http://www.pure-systems.com
[3] http://www.sparxsystems.eu

Vergleich des Paketdiagrammansatzes zum Modellieren im Großen auf Basis bereits verfügbarer Werkzeuge.

Im Gegensatz zur vorliegenden Arbeit, die sich mit Modellen und Werkzeugen zur modellgetriebenen Entwicklung von Softwareproduktlinien beschäftigt, beschreibt Alexander Dotor in seiner Dissertation [Dot10] die konkrete Realisierung einer Produktlinie für Versionskontrollsysteme mit Hilfe dieser Werkzeuge. In seiner Arbeit wird die Domäne ausführlich analysiert und das Ergebnis dieser Analyse in einem umfassenden Featuremodell beschrieben. Desweiteren wird dargelegt, wie die einzelnen Bestandteile von Versionskontrollsystemen entkoppelt werden können, so dass eine weitgehende Orthogonalität erreicht werden kann.

1.8 Zusammenfassung

Auch in der heutigen Zeit existieren keine hinreichenden Lösungen, um der Softwarekrise effizient zu begegnen. Gerade in Zeiten der weltweiten Wirtschaftskrise ist es aber unerlässlich, neue Wege zur Minimierung der Kosten der Softwareentwicklung bei gleichzeitiger Steigerung der Softwarequalität zu beschreiten. Diese Arbeit beschreibt Konzepte, um zwei bekannte und über lange Jahre erfolgreich eingesetzte Techniken - Softwareproduktlinien und modellgetriebene Softwareentwicklung - effizient miteinander zu kombinieren. Zu diesem Zwecke wurde eine umfassende Werkzeugunterstützung durch Wiederverwendung bereits existierender Werkzeuge geschaffen, die durch sinnvolle Neuentwicklungen ergänzt wurden, um signifikante Lücken in der Werkzeugkette zu schließen.

So wurde das Paketdiagramm-Werkzeug erstellt, das umfassende Unterstützung für die Disziplin *Modellieren im Großen* bietet, einem Bereich, der von bisherigen auf dem Markt befindlichen Werkzeugen nur unzureichend (vor allem im Hinblick auf die Anforderungen bei der Erstellung einer Softwareproduktlinie) unterstützt wird.

Desweiteren wurden Werkzeuge geschaffen, die einerseits die Abbildung von Elementen des Featuremodells auf Domänenmodellelemente ermöglichen, und andererseits aus einem annotierten Domänenmodell konkrete Produktvarianten ableiten können. Dabei wurden umfassende Konsistenzbedingungen ermittelt, die von den angebotenen Werkzeugen realisiert werden.

1.9 Übersicht

Kapitel 2 gibt einen detaillierten Überblick über Softwareproduktlinien. Die unterschiedlichen Ebenen Domänenentwicklung und Anwendungsentwicklung werden genauer beleuchtet und das grundlegende Konzept der Variabilität wird erklärt. Weiterhin wird das im Rahmen dieser Arbeit verwendete Werkzeug zur Erstellung des Featuremodells vorgestellt.

Kapitel 3 beschreibt ausführlich die unterschiedlichen Konzepte der modellgetriebenen Softwareentwicklung und speziell den MDA-Ansatz der OMG. In diesem Kapitel wird außerdem das im Rahmen dieser Arbeit eingesetzte CASE-Werkzeug Fujaba [Zün02] vorgestellt und dessen Mehrwert gegenüber anderen UML-Werkzeugen diskutiert.

Kapitel 4 widmet sich domänenspezifischen Sprachen und deren Einsatz im Bereich der Entwicklung von Softwareproduktlinien. Es wird außerdem der Bezug zum Ansatz der generativen Programmierung [CE00] hergestellt. Abschließend wird eine Methode aufgzeigt, um mit Hilfe von Fujaba und Rahmenwerken aus der Eclipse Umgebung graphische Editoren modellgetrieben zu entwickeln. Auf diese Weise kann sowohl die abstrakte als auch die konkrete Syntax von domänenspezifischen Sprachen durch Modelle beschrieben werden, und entsprechende Werkzeuge können automatisch aus dieser formalen Beschreibung generiert werden.

In Kapitel 5 wird die Disziplin Modellieren im Großen vorgestellt und die Notwendigkeit von geeigneter Werkzeugunterstützung dafür motiviert. Das im Rahmen dieser Arbeit erstellte Werkzeug zur Unterstützung von Modellieren im Großen wird detailliert beschrieben und ausführlich mit bekannten und weit verbreiteten Werkzeugen verglichen. Die in Kapitel 4 vorgestellten Techniken zur modellgetriebenen Entwicklung graphischer Werkzeuge werden zur Erstellung des Paketdiagrammeditors eingesetzt.

Die Kopplung von Featuremodell und ausführbarem Domänenmodell wird in Kapitel 6 beschrieben. Zunächst wird ein allgemeines auf UML basierendes Konzept zur Realisierung der Kopplung vorgestellt. Anschließend werden umfangreiche Bedingungen formuliert, um die Konsistenz der konfigurierten Domänenmodelle sicherzustellen. Nach der Beschreibung der Realisierung des Werkzeugs erfolgt ein ausführlicher Vergleich mit verwandten Ansätzen.

Die Arbeit wird durch einen ausführliche Zusammenfassung und einen Ausblick auf künftige mögliche Forschungsarbeiten in diesem Kontex abgeschlossen.

2 Produktlinien

In unserem täglichen Leben werden wir ständig mit unterschiedlichen Formen von Produktlinien konfrontiert. So entspringen beispielsweise Fahrzeuge, Hardwareartikel oder gar Produkte von bekannten Fastfood-Ketten einer Produktlinie. Dies folgte aus der Erkenntnis, dass ähnliche Produkte immer wiederkehrende Produktionsabläufe verlangen. Um diese Abläufe zu verkürzen und damit Kosten einzusparen, wurde das Konzept der geplanten Wiederverwendung durch Produktlinien entwickelt. Im Bereich der Softwaretechnik ist das Konzept der Softwareproduktlinien in der industriellen Praxis noch nicht so weit verbreitet, wie in anderen Bereichen, z.B. der Fertigung. Wohl aber ist in den letzten Jahren ein zunehmender Trend hinsichtlich der Entwicklung von Softwareproduktlinien auszumachen, was auch durch zahlreiche Konferenzen zu diesem Thema mit großer Beteiligung von Industriepartnern manifestiert wird.

Das Software Engineering Institute (SEI) der Carnegie Mellon Universität definiert den Begriff Softwareproduktlinie wie folgt:

> „Eine Softwareproduktlinie ist eine Menge von softwareintensiven Systemen, die eine gemeinsame, verwaltete Menge an Features teilen, die den spezifischen Bedürfnissen eines speziellen Marktsegments genügen und die auf Basis einer Menge an Kernkomponenten nach einer bestimmten Vorschrift entwickelt wurden.“[CN01], S. 5

Neben der Einsparung von Entwicklungskosten werden aber noch weitere Ziele durch die Entwicklung von Softwareproduktlinien verfolgt. Im Gegensatz zur Massenproduktion, bei der jeder Kunde dasselbe Produkt erhält, wird hier der Ansatz der sog. *Mass-Customization* verfolgt, also der Möglichkeit, jedem Kunden ein Produkt bereit zu stellen, das seinen spezifischen Anforderungen genügt. Da das Kernkonzept der Produktlinien auf geplanter Wiederverwendung der Kernkomponenten (der sog. *Core Assets*) basiert, ist eine höhere Produktqualität zu erwarten, da auf bekannten und bewährten Technologien aufgebaut werden kann. Dies hat außerdem eine höhere Produktivität durch effizientere Ausnutzung der vorhandenen Ressourcen zur Folge und somit kürzere Entwicklungszeiten für die Produkte.

Zu Beginn des Kapitels werden zunächst die Entwicklungsprozesse im Kontext von Softwareproduktlinien vorgestellt. Anschließend werden Wege zur Erfassung von Variabilität vorgestellt. Danach folgt eine ausführliche Beschreibung des im Rahmen dieser Arbeit verwendeten Modellierungsansatzes.

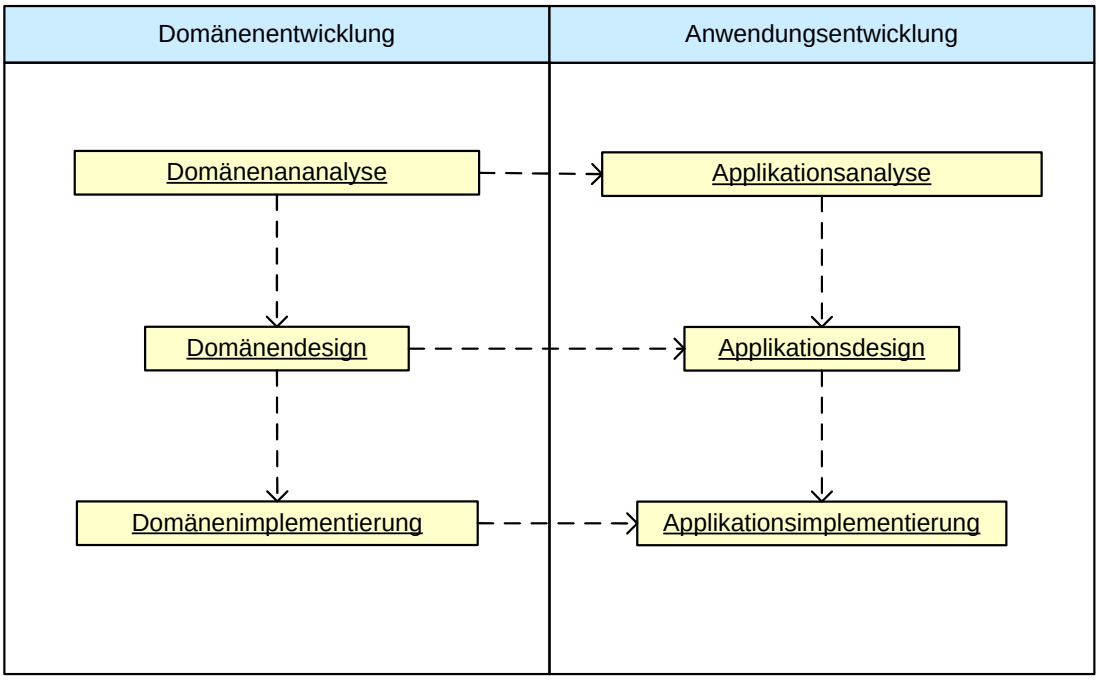

Abbildung 2.1: Referenzprozess der Softwareproduktlinienentwicklung nach [Lin02].

2.1 Entwicklungsprozesse

Bei der Entwicklung von Softwareproduktlinien werden die beteiligten Prozesse grob in zwei unterschiedlichen Ebenen klassifiziert: *Domänenentwicklung (Domain Engineering)* und *Anwendungsentwicklung (Application Engineering)*, wie Abbildung 2.1 zeigt.

2.1.1 Domänenentwicklung

Die Ziele in der Domänenentwicklung sind die Festlegung der gemeinsamen und variablen Teile der Softwareproduktlinie, sowie die Definition von deren Umfang ([PBL05], S. 23). Hierbei wird die Menge der Anwendungen festgelegt, die mit Hilfe der Produktlinie erstellt werden können. Weiterhin werden in dieser Disziplin die wiederverwendbaren Artefakte ermittelt und erstellt, mit Hilfe derer die gewünschte Variabilität erreicht wird. Van der Linden [Lin02] nennt als Teilprozesse der Domänenentwicklung:

Domänenanalyse Mit Hilfe der Domänenanalyse wird der Anwendungsbereich definiert und es werden die domänenrelevanten Informationen in einem Domänenmodell erfasst, das sowohl die Gemeinsamkeiten als auch die Unterschiede enthält.

Domänendesign Das Domänendesign dient der Festlegung einer generischen Architektur für die Softwareproduktlinie.

Domänenimplementierung Hier steht die Implementierung der wiederverwendbaren Bestandteile auf Basis der in den ersten Schritten erstellten Modelle im Vordergrund.

Die Domänenanalyse wird dabei im Kontext dieser Arbeit mittels der Feature-orientierten Domänenanalyse (kurz FODA) nach Kang et. al [KCH+90] durchgeführt. Diese Technik ist die Basis für eine Vielzahl von davon abgeleiteten Methoden zur Domänenanalyse, wie etwa FORM (Feature-oriented reuse method [KKL+98, KLD02]) oder der Einbeziehung von Kardinalitäten von Features (Cardinality-based feature models [CHE05]). Diese Methode wurde ausgewählt, da sie am ausgereiftesten und zudem sehr gut dokumentiert ist. Auch in der industriellen Anwendung hat FODA eine große Akzeptanz, was sich unter anderem dadurch äußert, dass kommerzielle Werkzeuge zur Produktlinienentwicklung, wie etwa *pure::variants*[1], diesen Ansatz unterstützen. FODA benutzt die Modellierungskonzepte *Abstraktion* und *Verfeinerung*, um die gemeinsamen bzw. unterschiedlichen Features von Systemen zu identifizieren und daraus hochgradig wiederverwendbare Produkte ableiten zu können. Mittels Abstraktion werden zunächst die generischen Bestandteile der Softwareproduktlinie für die Domäne entwickelt. Dabei werden alle speziellen Eigenschaften, in denen sich die einzelnen Produkte unterscheiden durch Abstraktion eliminiert. In der Anwendungsentwicklung (siehe Abschnitt 2.1.2) werden konkrete Produkte anschließend durch Verfeinerung dieser generischen Bestandteile erstellt. Das Ergebnis der Domänenanalyse wird in einem sogenannten *Featuremodell* (siehe Abschnitt 2.3.1) erfasst.

2.1.2 Anwendungsentwicklung

Der Begriff *Anwendungsentwicklung* bezeichnet den Vorgang, konkrete Produkte auf Basis der Produktlinie zu entwickeln. Dabei werden die Artefakte benutzt, die das Resultat der Disziplin *Domänenentwicklung* sind, um den Ansatz der geplanten Wiederverwendung zu nutzen. Der Unterschied der Aktivitäten in der Domänenentwicklung im Vergleich zur Anwendungsentwicklung liegt darin, wiederverwendbare Artefakte für eine Klasse von Systemen bereitzustellen, während die Anwendungsentwicklung die Entwicklung einer optimalen Lösung für ein einzelnes System zum Ziel hat.

Anwendungsentwicklung hat folgende Ziele (siehe [PBL05], S. 30f):

Wiederverwendung Es soll ein möglichst hoher Grad an Wiederverwendung von Domänenartefakten erfolgen, wenn ein Produkt aus der Produktlinie entwickelt wird.

Gemeinsamkeiten und Unterschiede die in der Produktlinie realisiert sind sollen während der Produktentwicklung ausgenutzt werden.

Dokumentation der Artefakte der Applikation und deren Verbindung zu den Domänenartefakten.

Binden von Variabilität anhand der spezifischen Anforderungen der Applikation.

[1] http://www.pure-systems.com

Abschätzen der Auswirkungen von Unterschieden zwischen Anforderungen der Applikation und der Domäne auf die Architektur, die Komponenten und Tests.

Die Teilprozesse nach van der Linden [Lin02] umfassen hierbei:

Applikationsanalyse Die spezifischen Anforderungen an das konkrete Produkt werden hier ermittelt. Der erreichbare Grad der Wiederverwendung hängt stark von diesen Anforderungen ab, daher ist der Hauptaugenmerk das Erkennen von Unterschieden zwischen den spezifischen Produktanforderungen und den Eigenschaften der Produktlinie.

Applikationsdesign In diesem Schritt wird die Architektur des Produkts erstellt. Dafür wird die Referenzarchitektur verwendet, und die benötigten Teile daraus ausgewählt, konfiguriert und mit produktspezifischen Anpassungen ergänzt.

Applikationsimplementierung Zuletzt werden in der Implementierungsphase die wiederverwendbaren Softwarekomponenten ausgewählt und konfiguriert. Darüberhinaus werden auch die spezifischen Erweiterungen implementiert. Diese einzelnen Bestandteile werden abschließend zum finalen Produkt kombiniert.

2.2 Erfassen von Variabilität

In der Entwicklung von Softwareproduktlinien treten unterschiedliche Typen von Variabilität auf, die während der Entwicklung nicht isoliert voneinander betrachtet werden können. Die unterschiedlichen Typen von Variabilität sind in [BKPS04] beschrieben und umfassen:

Variabilität in Features Konkret bedeutet dies, dass aus der Produktlinie Produkte mit unterschiedlichen Eigenschaften bzw. Funktionsumfang abgeleitet werden können. So können in der Produktlinie für Softwarekonfigurationsverwaltungssysteme beispielsweise einstufige oder zweistufige Versionsbäume verwendet werden.

Variabilität in Abläufen Dies bedeutet, dass auf Basis einer Produktlinie mit gleichen Features (z. B. zweistufiger Versionsbaum) Produkte in Prozessschritten varriieren, aber das gleiche Ergebnis liefern. So kann beispielsweise das Erstellen einer neuen Version im Versionsbaum zuvor ein explizites Sperren der Vorgängerversion erfordern oder auch nicht. Durch Variabilität in Abläufen können gewisse Funktionen an unternehmensspezifische Anforderungen angepasst werden.

Variabilität im Datenumfang Auf Basis einer Produktlinie können Produkte mit unterschiedlichen Datenobjekten und Attributen abgeleitet werden. Zum Beispiel können für eine *checkin*-Operation in einem Versionskontrollsystem zusätzlich zur aktuellen Version weitere Daten, wie etwa ein Kommentar, der die Änderungen beschreibt, erforderlich sein.

Variabilität im Datenformat Auf Basis einer Produktlinie können Produkte mit unterschiedlichen Datenformaten und/oder -typen je Attribut abgeleitet werden. Bei Versionskontrollsystemen können Unterschiede zwischen Versionen von Textdateien z. B. zeichen- oder zeilenbasiert erfasst werden.

Variabilität im Systemzugang Diese Art von Variabilität besagt, dass auf Basis einer Produktlinie Produkte mit Unterschieden hinsichtlich der Zugangsart zum gleichen System abgeleitet werden können. So kann ein Versionskontrollsystem den Zugang beispielsweise über einen Internetbrowser, einer Erweiterung für den Dateisystemmanager oder auch durch spezielle Erweiterungen von Entwicklungsplattformen (wie etwa Eclipse) zur Verfügung stellen, um so Benutzern mit unterschiedlichen Präferenzen die jeweiligen Systemzugänge anzubieten.

Variabilität in Benutzerschnittstellen Mit Hilfe dieser Art der Variabilität lassen sich Produkte auf Basis einer Produktlinie ableiten, die mit unterschiedlichen Benutzerschnittstellen versehen sind. So kann beispielsweise der Versionsgraph von Versionskontrollsystemen rein textuell, als Baumsicht, oder in einer Repräsentation, die aus Knoten und Kanten besteht dargestellt werden.

Variabilität in Systemschnittstellen Dies bedeutet konkret, dass Produkte mit unterschiedlichen Schnittstellen zu Legacy- oder Drittsystemen auf Basis einer Produktlinie erstellt werden können. Soll ein Versionskontrollsystem im Rahmen eines Systems zur kontinuierlichen Integration[2] verwendet, oder Werkzeugen für integrierte Softwaremessung [Dau08] zugänglich gemacht werden, so sind dafür unterschiedliche Schnittstellen erforderlich.

Variabilität in der Qualität Diese Art von Variabilität bedeutet, dass auf Basis einer Produktlinie Produkte gebildet werden können, die unterschiedlichen Qualitätsansprüchen genügen. In diesem Kontext bedeutet Qualität eine Klassifikation hinsichtlich Verfügbarkeit, Sicherheit, Leistung usw. So kann ein Versionskontrollsystem, das für Softwareprojekte von mehreren hundert Mitarbeitern, die an weltweit verteilten Standorten arbeiten, andere Anforderungen an Wartungsfenster und Reaktionszeiten stellen, als ein System, das nur an einem lokalen Standort mit wenigen Mitarbeitern eingesetzt wird.

Wie bereits in den vorigen Unterabschnitten ausgeführt wurde, werden in der Disziplin *Domänenentwicklung* die Artefakte der Produktlinie definiert, die die Variabilität realisieren. In der *Anwendungsentwicklung* werden diese Artefakte zu individuellen Produkten durch gezielte Wiederverwendung kombiniert. Um dies zu erreichen, ist eine explizite Darstellung der Variabilität zwingend notwendig. Daher werden diese unterschiedlichen Typen der Variabilität in einem allgemeinen Variabilitätsmodell erfasst. Dieses Modell verwendet *Variationspunkte* und *Varianten* [BKPS04, PBL05]. Pohl et. al

[2] Der Begriff kontinuierliche Integration (engl. continous integration) beschreibt den Prozess des regelmäßigen vollständigen bildens und testen einer Applikation. Mit dieser Vorgehensweise können größere Änderungen inkrementell in kleinen Schritten in ein Repositorium eingespielt und etwaige Integrationsprobleme zeitnah entdeckt werden.

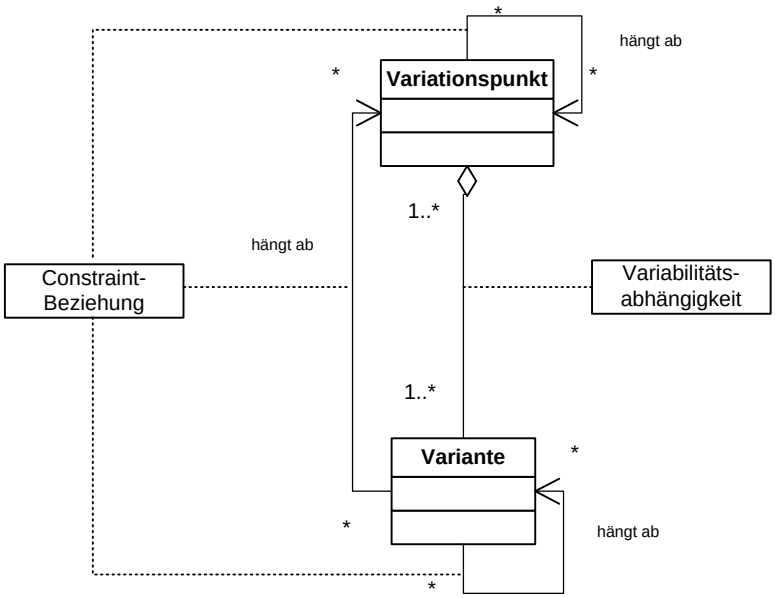

Abbildung 2.2: allgemeines Variabilitätsmodell nach [BKPS04].

[PBL05] formulieren drei Fragen, die bei der Charakterisierung von Variabilität in einer Softwareproduktlinie hilfreich sind:

1. Was variiert?
2. Warum variiert es?
3. Wie variiert es?

Hierbei unterscheiden die Autoren die Begriffe *Variabilitätssubjekt* und *Variabilitätsobjekt*. Während ein Variabilitätssubjekt einen variablen Gegenstand der realen Welt oder variable Eigenschaft eines solchen repräsentiert, stellt ein Variabilitätsobjekt eine entsprechende Instanz eines Variabilitätssubjekts dar (vgl. [PBL05], S. 60). Während Jacobson et al. [JGJ97] den Begriff *Variationspunkt* als „ein oder mehrere Stellen, an denen Variation auftritt“ bezeichnen, formulieren Pohl et al. den Begriff allgemeiner, als „... Repräsentation eines Variabilitätssubjekts innerhalb von Domänenartefakten, die durch Kontextinformationen angereichert wird“. Folglich stellt eine Variante „... eine Repräsentation eines Variabilitätsobjekts innerhalb von Domänenartefakten“ dar. Abbildung 2.2 zeigt das allgemeine Variabilitätsmodell inklusive der Abhängigkeiten von Variationspunkten und Varianten nach [BKPS04, BFG+01]. Hierbei beschreibt die Aggregation *Variabilitätsabhängigkeit* die Art, wie eine Variante einem Variationspunkt zugeordnet ist, also beispielsweise optional, alternativ, mit gegenseitigem Ausschluß, oder obligatorisch.

Außerdem werden zwei Arten von Constraintbeziehungen von Abhängigkeiten zwischen Variationspunkten oder Varianten untereinander unterschieden: (1) **Requires-**

Beziehungen, die besagen, dass eine Variante bzw. ein Variationspunkt eine andere Variante bzw. einen anderen Variationspunkt benötigen und (2) **Excludes-Beziehungen**, die einen gegenseitigen Ausschluß repräsentieren.

2.3 Gewählter Modellierungsansatz

Im Kontext dieser Arbeit wurden Featuremodelle zur Beschreibung von varibalen und invarianten Bestandteilen von Softwareproduktlinien eingesetzt. Featuremodelle wurden von Kang et al. im Rahmen der Feature orientierten Domänenanalyse (FODA) eingeführt [KCH+90]. Gegenüber dem allgemeinen Variabilitätsmodell (vgl. Abbildung 2.2) bieten Featuremodelle eine deutlich höhere Ausdrucksmächtigkeit, vor allem durch die Möglichkeit, komplexe Hierarchien durch Schachtelung von Features darzustellen [Bat05]. So werden etwa Beziehungen zwischen Vater- und Kindfeatures mit unterschiedlicher Semantik hinsichtlich der Auswahl bei der Konfigurierung definiert: (1) *und* bedeutet, dass alle Kindelemente ausgewählt werden müssen, (2) *exklusives oder* bedeutet, dass nur eine Alternative ausgewählt werden darf, (3) *inklusives oder* beschreibt die Auswahl von einem oder mehreren Kindelementen, zusätzlich gibt es noch (4) *obligatorische* und (5) *optionale* Kinder.

Weiterhin ist FODA sehr ausgereift, gut dokumentiert und weit verbreitet. Es ist anzumerken, dass für die modellgetriebene Entwicklung von Softwareproduktlinien nach dem in dieser Arbeit beschriebenen Verfahren ebenso aus FODA hervorgegangene Methoden verwendet werden können, solange das Ergebnis der Domänenanalyse in einem Featuremodell erfasst werden kann.

2.3.1 Featuremodell

Zur Erstellung des Featuremodells für die Produktlinie von Softwarekonfigurationsverwaltungssystemen wurde die Software „FeaturePlugin“ von Michal Antkiewicz und Krzysztof Czarnecki [AC04] verwendet. FeaturePlugin ist eine Eclipse Erweiterung, die es dem Benutzer erlaubt, Featurediagramme und darauf aufbauende Konfigurationen zu erstellen. Alle invarianten und veränderlichen Bestandteile einer Softwarefamilie werden in diesem Modell erfasst. Zur Beschreibung eines Featuremodells werden im Wesentlichen zwei Arten von Elementen eingesetzt: Features und Featuregruppen. Mittels Featuregruppen lassen sich unterschiedliche Abhängigkeitsbeziehungen von übergeordneten und untergeordneten Features ausdrücken, wie etwa Alternativen oder Mehrfachauswahl. Czarnecki und Antkiewicz verwenden in ihrem Werkzeug eine von ihnen entwickelte Variante von FODA, die zusätzlich die Angabe von Kardinalitäten an einzelnen Features erlaubt [CHE05].

Das in Abbildung 2.3 abgebildete Featuremodell zeigt einen kleinen Ausschnitt aus einem vereinfachten Featuremodell der MOD2-SCM Produktlinie. Das komplette Featuremodell der MOD2-SCM Produktlinie wird ausführlich in der Dissertation von Alexander Dotor erklärt [Dot10]. Zum Verständnis der Konzepte und Vorgehensweisen ist im

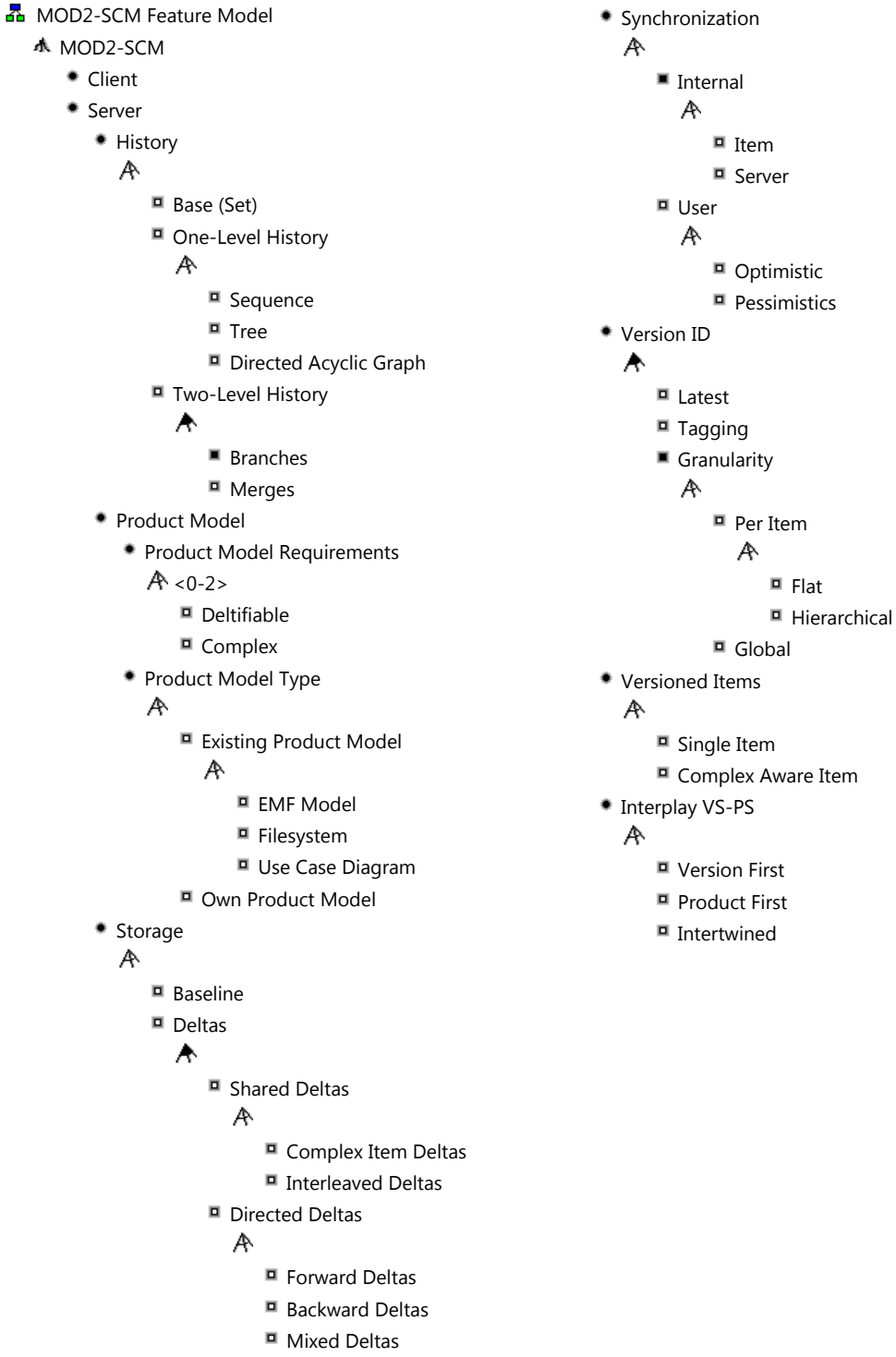

Abbildung 2.3: Ausschnitt eines vereinfachten Featuremodells der MOD2-SCM Produktlinie.

Rahmen der hier vorliegenden Arbeit ein stark vereinfachtes Featuremodell ausreichend. Die in Abschnitt 1.5 eingeführten Konzepte der Softwarekonfigurationsverwaltung sind in diesem einfachen Beispiel erfasst.

Jedes Featuremodell besitzt genau ein Wurzelelement, das hier mit dem Namen *MOD2-SCM* bezeichnet ist. Da in der MOD2-SCM Produktlinie ein Softwarekonfigurationsverwaltungssystem derzeit aus einem Server mit einem zugehörigen Client besteht, enthält das Wurzelelement zwei direkte Kinder: Client und Server, die beide obligatorisch sind (dargestellt durch einen schwarz ausgefüllten Kreis). Optionale Vaterfeatures werden in dem Werkzeug durch einen nicht ausgefüllten Kreis dargestellt (im Beispiel nicht vorhanden). Obligatorische und optionale Kindfeatures werden durch ausgefüllte bzw. nicht ausgefüllte Quadrate repräsentiert. Im vorliegenden Beispiel wird nur die Serverkomponente erläutert, daher wurde der zu dem Feature Client gehörende Teilbaum im Beispiel ausgeblendet. Die zentrale Komponente von Softwarekonfigurationsverwaltungssystemen ist die Versionskontrolle. Die dafür notwendigen Features, die ein entsprechender Server bereitstellen muss, sind:

History Das Feature History beschreibt die Art und Weise, wie die Versionsgeschichte von versionierten Objekten verwaltet wird (siehe Abschnitt 1.5.2). Der Ausschnitt aus dem Featuremodell in Abbildung 2.3 zeigt hier die einfache Speicherung von Versionen in einer Menge (*Base (Set)*), als einstufige (*Sequence*, *Tree* oder *Directed Acyclic Graph*) oder zweistufige Entwicklungsgeschichte, die Zweige (*Branches*) und Verschmelzungen (*Merges*) bietet. Während die Kindelemente des Features *One-Level History* Alternativen darstellen (also ein exklusives Oder darstellen), kann bei der zweistufigen Entwicklungsgeschichte zwischen Verzweigungen oder Verzweigungen und Verschmelzungen (also ein inklusives Oder) gewählt werden.

Product Model Das Feature Product Model beschreibt das Produktmodell, aus dem die versionierten Objekte stammen. Hier kann einerseits auf eines der drei bereits im Rahmen von MOD2-SCM realisierten Produktmodelle (Dateisystem, EMF Modell oder Anwendungsfalldiagramm) zurückgegriffen werden, oder es kann ein eigenes Produktmodell definiert werden. Hierfür können die spezifischen Anforderungen der „Möglichkeit der Erzeugung von Deltas“ bzw. die „Möglichkeit der Speicherung von komplexen Objekten“ gewählt werden. Der Unterschied in der Darstellung des entsprechenden Symbols (verglichen mit dem inklusiven Oder bei der zweistufigen Entwicklungsgeschichte) liegt in der Kardinalität begründet. Während hier die Anforderungen an das Produktmodell durch optionale Kindfeatures (d.h. die Kardinalität der Kinder beträgt [0..*]) ausgedrückt werden, ist bei der zweistufigen Entwicklungsgeschichte die Auswahl des Kindes *Branches* obligatorisch (d.h. die Kardinalität der Kinder beträgt hier [1..*]).

Storage Das Feature Storage beinhaltet die derzeit im Rahmen von MOD2-SCM implementierten Arten der Speicherung des Inhalts von versionierten Objekten (siehe Abschnitt 1.5.3). Es kann entweder der komplette Inhalt für jede Version abgelegt

werden (*Baseline*), oder es können unterschiedliche Varianten der Speicherung von Unterschieden zwischen den einzelnen Versionen ausgewählt werden.

Synchronization Das Feature Synchronization erlaubt die Wahl von unterschiedlichen Arten der Synchronisation, wie etwa pessimistische oder optimistische Sperren (siehe Abschnitt 1.5.7).

Version ID Hier werden die unterschiedlichen Möglichkeiten zur Vergabe von Versionsbezeichnern erfasst. Zum einen kann die Granularität festgelegt werden (ein Versionsbezeichner für jedes Objekt, oder ein globaler Versionsbezeichner). Zum anderen können mehrere versionierte Objekte mit Markierungen versehen werden (*Tagging*) oder es kann die jeweils letzte Version eines versionierten Objekts speziell ausgezeichnet werden (*Latest*).

Versioned Items Bei der Auswahl von versionierten Objekten unterscheidet die MOD2-SCM Produktlinie zwischen einzelnen (atomaren) Objekten und komplexen Objekten, die durch Komposition zusammengesetzt werden (etwa Verzeichnisse, oder Diagrammelemente).

Interplay VS-PS Die Auswahl der Objekte kann nach den in Abschnitt 1.5.4 vorgestellten Konzepten erfolgen.

Das Ziel bei der Erstellung des Featuremodells war, eine möglichst *orthogonale Kombination* von Produkteigenschaften in den einzelnen Konfigurationen zu ermöglichen. An dem in Abbildung 2.3 dargestellten Featuremodell der MOD2-SCM Produktlinie, erkennt man, dass die Produkteigenschaften *History* und *Storage* oder auch *Storage* und *Product Model* sowie *Storage* und *Version* voneinander unabhängig sind und somit beliebig kombiniert werden können. Abbildung 2.4 verdeutlicht die orthogonal zum Versionsgraphen wählbaren Arten der Deltaspeicherung.

Allerdings gibt es noch inhärent gekoppelte Eigenschaften, die nicht aus dem Featuremodell ersichtlich sind. So erfolgt beispielsweise die Berechnung der hierarchischen Versionsnummern *VersionID* → *Granularity* → *Per Item* → *Hierarchical* durch Analyse des Versionsbaums. Dieser Umstand kann im Featuremodell durch Angabe von *Constraints* ausgedrückt werden, etwa, dass eine Wahl der hierarchischen Vergabe von Versionsnummern eine automatische Auswahl des Features für den Versionsbaum bedingt. Die gängigsten Anwendungsbeispiele für Constraints, die nicht durch Kardinalitäten von Features oder Featuregruppen ausgedrückt werden können sind *Implikation* und *Ausschluss* [AC04]. Bedingt durch das vom Werkzeug verwendete Metamodell können diese Bedingungen im FeaturePlugin durch XPath 2.0 [Wor07] Ausdrücke realisiert werden. Ein Constraint, das die Wahl des Features für eine baumartige Versionsspeicherung impliziert, sofern das Feature für die hierarchische Vergabe der Versionsnummern gewählt wurde sieht wie folgt aus:

```
if (//hierarchical) then (//tree) else true();
```

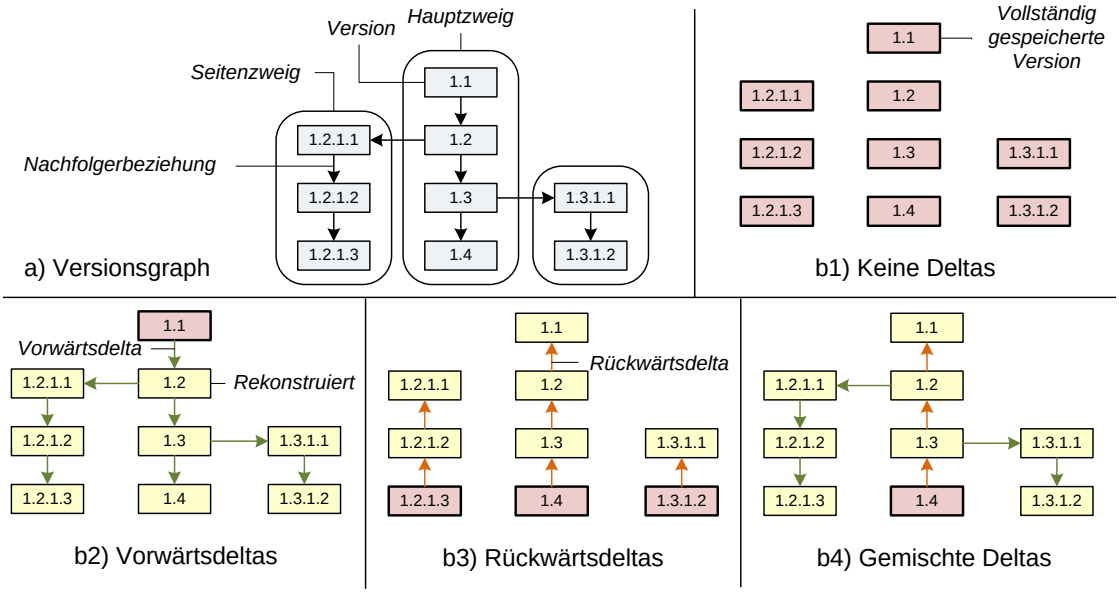

Abbildung 2.4: Mögliche Art der Versionsspeicherung und davon unabhängige Arten der Deltaspeicherung.

Dabei werden die einzelnen Knoten über ihren eindeutigen Bezeichner, in diesem Falle **hierarchical** und **tree** identifiziert.

Das verwendete Werkzeug zur Featuremodellierung, *FeaturePlugin*, erlaubt bei einzelnen Features ebenfalls die Angabe von Kardinalitäten. Damit wäre es beispielsweise möglich, mehrere Produktmodelle gleichzeitig auszuwählen und für jedes Produktmodell eine eigene Form der Deltaspeicherung zu selektieren. Von dieser Möglichkeit wurde aber bisher im Rahmen des MOD2-SCM Projektes noch kein Gebrauch gemacht, d.h. im Kontext dieser Arbeit haben alle Features die maximale Kardinalität 1.

2.3.2 Konfigurierung

Die Konfigurierung beschreibt den Prozess, in dem eine konkrete Konfiguration, die konform zu einem Feature Diagramm ist, durch die Auswahl und das Klonen von Features erstellt wird. Abbildung 2.5 zeigt einen Ausschnitt einer Konfiguration der MOD2-SCM Produktlinie, die ein CVS-ähnliches System ergibt. In der Abbildung wurden nicht selektierte Zweige bzw. obligatorische Features zu Gunsten einer besseren Lesbarkeit ausgeblendet. Das verwendete Versionsmodell unterstützt Verzweigungen und Verschmelzungen. Die versionierten Objekte sind Dateisystemobjekte. Artefakte werden nach der gemischten Deltaspeicherung abgelegt, d.h. für den Hauptzweig werden Rückwärtsdeltas verwendet, für die Seitenzweige Vorwärtsdeltas. Die Auswahl der Objekte erfolgt nach dem *Product first* Ansatz. Die Konfiguration wurde auf Basis des in Abbildung 2.3 gezeigten Featuremodells erstellt. Das Werkzeug „FeaturePlugin“ unterstützt den Anwender bei der Konfigurierung, durch Auswahlboxen bei optionalen und gruppierten

Features. Wird ein solches Feature aus einer Gruppe, die ein *exklusives Oder* darstellt ausgewählt (dargestellt durch ein Quadrat mit einem Haken), so werden automatisch alle verbleibenden Features dieser Gruppe auf *nicht ausgewählt* (dargestellt durch ein graues Quadrat mit einem X) gesetzt. Ebenso werden alle in den Constraints formulierten Implikations- und Exklusionsbedingungen bei jeder Änderung ausgewertet und auf die Konfiguration angewandt.

2.3.3 Produktlinienprozess

Auch der im Kontext dieser Arbeit verwendete Modellierungsansatz zur modellgetriebenen Entwicklung von Softwareproduktlinien [BDW10] (siehe Abbildung 2.6) unterscheidet die Ebenen Domänenentwicklung und Anwendungsentwicklung [Lin02].

Im Bereich der Domänenentwicklung wird die Domäne unter zu Hilfenahme von FODA oder darauf basierenden Techniken analysiert. Ein Featuremodell als Resultat dieser Analyse [KCH$^+$90] beschreibt variable und feste Bestandteile einer Produktlinie, alternative Ausprägungen dieser variablen Teile und deren Beziehungen untereinander. Das zur Erstellung des Featuremodells verwendete Werkzeug wurde im vorigen Abschnitt vorgestellt. Basierend auf dem Featuremodell wird das ausführbare Domänenmodell entwickelt, das mit Hilfe von Fujaba [Zün02] (siehe Abschnitt 3.4) erstellt wird. Dieses Domänenmodell enthält die Realisierung aller im Featuremodell erfassten Features und enthält sowohl Struktur-, als auch Verhaltensmodelle. Im Gegensatz zur klassischen Aufteilung im Produktlinienprozess in Entwurf und manuelle Implementierung (siehe auch Abbildung 2.1) erfolgt die Vorgehensweise hier komplett modellgetrieben.

Die Anwendungsentwicklung hingegen wird bei dem in dieser Arbeit vorgestellten Ansatz zu einem reinen *Konfigurierungsprozess* reduziert. Im Gegensatz zum klassischen Vorgang in der Domänenentwicklung, bei dem produktspezifische Erweiterungen bzw. Anpassungen während diesem Vorgang als Ergänzung zu den wiederverwendeten Komponenten entwickelt werden, wird hier lediglich das Featuremodell konfiguriert. Dies geschiet durch Selektieren der spezifischen Eigenschaften des gewünschten Systems im Featuremodell. Anschließend wird die resultierende Konfiguration dazu verwendet, um das während der Domenenentwicklung erstellte Domänenmodell zu konfigurieren. Im letzten Schritt wird schließlich aus dem konfigurierten Domänenmodell ausführbarer Quelltext generiert, der in der Regel keinerlei manuellen Anpassungen mehr bedarf. Lediglich die Konfigurierung des Featuremodells muss hierbei interaktiv durchgeführt werden. Die weiteren Schritte erfolgen anschließend komplett automatisiert.

Ein Hauptunterschied zum herkömmlichen Produktlinienprozess, der in Abschnitt 2.1 beschrieben ist, ist zum einen die komplett modellgetriebene Entwicklung des Domänenmodells. Im Gegensatz zur Trennung zwischen dem Architekturentwurf (im Prozess in Abbildung 2.1 als Domänendesign bezeichnet) und der darauf folgenden *manuellen* Implementierung (Domänenimplementierung) wird das im Kontext der vorliegenden Arbeit das gesamte Domänenmodell einschließlich der Architektur durch formale Modelle beschrieben. Während im klassischen Produktlinienprozess bei der Ableitung einzelner

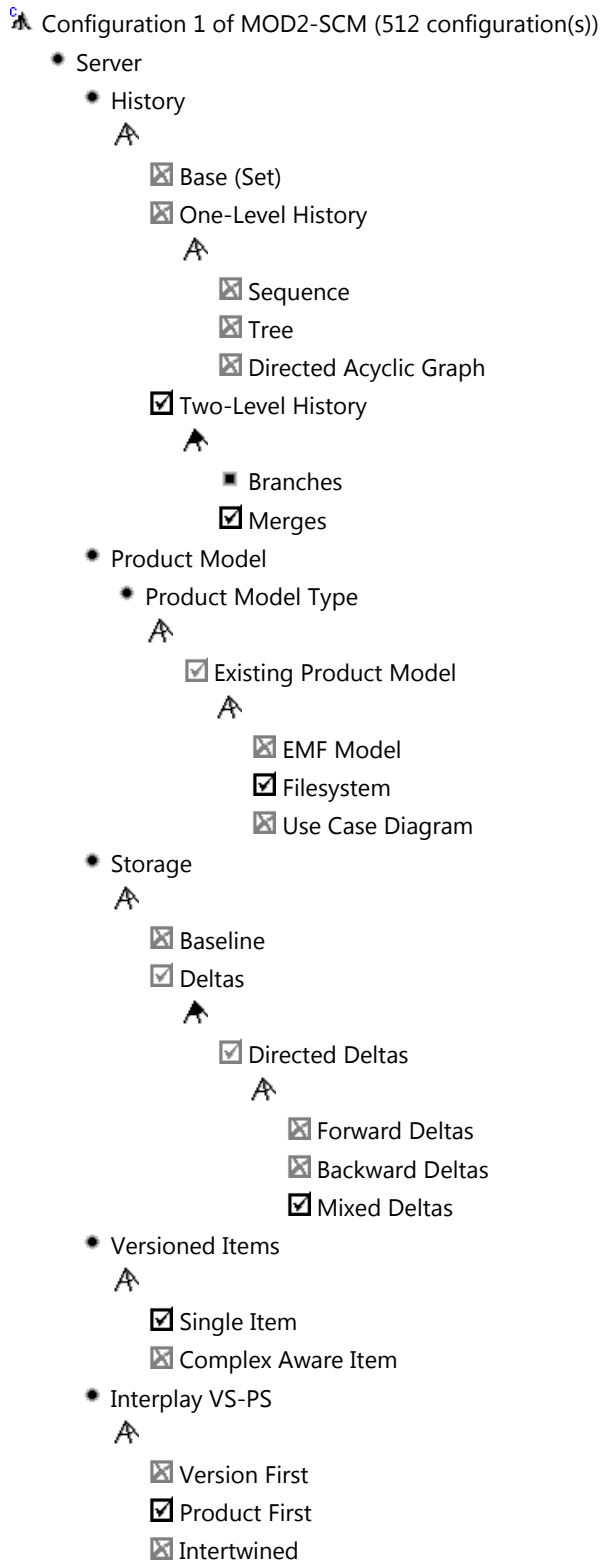

Abbildung 2.5: Ausschnitt aus einer möglichen Konfiguration, die ein CVS-ähnliches System ergibt.

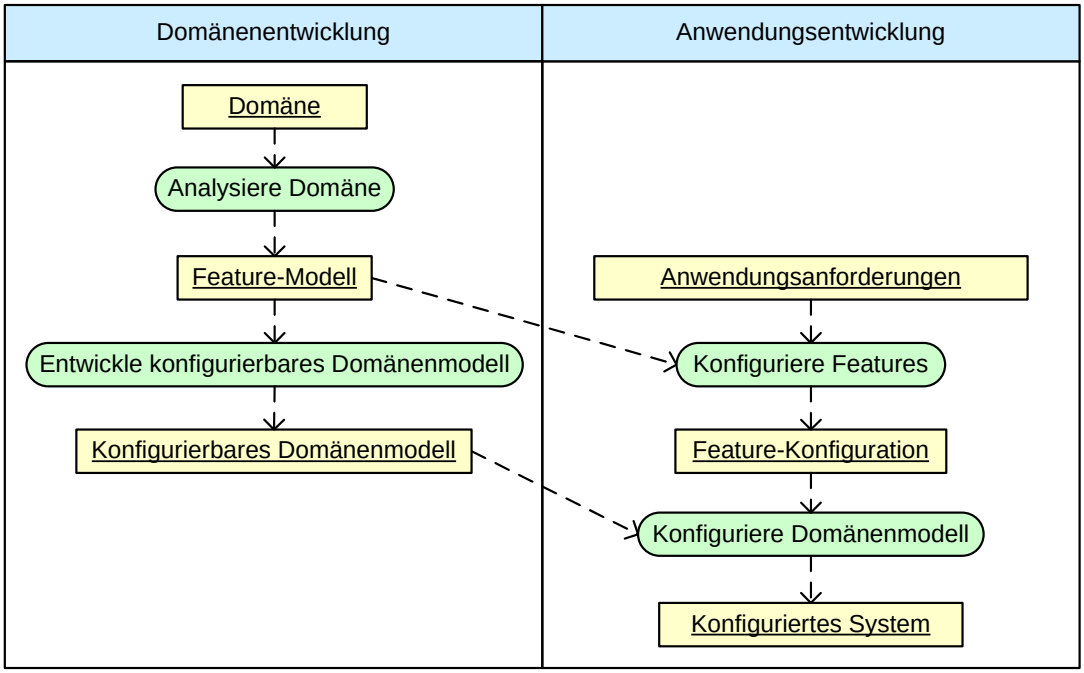

Abbildung 2.6: verwendeter Produktlinienprozess

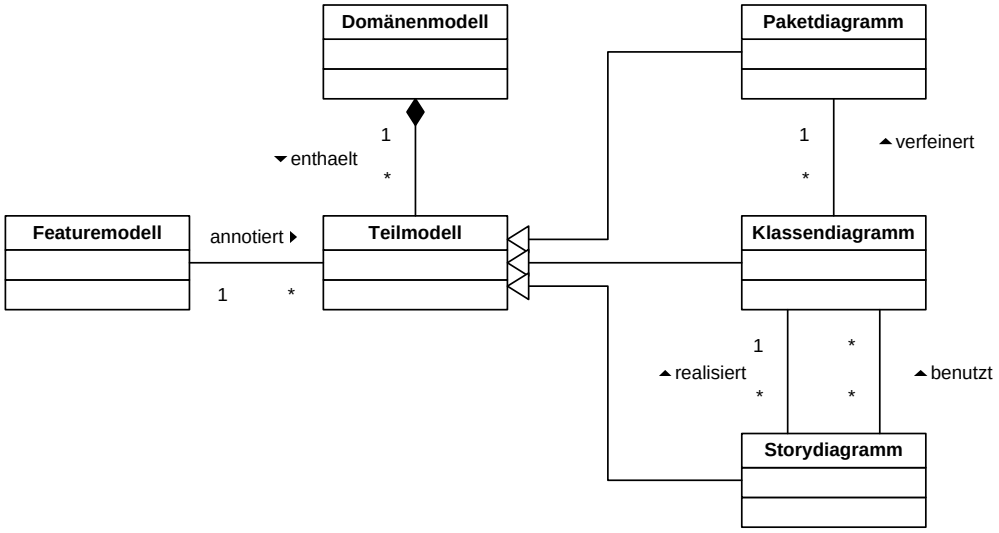

Abbildung 2.7: Modelle und deren Beziehungen

Produkte aus der Produktlinie wieder die Schritte Entwurf und Implementierung durchgeführt werden müssen, um Integration und etwaige produktspezifische Anpassungen zu realisieren, ist im hier vorgestellten Ansatz lediglich ein interaktiver Konfigurierungsschritt auf Basis des Featuremodells erforderlich. Mittels der hieraus resultierenden Kon-

figuration erfolgt die Ableitung des finalen Produktes dann vollständig automatisiert, was eine enorme Aufwandsreduzierung zur Folge hat. Auch Charles W. Krueger schlägt eine *pro-aktive* Herangehensweise zur Entwicklung einer Softwareproduktlinie vor, wenn die Anforderungen an die Produkte wohldefiniert und stabil sind. In diesem Falle nimmt die Domänenentwicklung ebenfalls einen Großteil des Entwicklungsaufwands ein und die Ableitung einzelner Produkte ist im Idealfall ein reiner Konfigurierungsprozess [Kru06]. Allerdings operiert das Werkzeug Gears von Krueger's Firma BigLever Software[3] rein auf dem Quelltext, bietet also keinerlei Unterstützung für einen modellgetriebenen Ansatz [Kru08].

Da an diesem Prozess unterschiedliche Modelltypen beteiligt sind, soll Abbildung 2.7 einen Überblick über diese Modelltypen und die zwischen ihnen bestehenden grobgranularen Beziehungen vermitteln. Die Eigenschaften der Softwareproduktlinie werden in dem *gloablen Featuremodell* erfasst. Sämtliche Teilmodelle des ausführbaren Domänenmodells nehmen darauf Bezug. Die Teilmodelle des Domänenmodells umfassen

Paketdiagramme Zum *Modellieren im Großen* werden Paketdiagramme eingesetzt (siehe Kapitel 5). Diese definieren auf grobgranularer Ebene die Struktur des zu erstellenden Systems, indem logische Bestandteile anhand einer Paketstruktur gegliedert und die Abhängigkeiten untereinander durch Importbeziehungen ausgedrückt werden.

Klassendiagramme Jedes Paket, das während der Disziplin Modellieren im Großen definiert wurde, wird anschließend durch ein Klassendiagramm verfeinert.

Storydiagramme Ein Storydiagramm [Zün02] realisiert das Verhalten von genau einer Methode. Es ähnelt einem UML2-Interaktionsübersichtsdiagramm, da es aus einem Aktivitätsdiagramm besteht, dessen Knoten durch Storymuster verfeinert werden. Diese Diagramme lassen sich in ausführbaren Java Quelltext übersetzen, so dass das konfigurierte Domänenmodell ausführbar ist.

Die Kopplung von Featuremodell und Domänenmodell ist in Kapitel 6 beschrieben, ebenso wie die Ableitung von konkreten Produkten aus dem Domänenmodell.

2.4 Zusammenfassung

In diesem Kapitel wurden die Grundlegenden Begriffe der Entwicklung von Softwareproduktlinien eingeführt. Der zweistufige Ansatz – Domänenentwicklung und Anwendungsentwicklung – wurde erläutert und außerdem wurden die unterschiedlichen Typen von Variabilität aufgezeigt. Im Folgenden wurde dargelegt, wie der dieser Arbeit zu Grunde liegende Ansatz der modellgetriebenen Entwicklung von Softwareproduktlinien in den

[3] http://www.biglever.com

Entwicklungskontext eingegliedert wurde, bzw. was die Besonderheiten des vorliegenden Ansatzes sind. Die wesentliche Eigenschaft ist die Reduzierung der Anwendungsentwicklung auf einen reinen Konfigurierungsprozess. Gegenüber der klassischen Entwicklung von Softwareproduktlinien stellt dies einen fundamentalen Unterschied dar. Der Ansatz basiert auf den weit verbreiteten Featuremodellen, die das Resultat der feature-orientierten Domänenanalyse (FODA) bzw. daraus entstandener Methoden zur Domänenanalyse sind. Auf Basis dieser Modelle, die sämtliche invarianten und variablen Bestandteile einer Produtklinie beschreiben, wird anschließend das Domänenmodell erstellt, das alle Varianten in sich vereint. Durch Erstellen einer Konfiguration auf Basis des Featuremodells und durch Anwendung dieser Konfiguration auf das Domänenmodell können die konkreten Produkte aus der Produktlinie abgeleitet werden.

3 Modellgetriebene Entwicklung

In diesem Kapitel wird der Begriff „modellgetriebene Entwicklung“ erläutert. Der von der Object Management Group (OMG) entwickelte Ansatz der modellgetriebenen Architektur (Model Driven Architecture) mit der standardisierten Modellierungssprache UML wird vorgestellt. Im Anschluss daran wird das im Rahmen dieser Arbeit verwendete Modellierungswerkzeug Fujaba eingeführt.

3.1 Definition

> „Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“[SVEH07], S. 11

Ein *formales Modell* beschreibt hierbei vollständig einen bestimmten Aspekt der Software. Dies müssen nicht zwingend UML-Modelle sein, sondern auch textuelle Modelle können eine formale Beschreibung eines Sachverhalts liefern. Häufig werden formale Modelle lediglich zur Dokumentation oder als grundlegende Spezifikation für eine manuelle Implementierung verwendet. In diesem Falle handelt es sich aber nicht um *modellgetriebene Softwareentwicklung*, sondern allenfalls um *modellbasierte Softwareentwicklung*. Ziel bei der modellgetriebenen Softwareentwicklung ist es immer, aus den formalen Modellen *lauffähige* Software zu erhalten. Dies kann prinzipiell auf zweierlei Arten geschehen:

Generatoren Bei der Verwendung von Generatoren, die ähnlich wie ein Compiler arbeiten, wird aus einem Modell Quelltext für die Zielplattform erzeugt. Dies kann z.B. Java oder C++ Quelltext sein.

Interpreter Werden hingegen Interpreter verwendet, so werden die Modelle zur Laufzeit ausgeführt.

In beiden Fällen ist das Resultat ein Modell, das vom Rechner ausgeführt werden kann. Dieser Prozess erfolgt außerdem *automatisch*. Hierbei wird der Quelltext in einem inkrementellen Prozess nach Änderungen am Modell immer wieder neu erzeugt. Eine einmalige Quelltexterzeugung und anschließende Änderungen direkt im Quelltext sind also nicht mehr notwendig, da jetzt die Modelle die Rolle des Quelltextes einnehmen. Die generierten Artefakte sind nur ein Zwischenschritt eines automatischen Build-Prozesses, der ausgehend vom Modell ein lauffähiges Programm erzeugt. Üblicherweise werden diese formalen Modelle in sogenannten CASE (Computer aided software engineering) Werkzeugen erstellt [KWB03].

3.2 Motivation

Software nimmt mittlerweile eine kritische Rolle in vielen Unternehmen ein. Eine beachtliche Anzahl von Geschäftsprozessen laufen heute softwaretechnisch automatisiert ab. Neben einem wichtigen Erfolgsfaktor hat sich die Software also auch mitunter zu einem entscheidenden Wettbewerbsvorteil vieler Unternehmen entwickelt. Dem steigenden Bedarf an softwaretechnischen Lösungen stehen gegenüber (siehe [GPR06], S. 9f):

Komplexität: immer größer werdende und damit schwieriger zu beherrschende Komplexität

Anforderungen: steigende Anforderungen an die Leistungsfähigkeit und Verfügbarkeit

Technologiezyklen: immer kürzere Technologiezyklen (sowohl bei Hardware- als auch bei Softwareplattformen)

Dynamik: dynamische Anforderungen, die häufigen Änderungen unterworfen sind

Kosten: hoher Druck zur Kostenreduzierung (gerade jetzt in Zeiten der Wirtschaftskrise)

Die Erkenntnisse aus der Softwarekrise und die daraus resultierende Forderung nach ingenieurmäßigen Techniken bei der Erstellung von Software führte zu einer stetigen Verbesserung der Entwicklungsprozesse, die bis heute noch nicht abgeschlossen ist. Trotz vielfältiger Innovationen fehlen bis dato immer noch geeignete Prozesse, die in puncto Flexibilität und Effizienz den klassischen Ingenieurprozessen nahe kommen. Seit nunmehr fast 60 Jahren wird bereits Software entwickelt. In den Anfängen dominierten die knappen und teuren Hardware-Ressourcen, weshalb die Softwareentwicklung nur eine untergeordnete Rolle einnahm. In der damaligen Zeit war die Größe der Software überschau- und beherrschbar, die Kodierung erfolgte binär in der jeweiligen Maschinensprache. Doch die Komplexität der Software-Systeme stieg rasch an. Aus der Erkenntnis, dass die für den Menschen kaum verständlichen binären Maschinencodes kaum dazu geeignet waren Software zu entwickeln, wurden die Assembler-Sprachen eingeführt. Auf einer gegenüber den binären Maschinencodes höheren Abstraktionsebene war es den Programmierern somit möglich, auf eine einfachere Art und Weise Software zu erstellen. Das in Assembler geschriebene Programm wurde anschließend in ein Maschinenprogramm übersetzt. Auch die Assembler-Sprachen hatten den Nachteil, dass man die zu lösenden Probleme nicht direkt formulieren konnte. John W. Backus entwickelte die erste höhere (problemorientierte) Programmiersprache FORTRAN [Bac98], die als Vorläufer weiterer höherer Sprachen gilt. Gegenüber der Assembler-Programmierung abstrahiert eine höhere Programmiersprache von Registern, Stacks und Statusflags, da der Compiler diese Arbeit bei der Übersetzung in ein ausführbares Programm übernimmt. Mitte der 60er Jahre des vorigen Jahrhunderts fielen die Preise für die Hardware, während gleichzeitig der Bedarf an immer komplexer werdender Software stetig anstieg. Die zunehmende Komplexität führte mangels geeigneter Prozesse dazu, dass immer mehr Softwareprojekte scheiterten. Als Reaktion auf diesen Missstand wurde auf der NATO-Konferenz 1968

die Softwarekrise ausgerufen und die Forderung nach einer geeigneten Ingenieursdisziplin für die Softwareentwicklung immer lauter. Der Begriff des Software-Engineering [Boe76], der strukturierten Vorgehensweise bei der Erstellung von Software wurde in dieser Zeit geprägt.

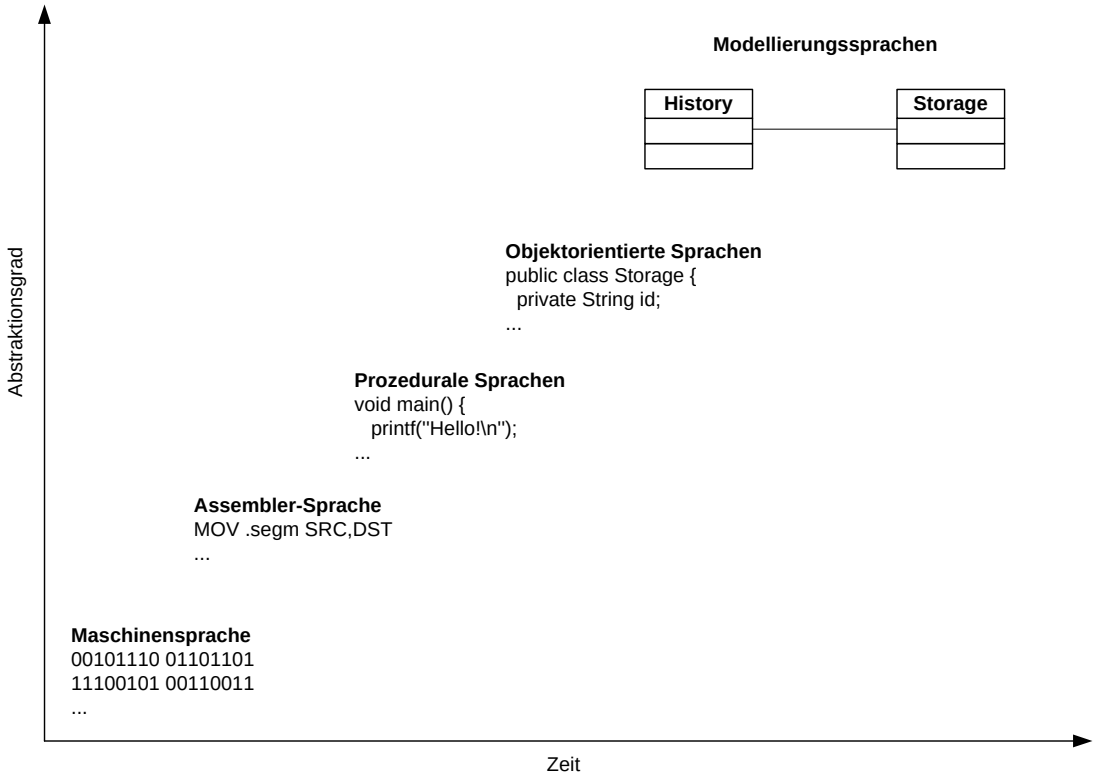

Abbildung 3.1: Zunehmender Abstraktionsgrad im Software-Engineering [GPR06], S. 15.

Diese neue Disziplin brachte nicht nur strukturierte Programmiersprachen mit Kontrollstrukturen und Fallunterscheidungen (z. B. Pascal oder C), sondern auch systematische Methoden, wie etwa die Strukturierte Analyse [DeM78] und das Strukturierte Design [YC79]. In der *Strukturierten Analyse* wurden erstmals mit Datenflussdiagrammen Prozesse auf abstrakte Weise modelliert. Die einheitliche Definition von Daten wurde dabei mit Hilfe von *Data Dictionaries* garantiert. Strukturdiagramme dienten im *Strukturierten Design* dazu, die ermittelten Funktionen in hierarchisch organisierten Modulen festzuhalten. Diese Diagramme waren ein Hilfsmittel zur Visualisierung von Abläufen. Außerdem wurden dadurch die Schnittstellen zwischen den einzelnen Modulen beschrieben. In der Folge wurde das Objektorientierte Paradigma als Lösung für das Ende der 80er Jahre aufgetretene Problem der geeigneten Wiederverwendung von Softwarebestandteilen entworfen. Objektorientierte Programmiersprachen, wie Smalltalk, Java oder C++ entstanden. Sie sehen ein System als Zusammensetzung vieler Objekte an, wobei jedes Objekt ein definiertes Verhalten, einen inneren Zustand und eine eindeutige Iden-

tität besitzt ([GPR06], S. 13). Bei der objektorientierten Programmierung werden Daten und Methoden nicht mehr getrennt, sondern in einem Objekt gekapselt. Durch Vererbung und Polymorphie soll die Möglichkeit zur Wiederverwendung verbessert werden. In Folge dessen wurden geeignete objektorientierte Methoden entwickelt, wie etwa die *Object Modeling Technique (OMT)* von James Rumbaugh [RBP+91] oder das *Object Oriented Design (OOD)* von Grady Booch [Boo94]. Bei diesen Vorgehensweisen wurde abermals intensiv der Gebrauch von Modellen und entsprechender Werkzeuge motiviert. Die *Unified Modeling Language (UML)* schließlich vereint die Vielzahl an objektorientierten Methoden in einer standardisierten Modellierungssprache. Aus dieser Entwicklung lässt sich das Mittel der Abstraktion als Trend im Software-Engineering ausmachen, wie auch Abbildung 3.1 verdeutlicht.

Ziel der modellgetriebenen Softwareentwicklung ist es, den gesamten Lebenszyklus einer Software mit Hilfe von ausdrucksstarken Modellen und deren kompakter Darstellung zu beschreiben. Die UML ist dabei der Versuch, eine standardisierte, domänenunabhängige Modellierungssprache anzubieten.

3.3 MDA als Spezialisierung

Model Driven Architecture (MDA) ist ein von der *Object Management Group (OMG)* definiertes Rahmenwerk für die Softwareentwicklung [KWB03]. Die Kernidee von MDA ist die Beschreibung von Softwarekomponenten unabhängig von deren technischen Umsetzungen. Hierzu werden verschiedene Abstraktionsebenen definiert für die Modelle erstellt werden. Die konzeptionelle Basis des Ansatzes bilden hierbei die *plattformunabhängigen (platform independent model - PIM)* sowie die *plattformspezifischen (platform specific model - PSM)* Modelle. Das *plattformunabhängige Modell* beschreibt formal die Struktur und Funktionalität eines Systems, unabhängig von den implementierungstechnischen Details [GPR06]. Der Übergang von den abstrakten zu den technischen Modellen erfolgt hierbei vorzugsweise automatisiert mit Hilfe von Transformationswerkzeugen, wobei die Transformation in einer separaten Transformationsbeschreibung definiert ist [GPR06]. Schlussendlich werden dann die plattformabhängigen (PSM) Modelle in den Quelltext der Zielplattform transformiert. Die Vorteile dieser Herangehensweise liegen auf der Hand (vgl. [KWB03], S. 9ff.):

Produktivität Der Fokus des MDA Ansatzes zielt darauf ab, plattformunabhängige Modelle zu entwickeln und die plattformabhängigen daraus durch gezielte Transformationen abzuleiten. Diese Transformation muss zwar definiert werden, aber dies ist ein einmaliger Vorgang, da diese Transformation zur Erstellung einer Vielzahl von Systemen verwendet werden kann. Durch die Arbeit mit den plattformunabhängigen Modellen und der daraus resultierenden Tatsache, dass der Entwickler sich nicht mit technischen Details aufhalten muss, wird die Produktivität auf zweierlei Arten gesteigert. Zum einen müssen die Entwickler des plattformunabhängigen Modells sich nicht um plattformspezifische Details kümmern und zum anderen führt die Abstraktion vom Quelltext zum PIM dazu, dass der Entwickler mehr Zeit zur

Lösung des eigentlichen Problems zur Verfügung hat und das resultierende System die Anforderungen des Kunden besser erfüllt.

Portabilität Durch die Konzentration auf die Erstellung plattformunabhängiger Modelle wird in der MDA die Portabilität auf unterschiedliche Zielplattformen schon per Definition erreicht. Ein und dasselbe PIM kann durch verschiedene Transformationen in eine Vielzahl von PSMs für unterschiedliche Plattformen überführt werden. Alles, was auf Ebene der PIM spezifiziert ist, ist daher komplett portabel. Der Grad an Portabilität hängt allerdings stark von den Werkzeugen zur automatischen Transformation ab.

Interoperabilität Verschiedene PSMs, die aus einem PIM erzeugt werden, können untereinander in Beziehung stehen. Diese Beziehungen werden in der MDA als *Brücken (bridges)* bezeichnet. Um eine Kommunikation von PSMs für unterschiedliche Plattformen zu gewährleisten, muss ebenso eine Transformation der verwendeten Konzepte zwischen den jeweiligen Plattformen erfolgen. MDA stellt daher sicher, dass bei der Transformation zu den jeweiligen PSMs auch die entsprechenden Brücken mit erzeugt werden.

Wartung und Dokumentation Im MDA Lebenszyklus, liegt der Fokus auf der Erstellung des plattformunabhängigen Modells auf einer höheren Abstraktionsebene als der Quelltext. Durch Transformationen wird dieses Modell letztendlich in Quelltext überführt, daher stellt das Modell eine exakte Repräsentation des Codes dar. Mit anderen Worten nimmt das Modell somit auch die Funktion einer Dokumentation auf höherer Ebene ein, die ebenfalls für ein Software System benötigt wird. Der Hauptunterschied liegt darin, dass Systemänderungen durch Änderungen am PIM durchgeführt werden, und die Dokumentation somit immer dem aktuellen Stand entspricht.

3.3.1 MOF

Die OMG definiert die *Meta Object Facility (MOF)* [OMG06a, OMG07], eine Sprache zur Definition von Metamodellen. MOF stellt somit ein *Meta-Metamodell* dar. Auch die in Abschnitt 3.3.2 eingeführte Modellierungssprache UML wurde mit Hilfe von MOF beschrieben. MOF ermöglicht die Interoperabilität zwischen den plattformunabhängigen und den plattformspezifischen Modellen durch die Bereitstellung von Abbildungsregeln auf Technologien wie XML oder Java [HK05]. Der gemeinsame Kern von MOF und UML wird in der UML Infrastruktur (im Paket *Core*) [OMG09a] beschrieben.

Die OMG verwendet vier Meta-Ebenen, die von den Instanzen (M0) bis zum Meta-Metamodell (M3) durchnummeriert sind (siehe Abbildung 3.2). MOF ist dabei auf der Ebene M3 definiert, die UML auf M2. Ein von einem Benutzer in einem UML Werkzeug erstelltes Modell befindet sich auf der Ebene M1, und konkrete Laufzeitinstanzen dieses Modells bilden die Ebene M0. D.h. die UML, als Metamodell betrachtet, stellt eine Instanz von MOF dar, was wiederum bedeutet, dass Bestandteile der UML Instanzen von MOF Elementen sind.

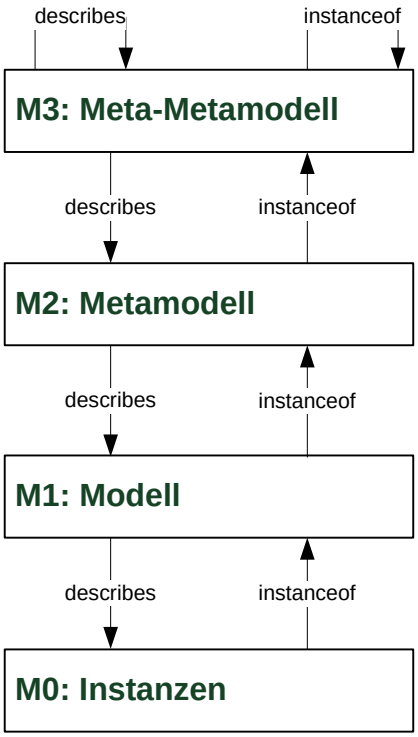

Abbildung 3.2: Meta-Ebenen der OMG [SVEH07].

3.3.2 UML

Die *Unified Modeling Language (UML)* [OMG09a, OMG09b] ist innerhalb der MDA die Sprache zur Erstellung von Modellen. Die Modellierungssprache UML erlaubt die Spezifikation von Systemen unabhängig von Fach- oder Spezialgebieten. Hierzu werden eine Fülle von Notationselementen und Diagrammarten bereitgestellt, sowie Erweiterungsmechanismen, um auf Basis dieser Standardkonstrukte eigene Diagrammarten bzw. Modifikationen von bestehenden Diagrammen zu erstellen. Da es sich bei der UML weder um ein Prozessmodell noch um eine Methode handelt, existieren keine Anleitungen dahingehend, welche Diagrammart wann benutzt werden sollte. UML geht weitestgehend aus den unterschiedlichen Ansätzen von Ivar Jacobson – *Object-Oriented Software Engineering (OOSE)* [JCJv92], James Rumbaugh – *Object Modeling Technique (OMT)* [RBP+91] – sowie von Grady Booch – *Object-Oriented Design (OOD)* [Boo94] – hervor.

Da sowohl MDA als auch die UML von der OMG entwickelt und propagiert werden, liegt es nahe, dass die UML die Modellierungssprache darstellt, um die plattformunabhängigen Modelle des Model-driven Architecture Ansatzes zu erstellen. Das Metamodell von UML wird formal mit Hilfe von MOF (siehe Abschnitt 3.3.1) beschrieben. Ein Vorteil hiervon ist die Trennung der darstellungsunabhängigen Struktur und der Bedeutung (abstrakte Syntax) von der Notation eines Modells (konkrete Syntax).

Die *Erweiterbarkeit* der UML wird durch den Mechanismus der sogenannten *UML-Profile* erreicht. Sie bieten die Möglichkeit, basierend auf Sprachkonstrukten der UML eigene Erweiterungen und Spezialisierungen der Sprache in Form einer domänenspezifischen Sprache zu definieren. Mit Hilfe von MOF können aber sogar mit den Profilen komplett eigene Metamodelle erstellt werden, die dann Grundlage für neue Sprachen darstellen.

Die UML Spezifikation besteht im Wesentlichen aus fünf Kernkomponenten:

UML Infrastructure Die UML Infrastructure ist die Basis der Architektur der UML. Sie definiert den Sprachkern für eine Metasprache.

Meta Object Facility MOF dient zur Erstellung von Metamodellen (wie etwa dem UML Metamodell). MOF importiert die in der UML Infrastructure definierte abstrakte Syntax und erweitert diese.

UML Superstructure Die UML Superstructure definiert die Diagrammarten der UML und bildet somit das Metamodell der UML-Diagramme.

XML Metadata Interchange XMI dient zum Austausch der Modelle zwischen Modellierungswerkzeugen bzw. als Eingabe für Generatoren.

Object Constraint Language OCL ist eine formale, textuelle Notation die, basierend auf mathematischen Konzepten, eine Präzisierung von Modellen erlaubt.

UML unterscheidet dreizehn verschiedene Diagrammarten, die sich in die Kategorien *Strukturdiagramme* und *Verhaltensdiagramme* klassifizieren lassen. Abbildung 3.3 gibt einen Überblick über die verschiedenen UML-Diagrammarten. Während die Strukturdiagramme die Modellierung des statischen Aufbaus eines Systems aus unterschiedlichen Gesichtspunkten erlauben, dienen die Verhaltensdiagramme dazu, die dynamischen Aspekte eines Systems darzustellen. Die Strukturdiagramme umfassen Paketdiagramme, Klassendiagramme, Objektdiagramme, Verteilungsdiagramme, Komponentendiagramme sowie Kompositionsstrukturdiagramme. Zu den Verhaltensdiagrammen zählen hingegen Anwendungsfalldiagramme, Interaktionsdiagramme, Aktivitätsdiagramme, Zustandsdiagramme, Sequenzdiagramme, Kommunikationsdiagramme, Zeitdiagramme und Interaktionsübersichtsdiagramme.

In den folgenden Unterabschnitten werden Diagrammarten aus der UML vorgestellt, die im Rahmen dieser Arbeit von Bedeutung sind.

Paketdiagramm

Paketdiagramme dienen nicht nur der Spezifikation der eigentlichen Struktur des modellierten Systems, sondern sie erlauben auch die Strukturierung des Modells durch Gruppierung und hierarchische Anordnung von Diagrammen und Modellelementen.

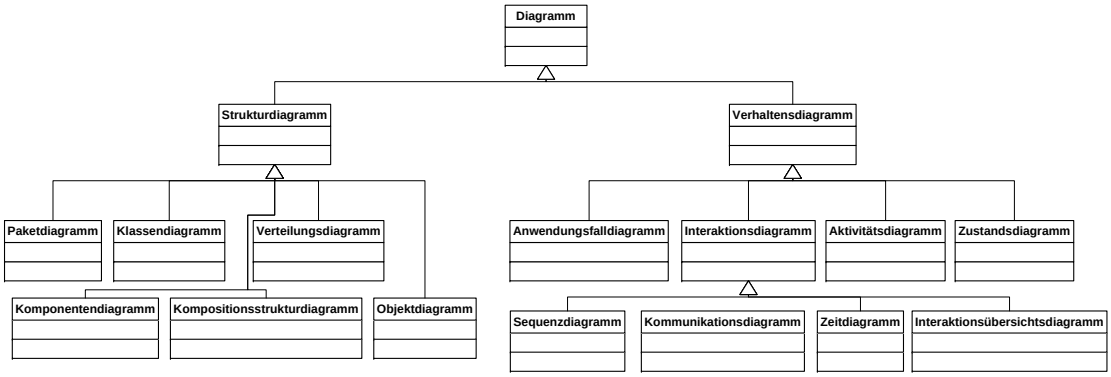

Abbildung 3.3: Die dreizehn Diagrammarten der UML 2.x [HK05].

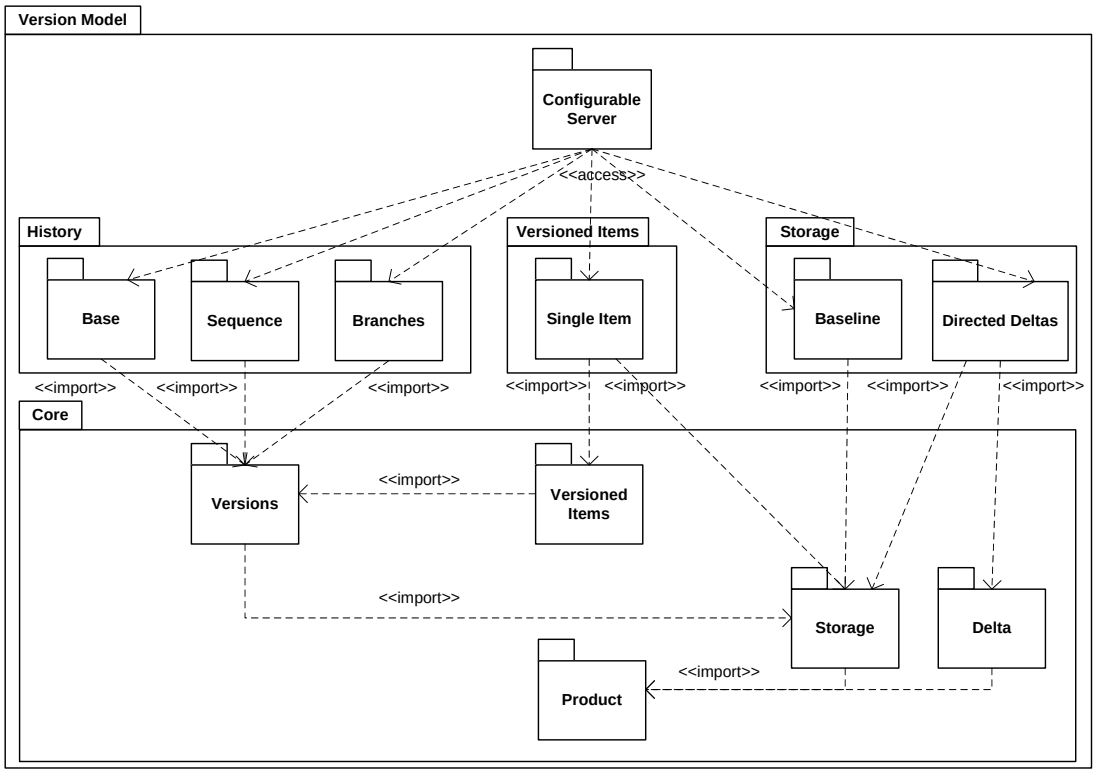

Abbildung 3.4: Ausschnitt aus dem Paketdiagramm des MOD2-SCM Systems.

Ein Paket erlaubt die Gruppierung beliebiger paketierbarer UML-Elemente, wie z.b. Klassen, Datentypen, Aufzählungstypen, Komponenten oder auch Pakete. Abbildung 3.4 zeigt die unterschiedlichen Darstellungsarten von Paketen in der UML als großes Rechteck mit aufgesetztem kleinen Rechteck. Der Name des Pakets kann in einen der beiden Rechtecke angegeben sein, während die gruppierten Elemente nur im großen Rechteck

Platz finden. Ein Paket definiert einen Namensraum und ein Element kann nur höchstens einem Paket angehören. Innerhalb eines Pakets müssen die Namen der darin enthaltenen Elemente eindeutig sein. Systemweit eindeutige Namen erhält man durch den *qualifizierten Namen*, der durch Voranstellen aller direkt und indirekt übergeordneter Paketnamen aus dem Elementnamen gebildet wird. Innerhalb eines Pakets können sich Elemente über ihren Namen referenzieren, ebenso wie Elemente aus (indirekt) übergeordneten Paketen, sofern nicht ein Element gleichen Namens im entsprechenden Paket sie verdeckt. Elemente aus anderen Paketen können mit Hilfe des qualifizierten Namens referenziert werden, oder aber Pakete und deren Elemente können durch Importbeziehungen in den Namensraum des importierenden Elements hinzugefügt, und somit direkt referenzierbar gemacht werden. Elemente eines Pakets besitzen entweder *private* oder *öffentliche Sichtbarkeit*. Öffentliche Elemente sind global sichtbar, private hingegen nur innerhalb des Pakets und von ihm eingeschlossenen Paketen. Es ist zu beachten, dass jedem Element eine Sichtbarkeit zwingend zugeordnet werden muss. Die UML sieht hierfür keinen Standardwert sowie keinen undefinierten Wert vor. Bei den Importbeziehungen wird zwischen Paket- und Elementimporten unterschieden, die jeweils ebenfalls private oder öffentliche Sichtbarkeit besitzen können. Durch Importe können Namenskonflikte entstehen, die aufgelöst werden müssen. Wird z. B. der Name eines importierten Elements durch ein Element gleichen Namens aus einem direkt oder indirekt übergeordneten Paket benutzt, so verdeckt der importierte Name den des Elements aus dem übergeordneten Paket. Soll dieses verdeckte Element dennoch benutzt werden, so ist es über den qualifizierten Namen anzusprechen. Wird hingegen der Name eines importierten Elements im importieren Paket bereits verwendet, so verdeckt der Name des existierenden Elements das importierte Element, d.h. die entsprechenden Elemente werden nicht dem importierenden Namensraum hinzugefügt und müssen folglich über den qualifizierten Namen referenziert werden. Die Notation einer Importbeziehung erfolgt als gerichtete Beziehung ausgehend vom importierenden Namensraum und wird als gestrichelte Linie mit Pfeilspitze in Richtung des importierten Elements dargestellt. Die Unterscheidung zwischen öffentlichen und privaten Importen erfolgt durch die Schlüsselwörter ≪*import*≫ und ≪*access*≫. Eine Besonderheit bei öffentlichen Importen ist deren Transitivität. Dies bedeutet, dass ein öffentlicher Import eines Elements wiederum auch dessen öffentlich importierte Elemente dem importierenden Namensraum hinzufügt. Abbildung 3.4 zeigt diese unterschiedlichen Arten der Importbeziehung. Das Paket **ConfigurableServer** etwa verwendet private Paketimporte, um beispielsweise auf Elemente aus dem Paket **History::Base** zuzugreifen, während dieses Paket seinerseits einen öffentlichen Import auf das Paket **Core::Versions** definiert.

Neben den Importbeziehungen bietet die UML auch noch die Möglichkeit der Verschmelzung von Paketen. Sollen Pakete erweitert oder angepasst werden, so kann es notwendig sein die in dem Paket enthaltenen Elemente zu modifizieren. Im Gegensatz zu den Importen kann eine Verschmelzungsbeziehung nur auf Paketebene angewendet werden. Sie wird ebenfalls mit einer gestrichelten Linie mit Pfeilspitze und dem Schlüsselwort ≪*merge*≫ dargestellt. Dabei wird der Inhalt des Basispakets (am Ende mit der Pfeilspitze) mit dem Inhalt des Verschmelzungspakets verschmolzen. Die komplexen Regeln, die dabei zu beachten sind, sind in [HK05] bzw. der UML Spezifikation [OMG09b] nachzu-

lesen. Da Verschmelzungsbeziehungen für die vorliegende Arbeit nicht von Bedeutung sind, werden sie hier nicht weiter diskutiert.

Klassendiagramm

Klassendiagramme dienen zur Spezifikation von Datenstrukturen des Systems. Die zu Grunde liegenden Konzepte sind der konzeptuellen Datenmodellierung und der objektorientierten Softwareentwicklung entnommen. In Klassendiagrammen werden Klassen mitsamt ihren Attributen und Operationen modelliert, sowie die Beziehungen der Klassen untereinander durch Assoziationen. Abbildung 3.5 zeigt einen Ausschnitt eines Klassendiagramms aus dem MOD2-SCM System, das zur Modellierung von Versionsgraphen mit Verzweigungen dient.

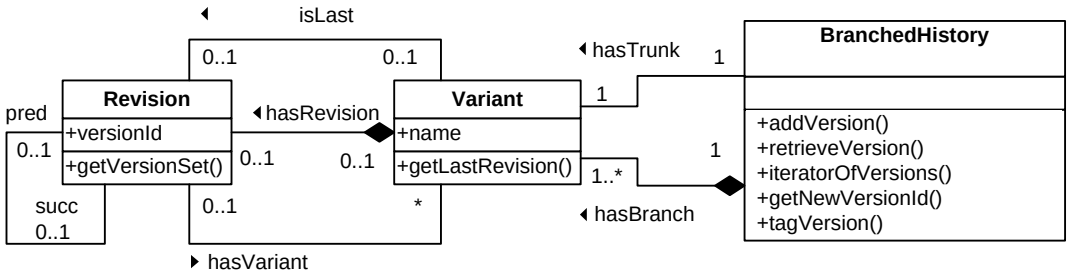

Abbildung 3.5: Ausschnitt eines Klassendiagramms.

Das Diagramm zeigt drei Klassen **Revision**, **Variant** und **BranchedHistory**, die als Rechtecke dargestellt sind. In dem Bereich unterhalb des Klassennamens werden sämtliche Attribute der Klasse angezeigt. Im unteren Bereich werden die von der Klasse bereitgestellten Operationen aufgelistet. Neben dem Klassennamen kann der erste Abschnitt noch Stereotypen, Schlüsselworter und Eigenschaftsangaben beinhalten (in der Abbildung nicht dargestellt). Beispiele für Stereotypen wären etwa der Stereotyp ≪**entity**≫, ein Schlüsselwort stellt beispielsweise ≪**interface**≫ dar. Eigenschaftsangaben werden durch Auflistung in geschwungenen Klammern dargestellt. Sie werden unter anderem dafür benutzt, abstrakte Klassen {**abstract**} zu kennzeichnen. Der zweite Abschnitt enthält die Attribute und deren Sichtbarkeiten. Analog zu den objektorientierten Programmiersprachen wird hier ebenso zwischen Instanz- und Klassenattributen unterschieden. Klassenattribute werden unterstrichen dargestellt. Ist der Wert eines Attributs aus anderen Informationen berechenbar, so spricht man von abgeleiteten Attributen. Dies wird durch einen vorangestellten Schrägstrich symbolisiert. Kann ein Attribut mehrere Werte annehmen, so wird dies durch ein Intervall angezeigt. Die Angabe des Typs ist optional und kann sprachspezifisch erfolgen, ebenso wie der Standardwert eines Attributs. Zusätzlich ist noch die Angabe weiterer Eigenschaften von Attributen, wie Eindeutigkeit oder Unveränderbarkeit nach der Initialisierung usw. möglich. Im Abschnitt, der die Operationen enthält, wird ebenfalls zunächst die Sichtbarkeit einer Operation dargestellt, gefolgt vom Namen und der Parameterliste. Anschließend folgt die Angabe des Rückgabewerts.

Ähnlich wie bei den Attributen symbolisiert eine Unterstreichung eine Klassenoperation.

Neben Vererbungsbeziehungen, die vertikale Abhängigkeiten im Modell ausdrücken, gibt es horizontale Beziehungen zwischen Klassen, die durch Assoziationen ausgedrückt werden. In der Darstellung einer Assoziation in einem UML Klassendiagramm, die in ihrer einfachsten Form lediglich durch eine Kante symbolisiert wird, können sie benannt werden und es kann die Leserichtung vorgegeben werden. Assoziationen können ebenso wie Attribute durch Berechnung hervorgehen und werden dementsprechend ebenfalls mit einem dem Namen vorangestellten Schrägstrich gekennzeichnet. Die an der Beziehung beteiligten Objekte nehmen im Modell spezifische Funktionen ein. Diese Funktionen werden durch die Rollennamen an den Assoziationsenden symbolisiert. Hier können außerdem noch Sichtbarkeits- und Multiplizitätsangaben erfolgen. In der Abbildung wird so etwa eine Beziehung zwischen **Variant** und **Revision** mit dem Namen **isLast** eingeführt. Sie dient dazu, die jeweils letzte Revision eines Zweiges zu ermitteln. Dementsprechend ist die Multiplizität an beiden Rollenenden **[0..1]**.

Assoziationen, die einem Objekt eine Menge von Partnerobjekten (Multiplizität > 1) zuordnen, sind per Definition ungeordnet. Durch die vorgegebene Eigenschaft {*ordered*} an dem Assoziationsende wird angegeben, dass die zugehörige Menge einer vorgegebenen Ordnungsrelation entspricht, die durch Operationen, die die Assoziation verwalten, aufrecht erhalten werden muss. Darüber hinaus kann zusätzlich noch die Eindeutigkeit von in der Menge enthaltenen Objekten durch Angabe des Schlüsselworts {*unique*} erfordert werden. Desweiteren können Assoziationskanten auch gerichtet sein, um die explizite Navigationsrichtung von einem Objekt zu dessen Partnerobjekt vorzuschreiben. Diese Richtung wird durch eine Pfeilspitze am entsprechenden Assoziationsende dargestellt. In diesem Fall ist das Assoziationsende *navigierbar*. Ein *nicht navigierbares* Ende wird durch ein „X“ am entsprechenden Assoziationsende angezeigt. Während der Standard bei keiner Angabe einer Navigationsrichtung von undefiniert navigierbaren Assoziationen ausgeht, wird dieser Fall in der Praxis und der Werkzeugunterstützung pragmatischer betrachtet, indem keine Angabe von Navigationsrichtungen bidirektionale Navigierbarkeit impliziert und nur die explizite Angabe einer Navigationsrichtung eine unidirektionale Assoziation ausdrückt. Eine navigierbare Assoziation bedeutet semantisch ein Attribut der Klasse am gegenüberliegenden Assoziationsende.

UML erlaubt auch die Angabe von Teil- bzw. Vereinigungsmengen von Assoziationen durch die Schlüsselwörter {*subsets*} bzw. {*union*}. Assoziationen können desweiteren qualifiziert sein, d.h. bei Assoziationsenden mit Multiplizität größer eins wird die Menge der Verbindungen durch ein oder mehrere qualifizierte Attribute in disjunkte Teilmengen zerlegt. Ist dieses Attribut ein Schlüssel, so enthalten diese Teilmengen genau ein Element. Mit Assoziationen können auch „Teile-Ganzes-Beziehungen“ ausgedrückt werden. Dies geschieht mittels Aggregationen bzw. Kompositionen, wobei erstere durch eine Raute am Rollenende, das zur Klasse führt, die das „Ganze“ markiert, dargestellt wird. Die Komposition stellt eine strengere Form der Aggregation dar und wird durch eine ausgefüllte Raute symbolisiert. Bei einer Komposition dürfen die entsprechenden „Tei-

le“ nur Teil von höchstens einem Ganzen sein. Ist die Raute mit Multiplizität 1 versehen, so bedeutet dies, dass die Teile höchstens so lange existieren, wie auch das Ganze existiert, außer sie werden vor Ende der Lebensdauer des Ganzen an ein anderes Ganze weitergereicht. Abbildung 3.5 zeigt eine solche Kompositionsbeziehung mit Multiplizität 1 an der Raute zwischen **BranchedHistory** und *Variant*. Ein Löschen des Versionsbaums hat somit das Löschen aller darin enthaltenen Zweige zur Folge.

3.3.3 OCL

Mit Hilfe der *Object Constraint Language* [OMG06b, WK03] bietet die OMG eine Möglichkeit zur deklarativen Präzisierung von Modellen, indem für die Verwendung von Modellelementen Bedingungen formuliert werden können, die zur Laufzeit des Modells jederzeit erfüllt sein müssen. Somit können dynamische Aspekte in UML-Modellen präzisiert werden. Die Object Constraint Language wurde 1997 von Rational entwickelt und ist eine textuelle Sprache, die auf den mathematischen Prinzipien von Mengenlehre und Prädikatenlogik basiert. OCL erlaubt das Formulieren von Ausdrücken, die bespielsweise Ableitungsregeln für Attribute beschreiben, den Rumpf von Abfrageoperationen bilden oder Invarianten bzw. Vor- und Nachbedingungen darstellen. Durch OCL wird die Ausdrucksmächtigkeit von UML und MOF erhöht. Die Object Constraint Language ist eine funktionale Spezifikationssprache, das bedeutet, dass ein OCL Ausdruck immer beschreibt, was der Wert ist, aber nie vorschreibt, wie die Berechnung dafür stattfindet. Natürlich können diese Ausdrücke in Programmiersprachen, wie etwa Java, übersetzt werden, um die Berechnung zu implementieren.

Im MDA Kontext wird OCL dafür verwendet, Quellmodelle und Sprachdefinitionen präziser zu formulieren bzw. Transformationen zu definieren. Viele Transformationen können nur unter bestimmen Voraussetzungen angewendet werden. Die Bedingungen dafür können ebenfalls mit Hilfe von OCL angegeben werden. Sie erfolgen jeweils für die Metamodelle der Quell- und Zielsprachen.

3.3.4 Ausführbare Modelle

Die Idee ausführbarer Modelle besagt, dass mit Hilfe von ausdrucksstarken und präzisen Sprachen, die damit erstellten Modelle von einem Modell-Compiler in ausführbare Programme übersetzt werden können [GPR06]. Dieser Ansatz wurde im Bereich von eingebetteten Systemen bereits erfolgreich umgesetzt. Dies wurde durch die Entwicklung eigener Modellierungssprachen erreicht, die mit Teilen der UML kombiniert wurden. Zwei prominente Vertreter dieser Modellierungssprachen sind die *Real-time Object-Oriented Modeling Language (ROOM Modeling Language)* [SGW94], sowie die *Specification and Description Language (SDL)* [BH89]. Mellor und Balcer [MB02] entwickelten mit *Executable UML (xUML)* einen Ansatz, der verspricht, nicht nur im Bereich eingebetteter Systeme zu funktionieren. Sie beschreiben einen Ansatz, der eine präzise Verwendung von UML beschreibt, um damit die erstellten Modelle ausführbar zu machen. Fujaba stellt einen weiteren Ansatz zur Erstellung ausführbarer Modelle dar. Dieser wird in Abschnitt 3.4 eingehend erläutert.

3.3.5 EMF

Das Eclipse Modeling Framework[1] [SBPM09] bietet ebenfalls Technologien zur Definition von Metamodellen. Aufgrund der starken Verbreitung setzen zunehmend Hersteller von Modellierungswerkzeugen auf diese Technologie. Das Metamodell von EMF ist *Ecore*, und ähnelt EMOF – essential MOF[2] – sehr stark. Ecore Modelle erlauben somit EMF-Modelle auf einfache Art und Weise zu definieren und diese als Grundlage für modellgetriebenes Arbeiten zu verwenden. Ein großer Unterschied zur UML ist hierbei, dass der Benutzer das gesamte Metamodell selbst definieren kann, was den Vorteil hat, dass es genau den zur Beschreibung der Domäne benötigten Abstraktionsgrad besitzt[SVEH07]. EMF bietet außerdem einen Generator, um aus einem Ecore Metamodell Java-Klassen, die der (statischen) Modellbeschreibung entsprechen, zu erzeugen. Im EMF Umfeld gibt es zahlreiche Werkzeuge, die die Erstellung von textuellen und auch grafischen Editoren für Ecore-basierte Metamodelle erleichtern, wie etwa GMF (Graphical Modeling Framework[3])[Gro09].

Steinberg et al. bezeichnen in ihrem Buch EMF als „MDA mit Stützrädern“ (siehe [SBPM09], S. 15), da EMF zwar den MDA Ansatz verfolgt, zu jederzeit aber ebenso die manuelle Erstellung von Quelltext parallel zu existierenden Modellen erlaubt. Somit ist es sehr einfach möglich nur Teile eines Systems zu generieren, während der Rest durch „klassische“ Programmierung ergänzt wird. Während UML dem Paradigma von Jean Bézivin von „Alles ist ein Modell (Everything is a Model)“ [Béz05] entspricht, und das andere Extrem die komplett modellfreie Entwicklung („Nothing is a model“ – Jean-Marie Favre) darstellt, befindet sich der Ansatz von EMF in der Mitte zwischen diesen beiden Extremen, mit dem aus Sicht der Väter von EMF genau richtigen Abstraktionsgrad. Dies entspricht dem Modellierungsparadigma, das auch Hans Vangheluwe verfolgt [Van06]:

> „Model everything, on the right level of abstraction, using the most appropriate formalism.“

Das Metamodell von EMF, Ecore, ist selbst ein EMF-Modell und somit sein eigenes Metamodell – ein Meta-Metamodell. Im Vergleich zu UML beschränkt sich EMF rein darauf, Klassen zu modellieren.

Da die OMG ihre Standards nur in Form von Spezifikationen definiert, die konkrete Umsetzung aber Aufgabe der jeweiligen Werkzeughersteller ist, hat dies zur Folge, dass in der Vergangenheit viele UML Werkzeuge entwickelt wurden, die eine schlechte Interoperabilität untereinander besitzen. Das Eclipse UML2 Projekt soll dieses Problem lösen, indem das UML2 Metamodell mit Hilfe des Ecore Metamodells definiert und allen Werkzeugherstellern zur Verfügung gestellt wird. Der Vorteil hierbei ist, dass alle Transformations- und Generatorsprachen, die mit EMF-Modellen arbeiten auch mit

[1] http://www.eclipse.org/modeling/emf/
[2] EMOF stellt eine auf die relevanten Konzepte reduzierte Version von MOF dar.
[3] http://www.eclipse.org/modeling/gmf/

Eclipse UML2-Modellen arbeiten können, was den Aufwand zur Erstellung neuer Werkzeuge beträchtlich reduziert. Im EMF Umfeld gibt es außerdem noch nützliche Erweiterungen zur Validierung (EMF Validation) oder zum Vergleich (EMF Compare) von Modellen, auf die hier aber nicht näher eingegangen wird.

3.4 Fujaba

Fujaba[4] [Zün02, The05] ist eine Modellierungsumgebung, basierend auf Graphtransformationen. Die Entwicklung von Fujaba begann ca. 1997 und integrierte das Konzept der Graphtransformationen in die objektorientierte Softwareentwicklung. Der wesentliche Mehrwert von Fujaba verglichen mit anderen CASE Werkzeugen liegt in den sogenannten Storydiagrammen und der Fähigkeit aus dieser Spezifikation ausführbaren Quelltext zu generieren. Fujaba wird an mehreren Standorten im In- und Ausland weiterentwickelt und deckt im wesentlichen die Bereiche (1) (Re-)Engineering, (2) eingebettete Echtzeitsysteme, (3) Lehre, (4) Spezifikation von verteilten Kontrollsystemen, (5) Integration mit der Eclipse Plattform und (6) MOF-basierte Integration von (Re-)Engineering Werkzeugen ab. Im Rahmen der vorliegenden Arbeit wurde Fujaba zur modellgetriebenen Softwareentwicklung eingesetzt. Fujaba bietet eine formale, visuelle Spezifikationssprache, die auf UML Klassendiagrammen, UML Aktivitätsdiagrammen und einer Graphtransformationssprache – den sogenannten Story Mustern – basiert. Weitere Diagrammarten werden über Plug-Ins unterstützt. Diese weiteren Diagrammarten sind aber nicht von Bedeutung für die vorliegende Arbeit und werden daher nicht näher betrachtet.

3.4.1 Statisches Modell: UML-Klassendiagramm

Zur Spezifikation der statischen Struktur eines Softwaresystems werden UML 1.x Klassendiagramme verwendet. Während abgeleitete Attribute nicht unterstützt werden, bietet Fujaba die Möglichkeit, für Assoziationen das Konstrukt ≪**Virtual Path**≫ zu verwenden. Hiermit kann der Benutzer Berechnungsregeln für den Zugriff auf Assoziationen selbst spezifizieren. Diese werden an Stelle der Standardzugriffsmethoden von Fujaba verwendet, um mit den Assoziationen zu arbeiten.

3.4.2 Dynamisches Modell: Storydiagramm

Mit Hilfe von Storydiagrammen werden die dynamischen Aspekte eines Systems beschrieben [FNTZ99]. Sie basieren auf UML Aktivitätsdiagrammen und den Story Mustern. Ein Storydiagramm beschreibt dabei das Verhalten von genau einer Operation einer Klasse aus dem Klassendiagramm. Durch Storydiagramme ist es möglich, in einem Graphmodell direkte Unterstützung für geordnete, sortierte und qualifizierte Assoziationen, die aus dem objekt-orientierten Datenmodell stammen, anzubieten. Graphschemata werden mit Hilfe von UML Klassendiagrammen spezifiziert. Die graphische Repräsentation von Kontrollstrukturen erfolgt mit Hilfe von UML Aktivitätsdiagrammen während

[4] http://www.fujaba.de

UML Kollaborationsdiagramme[5] als Notation für Graphersetzungsregeln verwendet werden. Die Übersetzung von Storydiagrammen in Java Klassen und Methoden erlaubt eine nahtlose Integration von Systemteilen, die mittels Objektorientierung einerseits und durch Graphgrammatiken andererseits spezifiziert wurden. Fujaba bietet hierbei keine Backtracking Semantik [FNTZ99], was die direkte Übersetzung von Storydiagrammen in objekt-orientierten Quelltext erleichtert.

Abbildung 3.6 zeigt ein Storydiagramm, das im MOD2-SCM Projekt zur Implementierung der Speicherung von gemischten Deltas dient. Da Storydiagramme auf UML Aktivitätsdiagrammen zur visuellen Repräsentation des Kontrollflusses basieren, bestehen sie aus Aktivitäten, die durch Transitionen miteinander verbunden sind. Diese Aktivitäten werden durch Rechtecke mit abgerundeten Ecken symbolisiert. Die Ausführung startet bei der eindeutigen Startaktivität, die durch einen ausgefüllten schwarzen Kreis markiert wird. Der Kontrollfluss folgt den ausgehenden Transitionen. Sind an einer Aktivität mehrere ausgehende Transitionen vorhanden, so werden diese mit boolschen Ausdrücken überwacht, die in eckigen Klammern angegeben werden. Der Kontrollfluss endet bei einer[6] Stopaktivität, die durch einen ausgefüllten Kreis, der mit einer zusätzlichen abgesetzten Kreislinie dekoriert wird, dargestellt wird.

Storydiagramme können zwei unterschiedliche Arten von Aktivitäten enthalten: (1) Story-Muster und (2) Statement Aktivitäten. Während letztere handgeschriebenen Quelltext enthält, der bei der Codeerzeugung 1:1 an der entsprechenden Stelle eingesetzt wird, stellt ein Story-Muster eine Graphersetzungsregel dar, die die linke und rechte Regelseite in einem einzigen Bild zeigt. Die linke Seite der Graphersetzungsregel wird durch eine Art Objektdiagramm dargestellt und zeigt den Ausschnitt des Objektgraphen, der geändert werden soll. Die Erzeugung neuer Knoten und Kanten wird durch Darstellung der jeweiligen Objekte in grüner Farbe und das Schlüsselwort «*create*» symbolisiert, die Löschung hingegen wird mit roter Farbe und dem Schlüsselwort «*destroy*» angezeigt. In Storydiagrammen wird zwischen gebundenen und ungebundenen Variablen unterschieden. Während ungebundene Variablen immer durch den Variablennamen gefolgt von dem Objekttyp (also z.B. **anItem : StorageItem**) dargestellt werden, werden gebundene Variablen nur mit ihrem Namen angezeigt. Methodenparameter und die Selbstreferenz **this** sind ebenfalls gebundene Variablen. Ein Link in einem Story-Muster gibt an, dass die durch die entsprechenden Objekte repräsentierten Variablen miteinander verbunden sind. Die Ausführung eines Story-Musters erfolgt durch Binden aller ungebunden Variablen, so dass die dadurch ausgedrückte Bedingung erfüllt wird. Ist dies möglich, so werden die angegebenen Veränderungen ausgeführt und das Story-Muster wird erfolgreich abgearbeitet, anderfalls schlägt die Abarbeitung fehl. Bei Objekten können zusätzlich noch Vergleiche bzw. Wertzuweisungen an Attributen erfolgen. Eine Zuweisung wird ebenfalls durch grüne Farbe symbolisiert. Verzweigungen im Kontrollfluss, bzw. die Zusammenführung von Kontrollflüssen, werden durch Rauten dargestellt. Eine Iteration

[5] Das Fujaba Metamodell basiert noch auf UML 1.3. In der aktuellen Version 2.2 von UML gibt es keine Kollaborationsdiagramme mehr, sie wurden ersetzt durch Kommunikationsdiagramme

[6] Im Gegensatz zur eindeutigen Startaktivität kann ein Storydiagramm beliebig viele Stopaktivitäten besitzen.

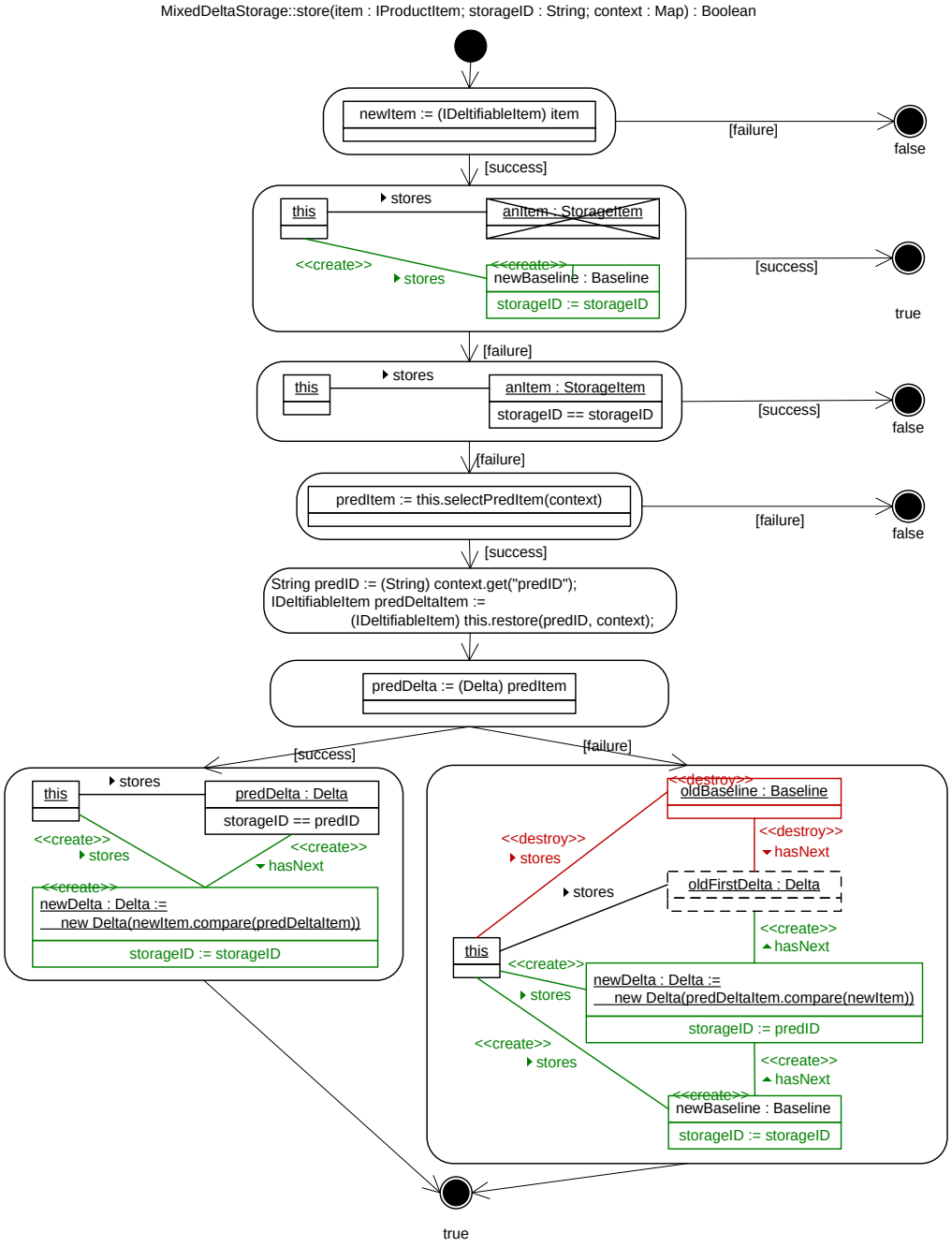

Abbildung 3.6: Storydiagramm zur Speicherung von gemischten Deltas.

kann durch eine sogenannte *for-each* Aktivität ausgeführt werden. Im Gegensatz zu einem normalen Story-Muster wird diese Art von Story-Muster durch ein abgerundetes Rechteck mit einem leicht versetzten zweiten abgerundetem Rechteck dargestellt. Die

Iteration läuft dabei so lange, bis das dargestellte Muster im Objektgraphen nicht mehr gefunden werden kann.

3.5 Zusammenfassung

In diesem Kapitel wurden die Grundlagen der modell-getriebenen Softwareentwicklung erklärt. Nach der Definition des Begriffs erfolgte eine kurze geschichtliche Motivation, die die Notwendigkeit der Verwendung formaler Modelle bei der Entwicklung komplexer Softwaresysteme aufzeigt. Im Anschluss wurde die von der Object Management Group (OMG) propagierte Vorgehensweise der *Model Driven Architecture (MDA)* erläutert. Diese Art der modell-getriebenen Softwareentwicklung setzt ganz auf die von der OMG befürwortete Modellierungssprache UML. Im Folgenden wurden die Kernbestandteile der UML kurz dargelegt und zwei der für diese Arbeit relevanten Diagrammarten erläutert. Die *Meta Object Facility (MOF)*, das Metamodell von UML bzw. die Kernkomponente *EMOF* bildet die Grundlage des *Eclipse Modeling Framework (EMF)*, das im Anschluss kurz eingeführt wird. EMF bildet mit der Integration in die Eclipse Plattform und den bereitgestellten Codegeneratoren die Basis zur Entwicklung von Modellen für domänenspezifische Sprachen. Im letzten Abschnitt wurde das CASE Werkzeug Fujaba mit der formalen, auf Graphersetzungsregeln basierenden Sprache der Storydiagramme eingeführt. Durch die Möglichkeit, ausführbare Modelle zu spezifizieren, die mit Hilfe des Fujaba Compilers in ausführbaren Quelltext übersetzt werden können, wird Fujaba im Rahmen dieser Arbeit zur Spezifizierung des Domänenmodells in der *Domänenentwicklung* eingesetzt.

4 Generierung graphischer Werkzeuge für DSLs

Die primäre Zielsetzung dieser Arbeit ist die Entwicklung von Werkzeugen zur Entwicklung von Softwareproduktlinien. Dabei stehen speziell Werkzeuge zur Verbindung von Featuremodellen mit den Domänenmodellen und zur Generierung der einzelnen Produktinstanzen im Vordergrund. Um dieses Ziel zu erreichen, werden die dafür benötigten Werkzeuge mit Hilfe von Fujaba modellgetrieben entwickelt, indem eine Modellierungssprache in Fujaba definiert wird, aus der dann ausführbarer Quelltext erzeugt wird. Im Anschluss daran werden (wie im Falle des in Kapitel 5 vorgestellten Paketdiagrammeditors) interaktive graphische Werkzeuge entwickelt. Allgemein formuliert bedeutet dies, dass zuerst eine Sprache definiert wird und anschließend aus dieser Definition Werkzeuge generiert werden. Dieser DSL-Ansatz kann als Produktlinienansatz interpretiert werden, da die generierten Werkzeuge die Instanzen der Produktlinie darstellen.

Der Beitrag dieser Arbeit zur Entwicklung von domänenspezifischen Sprachen liegt in der Verbindung von Fujaba (zur Definition der abstrakten Syntax bzw. der Applikationslogik) und GMF (zur Generierung von graphischen Editoren) mit Hilfe der EMF-Codegenerierung für Fujaba [BDG07]. Dieses Verfahren wurde im Rahmen dieser Arbeit zur Erstellung des Paketdiagrammeditors (siehe Kapitel 5) angewendet, es ist aber für beliebige DSLs anwendbar.

In weiteren Verlauf dieses Kapitels wird zunächst der Begriff *domänenspezifische Sprache* erläutert und die Relevanz für Softwareproduktlinien dargelegt. Anschließend folgt eine Diskussion des modellgetriebenen Entwicklungsansatzes für domänenspezifische Sprachen, der im Rahmen dieser Arbeit verwendet wurde.

4.1 Domänenspezifische Sprachen

In Kapitel 3 wurde die Benutzung von formalen Modellen zur Spezifikation von Softwaresystemen erläutert. Die beschriebenen Modellierungssprachen, wie etwa UML [OMG09b], sind dabei sehr mächtig und für ein breites Spektrum an Anwendungen ausgelegt. Eine domänenspezifische Sprache hingegen, im englischen domain-specific language oder kurz DSL genannt, zeichnet sich gegenüber herkömmlichen Programmier- bzw. Modellierungssprachen durch eine Spezialisierung auf eine konkrete Anwendungsdomäne aus. Dadurch lassen sich mit ihr typischerweise nur die Aufgabenstellungen aus dieser Domäne lösen. Die in einer DSL implementierten Lösungen sind geprägt durch einen geringeren Aufwand durch die Verwendung von auf die Domäne spezialisierten Sprachelementen

und natürlicher Begriffe aus der Anwendungsdomäne. In letzter Zeit entstanden vermehrt solche Sprachen bzw. Werkzeuge, die die Erstellung von domänenspezifischen Sprachen erlauben, wie etwa das Eclipse Modeling Project [Gro09].

4.1.1 Motivation

Auch heute noch wird Software mit universell einsetzbaren Programmiersprachen entwickelt. Diese Sprachen, wie etwa Java, C++, C# oder ähnliche, verfügen über eine sehr große Ausdrucksmächtigkeit, was dazu führt, dass sie für fast jeden denkbaren Zweck einsetzbar sind. Diese hohe Flexibilität führt auf der anderen Seite allerdings zu zunehmender Komplexität und Unübersichtlichkeit, was wiederum gut ausgebildete Entwickler erfordert. Aber auch diese gut ausgebildeten Entwickler benötigen für die Umsetzung konkreter Problemstellungen eine genaue Spezifikation des zu entwickelnden Programms. Nachdem das Programm erstellt wurde sind aufwändige Tests nötig, um sicherzustellen, dass es zuverlässig und gemäß der Spezifikation funktioniert.

In der Praxis entstehen dadurch diverse Probleme. Zum einen entstehen hohe Aufwände für die Erstellung von Spezifikationen und Tests und zum anderen ist die entstandene Software dennoch häufig fehleranfällig und erfüllt nicht genau die Vorgaben der Spezifikation. Dies führt zu mitunter kosten- und zeitintensiven Nachbesserungen. An dieser Stelle können nun domänenspezifische Sprachen in gewissen Situationen zur Lösung dieser Probleme beitragen, indem die Software nicht mehr mittels universell einsetzbarer Sprachen entwickelt wird, sondern spezielle, auf die Anwendungsdomäne optimierte Sprachen verwendet werden. Allgemein formuliert bieten domänenspezifische Sprachen einen hohen Abstraktionsgrad und sie stellen spezifische Konzepte aus der Anwendungsdomäne zur Verfügung. Quelltext, der in einer solchen Sprache erstellt wurde kann später voll automatisch in Quelltext einer beliebigen universellen Sprache übersetzt werden. Aufgrund des normalerweise deutlich geringeren und auf die jeweilige Domäne angepassten Sprachumfangs einer DSL verglichen mit einer herkömmlichen Programmiersprache, ist deutlich weniger Zeit und Quelltext notwendig und das Programm kann unter Umständen auch durch Anwendungsexperten erstellt werden.

Die Erstellung eines Programms in der DSL erfolgt mit Hilfe von speziellen Editoren. Diese Editoren stellen Sprachelemente der DSL durch Text oder graphische Elemente bereit. Durch Kenntnis der Anwendungsdomäne können bei der Entwicklung dieser Editoren bereits umfangreiche Konsistenzprüfungen implementiert werden, so dass eine weitreichende Sicherstellung fachlicher Rahmenbedingungen gewährleistet ist.

In diesem Zusammenhang könnte man Fujaba selbst, aber auch die damit definierten Sprachen als domänenspezifische Sprache zur Beschreibung von Softwaresystemen auffassen. Die hohe Komplexität der Modellierungssprache UML, die sich durch die Vielzahl an angebotenen Diagrammarten ausdrückt, wird in Fujaba auf zwei wesentliche Diagrammarten reduziert: Klassendiagramme und Storydiagramme. Während UML ein breites Spektrum an möglichen Einsatzgebieten abdeckt, zielt Fujaba einzig auf die Erstellung ausführbarer Modelle von Softwaresystemen ab.

4.1.2 Unterschied zur MDA

Beim von der OMG propagierten Model Driven Architecture Ansatz findet die Spezifikation des Softwaresystems gewöhnlich in zwei Ebenen statt: es werden plattformunabhängige Modelle (PIMs) und plattformspezifische Modelle (PSMs) verwendet. Die PSMs gehen dabei aus den PIMs durch Transformation hervor und dienen als Basis für die Codeerzeugung. Das Ziel von MDA ist, dasselbe PIM auf verschiedenen Software Plattformen nutzen zu können. Die OMG verfolgt außerdem das Ziel, alle Modellformate zu standardisieren und somit einen einfachen Austausch der Modelle zwischen den Werkzeugen unterschiedlicher Hersteller zu gewährleisten. Dies unterscheidet sich fundamental vom Ansatz der domänenspezifischen Sprachen. Durch die strikte Fokussierung auf die jeweilige Anwendungsdomäne steckt ein hoher Grad an Fachwissen in einer DSL. Plattformunabhängigkeit spielt hierbei – wenn überhaupt – nur eine untergeordnete Rolle. Es ist anzumerken, dass Plattformunabhängigkeit sehr wohl auch mit DSLs erreicht werden kann, indem verschiedene Generatoren eingesetzt werden, die für die spezifische Zielplattform hin optimiert sind. Juha-Pekka Tolvanen sieht das Hauptziel der Modellierung mit domänenspezifischen Sprachen darin, die Produktivität der Entwickler durch Erhöhung des Abstraktionsgrades zu verbessern. Er führt weiterhin aus, dass die Anwendung von MDA zudem gewisse Kenntnisse der Methodik voraussetzt, die erst durch Erfahrung erlangt werden müssen [Tol06].

4.2 Anwendung für Produktlinien

In [CE00] beschreiben Czarnecki und Eisenecker Methoden, Werkzeuge und Anwendungen der sogenannten *generativen Programmierung*. Unter dem Begriff *generative Programmierung* (GP) versteht man ein Paradigma in der Softwareentwicklung, bei dem Softwarefamilien derart modelliert werden, dass anhand einer detaillierten Anforderungsspezifikation ein hochgradig angepasstes und optimiertes Zwischen- oder Endprodukt automatisch aus elementaren, wiederverwendbaren Komponenten erzeugt werden kann. In [Cza04] beschreibt Czarnecki den Unterschied zwischen der Entwicklung von Softwarefamilien und der Produktlinienentwicklung. Während bei der Entwicklung von Softwarefamilien die Erzeugung von Systemen aus gemeinsamen Komponenten im Vordergrund steht, umfasst die Entwicklung von Softwareproduktlinien zusätzlich noch die Ermittlung des Bereichs, den die Produktlinie abdeckt sowie die Verwaltung von gemeinsamen Produkteigenschaften aus der Marketingperspektive. Folglich findet auch bei der Entwicklung von Softwarefamilien eine Unterscheidung zwischen *Domänenentwicklung* und *Anwendungsentwicklung* statt (vgl. Kapitel 2).

Die Idee hinter der Entwicklung von Softwarefamilien ist die Ausnutzung der Gemeinsamkeiten von Systemen einer bestimmten Domäne und der systematischen Verwaltung ihrer Unterschiede [Cza04]. Das in Abbildung 4.1 dargestellte *Generative Domänenmodell* zeigt den zu Grunde liegenden Ansatz. Die Beschreibung des Problemraums, der die domänenspezifischen Begriffe und Merkmale enthält, erfolgt durch eine domänenspezifische Sprache (DSL). Mit Hilfe dieser DSL wird nun ein Mitglied der Systemfamilie

spezifiziert. Mittels eines Konfigurators, der das Konfigurationswissen implementiert, wird die Spezifikation des konkreten Produkts durch Standardvorgaben ergänzt sowie Abhängigkeiten und ungültige Konfigurationen geprüft. Die Umsetzung der Konfiguration erfolgt durch Kombination der vorgefertigten Komponenten, die Bestandteil des Lösungsraums sind, zu einem ausführbaren System. Ziel der generativen Programmierung ist hierbei die vollständig automatisierte Erstellung dieser ausführbaren Systeme.

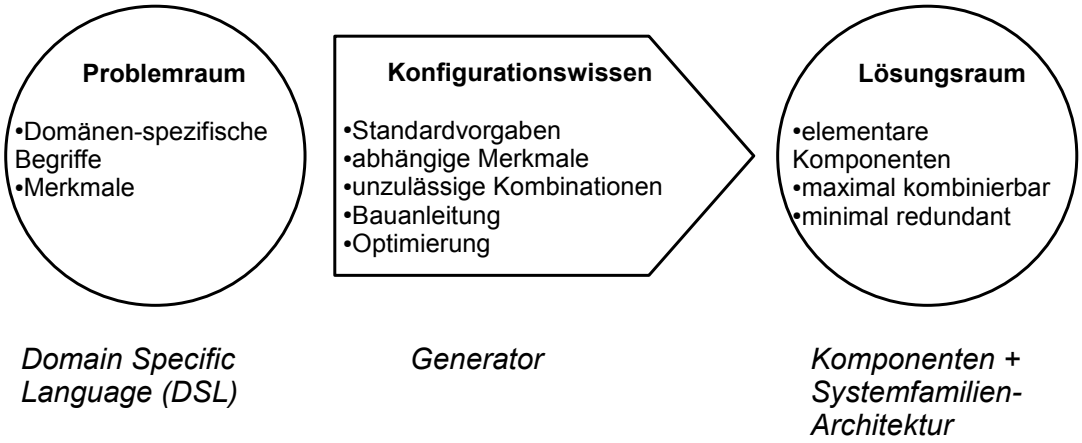

Abbildung 4.1: Generatives Domänenmodell, siehe [GPR06] S. 45

Das Konzept der generativen Programmierung hat einige Gemeinsamkeiten mit dem MDA Ansatz der OMG. Während der Hauptunterschied der beiden Ansätze darin besteht, dass die MDA Softwarefamilien überhaupt nicht berücksichtigt, entspricht die Transformation der plattformunabhängigen Modells (PIM) in ein plattformspezifisches (PSM) hingegen der Abbildung des Problemraums auf den Lösungsraum [Cza04]. Laut Czarnecki [Cza04] können aber durchaus auch interessante Synergien aus beiden Ansätzen gewonnen werden. Die Vorteile der MDA liegen seiner Meinung nach unter anderem in den Standards zur Beschreibung und Manipulation von Modellierungssprachen. Dies führt dazu, dass die generativen Konzepte mehr und mehr in Umlauf gebracht werden. Desweiteren zielt die MDA auf Plattformunabhängigkeit ab, was in anderen Worten Softwarefamilien im Bezug auf Variation in der Technologie bedeutet. Im Gegensatz dazu behandelt die generative Softwareentwicklung Variabilität in Technologie und Anwendungsdomäne. Abbildung 4.2 zeigt eine Klassifizierung der generativen Programmierung und MDA im Bezug auf Variabilität unter diesen beiden Aspekten.

4.3 Modellgetriebene Entwicklung von DSLs

In diesem Abschnitt wird eine allgemeine Vorgehensweise zur Generierung graphischer Werkzeuge mit Hilfe von Fujaba und GMF vorgestellt. Auf diese Weise können domänenspezifische Sprachen und die zugehörigen Editoren modellgetrieben entwickelt werden.

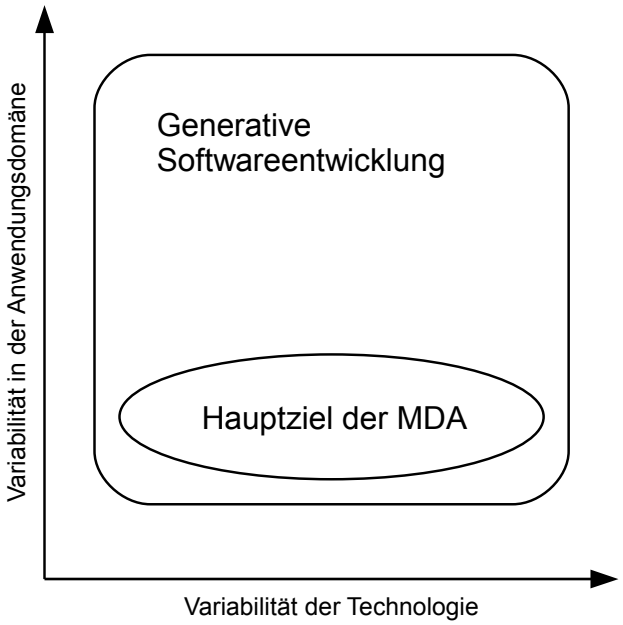

Abbildung 4.2: Zusammenhang zwischen generativer Softwareentwicklung und MDA nach Czarnecki [Cza04].

Im Rahmen dieser Arbeit wurde etwa der in Kapitel 5 vorgestellte Paketdiagrammeditor auf diese Weise entwickelt. Die Grundlagen des Generierungsansatzes für graphische Werkzeuge werden im Folgenden erläutert. Diese Editoren können graphische oder textuelle Unterstützung anbieten, oder aber auch die Kombination von graphischer und textueller Notation. Monticore[1], ein auf Graphgrammatiken basierendes System zur Generierung von DSLs mit syntaxgestützten textuellen Editoren, oder MetaEdit+[2] zur Generierung von graphischen Werkzeugen wären hier als Beispiele zu nennen. Aber auch mit Hilfe von Fujaba und den für Eclipse konzipierten Modellierungswerkzeugen EMF (Eclipse Modeling Framework) und GMF (Graphical Modeling Framework) kann dies auf einfache Art und Weise erreicht werden. David S. Frankel argumentiert in seinem Buch *Model Driven Architecture: Applying MDA to Enterprise Computing* [Fra03], dass ein kritischer Aspekt für den Erfolg von modellgetriebener Softwareentwicklung die Benutzung eines gemeinsamen Metamodells ist. Bei der modellgetriebenen Entwicklung von DSLs mit den Bestandteilen des Eclipse Modeling Projects[3] ist Ecore dieses gemeinsame Metamodell.

Abbildung 4.3 zeigt eine Übersicht über das Eclipse Modeling Project. Das Eclipse Modeling Framework bildet zusammen mit *Query : Validation : Transaction* den Kern. Darauf aufbauend folgen Technologien zur Modelltransformation. Dies sind so-

[1] http://www.monticore.org/
[2] http://www.metacase.com/
[3] http://www.eclipse.org/modeling/

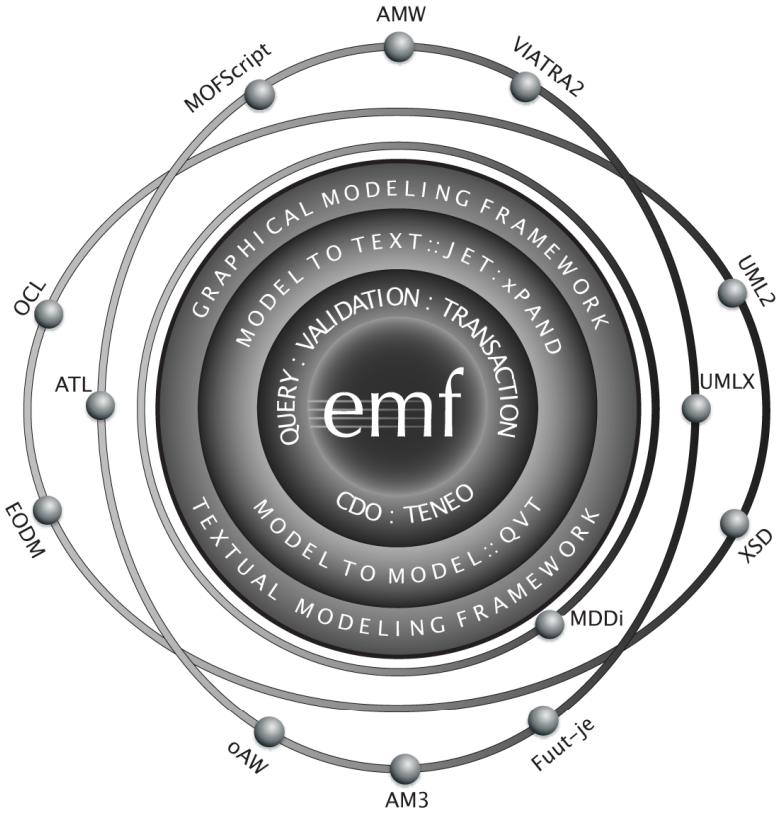

Abbildung 4.3: Übersicht über das Eclipse Modeling Project nach [Gro09], S. 9.

wohl Modell-zu-Text Transformationen, wie etwa JET (Java Emitter Templates) oder xPand, die zur Codeerzeugung verwendet werden, als auch Modell-zu-Modell Transformationen, wie etwa eine Implementierung von *QVT (Query, View, Transformation)*. Darauf folgen dann zwei Rahmenwerke zur Erstellung von textuellen (*Textual Modeling Framework*) bzw. graphischen Editoren (*Graphical Modeling Framework*) für die erstellte DSL. Mit Hilfe des Eclipse Modeling Project kann Eclipse als *„language workbench“*, wie Martin Fowler [Fow05] Werkzeuge zur generativen Entwicklung von DSLs bezeichnet, verwendet werden.

4.3.1 Definition der abstrakten Syntax

Im Kontext von DSLs wird oftmals von abstrakter und konkreter Syntax gesprochen. Mit der abstrakten Syntax wird in diesem Zusammenhang das Metamodell der DSL bezeichnet, die konkrete Syntax beschreibt die textuelle oder graphische Notation. Typischerweise beginnt die Entwicklung einer DSL immer mit der Spezifikation ihrer abstrakten Syntax. Um im weiteren Verlauf die Vorteile des Eclipse Modeling Projects zur einfachen Erstellung von domänenspezifischen Sprachen nutzen zu können, muss das Metamodell, die Beschreibung der abstrakten Syntax der DSL, als Ecore Modell vorliegen.

Da aber mit Ecore keine ausführbaren Modelle erzeugt werden können, wird Fujaba zur Spezifikation der abstrakten Syntax verwendet. Mit Hilfe der EMF-Codegenerierung für Fujaba [BDG07] ist es möglich, aus dem Fujaba Klassendiagramm ein Ecore Modell zu erzeugen[BDW07]. Die EMF-Codegenerierung wurde in Kooperation mit der Universität Kassel entwickelt. Die in den Storydiagrammen spezifizierten Methodenimplementierungen werden dabei direkt in Java Quelltext übersetzt. In einem zweiten Generierungsschritt werden mit Hilfe des EMF-Codegenerators die fehlenden Quelltextfragmente aus dem Ecore Modell dazu erzeugt. Das Endergebnis ist somit ausführbarer Java Quelltext.

Hierbei ist der Einsatz von Fujaba nicht zwingend notwendig. Soll die DSL nicht vollständig modellgetrieben entwickelt werden, so kann das Ecore Modell mit geeigneten Ecore Werkzeugen spezifiziert werden. Eine Verfeinerung von Struktur und Semantik der DSL kann anschließend beispielsweise mit Hilfe der Object Constraint Language (OCL) erfolgen, um Unterstützung für Abfragen, Transaktionen und Validierung hinzuzufügen.

Codeerzeugung

Um aus einem Ecore Modell mit Hilfe der Modell-zu-Text Transformation Quelltext zu erzeugen, muss ein sogenanntes Generatormodell erstellt und mit zusätzlichen, für die Codeerzeugung notwendigen Informationen versehen werden. Dieses Generatormodell dient als Eingabe für die *Java Emitter Templates (JET)*, die zur Erzeugung von Java und anderen Textdateien eingesetzt werden. Die Ausgabe kann dabei durch eigene Templates angepasst werden. Der generierte Quelltext kann durch manuelle Erweiterungen, wie etwa Methodenimplementierungen ergänzt werden. Bei einer erneuten Codeerzeugung kann dieser Quelltext durch Java Annotationen vor Veränderungen durch die automatische Codegenerierung geschützt werden. Bei der Verwendung von Fujaba zur Erstellung eines ausführbaren Modells zur Beschreibung der abstrakten Syntax sind in der Regel keine manuellen Anpassungen des Quelltextes notwendig. Allerdings ist hierbei zu beachten, dass ein zweistufiger Prozess zur Generierung von ausführbarem Quelltext durchlaufen werden muss. Zuerst wird aus dem Fujaba Modell mit Hilfe der EMF-Codegenerierung ein entsprechendes Ecore Modell, sowie Java Klassen mit den Methodenimplementierungen erzeugt. Anschließend muss zu dem generierten Ecore Modell ein zugehöriges Generatormodell erstellt werden, das anschließend als Basis für die Codeerzeugung mit Hilfe der Java Emitter Templates dient. Aus dem Generatormodell wird Quelltext für mehrere Aufgaben erzeugt:

Abstrakte Syntax Zum einen wird das spezifizierte Ecore Modell in ein Java Modell übersetzt.

Änderungsoperationen Es wird Quelltext erzeugt, der Kommandos und Transaktionen zur Veränderung des Laufzeitmodells in Form von Knoten und Kantenmanipulationen im EMF Kontext erlaubt.

Baumeditor Ein graphischer Editor wird erzeugt, der Instanzen des Modells in einer Baumansicht darstellt und die oben genannten Änderungsoperationen verwendet.

Testfälle Aus dem Ecore Modell werden zudem noch einfache Testfälle erzeugt, die vom Benutzer durch weitere Testfälle ergänzt werden können.

4.3.2 Definition der konkreten Syntax

Oftmals wird eine DSL mittels graphischer Notation repräsentiert. Selbstverständlich ist dies nicht bei jeder Art von domänenspezifischer Sprache möglich, oder nicht alle Aspekte der DSL können adäquat in graphischer Art und Weise dargestellt werden. Daher ist auch eine Kombination von textueller und graphischer Notation für die Darstellung der konkreten Syntax möglich und oftmals auch die beste Lösung. In diesem Abschnitt wird jedoch nur die Erstellung einer graphischen Notation für die DSL behandelt. Das Graphical Modeling Framework (GMF) benutzt zur Spezifikation der einzelnen Komponenten eines graphischen Editors eine Sammlung von EMF Modellen, die ihrerseits wieder DSLs darstellen. Mit Hilfe von GMF wird die konkrete Syntax der DSL entwickelt.

GMF Überblick

GMF setzt auf die Technologie des *Graphical Editing Framework (GEF)*[4] von Eclipse auf, das eine Unterstützung für einheitliche Implementierungen von graphischen Editoren bietet. GEF basiert auf dem *Model-View-Controller (MVC)* Muster (siehe [FF04]). Hierdurch ist eine klare Trennung des Datenmodells und der entsprechenden Repräsentation an der Benutzerschnittstelle möglich. Modell und Benutzerschnittstelle (View) sind durch den Controller miteinander verbunden. Die Aufgaben des Controllers umfassen die Aktualisierung der Anzeige, falls sich das Datenmodell ändert und die Weiterleitung von Anfragen von der Benutzerschnittstelle zum Modell. Im Zuge eines generativen Ansatzes kombiniert GMF nun EMF und GEF, um graphische Editoren zu erstellen. GMF besteht dabei aus zwei Komponenten:

Laufzeitkomponente Die Laufzeitkomponente übernimmt die Verbindung von EMF und GEF und stellt außerdem eine Reihe von Diensten und Programmierschnittstellen (API) zur Verfügung, die dem Entwickler eine weitere Anpassung des generierten Editors erlauben.

Werkzeugkomponente Die Werkzeugkomponente ermöglicht einen modellgetriebenen Ansatz zur Beschreibung der graphischen Elemente, der Werkzeugpalette und der Abbildung auf das Domänenmodell. Außerdem enthält die Werkzeugkomponente Codegeneratoren zur Erzeugung von lauffähigem Quelltext für den graphischen Editor.

GMF benutzt ein separates Laufzeitmodell, um die Informationen aus dem Diagramm, wie etwa Größe und Position der angezeigten Elemente, unabhängig von den Informationen des Domänenmodells zu speichern.

Abbildung 4.4 zeigt die am modellgetriebenen Prozess zur Entwicklung einer DSL mit graphischer Notation beteiligten Modelle und deren Abhängigkeiten untereinander.

[4] http://www.eclipse.org/gef/

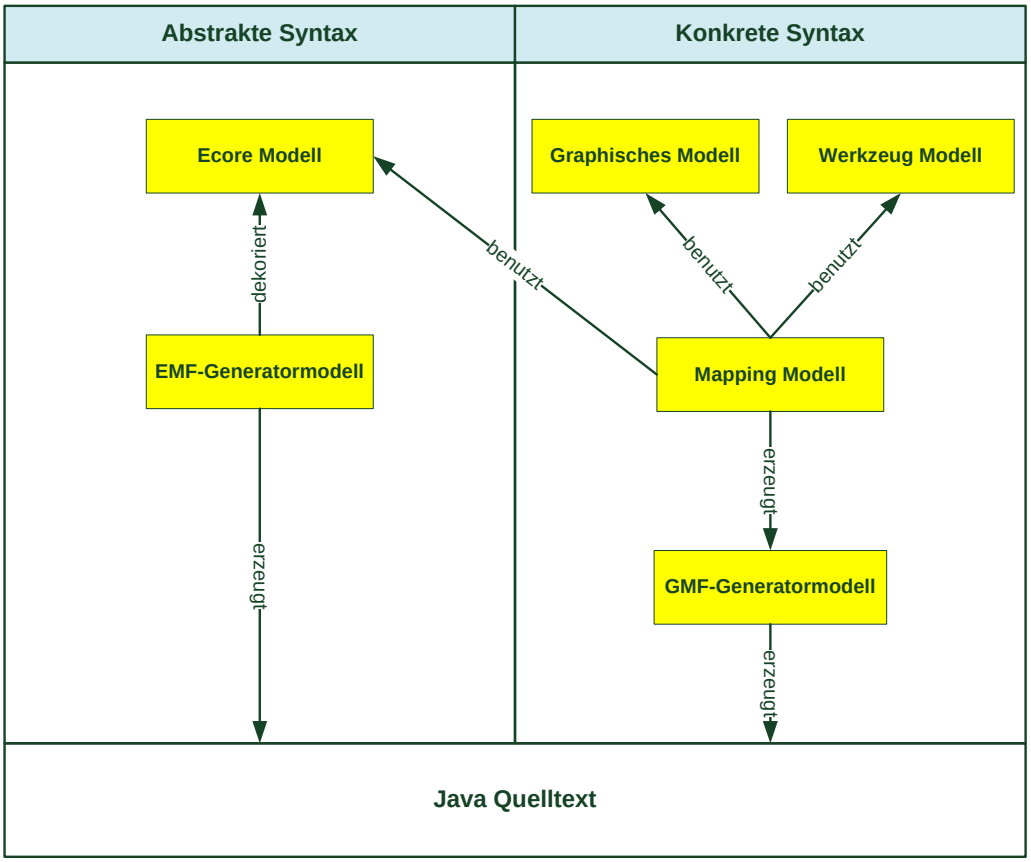

Abbildung 4.4: Modelle und Abhängigkeiten bei der modellgetriebenen Entwicklung von DSLs mit graphischer Notation.

Arbeitsablauf

In Abbildung 4.5 ist ein typischer Arbeitsablauf bei der Erstellung eines graphischen Editors mit Hilfe von GMF zu sehen. Im Unterschied zur Modellierung mit EMF sind in diesem Fall drei verschiedene Modelle notwendig. Zusätzlich zu dem in EMF definierten Domänenmodell, muss je ein Modell zur Beschreibung der graphischen Elemente (*Graphical Definition Model, *.gmfgraph*), sowie zur Definition der Werkzeugpalette (*Tooling Definition Model, *.gmftool*) erstellt werden. Im Anschluss daran werden diese Modelle miteinander verknüpft, und so eine Verbindung zwischen Elementen des Domänenmodells, ihrer zugehörigen graphischen Repräsentation und den zur Erstellung benötigten Werkzeugen hergestellt. Dieses Modell, *Mapping Model (*.gmfmap)* genannt, muss nun noch mit zusätzlichen, für die Codeerzeugung benötigten Informationen angereichert werden, so dass im letzten Schritt ein lauffähiger Editor erzeugt werden kann.

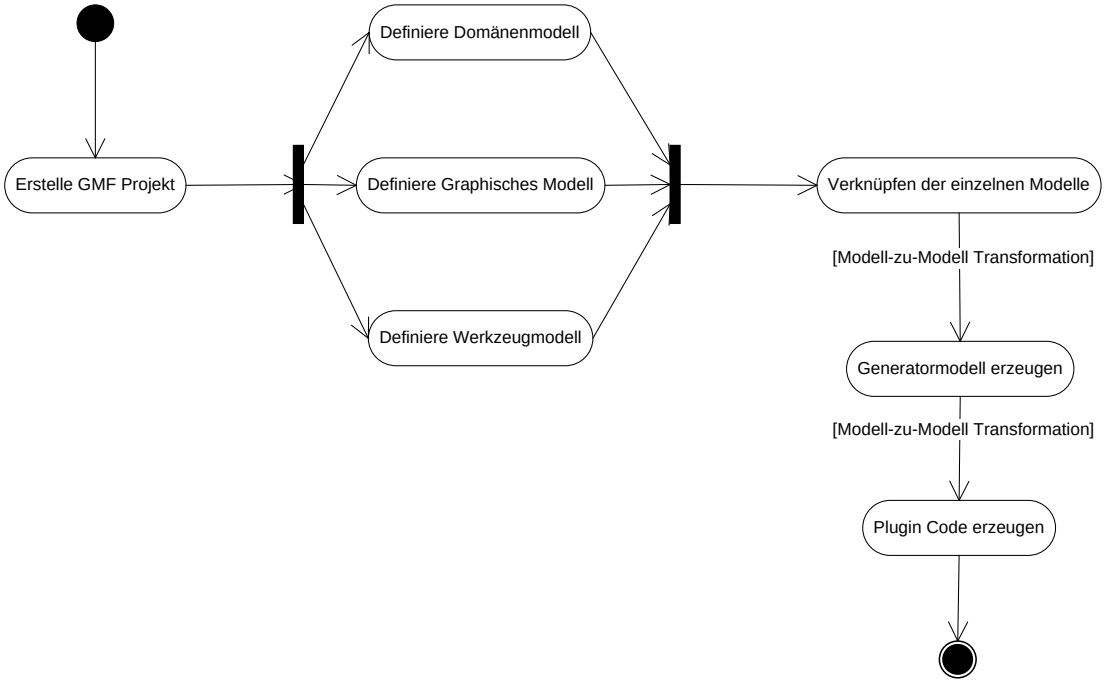

Abbildung 4.5: Typischer Arbeitsablauf bei der Modellierung mit GMF.

Definition der graphischen Elemente

Die Definition der graphischen Elemente des Editors erfolgt im sogenannten *Graphical Definition Model*, das aus zwei Teilen besteht und die konkrete Syntax beschreibt. Zum einen enthält das Modell eine *Figure Gallery*, in der das Aussehen von wiederverwendbaren Formen (Rechtecke, Linien, Textfelder) festgelegt wird. Neben den vorgegebenen einfachen geometrischen Formen zur Darstellung von Knoten und Kanten, können auch selbstdefinierte Formen eingebunden werden. Diese Formen können dabei entweder als SVG (Scalable Vector Graphics) – einer auf XML basierenden Sprache zur Beschreibung von Vektorgrafiken – vorliegen oder direkt als Java Implementierung basierend auf *Draw2D*[5]. Der zweite Teil beschreibt die Elemente, die auf der Zeichenfläche des Editors dargestellt werden. Diese referenzieren die in der Figure Gallery definierten Elemente, um Knoten, Kanten, Bereiche und Diagrammbeschriftungen zu beschreiben. Die Wiederverwendbarkeit der Figure Gallery stellt hierbei einen bedeutenden Schritt zur Umsetzung des modellgetriebenen Ansatzes dar. Viele Diagramme erfordern ähnlich aussehende Elemente, so dass hiermit eine erhebliche Aufwandsreduzierung möglich ist. Um die Erstellung der Formen zu erleichtern existiert ein experimenteller WYSIWYG Editor.

[5]Draw2D ist Bestandteil des Eclipse Rahmenwerks. Das Graphical Editing Framework, auf dem auch GMF basiert, benutzt Draw2D im Rahmen des MVC-Musters zur Realisierung der *View*-Komponente

Definition der Werkzeugpalette

Graphische Editoren enthalten für gewöhnlich eine Werkzeugpalette, damit der Benutzer des Editors die entsprechenden Knoten- und Kantentypen komfortabel in das Diagramm einfügen kann. GMF bietet mit dem *Tooling Definition Model* eine Möglichkeit, Elemente der Werkzeugpalette, der Werkzeugleiste und verschiedene Menüs für ein Diagramm zu spezifizieren. Allerdings ist es mit der aktuellen (Stand 09. April 2010) Version des Generators nur möglich, die Elemente der Werkzeugpalette in Quelltext umzuwandeln. Selbstverständlich ist es ebenso möglich, einen Editor ohne Werkzeugpalette zu erstellen, wenn etwa Diagramme nur angezeigt und nicht verändert werden sollen.

Verbindung der Modelle

Das wichtigste aller in GMF verwendeten Modelle stellt das sogenannte *Mapping Model* dar. Elemente des Domänenmodells werden hier mit den entsprechenden Werkzeugdefinitionen und den Elementen der graphischen Definition (Knoten und Kanten) verbunden. Dieses Modell dient als Grundlage für das Generatormodell, das mittels Modelltransformation erzeugt wird. Im Mapping Model wird OCL in vielfältiger Art und Weise eingesetzt. Zum einen dient es dazu, Eigenschaftswerte für erzeugte Elemente zu initialisieren, Bedingungen für Knoten und Kanten anzugeben und zum anderen können damit Modellprüfungen und Metriken auf dem Modell realisiert werden.

Generatormodell

Das von GMF verwendete Generatormodell weist gewisse Ähnlichkeiten zum Generatormodell von EMF auf. Auch hier werden mit Hilfe dieses Modells zusätzliche, zur Codeerzeugung benötigte Informationen dem Mapping Model hinzugefügt. In der aktuellen Implementierung von GMF wird diese Modelltransformation des Mapping Models in das Generatormodell durch Java Quelltext umgesetzt. Der Plan für die Weiterentwicklung von GMF sieht aber vor, diese Transformation durch QVT zu beschreiben, was die Flexibilität und Anpassbarkeit für den Anwender verbessert [Gro09]. Analog zur EMF-Codegenerierung können auch für die GMF Codeerzeugung eigene Vorlagen verwendet werden. Der Unterschied besteht lediglich in der verwendeten Sprache. Im Gegensatz zu den JET-Templates, die EMF verwendet, kommt bei GMF Xpand zum Einsatz.

4.3.3 Modell-zu-Text Transformationen

Ziel von domänenspezifischen Sprachen ist eine Erhöhung des Abstraktionsgrads bei der Entwicklung von lauffähigen Softwaresystemen. Daher muss aus dem mit der DSL beschriebenen Modell am Ende typischerweise Quelltext erzeugt werden. Innerhalb des Eclipse Modeling Projects gibt es mehrere Ansätze, um Text aus Modellen zu erzeugen. Das M2T Projekt enthält hierfür die Java Emitter Templates (JET) und Xpand[6]. Während die JET Technologie als Codeerzeugung für EMF dient, wird Xpand bei der

[6] http://www.eclipse.org/modeling/m2t/?project=xpand

Erstellung eigener DSLs favorisiert. Ursprünglich wurde Xpand im Rahmen von *openArchitectureWare (oAW)*[7] entwickelt. Durch die Migration von oAW in das Eclipse Modeling Project, wird Xpand nun in diesem Kontext gepflegt und weiterentwickelt. Xpand dient außerdem zur Generierung von Quelltext aus dem GMF Generatormodell.

4.3.4 Modellgetriebene Entwicklung graphischer Editoren mit Fujaba, EMF und GMF

Die Produktivität bei der Entwicklung von graphischen Werkzeugen aber auch von DSLs kann durch die Verwendung von Fujaba weiter gesteigert werden. Die Möglichkeit, mit Fujaba ausführbare Modelle zu spezifizieren, erlaubt so die Definition des Metamodells des Werkzeugs bzw. der abstrakten Syntax der DSL, ohne weitere manuelle Anpassungen und Erweiterungen des generierten Quelltextes.

GMF andererseits erlaubt die schnelle Generierung von graphischen Editoren, für beliebige EMF-basierte objektorientierte Datenmodelle mit Hilfe des Graphical Editing Framework (GEF)[8] von Eclipse. Allerdings ist die Funktionalität der generierten Editoren auf vordefinierte elementare Graphoperationen beschränkt, wie etwa das Einfügen oder Löschen einzelner Knoten oder Kanten. Diese Einschränkung rührt daher, dass der Fokus von EMF auf der strukturellen Modellierung liegt und bisher noch keine Unterstützung zur Codegenerierung aus Verhaltensmodellen besteht. Dies bedeutet, dass bei der herkömmlichen Verwendung von EMF die Methodenrümpfe immer noch manuell ausprogrammiert werden müssen. Diese Lücke wird durch die Verwendung von Fujaba geschlossen.

Die Kombination beider Ansätze ermöglicht so die modellgetriebene Entwicklung von interaktiven Systemen. Die modellgetriebene Entwicklung umfasst hierbei sowohl die Benutzerschnittstelle als auch die Applikationslogik, was eine signifikante Reduzierung der benötigten Entwicklungszeit zur Folge hat. Abbildung 4.6 verdeutlich das Zusammenspiel der einzelnen Werkzeuge [BDG07]. Die Erzeugung von EMF-kompatiblen Java Quelltext kann in zwei voneinander getrennte Teile zerlegt werden, die in Codeerzeugung für strukturelle Modelle und für Verhaltensmodelle klassifiziert werden. Der erste Teil bildet die strukturellen Elemente des Fujaba Modells (wie etwa Klassen und Assoziationen) auf die entsprechenden Bestandteile des Ecore Modells ab. Im zweiten Teil wird die Verhaltensspezifikation (die Storydiagramme) direkt in Java Quelltext übersetzt. Das Resultat ist zunächst eine unvollständige Java Klasse, die nur die implementierten Methoden enthält. Die Klassendeklaration, sowie die Attribute und Assoziationen der Klasse werden in einem zweiten Schritt mit Hilfe von EMF aus den Informationen des Ecore Modells generiert und mit den bereits vorhandenen Methodenrümpfen gemischt.

Eine Erweiterung der Fujaba Codegenerierung [GSR05] erlaubt die Erzeugung einer XMI Datei, die das Ecore Modell repräsentiert [BDG07]. Dabei werden die Fujaba Klassen auf EMF Klassen abgebildet. Zusätzliche Anpassungen an den Templates zur Co-

[7] http://www.openarchitectureware.org
[8] http://www.eclipse.org/gef/

deerzeugung erlauben die Generierung der Methodenrümpfe aus den Storydiagrammen. Hier müssen allerdings die EMF-spezifischen Methoden zum Erzeugen und Löschen von Objekten anstatt der üblichen Fujaba-spezifischen Methoden verwendet werden. Bei der Verwendung von EMF erfolgt beispielsweise die Objekterzeugung mit Hilfe des Factory-Musters [GHJV94], anstelle des **new**-Operators. Bei der Abfrage und Manipulation von Links sind ebenfalls Anpassungen hinsichtlich der von EMF benutzten Zugriffsmethoden notwendig.

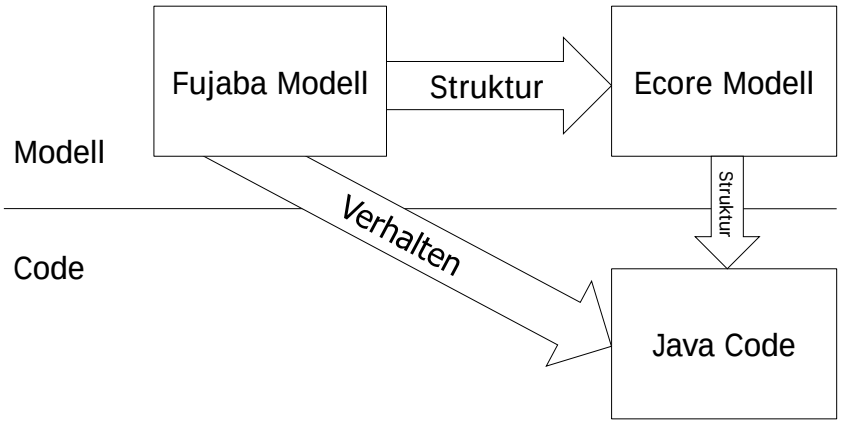

Abbildung 4.6: Zusammenhang zwischen Fujaba und EMF Modellen bei der Generierung von EMF kompatiblen Code aus Fujaba.

Durch die unterschiedliche Ausdrucksmächtigkeit der verwendeten Modelle sind jedoch einige Einschränkungen bei der Modellierung mit Fujaba zu beachten. Tabelle 4.1 vergleicht die Sprachen Fujaba und Ecore.

Dies bedeutet für die Modelltransformation von Fujaba zu EMF, dass einige Eigenschaften, die im Fujaba Metamodell ausgedrückt werden können, verloren gehen. Anhand der Tabelle ist zu erkennen, dass von den betrachteten Modellen, Fujaba klar das ausdrucksmächtigere Modell ist. Tabelleneinträge, die mit fettgedruckter Schrift dargestellt sind, symbolisieren Eigenschaften des Fujaba Metamodells, die bei der Transformation in ein Ecore Modell verloren gehen. Anhand der Tabelle wird ebenfalls die Notwendigkeit des zweigeteilten Ansatzes zur Codegenerierung ersichtlich. Da EMF keine Unterstützung für Codeerzeugung aus Verhaltensmodellen bietet, muss dieser Teil des Fujaba Modells mittels der Modell-zu-Text Transformation aus Fujaba heraus erfolgen. Die fehlenden Teile der Beschreibung des statischen Modells erfolgt durch die Codegenerierung von EMF. Eine Alternative wäre die vollständige Generierung von EMF-kompatiblen Java Code direkt aus Fujaba heraus. Diese Herangehensweise hätte aber den Nachteil, dass Änderungen an der EMF-Codegenerierung analog auch in der Fujaba Codeerzeugung durchgeführt werden müssen. Der Wartungsaufwand wäre also ungleich höher als im gewählten Ansatz.

Abschnitt 5.5.1 zeigt am Beispiel des Paketdiagramm Werkzeugs, wie ein Fujaba Modell mit Hilfe der EMF-Codeerzeugung als Grundlage für die Entwicklung eines graphi-

Eigenschaft	Fujaba 5	Ecore 2.3.1
Klassen		
abstrakt	ja	ja
Interface	ja	ja
Referenz	ja	ja
Attribute		
nur-lesen	nein	ja
final	ja	nein
static	ja	nein
transient	ja	ja
Standardwert	ja	ja
Methoden		
public	ja	ja
protected	ja	nein
private	ja	nein
static	ja	nein
Ausnahmen	ja	ja
Assoziationen		
1:1	ja	ja
1:n	ja	ja
n:n	ja	ja
Unidirektional	ja	ja
geordnet	ja	ja
int. qualifiziert	ja	ja
ext. qualifiziert	ja	nein
Verhalten		
Verhaltensmodell	ja	nein

Tabelle 4.1: Vergleich des Fujaba- und Ecore-Metamodells.

schen Editors mit GMF verwendet werden kann. Abschnitt 5.5.2 beschreibt anschließend ausführlich alle notwendigen Schritte, um einen graphischen Editor generativ zu erstellen.

4.4 Diskussion

Fujaba stellt ein mächtiges Werkzeug zur modellgetriebenen Softwareentwicklung dar. Die Möglichkeit, ausführbare Modelle auf einer Abstraktionsebene über dem Quelltext zu erzeugen, stellt einen erheblichen Mehrwert gegenüber anderen UML Werkzeugen dar. Dennoch erfordert speziell die Entwicklung von graphischen Benutzeroberflächen für die erstellten Modelle oftmals einen erheblichen Aufwand an manueller Implementierung.

Das Graphical Modeling Framework (GMF) von Eclipse versucht dem entgegenzuwir-

ken, indem es dem Benutzer einen generativen Ansatz zur Modellierung von graphischen Editoren anbietet. Die Generierung von Editoren ist dabei aber nur für Domänenmodelle möglich, die mit Ecore beschrieben wurden. Liegt kein Ecore Modell vor, oder kann das bestehende Modell nicht in ein Ecore kompatibles Modell transformiert werden, so hat der Benutzer die Möglichkeit, das Graphical Editing Framework (GEF), ein objektorientiertes Rahmenwerk, zu benutzen. Allerdings muss in diesem Fall der graphische Editor komplett manuell implementiert werden.

Bei der Modellierung eines graphischen Werkzeugs mit GMF werden unterschiedliche Modelle benötigt, die jeweils durch eine eigene DSL mit zugehörigem Baumeditor beschrieben sind. Im Rahmenwerk existiert derzeit noch kein umfassendes Werkzeug, das die Spezifikation aller für die Editorgenerierung notwendigen Modelle in einem einzigen Schritt erlaubt.

Mit GMF ist es zwar möglich, sehr schnell einfache graphische Editoren mit einfachen geometrischen Formen zur Darstellung von Knoten und Kanten zu erzeugen, sind allerdings etwas kompliziertere Darstellungsarten notwendig, so ist auch hier ein nicht unerheblicher Aufwand von Nöten. In vielen Fällen müssen die zur Darstellung verwendeten Elemente manuell implementiert und in der graphischen Definition lediglich referenziert werden. Das gleiche gilt für Editieroperationen. GMF unterstützt nur einfache Operationen wie Erzeugen, Löschen oder Umbenennen von Knoten und Kanten. Sind an einer komplexen Transaktion im Domänenmodell (die mittels eines Storydiagramms beschrieben ist und für die ausführbarer Quelltext erzeugt wurde) jedoch mehrere Knoten und Kanten beteiligt, so muss der Aufruf dieser Operation beispielsweise durch einen Kontexmenüaufruf manuell implementiert werden. GMF bietet aktuell keine Möglichkeit, Operationen, die über die Werkzeugpalette aufgerufen werden, frei zu definieren, da an dieser Stelle immer die von EMF generierten Methoden zur Erzeugung neuer Objekte verwendet werden.

4.5 Zusammenfassung

In diesem Kapitel wurden domänenspezifische Sprachen als Mittel zur Erhöhung des Abstraktionsgrades bei der Entwicklung von Softwaresystemen vorgestellt. Durch die hohe Spezialisierung auf eine Anwendungsdomäne beschränkt sich der Sprachumfang auf Begriffe der Domäne, was eine natürlichere Modellierung des eigentlichen Problems erlaubt. Gerade auch im Bereich der Softwareproduktlinienentwicklung finden generative Herangehensweisen, wie etwa durch die Benutzung von DSLs, immer mehr Anklang. Da eine Produktlinie in der Regel für eine bestimmte Domäne erstellt wird, und das entwickelnde Softwareunternehmen meist über weitreichendes Domänenwissen verfügt, liegt die Entwicklung von domänenspezifischen Sprachen mit geeigneten Generatoren nahe.

Auf Basis der Eclipse Plattform existieren etliche Rahmenwerke, die zu diesem Zwecke eingesetzt werden können. Sie alle sind unter dem *Eclipse Modeling Project* zusammen-

gefasst. Die Werkzeuge aus dem Eclipse Modeling Project erlauben den Entwicklern von domänenspezifischen Sprachen die modellbasierte Entwicklung der abstrakten und konkreten Syntax, sowie unterschiedliche Arten von Modelltransformationen (Modell-zu-Modell oder Modell-zu-Text).

Da das Eclipse Modeling Project eine Codeerzeugung aus Verhaltensmodellen nicht unterstützt, besteht an dieser Stelle noch eine Lücke in dem vollständig modellgetriebenen Entwicklungsansatz. Diese Lücke kann durch die Verwendung von Fujaba und der für EMF angepassten Codeerzeugung geschlossen werden. Somit ist es möglich, Werkzeuge mit graphischen Editoren in kurzer Zeit modellgetrieben zu entwickeln.

5 Architekturmodellierung

5.1 Modellieren im Großen

Im Lebenszyklus eines Softwaresystems erfolgt der Übergang vom Problemraum, der in der Anforderungsanalyse erfasst wurde, zum Lösungsraum - der Implementierung des Softwaresystems - in zwei Schritten: (1) Programmieren im Großen und (2) Programmieren im Kleinen. Am Ende der Disziplin *Programmieren im Großen* steht der Produktentwurf, d.h. die Softwarearchitektur und die Spezifikation der Softwarekomponenten. Eine Definition für Programmieren im Großen lautet etwa:

> „Alle Aktivitäten oberhalb der Realisierung einzelner Module, insbesondere die Definition und Modifikation der Gesamtstruktur (Gesamtarchitektur) eines Softwaresystems entsprechend der Anforderungsspezifikation“ [Nag90].

Zu dem Begriff „Architektur“ findet man in einschlägiger Literatur zu UML folgende Definition:

> „Eine Architektur ist die Menge signifikanter Entscheidungen über die Organisation eines Softwaresystems, die Auswahl struktureller Elemente und deren Schnittstellen, durch die das System aufgebaut wird, zusammen mit deren Verhalten, das durch die Zusammenarbeit dieser Elemente spezifiziert wird, die Zusammensetzung dieser Elemente in nach und nach größer werdende Subsysteme, und dem Architekturstil, der diese Organisation prägt“ [RJB99].

Überträgt man diese Praktiken nun auf die modellgetriebene Entwicklung eines Softwaresystems, so bleibt die Aufgabe, eine geeignete Notation zur Darstellung zu finden. Auf den ersten Blick findet man in der UML Paketdiagramme, die sich für diese Disziplin anbieten. Sie sollen auf grobgranularer Ebene einen Überblick über das Softwaresystem bieten und die Definition von Abhängigkeiten zwischen den funktionalen Einheiten mittels Importen erlauben. Unsere Erfahrungen mit größeren Projekten in den letzten Jahren, speziell aber mit der Entwicklung der modellgetriebenen Produktlinie für Softwarekonfigurationsverwaltungssysteme, zeigen uns aber, dass dies nicht uneingeschränkt zutrifft [BDFW08], [BDW08a].

Im weiteren Verlauf dieses Kapitels werden zunächst UML Paketdiagramme und ihre Verwendung in der Disziplin Modellieren im Großen diskutiert. Anschließend wird der im Rahmen dieser Arbeit erstellte Paketdiagrammeditor sowohl auf konzeptioneller, als auch auf technischer Ebene ausführlich erklärt. Nach einer Einführung in die Arbeit mit dem Werkzeug wird ein umfassender Vergleich mit existierenden Werkzeugen durchgeführt.

5.2 Paketdiagramme in UML

Wie bereits in Abschnitt 3.3.2 erwähnt, gehören Paketdiagramme in der UML zu den sog. Strukturdiagrammen. Mit Hilfe von Strukturdiagrammen werden statische Komponenten eines Softwaresystems beschrieben. Für nichttriviale Problemstellungen können einzelne Diagramme, etwa das Klassendiagramm, sehr schnell unübersichtlich werden. Durch „Paketierung“ von UML-Modellelementen, wie z. B. Klassen, Datentypen, Aufzählungstypen usw., wird ein Strukturierungsmechanismus angeboten und es werden einige Konsistenzbedingungen definiert. Beispielsweise definiert ein Paket einen Namensraum, dem alle in dem Paket enthaltenen Modellelemente angehören. Namen von Modellelementen müssen eindeutig innerhalb des Pakets sein. Folglich können Elemente mit identischen Namen nur existieren, wenn sie in unterschiedlichen Paketen liegen. Diese Elemente werden im System global eindeutig über den sog. *qualifizierten Namen* identifiziert. Diesen erhält man durch Voranstellen der Namen aller übergeordneten Pakete eines Elements, beginnend mit dem äußersten Paket und endend mit dem Namen des Elements selbst. Der Zugriff auf Elemente erfolgt innerhalb eines Pakets ohne weiteres über deren Namen. Auch Elemente in übergeordneten Paketen können im untergeordneten Paket über den Namen referenziert werden, sofern im untergeordneten Paket nicht ein Element mit gleichen Namen vorhanden ist und das übergeordnete Element somit verdeckt. Will man auf Elemente aus anderen Paketen zugreifen, so kann man entweder deren qualifizierten Namen verwenden, oder man importiert diese Elemente, wodurch die entsprechenden Namen dem Namensraum des importierenden Pakets hinzugefügt werden. Durch Verwendung von Importen sind diese importierten Elemente dann einfach über ihren Namen (ohne Qualifizierung) referenzierbar. Den Vorteil für den Modellierer, von überall im Modell über den qualifizierten Namen auf beliebige Elemente zugreifen zu können, erkauft man sich durch einen erhöhten Übersichtlichkeitsverlust. Während Importe im Paketdiagramm durch Pfeile zwischen den beiden am Import beteiligten Elementen visualisiert wird, existiert keinerlei graphische Darstellung für die Verwendung von Zugriffen auf Elemente über den qualifizierten Namen. Wird ein Softwaresystem entwickelt, bei dem der Erfolg massiv davon abhängt, dass Kopplungen zwischen den einzelnen Einheiten minimal gehalten werden müssen, wie es z. B. bei der Entwicklung von Softwareproduktlinien[1] der Fall ist, ist eine Möglichkeit zur Verwaltung dieser Abhängigkeiten unabdingbar. Im schlimmsten Fall könnte sogar ein Softwaresystem in UML modelliert werden, bei dem der Zugriff auf Elemente ausschließlich über den qualifizierten Namen erfolgt und somit auf der Ebene der Paketdiagramme keinerlei Importbeziehungen vorhanden sind. Ein Überblick über Abhängigkeiten in der Architektur ist somit überhaupt nicht möglich.

Weiterhin werden Paketdiagramme in existierenden CASE Werkzeugen zumeist recht stiefmütterlich behandelt. Manche Werkzeuge, wie Fujaba, bieten auch überhaupt keine

[1] Im Fall der hier als Anwendungsbeispiel verwendeten Produktlinie für Softwarekonfigurationsverwaltungssysteme ist dieser Punkt sogar essentiell, da die verschiedenen Module innerhalb der Produktlinie möglichst orthogonal kombiniert werden sollen, um das resultierende System möglichst flexibel an die gewünschten Anforderungen anpassen zu können.

Unterstützung für Paketdiagramme an. Sofern Paketdiagramme in den Werkzeugen unterstützt werden, beschränken diese sich zumeist auf die Unterstützung von öffentlichen Paketimporten. Private Importe oder Elementimporte werden zumeist nicht angeboten, da diese Werkzeuge die definierten Sichtbarkeiten im weiteren Modellierungsprozess entweder nicht berücksichtigen, oder implizit über den qualifizierten Namen auf nicht sichtbare Elemente zugreifen.

Aufgrund dessen wurde ein Paketdiagrammeditor entwickelt, der weniger liberal als gängige Werkzeuge ist und infolge dessen den Zugriff auf Elemente über den qualifizierten Namen nicht unterstützt. Im weiteren Verlauf des Kapitels wird dieser neu entwickelte Paketdiagrammeditor und seine Integration in Fujaba und EMF näher beleuchtet.

5.3 Pakete und Abhängigkeiten im Modellierungsprozess

Im folgenden Abschnitt wird die Notwendigkeit eines Paketdiagrammeditors, der restriktiver in Bezug auf die Zugriffsmöglichkeiten von Elementen untereinander ist, als es die UML Spezifikation erlaubt, vorgestellt. Ausgehend vom eingesetzten Modellierungswerkzeug Fujaba wird die Notwendigkeit eines Paketdiagrammeditors mit dedizierter Unterstützung für die Anforderungen bei der Disziplin *Modellieren-im-Großen* motiviert. Anschließend wird die modellgetriebene Entwicklung des neuen Werkzeugs ausführlich beleuchtet und die Integration in den modellgetriebenen Produktlinien Prozess beschrieben. Ein Vergleich mit bestehenden Werkzeugen schließt das Kapitel ab.

5.3.1 Abhängigkeiten zwischen den Modulen

Gerade in großen Softwaresystemen ist es unerlässlich, die einzelnen Module und deren Abhängigkeiten untereinander zu verwalten, vor allem dann, wenn eine Software-Produktlinie entstehen soll. Sich gegenseitig ausschließende Features dürfen demnach keine Abhängigkeiten untereinander besitzen, da in einer Konfiguration jeweils nur ein einziges Feature gewählt werden kann. Aber auch voneinander unabhängige Features, wie z. B. für die Module zur Delta- bzw. Versionsspeicherung sollen möglichst wenige gegenseitige Abhängigkeiten aufweisen. Wie bereits in Abschnitt 3.3.2 beschrieben, können Paketdiagramme für diese Aufgabe verwendet werden. Zur Zeit existiert weder in Fujaba noch in EMF bzw. auf EMF aufbauenden Modellierungswerkzeugen wie *Topcased* ein Paketdiagrammeditor, der sowohl die unterschiedlichen, in der UML definierten, Importbeziehungen zulässt als auch eine Validierung für auf dem Paketmodell aufbauende Klassendiagramme unterstützt [BDW08a]. Auch kommerzielle UML Werkzeuge, wie etwa Rational Rose[2] oder Rational Software Architect[3] bieten derartige Funktionen leider nicht an. Dabei ist es gerade für den Erfolg bei der Entwicklung von Software-Produktlinien mit lose gekoppelten Modulen essentiell, Überblick und Kontrolle über die

[2] http://www.ibm.com/software/awdtools/developer/rose/
[3] http://www.ibm.com/software/awdtools/architect/swarchitect/

Abhängigkeiten der einzelnen funktionalen Einheiten zu besitzen.

Diese Erkenntnisse und die Tatsache, dass UML Paketdiagramme bisher in allen gängigen Werkzeugen mehr oder weniger vernachlässigt wurden, deutet darauf hin, dass Pakete in der UML in erster Linie zur Definition von Namensräumen und nicht zur Architekturmodellierung genutzt werden.

5.3.2 Reverse Engineering des Java Codes

Da Fujaba wie oben beschrieben keinen Paketdiagrammeditor besitzt, mit dessen Hilfe Paketimporte erstellt, oder auch Klassen den einzelnen Paketen zugeordnet werden können, war der erste Ansatz eine Analyse der Importe auf Basis des erzeugten Java Codes. Hierfür wurde das Fujaba Modell mit Hilfe der Fujaba Codegenerierung nach Java transformiert. Der resultierende Quellcode wurde mit dem Werkzeug eUML2 analysiert. eUML2 bietet ein Reverse Engineering von Java Code zu UML Diagrammen. Abbildung 5.1 zeigt das Ergebnis des Reverse Engineering des Java Quellcodes. Hierbei ist zu beachten, dass das Diagramm Java-Pakete und deren Abhängigkeiten über Importe darstellt und nicht UML-Pakete und deren Importe. Es werden Importe für Vererbungsbeziehungen, Interface-Realisierungen und Methodenaufrufe angezeigt. Das angezeigte Ergebnis wirft zunächst Zweifel auf, ob die Vorgabe *modulares Design und Austauschbarkeit der Module* noch gegeben ist. Zu dicht erscheint der Graph der Importbeziehungen.

Importe in Java sind von einer anderen Granularität als Importe in UML2 Paketdiagrammen. Auf der Ebene des Quellcodes werden nahezu ausschließlich Importe verwendet, die in etwa den privaten Elementimporten aus der UML entsprechen. Abbildung 5.2 zeigt einen Ausschnitt aus dem Fujaba Modell der Produktlinie. Die Assoziation zwischen **IVersionedItem** aus dem Paket **de.ubt.ai1.mod2scm.core.versioned_items** und **IVersion** aus dem Paket **de.ubt.ai1.mod2scm.core.versions** erfordert auf konzeptioneller Ebene die gegenseitige Sichtbarkeit beider Klassen. Da die Assoziation bidirektional ist und die Navigierbarkeit nicht eingeschränkt ist, kann man von **IVersionedItem** durch Traversieren der Assoziation **IVersion** erreichen und umgekehrt. Auf der Ebene der Paketdiagramme lässt sich dies auf vier verschiedene Arten ausdrücken:

1. durch einen öffentlichen Paketimport von **de.ubt.ai1.mod2scm.core.versions** im Paket **de.ubt.ai1.mod2scm.core.versioned_items** (und umgekehrt)
2. durch einen privaten Paketimportvon **de.ubt.ai1.mod2scm.core.versions** im Paket **de.ubt.ai1.mod2scm.core.versioned_items** (und umgekehrt)
3. durch einen öffentlichen Elementimport von **IVersion** in **IVersionedItem** (und umgekehrt)
4. durch einen privaten Elementimport von **IVersion** in **IVersionedItem** (und umgekehrt)

Importe in Java und in UML unterscheiden sich deutlich voneinander. Während der Paket- und Importbegriff in UML sehr liberal interpretiert wird und sogar die Mo-

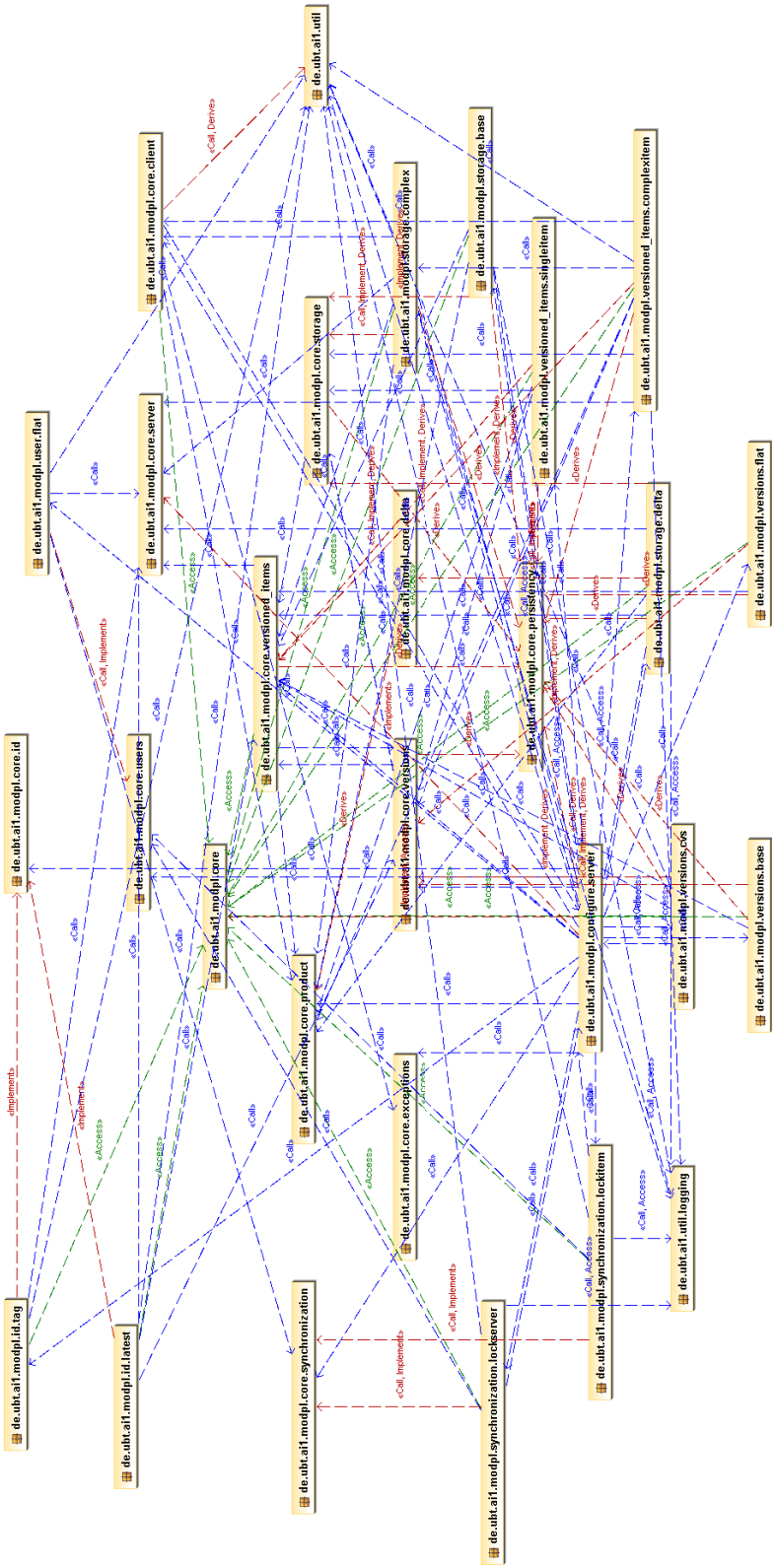

Abbildung 5.1: Paketdiagramm mit Abhängigkeiten als Ergebnis des Reverse Engineerings des erzeugten Java Codes.

difikation von öffentlich importierten Elementen im importierenden Namensraum erlaubt ([OMG09a], S. 143), kennt Java nur private Importe und erlaubt keine Modifikation von importierten Elementen. Durch die Beschränkung auf private Importe ergibt sich im Reverse Engineering ein wesentlich dichterer Graph, als er durch öffentliche Importe ausgedrückt werden kann. Weiterhin handelt es sich bei den Java Importen um Importe auf Implementierungsebene und hängen deshalb sehr stark von der jeweiligen Codeerzeugung ab. Insbesondere bei bidirektionalen Assoziationen führt dies zu Problemen, da auf der Implementierungsebene zyklische Abhängigkeiten entstehen. Auf der konzeptionellen Ebene hingegen kann eine Assoziation in den meisten Fällen sinnvoll *einem* Paket zugeordnet werden (im Beispiel kann die Assoziation dem Paket **de.ubt.ai1.mod2scm.core.versioned_items** zugeordnet werden), was zu einer gerichteten Beziehung im Paketdiagramm führt.

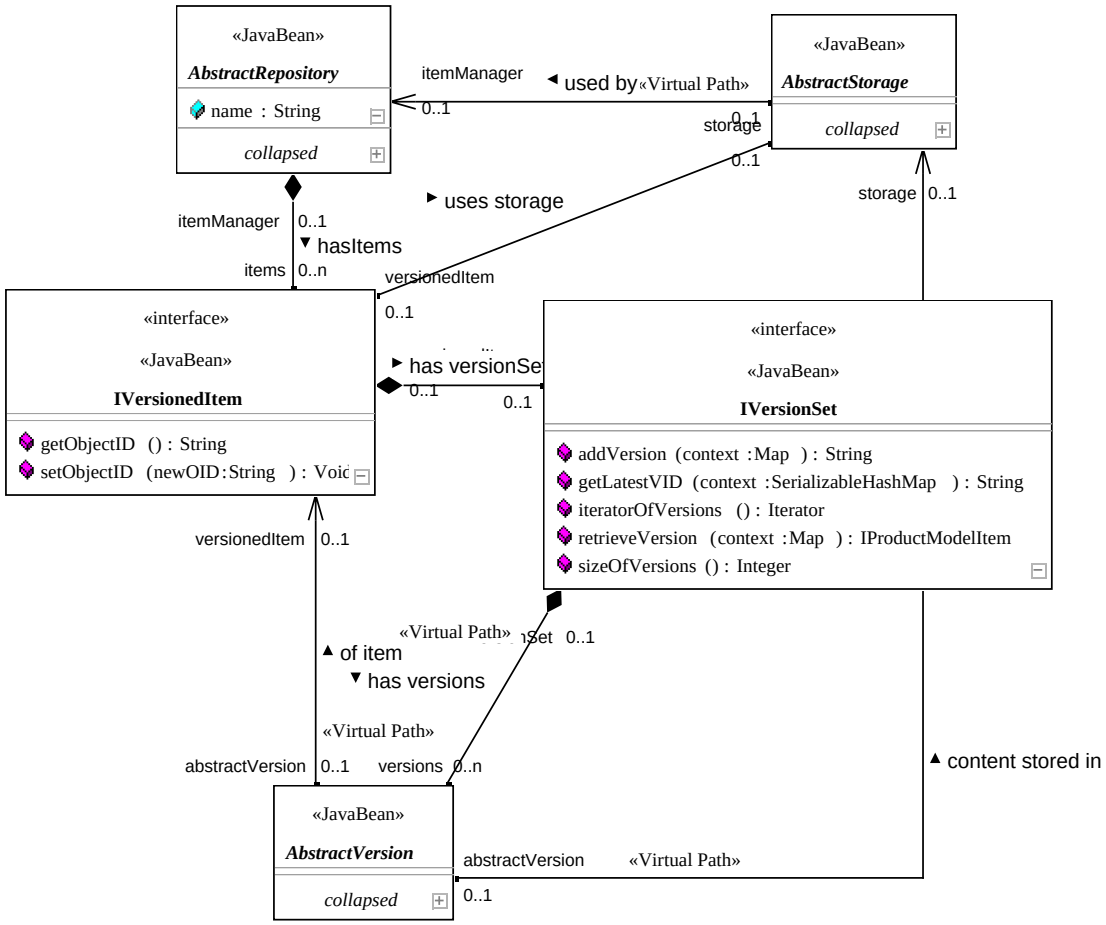

Abbildung 5.2: Ausschnitt aus dem Klassendiagramm des MOD2-SCM Systems.

Die Listings 5.1 und 5.2 zeigen den generierten Java Quelltext für beide Klassen.

Anhand des Java Codes wird ersichtlich, dass in beiden Klassen die jeweils andere Klasse importiert wird.

```
1  package de.ubt.ai1.modpl.core.versioned_items;
2
3  import de.ubt.ai1.modpl.core.versions.IVersionSet;
4  ...
5  public interface IVersionedItem
6  {
7          ...
8     /**
9      * <pre>
10     *           0..1      has versionSet      0..1
11     * IVersionedItem ------------------------- IVersionSet
12     *           versionedItem                 versionSet
13     * </pre>
14     */
15    public static final String PROPERTY_VERSION_SET = "versionSet";
16    public boolean setVersionSet (IVersionSet value);
17    public IVersionSet getVersionSet ();
18    ...
19 }
```

Listing 5.1: Generierter Java Quelltext für **IVersionedItem**.

```
1  package de.ubt.ai1.modpl.core.versions;
2
3  import de.ubt.ai1.modpl.core.versioned_items.IVersionedItem;
4  ...
5  public interface IVersionSet extends PropertyChangeClient
6  {
7          ...
8     /**
9      * <pre>
10     *           0..1      has versionSet      0..1
11     * IVersionSet ------------------------- IVersionedItem
12     *           versionSet                 versionedItem
13     * </pre>
14     */
15    public boolean setVersionedItem (IVersionedItem value);
16    public IVersionedItem getVersionedItem ();
17         ...
18 }
```

Listing 5.2: Generierter Java Quelltext für **IVersionSet**.

Das schwerwiegendste Problem des Reverse Engineering Ansatzes ist aber die Tatsache, dass diese Vorgehensweise nicht dazu genutzt werden kann, um die Architektur vorab zu planen. Es steht lediglich ein Kontrollmechanismus zur Verfügung, um ein bereits implementiertes System a posteriori zu analysieren. Für eine a priori Definition der zu erstellenden Architektur muss demnach eine andere Lösung gefunden werden.

5.4 Paketdiagrammeditor

Aufgrund der gewonnenen Erkenntnisse und der Notwendigkeit, die Abhängigkeiten der einzelnen Module schon während der Modellierung des Softwaresystems zu erkennen, bzw. eine Architektur von vornherein definieren zu können, wurde ein eigener Paketdiagrammeditor entwickelt. Die Anforderungen an den Editor waren:

- Darstellen von Paketen in geschachtelter Form
- Unterstützung aller in der UML definierten Importe
- Verbot der Benutzung von voll qualifizierten Paketnamen, um auf Elemente zuzugreifen
- Möglichkeit der Validierung des Modells anhand der Vorgaben im Paketdiagramm
- Integration in vorhandene CASE Werkzeuge (z. B. Fujaba oder Werkzeuge für EMF)
- Möglichkeit des *forward engineering*, d.h. das Paketdiagramm wird erstellt, bevor die Klassendiagramme erstellt werden.
- Unterstützung von echtem *Roundtrip engineering*, d.h. während des Entwicklungsprozesses ist ein permanenter Abgleich zwischen Paketdiagramm und Klassendiagramm möglich

5.4.1 UML Metamodell

Die für die Realisierung von Paketen und Beziehungen zuständigen Klassen, sind im **Kernel package** der UML 2 Spezifikation [OMG09b] definiert. Das **Kernel package** stellt das Herz der UML dar und die Metaklassen jedes anderen Pakets hängen direkt oder indirekt davon ab. In ihm sind die Kernkonzepte zur Modellierung mit der UML definiert, inklusive Klassen, Assoziationen und Paketen. Das **Kernel package** verwendet die Pakete **Constructs** und **PrimitiveTypes** aus der *InfrastructureLibrary* [OMG09a] und erweitert diese mit zusätzlichen Features, Assoziationen und Superklassen. Das für die Paketdiagramme zugrundeliegende Konzept der Namensräume wird ebenfalls im **Kernel package** definiert (siehe [OMG09b], S. 26). Abbildung 5.3 zeigt das zugehörige Klassendiagramm. Ein Namensraum (**Namespace**) enthält Kindelemente, die über die Assoziationen **+/member** und **+/ownedMember** erreicht werden können. Desweiteren kann ein Namensraum beliebig viele **ElementImport**e und **PackageImport**e besitzen. Die über diese

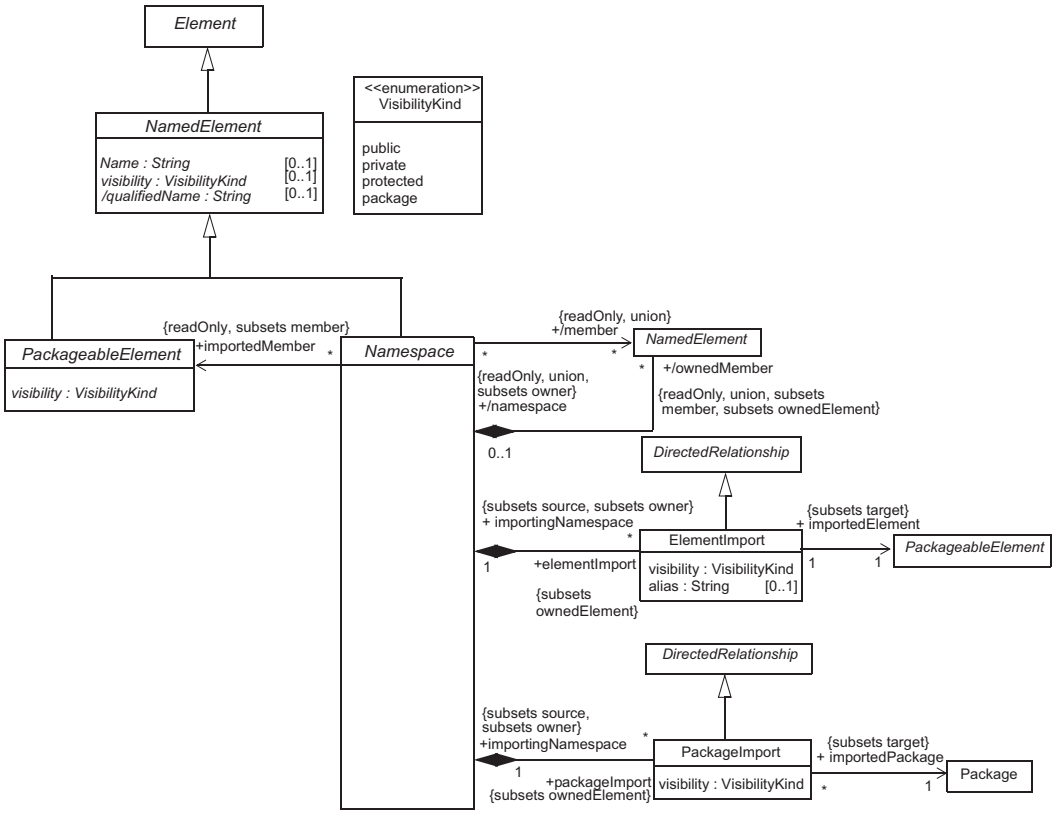

Abbildung 5.3: Das Namespaces Diagramm des **Kernel package** der UML Spezifikation ([OMG09b], S. 26).

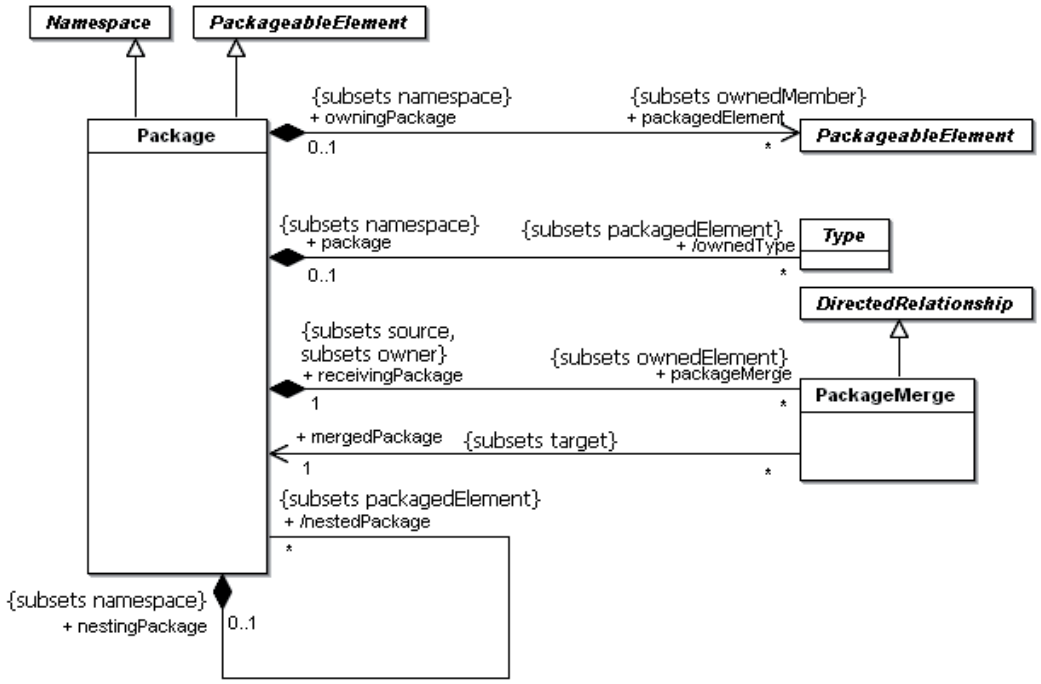

Abbildung 5.4: Das Packagess Diagramm des **Kernel package** der UML Spezifikation ([OMG09b], S. 34).

Importe importierten Elemente vom Typ **PackageableElement** sind über die Assoziation **+importedMember** zu finden.

Das Konzept der Pakete und deren Schachtelung wird in Abbildung 5.4 deutlich. Ein Paket (**Package**) kann zum einen wiederum Pakete beinhalten und so eine geschachtelte Hierarchie darstellen, zum anderen aber können auch beliebige **PackageableElement**s und Typen in dem Paket enthalten sein. Die in Abbildung 5.4 dargestellten Verschmelzungsbeziehungen von Paketen (**PackageMerge**) werden in der UML Spezifikation massiv verwendet. Für ein Werkzeug, das dazu dient eine Architektur zu beschreiben und Abhängigkeiten zwischen einzelnen Paketen darzustellen sind sie jedoch nicht geeignet. Daher wurden sie im Zuge dieser Arbeit nicht weiter betrachtet und werden daher hier auch nicht näher erläutert.

Sichtbarkeitsregeln

Wie oben bereits erwähnt, enthalten Pakete Elemente, die dann in dem vom Paket beschriebenen Namensraum definiert werden. Diesen enthaltenen Elementen werden *Sichtbarkeiten*, ähnlich zu Merkmalen von Klassen, zugeordnet (siehe auch die Aufzählung **VisibilityKind** in Abbildung 5.3). Die Sichtbarkeit eines solchen Elements kann allerdings im Gegensatz zu Sichtbarkeiten von Klassendiagrammelementen nur *privat* (dargestellt durch „- “) oder *öffentlich* (dargestellt durch „+ “) sein. Private Elemente sind nur innerhalb des Pakets, in dem sie definiert sind, und in von diesem Paket eingeschlossenen

Paketen sichtbar, während öffentliche Elemente global sichtbar sind. Die von den Klassen bekannten Sichtbarkeiten *Paket* und *geschützt* machen bei Paketen keinen Sinn, da erstere sowieso immer erfüllt ist und die Sichtbarkeit *geschützt* aufgrund der fehlenden Möglichkeit der Generalisierung von Paketen hinfällig ist. Die UML fordert, dass jedem Element eine Sichtbarkeit zugeordnet werden *muss*. Laut der Spezifikation gibt es dafür keinen Standardwert und auch keinen undefinierten Wert.

Nachfolgend werden nun die Regeln von UML beschrieben, die die Sichtbarkeit eines Elements festlegen. Ein Element eines Pakets hat auf alle Elemente Zugriff, die innerhalb dieses Pakets sichtbar sind. Die sichtbaren Elemente werden durch folgende Sichtbarkeitsregeln bestimmt:

- Elemente, die in einem Paket enthalten sind, sind darin sichtbar.
- In geschachtelten Paketen sind alle Elemente sichtbar, die in den umschließenden Paketen sichtbar sind und nicht durch innere Deklarationen verdeckt werden.
- Über eine Import-Beziehung importierte Elemente sind in dem Paket, das die Import-Beziehung besitzt, sichtbar.
- Bei Import-Beziehungen zu anderen Paketen sind alle öffentlich sichtbaren Elemente des importierten Namensraum im importierenden Paket sichtbar.
- Die öffentliche Import-Beziehung ist im Gegensatz zur privaten transitiv, d.h. ein Paket, das ein weiteres Paket öffentlich importiert, sieht auch alle öffentlichen Importe des importierten Pakets.

Aus diesen Sichtbarkeitsregeln folgt, dass ein Paket nicht auf den Inhalt seiner geschachtelten Pakete zugreifen kann, außer es definiert Import-Beziehungen zu den geschachtelten Paketen. Selbst dann hat es nur Zugriff auf die öffentlich sichtbaren Elemente.

Weiterhin ist der Zugriff auf Elemente von überall her erlaubt, sofern man das Element mit dessen voll qualifizierten Namen referenziert. Da sich dieser Zugriff folglich nicht durch eine Importbeziehung darstellen lässt, wurde bei der Konzeption des Paketdiagrammeditors ausdrücklich auf die Möglichkeit, auf Elemente über deren voll qualifizierten Namen zuzugreifen, verzichtet. Es kann aber durchaus noch erforderlich sein, Elemente zur eindeutigen Unterscheidung mit dem qualifizierten Namen anzusprechen, etwa wenn die importierten Pakete Elemente mit gleichen Namen enthalten, oder wenn ein lokales Element ein importiertes Element gleichen Namens überdeckt. Die Importbeziehung ist aber dennoch im Diagramm vorhanden und erlaubt Aussagen über Abhängigkeiten.

5.4.2 Anforderungen an das neu erstellte Werkzeug

Das Blockdiagramm in Abbildung 5.5 zeigt die Kernfunktionen des neu entwickelten Paketdiagrammeditors im Zusammenspiel mit den bereits vorhandenen Modellierungswerkzeugen für Ecore-Modelle und für Fujaba. Der Editor kann sowohl als eigenständiges

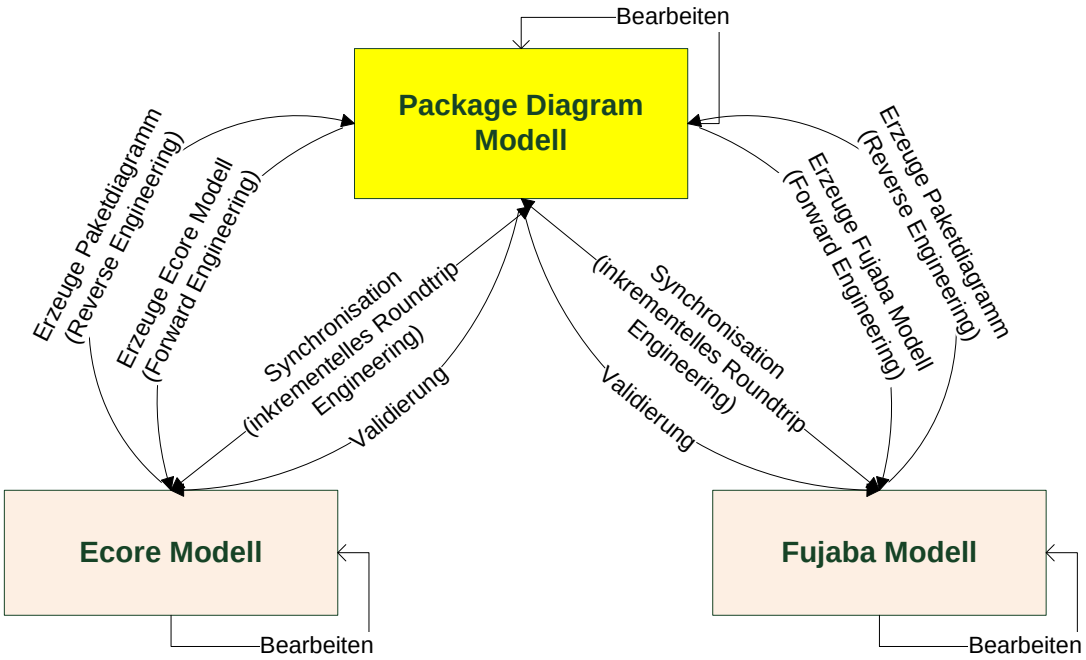

Abbildung 5.5: Der Paketdiagrammeditor und seine Integration in die bestehende Werkzeuglandschaft

Werkzeug zur graphischen Spezifikation und Visualisierung von Architekturen basierend auf Paketen genutzt, als auch im Zusammenspiel mit bereits vorhandenen Modellierungswerkzeugen eingesetzt werden. Während eine Beschränkung auf ein eigenständiges Werkzeug den Modellierungsprozess für die Disziplin *Modellieren im Großen* nur sehr eingeschränkt unterstützen würde, so bietet eine Integration in bestehende Modellierungswerkzeuge die Möglichkeit, einen Abgleich zwischen den Schritten *Modellieren im Großen* und *Modellieren im Kleinen* herzustellen. Hierbei können Konsistenzbedingungen effektiv überwacht und eventuelle Konsistenzverletzungen sofort angezeigt werden.

Der Modellierer erhält hierbei auf vielfältige Art und Weise Unterstützung. Die Integrationsfunktionen lassen sich grob gliedern in:

Forward Engineering Hier wird das Gerüst eines Modells aus einem bereits bestehenden Paketdiagramm generiert. Dies geschieht in einem Stapelverarbeitungsprozess, d. h. das komplette Paketdiagramm wird in einem Schritt verarbeitet.

Reverse Engineering In diesem Fall wird aus einem bereits bestehenden Modell ein Paketdiagramm erzeugt. Dies geschieht ebenfalls in einem Stapelverarbeitungsprozess.

Validierung Ein Modell wird anhand der im zugehörigen Paketdiagramm festgelegten Sichtbarkeiten geprüft. Die Prüfung erfolgt nach Benutzeraufforderung.

Restriktives Bearbeiten Der Editor für das Modell wird hierbei beschränkt, so dass nur

die vom aktuellen Element aus sichtbaren Typen für Operationen zur Verfügung stehen. Werden dennoch nicht sichtbare Typen ausgewählt, so wird das Paketmodell entsprechend angepasst.

Synchronisation Im Gegensatz zu den Stapelverarbeitungsprozessen Forward bzw. Reverse Engineering, erfolgt die Synchronisation zwischen dem Modell und dem Paketdiagramm inkrementell. Änderungen können so in beide Richtungen propagiert werden (*Roundtrip Engineering*).

5.5 Eigenständiges Werkzeug

Dieser Abschnitt befasst sich mit der Implementierung des eigenständigen Werkzeugs mittels Fujaba und GMF. Abbildung 5.6 zeigt die für die Disziplin Modellieren im Großen entwickelten Werkzeuge und deren Integration in die bestehende Werkzeuglandschaft im Überblick.

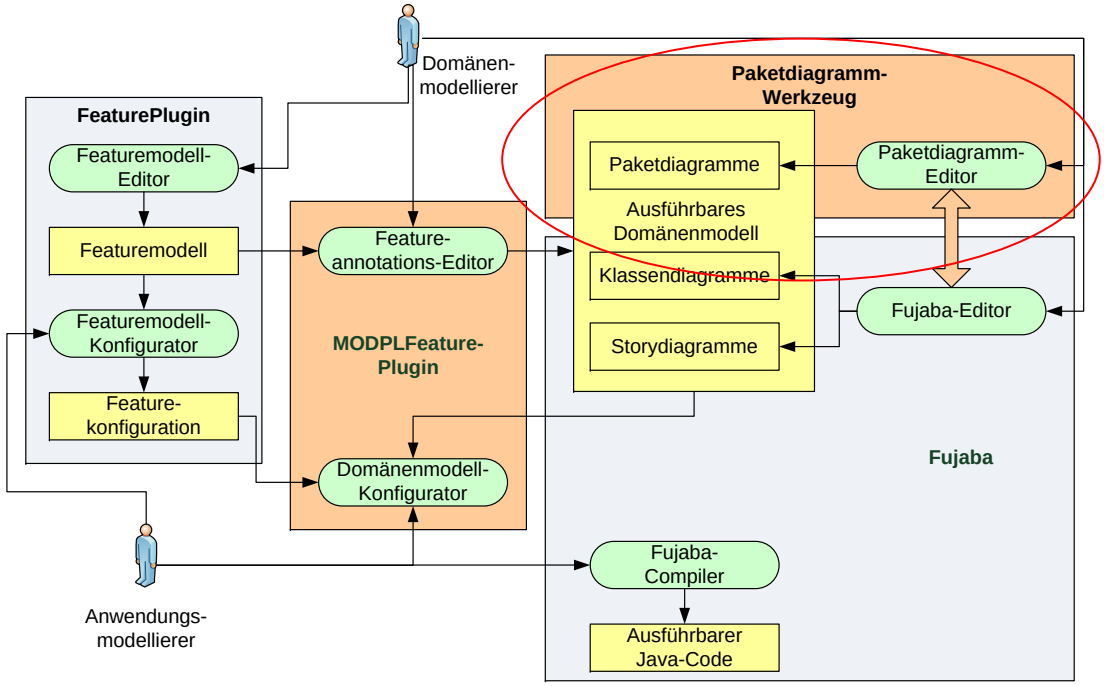

Abbildung 5.6: Integration der Werkzeuge zum Modellieren im Großen.

5.5.1 Fujaba Modell

Statisches Modell

Aufgrund unserer bisherigen Erfahrungen hinsichtlich modellgetriebener Entwicklung von graphischen Werkzeugen [BD06, BDW07, BDFW08], wurde der Paketdiagrammedi-

tor mit Hilfe von Fujaba und GMF [Gro09] erstellt. Auf Basis des im vorigen Abschnitts erläuterten UML Metamodells für Paketdiagramme wurde ein entsprechendes Fujaba Modell erstellt. Im wesentlichen ist das in Fujaba erzeugte Metamodell eine Teilmenge des UML Metamodells. Allerdings ist dieses Modell ein Werkzeugmodell, das werkzeugspezifische Anforderungen erfüllen muss. Da z. B. der graphische Editor für dieses Modell mit GMF (siehe Abschnitt 5.5.2) erstellt wird, folgen daraus spezifische Anforderungen, wie etwa die Verwendung der Klasse **EList** zur Speicherung von Mengen. Abbildung 5.7 zeigt eine Übersicht des zugehörigen in Fujaba erstellten Klassendiagramms. Sämtliche Attribute und Methoden wurden zugunsten einer besseren Lesbarkeit ausgeblendet.

Die Klasse **PackagesDiagram** bildet das Wurzelelement eines jeden Paketdiagramms. Sie erbt von **Diagram**, das seinerseits eine Enthaltenseinsbeziehung für Objekte vom Typ **Element**, der Basisklasse aller in der UML definierten Elemente, enthält. Dieser Klassenentwurf ist der Benutzung des *Graphical Modeling Framework (GMF)* geschuldet, da für die Erstellung eines graphischen Editors mit GMF genau eine Klasse aus dem Domänenmodell die Zeichenfläche repräsentieren muss - hier ist es die Klasse **PackagesDiagram**. Desweiteren enthält diese Klasse Hilfsmethoden (siehe Abbildung 5.8), z. B. für die Berechnung von Sichtbarkeiten zwischen Paketen, die im weiteren Verlauf noch genauer beschrieben werden.

Die Methode **addPackageContent(Package, BasicEList):EList** ist eine rekursive Methode, die ausgehend von einem Paket, alle darin enthaltenen Unterpakete und Typen in einer Liste zurückliefert. Sie wird von der Methode **getAllElements():EList** aufgerufen. Die Methode **getNamespace(String):Namespace** liefert ein Objekt des Typs **Namespace** anhand eines voll qualifizierten Namens zurück. **getPackage(String):Package** tut dies analog für Pakete. Mit Hilfe der Methode **isAccessAllowed(String, String):Boolean** können die Zugriffsberechtigungen von Paketen bzw. Elementen überprüft werden.

Alle Interfaces und Klassen (mit Ausnahme von **Diagram**, **PackagesDiagram** und den Aufzählungen) erben von dem Interface **Element**. Es definiert keinerlei Methoden, sondern stellt lediglich die Basis der Vererbungshierarchie für in Paketdiagrammen vorkommenden Elementen dar. Für die Verwendung eines GMF Editors ist es zwingend erforderlich, dass alle im Editor vorkommenden Objekte direkt von der Wurzelinstanz besessen werden lönnen. Dies ist somit erfüllt. Dass alle im Modell vorhandenen Objekte von dort aus auch erreichbar sind, muss über weitere Kompositionsbeziehungen sichergestellt werden („Enthaltenseins-Hierarchie“).

Das Interface **NamedElement** erbt direkt von **Element** und stellt zusätzlich noch die Attribute *name* und *visibility* bereit. Klassen, die dieses Interface implementieren, besitzen somit einen Namen und eine Sichtbarkeit. Die Sichtbarkeit wirkt sich entsprechend den in Abschnitt 5.4.1 beschriebenen Sichtbarkeitsregeln aus.

PackageableElement ist ebenfalls ein Interface ohne zusätzliche Methoden und Eigenschaften. Objekte dieses Typs können Bestandteile eines Pakets sein.

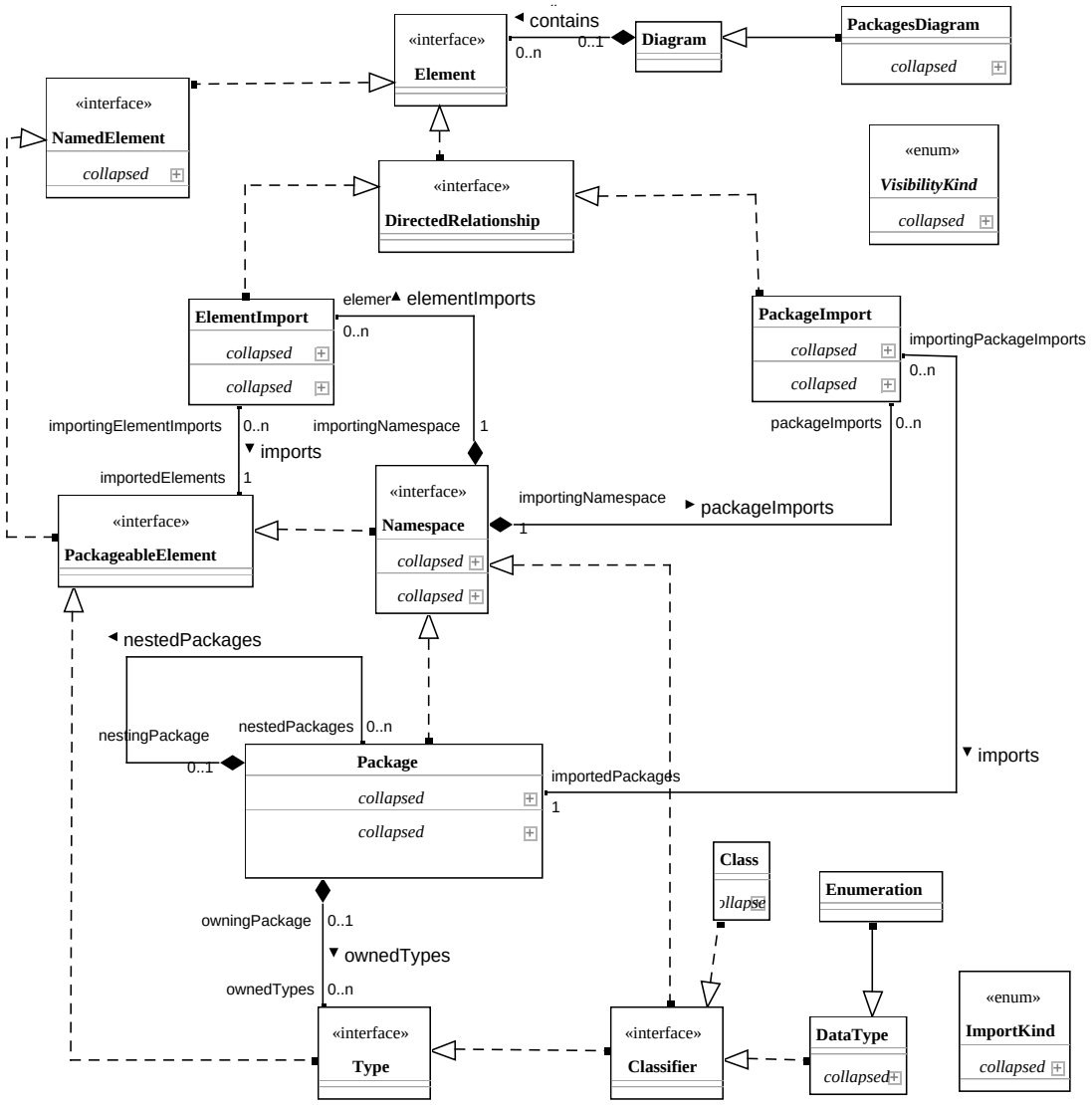

Abbildung 5.7: Klassendiagramm des Paketdiagramm Modells in Fujaba.

PackagesDiagram

- addPackageContent (pack:Package , list:BasicEList) : Voic
- getAllElements () : EList
- getNamespace (qualifiedName:String) : Namespace
- getPackage (qualifiedName:String) : Package
- isAccessAllowed (fromNamespace :String, toNamespace :String) : Boolean

Abbildung 5.8: Die Klasse PackagesDiagram.

Das Interface **Namespace** erbt nun von **PackageableElement**. Die beiden boolschen Attribute **calculatingAccessiblePackages** und **calculatingImports** werden zur Verhinderung von zyklischen Aufrufen bei der Berechnung von Sichtbarkeiten und Importbeziehungen benötigt, auf die im späteren Verlauf noch genauer eingegangen wird. Bei der Ermittlung von importierten Elementen (Typen oder Paketen) kann der Namensraum, von dem die Berechnung ausgeht, selbst wieder erreicht werden, falls zyklische Import-Beziehungen vorliegen. Ist dies der Fall, so terminiert die Berechnung nicht, falls nicht für jeden bereits erreichten Namensraum das boolsche Attribut gesetzt und in der Berechnungsmethode abgeprüft wird, um den Zyklus zu unterbrechen. Weiterhin deklariert das Interface **Namespace** Methoden mit folgender Semantik:

- **getImportedMembers()** liefert eine Liste aller Typen, die in dem aktuellen Namensraum über Importe sichtbar sind.
- **getOwnedMembers()** liefert eine Liste aller Typen, die dieser Namensraum besitzt, d.h. die innerhalb des Namensraums deklariert wurden.
- **getMembers()** liefert eine Vereinigungsmenge aller importierten und vom Namensraum besessenen Typen.
- **getAccessibleMembers()** liefert eine Liste aller Typen, auf die vom aktuellen Namensraum aus zugegriffen werden kann. Die Liste beinhaltet alle vom Namensraum besessenen und importierten Typen, sowie alle sichtbaren Typen aus umschließenden Namensräumen.
- **getAccessiblePackages()** liefert eine Liste aller Pakete, die vom aktuellen Namensraum aus sichtbar sind.

Die Klasse **Package** erbt von **Namespace** und implementiert die oben angesprochenen Methodendeklarationen. Die Klasse Package definiert eine Enthaltenseins-Beziehung durch eine 1:n-Assoziation auf sich selbst. Durch diese Assoziation (Komposition) lässt sich eine Schachtelung der Pakete ausdrücken. Zusätzlich werden noch folgende Methoden definiert:

- **getPackagedElements()** liefert eine Liste der im Paket enthaltenen Elemente und ist somit gleichbedeutend mit der Methode *getOwnedMembers()*.
- **getVisibleMembers()** liefert eine Liste aller öffentlich sichtbaren Typen, die sichtbar sind, sofern das umgebende Paket mit einem öffentlichen Import sichtbar gemacht wird (vgl. Transitivität von öffentlichen Importen, in Abschnit 5.4.1).
- **makesVisible(element : NamedElement)** ist eine Hilfsmethode, mit der öffentliche Typen in der Methode *getVisibleMembers()* überprüft werden. Entsprechend des Ergebnisses des Methodenaufrufs wird ein Element in die Liste, die *getVisibleMembers()* zurückliefert, eingefügt oder nicht.

Das Interface **Type** erbt von **PackageableElement** und deklariert selbst keine Methoden oder Eigenschaften. Objekte von Typ **Type** können Bestandteile von Paketen sein, oder auch auf der oberstene Ebene eines Paketdiagramms liegen.

Classifier ist eine Unterklasse von **Type** und **Namespace** und deklariert ebenfalls keine Methoden oder Eigenschaften.

Die Klasse **Class** erbt von **Classifier** und implementiert die von **Namespace** geerbten Methoden. Im Unterschied zu Package liefert die Methode **getOwnedMembers()** hier aber eine leere Liste, da die *ownedMembers* einer Klasse (z. B. Attribute, Methoden usw.) nicht im Paketdiagramm angezeigt werden sollen.

Die Klasse **DataType** erbt wie **Class** von **Classifier**, allerdings liefern die implementierten Methoden hier immer eine leere Liste, da Datentypen keine anderen Typen referenzieren können.

Die Klasse **ElementImport** erbt von **DirectedRelationship**, das seinerseits von **Element** erbt, aber keine weiteren Methoden oder Eigenschaften definiert. **ElementImport** enthält das Attribut **visibility**, um den Sichtbarkeitstyp (öffentlich oder privat) zu speichern und ein Attribut **visibilityLabel**, um die entsprechende Kennzeichnung im Paketdiagrammeditor darzustellen («**import** » für öffentliche Import und «**access** » für private). Der Sichtbarkeitstyp wird durch den Aufzählungsdatentyp **VisibilityKind** ausgedrückt. **ElementImport**e sind immer Bestandteil genau eines Namensraums (**Namespace**), ausgedrückt durch eine Kompositionsbeziehung, und sie referenzieren immer genau ein Objekt vom Typ **PackageableElement**.

PackageImport erbt ebenfalls von **DirectedRelationship**. Die übrigen Eigenschaften sind analog zu **ElementImport**. Mit Hilfe dieser beiden Klassen werden die unterschiedlichen Importbeziehungen in Paketdiagrammen realisiert.

Dynamisches Modell

Die in diesem Abschnitt vorgestellten Methoden realisieren die Ermittlung von sichtbaren Elementen, berechnen die im Klassendiagramm mit dem Stereotyp «**Virtual Path** » versehenen Assoziationen und stellen Hilfsmethoden zur einfachen Benutzung des Paketdiagramm Modells zur Verfügung. Diese Methoden werden von dem auf dem Modell aufbauenden Editor zur Validierung verwendet. In diesem Falle dient das dynamische Modell zur Berechnung der komplexen Pfadausdrücke (es werden keine Graphtransformationen benutzt). Diese Abfragen werden immer dann benutzt, wenn alle von einem Element aus sichtbaren Elemente berechnet werden, beispielsweise um festzustellen, ob Element A von Element B aus sichtbar ist.

Klasse PackagesDiagram

Das in Abbildung 5.9 dargestellte Storydiagramm zeigt den Einstiegspunkt für die Berechnung der Sichtbarkeiten, realisiert durch die Methode **isAccessAllowed (String fromNamespace, String toNamespace)** in der Klasse **PackagesDiagram**. Die Methodenparameter sind Zeichenketten mit den voll qualifizierten Namen der zu überprüfenden Elemente. Die Methode prüft, ob der Namensraum, der im Parameter **toNamespace** übergeben wird, vom Namensraum **fromNamespace** aus erreicht werden kann. Ecore Basistypen und Elemente außerhalb von Paketen sind überall sichtbar. Eine entsprechende Fallunterscheidung prüft den Zielnamensraum daraufhin ab. Enthalten beide Übergabeparameter Namen für gültige Namensräume, so werden die entsprechenden Objekte mittels der Funktion `getNamespace` ermittelt. Sind beide im Paketdiagramm vorhanden, so werden mit Hilfe dieser Objekte dann vier verschiedene Fälle geprüft:

1. die Quelle ist eine Klasse, das Ziel ist ebenfalls eine Klasse, Datentyp oder eine Aufzählung.
2. die Quelle ist ein Paket, das Ziel Klasse, Datentyp oder Aufzählung.
3. die Quelle ist eine Klasse, das Ziel ein Paket.
4. die Quelle und das Ziel sind Pakete.

Die im Storydiagramm verwendeten Links zwischen den Objekten *from* und *to* sind *virtuelle Pfade*, d.h. sie werden zur Laufzeit mit Hilfe der Methoden **getAccessibleMembers()** und **getAccessiblePackages()** der Klassen **Class** und **Package** berechnet. Da eine Klasse bzw. ein Paket eine reflexive Sichtbarkeit besitzt, werden in den jeweiligen Story Pattern die „**maybe ...** “-Bedingungen benötigt, um die isomorphe Bindung zwischen den Objekten zuzulassen[4].

Die Methode **getNamespace(String)** ermittelt ein beliebiges Objekt des Paketdiagramms anhand des voll qualifizierten Namens und liefert eine Referenz darauf zurück. Abbildung 5.10 zeigt das zugehörige Storydiagramm. Hier wird zuerst der übergebene voll qualifizierte Name an den Punkten aufgetrennt und die einzelnen Fragmente werden in einem String Array abgespeichert. Das erste Element muss auf der obersten Ebene des Paketdiagramms liegen. Von diesem Element ausgehend werden über die Kante *ownedMembers* sämtliche verschachtelten Elemente durchlaufen und deren Namen mit den jeweiligen Einträgen des String Arrays verglichen. Wird schließlich das gesuchte Element

[4]Unter „binden“ versteht man die Wertzuweisung an eine Variable. Beim traversieren von Links werden beiden an der Assoziation beteiligten Objekte Werte zugewiesen. Ein Isomorphismus ist eine bijektive Abbildung. In unserem Falle handelt es sich um eine Abbildung eines Objektes auf eine Variable (Wertzuweisung). Enthalten nun beide an der Assoziation beteiligten Klassen eine Referenz auf dasselbe Objekt, so ist die Abbildung nicht mehr bijektiv. Standardmäßig bricht Fujaba die weitere Bearbeitung ab, sobald der Isomorphismus verletzt ist. Das entsprechende Story Pattern wird dann über die *Failure* Kontrollflusskante verlassen. Die Verwendung der *maybe*-Bedingung schaltet die Isomorphieprüfung für dieses Story Pattern ab.

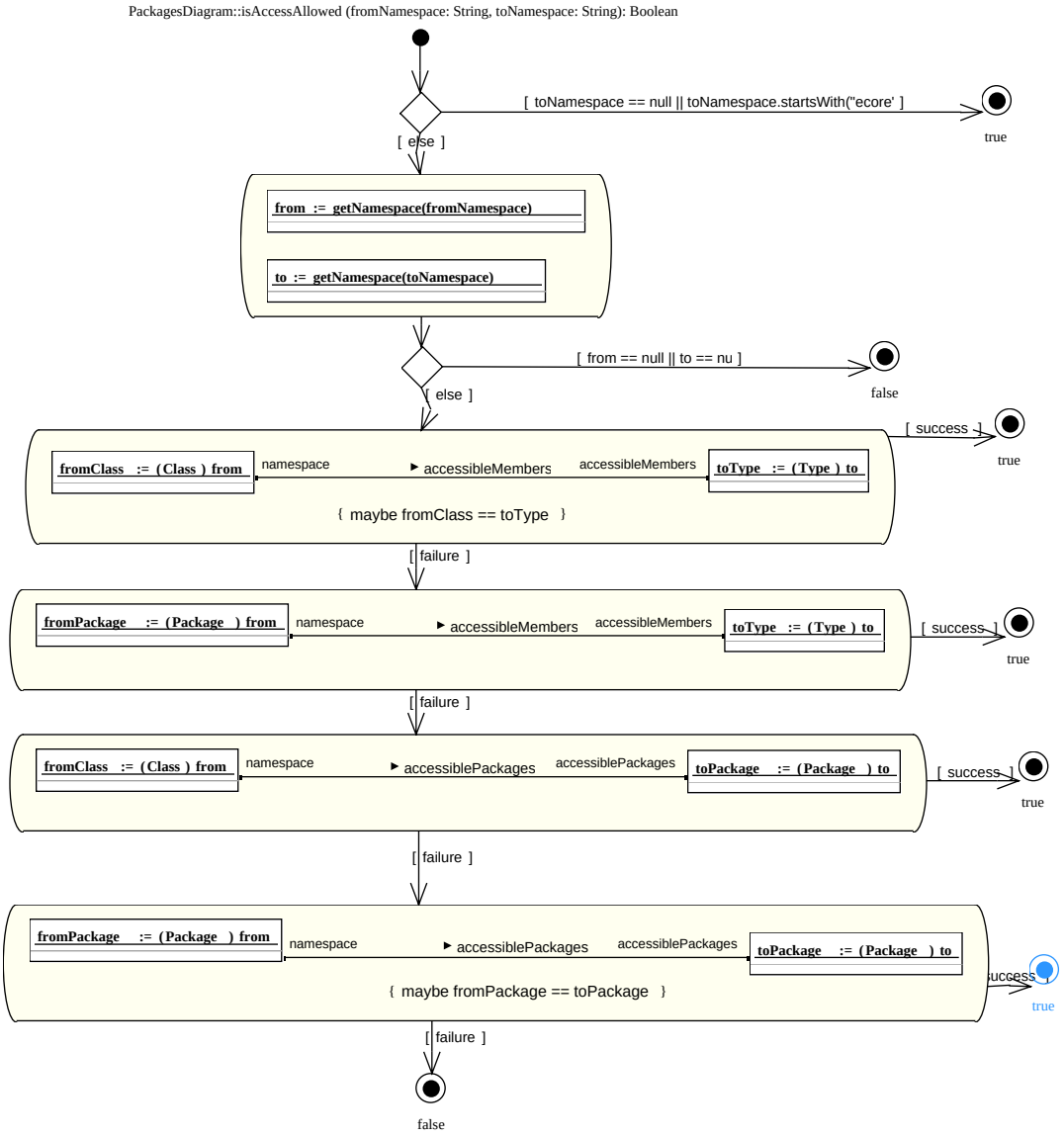

Abbildung 5.9: Das Storydiagramm, das die Semantik der Methode **isAccessAllowed** beschreibt.

gefunden, so wird eine Referenz darauf zurückgegeben.

Die Methoden **getAllElements()** und **addPackageContent** sind Hilfsmethoden, die sämtliche Elemente (Pakete, Klassen, Datentypen und Aufzählungstypen) des aktuellen Modells zurückliefern. Dabei wird in **getAllElements()** über alle Pakete der obersten Ebene iteriert und dann für diese Pakete die rekursive Methode **addPackageContent** aufgerufen, um den kompletten Inhalt des Pakets incl. aller Subpakete zu ermitteln.

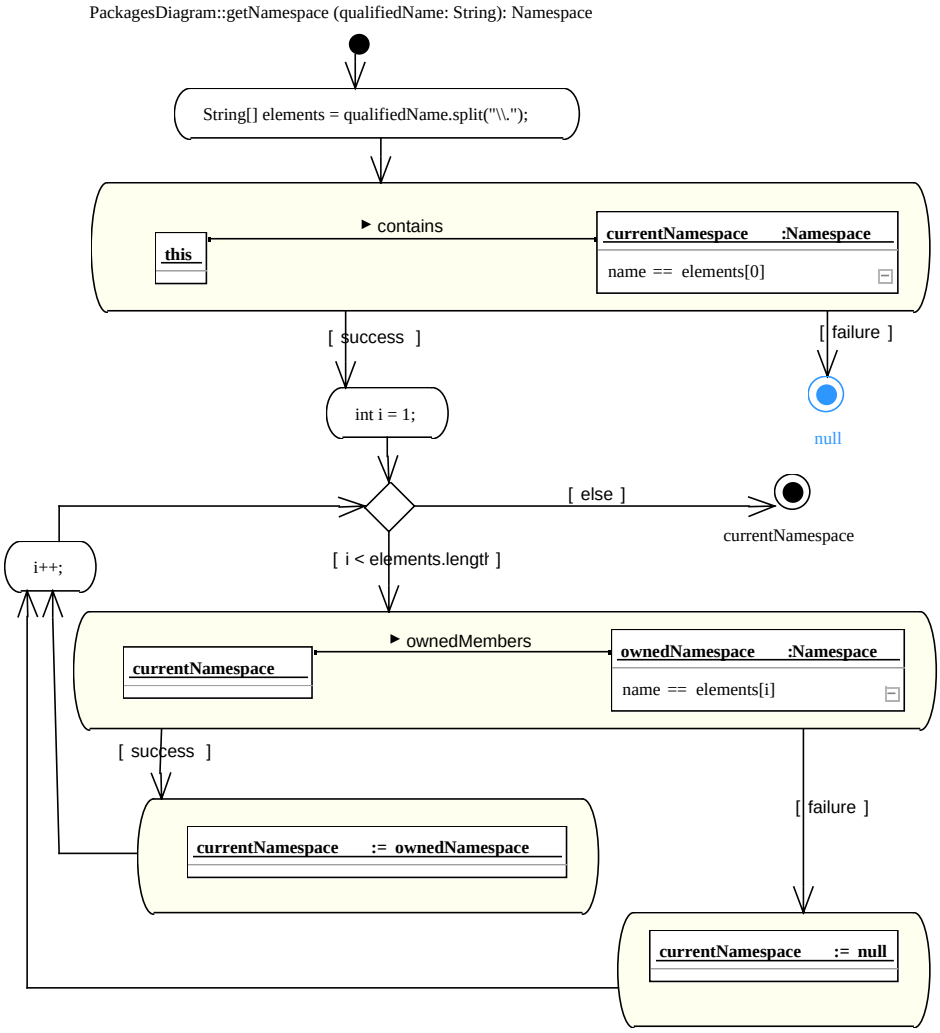

Abbildung 5.10: Implementierung der Methode **getNamespace**.

Klasse Package

Die Methode **getImportedMembers()** der Klasse **Package** ermittelt alle von diesem Paket importierten Pakete und Typen. Hierbei ist zu beachten, dass bei der Berechnung eine Endlosschleife auftreten kann. Als triviales Beispiel sei hier der Fall zweier Pakete genannt, die sich gegenseitig importieren. In der Methode erfolgt die Ermittlung der importierten Typen, indem die sichtbaren Elemente von Paketen betrachtet werden, die über eine Importbeziehung erreichbar sind (siehe Abbildung 5.11). Abschließend werden noch die über Elementimporte erreichbaren öffentlichen Typen dem Ergebnis hinzugefügt. Die Endlosschleife ergibt sich nun dadurch, dass in der Methode getVisibleMembers(), die die sichtbaren Elemente von Paketen ermittelt, wiederum die Methode getMembers () aufgerufen wird, die ihrerseits getImportedMembers() aufruft. Zur Vermeidung dieser Situation wurde die boolsche Variable calculatingImports eingeführt, die als Abbruchkri-

terium dient. Zu Beginn der Berechnung wird überprüft, ob die Variable den Wert false enthält. Ist dies der Fall, so wird der Wert auf true geändert und die Berechnung durchgeführt. Vor Rückgabe des Ergebnisses, wird die Variable wieder zurückgesetzt.

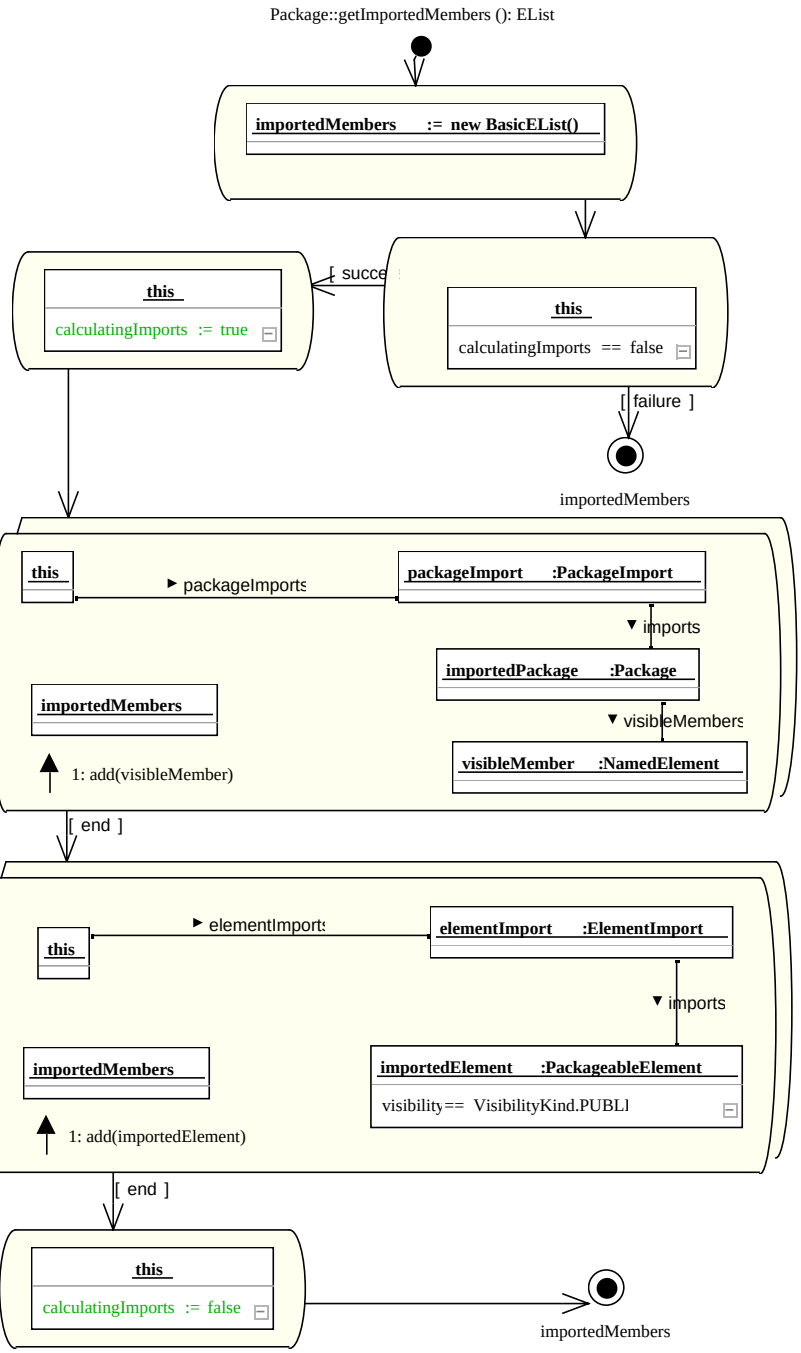

Abbildung 5.11: Implementierung der Methode **getImportedMembers**.

Die Methode **getMembers()** bildet die Vereinigungsmenge der Ergebnisse der Methoden **getOwnedMembers()** und **getImportedMembers()**.

Bei der Berechnung der sichtbaren Elemente, die in der Methode **getVisibleMembers()** erfolgt, besteht wie auch schon bei der Methode zur Ermittlung der importierten Elemente die Gefahr einer Endlosschleife. Auch hier wird ein boolsches Attribut (**calculatingVisibleMembers**) als Abbruchkriterium verwendet. In der Methode werden alle von **getMembers()** ermittelten Typen durchlaufen und mit Hilfe der Methode **makesVisible** auf folgende Kriterien überprüft:

- handelt es sich um einen öffentlichen Typen des eigenen Namensraums
- ist der Typ durch einen öffentlichen Import erreichbar
- handelt es sich um einen sichtbaren Typ eines Pakets, das über einen öffentlichen Paketimport erreichbar ist

Hiermit wird die Transitivität der öffentlichen Paketimporte realisiert. Abbildung 5.12 zeigt das zur Methode gehörende Story Diagramm.

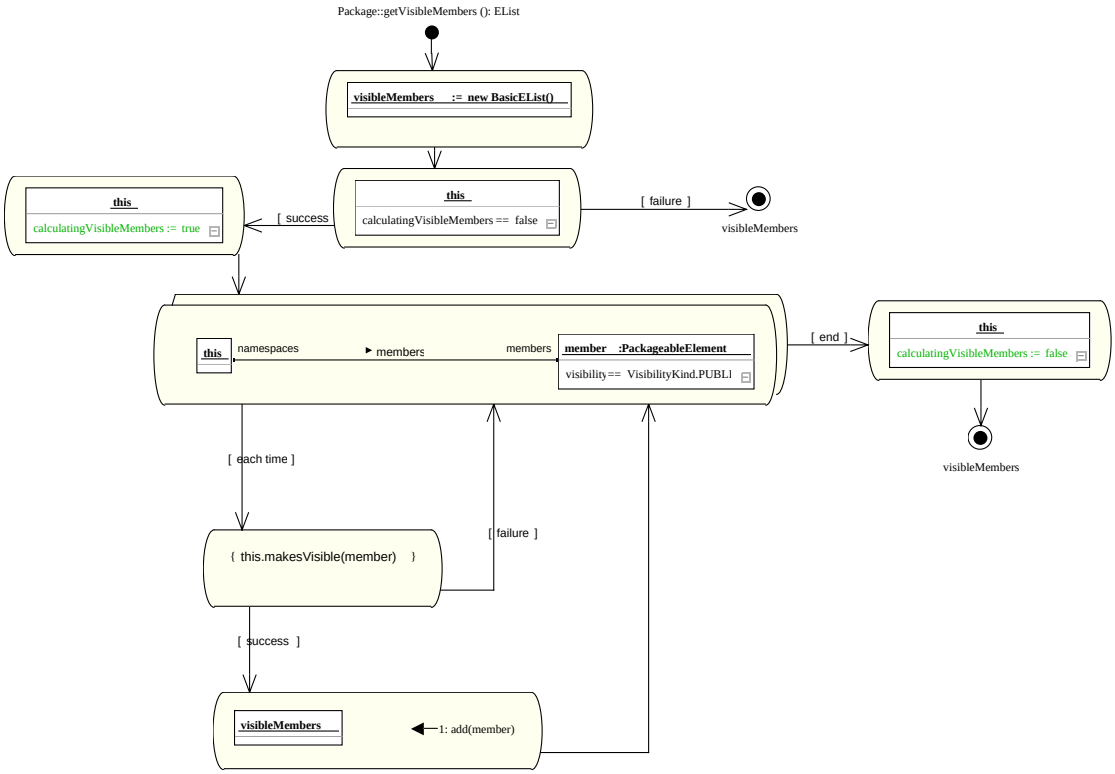

Abbildung 5.12: Implementierung der Methode **getVisibleMembers**.

Die Methode **getVisiblePackages()** hingegen liefert alle sichtbaren Pakete zurück. Dies wird durch iterieren über alle öffentlichen Paketimporte und zurückliefern des Ergebnisses von **getVisiblePackages()** auf den jeweiligen Zielpaketen der Importe erreicht. Auch hier wird eine Endlosschleife durch den Parameter **calculatingVisiblePackages** verhindert.

Die Methode **getAccessibleMembers()** liefert die Vereinigungsmenge des Ergebnisses der Funktion **getMembers()** mit dem Ergebnis des Aufrufs von **getAccessibleMembers()** des umschließenden Pakets, sofern dieses existiert.

getAccessiblePackages() vereinigt die Ergebnismengen der Aufrufe von **getVisiblePackages()** und **getAccessiblePackages()** des umschließenden Paketes mit dem Paket selbst und dessen umschließenden Paket. Eine Endlosschleife wird durch den Parameter **calculatingAccessiblePackages** verhindert.

Klasse Class

Die Methode **getOwnedMembers()** liefert eine leere Liste zurück, da im Paketdiagramm keine Kindelemente von Klassen, wie z. B. Attribute, Methoden usw. angezeigt werden sollen, sondern nur die Klassen selbst.

Der Rückgabewert von **getImportedMembers()** ist die Menge aller über Paketimporte erreichbaren Pakete vereinigt mit der Menge aller über Elementimporte erreichbaren Typen. Diese Menge wird ebenfalls beim Aufruf von **getMembers()** zurückgegeben, da hier die Vereinigung der Ergebnisse von **getOwnedMembers()** und **getImporteMembers()** gebildet wird.

getAccessibleMembers() kombiniert das Ergebnis von **getMembers()** mit dem von **getAccessibleMembers()** des umschließenden Pakets.

In der Methode **getAccessiblePackages()** wird zunächst über alle mittels Paketimporten erreichbaren Pakete iteriert und das Ergebnis des Aufrufs von **getVisiblePackages()** auf den jeweiligen Paketen der Ergebnismenge hinzugefügt. Danach wird noch das umschließende Paket, sowie das Ergebnis des Aufrufs von **getAccessiblePackages()** auf dem umgebenden Paket ergänzt. Abbildung 5.13 zeigt die zugehörige Methodenimplementierung.

Die hier in diesem Abschnitt vorgestellten Methoden werden unter anderem von den im Editor implementierten und in Abschnitt 5.6.5 vorgestellten Validierungsregeln benutzt. Hierzu ist noch zu bemerken, dass die Methoden **getAccessiblePackages()** und **getVisiblePackages()** in den Klassen **Package** und **Class** nicht notwendig wären, sofern alle Typen des zu überprüfenden Modells auch im Paketdiagramm enthalten wären. Da dies aber in der Regel nicht der Fall ist, muss für die Überprüfung der Sichtbarkeit von nicht im Paketdiagramm enthaltenen Zieltypen stellvertretend die Sichtbarkeit des den Zieltyp umschließenden Pakets geprüft werden. Außerdem wird hierbei noch die Annahme

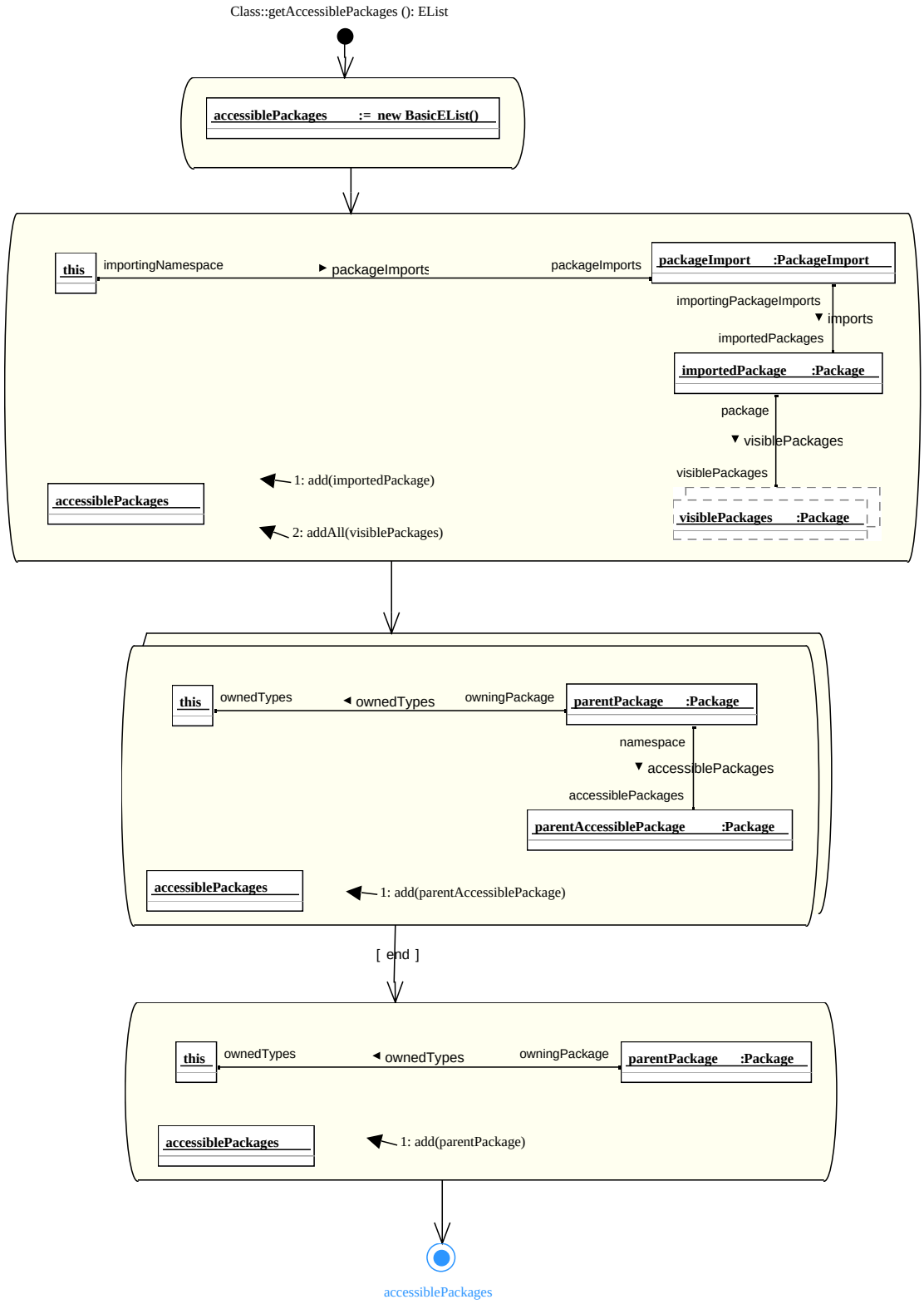

Abbildung 5.13: Implementierung der Methode **getAccessiblePackages**.

getroffen, dass nicht im Paketdiagramm enthaltene Zieltypen grundsätzlich öffentliche Sichtbarkeit besitzen und daher sichtbar sind, wenn die Typen des entsprechenden umschließenden Pakets sichtbar sind[5].

5.5.2 GMF Modell

Um mit Hilfe des Graphical Modeling Frameworks von Eclipse (GMF) [Gro09] ein graphisches Modell zu erstellen, aus dem dann ein Editor generiert wird, muss das zugrunde liegende Domänenmodell in EMF [SBPM09] vorliegen. Durch Erweiterungen und Anpassungen der Codegenerierung von Fujaba, ist es möglich, ein in Fujaba spezifiziertes Domänenmodell in ein EMF Modell zu transformieren [BDG07]. Es werden Java Klassen mit den Methodenrümpfen erzeugt, sowie eine Ecore Spezifikation, aus der mit Hilfe der EMF-Codegenerierung der Code für das Modell, die Editierfunktionen und ein einfacher Baumeditor erzeugt werden kann. Der Vorteil der Fujaba Codegenerierung ist, dass der Code, der aus dem dynamischen Modell erzeugt wird, erhalten bleibt. Das graphische Modell besteht im Wesentlichen aus drei Bestandteilen: (1) einer graphischen Definition der Knoten und Kanten im Modell (**Graphical Definition Model**), (2) einer Definition der Werkzeugpalette, um im Editor neue Knoten und Kanten erzeugen zu können (**Tooling Definition Model**) und (3) einem Modell, dass die Spezifikation des Domänenmodells, sowie die beiden vorher erwähnten Modelle zusammenfügt (**Mapping Model**).

Graphical Definition Model

Das Graphical Definition Model (*gmfgraph*) besteht aus drei Ebenen: Zuerst definiert man über *figures* die visuelle Repräsentation von Modellelementen (Knoten und Kanten) im Diagramm. Danach werden sog. *figure descriptors* und *accessors* angelegt, um auf die im ersten Schritt angelegten Elemente im nächsten Schritt referenzieren zu können. Zuletzt werden nun Diagramm Elemente definiert, die dann im Mapping Model benutzt werden können und die spezifischen Layoutinformationen für die einzelnen Elemente enthalten. Durch diese drei Ebenen erreicht man eine hohe Flexibilität im Graphical Definition Model, da ein hoher Grad an Wiederverwendbarkeit garantiert ist.

Figure können beispielsweise dazu verwendet werden, neue *figures* zu konstruieren und *figure descriptors* können von mehreren Diagrammelementen verwendet werden. Desweiteren kann ein und dasselbe Diagrammelement in mehreren Abbildungen (*mappings*) eingesetzt werden. Abbildung 5.14 zeigt das Graphical Definition Model des Paketdiagrammeditors.

Die Definition enthält unter anderem die wiederverwendbaren Elemente, wie z. B. die *Polyline Decoration OpenArrow*, die bei den *Figure* Elementen **ElementImportFigure_public**, **ElementImportFigure_private**, **PackageImportFigure_public** und **PackageImportFigure_private** als Pfeilspitze verwendet wird.

[5] Anmerkung: Dies könnte aber auch umgekehrt festgelegt werden: alle Elemente, die nicht im Paketdiagramm auftreten besitzen private Sichtbarkeit.

Abbildung 5.14: Das Graphical Definition Model des Paketdiagrammeditors.

Abbildung 5.15: Das Tooling Definition Model des Paketdiagrammeditors.

Bei der Darstellung der Importe wird anhand der Sichtbarkeit unterschieden, wie die visuelle Repräsentation des jeweiligen Imports aussieht. Öffentliche Importe werden als grüne gestrichelte Linie dargestellt, bei privaten Importen ist die Linie rot. Bei Paket Importen ist zusätzlich die Liniendicke gegenüber den Element Importen erhöht. Für die als Knoten angezeigten Modellelemente **Class**, **DataType**, **Enumeration** und **Package** sind ebenfalls *Figure Descriptor*s erstellt worden. Abschließend erfolgt die Auflistung der logischen Grafikelemente, denen je eine Figure zur Darstellung zugeordnet wird. Die Unterscheidung hierbei erfolgt nach Knoten (*Node*s in Abb. 5.14), Kanten (*Connection*s), Abschnitte (*Compartment*s) und Etiketten, die Zeichenketten anzeigen (*Diagram Label*s).

Ein Abschnitt (*Compartment*) kann dafür verwendet werden, um Modellelemente nicht nur auf der Zeichenfläche (dem *Canvas*) des Editors anzuzeigen, sondern auch als „Kind-Elemente“von Knoten. In unserem Fall können somit Pakete, Klassen und Datentypen auch innerhalb eines umgebenden Paketes angezeigt werden.

Tooling Definition Model

Das Tooling Definition Model ist eines der einfachsten Modelle in GMF. Primär wird mit dem Modell die Werkzeugpalette des resultierenden Editors bestimmt, die Werkzeuge zur Erstellung von Knoten und Kanten enthält. In unserem Fall ist das Tooling Definition Model in zwei Gruppen (*Tool Groups* in Abb. 5.15) aufgeteilt. Zum einen in die Gruppe namens *Nodes*, die Werkzeuge enthält, um Elemente im Diagramm zu erzeugen, die als Knoten dargestellt werden (wie z. B. Pakete, Klassen, Datentypen und Aufzählungen). Zum anderen gibt es die Gruppe *Imports*, die wiederum Werkzeuge bereitstellt, mit denen dem Diagramm Kanten hinzugefügt werden können. Diese Kanten sind die vier

oben erwähnten Importbeziehungen zwischen den einzelnen Elementen. Abbildung 5.15 zeigt das Tooling Definition Model des Paketdiagrammeditors.

Mapping Model

Das Mapping Model ist das Kernstück des GMF Models, das die von einander unabhängigen Modelle der graphischen Definition und der Werkzeugpalette zusammenfügt. Das Mapping Model wird in ein oder mehrere Generatormodelle transformiert, die die Templates für die Codeerzeugung erstellen.

Das *Canvas Mapping* Element ist zwingend notwendig, da es die Zeichenfläche des Diagramms repräsentiert. Der Assistent, der den Benutzer bei der Erstellung des Mapping Models unterstützt, fügt dem Canvas Mapping das ausgewählte *Domain Model* - das *EPackage* aus dem zugrunde liegenden EMF Modell, in unserem Fall das Paket **packagesdiagram**, dessen Wurzelelement (eine *EClass*, im konkreten Fall die Klasse **PackagesDiagram**), den *Diagram Canvas* aus dem Graphical Definition Model und die *Palette* aus dem Tooling Definition Model hinzu.

Die Einträge *Top Node Reference* für **Class**, **Package**, **DataType** und **Enumeration** repräsentieren Elemente, die auf der Zeichenfläche des Diagramms erzeugt werden. Diese Elemente enthalten eine sogenannte *Child reference* zu einem einzelnen *Node mapping*. Diese Elemente müssen irgendwo im zugehörigen Domänenmodell „enthalten“sein, sobald sie als Diagrammelemente instanziiert werden. Die Eigenschaft *Containment Feature* gibt an, wo diese neuen Objekte eingefügt werden. In unserem Fall sind die Elemente in der Assoziation von **Diagram** zu **Element** enthalten (vgl. Abbildung 5.7).

Das oben angesprochene *Node mapping* verbindet einen Knoten im Diagramm, das Werkzeug und das zugehörige Element aus dem Domänenmodell. Soll ein Diagrammelement wiederum Diagrammelemente enthalten können, so definiert man ein sogenanntes *Compartment Mapping*. Im Falle des Paketdiagrammeditors enthält die *Top Node Reference* für **Package** ein Compartment Mapping, um im Paket enthaltene Subpakete (im Modell in Abb. 5.7 ausgedrückt durch die Kompositionsbeziehung namens **nested Packages**), Klassen, Datentypen oder Aufzählungstypen (repräsentiert durch die Komposition **owned Types** in Abb. 5.7) anzeigen zu können. Abbildung 5.16 zeigt exemplarisch für das Element **Package** die Verbindung von Modellelement, graphischem Element und dem entsprechenden Werkzeug.

Neben den *Top Node Reference*s enthält das Mapping Model noch *Link Mapping*s. Sie werden dafür verwendet, um eine *Connection* aus dem Graphical Definition Model mit einem Element aus dem Domänenmodell und einem Werkzeug aus der Palette zu verbinden.

In unserem Fall repräsentieren die Link Mappings Klassen aus dem Domänen Modell, genauer repräsentieren sie **ElementImport** und **PackageImport**. Durch die jeweilige Unterscheidung der Sichtbarkeiten (öffentlich oder privat) ergeben sich in der Summe vier Link Mappings. Im endgültigen Diagramm werden die Klassen für die entsprechen-

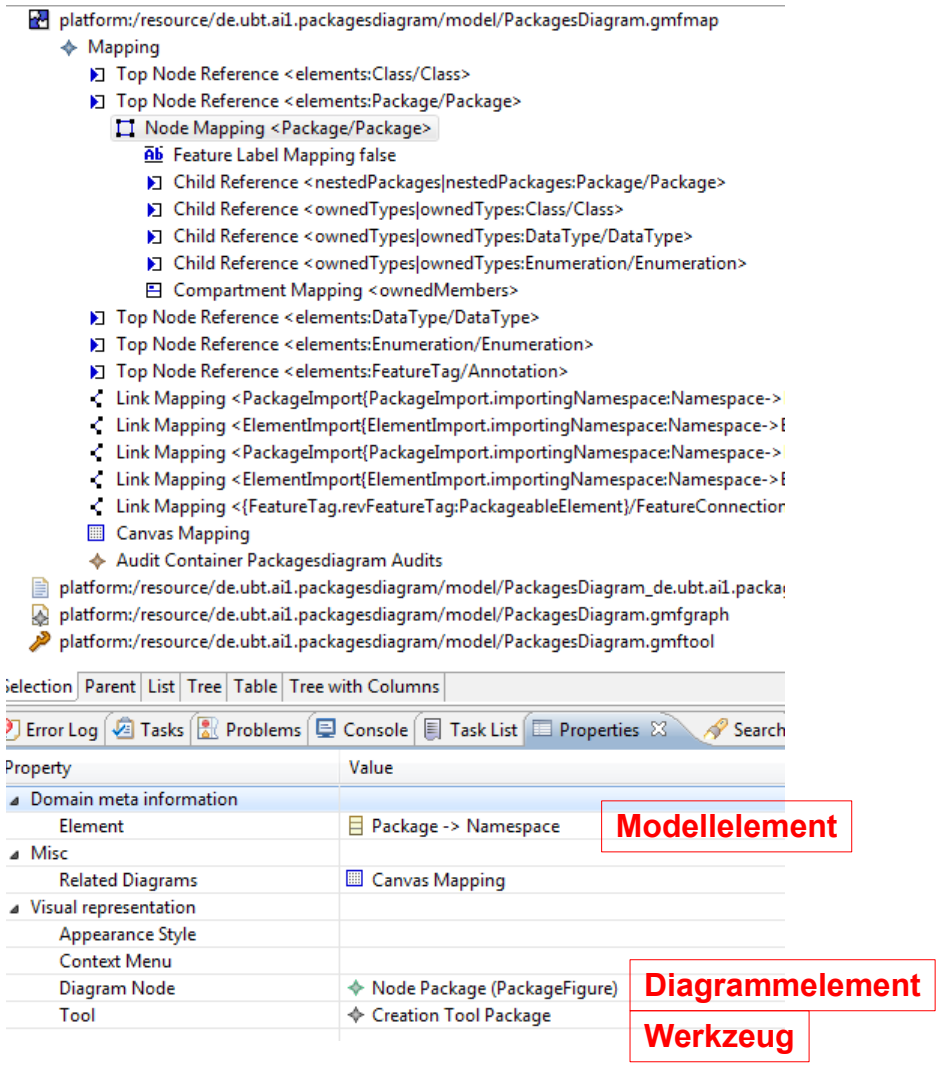

Abbildung 5.16: Verbindung von Modellelement, graphischem Element und Werkzeug im Mapping Model.

den Importe als Kante zwischen Paketen oder Typen dargestellt. Um eine eindeutige Zuordnung der Link Mappings zu den Domänenmodellelementen vorzunehmen, kommt OCL zum Einsatz. Ein einfaches OCL Constraint wie z. B. **Constraint self.visibility = VisibilityKind::PUBLIC** wird dafür verwendet, um öffentliche von privaten Importen zu unterscheiden und die dafür vorgesehene Repräsentation im Diagramm auszuwählen. Die ebenfalls zum Link Mapping gehörenden Einträge *Feature Seq Initializer* werden benötigt, um beim Anlegen eines entsprechenden Imports über die Palette die korrekte Sichtbarkeit zu setzen.

Neben den Definitionen der *Top Node References* und *Link Mappings*, kann man in dem Mapping Model auch sogenanne *Audit Container* spezifizieren, die wiederum *Audit*

*Rule*s enthalten. Diese Regeln können in OCL, Java, regexp oder nregexp angegeben werden. Desweiteren können sie „live“oder nach Nutzeraufforderung ausgeführt werden. Das Mapping Model in Abbildung 5.16 benutzt *Audit Rule*s, um die Bedingung der eindeutigen Namen von Elementen innerhalb eines Pakets sicherzustellen. Die Regel ist in OCL spezifiziert und lautet:

```
1 self.owningPackage->forAll(p | p.getOwnedMembers()->forAll(other
2 | self <> other implies self.name <> other.oclAsType(
    packagesdiagram::
3 NamedElement).name))
```

Listing 5.3: OCL-Regel zur Sicherstellung von eindeutigen Namen

Diese Bedingung wird für jeden Typ und jedes Paket angewendet, self bezieht sich also entweder auf einen Typen oder auf ein Paket. Von self aus wird dann zum umschließenden Paket navigiert und dessen Inhalt durchlaufen. Alle darinliegenden Typen und Pakete werden dabei einmal in other abgelegt und mit self verglichen. Handelt es sich bei self und other nicht um dasselbe Objekt, so müssen die Namen der beiden Objekte unterschiedlich sein. Wird diese Regel beim Anlegen von neuen Elementen verletzt, so erfolgt eine entsprechende Rückmeldung an den Benutzer.

Wie bereits am Anfang des Abschnitts erläutert, werden nun aus dem Mapping Model ein oder mehrere Generatormodelle erzeugt. Aus einem Generatormodell kann nun der ausführbare Code für den graphischen Editor erzeugt werden.

5.5.3 Hilfsklassen

Neben den bisher vorgestellen Modellen, aus denen jeweils der ausführbare Code erzeugt wird, enthält der Paketdiagrammeditor noch zwei wichtige Hilfsklassen, die komplett manuell implementiert wurden.

NamespaceUtil

Die Klasse **NamespaceUtil** im Paket **de.ubt.ai1.packagesdiagram.util** enthält eine überladene Methode **getFullyQualifiedName**, die für folgende Modellelemente, die als Methodenargument übergeben werden, den voll qualifizierten Namen als String zurückliefert:

- **EPackage**
  Für ein EPackage (Bestandteil eines EMF Modells), wird die Eigenschaft *NS Prefix* zurückgeliefert, die den voll qualifizierten Paketnamen enthält.

- **EClass**
  Der voll qualifizierte Name des Pakets, in dem das aktuelle Element liegt wird mit „.“ und dem Namen des entsprechenden *EClassifier*s ergänzt.

- **EObject**
  In diesem Fall wird anhand des tatsächlichen Typs des übergebenen EObjects eine der oben beschriebenen Methoden, oder für den Fall einer *EReference* der voll qualifizierte Name des davon referenzierten Typs (mithilfe der Methode *getFullyQualifiedName*) aufgerufen. Für alle anderen von EObject abgeleiteten Typen wird „unknown“zurückgeliefert.

- **Package**
  Diese rekursive Methode ermittelt den voll qualifizierten Namen des gewünschten Pakets durch Konkatenation des voll qualifizierenden Namens des umschließenden Pakets mit „.“und dem Namen des aktuellen Pakets.

- **Type**
  Für die Type Objekte wird der voll qualifizierte Name des umschließenden Pakets wie oben beschrieben ermittelt. Anschließend erfolgt die Konkatenation von „.“und der Name des Typs.

PackagesDiagramBuilder

Die Klasse **PackagesDiagramBuilder** im Paket **de.ubt.ai1.packagesdiagram.builder** dient dazu, programmatisch ein neues Paketdiagramm bzw. neue Elemente in einem Paketdiagramm zu erzeugen. Eine Instanz eines PackagesDiagramBuilders referenziert genau ein PackagesDiagram Objekt. Von diesem Wurzelobjekt aus sind alle Elemente des Paketdiagramms erreichbar. So ist es unter anderem möglich, mittels des voll qualifizierten Namens eines Elements, die entsprechende Referenz aus dem Paketdiagramm zu erhalten (**PackagesDiagramBuilder::get(String fullQualifiedName)**).

Desweiteren stellt die Klasse PackagesDiagramBuilder factory-Methoden zur Erzeugung von neuen Elementen (wie z. B. Paketen, Klassen, Datentypen und Aufzählungstypen) im Paketdiagramm zur Verfügung (vgl. Factory-Pattern in [FF04]).

Für das Erzeugen von Importbeziehungen im Paketdiagramm steht die eingebettete Klasse **ImportBuilder** zur Verfügung. Sie besitzt Methoden zum Festlegen der Sichtbarkeitseigenschaften (**publicly()** oder **privately()**), die jeweils eine Instanz vom Typ **ImportBuilder** zurückliefern. Desweiteren stehen Methoden zur Erzeugung von Paket- oder Elementimporten zur Verfügung (**importPackage(String fullQualifiedName)** und **importElement(String fullQualifiedName)**). Als Einstiegspunkt zur Erzeugung von Importen dient die Methode **let(String fullQualifiedName)** in der Klasse **PackagesDiagramBuilder**. Sie liefert anhand des im Aufrufparameter übergebenen voll qualifizierten Namen des importierenden Namensraums eine Instanz der Klasse **ImportBuilder**, auf der dann die Sichtbarkeitseigenschaften und der Zielnamensraum gesetzt werden können. Der Import des Pakets **de.ubt.ai1.modpl.core.product** im Paket **de.ubt.ai1.modpl.core.delta** ließe sich somit wie folgt ausdrücken:

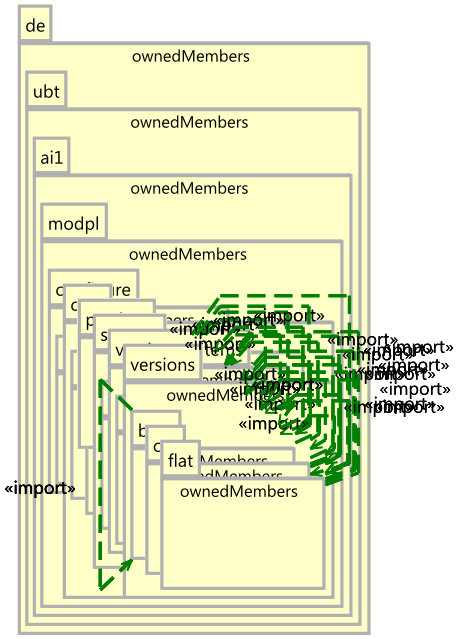

Abbildung 5.17: Darstellung des Paketdiagramm Modells, nach der Initialisierung der graphischen Ansicht durch die GMF Laufzeitumgebung.

```
builder.let("de.ubt.ai1.modpl.core.delta").publicly().importPackage("de.ubt.ai1.modpl.core.product");
```

Die Klasse *PackagesDiagramBuilder* arbeitet nach außen hin mit den voll qualifizierten Namen der Elemente und ermöglicht so ein einfaches Anlegen oder Auffinden von Elementen im Objektgraphen.

5.5.4 Erweiterungen des graphischen Editors

Um die Benutzbarkeit des generierten Editors vor allem bei größeren Paketmodellen zu erhöhen, wurden die nachfolgend beschriebenen Erweiterungen implementiert.

Automatisches Layout

Beim Initialisieren der graphischen Ansicht eines Paketdiagramm Modells, werden die entsprechenden graphischen Elemente durch die GMF Laufzeitumgebung mit Standardwerten für Größe und Position relativ zum Vaterelement eingefügt. Hieraus resultiert eine Darstellung, wie sie in Abbildung 5.17 zu sehen ist.

Es ist offensichtlich, dass anhand dieser Darstellung keine Aussagen über die Abhängigkeit der Pakete getroffen werden können. Der Benutzer kann nun entweder die Pakete einzeln per „Drag and Drop“ neu anordnen, oder aber die Funktion zur automatischen Optimierung des Layouts starten. Sie ist über den Kontextmenüeintrag *Optimize Layout* zu erreichen und in der Klasse **LayoutOptimizer** im Paket **de.ubt.ai1.packagesdiagram.diagram.extended.layout** implementiert. Die dort implementierte *Action*

ermittelt alle Knoten des Diagramms und übergibt die resultierende Liste an den **LayoutService**[6] aus der GMF Laufzeitumgebung (definiert im Paket: **org.eclipse.gmf.runtime.diagram.ui.services.layout**). Die Baumstruktur der Knoten wird nun in einem Bottom-Up Durchlauf abgearbeitet und dabei werden die einzelnen Pakete so nebeneinander angeordnet[7], dass keine Überlappungen in der Darstellung mehr bestehen.

Abbildung 5.18 zeigt den Inhalt des Paketdiagrammeditors nach der automatischen Optimierung des Layouts.

Teilbaum des Modells in separatem Fenster anzeigen

Sobald das Softwaresystem größer wird, steigt automatisch auch die Anzahl der Pakete bzw. deren Abhängigkeiten untereinander. Auch dies trägt zu einer gerineren Übersichtlichkeit bei. Daher bietet der Paketdiagrammeditor die Möglichkeit, Subpakete in einem separaten Editorfenster anzuzeigen. Abbildung 5.19 zeigt die Darstellung des Subpakets **de.ubt.ai1.mod2scm.core**.

Die Realisierung der Subpaketdarstellung wird durch das sogenannte *Diagram Partitioning*[8] erreicht. Hierbei werden weitere GMF Modelle für ein zweites Editor Plugin definiert, das auf der selben Instanz des semantischen Modells (das zugrunde liegende EMF Modell des Paketdiagramms) arbeitet. Es werden wiederum die graphischen Elemente definiert und eine Abbildung auf die Bestandteile des Metamodells erstellt. Allerdings ist nun zu beachten, dass in den erzeugten Generatormodellen für beide Editoren für das zur Darstellung verwendete *EditPart* eine *Open Diagram Behaviour* edit policy angelegt wird (vgl. Abschnitt 5.5.2). Es ist zu beachten, dass diese manuelle Änderung bei jeder Neuerzeugung des Generatormodells verloren geht und erneut vorgenommen werden muss. Abbildung 5.20 zeigt einen Ausschnitt des angepassten Generatormodells. Das für die Darstellung der Pakete zuständige Edit Part ist *Gen Top Level Node PackageEditPart*. Für diesen Knoten wird als neues Kindelement die *Open Diagram Behaviour OpenDiagramEditPolicy* erzeugt. Nachdem der Code für den graphischen Editor neu generiert wurde, kann die Subpaketdarstellung durch Doppelklick im oberen Bereich der Paketfiguren geöffnet werden.

Erreichbare Pakete graphisch hervorheben

Mit steigender Anzahl von Importabhängigkeiten der Pakete untereinander, wird es zunehmend schwerer zu erkennen, welche Pakete bzw. Elemente von einem anderen Paket oder Element aus importiert werden. Der Paketdiagrammeditor zeigt deshalb zum einen alle von dem im graphischen Editor selektierten Paket aus erreichbaren Pakete auf einer

[6] http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.gmf.doc/examples-guide/diagram/layoutServiceExample.html

[7] Diese Art des Layouts wird in der zugehörigen Dokumentation als *Square* Layout bezeichnet. Hierbei wird eine rechteckige Fläche anhand der Anzahl der übergebenen Elemente berechnet, und die Elemente darin anhand eines Gitters ausgerichtet

[8] http://wiki.eclipse.org/Diagram_Partitioning
Stand 03. November 2009

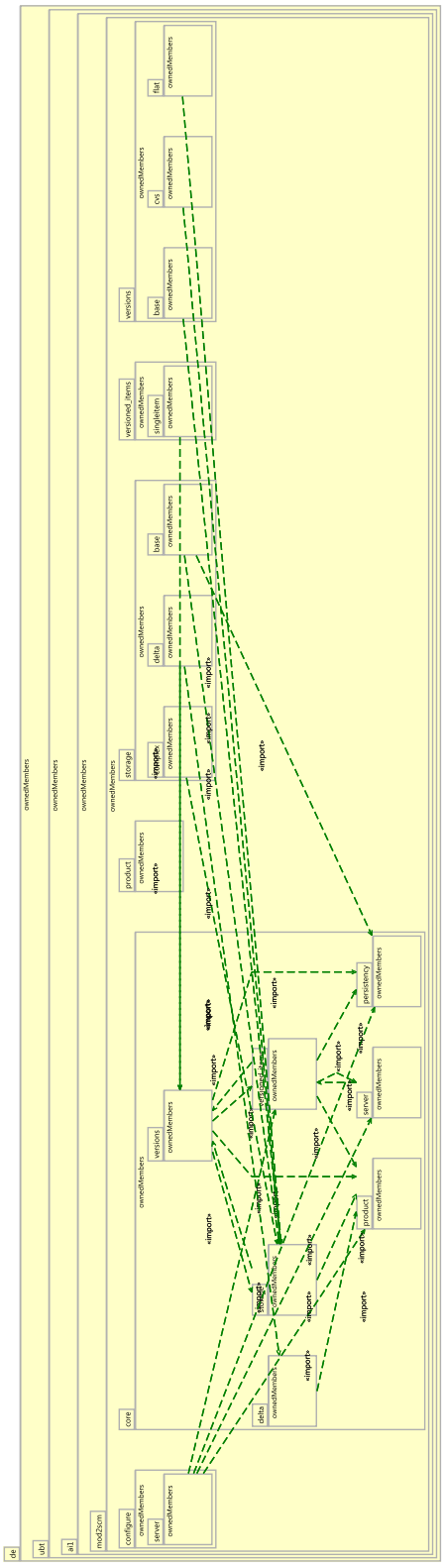

Abbildung 5.18: Darstellung des Paketdiagramm Modells, nach Aufruf des Layout Optimierers.

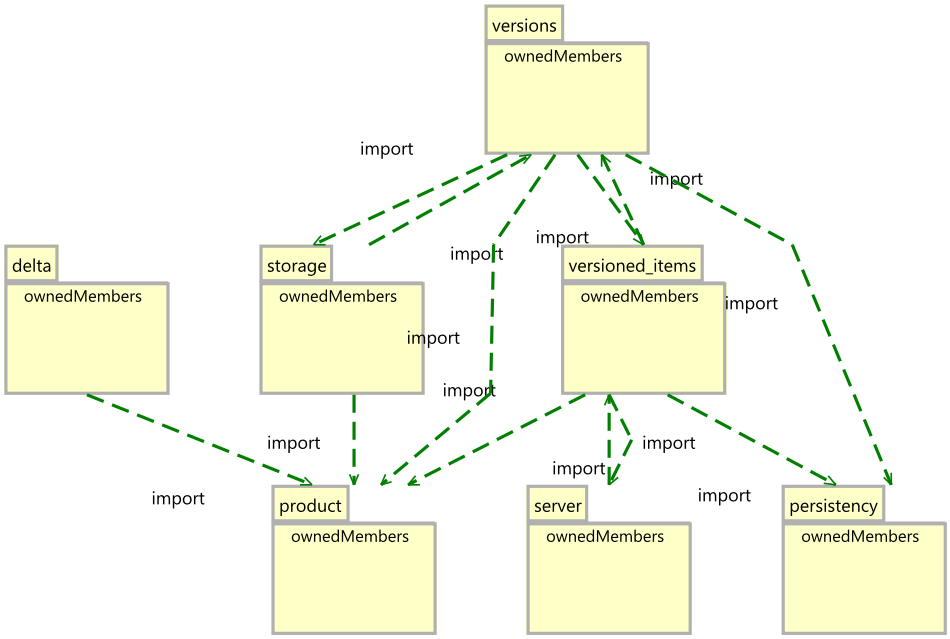

Abbildung 5.19: Darstellung des Inhalts von Subpaketen.

speziellen Eigenschaftenseite an, die als Eclipse View bereitsteht.

Zum anderen werden in der graphischen Darstellung alle vom ausgewählten Paket oder Element aus erreichbaren Paketdiagrammelemente graphisch hervorgehoben und auch die zugehörigen Importbeziehungen werden mit einer durchgezogenen anstatt einer gestrichelten Linie dargestellt. Die standardmäßig sichtbaren umschließenden Pakete werden aber nicht farblich hervorgehoben, um die Übersichtlichkeit nicht einzuschränken.

In beiden Fällen werden die vom Paketdiagramm Modell zur Verfügung gestellten Methoden **getAccessibleMembers()** und **getAccessiblePackages()** zur Ermittlung aller erreichbaren Elemente verwendet.

Desweiteren wurde der graphische Editor um eine Baumansicht ergänzt, die die Pakethierarchie darstellt. Bei Auswahl eines Paketes im Baum, wird das entsprechende Element im graphischen Editor, sowie auch alle von diesem Element aus erreichbaren Paketdiagrammelemente farblich hervorgehoben. Abbildung 5.21 zeigt den graphischen Editor mit der Baumansicht und dem im Baum selektierten Paket **de.ubt.ai1.mod2scm.storage.base**. Das Paket selbst besitzt nur zwei öffentliche Paketimporte: zu **de.ubt.ai1.mod2scm.core.storage** und zu **de.ubt.ai1.mod2scm.core.persistency**. Da diese Pakete selbst andere Pakete öffentlich importieren, ergibt sich wegen der Transitivität der öffentlichen Paketimporte das gezeigte Bild.

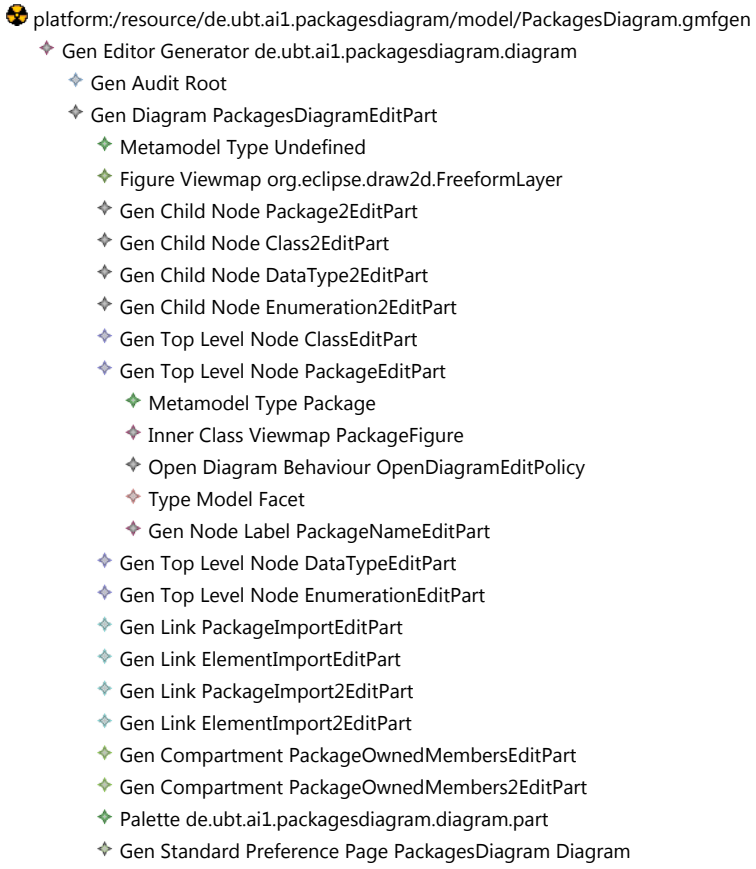

Abbildung 5.20: Modifiziertes Generatormodell zur Darstellung von Subpaketen.

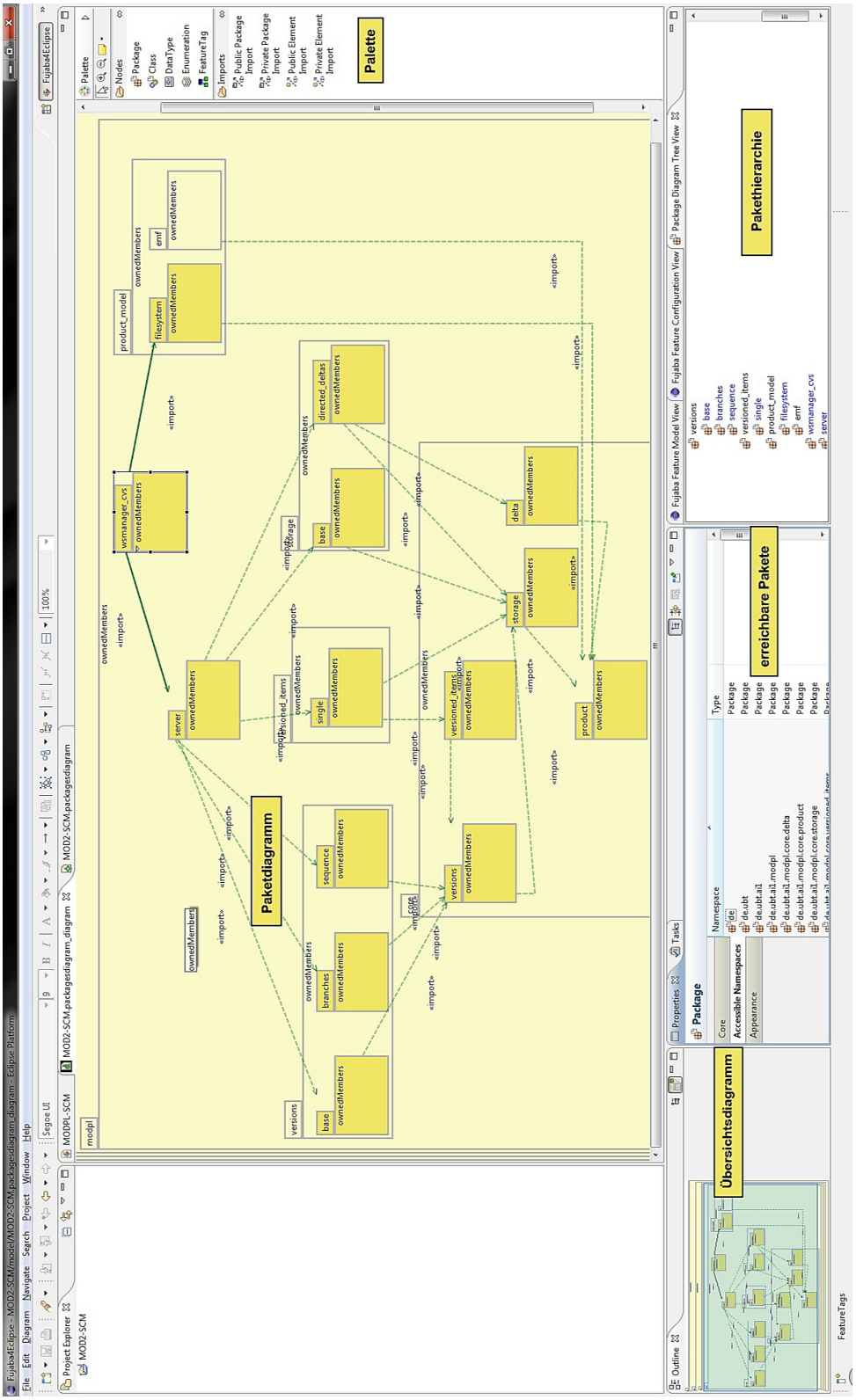

Abbildung 5.21: Farbliche Hervorhebung von selektierten Paketen und der von dort aus erreichbaren Elemente.

5.5.5 Optimierung der Importe durch Reduktion

Modelle für nicht-triviale Probleme sind erfahrungsgemäß sehr groß und enthalten eine Vielzahl von Paketen und Importbeziehungen. Dies führt dazu, dass es für den Modellierer zunehmend schwieriger wird, die Abhängigkeiten zu erkennen. Es ist also notwendig, die Zahl der Importe wenn möglich zu verringern. Dazu kann man die Eigenschaften der UML Importe, wie etwa die Transitivität der öffentlichen Importe, ausnutzen. Weiterhin können auch bei der automatischen Berechnung von Importen (siehe Abschnitt 5.6.4) unter Umständen Redundanzen auftreten, die aber ebenfalls durch Ausnutzung der Transitivität öffentlicher Importe vermieden werden können. Daher wurden Algorithmen implementiert, die in einem Nachbearbeitungsschritt versuchen, die Zahl der im Paketdiagramm enthaltenen Importe zu minimieren. Bei dem Reduktionsschritt wird zunächst im Speicher eine Kopie des Paketdiagramms erzeugt und anschließend werden sukzessive einzelne Importbeziehungen entfernt. Nach jedem Löschen eines Imports wird überprüft, ob das Ziel des entfernten Imports von der Quelle aus noch sichtbar ist. Ist dies der Fall, so handelt es sich um einen redundanten Import und er kann endgültig gelöscht werden. Ist das Element jedoch nicht mehr sichtbar, so muss der Import wieder eingefügt werden. Diese automatische Optimierung der Importbeziehungen ersetzt aber keinesfalls die manuelle Nachbearbeitung durch den Modellierer. Sie ist allenfalls als Hilfsmittel gedacht, um die Zahl der automatisch generierten Importe beim Roundtrip Engineering zu begrenzen und so die Übersichtlichkeit zu erhöhen. Für den Durchlauf durch das Paketdiagramm wurden drei verschiedene Strategien implementiert:

Bottom-up Strategie

Hierbei wird zunächst die Paketstruktur vom innersten Paket nach außen hin durchlaufen. Es werden alle Paketimporte überprüft, die ein Paket außerhalb des Elternpakets referenzieren. Danach werden die restlichen Importbeziehungen des aktuellen Pakets betrachtet. Referenziert nun eine vorher definierte Anzahl oder Prozentsatz von Paketen innerhalb einer Ebene ein beliebiges Paket außerhalb, so können diese einzelnen Importe durch einen einzigen Paketimport ausgedrückt werden, der vom Elternpaket ausgeht.

Outer-first Strategie

Hierbei wird auch bei den Blättern begonnen. Allerdings werden hier bei den einzelnen Paketen zuerst Importbeziehungen betrachtet, die außerhalb des umschließenden Pakets enden.

Vereinigen von Importen auf der nächsthöhreren Ebene

Bei dieser Strategie werden Importe von Subpaketen durch einen Import des umschließenden Pakets ausgedrückt. Dies macht aber nur Sinn, wenn mindestens zwei Importe von Subpaketen jeweils das gleiche Ziel haben. Abbildung 5.22 zeigt einen Ausschnitt aus dem MOD2-SCM Paketdiagramm vor (links im Bild) und nach der Optimierung (rechts im Bild). Die drei Pakete base, branches und sequence aus dem Paket versions importieren

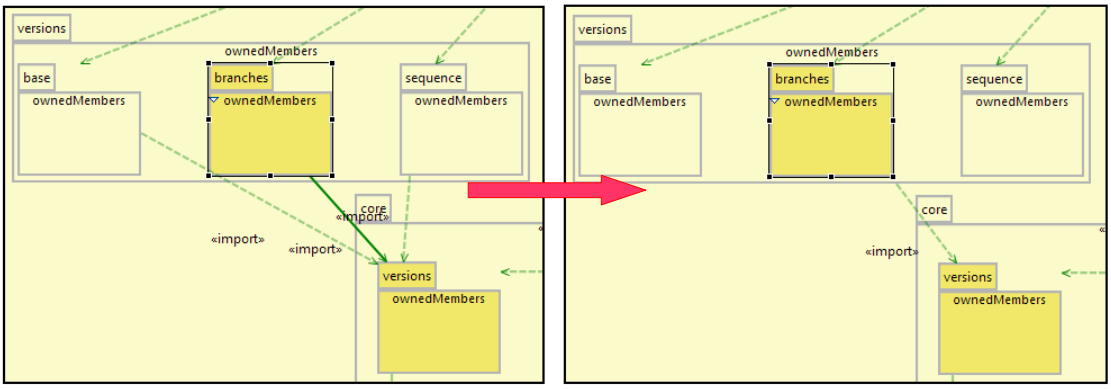

Abbildung 5.22: Reduktion der Importe

jeweils das Paket core.versions. In diesem Falle, können die öffentlichen Importbeziehungen zu einem einzelnen Paketimport, der vom umschließenden Paket (versions) ausgeht, zusammengefasst werden.

5.6 Integrator Werkzeug für bestehende Modellierungswerkzeuge

Im folgenden Abschnitt wird die Integration des Paketdiagrammeditors in die bestehenden Modellierungswerkzeuge für Ecore Modelle und in Fujaba beschrieben. Diese Integration ist essentiell, wenn der Editor sinnvoll in der Disziplin *Modellieren-im-Großen* eingesetzt werden soll. Andernfalls würden die Paketdiagramme wiederum zu einem reinen Dokumentationsmedium degradiert. Die Fujaba Integration umfasst alle in Abschnitt 5.4.2 beschriebenen Punkte, da Fujaba auch für die modellgetriebene Entwicklung der MOD2-SCM Produktlinie eingesetzt wurde und als einziges Werkzeug ausführbaren Code aus Modellen erzeugen kann. Die Umsetzung des Integrators für Ecore enthält die Punkte **Forward Engineering**, **Reverse Engineering** und **Validierung**. Da für die Fujaba Integration auch Änderungen an Fujaba selbst notwendig waren, werden diese im folgenden Unterabschnitt näher erläutert.

5.6.1 Integration des Paketdiagrammeditors in Fujaba

Im folgenden Abschnitt wird beschrieben, wie der neu entwickelte Paketdiagrammeditor in das bereits vorhandene, und zur modellgetriebenen Softwareentwicklung benutzte, CASE Werkzeug *Fujaba*[9] integriert wurde. Insbesondere sollen bei der Modellierung von Klassen oder Methoden die Sichtbarkeiten überprüft und ggf. direkt Anpassungen am

[9]Von Fujaba existieren im Wesentlichen vier verschiedene Versionen, die sich zum Teil in der verwendeten Benutzeroberfläche (Swing, GEF, Eclipse mit Swing) unterscheiden, oder gar wie Moflon auf einem anderen Metamodell (MOF) basieren und mächtigere Modellierungstechniken wie Triplegraphgrammatiken bereitstellen. Der Paketdiagrammeditor wurde primär in die unter Eclipse lauffähige Fujaba Version mit Swing GUI - Fujaba4EclipseSwingUI genannt - integriert.

Paketmodell vorgenommen werden. Fujaba kennt keine Paketsichtbarkeiten und nimmt dementsprechend auch keine Überprüfung bei Modellierungsoperationen vor. Folglich sind alle Elemente, die im aktuell geöffneten Fujaba Projekt vorhanden sind und alle Elemente aus importierten Projekten immer von überall aus sichtbar. Dies ist auch konform mit der UML Spezifikation, da ja der Zugriff auf Elemente über deren voll qualifizierten Namen immer erfolgen kann (siehe Abschnitt 5.4.1). Da aber zum einen ab einer gewissen Projektgröße die Übersicht sehr stark leidet, wenn in allen Dialogen jeweils alle im gesamten Modell verfügbaren Typen angezeigt werden, zum anderen der Modellierer keinerlei Informationen über Abhängigkeiten zwischen den einzelnen Bausteinen des zu entwickelnden Systems abrufen kann, bietet sich eine engere Integration des Paketdiagrammeditors als von der UML vorgesehen an.

5.6.2 Das Connector Plugin

Um auf das Paketmodell in Fujaba zugreifen zu können, wurde ein Plugin für *Fujaba4EclipseSwingUI* entwickelt, das vollen bidirektionalen Zugriff zwischen beiden Metamodellen (das Fujaba auf der einen, das Paketdiagramm Metamodell auf der anderen Seite) erlaubt. Abbildung 5.23 zeigt die für das Connector Plugin relevanten Teile des Fujaba Metamodells.

Da in diesem Kontext einige Anpassungen an Fujaba (vor allem an der Benutzeroberfläche) nötig waren, wurde das Connector Plugin nicht vollständig modellgetrieben entwickelt. Große Teile des Programmcodes wurden manuell erstellt. Der Zugriff auf das Paketdiagramm Metamodell gestaltete sich dank der in Abschnitt 5.5.3 beschriebenen Hilfsklassen sehr einfach. Kernfunktionen des Plugins sind:

- Erstellen eines neuen Fujaba Modells auf Basis eines Paketdiagramms
- Bidirektionale Synchronisation zwischen Paketdiagramm Modell und Fujaba Modell
- Erstellen eines Paketdiagramms anhand eines bereits bestehenden Fujaba Modells, incl. Berechnung der Importe
- Überprüfen von Sichtbarkeitsregeln während des Editieren des Fujaba Modells und entsprechende Anpassung des Paketdiagramms
- Einschränken der in Fujaba Dialogen angezeigten Elemente auf die nach der im Paketmodell definierten Sichtbarkeiten

Der gewählte Ansatz eines Integrators [BDK09] zwischen beiden Modellen hat den Vorteil, dass das Fujaba Metamodell nicht verändert werden muss. Dadurch bleibt die volle Kompatibilität zu bereits vorhandenen, mit Fujaba erstellten Domänenmodellen vorhanden. Auf zukünftige Änderungen am Fujaba Metamodell kann mit Anpassungen des Integrators leicht reagiert werden.

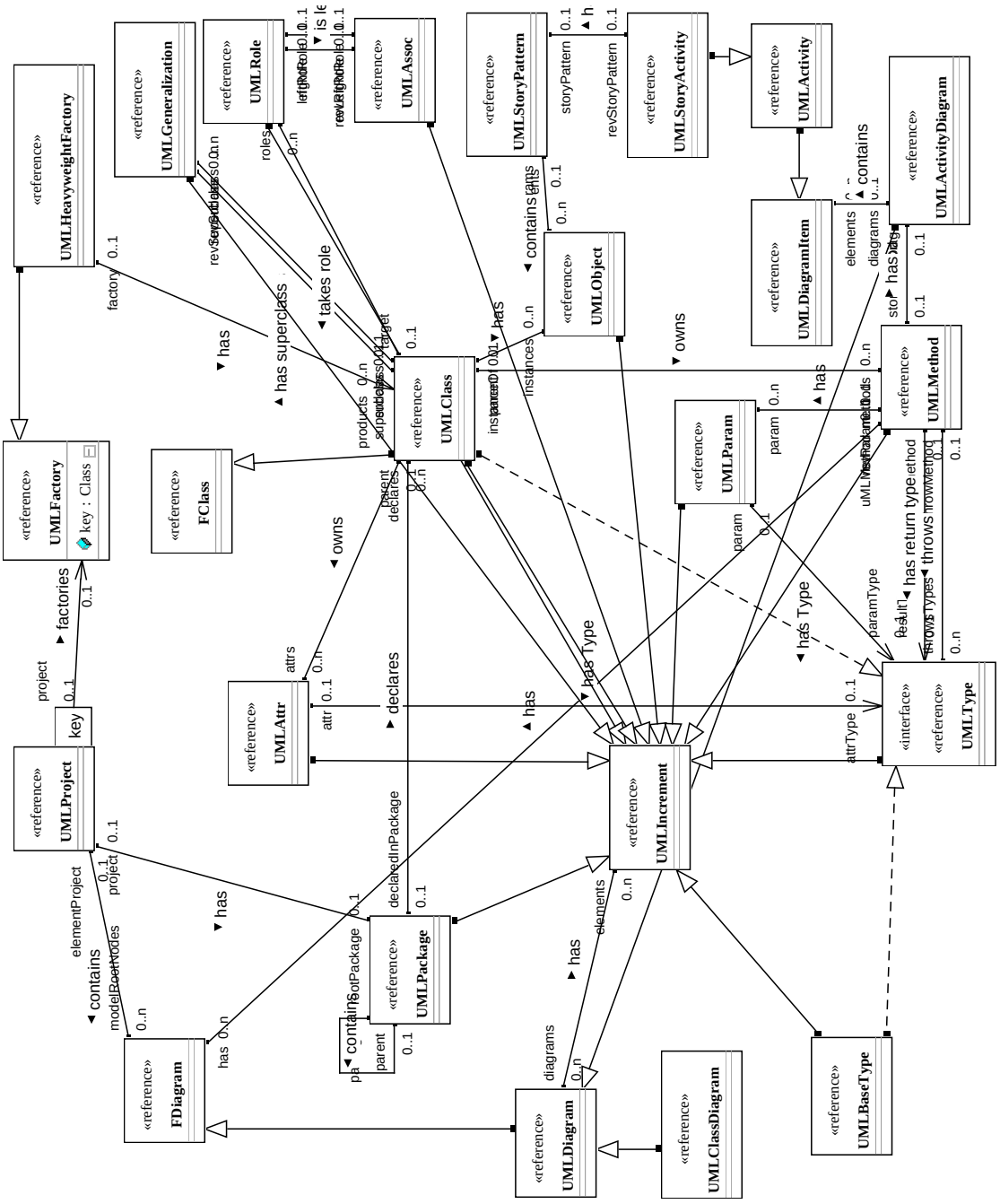

Abbildung 5.23: Ausschnitt aus dem Fujaba Metamodell.

```
1  private UMLPackage createPackageInFujaba(Package pdPackage,
2    FFactory<UMLPackage> factory) {
3     UMLProject project = (UMLProject) FrameMain.get().
          getSelectedProject();
4       UMLPackage root = project.getRootPackage();
5
6       UMLPackage currentPackage = factory.create();
7       currentPackage.setName(pdPackage.getName());
8
9       if (pdPackage.getNestingPackage() != null) {
10      currentPackage.setParent(root.findPackage(NamespaceUtil.
            getFullyQualifiedName(pdPackage.getNestingPackage())));
11      }
12      else
13              currentPackage.setParent(root);
14
15      return currentPackage;
16 }
```

Listing 5.4: Quelltext der Methode zum Anlegen eines Pakets in Fujaba als Pendant zu einem bestehenden Paket im Paketdiagramm Modell.

5.6.3 Forward Engineering

Um Paketdiagramme als a priori Hilfsmittel zur Planung von Architekturen nutzen zu können, muss eine Möglichkeit geschaffen werden, das im Paketdiagramm erstellte grobe Gerüst in das gewünschte Zielmodell (Ecore oder Fujaba) zu überführen. Da beide Zielplattformen Pakete, nicht aber die Importe zwischen ihnen unterstützen, wird in einem Stapelverarbeitungsprozess die Pakethierarchie ausgehend vom Wurzelpaket durchlaufen und für jedes Paket im Paketdiagramm ein Pendant im jeweiligen Zielmodell angelegt. Im Falle von Ecore wird entsprechend eine Instanz der Klasse **EPackage** mit gleichlautendem Namen an der gleichen Stelle im Paketbaum erzeugt, während in Fujaba eine Instanz der Klasse **UMLPackage** erzeugt wird. Auch hier wird der Name und das Vaterpaket entsprechend gesetzt. Enthalten die Pakete darüberhinaus auch noch Elemente (Klassen, Datentypen oder Aufzählungen), so werden auch hierfür die entsprechenden Objekte im Zielmodell angelegt. Da man im Paketdiagramm nur den Namen des Elements spezifizieren kann, müssen Attribute, Methoden und Assoziationen vom Modellierer manuell während des Verfeinerungsschritts in der Disziplin *Modellieren-im-Kleinen* angelegt werden. Listing 5.4 zeigt die Methode, die ein neues Fujaba Paket zu einem bestehenden Eintrag im Paketdiagramm anlegt.

5.6.4 Reverse Engineering

Ziel des Reverse Engineering ist es, aus einem bestehenden Ecore oder Fujaba Modell ein Paketdiagramm abzuleiten, das einen Überblick der Architektur liefert. Dies kann prinzipiell in mehreren Schritten, je nach Benutzervorgabe geschehen:

Pakethierarchie Für eine weitere Bearbeitung durch den Benutzer wird zunächst nur die Pakethierarchie aus dem Modell extrahiert und in einem neuen Paketdiagramm gespeichert.

Hinzufügen von Elementen Ausgewählte Elemente (Klassen, Datentypen, Aufzählungen) aus den jeweiligen Quellmodellen werden in das Paketdiagramm eingefügt.

Berechnung der Importe Anhand des Quellmodells können automatisch alle Importe berechnet werden, die benötigt werden, damit das Quellmodell anhand des Paketdiagramms erfolgreich validiert werden kann.

Automatisches Erzeugen von Paketdiagrammen aus Ecore Modellen

Um ein initiales Paketdiagramm anhand eines bestehenden Ecore Modells zu erzeugen, wurde eine Funktion „Initialize/Update Package Diagram“ mittels einer sogenannten *objectContribution*, die im Erweiterungspunkt *org.eclipse.ui.popMenus* definiert ist, in die Eclipse Umgebung integriert. Der Erweiterungspunkt referenziert eine Aktion (**de.ubt.ai1.packagesdiagram.ecoreimport.actions.PackagesDiagramInitialization**, die ausgeführt wird, sofern eine Datei mit der Endung „.ecore“ ausgewählt ist. Es wird eine Instanz der Klasse **de.ubt.ai1.packagesdiagram.ecoreimport.Ecore2PackagesDiagramConverter** mit der ausgewählten Ecore Resource erzeugt. Die Klasse enthält die Methode *convert*, die das ausgewählte Ecore Modell mittels eines *TreeIterators* traversiert und für jedes gefundene Paket (**EPackage**) mit Hilfe des in Abschnitt 5.5.3 vorgestellten *PackagesDiagramBuilder* einen entsprechenden Eintrag im Paketdiagramm erzeugt (hierfür wird zuerst mit der in Abschnitt 5.5.3 erwähnten Klasse *NamespaceUtil* der voll qualifizierte Name des Pakets ermittelt). Wird diese Methode auf einem bereits bestehenden Paketdiagramm ausgeführt, so wird eine Aktualisierung der Pakethierarchie anhand des ausgewählten Ecore Modells durchgeführt.

Neben der oben erwähnten Funktion zum Hinzufügen bzw. Aktualisieren von Paketen des gesamten Ecore Modells, können auch einzelne Bestandteile aus dem Ecore Modell dem Paketdiagramm hinzugefügt werden. Hierfür wurde eine entsprechende Aktion **de.ubt.ai1.packagesdiagram.ecoreimport.actions.EObjectImport** ebenfalls als *objectContribution* der Eclipse Umgebung hinzugefügt. Diese Aktion stellt die Funktion namens „Add to Package Diagram“ bereit, die verfügbar ist, sobald im Ecore Modell ein Element vom Typ *EObject* selektiert ist. Es wird eine Instanz vom Typ **de.ubt.ai1.packagesdiagram.ecoreimport.Ecore2PackagesDiagramImporter** erzeugt und die Methode *importEObject* aufgerufen. Falls das selektierte Element eine Klasse (*EClass*), eine Aufzählung (*EEnum*), ein Datentyp (*EDataType*) oder ein Paket (*EPackage*) ist, wird wie oben erwähnt, das entsprechende Element im Paketdiagramm erzeugt und in den Objektgraphen eingefügt. Ist das gewählte Element von Typ *EPackage*, so werden auch alle darin geschachtelten Pakete dem Paketdiagramm hinzugefügt.

Automatisches Erzeugen von Paketdiagrammen aus Fujaba Modellen

Um ein Paketdiagramm aus einem bereits bestehenden Fujaba Modell zu erzeugen, wird zunächst ein neues Paketdiagramm erstellt. Der Name der Modelldatei wird analog zum Namen des Fujaba Modells gewählt. Anschließend wird das Fujaba Modell ausgehend vom Wurzelpaket durchlaufen. Die Klasse UMLProject, die das aktuelle Fujaba Projekt repräsentiert, das das gewünschte Fujaba Modell enthält, besitzt eine Referenz auf das Wurzelpaket. Von dieser Stelle ausgehend wird der transitive Abschluß aller Kindpakete gebildet, die vom Wurzelpaket aus erreichbar sind. Sämtliche auf diesen Pfaden befindliche Pakete werden mit Hilfe der Klasse PackagesDiagramBuilder nun auch im Paketdiagramm erzeugt. Durch Verwendung des voll qualifizierten Namens beim Erzeugen, erfolgt automatisch die Erzeugung an der richtigen Stelle innerhalb der Pakethierarchie. Klassen bzw. Datentypen werden in diesem Schritt nicht betrachtet, da das Paketdiagramm eine grobgranulare Sicht der Architektur darstellen und außerdem keine Konkurrenz zu den herkömmlichen UML Klassendiagrammen bilden soll. Selbstverständlich können vom Modellierer aber auch ausgewählte Klassen dem Paketdiagramm explizit hinzugefügt werden. Dies geschieht aber immer pro selektiertem Element im Fujaba Klassendiagramm und nur per Benutzerinteraktion. Auch hierfür wird wiederum die Hilfsklasse PackagesDiagramBuilder verwendet, die sicherstellt, dass die Klasse anhand ihres qualifizierten Namens auch im korrekten Paket angelegt wird.

Ermittlung von Import-Beziehungen

Neben dem Exportieren von Paketen bietet die Reverse Engineering Funktion des Integrators auch noch die Möglichkeit, durch Analyse des Quellmodells Importbeziehungen abzuleiten. Dazu muss der Benutzer die Art der gewünschten Importe (privat/öffentlich, Paket-/Elementimport) auswählen, die dann automatisch im Paketdiagramm erstellt werden. Die Vorgehensweise ist für beide Modellarten identisch: Die Ermittlung der Importe erfolgt anhand der verwendeten Klassen im Modell. Für jede Klasse wird überprüft, ob

- Attributtypen
- Methodenrückgabewerte, -parameter und von der Methode ausgelöste Ausnahmen
- Oberklassen
- Rollenenden von Assoziationen, die von dieser Klasse ausgehen

sichtbar sind. Ist dies nicht der Fall, so muss ein entsprechender Import im Paketdiagramm eingefügt werden.

In den beiden folgenden Unterabschnitten, werden die Besonderheiten bei der Umsetzung der Importermittlung für Ecore bzw. Fujaba Modelle näher beleuchtet.

Ermittlung von Import-Beziehungen in Ecore Modellen

Eine weitere *objectContribution* realisiert die Funktion „Update Imports in Package Diagram“. Die zugehörige Action **PackagesDiagramImportsInitialization** im Paket **de.ubt.ai1.packagesdiagram.ecoreimport.actions** instanziiert dann entsprechend der Benutzervorgabe ein Objekt vom Typ **Ecore2ElementImportConverter** oder **Ecore2PackageImportConverter** (beide stammen aus dem Paket **de.ubt.ai1.packagesdiagram.ecoreimport.converter**, siehe Klassendiagramm in Abbildung 5.24). Beide ImportConverter Klassen erben von der abstrakten Klasse **BaseConverter**, die eine abstrakte Methode **convert()** definiert. In den überschriebenen convert-Methoden wird jeweils eine Instanz der Klasse **ImportNotifier** erzeugt. Diese Klasse enthält Methoden zum Traversieren des Ecore Modells mittels eines sogenannten **TreeIterator**s. Dieser Iterator durchläuft das Ecore Modell beginnend von der Wurzel. Für jede Klasse (*EClass*), die in dem Ecore Modell definiert ist, werden nun die Typen sämtlicher Attribute, Rückgabewerte von Methoden, Übergabeparameter von Methoden, Ausnahmen von Methoden und die Supertypen ermittelt.

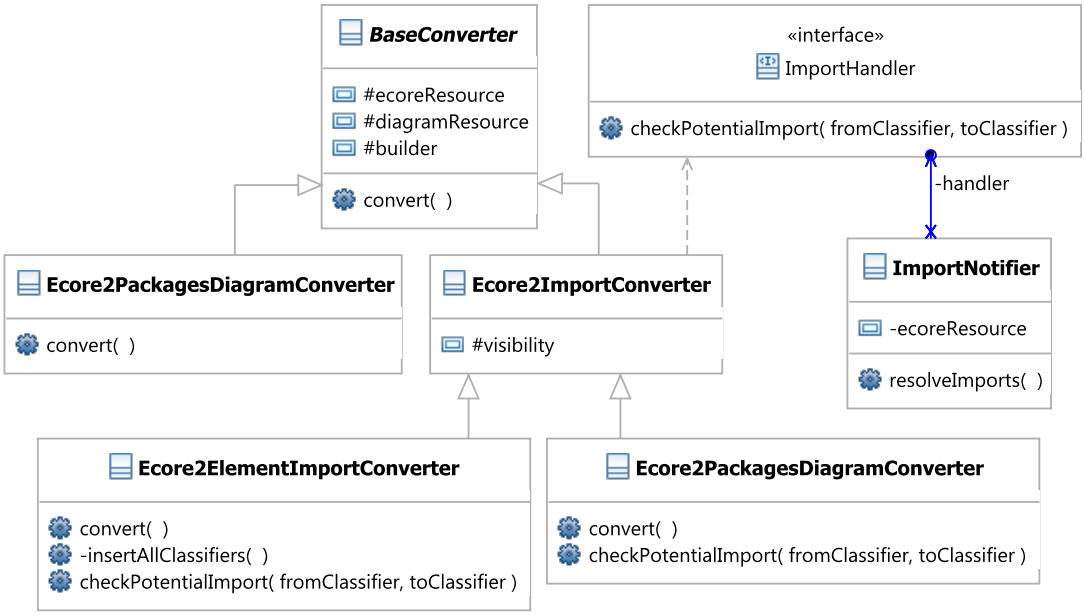

Abbildung 5.24: Klassendiagramm des ImportHandler Moduls.

Abbildung 5.25 zeigt einen Ausschnitt des Objektdiagramms des Ecore Modells. Es werden exemplarisch zwei Methoden der Klasse **ConfigurableRepoServer** (repräsentiert durch das Objekt *EClassImpl*) dargestellt: *checkOut* und *commit*. Beide Objekte sind vom Typ *EOperationImpl*. Für die *commit* Methode wurden außerdem noch die Ausnahme (ItemNotFoundException) und der Übergabeparameter (*EParameterImpl*) dargestellt.

Anschließend wird mit der von der Klasse PackagesDiagram zur Verfügung gestellten Methode isAccessAllowed (siehe Abschnitt 5.5.1) überprüft, ob der Typ auch sichtbar ist. Ist dies nicht der Fall, so wird ein Import entsprechend der Benutzervorgabe erzeugt. Dies bedeutet für den Fall von Paketimporten, dass die umschließenden Pakete der ak-

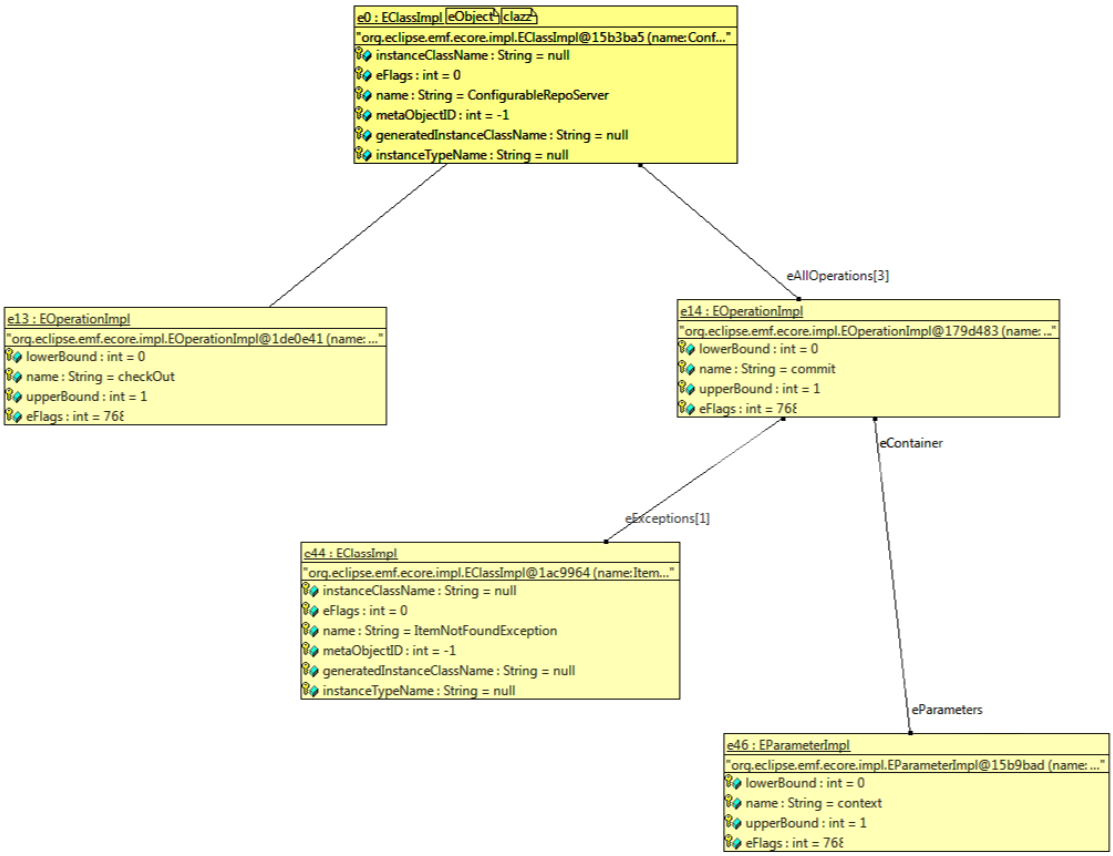

Abbildung 5.25: Objektdiagramm eines Teils eines EMF Modells.

tuellen Klasse und des referenzierten Typs ermittelt werden und ein Paketimport mit der gewünschten Sichtbarkeit zwischen beiden Paketen erzeugt wird. Im Falle von Elementimporten wird ein neuer Import zwischen der Klasse und dem referenzierten Typ erstellt.

Ermittlung von Import-Beziehungen in Fujaba Modellen

Für die Berechnung der benötigten Importe anhand eines Fujaba Modells, wurde die Klasse **FujabaModelImporter** aus dem Paket **de.ubt.ai1.packagesdiagram.fujaba** verwendet. Die verwendeten Methoden wurden mit Hilfe von Storydiagrammen in Fujaba beschrieben. Im Gegensatz zur in Abschnitt 5.6.4 beschriebenen Vorgehensweise, werden nun alle Klassen des Projekts durchlaufen. Für jede Klasse werden die in 5.6.4 aufgelisteten Typen überprüft. Exemplarisch sei hier die Prüfung der Sichtbarkeit von Rollenenden anhand des in Abbildung 5.26 dargestellten Storydiagramms erläutert.

Ausgehend von der übergebenen Klasse **clazz** werden zunächst alle Assoziationen (**UMLAssoc**) ermittelt, an der diese Klasse beteiligt ist. Anschließend wird der Typ (targetClass) des gegenüberliegende Rollenende (rightRole) der Assoziation ermittelt. Ebenso werden die umschließenden Pakete der beiden an der Assoziation beteiligten Klassen

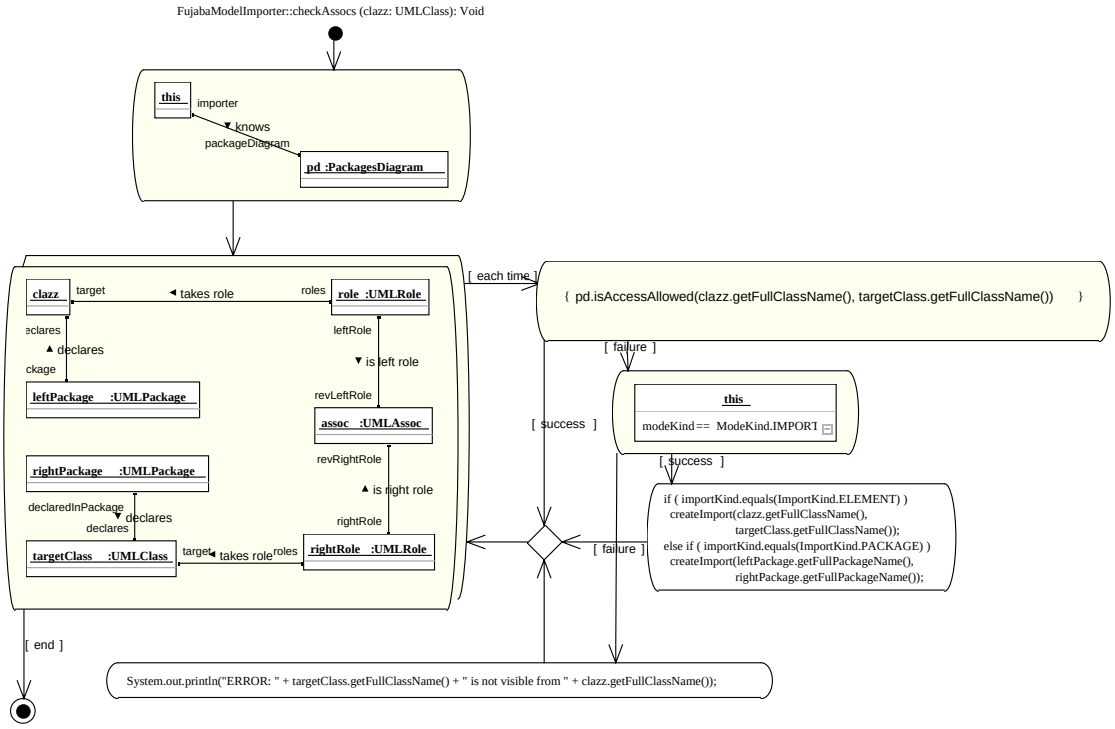

Abbildung 5.26: Storydiagramm zur Überprüfung der Sichtbarkeit von Rollenenden von Assoziationen

ermittelt (leftPackage und rightPackage). Nun wird mit Hilfe der Methode isAccessAllowed überprüft, ob die „Zielklasse“ der Assoziation von der Quelle aus sichtbar ist. Ist dies nicht der Fall, so wird ein Import des vom Benutzer festgelegten Typs erzeugt.

Zusätzlich zu den in Abschnitt 5.6.4 erwähnten Klassendiagrammelementen, müssen im Falle von Fujaba auch noch Objektinstanziierungen in allen Storydiagrammen überprüft werden. Dazu werden alle Storydiagramme zu Methoden der aktuellen Klasse ausgewertet. Ein Storydiagramm besteht aus einer Start- und Stopaktivität und beliebig vielen sogenanten **UMLStoryActivity** Instanzen. Ist diese Activität durch ein **UMLStoryPattern** beschrieben, so müssen alle in diesem Storypattern vorkommenden Objekttypen auf die Sichtbarkeit hin überprüft und ggf. ein Import erzeugt werden.

5.6.5 Validierung

Validierung des Ecore Modells

Da das Ecore Modell und das Paketdiagramm unabhängig voneinander modifiziert werden können, ist es notwendig, Mechanismen zur Gültigkeitsprüfung bereit zu stellen. Dabei soll sichergestellt werden, dass das im Ecore Modell dargestellte System, den im Paketdiagramm spezifizierten Vorgaben hinsichtlich der Sichtbarkeitsbeziehungen der Pakete entspricht. Die Validierung liefert Informationen, ob die vordefinierte Abhängig-

keit der Pakete untereinander erfüllt ist, oder ob im Klassendiagramm auf nicht sichtbare Elemente in anderen Paketen zugegriffen wird. Die definierten Sichtbarkeiten können im Klassendiagramm auf mehrere Arten verletzt werden:

1. durch Erben von nicht sichtbaren Elementen
2. durch Assoziationen auf nicht sichtbare Elemente
3. durch Verwendung von nicht sichtbaren Elementen als Attribute in Klassen
4. durch Verwendung von nicht sichtbaren Elementen als Rückgabewert von Methoden
5. durch Verwendung von nicht sichtbaren Elementen als Übergabeparameter in Methoden
6. durch Verwendung von nicht sichtbaren Elementen als Ausnahmetyp von Methoden

Der Ecore Validierungsmechanismus ist in einem separaten Plugin realisiert. Sämtliche benötigte Klassen befinden sich im Paket **de.ubt.ai1.packagesdiagram.validation** bzw. dessen Subpaketen. Bei der Implementierung wird auf das EMF Validation Framework [SBPM09] zurückgegriffen, da das zu prüfende Modell als Ecore Modell vorliegt.

Die Notation von Bedingungen und Invarianten in EMF lehnt sich stark an deren Definition in der UML an. Dort ist eine Bedingung ein Ausdruck, der zu einem gewissen Zeitpunkt gültig sein muss, während eine Invariante eine Behauptung darstellt, die immer erfüllt sein muss. Von Haus aus unterstützt das Rahmenwerk Java und OCL zur Definition von Validierungsregeln. Desweiteren können weitere Validierungssprachen eingebunden werden. Zur Laufzeit kann die Validierung auf zwei Arten erfolgen: statisch, also durch den User angestoßen (als sog. *Batch-Job*), oder dynamisch und damit kontinuierlich (*live*).

Eine Validierungsregel bezieht sich immer sowohl auf einen bestimmten Typ von Modellelement als auch auf ein spezifisches Attribut. Wird die Regel bei einer Instanz dieses Typs verletzt, verwendet der Editor diese Information, um das entsprechende Attribut für den Benutzer erkennbar in der graphischen Benutzeroberfläche zu markieren. Das Rahmenwerk erlaubt es außerdem, den Grad der Verletzung zu spezifizieren (z.B. „Warning“ oder „Error“). Ebenfalls wird eine spezielle Sicht in Eclipse bereitgestellt, die das Modellelement, den Ersteller des Modellelements, den Grad der Verletzung sowie eine textuelle Beschreibung der Verletzung anzeigt.

Um die EMF Validierung per OCL für den Paketdiagrammeditor nutzen zu können, muss das Rahmenwerk entsprechend angepasst bzw. erweitert werden. Über den Erweiterungspunkt *org.eclipse.emf.validation.contraintProviders* ist es möglich, die spezifischen Validierungsregeln zu definieren. Um den Validierungsregeln Zugriff auf das Paketdiagramm Modell zu ermöglichen, wird der OCL Umgebung das Wurzelelement

(**PackagesDiagram**) per Parameter übergeben. Desweiteren wird eine Methode *qualifiedName()* definiert, die für jedes zu überprüfende Objekt den voll qualifizierten Namen zurückliefert. Die Implementierung erfolgt unter Verwendung der in Abschnitt 5.5.3 vorgestellten Hilfsklasse. Diese projektspezifischen Erweiterungen wurden mittels der Klassen **CustomEcoreEnvironment**, **CustomEcoreEnvironmentFactory** und **CustomEcoreEvaluationEnvironment** im Paket **de.ubt.ai1.packagesdiagram.validation.ecore** realisiert. Die angepasste OCL Umgebung wird mittels der Factory Klasse initialisiert, die selbst wiederum in dem Kommando **ModelValidation** des Pakets **de.ubt.ai1.packagesdiagram.validation.ecore.commands** instanziiert wird. Das Kommando **ModelValidation** wird durch die Aktion „Validate Model...“ aus dem Kontextmenü des Ecore Baumeditors von Eclipse aufgerufen. Der Benutzer muss lediglich das Paketdiagramm Modell, auf dessen Basis die Validierung durchgeführt werden soll, per Dialog auswählen.

Am Ende der Validierung wird entweder eine Erfolgsmeldung ausgegeben, oder eine Liste mit sämtlichen Modellelementen angezeigt, die die im Paketdiagramm vorgegebenen Sichtbarkeitsregeln verletzen. Die entsprechenden Markierungen mit Verweisen auf die zugehörigen Ecore Modellelemente werden in die sogenannte *Problems View* von Eclipse eingetragen.

Per Doppelklick auf einen Eintrag in der Liste oder in der *Problems View* werden die Modellelemente, die die Sichtbarkeitsregeln verletzen, in der Baumansicht des EMF Editors markiert, wie Abbildung 5.27 zeigt.

Verwendete OCL Regeln

Zur Sicherstellung der Konformität des zu prüfenden Ecore Modells und des zugrunde liegenden Paket Modells, müssen alle Instanzen der relevanten Typen in dem Ecore Modell überprüft werden. Dies sind

- EClass
- EAttribute
- EParameter
- EOperation
- EReference

Für diese Typen werden über den oben erwähnten Erweiterungspunkt des EMF Validation Framework (*org.eclipse.emf.validation.constraintProviders*) nun die entsprechenden Validierungsregeln mittels OCL-Ausdrücken definiert. In jedem dieser Ausdrücke werden die möglichen Kombinationen von Ausgangstyp bzw. -paket und Zieltyp bzw. -paket auf die Sichtbarkeit hin überprüft. Damit eine solche Regel erfolgreich angewendet werden kann, muss mindestens eine dieser Kombinationen ein positives Resultat zurückliefern. Zur Ermittlung der Sichtbarkeit wird wiederum auf die Methode **isAccessAllowed** der Klasse **PackagesDiagram** zurückgegriffen (siehe Abschnitt 5.5.1). Die Methode

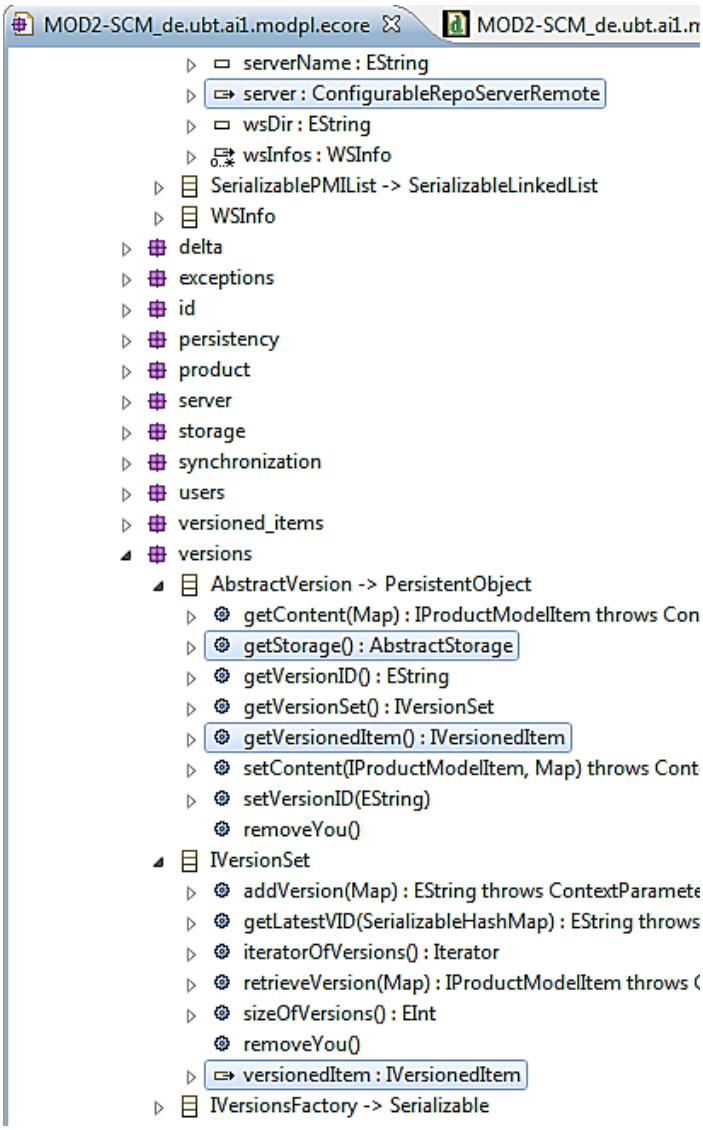

Abbildung 5.27: Hervorheben von Elementen, die die Sichtbarkeitsregeln verletzen.

```
1  packageModel.isAccessAllowed(
2   <Ausgangstyp>.qualifiedName(),
3   <Zieltyp>.qualifiedName() )
4  or
5  packageModel.isAccessAllowed(
6   <Ausgangstyp>.ePackage.qualifiedName(),
7   <Zieltyp>.qualifiedName() )
8  or
9  packageModel.isAccessAllowed(
10  <Ausgangstyp>.qualifiedName(),
11  <Zieltyp>.ePackage.qualifiedName() )
12 or
13 packageModel.isAccessAllowed(
14  <Ausgangstyp>.ePackage.qualifiedName(),
15  <Zieltyp>.ePackage.qualifiedName() )
```

Listing 5.5: OCL Regeln für die möglichen Kombinationen von Ausgangstyp / -paket und Zieltyp / -paket

benötigt jeweils den voll qualifizierten Namen des Ausgangs- bzw. des Zielnamensraums. Da dem EMF Validation Framework über die oben angesprochenen Erweiterungen die Instanz des verwendeten Paketdiagramm-Modells, sowie eine Methode zur Ermittlung des voll qualifizierten Namens eines Typs oder Pakets zur Verfügung steht, lassen sich die OCL-Ausdrücke wie in Listing 5.5 dargestellt formulieren. Zum umschließenden Paket des jeweiligen Typs navigiert man durch den Ausdruck *ePackage*.

Für die möglichen Werte für die Ausgangs- bzw. Zieltypen ergibt sich folgende Tabelle (siehe Tabelle 5.1) abhängig von den zu betrachtenden EMF Typen.

Validierung des Fujaba Modells

Neben den Methoden zum Erzeugen und Abgleichen der beiden Modelle enthält der Integrator auch Mechanismen zur Validierung eines Fujaba Modells anhand eines Paketdiagramms. Der Benutzer kann somit jederzeit überprüfen, ob das feingranulare Modell den Vorgaben hinsichtlich Kopplung und Abhängigkeiten der in der Disziplin „Modellieren im Großen“ erstellten Spezifikation entspricht. Da die Prüfung der Validität auf einer Instanz des Fujaba-Metamodells durchgeführt wird, wurden die entsprechenden Methoden vollständig in Fujaba modelliert. Abbildung 5.28 zeigt einen Ausschnitt des Klassendiagramms. Die zentrale Instanz des Integrators PackagePlugin besitzt zur Laufzeit eine Instanz der Klasse FujabaModelValidator. Dort sind die einzelnen Methoden zur Validitätsprüfung definiert. Die Überprüfung der Einhaltung der Sichtbarkeitsregeln erfolgt immer pro Klasse. Alle Kindelemente einer Klasse müssen somit auf gültige Sichtbarkeit überprüft werden. Dies sind (siehe auch Abbildung 5.23):

- **Attribute** der Klasse
- **Methoden** der Klasse

EMF Typ	Ausgangstyp	Zieltyp
EReference	self.eContainingClass	self.eReferenceType
EOperation	self.eContainingClass	self.eType
EOperation	self.eContainingClass	self.eExceptions-> forAll(exception | <Ausdruck 1>)
EParameter	self.eOperation. eContainingClass	self.eType
EAttribute	self.eContainingClass	self.eAttributeType
EClass	self	self.eSuperTypes-> forAll(superType | <Ausdruck 1>)

Tabelle 5.1: Ausgangs und Zieltypen für die jeweiligen in den OCL Bedingungen geprüften EMF Typen.

- **Rollen**, die Enden von Assoziationen sind, die an dieser Klasse beginnen
- **Vererbungskanten**, die von dieser Klasse ausgehen und an Oberklassen enden

Hierbei ist bei den Methoden zu beachten, dass hierfür Rückgabetypen, Typen von Parametern und Ausnahmen sowie auch die in den Storydiagrammen vorkommenden Objekttypen überprüft werden.

Es wird jeweils anhand des Paketdiagramms überprüft, ob die entsprechenden Typen von der aktuellen Klasse aus sichtbar sind. Ist dies nicht der Fall, so wird das entsprechende Element (in diesem Fall ein Objekt vom Typ **UMLIncrement**) in die Liste **invalidItems** eingefügt, um dann bei Bedarf an der Benutzeroberfläche markiert zu werden. Abbildung 5.29 zeigt das Story-Diagramm für die Methode zur Sichtbarkeitsprüfung von Rollen einer Klasse. Ausgehend von der übergebenen Klasse **clazz** wird über alle der Klasse zugeordneten Rollen **role** vom Type **UMLRole** iteriert. Eine Rolle ist immer Bestandteil einer Assoziation **UMLAssoc**, die ihrerseits immer genau zwei Rollen (**UMLRole**) besitzt. Somit kann der Typ der gegenüberliegenden Rolle ermittelt werden. Bei der Übersetzung in Quelltext wird dieser Typ der gegenüberliegenden Rolle als Attribut mit den entsprechenden Zugriffsmethoden in die aktuelle Klasse generiert, daher muss dieser bei der Sichtbarkeitsprüfung ermittelt werden. Nun wird jeweils geprüft, ob der Typ der gegenüberliegenden Rolle von der aktuellen Klasse aus sichtbar ist, oder ob vom umschließenden Paket der aktuellen Klasse das umgebende Paket des gegenüberliegenden Typs sichtbar ist. Ist dies nicht der Fall, so wird das entsprechende Element, in diesem Fall die gegenüberliegende Rolle rightRole, in die Liste invalidItems eingefügt und an der Benutzeroberfläche in roter Farbe dargestellt. Der Benutzer hat nun die Möglichkeit, sich für jedes Element, das die Sichtbarkeitsregeln verletzt, einen Lösungsvorschlag anzeigen zu lassen und auch auszuwählen. Anhand der Benutzerauswahl wird dann ein entsprechender Import im Paketdiagramm automatisch erzeugt und anschließend eine erneute Validierung durchgeführt. Der Benutzer kann so nun sukzessive alle Störungen beseitigen.

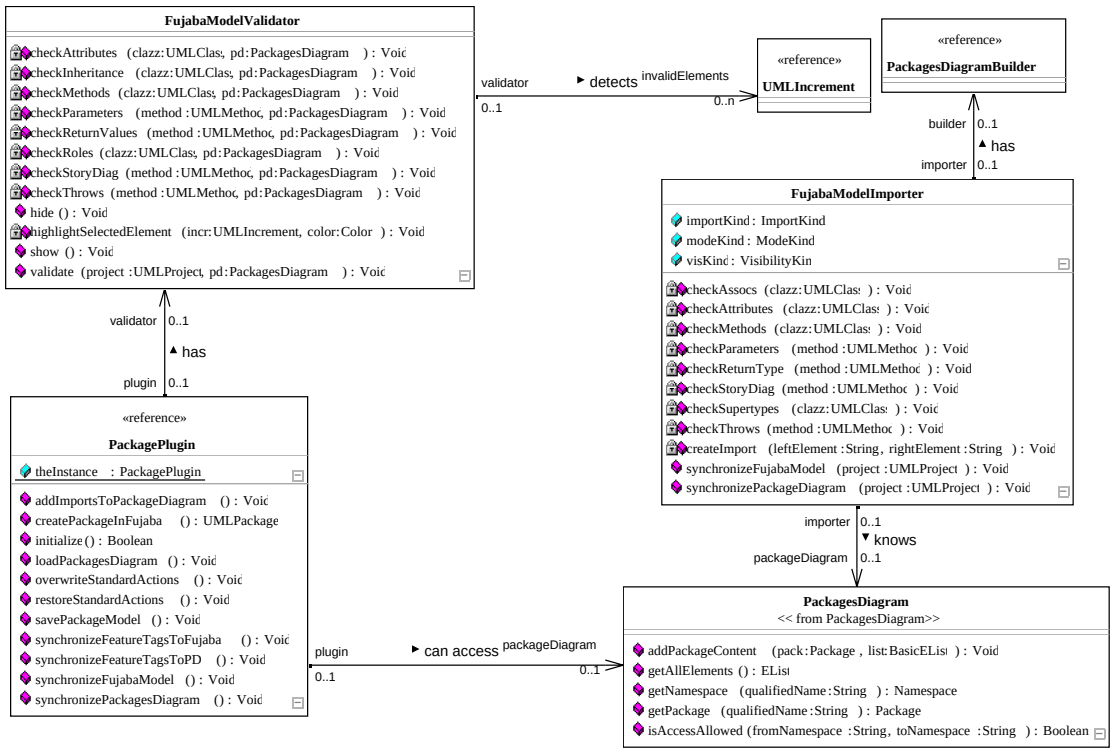

Abbildung 5.28: Das Klassendiagramm des Integrators.

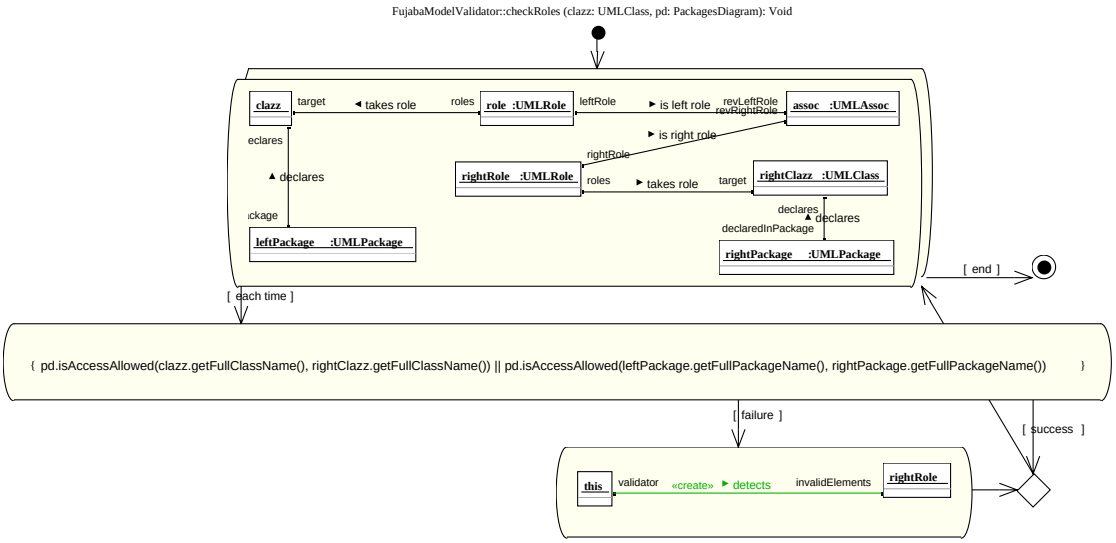

Abbildung 5.29: Das Story-Diagramm zur Überprüfung der Sichtbarkeiten von Rollen einer Klasse.

5.6.6 Restriktives Bearbeiten

Der Integrator bietet zusätzlich noch Funktionen, um bei Bedarf die Vorgaben des Paketdiagramms hinsichtlich der Sichtbarkeiten auch während der Bearbeitung des Fujaba Modells in jedem Bearbeitungsschritt zu überwachen. Dazu wurde die Fujaba Benutzeroberfläche erweitert, indem Subklassen der jeweils verwendeten Fujaba Dialoge erstellt wurden, die die zusätzliche Funktionalität bereitstellen. Die vom Connector Plugin bereitgestellten Dialoge befinden sich im Paket **de.ubt.ai1.packagesdiagram.fujaba.gui**. Dialoge für nachfolgend aufgelistete Operationen werden bereitgestellt:

- Erzeugen von Klassen
- Erzeugen von Assoziationen
- Anlegen von Attributen
- Anlegen von Methoden (Rückgabetyp und Übergabetyp)
- Anlegen von Objekten in Storydiagrammen
- Hinzufügen von Klassen zu Klassendiagrammen

Die Funktionalität der Dialoge wurde dahingehend erweitert, dass zum einen der Benutzer die Möglichkeit hat, die Auswahl der angezeigten Typen auf die vom aktuellen Element aus sichtbaren Elemente einzuschränken. Außerdem wird bei Verwendung von nicht sichtbaren Elementen eine entsprechende Warnung angezeigt und der Benutzer kann, um die gewünschte Operation erfolgreich abzuschließen einen entsprechenden Import im Paketdiagramm einfügen (auch hier besteht die Auswahl aus den vier unterschiedlichen in den vorigen Abschnitten erläuterten Importbeziehnungen). Auch hier erfolgt die Ermittlung der sichtbaren Elemente über die vom Paketdiagramm zur Verfügung gestellten und in Abschnitt 5.5.1 erläuterten Methoden.

Neben den oben erwähnten Erweiterungen erlaubt es das Plugin, Klassendiagramme einem Paket zuzuordnen. Diese Option war bereits im Fujaba Metamodell vorhanden, wurde aber von der Oberfläche nicht unterstützt. Aufgrund der Erfahrung, die in zahlreichen mit Fujaba realisierten Projekten gewonnen wurde, ist eine Beschränkung auf ein Klassendiagramm pro implementiertem Paket durchaus sinnvoll und der Übersichtlichkeit des resultierenden Modells zuträglich [BDW08a]. Mit Hilfe dieser Zuordnung ist es dann möglich, die Auswahl der Elemente, die zum Klassendiagramm hinzugefügt werden können, auf die von diesem Paket aus erreichbaren zu beschränken. Natürlich kann die Zuordnung Klassendiagramm - Paket auch wieder aufgehoben werden.

Neben der Anpassung der Dialoge waren auch Änderungen der Aktionen, die diese Dialoge aufrufen notwendig. Fujaba stellt für Plugins XML Deskriptoren bereit, mit deren Hilfe man auf einfache Art und Weise Schaltflächen zur Menüleiste hinzufügen sowie neue Einträge in das Hauptmenü und die Kontextmenüs einfügen kann. Leider

ist es über diesen Mechanismus nicht möglich, bereits bestehende Einträge, wie z.B. die Standardaktionen, zu überschreiben. Daher müssen diese Standardaktionen, wie z. B. das Anlegen von Klassen, Attributen, Methoden, Assoziationen usw., zur Laufzeit ausgetauscht werden, damit die neuen Dialoge angesprochen werden können.

Der Restriktive Bearbeitungsmodus wird nicht erzwungen, sondern kann auf Benutzerwunsch aktiviert bzw. wieder deaktiviert werden.

5.6.7 Synchronisation

Da bei der Entwicklung großer Softwaresysteme während der Implementierungsphase häufig Änderungen auftreten, die auch die Gesamtarchitektur betreffen, ist es nicht ratsam, Paketdiagramme rein als *Forward Engineering* Werkzeug einzusetzen. Die Verwendung als *Reverse Engineering* Hilfsmittel ist ebenfalls abzulehnen, da hiermit, wie bereits in Abschnitt 5.3.2 erläutert, keine Vorabplanung der Architektur möglich ist und die Paketdiagramme lediglich als Dokumentationsinstrument dienen, die den aktuellen Zustand des Modells widerspiegeln. Um den dynamischen Anforderungen während des Softwareentwicklungsprozesses gerecht zu werden, muss vielmehr die Möglichkeit bestehen, an beiden Modellen unabhängige Änderungen vorzunehmen, die auch von unterschiedlichen Benutzern vorgenommen werden können und diese Änderungen in beide Richtungen zu propagieren.

Im Gegensatz zu den Stapelverarbeitungsprozessen im Bereich *Forward* bzw. *Reverse Engineering*, bei denen *Master* der jeweiligen Operation klar festgelegt ist (das Paketdiagramm beim Forward, das Fujaba Modell beim Reverse Engineering), hängt die Wahl des Masters hier von der gewählten Synchronisationsoperation ab. Folglich wurden auch unterschiedliche Methoden implementiert, die den Abgleich zwischen Paketdiagramm und Fujaba Modell jeweils von der ein oder anderen Richtung aus realisieren. Im entsprechenden Synchronisierungsschritt wird nun das jeweilige Quellmodell durchlaufen und für jedes Element wird eine Entsprechung im Zielmodell gesucht. Ist das Element im Zielmodell nicht vorhanden, so wird es dort neu erzeugt. Listing 5.6 zeigt die Methode, die den Abgleich des Fujaba Modells nach Änderungen am Paketdiagramm realisiert. Es werden alle im Paketdiagramm vorhandenen Elemente durchlaufen. Ist das aktuell betrachtete Element ein Paket, so wird anhand des qualifizierten Namens das Pendant in Fujaba gesucht. Existiert dies nicht, so wird es neu erzeugt. Sind im Paketdiagramm auch Elemente in Paketen vorhanden, so wird in Fujaba das entsprechende Klassendiagramm gesucht. Nach den Modellierungsvorschlägen in [BDW08a] wird jedes Klassendiagramm einem Paket zugeordnet und der Name des Klassendiagramms ist gleich dem Paketnamen. Existiert das Klassendiagramm nicht, so wird es entsprechend den Vorgaben angelegt. Anschließend werden in diesem neuen Klassendiagramm die Kindelemente (Klassen, Datentypen, Aufzählungstypen) des Pakets erzeugt. Ist das Klassendiagramm hingegen schon vorhanden, so wird geprüft, ob auch die im Paketdiagramm definierten Elemnte bereits existieren. Ist dies nicht der Fall, so werden sie ebenfalls neu erzeugt.

Anders als beim strikten *Forward* oder *Reverse Engineering* reicht es beim inkremen-

```
1  public void synchronizeFujabaModel() {
2   Iterator<Element> it = packagesDiagram.getAllElements().iterator();
3   ...
4
5   while ( it.hasNext() ) {
6          Element currentElem = it.next();
7          if (currentElem instanceof Package) {
8           Package pack = (Package) currentElem;
9           UMLPackage umlPackage = rootPackage.findPackage(
                NamespaceUtil.getFullyQualifiedName(pack));
10
11          if (umlPackage == null)
12           umlPackage = createPackageInFujaba(pack, packFactory);
13          if (pack.getOwnedTypes().size() > 0) {
14           UMLClassDiagram diag = getDiagram(NamespaceUtil.
                getFullyQualifiedName(pack));
15          if (diag == null) {
16           diag = classDiagFactory.create();
17           diag.setName(NamespaceUtil.getFullyQualifiedName(pack));
18           umlPackage.addToDiagrams(diag);
19          } else
20           diag = (UMLClassDiagram) project.getFirstFromDiagrams();
21          Iterator typeIter = pack.getOwnedTypes().iterator();
22                  while (typeIter.hasNext()) {
23                  Type type = (Type) typeIter.next();
24                  if ( !umlPackage.hasKeyInDeclares(type.getName()) ) {
25                          UMLClass clazz = classFactory.create();
26                          clazz.setDeclaredInPackage(umlPackage);
27                          clazz.setName(type.getName());
28                          clazz.addToDiagrams(diag);
29                  }
30          }
31          }
32  }
33 }
34 }
```

Listing 5.6: Quelltext der Methode zur Synchronisierung des Fujaba Modells nach Änderungen am Paketdiagramm.

tellen *Roundtrip Engineering* nicht aus, nur die neu erzeugten Elemente abzugleichen, es müssen auch alle durchgeführten Löschoperationen synchronisiert werden. Hierfür wird dann nach dem Anlegen der neu hinzugekommenen Elemente ein weiterer Durchlauf gestartet, der im Zielmodell alle Elemente ohne Entsprechung im Quellmodell ermittelt und diese dem Benutzer in einem Auswahldialog anzeigt. Der Benutzer kann nun entscheiden, ob diese Elemente im Zielmodell gelöscht werden sollen.

5.7 Arbeiten mit dem Paketdiagramm Werkzeug

Dieser Abschnitt veranschaulicht die Arbeit mit dem Integrator des Paketdiagramm Werkzeugs für Fujaba. Da das MOD2-SCM Projekt mit Fujaba realisiert wurde, werden sämtliche Anwendungsfälle anhand dieses realen Modells erläutert, wenngleich auch für Ecore Modelle *Forward* und *Reverse Engineering* wie auch die *Validierung* möglich wäre.

5.7.1 Forward Engineering

Soll das Paketdiagramm als reines Werkzeug zur a priori Definition der Software Architektur verwendet werden, so wird das finale Paketdiagramm in ein grobes Gerüst für das feingranulare Modell überführt. Abbildung 5.30 zeigt im linken Bereich einen Architekturentwurf mittels Paketen und deren Abhängigkeiten. Die Unterpakete des Kernpakets **core**, sowie die des Pakets **storage** enthalten außerdem bereits Klassen. Die unterschiedlichen Pakete sind durch öffentliche Paketimporte miteinander verbunden. Aus diesem Paketdiagramm kann nun mittels des *Forward Engineering* Schritts ein grobes Gerüst eines Fujaba Modells zur anschließenden Verfeinerung und Ausimplementierung erstellt werden. Das Ergebnis dieses Schrittes ist im rechten Bereich von Abbildung 5.30 zu sehen. Wie im unteren Bereich zu erkennen ist, wurde die Pakethierarchie entsprechend der Vorgaben im Paketdiagramm in Fujaba nachgebaut. Es werden für alle Pakete, die im Paketdiagramm bereits Elemente enthalten, in Fujaba Klassendiagramme angelegt, die den qualifizierten Namen des jeweiligen Pakets tragen, und die auch diesem Paket zugeordnet sind. Ebenso werden die entsprechenden Klassen, die in den jeweiligen Paketen definiert sind, angelegt und in die korrespondierenden Klassendiagramme eingefügt. Im weiteren Modellierungsprozess können diese Klassen nun mit Attributen, Methoden und Assoziationen verfeinert werden.

5.7.2 Reverse Engineering

Das *Reverse Engineering* dient dazu, aus einem bereits bestehenden Fujaba Modell ein Paketdiagramm abzuleiten, das optional auch noch berechnete Importbeziehungen enthalten kann. Dieses Modell kann entweder zu Dokumentationszwecken oder als Hilfsmittel zur Übersicht über die implementierte Architektur verwendet werden. Selbstverständlich ist es auch möglich mit Hilfe des in Abschnitt 5.6.7 vorgestellten *Roundtrip Engineering* ab diesem Zeitpunkt Änderungen an der Architektur im Paketdiagramm vorzunehmen und diese in das Fujaba Modell zu propagieren. Auch kann es im weiteren Verlauf als Grundlage für die Validierung (5.6.5) dienen. In Abbildung 5.31 ist links ein

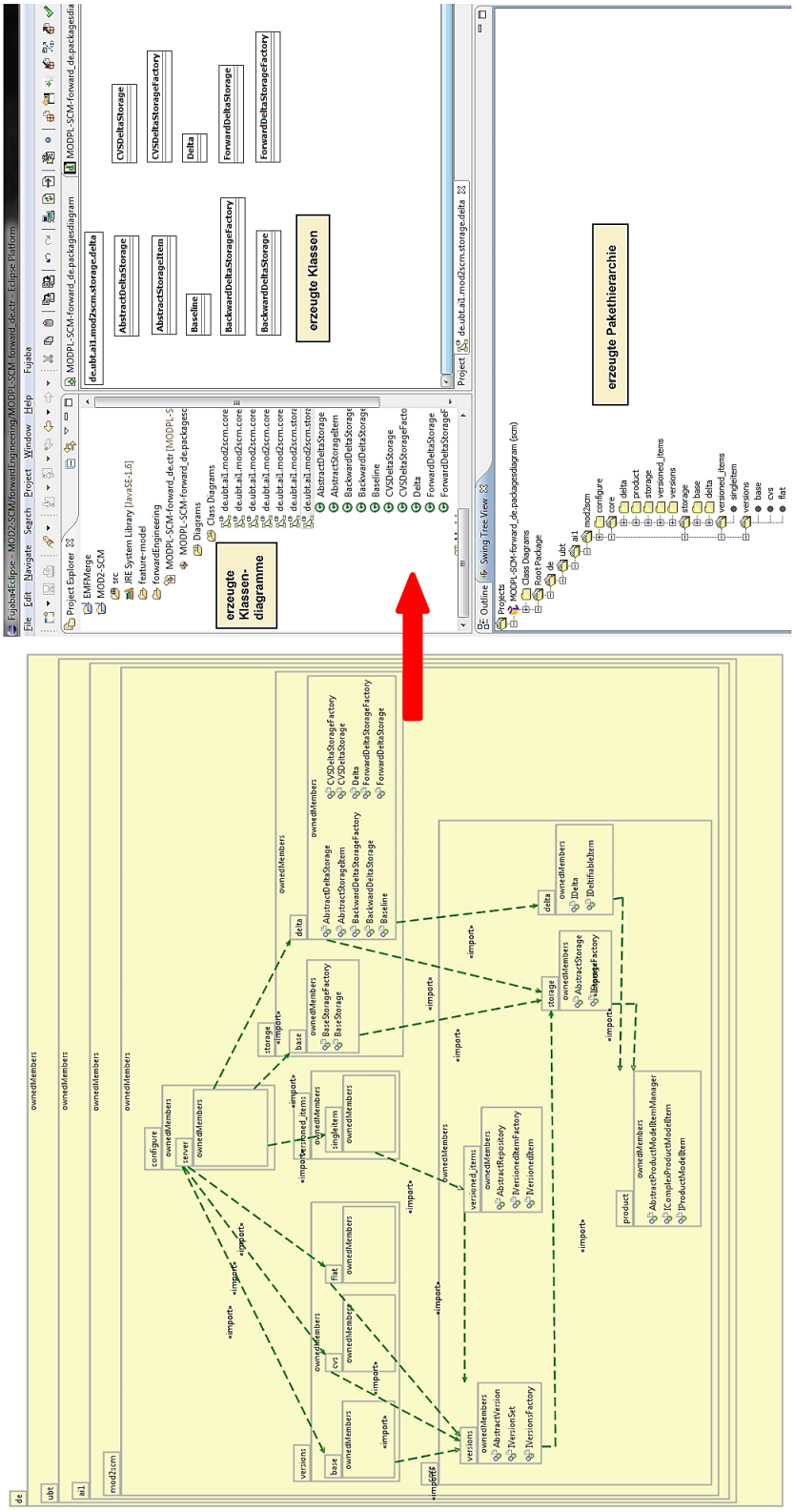

Abbildung 5.30: Erstellen eines groben Gerüsts eines neuen Fujaba Modells anhand eines Paketdiagramms im *Forward Engineering* Schritt.

Ausschnitt des Fujaba Modells zu sehen, aus dem mittels der *Reverse Engineering* Funktion des Integrators die Pakethierarchie exportiert wurde, sowie auch die Sichtbarkeitsbeziehungen zwischen den Paketen mit Hilfe von öffentlichen Paketimporten berechnet wurden.

5.7.3 Validierung

Um jederzeit die Konformität des Fujaba Modells mit der im Paketdiagramm definierten Architektur zu prüfen, hat der Modellierer die Möglichkeit, das Fujaba Modell zu validieren. Hier werden, wie in Abschnitt 5.6.5 erläutert, für jede Klasse die Sichtbarkeitsregeln angewendet und etwaige Regelverletzungen durch graphische Hervorhebung der betroffenen Elemente angezeigt. Abbildung 5.32 zeigt das Ergebnis der Validierung des Fujaba Modells nach einer Änderung am Paketdiagramm. Im aktuellen Paketdiagramm wurden gegenüber dem Diagramm, das in Abbildung 5.30 dargestellt wurde, die öffentlichen Paketimporte von **storage.delta** zu **core.storage** sowie von **core.delta** zu **core.product** entfernt. Anschließend wurde das Fujaba Modell validiert und alle Elemente, die die im Paketdiagramm festgelegte Sichtbarkeitsdefinition verletzen, wurden farblich hervorgehoben. Im Klassendiagramm zum Paket storage.delta sind folgende Regelverletzungen zu beobachten:

- Die Klasse **AbstractDeltaStorage** erbt von einer nicht sichtbaren Klasse (**AbstractStorage**)
- **AbstractDeltaStorage** und alle davon abgeleiteten Unterklassen benutzen das nicht sichtbare Element **IProductModelItem** als Methodenparameter
- Die Klasse **Baseline** hat das nicht sichtbare Interface **IProductModelItem** als Rollenende einer unidirektionale Assoziation
- Das Interface **IDeltifiableItem** erbt vom nicht sichtbaren Interface **IProductModelItem**

Der Modellierer hat nun die Möglichkeit, die Modelle entweder manuell so anzupassen, dass die Validität wieder hergestellt wird, oder er kann auch die *Quick Fix* Funktion des Integrators wählen. Dabei muss lediglich der zu verwendende Importtyp ausgewählt werden, der im Paketdiagramm dann automatisch eingefügt wird, um die Konformität des Fujaba Modells mit dem Paketdiagramm zu gewährleisten.

5.7.4 Restriktive Bearbeitung

Sobald in Fujaba das entsprechende Paketdiagramm geladen wurde, kann der Modellierer den restriktiven Bearbeitungsmodus aktivieren. In diesem Modus wird die Sichtbarkeit bei den in Abschnitt 5.6.6 aufgelisteten Editieroperationen geprüft. Wird eine Verletzung der Sichbarkeitsregeln festgestellt, so wird die Editieroperation unterbrochen und der Benutzer kann wählen, ob die Operation abgebrochen, oder ob sie nach Einfügen eines Imports im Paketdiagramm abgeschlossen werden soll. Abbildung 5.33

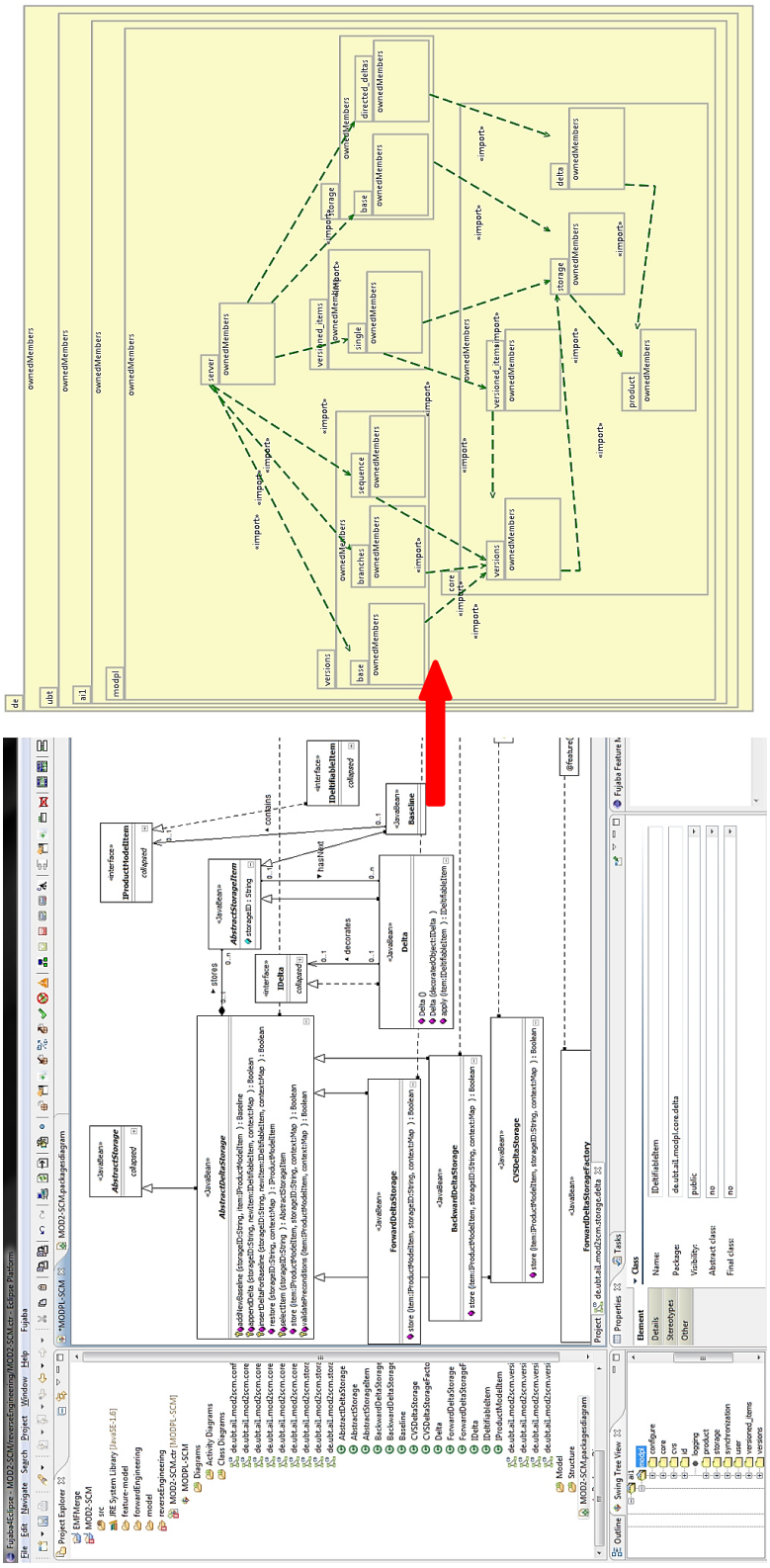

Abbildung 5.31: Erstellen eines Paketdiagramms anhand eines bereits bestehenden Fujaba Modells im *Reverse Engineering* Schritt.

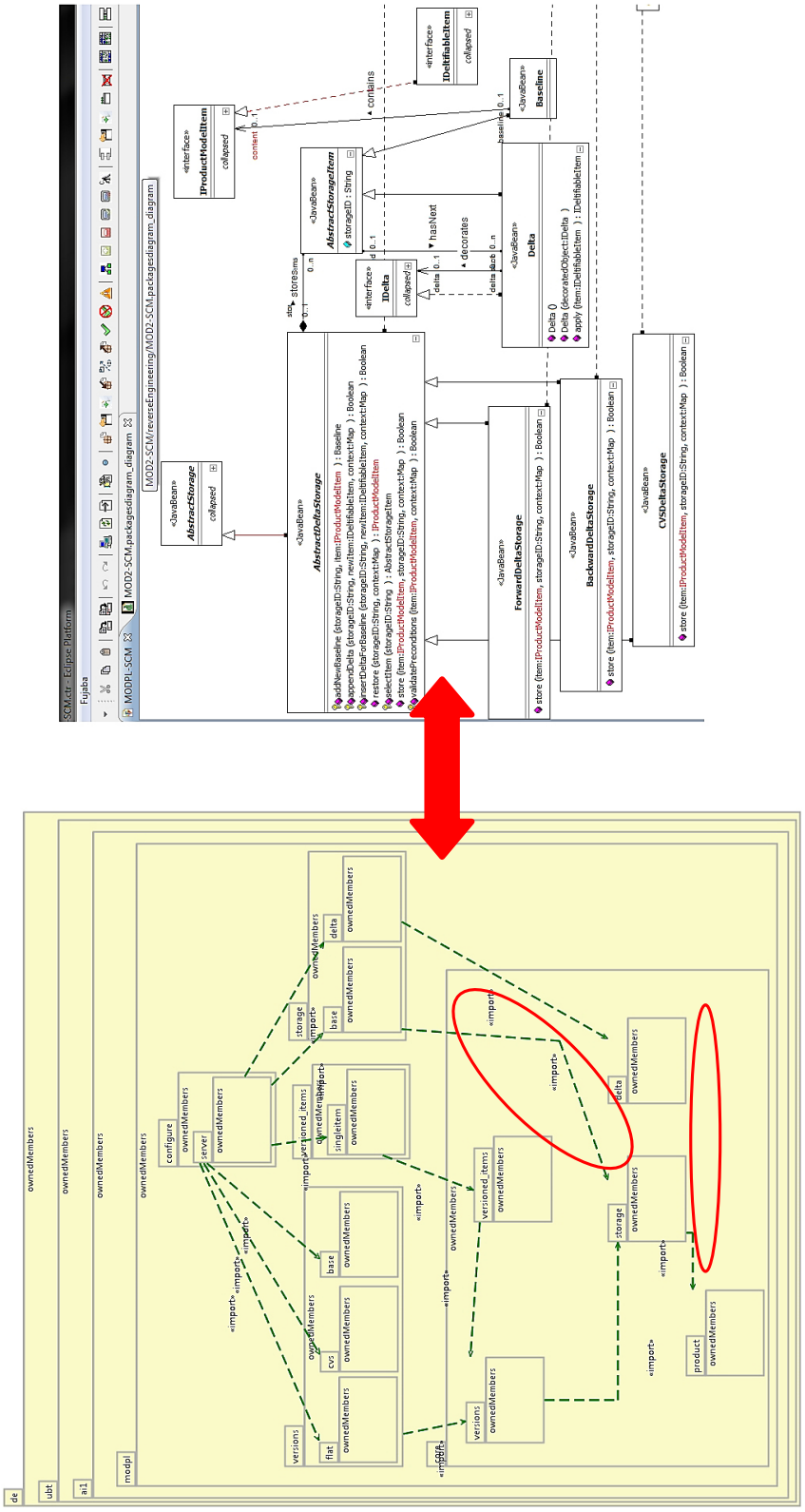

Abbildung 5.32: Validierung des Fujaba Modells nach Änderungen am Paketdiagramm.

zeigt links oben einen Ausschnitt des Paketdiagramms, das die Abhängigkeiten der Kernmodule des MOD2-SCM Systems widerspiegelt. Rechts oben ist ein Ausschnitt aus dem Fujaba Klassendiagramm zu sehen, das die Klassen **AbstractRepository** aus dem Paket **core.versioned_items** und **AbstractStorage** aus dem Paket **core.storage** zeigt. Der restriktive Bearbeitungsmodus ist aktiviert (zu erkennen an der rot eingerahmten gedrückten Schaltfläche). Nun soll eine gerichtete Assoziation von **AbstractStorage** zu **AbstractRepository** erstellt werden. Da zwischen den umschließenden Paketen keine Importbeziehung definiert ist und auch kein Elementimport zwischen beiden Klassen besteht, muss der Benutzer nun den entsprechenden Importtyp wählen, der automatisch im Paketdiagramm eingefügt wird, bevor diese Operation erfolgreich abgeschlossen werden kann. Weiterhin hat der Benutzer die Möglichkeit, die Operation abzubrechen. In diesem Fall wird im Klassendiagramm keine Assoziation erstellt, und auch das Paketdiagramm wird nicht verändert. Entscheidet sich der Benutzer, wie in Abbildung 5.33 dargestellt, für einen öffentlichen Paketimport, so wird dieser im Paketdiagramm neu erstellt (siehe Bild links unten) und anschließend wird die Assoziation im Fujaba Klassendiagramm eingefügt (Bild rechts unten).

5.7.5 Synchronisation

Exemplarisch wird hier die Synchronisation des Fujaba Modells nach einer Änderung des Paketdiagramms beschrieben. Die Synchronisation des Paketdiagramms nach einer Änderung in Fujaba verläuft nach dem gleichen Schema, so dass sie hier nicht weiter ausgeführt wird. Abbildung 5.34 zeigt ein Fujaba Modell vor und nach einer Änderung am zugehörigen Paketdiagramm, die per Synchronisation in das Fujaba Modell propagiert wurde. Im Vergleich zu dem in Abbildung 5.30 dargestellten Paketdiagramm, auf dem das in der Abbildung oben dargestellte Fujaba Modell basiert, wurde das Paket **versions.base** gelöscht und ein neues Paket **storage.complex** mit der Klasse **ComplexStorage** eingeführt. Der Synchronisationsmechanismus legt nun die neu zum Paketdiagramm hinzugekommenen Elemente im Fujaba Modell entsprechend an. Anschließend werden im Fujaba Modell alle Pakete ermittelt, für die es keine Korrespondenzen im Paketdiagramm gibt. Der Benutzer kann nun aus der Liste wählen, ob diese Pakete mit allen darin enthaltenen Klassen gelöscht werden sollen. Im vorliegenden Beispiel wurde einerseits das Paket versions.base mit dem gesamten Inhalt gelöscht, andererseits wurde das Paket storage.complex neu angelegt. Desweiteren wurde darin die Klasse ComplexStorage erzeugt und ein Klassendiagramm, das die neu erzeugte Klasse und das Paket enthält erstellt.

5.8 Bewertung und Vergleich mit bestehenden Paketdiagrammeditoren

5.8.1 Fujaba / Moflon

Eine Unterstützung für Paketdiagramme ist in Fujaba selbst nur rudimentär vorhanden. Das Metamodell enthält zwar entsprechende Strukturen (**UMLPackage**), jedoch fehlen

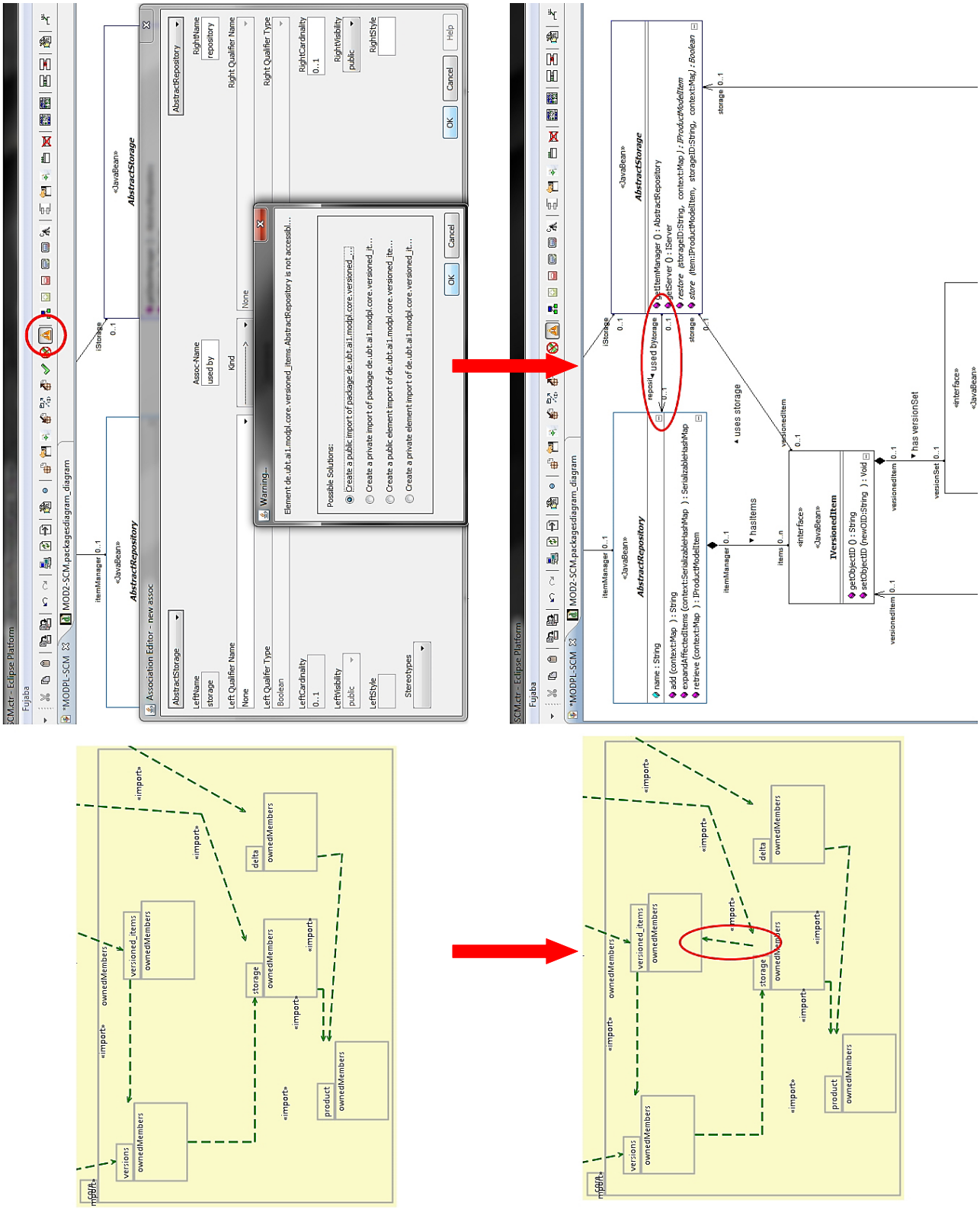

Abbildung 5.33: Restriktive Bearbeitung: Einfügen einer gerichteten Assoziation und Auswirkung auf das Paketdiagramm.

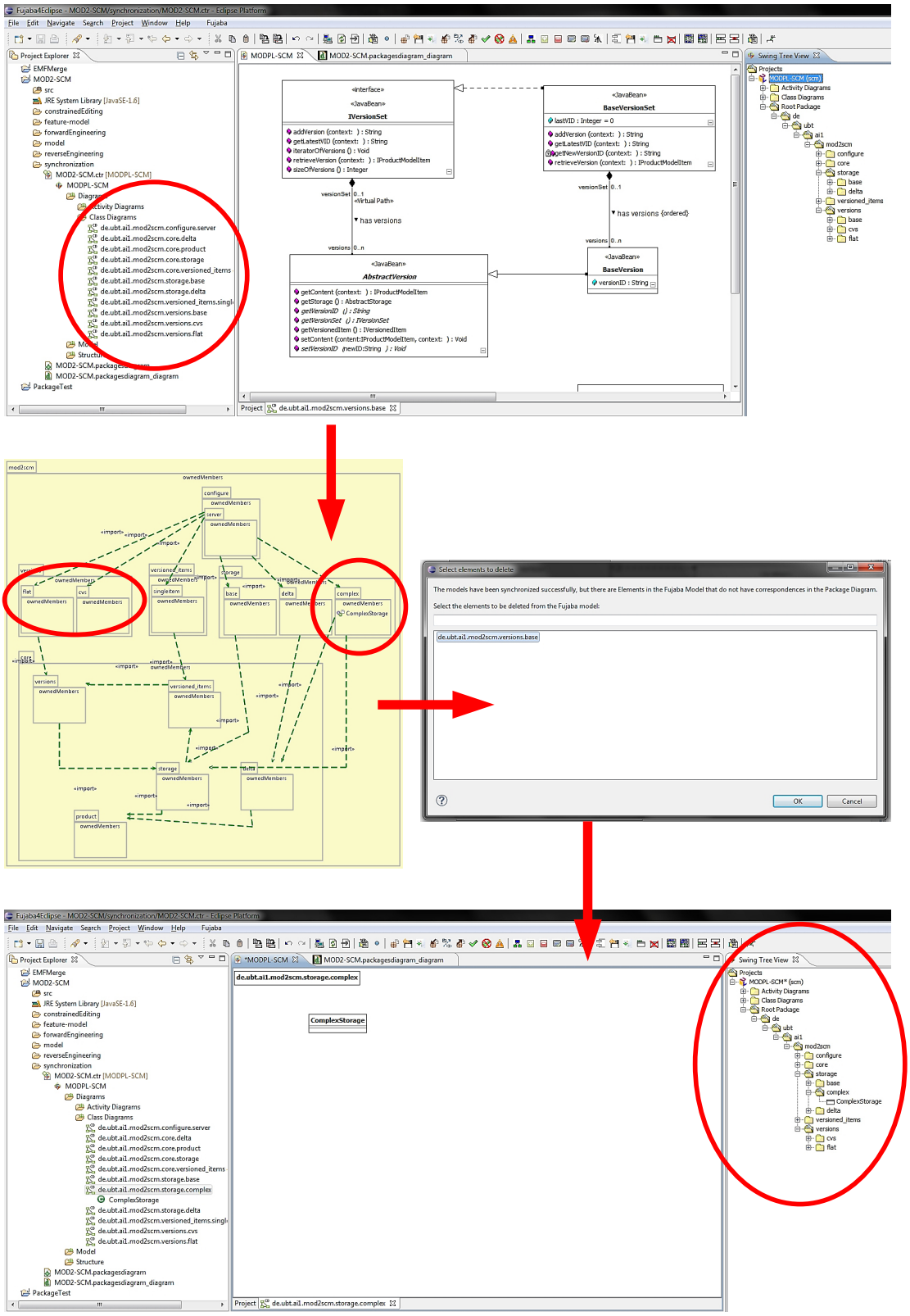

Abbildung 5.34: Synchronisation: Änderung des Paketdiagramms und anschließender Abgleich in Fujaba.

die zugehörigen Mechanismen, um die in der UML definierten Importbeziehungen darzustellen. Entsprechend spartanisch sind die Möglichkeiten, Paketdiagramme zu erstellen. Es besteht lediglich die Möglichkeit, Pakete auf der obersten Ebene zu zeichnen. Eine Verschachtelung von Paketen ist ebenso nicht implementiert, wie die Spezifizierung von Importbeziehungen zwischen den erstellten Paketen. Auch werden die angelegten Pakete im weiteren Modellierungsprozess nicht verwendet, beispielsweise beim Anlegen von Klassen. Unter Fujaba4EclipseSwingUI ist der in Fujaba enthaltene Paketdiagrammeditor nicht funktionsfähig.

Moflon[10][AKRS06], ein Fujaba Derivat, das MOF 2.0 [OMG06a] als Metamodell verwendet und Tripel Graph Grammatiken benutzt, bietet gegenüber Fujaba eine bessere Unterstützung von Paketdiagrammen an. Abbildung 5.35 zeigt ein Modell, das das Paket **de.ubt.ai1.mod2scm.core** in Moflon realisiert. Ein striktes Forward Engineering wird hierbei erzwungen. Das bedeutet, dass nur in bereits vorhandenen MOF-Paketen neue Elemente erstellt werden können. Bei Fujaba hingegen kann durch Definition einer Klasse gleichzeitig ein neues umschließendes Paket spezifiziert werden. Moflon erlaubt dem Modellierer die Erstellung von öffentlichen Paketimporten im graphischen Editor. Die Sichtbarkeit der so erzeugten Importe kann im Nachhinein mittels eines Eigenschaftendialogs noch auf privat geändert werden. Bei der Modellierung werden, ähnlich wie bei der Verwendung von Fujaba mit dem Paketdiagramm-Plugin (siehe vorheriger Abschnitt), nur die vom jeweiligen Element aus sichtbaren Typen angezeigt. Wird beispielsweise eine Assoziation zu einem nicht sichtbaren Element angelegt, so erzeugt Moflon automatisch einen privaten Elementimport. Leider wird dieser Elementimport in keinem Diagramm angezeigt und bleibt somit dem Benutzer verborgen. Auch hat der Benutzer keinerlei Einfluß auf den Typ des erzeugten Imports. Weiterhin gibt es keine Möglichkeit das Modell zu einem beliebigen Zeitpunkt zu validieren. Die Einschränkung der Anzeige in Auswahldialogen auf die sichtbaren Elemente lässt sich im Gegensatz zu der hier vorgestellten Integratorlösung für Fujaba auch nicht deaktivieren.

5.8.2 EMF / UML Tools

Ecore Diagramm Editor

Das EMF Rahmenwerk bietet neben dem Baumeditor für Ecore Modelle auch noch einen graphischen Editor an. Ausgehend von einer Datei mit der Endung **.ecore** kann ein sog. *Ecore Diagram* initialisiert werden. Dieser Diagrammeditor stellt Pakete und Klassen graphisch als Knoten dar. Beziehungen (wie z. B. Vererbungen oder Assoziationen) zwischen Klassen werden wie in anderen einschlägigen CASE Werkzeugen durch unterschiedliche Linien dargestellt. Da das EMF Metamodell keine Importe von Paketen oder Elementen vorsieht, ist es folglich auch mit dem graphischen Editor nicht möglich, derartige Beziehungen zwischen Paketen und Elementen zu modellieren. Pakete in EMF werden ähnlich wie Pakete in der Programmiersprache Java gehandhabt: Sie dienen zur

[10] **http://www.moflon.org**, Version 1.4.0 wurde am 23.11.2009 veröffentlicht. Mit diesem Softwarestand wurde der Vergleich mit dem in dieser Arbeit vorgestellten Paketdiagrammeditor durchgeführt.

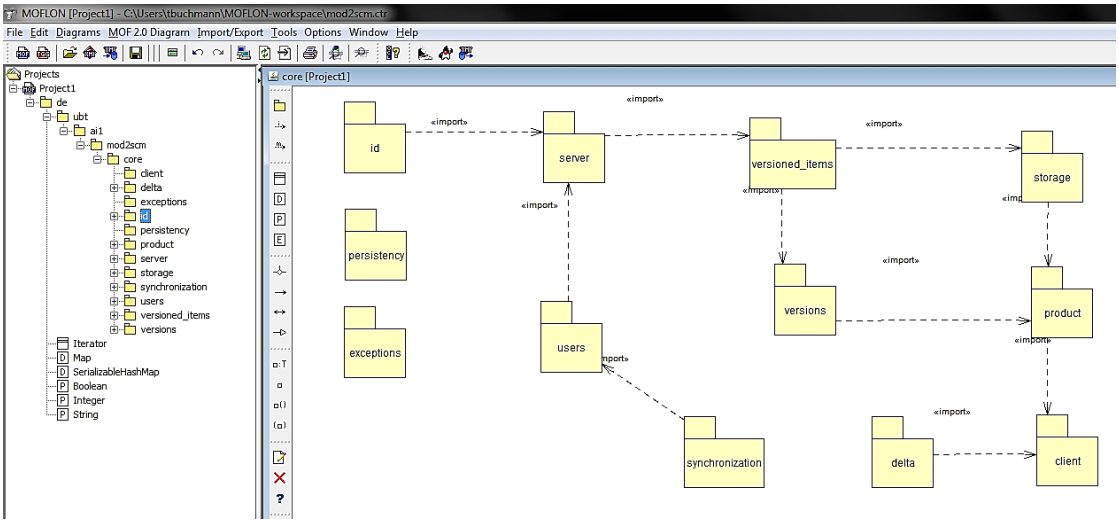

Abbildung 5.35: MOF-Paketdiagramme in Moflon

groben Gruppierung von Klassen und Datentypen.

Eclipse UML2

Wie der Name schon suggeriert, sind die UML2 Tools Teil des Eclipse Model Development Tools (MDT) Projektes. Sie enthalten GMF basierte Editoren zur Darstellung und Bearbeitung von UML Modellen und basieren auf dem MDT/UML2 Metamodell[11]. Hierbei wurde das UML2 Metamodell mit Hilfe von EMF modelliert. Es sind Editoren für folgende Diagrammarten verfügbar:

- Klassendiagramme
- Kompositionsstrukturdiagramme
- Komponentendiagramme
- Verteilungsdiagramme
- Diagramme zur Definition von Profilen
- Aktivitätsdiagramme
- Zustandsdiagramme
- Anwendungsfalldiagramme

Sequenzdiagramme und Zeitdiagramme befinden sich derzeit[12] in der Entwicklung.

[11] http://www.eclipse.org/modeling/mdt/?project=uml2
[12] Stand: 01.12.2009, http://wiki.eclipse.org/MDT-UML2Tools

Da der Diagrammtyp *Paketdiagramme* an sich nicht angeboten wird, können Pakete somit nur in Klassendiagrammen spezifiziert werden (ähnlich dem Ecore Diagram Editor). Paketimporte werden vom Werkzeug im Gegensatz zu Elementimporten nicht unterstützt. Der Modellierer hat zu jedem Zeit Zugriff auf alle im Modell vorhandenen Elemente, was schon bei relativ kleinen Projektgrößen sehr schnell unübersichtlich wird, da z. B. auch in Auswahldialogen zur Festlegung von Attributtypen immer alle im Modell vorhandenen Typen aufgeführt werden. Es fehlt ein Kontrollmechanismus, um die Abhängigkeiten der einzelnen Elemente untereinander überwachen und vor allem visualisieren zu können.

eUML2

Das kommerzielle Produkt *eUML2*[13] ermöglicht die Erstellung von UML 2.1 kompatiblen Klassen- und Sequenzdiagrammen auf Basis des Eclipse UML2 Metamodells. Es besteht aus vier Modulen:

- *eUML2 Modeler* ermöglicht die Erstellung von Klassen- und Sequenzdiagrammen in Eclipse
- *eDepend* ist ein Werkzeug zur Visualisierung von Abhängigkeiten im Programmcode. Es wurde für die in Abschnitt 5.3 beschriebenen Analysen der Abhängigkeiten im generierten Quelltext verwendet.
- *eEMF Modeler* erlaubt es, auf einfache Art Ecore Modelle graphisch zu erstellen
- *eDatabase* ist eine Sammlung von Werkzeugen zur Datenbankmodellierung in Eclipse.

Das Modul *eEMF Modeler* unterliegt hinsichtlich der Paketdiagramme bzw. der Definition von Sichtbarkeiten zwischen den Paketen den Beschränkungen des EMF Metamodells, d.h. es gibt keine dedizierten Paketdiagramme und es besteht keine Möglichkeit, Abhängigkeiten zwischen einzelnen Paketen mittels Importbeziehungen festzulegen.

eUML2 Modeler hingegen ist sehr auf die Kopplung mit Java ausgelegt. Ein Anlegen eines Paketes im graphischen Editor hat zur Folge, dass sofort ein entsprechendes leeres Java Paket im entsprechenden Quelltext Ordner in Eclipse erzeugt wird. Durch die starke Kopplung mit dem Java Code ist es folglich auch nicht möglich, Paketimporte zwischen den einzelnen Paketen zu definieren, da Java diesen Importtyp nicht anbietet. Die in Java vorhandenen Elementimporte werden aber ebenfalls von *eUML2 Modeler* nicht angeboten.

Topcased

Mit Hilfe des Opensource Werkzeugs *Topcased*[14] lassen sich auf dem Eclipse UML2 Metamodell basierende Modelle erstellen. Dabei werden folgende in der UML vorhandene

[13] http://www.soyatec.com/eUML2
[14] http://www.topcased.org

Diagrammarten unterstützt:

- Anwendungsfalldiagramme
- Klassendiagramme
- Komponentendiagramme
- Kompositionsstrukturdiagramme
- Verteilungsdiagramme
- Zustandsdiagramme
- Sequenzdiagramme
- Aktivitätsdiagramme

Paketdiagramme werden leider nicht explizit unterstützt. Der Anwender hat lediglich die Möglichkeit, Pakete innerhalb von Klassendiagrammen zu verwenden. Weiterhin erlaubt Topcased nur öffentliche Paketimporte als Sichtbarkeitsbeziehung zwischen den Paketen. Ebenso sind in Topcased immer alle im Modell spezifizierten Elemente in jedem Klassendiagramm referenzierbar, da auch hier vermutlich der voll qualifizierte Name verwendet wird, falls keine entsprechende Importbeziehung besteht. Topcased ist in der Lage, Diagramme zu bereits bestehenden Ecore und Eclipse UML Modellen zu erstellen.

Rational Rose / Rational Software Architect

Auch die UML Modellierungswerkzeuge der Firma Rational verhalten sich ähnlich den bereits in diesem Abschnitt vorgestellten quelloffenen Werkzeugen. Sowohl Rational Rose als auch Rational Software Architect (**RSA**) erlauben die Erstellung von Paketen innerhalb von Klassendiagrammen. Man kann als Modellierer auch Importbeziehungen zwischen Paketen spezifizieren. Modelliert man allerdings Klassen und Schnittstellen als Inhalt von Paketen, so sind dort wiederum sämtliche im gesamten Modell verfügbaren Elemente sichtbar und benutzbar, unabhängig davon, ob diese über einen Paketimport sichtbar sind, oder nicht. Abbildung 5.36 zeigt die modellierten Beziehungen zwischen den Kernpaketen. Zwischen den Paketen **storage** und **versioned_items** besteht keine Importbeziehung. Trotzdem kann im Paket **versioned_items** eine Assoziation zwischen **IVersionedItem** und **AbstractStorage** eingeführt werden, wie Abbildung 5.37 zeigt. Desweiteren ist es möglich, in der Klasse **AbstractStorage** Rückgabetypen zu verwenden, die im Paket **product** (hier das Interface **IProductModelItem**) definiert sind. Auch hier erfolgt der Zugriff auf diese Elemente, wie in der UML Spezifikation vorgesehen, über den voll qualifizierten Namen.

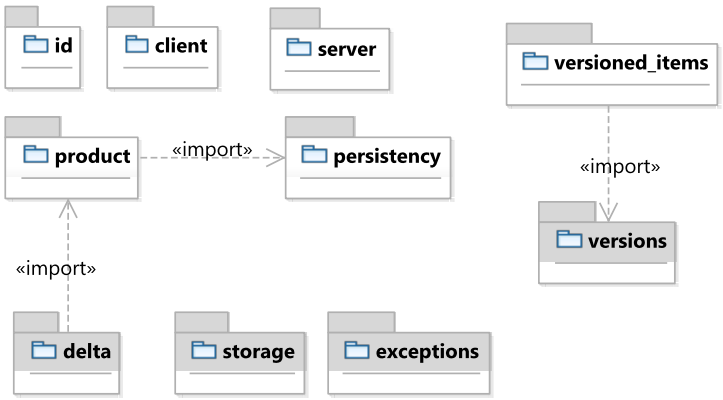

Abbildung 5.36: Die Abhängigkeiten zwischen den Kernpaketen, modelliert in Rational Software Architect.

Abbildung 5.37: Der Inhalt des Pakets **de.ubt.ai1.mod2scm.core.versioned_items**.

	Anzeige der Paket-hierarchie	öffentliche Paket-importe	private Paket importe	Element-importe
Moflon	+	+	○*	○†
Topcased	+	+	-	-
Rational Rose / RSA	+	+	+	-
Eclipse UML2	+	-	-	+
eUML2	+	-	-	-
entwickeltes Werkzeug	+	+	+	+

Tabelle 5.2: Werkzeugvergleich: Spracheigenschaften

*Der Editor erlaubt nur die Erstellung von öffentlichen Paketimporten. Nachdem ein Import erzeugt wurde, kann dessen Sichtbarkeit in einem separaten Dialog verändert werden.

†Der Editor verwendet zwar intern Elementimporte, der Benutzer hat jedoch keine Möglichkeit diese selbst anzulegen. Auch werden die automatisch erzeugten Elementimporte im Paketdiagramm nicht dargestellt.

Vergleich

Der Vergleich mit den oben aufgeführten Werkzeugen zeigt deutlich, dass Paketdiagramme in den existierenden Lösungen liberaler interpretiert werden und nicht als Hilfsmittel zur Beschreibung von Architekturen aufgefasst werden. Werden sie unterstützt, so zumeist nur zum strikten Forward Engineering, um die in den Klassendiagrammen spezifizierten Elemente zu gruppieren. Validierung, Reverse Engineering oder inkrementelles Roundtrip Engineering werden ebensowenig unterstützt wie eine Einschränkung des Bearbeitungsmodus. Einzig Moflon bietet in Teilbereichen eine ähnliche Funktionalität wie das im Rahmen dieser Arbeit entwickelte Paketdiagramm Werkzeug. Die Tabellen 5.2 und 5.3 stellen die Eigenschaften der betrachteten Werkzeuge noch einmal gegenüber.

5.9 Zusammenfassung und Fazit

In diesem Kapitel wurde die Entwicklung eines Werkzeugs zur Unterstützung der Disziplin „Modellieren im Großen“ motiviert. Dieses Werkzeug erlaubt die Definition einer Architektur eines Softwaresystems auf einer grobgranularen Ebene durch hierarchische Pakete und deren Beziehungen untereinander mittels Importen. Darüberhinaus wurde ein Integrator für die Fujaba und Ecore Modelle erstellt, der neben reinem Forward- und Reverse-Engineering *inkrementelles Roundtrip-Engineering* unterstützt und somit den Anforderungen eines dynamischen Entwicklungsprozesses Rechnung trägt. Die Möglichkeit zur Validierung der Modelle ist ebenso enthalten, wie ein spezieller beschränkter Editiermodus für das CASE-Werkzeug Fujaba. Gerade bei der Entwicklung von Softwareproduktlinien, in denen beliebige Features ausgetauscht werden können, ist es not-

	Forward Engineering	Reverse Engineering	Synchronisation	Restriktives Bearbeiten	Validierung
Moflon	+	-	-	+*	-
Topcased	+	-	-	-	-
Rational Rose / RSA	+	-	-	-	-
Eclipse UML2	+	-	-	-	-
eUML2	+	-†	-	-	-
entwickeltes Werkzeug	+	+	+	+	+

Tabelle 5.3: Werkzeugvergleich: Funktionen

*Im Gegensatz zu dem hier vorgestellten Werkzeug, kann der restriktive Bearbeitungsmodus in Moflon nicht deaktiviert werden.

†Es wird wohl das Reverse Engineering des Java-Quelltextes unterstützt, dies findet aber auf einer anderen Abstraktionsebene statt. Der hier vorgestellte Ansatz erlaubt das Reverse Engineering auf Modellebene.

wendig, einen Überblick über etwaige Abhängigkeiten der für die Realisierung dieser Features notwendigen Modellelemente zu besitzen. So hängt z. B. der Erfolg der modellgetriebenen Produktlinie für Softwarekonfigurationsverwaltungssysteme davon ab, eine möglichst lose Kopplung zwischen den einzelnen Modulen, wie etwa den Modulen zur Speicherung von Versionen und Deltas, zu erreichen [BDW08a]. Auch die Abhängigkeiten zu dem zugrunde liegenden Produktmodell sollen möglichst gering sein. Anhand eines (oder mehrerer) Klassendiagramme in einem UML Werkzeug ist dies aber selbst bei kleinen Projektgrößen schlichtweg nicht mehr möglich. Viele Werkzeuge und deren zugrunde liegenden Metamodelle (z. B. Ecore) unterstützen Pakete, so wie sie in der UML definiert sind, nicht in vollem Umfang. Meist werden sie nur dafür benutzt, um das Modell (und somit auch den daraus erzeugten Quelltext) grob zu strukturieren. Falls das Werkzeug zusätzlich noch die Definition von Importbeziehungen erlaubt, so haben diese dennoch keinen Einfluss auf Dialoge, in denen z. B. Typen von Attributen und Methoden in Klassen festgelegt werden können. Dort sind dann immer sämtliche im gesamten Modell definierten Typen sichtbar und können auch unabhängig von den festgelegten Importbeziehungen verwendet werden. Dies ist konform zur Spezifikation von UML [OMG09b], da sie besagt, dass jedes beliebige Element des UML Modells mittels des voll qualifizierten Namens referenziert werden kann. Die einzige Ausnahme stellt *MOFLON*[15] dar, das die Elemente in diesen Dialogen auf die sichtbaren Elemente beschränkt. Da MOFLON aus diversen Gründen (u.a. Verwendung von MOF Diagrammen anstatt UML Diagrammen, keine Möglichkeit der Generierung von EMF Code und somit keine modellgetriebene Entwicklung von graphischen Editoren) nicht als Entwicklungsplattform in Frage kam, wurde der in diesem Kapitel vorgestellte Paketdiagrammeditor

[15]http://www.moflon.org

als eigenständige Anwendung komplett modellgetrieben entwickelt und per Plugin an das für die Entwicklung der Produktlinie verwendete Werkzeug Fujaba angebunden.

Die Erfahrung aus der Entwicklung der Produktlinie für Softwarekonfigurationsverwaltungssysteme [BDW08a] hat aber gezeigt, dass neben dem Paketdiagrammeditor auch noch gewisse Konventionen bei der Modellierung eingehalten werden sollten, um die Übersichtlichkeit zu erhöhen. So sollte z. B. in Fujaba der Inhalt eines Paketes wenn möglich immer in einem Klassendiagramm modelliert werden. Zusätzlich dazu sollte dieses Klassendiagramm dann dem entsprechenden Paket zugeordnet werden. Der Paketdiagrammeditor unterstützt auch bewusst nicht den Zugriff auf Elemente über deren voll qualifizierten Namen, da so im schlimmsten Fall ein Paketdiagramm ohne Importbeziehungen entstehen könnte. Man könnte demzufolge keinerlei Aussagen über die Abhängigkeiten von einzelnen Elementen treffen. Somit würde es scheinen, als wären wenige bis keine Abhängigkeiten im Modell vorhanden, durch die exzessive Nutzung der qualifizierten Namen würde aber eine starke Kopplung auf konzeptioneller Ebene bestehen.

Auch die Vermischung der beiden Diagrammarten (Paket- und Klassendiagramme), die einige der hier vorgestellten Werkzeuge vornehmen ist nicht wünschenswert. Dadurch leidet ebenfalls die Übersichtlichkeit sehr stark, da keine Möglichkeit besteht, das komplette Paketdiagramm auf grob granularer Ebene zu betrachten.

Der hier vorgestellte Paketdiagrammeditor hat für die Generierung des Codes für die Zielplattform keinen Einfluss. Da in Java das Paketkonzept fundamental unterschiedlich von dem Paketkonzept der UML ist, werden in Java alle Zugriffe auf nicht-sichtbare Java Klassen auf private Elementimporte abgebildet. Gäbe es hingegen auch in Java ein Konzept von öffentlichen Paketimporten und Pakete als Typen, so hätte das Paketdiagramm auch Einfluss auf die Codegenerierung.

6 Verbindung Featuremodell - Domänenmodell

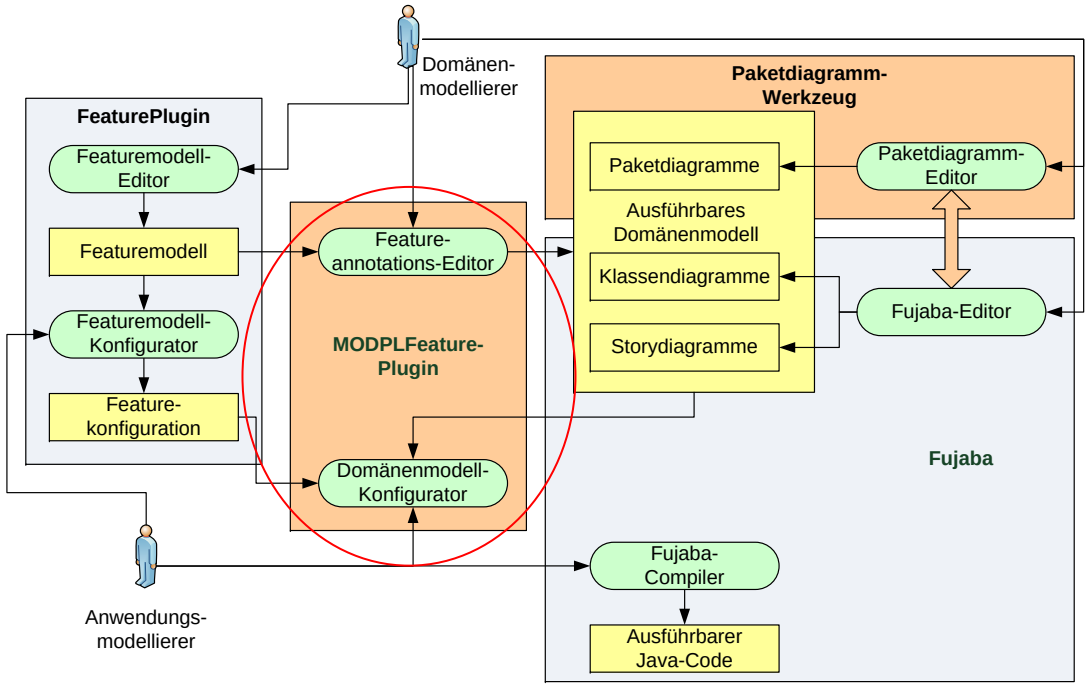

Abbildung 6.1: Beschreibung der Komponenten zur Abbildung der Features auf das Domäenenmodell.

Dieses Kapitel behandelt die Abbildung von Elementen des Featuremodells auf Domänenmodellelemente in der Domänenentwicklung (die Erstellung eines „annotierten Domänenmodells“). Die Umsetzung einer Konfiguration in der Anwendungsentwicklung erfolgt nach der Interaktiven Erstellung einer Konfiguration aus dem Featuremodell vollständig automatisiert. Abbildung 6.1 zeigt einen Überblick über die zu diesem Zwecke erstellten neuen Werkzeuge und ihre Einbettung in die Werkzeugkette zur modellgetriebenen Entwicklung von Softwareproduktlinien.

Die Darstellung von Variabilität in Featurediagrammen bzw. Featuremodellen als Resultat der Feature-orientierten Domänenanalyse [KCH+90] gibt es nun schon seit etwa 20 Jahren. Fast ebenso lange gibt es schon die Praktiken der modellgetriebenen Softwareent-

wicklung. Allerdings gibt es nach wie vor eine Reihe von ungelösten Fragen im Bezug auf die Kopplung von Featuremodellen und Domänenmodellen, wie auch der Begründer von FODA, Kyo Chul Kang, in seiner Keynote auf der Softwareproduktlinienkonferenz 2009 feststellte [Kan09]. Im folgenden Kapitel wird ein Ansatz zur Abbildung von Elementen des Featuremodells auf Domänenmodellelemente vorgestellt. Ziel des Ansatzes ist es, eine konkrete Konfiguration von Features anhand des zugrunde liegenden Domänenmodells abzubilden. Dies kann auf drei verschiedene Arten erfolgen: (1) durch reine Visualisierung im Modell, (2) durch Generierung von konfiguriertem Quelltext und (3) durch eine Modelltransformation des Domänenmodells in ein konfiguriertes Modell.

Nach einer Motivation folgt in diesem Kapitel die Beschreibung der wesentlichen Eigenschaften des im Rahmen dieser Arbeit gewählten Ansatzes. Anschließend werden die gewählte Methode zur Kopplung beider Modelle, sowie umfangreiche Konsistenzbedingungen diskutiert. Nach einer ausführlichen Betrachtung der verschiedenen Varianten zur Realisierung einer Konfiguration erfolgt zunächst eine Beschreibung der Funktionalität der erstellten Werkzeuge. Danach wird die Realisierung dieser Werkzeuge mit modellgetriebenen Techniken erläutert. Nach einer Einführung in die Arbeit mit den Werkzeugen wird das Kapitel durch einen umfangreichen Literaturvergleich abgeschlossen.

6.1 Motivation

Bei der Entwicklung von Produktlinien unterscheidet man die Ebenen *Domänenentwicklung* und *Anwendungsentwicklung*. In dem hier vorgestellten Ansatz wird während der Domänenentwicklung ein Domänenmodell entwickelt, in dem sich alle Varianten wiederfinden. Es gibt also nicht n (ggf. versionierte) Modelle für n Varianten. Während das Domänenmodell in unserem Ansatz mit Fujaba erstellt wird, sind die Varianten in einem Featuremodell erfasst. Anhand des Featuremodells kann ein konkretes Produkt durch Binden der Variabilität ausgewählt werden. Analog dazu müssen für ein konfiguriertes Domänenmodell alle nicht benötigten Bestandteile entfernt werden, damit das resultierende Modell der gewählten Konfiguration entspricht.

Bei der bedingten Übersetzung in Programmiersprachen, können Quelltextfragmente durch Präprozessordirektiven selektiert werden. Auf diese Weise kann dort Variabilität ausgedrückt werden. Der gesamte Quelltext stellt das Äquivalent zum Domänenmodell dar. Alle Varianten sind im Quelltext ausgedrückt und durch Präprozessoranweisungen entsprechend markiert und den jeweiligen Features aus dem Featuremodell zugeordnet.

Abbildung 6.2 zeigt die Zusammenhänge zwischen Featuremodell und Konfiguration sowie Domänenmodell und dem konfigurierten Modell. Während der Schritt vom Featuremodell zur Konfiguration bekannt ist und von allen gängigen Werkzeugen unterstützt wird, ist zu klären, wie das konfigurierte Modell aus dem Domänenmodell abgeleitet werden kann und wie die Informationen aus Featuremodell und Konfiguration in diesen Prozess einfließen. Analog zur bedingten Übersetzung müssen die Modellelemente der-

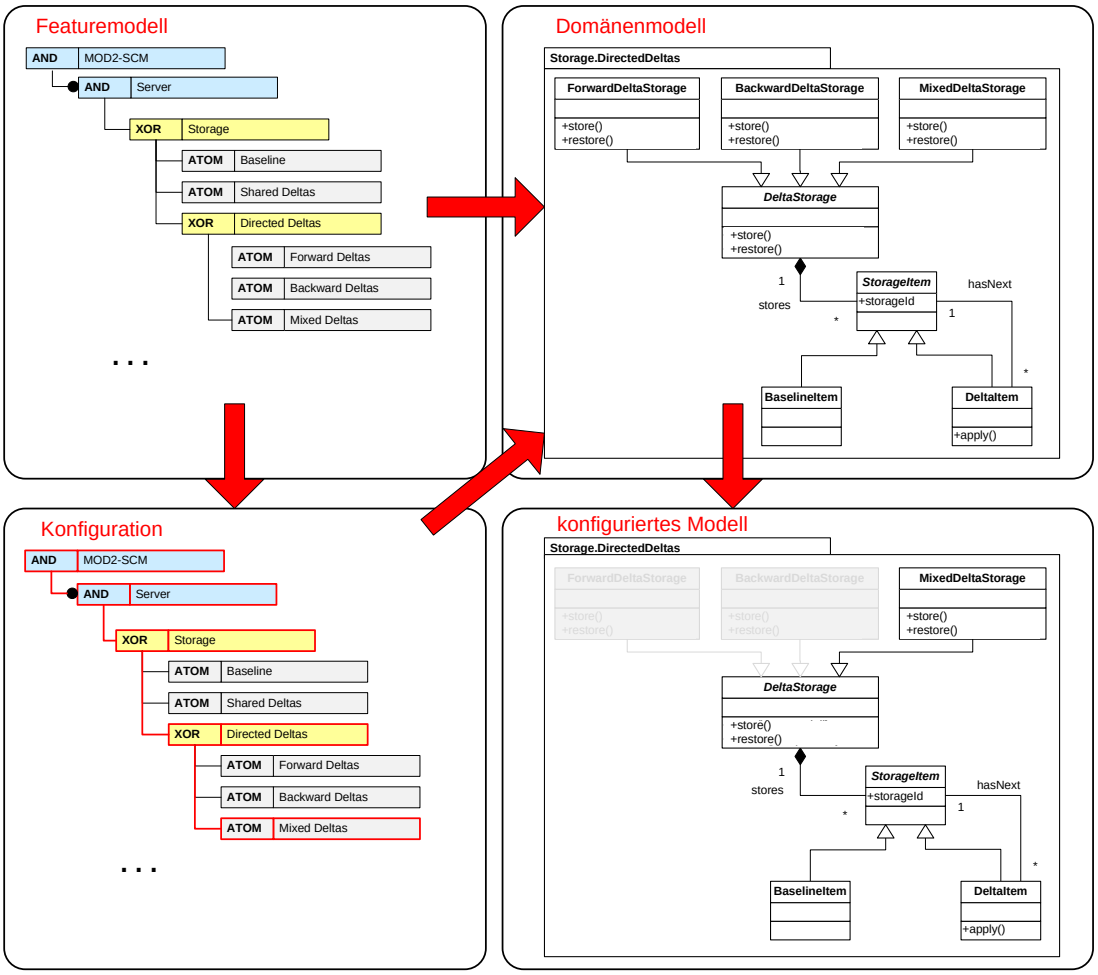

Abbildung 6.2: Zusammenhang zwischen Featuremodell, Konfiguration, Domänenmodell und konfiguriertem Modell.

art ausgezeichnet werden, dass eine Zuordnung zu den Varianten, die dadurch realisiert werden möglich ist. Insbesondere muss geklärt werden, auf welche Weise Modellelemente ausgezeichnet werden können und vor allem, wie der Benutzer den Überblick darüber behalten kann, welche Modellelemente ausgezeichnet sind und mit welchen Features sie dadurch verknüpft sind. Ein weiterer wichtiger Aspekt sind Konsistenzbedingungen, die dabei eingehalten werden müssen, damit zum einen auch alle Konfigurationen, die aus dem Featuremodell abgeleitet werden erzeugt werden können und zum anderen diese Konfigurationen in sich konsistent sind.

6.2 Wesentliche Eigenschaften des Ansatzes

Der im Laufe dieses Kapitels vorgestellte Ansatz zeigt eine Möglichkeit, Features auf Domänenmodellelemente abzubilden und daraus entsprechende Konfigurationen abzuleiten. Dies kann zum einen durch Visualisierung der Konfiguration im Fujaba Editor geschehen, aber auch durch die Erzeugung von konfiguriertem Quelltext oder der Transformation des Domänenmodells in ein konfiguriertes Modell. Die Abbildung von Features auf Elemente des Domänenmodells kann dabei in beliebiger Granularität erfolgen. In Abbildung 6.3 ist auf der linken Seite der Ausschnitt eines Featuremodells zu sehen, die rechte Seite zeigt einen Ausschnitt aus dem Domänenmodell, das aus einem Paketdiagramm, Klassendiagrammen und Storydiagrammen (aus Platzgründen nicht in der Abbildung dargestellt) besteht. Features können nun etwa Paketen zugewiesen oder auch auf einzelne Klassen abgebildet werden. Darüber hinaus können die Features selbstverständlich auf jedes Modellelement, also beispielsweise auch auf Attribute, Methoden, Methodenparameter, Assoziationen, Rollenenden, Vererbungsbeziehungen und Elemente von Storydiagrammen abgebildet werden.

Die folgenden Abschnitte zeigen, wie man den Ansatz der bedingten Übersetzung auf Modelle transferieren kann, um die entsprechenden Bestandteile im Domänenmodell den Features, die dadurch realisiert werden zuzuordnen. Diese Zuordnung entspricht einem boolschen Ausdruck, der wahr ist, wenn das entsprechende Feature in der Konfiguration enthalten ist. In diesem Fall ist das Modellelement auch Bestandteil des konfigurierten Modells. Im Umkehrschluss bedeutet dies, dass alle Modellelemente, für die der boolsche Ausdruck nicht erfüllt ist, aus dem Modell entfernt werden. Da auch hier analog zur Verwendung von Präprozessoren Probleme auftreten, die die Konsistenz des resultierenden Modells gefährden, werden notwendige Regeln aufgestellt, die dem entgegen wirken.

Die Realisierung der vorgestellten Konzepte erfolgt mit Hilfe einer Erweiterung des CASE Werkzeugs Fujaba. Durch ein Plugin wird dem Benutzer die Möglichkeit angeboten, Featuremodelle und Konfigurationen, die aus dem Werkzeug *FeaturePlugin* [AC04] exportiert wurden, einzulesen und die darin enthaltenen Informationen zur Auszeichnung bzw. Konfigurierung des Domänenmodells zu verwenden. Die Hierarchie der Features wird dem Benutzer dabei ebenso angezeigt wie deren Namen. Um Tippfehler bei der Vergabe von Features zu Modellelementen auszuschließen, erlaubt das Plugin nur die Vergabe von Features, die der Benutzer aus der Liste der importierten Features auswählen kann. Außerdem hat der Benutzer die Möglichkeit, die Vergabe von Features auf Plausibilität und Vollständigkeit zu überprüfen.

6.3 Kopplung Featuremodell und Domänenmodell

Die folgenden Abschnitte beschäftigen sich mit der Kopplung der beiden Modelle. Es wird eine Möglichkeit gezeigt, wie Elemente des Featuremodells auf Elemente des Domänenmodells abgebildet werden können. Da das Ziel von modellgetriebener Softwareentwicklung die Erzeugung von lauffähigem Code ist, werden Konsistenzbedingungen aufgestellt,

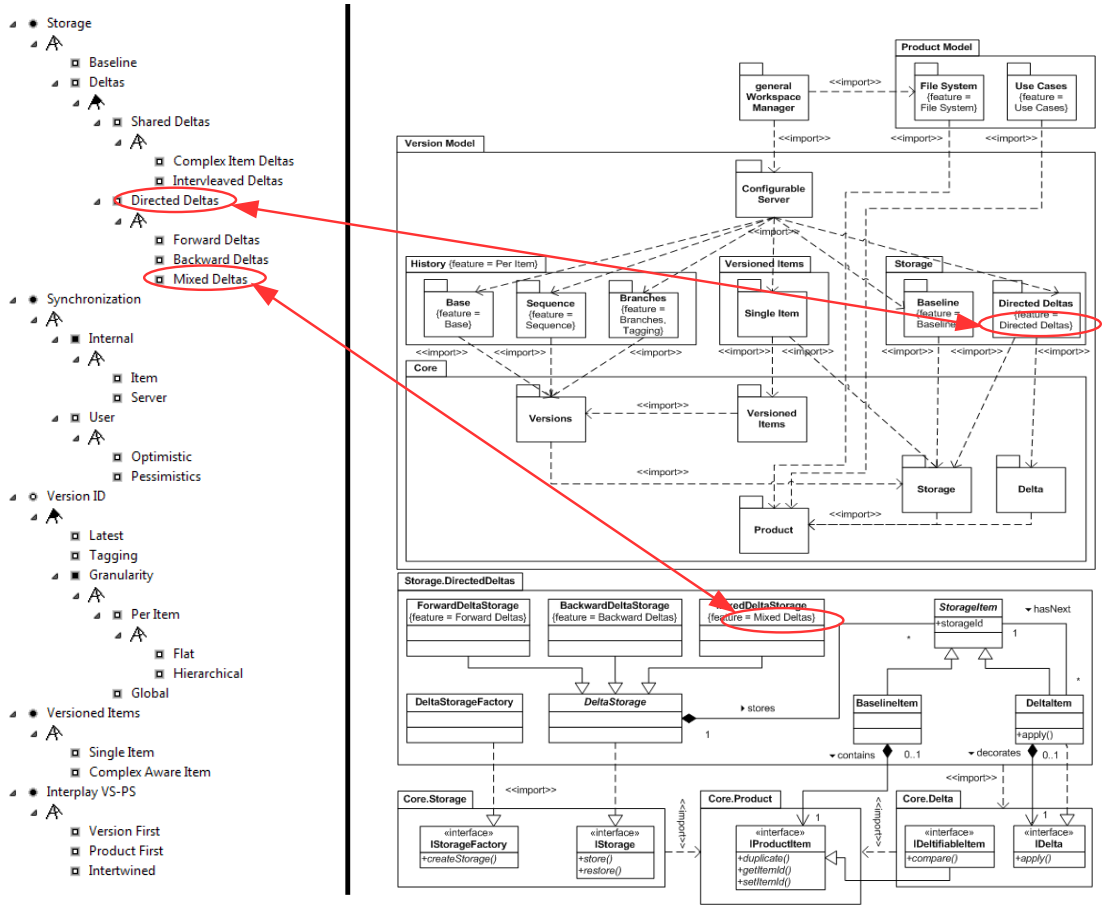

Abbildung 6.3: Abbildung von Elementen des Feature Modells auf Elemente des Domänenmodells.

die die Erzeugung von syntaktisch korrekten Quelltext so weit wie möglich sicherstellen sollen. Die Konfigurierung des Domänenmodells kann dabei auf unterschiedliche Arten erfolgen:

1. Konfigurierung zur Entwicklungszeit
   a) Konfigurierung des Modells durch Modelltransformation
   b) Direkte Erzeugung von konfiguriertem Quelltext
2. Konfigurierung zur Laufzeit mit Hilfe des Fabrik Musters [GHJV94]

6.3.1 Möglichkeit zur Darstellung von Variabilität im Domänenmodell

Fujaba und andere auf UML basierende CASE Werkzeuge bieten per se nur sehr eingeschränkte Möglichkeiten zur Modellierung von Variabilität. So ist es beispielsweise

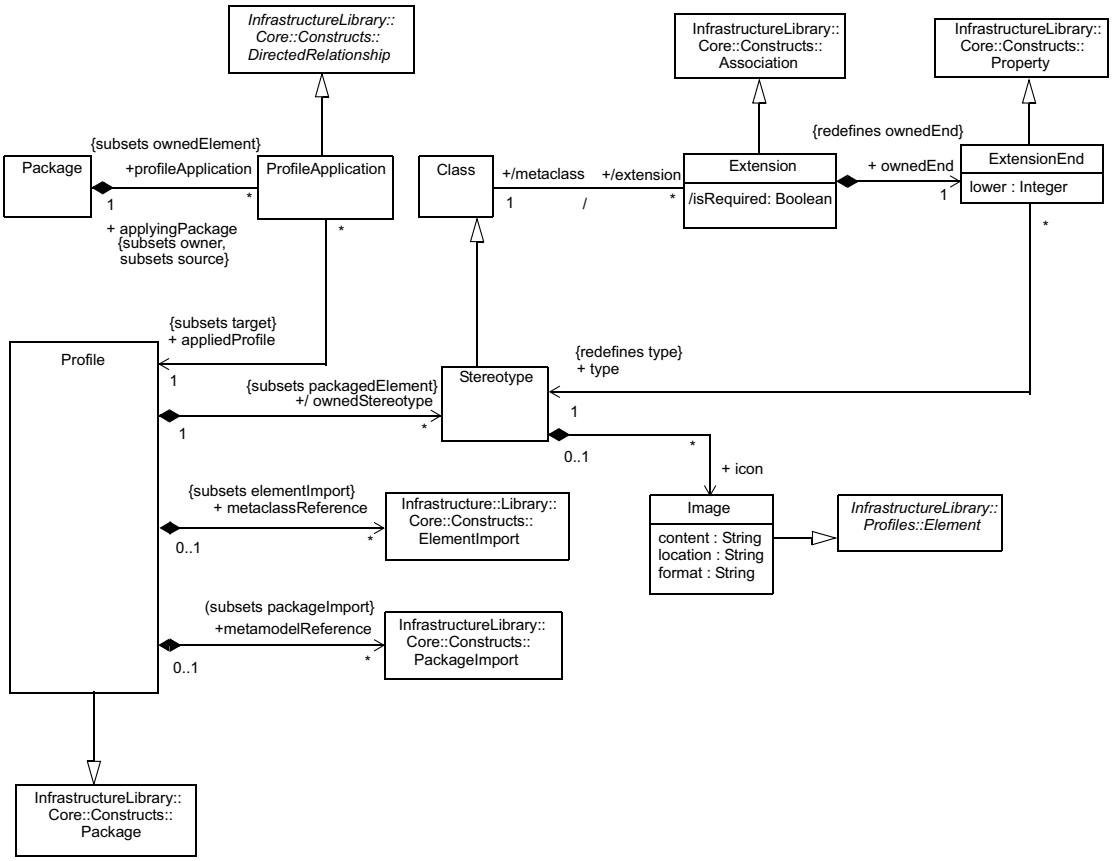

Abbildung 6.4: Das UML Profile Paket aus der UML Spezifikation (aus: [OMG09b], Seite 654).

nicht möglich, für eine Assoziation unterschiedliche Varianten hinsichtlich der Kardinalitäten ihrer Rollenenden zu definieren, oder eine Assoziation je nach gewähltem Feature an unterschiedlichen Klassen enden zu lassen. Ebenso ist es in den Storydiagrammen von Fujaba nicht möglich, bei Objektinstanziierungen den Typ des erzeugten Objektes variabel zu definieren.

Eine genaue Beschreibung, wie mit Hilfe von Fujaba und der Verwendung von Entwurfmustern und objektorientierten Techniken dennoch das Domänenmodell für die Produktlinie für Softwarekonfigurationsverwaltungssysteme erstellt werden kann, ist in der Doktorarbeit von Alexander Dotor [Dot10] zu finden.

6.3.2 Erweiterung von UML

Die Sprache UML wurde generell für ein sehr breites Anwendungsspektrum entworfen. Dennoch sind für manche Anwendungen Erweiterungen und Spezialisierungen von Nöten, etwa im Bereich von Echtzeitanwendungen oder auch für die Geschäftsprozessmodellierung (siehe auch [JSW99] oder [Sch02]). Für derartige Fälle hält die UML den Mechanismus der *UML Profile* für anwendungsspezifische Erweiterungen bereit. UML

Profile stellen die Grundlage für das von der OMG propagierte *Model Driven Architecture (MDA)* [Fra03] dar, der OMG Variante der modellgetriebenen Softwareentwicklung, indem sie eine Formalisierung von Domänenmodellen erlauben. Daher werden UML Profile immer für eine spezielle Anwendungsdomäne erstellt. Die UML 2.0 Superstruktur Spezifikation definiert bereits vier UML Profile für Industriestandards im Bereich Softwarekomponenten: (1) ein Profil für die Modellierung von *Enterprise Java Beans (EJB)*, (2) für .NET-Komponenten, (3) für Komponenten des *Component Object Model (COM)* und (4) für Komponenten des *Corba Component Model (CC)*. Genaugenommen ist UML somit nicht nur eine Sprache, sondern eine Grundlage für eine Familie von UML-basierten Sprachen. Im von der OMG verfolgten modell-getriebenen Entwicklungsansatz *MDA* ist dieser Erweiterungsmechnismus aufgrund der Notwendigkeit, verschiedene Systemaspekte und Abstraktionsebenen zu unterstützen, essentieller Bestandteil der Philosophie. Die Erweiterungen können mit Hilfe von *Stereotypen* und *Tagged values* erfolgen. Ein UML Profil ist eine Definition einer Menge von Stereotypen. **Stereotype** ist eine Spezialisierung von **Class**. Attribute, die einem Stereotyp zugeordnet werden, werden als *Tagged values* bezeichnet. Stereotypen erweitern die Elemente des UML Metamodells ([Fra03], S. 143). Die Spezifikation des UML Profile Paketes in UML 2.0 ist in Abbildung 6.4 zu finden. In UML Version 1.3 konnten *Tagged values* zur Erweiterung von Modellelementen auch ohne Stereotypen verwendet werden. Seit UML 1.4 wird diese Option nur noch aus Gründen der Abwärtskompatibilität unterstützt. Ab UML 2.0 kann ein *tagged value* nur noch als Attribut, das für einen Stereotyp definiert ist, dargestellt werden. Daher muss ein Modellelement mit einem Stereotyp erweitert werden, damit es mit *tagged values* ausgezeichnet werden kann (siehe [OMG09b], S. 677).

Hierarchie der Metamodellierung

Eine spezielle *Metadaten*-Architektur, auf der die UML basiert ist *MOF* (Meta-object facility) [OMG06a, OMG07]. MOF schließt die Lücke zwischen verschiedenen Metamodellen, indem es eine allgemeine Grundlage für Metamodelle bereitstellt. Innerhalb von MOF werden Daten im wesentlichen in vier Meta-Ebenen klassifiziert, die in Abbildung 6.5 dargestellt sind:

M0-Ebene Diese Ebene beschreibt die konkreten Instanzen, die ausgeprägten Daten.

M1-Ebene Diese Ebene enthält die Benutzermodelle, die die Daten der M0-Ebene definieren, wie etwa ein mit einem UML Werkzeug erstelltes Modell.

M2-Ebene Diese Ebene enthält die Meta-Modelle, die zur Erstellung der Benutzermodelle verwendet werden. Sie definieren, wie die Modelle aufgebaut und strukturiert sind. So definieren etwa Sprachelemente wie Klassen, Assoziationen und Attribute der UML 2.0 wie konkrete UML-Modelle aufgebaut sein können.

M3-Ebene Sie enthält Meta-Meta-Modelle und beschreibt eine abstrakte Ebene, die zur Definition der M2-Ebene verwendet wird.

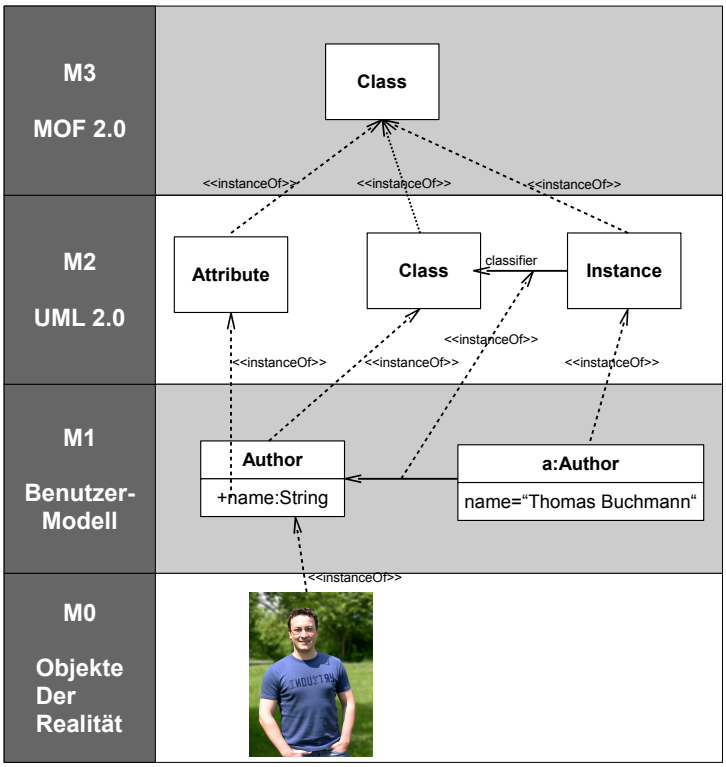

Abbildung 6.5: Hierarchie der Metamodellierung.

Stereotypen

Stereotypen repräsentieren das Kernkonzept sprachinhärenter Erweiterungsmechanismen. Sie erlauben sämtliche Metaklassen des UML-Metamodells (außer der Metaklasse *Stereotype* selbst) auf kontrollierte Art und Weise auf der Ebene M2 zu erweitern. Sie können wie alle anderen Metaklassen auch instanziiert und zur Modellierung auf der Ebene M1 verwendet werden. Ein Stereotyp stellt eine spezielle Klasse dar, die in der Darstellung zur Unterscheidung von normalen Klassen mit dem Schlüsselwort ≪*stereotype*≫ in Guillemets bezeichnet wird. Analog zu Klassen können einem Stereotyp auch Attribute zugeordnet werden, die bei Stereotypen jedoch auch als Eigenschaftsdefinitionen bezeichnet werden. Stereotypen können bei der Modellierung beliebigen Modellelementen zugeordnet werden. In der Darstellung wird der Name des Stereotyps in Guillemets bei dem entsprechenden Modellelement (etwa einer Klasse) angezeigt. Bei der Generierung von Quelltext können die Stereotypen dafür benutzt werden, um beispielsweise unterschiedliches Verhalten auszudrücken. So bewirkt etwa der Stereotyp ≪**JavaBean**≫ in Fujaba, die Generierung von speziellen Zugriffsmethoden für das Observer Muster [FF04].

Formale Definition des Profils zur Kopplung

Ein Profil kann mit dem gleichen Grad an Präzision wie ein Domänenmodell modelliert werden. Die von der OMG dafür vorgesehene Technik ist die Erstellung eines formalen

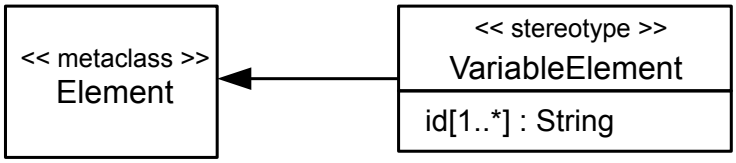

Abbildung 6.6: Formales Klassenmodell des Feature Profils.

UML Modells, das Stereotypen, Tags und die UML Meta-Modellelemente enthält, die erweitert werden. Im Sprachterminus der UML wird ein formales Modell eines UML Profils auch *virtuelles Meta-Modell* genannt. Abbildung 6.6 zeigt das Modell, das formal den Stereotyp des Profils definiert, das verwendet wird, um Elemente des Featuremodells auf Domänenmodellelemente abzubilden. Es wird ein neuer Stereotyp **VariableElement** eingeführt, der durch eine Klasse dargestellt wird, die mit dem Schlüsselwort ≪**stereotype** ≫ ausgezeichnet ist. Das UML Metamodellelement, das durch diesen Stereotyp erweitert wird, ist durch das Schlüsselwort ≪**metaclass** ≫ dargestellt. Das Modell sagt aus, dass der Stereotyp **VariableElement** das UML Meta-Modellelement **Element** erweitert. Tags, die mit dem Stereotyp zusammenhängen werden durch Attribute ausgedrückt, die zu der Klasse gehören, die den Stereotyp repräsentiert. In unserem Fall hat die Klasse **VariableElement** ein Attribut **id**, das eine Liste der eindeutigen Bezeichner der gewählten Features aus dem Featuremodell entspricht und somit eine eindeutige Zuordnung erlaubt. Mit Hilfe dieses Mechanismus kann also die in Abbildung 6.3 dargestellte Art der Kopplung von Featuremodell und Domänenmodell realisiert werden.

6.3.3 Abbildung von Features auf Domänenmodellelemente

Durch die Verwendung eines einzigen Domänenmodells, das alle Varianten der Produktlinie enthält folgt automatisch eine Begrenzung der Variabilität, da das Gesamtmodell den Konsistenzbedingungen des verwendeten Metamodells unterliegt. Elemente des Gesamtmodells können mit einzelnen Features, oder auch mit Featureausdrücken ausgezeichnet werden, um so die implizit enthaltenen Varianten entsprechend zu markieren. Bei der Konfigurierung des Domänenmodells werden nun alle Modellelemente, die nicht der gewählten Konfiguration entsprechen herausgefiltert. Eine Featureannotation[1] ist somit eine Bedingung, die erfüllt sein muss, damit das Modellelement auch Bestandteil des konfigurierten Modells ist.

Definition. Sei f ein Feature und C eine Konfiguration. $CM \subset DM$ sei ein konfiguriertes Modell des Domänenmodells und $d \in DM$ sei ein Element des Domänenmodells, das mit f annotiert ist. Dann gilt:

$$d \in CM \Leftrightarrow f \in C \tag{6.1}$$

[1] als Featureannotation wird die Vergabe des Stereotyps **VariableElement** an ein Modellelement bezeichnet.

6.3.4 Konsistenzbedingungen

Die Kopplung zwischen Featuremodell und Domänenmodell erfolgt also durch den Stereotyp **VariableElement**, der für beliebige Elemente des Domänenmodells vergeben werden kann. Der Wert des Attributs **id** enthält den eindeutigen Bezeichner eines oder mehrerer Elemente aus dem Featuremodell, die durch das entsprechende Domänenmodellelement realisiert werden. Das Werkzeug *FeaturePlugin* erzeugt zu jedem Eintrag automatisch einen eindeutigen Bezeichner, der für die Kopplung verwendet wird. Bei Verwendung eines Werkzeugs zur Featuremodellierung, das keine eindeutigen Bezeichner verwendet, kann ein eindeutiger Bezeichner anhand des Pfadnamens des Features, analog zum qualifizierten Namen von Paketen, gebildet werden. Ein Modellelement kann dabei mit diesem Stereotypen verknüpft werden und dieser Stereotyp kann in dem ihm zugeorneten *tagged value* den eindeutigen Bezeichner eines oder mehrerer Features aus dem Featuremodell enthalten. Durch Zuordnung von mehr als einem Feature an ein Modellelement können so komplexe Featureausdrücke durch Verwendung boolscher Operatoren zur Konjunktion, Disjunktion und Negation von einzelnen Features dargestellt werden. In der aktuellen Implementierung des in dieser Arbeit vorgestellten Werkzeugs ist bisher nur die Konjunktion implementiert. Bei der Kopplung der Modelle gelten folgende Konventionen:

Konfigurierung Bei der Konfigurierung des Domänenmodells werden alle Modellelemente, die nicht Teil der aktuellen Konfiguration sind, herausgefiltert.

Invarianten Unveränderliche Bestandteile der Produktlinie, d.h. Features, die in jeder Konfiguration vorhanden sein müssen, werden nicht auf Modellelemente abgebildet. Für die entsprechenden Modellelemente bedeutet dies im Umkehrschluss, dass sie ebenfalls in jeder Konfiguration vorhanden sind. Konzeptionell wird also jedem Modellelement ein Featureausdruck zugeordnet. Der Wert des Standardausdrucks ist **wahr**.

Varianten Realisiert ein Modellelement ein oder mehrere Features, so erhält es den Stereotyp **VariableElement** mit den entsprechenden eindeutigen Bezeichnern. Für die Konfiguration bedeutet dies, dass das Modellelement nur dann in der Konfiguration enthalten ist, wenn auch alle Features in der Konfiguration enthalten sind (d.h. die Auswertung erfolgt analog zu einem logischen *Und*-Operator). Eine mögliche Erweiterungsoption wäre die Unterstützung von Featureausdrücken mit logischen Operatoren.

Propagationsregeln

Um aus einem mit Stereotypen ausgezeichneten Domänenmodell eine Konfiguration abzuleiten, sei es durch Visualisierung, Generierung von konfiguriertem Quelltext oder die Erstellung eines konfigurierten Modells, müssen alle Modellelemente, die nicht Bestandteil der Konfiguration sind, herausgefiltert werden. Da in letzter Instanz aber immer lauffähiger Code erzeugt werden soll, werden im folgenden notwendige, aber nicht hinreichende Konsistenzbedingungen definiert, die die syntaktische Korrektheit des erzeugten

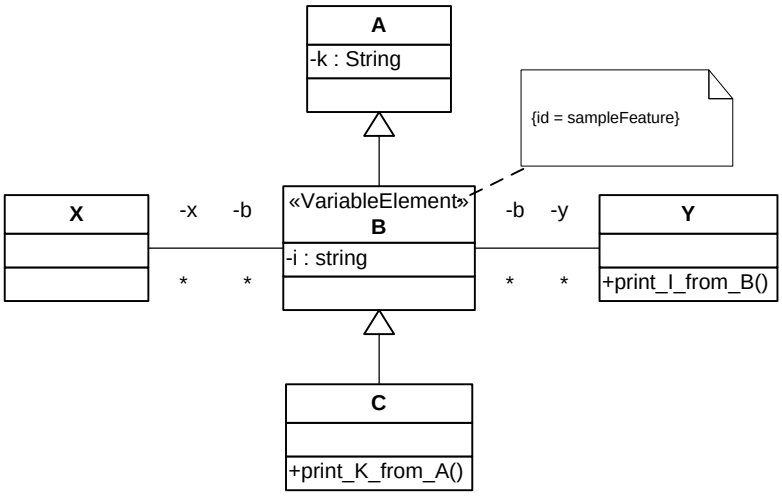

Abbildung 6.7: Einfaches Klassendiagramm mit einer ausgezeichneten Klasse.

Quellcodes weitestgehend sicherstellen. Abbildung 6.7 zeigt ein einfaches Klassendiagramm mit einer ausgezeichneten Klasse. Die nachfolgend aufgestellten Regeln werden anhand dieses Beispiels erläutert.

Im Beispiel in Abbildung 6.7 ist die Klasse **B** mit einem Feature mit dem Bezeichner *sampleFeature* ausgezeichnet. Bei einer Konfiguration, die das Feature *sampleFeature* nicht enthält, entstehen nun folgende syntaktische Fehler:

- Beide Vererbungsbeziehungen haben nun keine Quelle bzw. kein Ziel.
- Beide Assoziationen haben nur ein Rollenende.
- Die Klassen **X** und **Y** haben jeweils ein Rollenende eines nicht existierenden Typs.
- Die Methode **print_K_from_A** erbt nicht mehr direkt bzw. indirekt von **A**, d.h. es kann nicht mehr direkt auf das Attribut **k** der Klasse **A** zugegriffen werden, was z. B. zu Compilierfehlern führt.
- In der Methode **print_I_from_B** kann ebenfalls nicht mehr auf das Attribut **i** der Klasse **B** zugegriffen werden, da die Klasse **B** nicht existiert.

Unter der Annahme, dass das mit Stereotypen versehene Domänenmodell syntaktisch korrekt ist, sind Fehler im aus dem konfigurierten Model generierten Code das Resultat des Fehlens von Modellelementen, die nicht Bestandteil der aktuellen Konfiguration sind (und für die folglich auch kein Code erzeugt wird). Wie bereits eingangs erwähnt, sind Modellelemente ohne Featureannotation (also ohne den Stereotyp *VariableElement*) in jeder beliebigen Konfiguration vorhanden. Sie stellen die unveränderlichen Bestandteile der Softwareproduktlinie dar. Ein mit Stereotypen versehenes Element ist nur in der Teilmenge aller Konfigurationen vorhanden, die ebenfalls alle Features enthält, mit

denen dieses Modellelement ausgezeichnet ist. Nachfolgend werden daher Regeln formuliert, die die syntaktische Korrektheit der konfigurierten Modelle sicherstellen sollen. Die verwendete Regelsprache ist die auf Mengentheorie und Prädikatenlogik basierende *Object Constraint Language* [OMG06b, WK03]. Für Elemente, die von ausgezeichneten Modellelementen abhängen, bedeutet dies nun folgendes (siehe auch [BD09a]):

Regel 1. *Sei M die Menge aller Elemente des Domänenmodells und $A, B \in M$. C sei die Menge aller konfigurierten Domänenmodelle und $C_A = \{C|A \in C\}$ und $C_B = \{C|B \in C\}$. Dann gilt:*

$$A \rightarrow B \Rightarrow C_A \subset C_B. \tag{6.2}$$

Abstrahiert auf Stereotypen bedeutet dies, dass Regeln formuliert werden müssen, die Sicherstellen, dass jeder Stereotyp an einem Element auch an davon abhängigen Elementen vorhanden sein muss, d.h. die Menge der Stereotypen, die dem Modellelement B zugeordnet sind eine Teilmenge der Stereotypen von Modellelement A[2] ist. Die Einhaltung dieser Regel ist eine notwendige, aber keine hinreichende Bedingung für die Konsistenz des konfigurierten Modells bzw. des konfigurierten Quelltextes. Die Filterung obligatorischer Elemente, wie etwa die Startaktivität eines Storydiagramms, kann damit nicht verhindert werden.

Propagationsregeln gegen UML Metamodellverstöße

Die nachfolgenden Regeln wurden nach eingehender Analyse der UML Superstruktur Spezifikation [OMG09b] aufgestellt. Sie stellen notwendige Bedingungen auf, um die Wohlgeformtheit des konfigurierten Modells sicherzustellen, indem sie die Existenz des Stereotyps *VariableElement* an vom aktuellen Modellelement abhängigen Elementen fordern (analog zu Regel 1).

```
1 context Element:
2 Collection featureStereotypes = self.stereotype->select(stereotype.
    name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.ownedElement->forAll( o: Element | o.stereotype->exists(s:
    Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
    )
```

Listing 6.1: **Regel 2:** Propagation auf existenzabhängige Elemente.

[2] In diesem Fall ist die Richtung der Teilmengenrelation genau umgekehrt, da eine größere Anzahl von tagged values an einem Modellelement bedeutet, dass dieses Element in einer kleineren Anzahl von konfigurierten Domänenmodellen auftritt!

Diese Regel besagt, dass ein mit dem Stereotyp *VariableElement* ausgezeichnetes Element das Vorhandensein des Stereotyps und der zugeorneten tagged values an allen enthaltenen Elementen impliziert. Das Wurzelelement `Element` der UML Superstruktur Spezifikation, führt eine Komposition zwischen dem `owner`-Element und dessen `owned elements` ein 6.8. Eine genaue Analyse der Vererbungshierarchie der UML Superstruktur liefert eine Definition der Abhängigkeiten, wie sie in Tabelle 6.1 zu sehen ist.

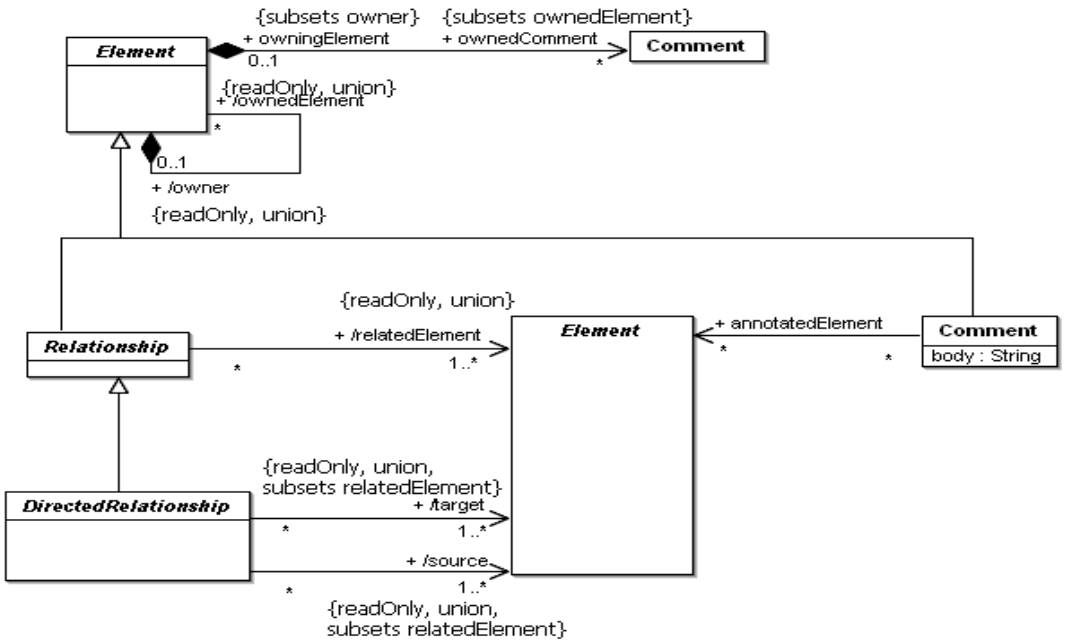

Abbildung 6.8: Die Definition der Beziehungen zwischen `owner` und `owned elements` ([OMG09b], p. 25)

Ausgezeichneter Typ	**existenzabhängige Elemente (Typ)**
`Association`	enthaltene Regeln (`Constraint`), ausgehende Importe (`ElementImport` oder `PackageImport`), Verallgemeinerungen zu Super-Assoziationen (`Generalization`), nicht navigierbare oder n-äre Rollen (`Property`)
`Class`	verschachtelte Klassen (`Class`), enthaltene Regeln (`Constraint`), ausgehende Importe (`ElementImport` oder `PackageImport`), Verallgemeinerungen zu Oberklassen (`Generalization`), definierte Operationen (`Operation`), definierte Attribute (`Property`)
`Constraint`	Constraint Definition (`ValueSpecification`)
`ElementImport`	*keine*
`Generalization`	*keine*
`Operation`	Vor-/Nachbedingungn und Bedingungen im Methodenrumpf (`Constraint`), Parameter (`Parameter`)
`Package`	enthaltene Assoziationen (`Association`), enthaltene Klassen (`Class`), owned rules (`Constraint`), ausgehende Importe (`ElementImport` oder `PackageImport`), Unterpakete (`Package`), ausgehende Paketverschmelzungsbeziehungen (`PackageMerge`)
`PackageMerge`	*keine*
`PackageImport`	*keine*
`Parameter`	Standardwert (`ValueSpecification`)
`Property`	Standardwert (`ValueSpecification`)
*`ValueSpecifications`**	*keine*

Tabelle 6.1: Propagationstabelle für konkrete Elemente des UML Klassendiagramms (Regel 2)

*Eine `ValueSpecification` ist tatsächlich eine abstrakte Klasse, die Ausdrücke bestehend aus verschiedenen Literalen und so genannten *opaque expressions* darstellt. Diese Ausdrücke sind Zeichenketten, die mit einer Sprache verknüpft sind, z. B. Java Quellcode oder OCL Ausdrücke, die einen eigenen Validator besitzen (wie der Java Compiler oder der OCL Checker). Eine komplette Definition ist in [OMG09b], S. 28ff und S. 101ff zu finden

Ausgezeichneter Typ	existenzabhängige Elemente (Typ)
`Association`	eingehender Element-Import (`ElementImport`), Verallgemeinerungen von Sub-Assoziationen (`Generalization`)
`Class`	eingehender Element-Import (`ElementImport`), Verallgemeinerungen von Sub-Klassen (`Generalization`)
`Package`	eingehende Importbeziehungen (`ElementImport` oder `PackageImport`), eingehende Paketverschmelzungsbeziehungen (`PackageMerge`)

Tabelle 6.2: Propagationstabelle für konkrete Elemente des UML Klassendiagramms (Regel 3)

```
1 context DirectedRelationship:
2 Collection featureStereotypes = self.target.stereotype->select(
    stereotype.name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.stereotype->exists(s: Stereotype | s.name = ''VariableElement''
    and s.id->includes(ids))
```

Listing 6.2: **Regel 3:** Propagation bei annotiertem Ziel einer gerichteten Beziehung.

Obige Regel besagt: „Ist das Ziel einer gerichteten Beziehung mit einem Stereotyp *VariableElement* versehen, so muss die Beziehung selber ebenfalls mit einem Stereotyp *VariableElement* ausgezeichnet werden, der mindestens die gleichen Werte für die tagged values enthält.“
Jede `gerichtete Beziehung` (`DirectedRelationship`) muss eine Quelle und ein Ziel besitzen ([OMG09b], p. 25). Die Quelle einer gerichteten Beziehung ist immer deren Besitzer, so dass aufgrund von Regel 2 (Listing 6.1) eine mit einem Stereotyp versehene Quelle eine ausgezeichnete gerichtete Beziehung impliziert. Die vorliegende Regel fordert nun selbiges auch für mit Stereotypen versehene Ziele einer gerichteten Beziehung. Die Abhängigkeiten für die konkreten Elemente sind in Tabelle 6.2 ersichtlich. Diese Tabelle zeigt im Gegensatz zu Tabelle 6.1 die „horizontalen“ Beziehungen im Metamodell.

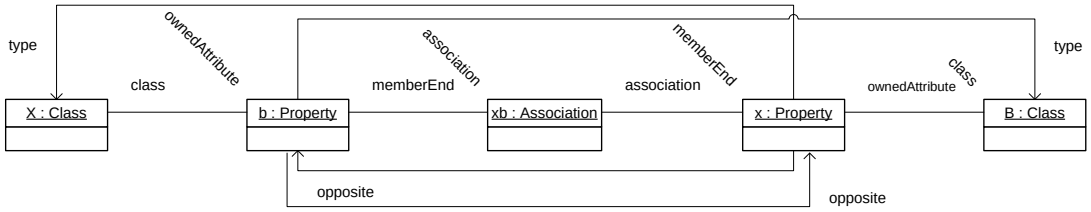

Abbildung 6.9: Objektdiagramm des Metamodells zum Klassendiagramm aus Abbildung 6.7.

```
1 context Association:
2 Collection featureStereotypes = self.memberEnd.stereotype->select(
    stereotype.name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.stereotype->exists(s: Stereotype | s.name = ''VariableElement''
    and s.id->includes(ids))
```

Listing 6.3: **Regel 4:** Propagation bei annotiertem Assoziationsende.

Diese Regel besagt: „Eine Assoziation muss mit einem Stereotyp *VariableElement* ausgezeichnet werden, falls ein Assoziationsende ebenfalls ausgezeichnet ist.“ Auch hier werden die tagged values übernommen.
Jede `Assoziation` (`Association`) erfordert wenigstens zwei Assoziationsenden. Ist nunmehr in einer Klasse ein Attribut mit einem Stereotyp versehen, so muss dieser Stereotyp an die Assoziation weiter propagiert werden, die an diesem Attribut endet. Abbildung 6.9 zeigt einen Ausschnitt des Objektdiagramms zum Klassendiagramm aus Abbildung 6.7. Wäre im Klassendiagramm nun das Attribut **b** der Klasse **X** ausgezeichnet, so muss folglich auch die Assoziation **xb** ausgezeichnet werden, da andernfalls bei einer Konfiguration, in der **b** nicht enthalten ist, die Assoziation nur ein Rollenende besitzt.

```
1 context Association:
2 Collection featureStereotypes = self.stereotype->select(stereotype.
    name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.memberEnd->forAll( m: Property | m.stereotype->exists(s:
    Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
    )
```

Listing 6.4: **Regel 5:** Propagation auf Assoziationsende(n), falls diese navigierbar sind.

Regel 5 besagt: „Wenn eine Assoziation mit dem Stereotyp *VariableElement* ausgezeichnet ist, so müssen auch ihre Enden ausgezeichnet und die tagged values übernommen

werden.“
Ist das Ende einer Assoziation (`Association`) navigierbar, so gehört es zu der entsprechenden Klasse (über die `ownedElement` Beziehung). In diesem Fall erfordert eine mit Stereotypen ausgezeichnete Assoziation, dass auch die entsprechenden Assoziationsenden ausgezeichnet werden. Das Objektdiagramm des UML Metamodells in Abbildung 6.9 zeigt, dass eine Assoziation immer zwei Rollenenden besitzt. Ist nun die Assoziation mit einem Stereotyp ausgezeichnet und nicht in der gewünschten Konfiguration enthalten, so dürfen auch die Rollenenden der Assoziation nicht in der Konfiguration enthalten sein. Da Rollenenden bei der Transformation in Java Quelltext durch Attribute der Klasse ausgedrückt werden, wäre die Assoziation somit im Quelltext noch vorhanden, wenn die Rollenenden im Modell während des Konfigurationsschritts nicht herausgefiltert werden.

```
1 context Property:
2 Collection featureStereotypes = self.type.stereotype->select(
    stereotype.name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.stereotype->exists(s: Stereotype | s.name = ''VariableElement''
    and s.id->includes(ids)))
```

Listing 6.5: **Regel 6:** Propagation bei annotiertem Typ .

Regel 6 bedeutet: „Ist ein Typ ausgezeichnet, so müssen auch alle Elemente dieses Typs mit einem entsprechenden Stereotyp samt tagged values versehen werden.“
Eine analoge Regel gilt ebenso für Rückgabetypen und Parametertypen von Operationen. Auch wird damit der Fall abgedeckt, dass das Ziel einer gerichteten Assoziation mit einem Stereotyp ausgezeichnet ist. Falls eine Klasse, die Endpunkt einer uni-direktionalen Assoziation ist, mit einem Stereotyp versehen ist, gibt es in dieser Klasse kein Attribut, das eine Referenz auf die Klasse, von der die Beziehung ausgeht enthält, da diese Richtung nicht navigierbar ist (siehe [OMG09b], S. 123ff). Daher ist das Attribut der Quell-Klasse, das die ausgezeichnete Klasse referenziert nur über die `type`-Assoziation, die in dieser Regel verwendet wird, mit dieser verbunden.

Regeln bei Verletzungen des Metamodells für Storydiagramme

Die im vorigen Abschnitt vorgestellten Regeln sind allgemeingültig für alle auf der UML basierenden Modellierungssprachen. Der folgende Abschnitt beschreibt Konsistenzregeln, die speziell für die Verhaltensmodellierung mittels Storydiagrammen in Fujaba gelten. Neben den Story Pattern, die Instanzen von Objekten enthalten und mittels Graphersetzungsregeln Änderungen am Objektgraph ausdrücken, erlaubt Fujaba auch die Verwendung von beliebigem Text in Bedingungen, Kollaborationsaufrufen und sogar in der Verwendung von sogenannten *Statement Activities*. Dies sind Platzhalter, in die handgeschriebener Quelltext eingefügt wird. Im Fujaba Metamodell hat jedes Modellelement Referenzen zu allen Instanzen, d.h. eine Klasse hat Referenzen auf all ihre

Objekte, ein Attribut hat Referenzen auf all seine Verwendungen in Attributzuweisungen, eine Assoziation hat Referenzen auf all ihre Instanzen in Story Pattern. Dies führt zu Regel 7:

```
1 context UMLClass, UMLAssoc, UMLAttr:
2 Collection featureStereotypes = self.stereotype->select(stereotype.
    name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.instances->forAll( i: Instance | i.stereotype->exists(s:
    Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
    )
```

Listing 6.6: **Regel 7:** Propagation auf Instanzen.

Regel 7 bedeutet: „Ist ein UML Element mit einem Stereotyp *VariableElement* versehen, so müssen auch sämtliche Instanzen dieses Elements damit ausgezeichnet und die tagged values übernommen werden.“

```
1 context UMLMethod:
2 Collection featureStereotypes = self.stereotype->select(stereotype.
    name = ''VariableElement'')
3 Collection ids = featureStereotypes.id->flatten()
4 inv: featureStereotypes->notEmpty()
5 implies
6 self.storydiag.stereotype->exists(s: Stereotype | s.name = ''
    VariableElement'' and s.id->includes(ids))
```

Listing 6.7: **Regel 8:** Propagation auf Methodenimplementierungen.

Jedes Storydiagramm spezifiziert das Verhalten einer einzelnen Methode. Wird nun im Klassendiagramm eine Methode mittels eines Stereotyps *VariableElement* ausgezeichnet, so muss folglich auch die zugehörige Methodenimplementierung mit dem Stereotyp und den zugehörigen tagged values ausgezeichnet werden.

```
1  Classifier::children(): Set(Classifier);
2  children = generalization.specific
3  Classifier::allChildren(): Set(Classifier);
4  allChildren = self.children()->union(self.children()->collect(c | c.
       allChildren()))
5
6  context UMLClass:
7  let superClasses = self.allParents()
8  let subClasses   = self.allChildren()
9  let subClassInstances = subClasses.methods().storydiag.ownedElements
       ->select(o: UMLObject | o.attrs->exists(superClasses->forAll(s:
       UMLClass | s.ownedAttributes.instances->forAll())))
10
11 Collection featureStereotypes = self.stereotype->select(stereotype.
       name = ''VariableElement'')
12 Collection ids = featureStereotypes.id->flatten()
13 inv: featureStereotypes->notEmpty()
14 implies
15 subClassInstances->forAll( i:UMLObject | i.stereotype->exists(s:
       Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
       )
```

Listing 6.8: **Regel 9:** Propagation auf Instanzen von Attributen aus Oberklassen, die in Methodenimplementierungen von Subklassen verwendet werden.

Regel 9 löst das Problem eines unterbrochenen Vererbungsbaums, indem alle Instanzen von Elementen ausgezeichnet werden, die nicht mehr vererbt werden. In [Baa03] wird gezeigt, dass das in der UML Spezifikation definierte Constraint **allParents()** den transitiven Abschluss aller direkten und indirekten Oberklassen einer Klasse bildet. Die Methode **allChildren()** wurde analog dazu formuliert. Die Regel besagt: „Ist ein UML Element mit einem Stereotyp *VariableElement* ausgezeichnet, so müssen alle Instanzen von erreichbaren Kindern (`owned elements`) ihrer Superklassen in Storydiagrammen aller Subklassen ebenfalls ausgezeichnet und die tagged values übernommen werden.“

```
1  context UMLObject:
2  Collection featureStereotypes = self.stereotype->select(stereotype.
       name = ''VariableElement'')
3  Collection ids = featureStereotypes.id->flatten()
4  inv: featureStereotypes->notEmpty()
5  implies
6  self.revSource->forAll( l:UMLLink | l.stereotype->exists(s:
       Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
       )
7  and
8  self.revTarget->forAll( t:UMLLink | t.stereotype->exists(s:
       Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
       )
9  and
10 self.attrs->forAll( a:UMLAttrExprPair | a.stereotype->exists(s:
       Stereotype | s.name = ''VariableElement'' and s.id->includes(ids))
       )
```

Listing 6.9: **Regel 10:** Propagation auf den Kontext von Instanzen.

Diese Regel stellt die syntaktische Korrektheit von Storypatterns sicher, bei denen auf feingranularer Ebene Stereotypen für einzelne Objektinstanzen vergeben wurden. Da ein Objekt (**UMLObject**) in einem Storymuster mit anderen Objekten durch Links (**UMLLink**) verbunden sein kann, muss sichergestellt werden, dass diese Links ebenfalls mit dem Stereotyp *VariableElement* und den zugehörigen tagged values ausgezeichnet werden, sofern das Objekct ausgezeichnet ist. Außerdem kann ein Objekt Attribut-Wert-Zuweisungen enthalten. Diese müssen dann ebenfalls entsprechend ausgezeichnet werden.

Werden nun die oben aufgezählten Regeln auf das Beispiel in Abbildung 6.7 angewendet, so ergibt sich folgendes Ergebnis: Regel 2 impliziert, dass die Attribute **x** und **y** ebenfalls mit Stereotypen ausgezeichnet werden, ebenso wie die Vererbungskante zu Klasse **A**. Die Vererbungskante von Klasse **C** zu Klasse **B** ist analog zu Regel 3 ausgezeichnet. Die Ausführung von Regel 4 bewirkt das Vorhandensein von Stereotypen an beiden Assoziationen. Die gegenüberliegenden Assoziationsenden (**b**) sind aufgrund von Regel 5 mit Stereotypen versehen. Auf das Beispiel angewendet bedeutet dies, dass bei einer Konfiguration, die das Feature *sampleFeature* (und somit die Klasse **B**) nicht enthält, ein wohlgeformtes Modell bzw. syntaktisch korrekter Quellcode entsteht. Die Anwendung der Regeln ist in Abbildung 6.10 dargestellt.

6.3.5 Erfüllbarkeit einer Konfiguration

Neben den im vorigen Abschnitt definierten Propagationsregeln, die anhand von Abhängigkeiten im Metamodell ermittelt wurden, ist auch eine Erfüllbarkeitsregel implementiert, die überprüft, ob die Konfiguration der Features auch erfüllbar ist. Da in der aktuellen Version des Werkzeugs eine Zuordnung mehrerer Features zu einem Modell-

Abbildung 6.10: Anwendung der Regeln auf das Klassendiagramm aus dem Beispiel.

element nur mittels einer Konjunktion möglich ist, kann das Modellelement nicht mit mehreren, sich gegenseitig ausschließenden Features ausgezeichnet werden. In diesem Falle muss dem Anwender die fehlerhafte Zuordnung angezeigt werden, da ansonsten für das Modellelement niemals Code erzeugt wird.

Regel 11. *Seien f_1 und f_2 zwei Features und FG die Menge sich gegenseitig ausschließender Features aus einer Gruppe. $d(f)$ sei die Menge aller mit einem Feature f annotierten Elemente des Domänenmodells. Dann gilt:*

$$\forall f_1, f_2 \in FG : f_1 \neq f_2 \Rightarrow d(f_1) \cap d(f_2) = \emptyset \tag{6.3}$$

6.3.6 Erkennen von möglichen Problemen

Neben den oben genannten schwerwiegenden Fehlern, die bei der Vergabe von Feature Annotationen auftreten können, gibt es noch eine Reihe von Warnungen, die dem Benutzer angezeigt werden.

Unvollständiges Domänenmodell

Sind beispielsweise nicht alle im Featuremodell vorhandenen optionalen Features auf Elemente des Domänenmodells abgebildet, so deutet dies darauf hin, dass das Domänenmodell unter Umständen noch nicht vollständig ist und gewisse Teile noch implementiert werden müssen.

Regel 12. *Sei $f \in F$ ein optionales Feature des Featuremodells. Sei $d(f)$ die Menge aller mit einem Feature f annotierten Elemente des Domänenmodells. Das Domänenmodell ist vollständig implementiert, wenn gilt:*

$$\nexists f \in F : d(f) = \emptyset \tag{6.4}$$

Inkonsistente Hierarchie

Durch Analyse der Hierarchie des Featuremodells und der Vererbungshierarchie im Domänenmodell können widersprüchliche Featureannotationen ermittelt werden. Ist beispielsweise eine Unterklasse mit einem übergeordneten Feature annotiert, während die zugehörige Oberklasse ein Kindelement des der Unterklasse zugeordneten Features enthält, so wird dies dem Benutzer angezeigt.

Um inkonsistente Hierarchien bereits bei der Annotation des Domänenmodells zu vermeiden, kann der Benutzer durch entsprechende Vorschläge für Feature Annotationen unterstützt werden. Hierzu wird ebenfalls die Vererbungshierarchie analysiert. Ist die

Oberklasse bereits mit Features versehen, die Kindelemente besitzen, so sind diese Kinder Kandidaten für Featureannotationen an den Unterklassen. Analog dazu wird bei bereits annotierten Unterklassen das möglicherweise übergeordnete Feature als Vorschlag für eine Featureannotation der Oberklasse unterbreitet.

6.4 Konfigurierung des Domänenmodells zur Entwicklungszeit

Nach der Abbildung des Featuremodells auf das Domänenmodell durch Auszeichnung aller Modellelemente, die Variationspunkte realisieren durch die entsprechenden Stereotypen, sind verschiedene Möglichkeiten denkbar, um eine Konfiguration des Domänenmodells zur Entwicklungszeit darzustellen.

6.4.1 Visualisierung im Modell

Die einfachste und naheliegendste Variante ist die Visualisierung einer Konfiguration des Domänenmodells durch Hervorheben von in der Featurekonfiguration vorhandenen Elementen, bzw. durch Ausblenden von Elementen, die nicht Teil der aktuellen Konfiguration sind. Der Modellierer hat somit zu jeder Zeit ein Hilfsmittel zur Verfügung, um den jeweiligen Ausschnitt des Modells zu überwachen.

6.4.2 Erzeugen von Quelltext

Eine weitere Möglichkeit, eine Konfiguration des Domänenmodells abzubilden ist die Erzeugung von konfiguriertem Quellcode. Dies bedeutet, dass nicht etwa das komplette Domänenmodell in Quelltext transformiert wird, sondern nur die Teile, die die aktuell gewählte Konfiguration beschreiben. Dafür wird ein Präprozessor benötigt, der vor der eigentlichen Codeerzeugung alle nicht in der Konfiguration vorhandenen Modellelemente von der Codegenerierung ausnimmt. Dieser Prozess ist vergleichbar mit der bedingten Übersetzung in Programmiersprachen. Elemente, für die die Bedingungen nicht gelten, werden vom Präprozessor entfernt, bevor der eigentliche Compiliervorgang stattfindet. Natürlich gelten dann hierfür auch die gleichen Nachteile, die auch für Compiler-Präprozessoren gelten: Es ist sehr einfach, syntaktisch fehlerhaften Quelltext zu erzeugen.

6.4.3 Erzeugen eines konfigurierten Modells

Die letzte Möglichkeit der Darstellung einer Konfiguration ist die Transformation des Domänenmodells in ein konfiguriertes Domänenmodell. Das konfigurierte Domänenmodell enthält nach dem Transformationsschritt alle Modellelemente, die Bestandteil der aktuellen Konfiguration sind. Gegenüber der Variante der Erzeugung von konfiguriertem Quelltext hat der Modellierer die Möglichkeit, auf einer höheren Abstraktionsebene Änderungen vorzunehmen, falls dies nötig ist. Nach Abschluss der Änderungen kann dann wiederum (ausführbarer) Quelltext erzeugt werden.

6.5 Werkzeuge: Funktionalität

Das für die Modellierung des Domänenmodells eingesetzte Werkzeug Fujaba bietet per se keinerlei Unterstützung für Featuremodelle. Das Fujaba Metamodell erlaubt zwar die Definition und Verwendung von Stereotypen für alle Metamodellelemente und auch die Definition neuer Stereotypen, allerdings wird keine Unterstützung für *tagged values* geboten. Der hier vorgestellte Ansatz kann aber dennoch leicht modifiziert verwendet werden, da Fujaba die Verwendung von Annotationen für beliebige Modellelemente unterstützt. Das Basiselement des UML Metamodells von Fujaba, **UMLIncrement** besitzt eine bidirektionale Assoziation zu beliebig vielen Elementen vom Typ **UMLTag** (hiermit werden in Fujaba Annotationen realisiert), wie der Ausschnitt des Metamodells in Abbildung 6.11 zeigt.

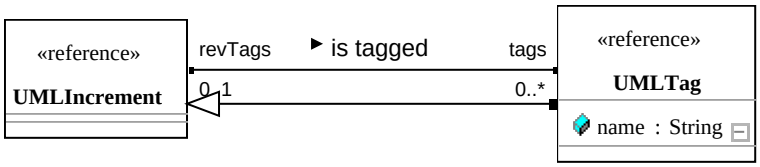

Abbildung 6.11: Ausschnitt aus dem Fujaba Metamodell.

Die Zuordnung von Features zu Modellelementen findet in der im Rahmen dieser Arbeit implementierten Lösung durch Vergabe von Annotationen mit dem eindeutigen Bezeichner des Features für das jeweilige Modellelement statt. Im weiteren Verlauf werden diese Annotationen als **Feature Tags** bezeichnet. Die Zuordnung von mehreren Features an ein Modellelement wird durch die Vergabe von mehreren entsprechenden Feature Tags realisiert. In der aktuellen Implementierung wird ein solcher Ausdruck als Konjunktion der einzelnen Features interpretiert. Dieser Abschnitt erläutert die Funktionalität der entwickelten Werkzeuge zur Abbildung der Features auf die Modellelemente [BD09b], der folgende Abschnitt behandelt Struktur und Aufbau, sowie die Implementierung der Propagationsregeln [BD09a] in einem Werkzeug zur Unterstützung der drei verschiedenen Arten der Konfiguration des Domänenmodells. Wie der Paketdiagrammeditor (siehe Kapitel 5) sind auch diese neu entwickelten Werkzeuge in erster Linie für den Einsatz mit der Eclipse-Version von Fujaba gedacht. Sie können auch mit der reinen Java Version von Fujaba verwendet werden, allerdings stehen dort Eclipse-spezifische Erweiterungen, wie etwa die Baumansicht für das Feature-Modell und die jeweilige Konfiguration, nicht zur Verfügung. Abbildung 6.12 zeigt eine Übersicht der Funktionen der entwickelten Werkzeuge und deren Einbettung in die bestehende Werkzeuglandschaft. Die Kernfunktionen des Werkzeugs sind:

Import von Feature Modellen und Konfigurationen aus Werkzeugen zur Featuremodellierung.

Auszeichnung von Modellelementen mit Feature Tags.

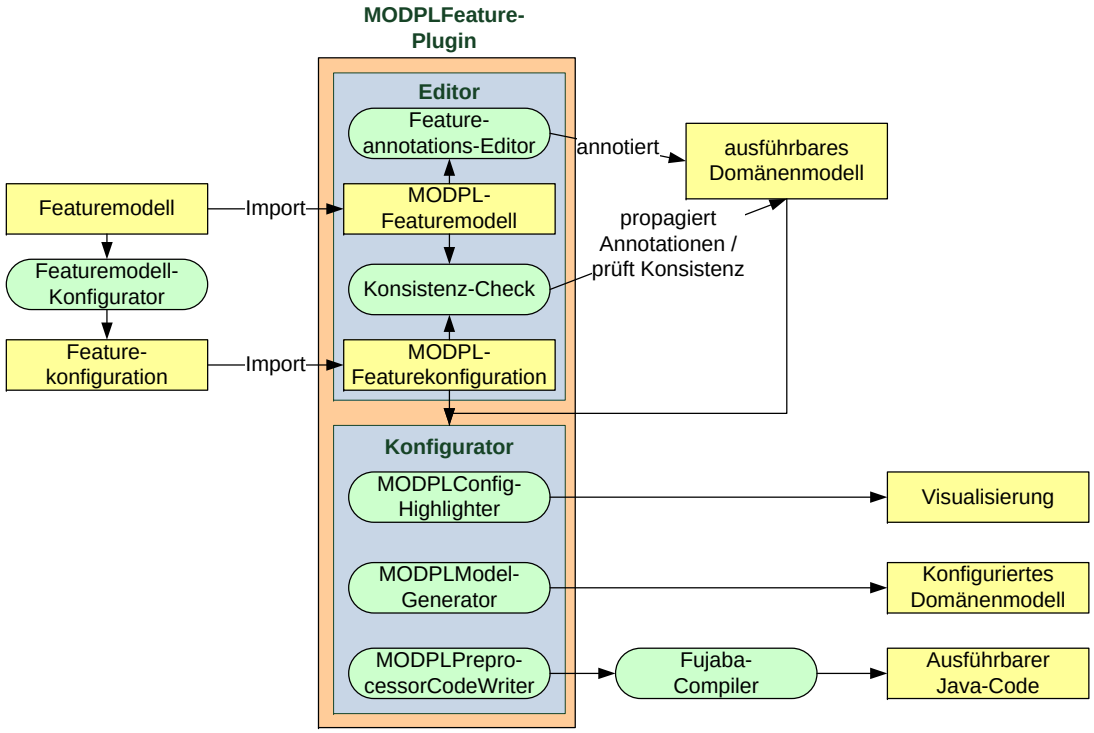

Abbildung 6.12: Überblick über die Funktionen des erstellten Werkzeugs.

Visualisierung von Features bzw. den zugehörigen Modellelementen, sowie von durch Regelanwendung automatisch hinzugefügten Feature Tags.

Konsistenzprüfung von vergebenen Features.

Propagation von Feature Tags auf abhängige Elemente, analog zu den in Abschnitt 6.3.4 aufgestellten Konsistenzbedingungen und Visualisierung der automatisch durch Regelanwendung hinzugefügten Feature Tags.

Konfigurierung des Modells. Die Konfigurierung kann hierbei auf drei verschiedene Arten erfolgen: (1) durch Visualisierung im Modell, (2) durch Generierung von konfiguriertem Quellcode oder (3) durch Erzeugung eines konfigurierten Modells.

Es wurde das Werkzeug **MODPLFeaturePlugin** entwickelt, mit dessen Hilfe Feature Modelle und Konfigurationen, die mit externen Werkzeugen erstellt wurden, importiert und auf Elemente des mit Fujaba erstellten Domänenmodells abgebildet werden können. Darüber hinaus bietet das Werkzeug Unterstützung für den Modellierer durch Anwendung und Visualisierung der Propagationsregeln aus Abschnitt 6.3.4. Dies bedeutet, dass dem Benutzer nach Anwendung der Propagationsregeln alle automatisch hinzugefügten Feature Tags im Modell angezeigt werden. Diese Visualisierung erfolgt hierbei keineswegs automatisch, sondern kann vom Benutzer bei Bedarf aktiviert bzw. deaktiviert werden. Aufgrund der möglicherweise recht großen Anzahl von automatisch

hinzugefügten Feature Tags ist dies auch sinnvoll, um die Übersichtlichkeit der einzelnen Diagramme nicht zu verschlechtern. Desweiteren wird überprüft, ob auch alle Features auf Modellelemente abgebildet wurden. Dies ist notwendig, um unvollständige Feature Annotationen zu erkennen. Ebenso werden inkonsistente Feature Annotationen überwacht und dem Benutzer angezeigt. Ein Beispiel für eine inkonsistente Feature Annotation wäre etwa die Vergabe eines Features an eine Subklasse, wenn die Vaterklasse bereits Kindelemente des neu zu vergebenden Features enthält, oder auch die Vergabe von zwei sich gegenseitig ausschließenden Features an ein und dasselbe Modellelement. Sind bereits Klassen mit Features ausgezeichnet, so können für Klassen, die sich in der entsprechenden Vererbungshierarchie befinden, Vorschläge für mögliche zu vergebende Features gemacht werden. Weiterhin werden im Baum ausgewählte Features im Fujaba Modell farblich hervorgehoben, um die Übersichtlichkeit zu erhöhen. Selbstverständlich können in dem hier vorgestellten modellgetriebenen Prozess zur Entwicklung von Produktlinien die Features nicht nur auf Fujaba Modellelemente abgebildet werden, sondern auch auf Elemente des Paketdiagramms. Die entsprechende Funktionalität wurde in den in Kapitel 5 vorgestellten Paketdiagrammeditor integriert. Abschnitt 6.6.4 beschreibt die dafür notwendigen Erweiterungen des Paketdiagrammeditors.

Der Konfigurator wird dafür verwendet, um ein in Fujaba erstelltes und mit Feature Tags ausgezeichnetes Domänenmodell, mit Hilfe der von MODPLFeaturePlugin bereitgestellten Daten zu konfigurieren. Diese Konfigurierung umfasst dabei die reine Visualisierung einer Konfiguration im Fujaba Editor, mit deren Hilfe der Benutzer sich zu jeder Zeit einen Überblick über die gewählte Konfiguration verschaffen kann. Weiterhin stellt der Konfigurator einen Präprozessor für den Codegenerierungs Mechanismus von Fujaba bereit, der analog zur bedingten Übersetzung bei Programmiersprachen, alle nicht zur Konfiguration gehörenden Modellelemente herausfiltert und somit die Erzeugung von Quelltextfragmenten für diese Elemente verhindert. Außerdem wird ein Modelltransformator angeboten, der das annotierte Domänenmodell anhand einer geladenen Konfiguration in ein konfiguriertes Modell überführt. Der Benutzer kann anschließend mit diesem Modell entweder weiterarbeiten und zusätzliche Funktionalität hinzufügen, oder direkt aus diesem Modell ebenfalls ausführbaren Quelltext generieren.

6.6 Werkzeuge: Realisierung

In diesem Abschnitt wird die technische Realisierung der beiden im vorigen Abschnitt vorgestellten Plugins erörtert und die Einbettung in die Werkzeugkette dargestellt.

6.6.1 MODPLFeaturePlugin

Klassendiagramm

Abbildung 6.13 zeigt das Klassendiagramm des erstellten Werkzeugs. MODPLFeaturePlugin wurde nahezu vollständig modellgetrieben mit Hilfe von Fujaba entwickelt. Über die Plugin Infrastruktur von Fujaba ist eine nahtlose Einbettung in die Fujaba Umgebung

gewährleistet. Die Zentrale Klasse ist **MODPLFeaturePlugin**. Sie erbt von **AbstractPlugin** aus dem Fujaba Projekt und bietet mit der Methode **initialize** den Prozedureinstiegspunkt für das Plugin. Die Klasse besitzt Assoziationen zu den Import Modulen (**ImportModule**), der Klasse, die die Propagationsregeln implementiert (**TagPropagator**), den jeweiligen Featuremodellen (**FeatureModel**) und Konfigurationen (**FeatureConfiguration**) und zu den hervorgehobenen Elementen (**FeatureTag**). Featuremodelle, Konfigurationen und Propagationsregeln sind jeweils eindeutig für ein geladenes Fujaba Modell. Da Fujaba in der Lage ist, mehrere Projekte gleichzeitig zu bearbeiten, erfolgt die Zuordnung über den eindeutigen Projektnamen. Im Klassendiagramm ist dies durch die qualifizierten Assoziationen ausgedrückt. Das Propagationsmodul hat Referenzen auf die automatisch hinzugefügten Feature Tags, sowie alle Elemente des Fujaba Metamodells, die noch nicht abgearbeitet wurden. Die im Fujaba Metamodell für Annotationen zuständige Klasse **UMLTag** wurde durch die Klasse **FeatureTag** spezialisiert. Es wurde ein Parameter hinzugefügt, der anzeigt, ob das Feature Tag vom Benutzer hinzugefügt wurde, oder durch die Propagationsregeln. Die von Fujaba für Annotationen erwartete Syntax ist ähnlich der für Annotationen in Java. Jede Annotation beginnt mit dem Symbol „@“, gefolgt von einer Zeichenkette, die den Namen repräsentiert und beliebig vielen Schlüssel-Wert Paaren, die von runden Klammern eingeschlossen werden, z.B. **@name(key=value)**. Bei der Vergabe von Feature Tags, ist der Name immer „feature“ und es gibt nur ein einziges Schlüssel-Wert Paar, den eindeutigen Bezeichner des Features aus dem Featuremodell, also z.B. **@feature(id="directedDeltas")**. Bei der Konkatenation mehrerer Features werden die entsprechenden Feature Tags zu dem Modellelement hinzugefügt.

Die Klasse **MODPLFeaturePlugin** verwendet das *Singleton*-Muster [FF04] und bietet Funktionen zum Aufruf der Propagationsregeln (z. B. **propagateAutomatically**, **removePropagatedTags**), zur Konsistenzprüfung (**checkExclusiveOrViolations**, **checkForUnusedFeatures**, **highlightViolations**), zur Visualisierung von Features und Modellelementen (**showTagsForFeature**) und zur Unterstützung des Modellierers bei der Vergabe von Feature Tags (**suggestFeatureTagForPackage**, **suggestTagsForSuperclass**).

Datenimport von FeaturePlugin

Da das Werkzeug *FeaturePlugin* (siehe 2.3.1) zur Erstellung von Featuremodellen in sich abgeschlossen ist und keinerlei Erweiterungspunkte bietet, kann ein Datenaustausch zwischen den einzelnen Werkzeugen nur durch Im- und Export über Dateien erfolgen. FeaturePlugin bietet die Möglichkeit, ein Featuremodell oder eine Konfiguration als XML Datei zu exportieren. Eine Konfiguration enthält dabei nur XML Elemente mit dem Namen *feature*, das zugrunde liegende Modell kann zudem noch XML Elemente namens *featureGroup* enthalten. Name, Kardinalität und eindeutiger Bezeichner der jeweiligen Modellelemente sind als Attribute innerhalb des jeweiligen XML Elements zu finden. Ein Ausschnitt aus einer XML Datei, die ein Featuremodell beschreibt, ist in Listung 6.10 zu sehen. Das Einlesen der XML Datei erfolgt mit Hilfe eines Import Moduls. Das für den Import von Featuremodellen bzw. Konfigurationen des Werkzeugs *FeaturePlugin* zuständige Modul ist **FMPDOMImportModule**. Diese Klasse benutzt die *DOM*

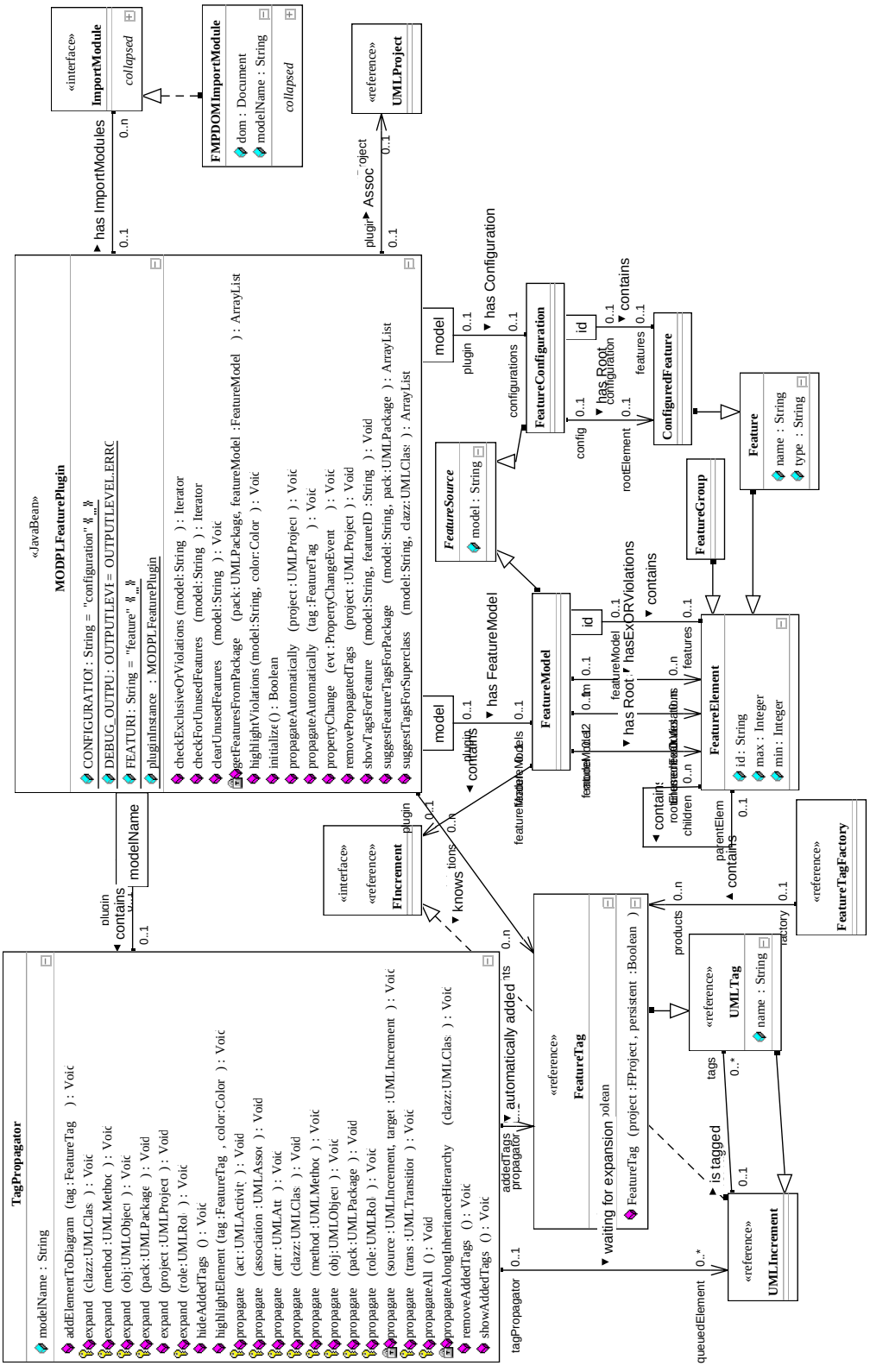

Abbildung 6.13: Das Klassendiagramm von MODPLFeaturePlugin.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<featureModel>
<feature min="1" max="1" name="MOD2-SCM" type="NONE" id="mOD2SCM">
...
 <feature min="1" max="1" name="Server" type="NONE" id="server">
...
  <feature min="1" max="1" name="Storage" type="NONE" id="storage">
   <featureGroup min="1" max="1" id="storageGroup">
    <feature min="0" max="1" name="Baseline" type="NONE" id="baseline
        ">
    </feature>
    <feature min="0" max="1" name="Deltas" type="NONE" id="deltas">
     <featureGroup min="1" max="2" id="deltaGroup">
      <feature min="0" max="1" name="Shared␣Deltas" type="NONE"
        id="sharedDeltas">
       <featureGroup min="1" max="1" id="sharedDeltaGroup">
        <feature min="0" max="1" name="Complex␣Item␣Deltas" type="
            NONE"
         id="complexItemDeltas">
        </feature>
        <feature min="0" max="1" name="Intervleaved␣Deltas" type="
            NONE"
         id="intervleavedDeltas">
        </feature>
       </featureGroup>
      </feature>
      <feature min="0" max="1" name="Directed␣Deltas" type="NONE"
       id="directedDeltas">
       <featureGroup min="1" max="1" id="directedDeltaGroup">
        <feature min="0" max="1" name="Forward␣Deltas" type="NONE"
         id="forwardDelta">
        </feature>
        <feature min="0" max="1" name="Backward␣Deltas" type="NONE"
         id="backwardDelta">
        </feature>
        <feature min="0" max="1" name="Mixed␣Deltas" type="NONE"
         id="cVSlikeDelta">
        </feature>
       </featureGroup>
      </feature>
     </featureGroup>
    </feature>
   </featureGroup>
  </feature>
...
</feature>
</featureModel>
```

Listing 6.10: Auszug aus der XML Datei mit dem Featuremodell aus Abbildung 2.3.

(Document Object Model)[3] Schnittstelle von Java[4]. DOM ist ein Standard des W3 Consortiums, der eine Programmierschnittstelle zur Verarbeitung von XML Dokumenten beschreibt. Er konkurriert mit einem zweiten in der Java Welt weit verbreitetem Standard: SAX (Simple API for XML). Im Gegensatz zu SAX erfolgt bei DOM der XML-Zugriff über einen Objektbaum, was für den Einsatzzweck im MODPLFeaturePlugin deutlich komfortabler ist.

Mit Hilfe der DOM Schnittstelle aus der Java-XML-API wird nun ein komplettes XML Dokument geparst und eine vollständige Darstellung des Dokuments im Speicher erstellt. Der Parser wird in der Java DOM Realisierung *DocumentBuilder* (**javax.xml.parsers.DocumentBuilder**) genannt. Dieser erzeugt zunächst eine Instanz des Dokuments (**org.w3c.dom.Document**) in Form einer Baumstruktur, die die Knoten des XML-Dokuments enthält. Jeder Baumknoten dieser Struktur implementiert die Schnittstelle **org.w3c.dom.Node**. Diese Baumdarstellung wird nun anhand der unterschiedlichen Knotentypen *feature* und *featureGroup* durchlaufen und es wird eine analoge Repräsentation in der von *MODPLFeaturePlugin* verwendeten Struktur erzeugt.

Die gemeinsame Oberklasse eines Featuremodells (**FeatureModel**) bzw. einer Feature Konfiguration (**FeatureConfiguration**) ist die Klasse **FeatureSource**. Sie enthält den Namen des Fujaba Modells zu dem das Modell bzw. die Konfiguration gehört als Attribut. Dieses Attribut ist der Parameter für die qualifizierten Assoziationen von **MODPLFeaturePlugin** auf die jeweiligen Modelle. Ein Featuremodell kann Feature Gruppen (**FeatureGroup**), sowie Features (**Feature**) beinhalten. **Feature** und **FeatureGroup** erben von **FeatureElement** und enthalten somit alle Attribute, die auch aus der XML Datei eingelesen werden können. Die Verknüpfung mit dem Featuremodell erfolgt ebenfalls mit Hilfe einer qualifizierten Assoziation, wobei das Attribut **id**, das den eindeutigen Bezeichner des Elements speichert, als Parameter dient. Um die Baumstruktur der Features abbilden zu können, besitzt **FeatureElement** eine Assoziation zur Darstellung von Vater-Kind-Beziehungen. Die Klasse **FeatureModel** hat weiterhin noch gerichtete Beziehungen zur Speicherung des Wurzelelements, zur Darstellung von Features, die noch nicht zur Auszeichnung im Domänenmodell verwendet wurden und zur Speicherung von Features, die gegen Konsistenzbedingungen verstoßen.

Konfigurationen (**FeatureConfiguraton**) enthalten konfigurierte Features (**ConfiguredFeature**), einer Subklasse von **Feature**. Die Zuordnung erfolgt auch hier mittels einer qualifizierten Assoziation mit dem Attribut **id** als Parameter.

Implementierung der Propagationsregeln

Die Klasse **TagPropagator** realisiert alle in Abschnitt 6.3.4 aufgestellten Regeln. Die Anwendung der Regeln erfolgt auf Instanzen des Fujaba UML-Metamodells. Für jedes geöffnete Fujaba Projekt gibt es eine Instanz des **TagPropagator**s, da die automatisch hinzugefügten **FeatureTag**s jeweils in einer Liste gespeichert werden. Dies ist notwendig, um beispielsweise die automatisch hinzugefügten FeatureTags an der Benutzerschnittstelle darstellen zu können. Um sämtliche Relevanten Modellelemente durchlaufen zu

[3] http://www.w3.org/DOM/
[4] https://jaxp.dev.java.net/

Ausgezeichneter Typ	existenzabhängige Elemente (Typ)
UMLPackage	Subpakete (**UMLPackage**) und im Paket definierte Klassen (**UMLClass**)
UMLClass	Attribute (**UMLAttr**) der Klasse, Methoden (**UMLMethod**), Rollenenden (**UMLRole**), Vererbungsbeziehungen (**UMLGeneralization**), Instanzen (**UMLObject**), Instanzen von Attributen in Oberklassen, die in Storydiagrammen von Methoden von Unterklassen auftreten.
UMLMethod	Storydiagramme, Methodenparameter (**UMLParam**)
UMLAttr	Instanzen von Attributen (**UMLAttrExprPair**)
UMLRole	zugehörige Assoziation (**UMLAssoc**), gegenüberliegendes Rollenende (**UMLRole**)
UMLAssoc	Rollenenden (**UMLRole**), Instanzen der Assoziation in Storydiagrammen (**UMLLink**)

Tabelle 6.3: Propagationstabelle für Elemente des Fujaba Metamodells

können, wird zunächst die Methode **expand** auf dem aktuellen **UMLProject** aufgerufen. Hierbei werden ausgehend vom Wurzelpaket, das das Wurzelelement für alle Fujaba Modellinstanzen darstellt, alle Pakete und darin enthaltene Klassen durchlaufen. Sind diese Modellelemente mit einem **FeatureTag** versehen, so werden sie in die Liste mit den zu bearbeitenden **UMLIncrement** Objekten eingetragen. Bei sämtlichen im Projekt vorhandenen Klassen werden zudem auch alle Kindelemente durchlaufen, also alle Attribute, Methoden und Rollenenden. Bei der Analyse jedes Rollenelements wird auch die zugehörige Assoziation bzw. das gegenüberliegenden Rollenende untersucht. Enthalten die entsprechenden Modellelemente **FeatureTag**s, so werden sie ebenfalls in die oben angesprochene Liste eingetragen. Anschließend wird die Methode **propagateAll** aufgerufen. Hier werden nun sukzessive alle Propagationsregeln angewandt, bis alle Elemente der Liste abgearbeitet sind. Je nach Typ der aktuell betrachteten Elemente werden die entsprechenden Regeln aus Abschnitt 6.3.4 verwendet. So werden etwa für alle Elemente des Typs **UMLClass** zunächst alle Attribute der Klasse mit Feature Tags versehen, anschließend alle Methoden, alle Rollen von Assoziationen, die an dieser Klasse enden, alle Vererbungsbeziehungen zu Ober- und Unterklassen und schließlich alle Instanzen der Klasse, die als Objekte in Storydiagrammen verwendet werden. Ebenso wird Regel 9 angewandt, die besagt, dass alle Attribute von Oberklassen in Storydiagrammen von Subklassen mit einem Feature Tag versehen werden müssen. Tabelle 6.3 zeigt, welche Elemente des Fujaba Metamodells abhängig von den Quelltypen mit Feature Tags ausgezeichnet werden.

Für jedes Paar von Metamodellobjekten aus Tabelle 6.3 wird dann die Methode **propagate(source : UMLIncrement, target : UMLIncrement)** aufgerufen. Sie erzeugt eine neue Instanz von **FeatureTag** mit dem eindeutigen Bezeichner des FeatureTag Objekts, das mit **source** assoziiert ist und zeichnet **target** damit aus. Für die spätere Anzeige der automatisch hinzugefügten FeatureTags wird das boolsche Attribut **propagated** auf den Wert

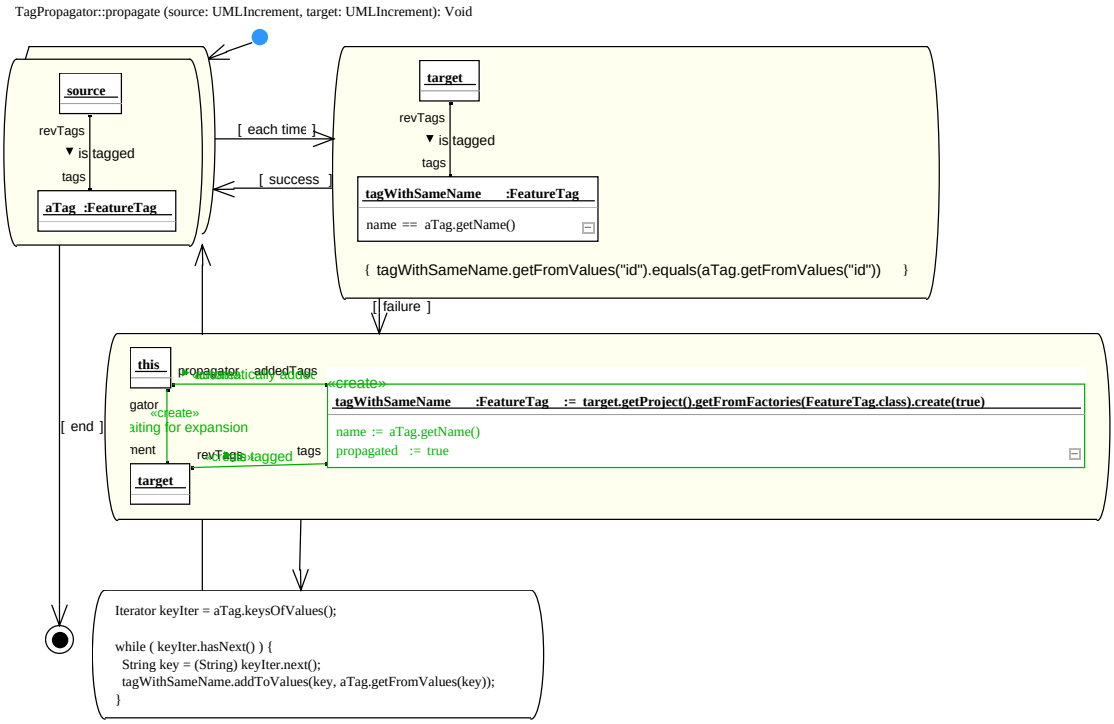

Abbildung 6.14: Das Storydiagramm zur Methode **propagate(source : UMLIncrement, target : UMLIncrement)**

true gesetzt. Abbildung 6.14 zeigt die zugehörige Methodenimplementierung in Fujaba.

6.6.2 Konfigurator

Der aktuelle Abschnitt beschreibt die Implementierung der unterschiedlichen Arten zur Konfigurierung des Domänenmodells. Sie reichen von reiner Visualisierung der aktuellen Konfiguration im Fujaba Editor, über die Generierung von Quelltext für die jeweilige Konfiguration bis hin zur Generierung eines konfigurierten Modells durch Modelltransformation. Abbildung 6.15 zeigt das Klassendiagramm des zugehörigen Werkzeugs.

Auch dieses Werkzeug wurde als Plugin für Fujaba realisiert. Die zentrale Klasse des Plugins stellt **MODPLCodeGen2PreprocessorPlugin**, die von **AbstractPlugin** aus dem Fujaba Rahmenwerk erbt, dar. Der Prozedureinstiegspunkt ist ebenfalls die Methode **initialize** und auch diese Klasse ist eine *Singleton*-Klasse. Um auf die jeweilige Konfiguration zugreifen zu können, besitzt die Klasse eine Referenz auf eine Instanz des *MODPLFeaturePlugin*, das im vorigen Abschnitt vorgestellt wurde. Desweiteren hält **MODPLCodeGen2PreprocessorPlugin** ebenfalls Referenzen auf die Klassen, die die unterschiedlichen Arten der Konfigurierung implementieren:

MODPLConfigHighlighter für die Visualisierung der Konfiguration

MODPLPreprocessorCodeWriter für die Erzeugung von konfiguriertem Quelltext

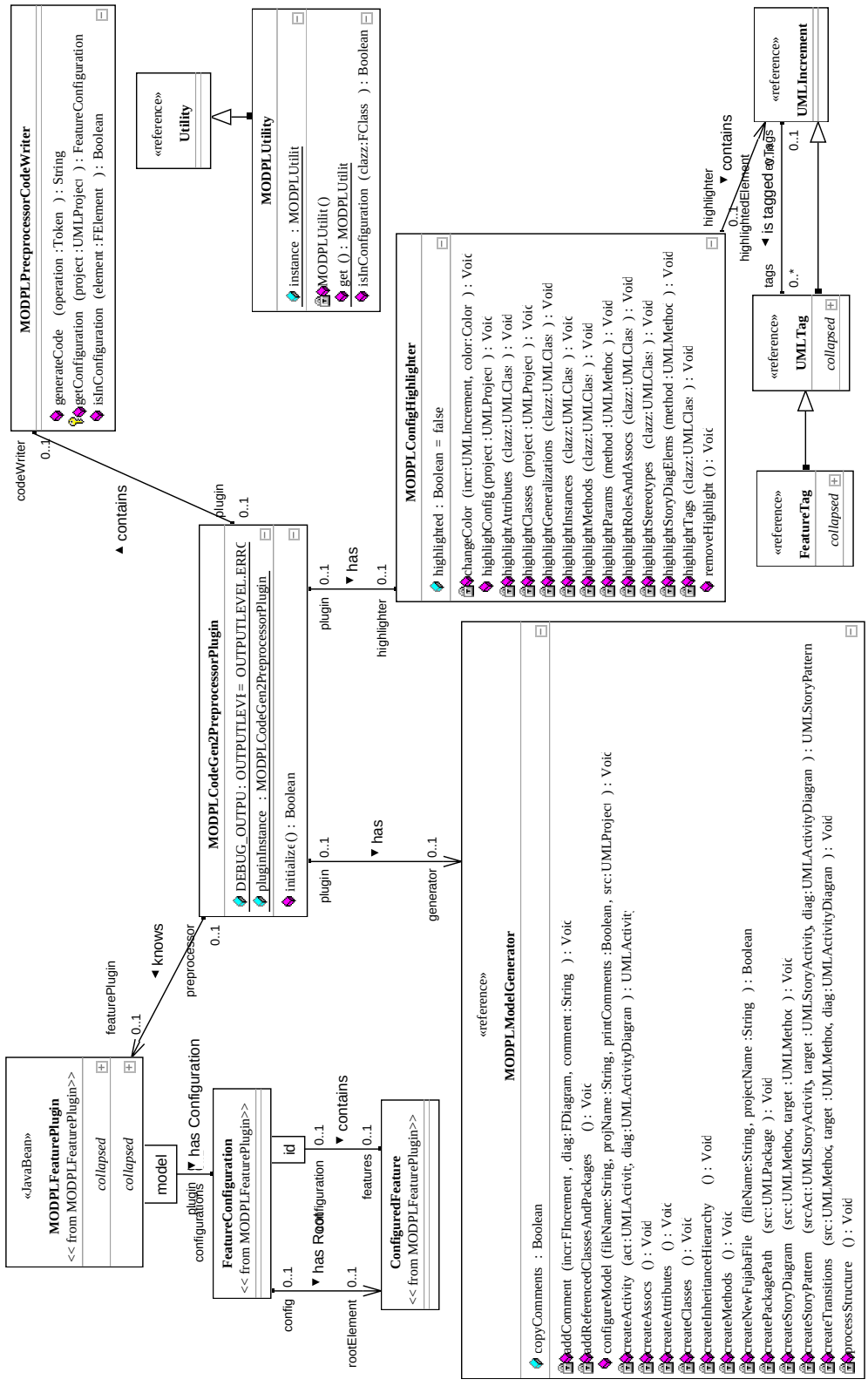

Abbildung 6.15: Klassendiagramm des Werkzeugs zur Konfiguration des Domänenmodells.

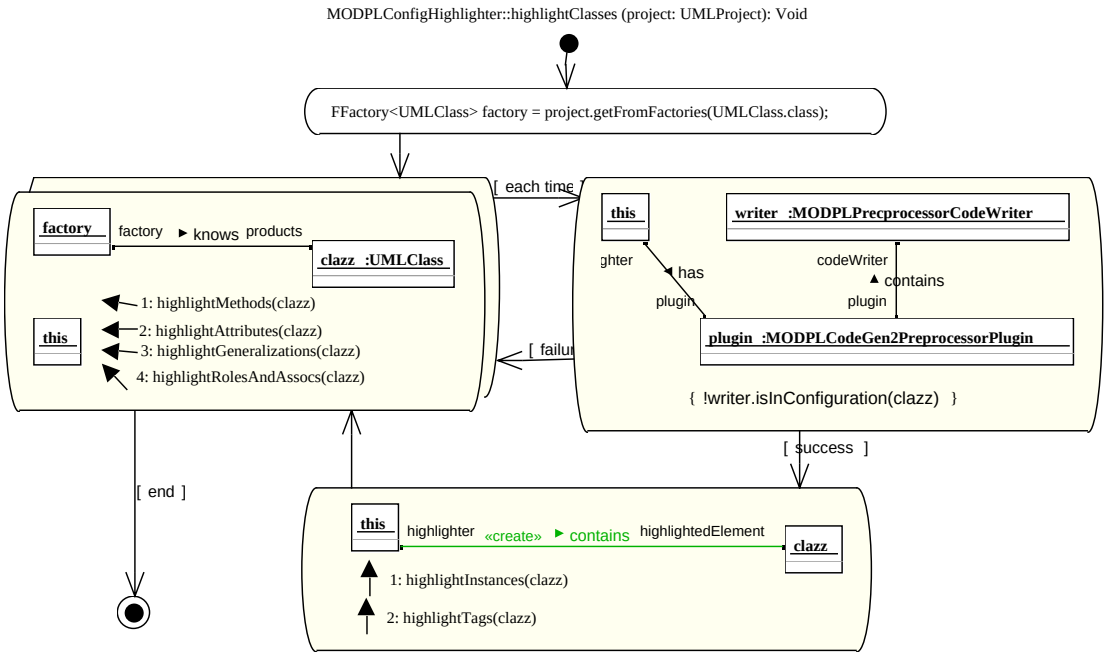

Abbildung 6.16: Storydiagramm zur Methode **highlightClasses**.

MODPLModelGenerator für die Erzeugung eines konfigurierten Modells

Visualisierung der Konfiguration

Die einfachste Art der Darstellung einer Konfiguration ist die Visualisierung im Domänenmodell. Hierbei hat der Modellierer jederzeit die Möglichkeit zur Kontrolle. Auswirkungen durch die Vergabe von Features an Modellelemente können so leicht überwacht werden. Bei der Darstellung einer Konfiguration werden alle Elemente, die nicht Teil der aktuellen Featureauswahl sind in hellgrauem Farbton dargestellt. Die Klasse **MODPLConfigurationHighlighter** implementiert die dafür benötigten Methoden. Durch Aufruf der Methode **highlightConfig** wird für das übergebene Projekt die entsprechende Konfiguration ermittelt. Anschließend werden die Propagationsregeln angewandt und alle Klassen des Projekts werden durchlaufen (siehe Abbildung 6.16).

Für jede Klasse wird geprüft, ob das Feature, mit dem die Klasse verknüpft ist, Bestandteil der gewählten Konfiguration ist. Ist dies nicht der Fall, so wird an der Benutzerschnittstelle die Farbe für das zur Darstellung des Modellelements zuständige grafische Element verändert. In diesem Fall wird die Farbe auf hellgrau gesetzt, um ein „Ausblenden“ zu simulieren. Modellelemente, die Teil der Konfiguration sind, werden unverändert dargestellt. Anschließend werden alle Instanzen der aktuellen Klasse ermittelt und verändert, sowie die mit der Klasse verbundenen Tags (Feature Tags oder auch **UMLTag**s). Auch alle ein- und ausgehenden Vererbungsbeziehungen, Attribute, Methoden und Rollenenden werden betrachtet.

Abbildung 6.17 zeigt exemplarisch die Implementierung der Methode, die Rollenenden

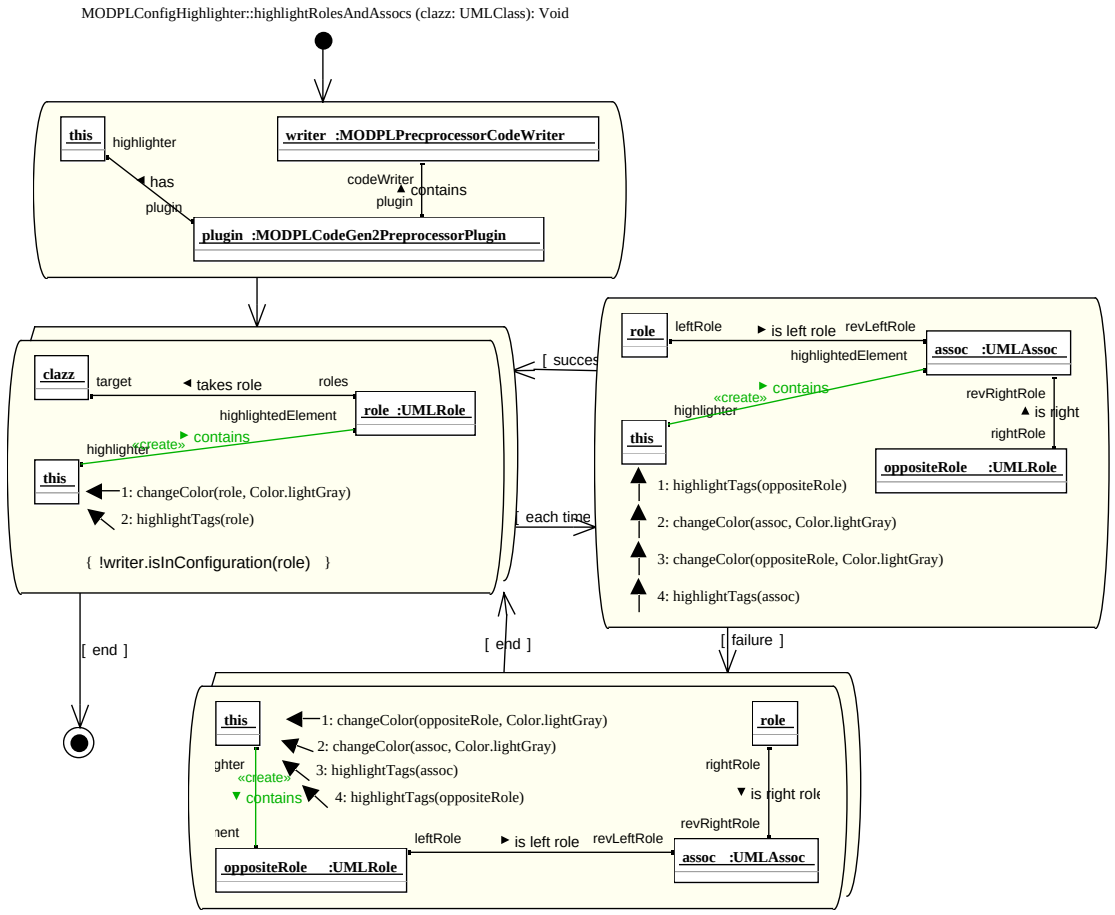

Abbildung 6.17: Storydiagramm zur Visualisierung von Rollenenden und Assoziationen, die nicht Teil der aktuellen Konfiguration sind.

und zugehörige Assoziationen, die nicht Teil der aktuellen Konfiguration sind, ermittelt und deren Darstellung entsprechend anpasst. Hierbei wird ausgehend von der übergebenen Klasse über alle Rollenenden iteriert, an denen diese Klasse beteiligt ist. Ist das mit dem Rollenende verknüpfte Feature nicht Teil der Konfiguration, so wird die Farbe des zur Darstellung des Rollenendes verantwortliche Objekt der Benutzerschnittstelle verändert. Anschließend wird die zugehörige Assoziation und das gegenüberliegende Rollenende ermittelt und beide zugehörigen grafischen Elemente ebenfalls in ihrer Farbe verändert.

Erzeugen von konfiguriertem Quelltext

Neben der visuellen Darstellung von Konfigurationen, unterstützt das Werkzeug ebenfalls die Transformation des Domänenmodells in Quelltext, der nur für die in der Konfiguration enthaltenen Modellelemente erzeugt wird. Dies wird durch einen Präprozessor für die Fujaba Codegenerierung [GSR05, BGSZ08] realisiert. Der Codegenerierungsmecha-

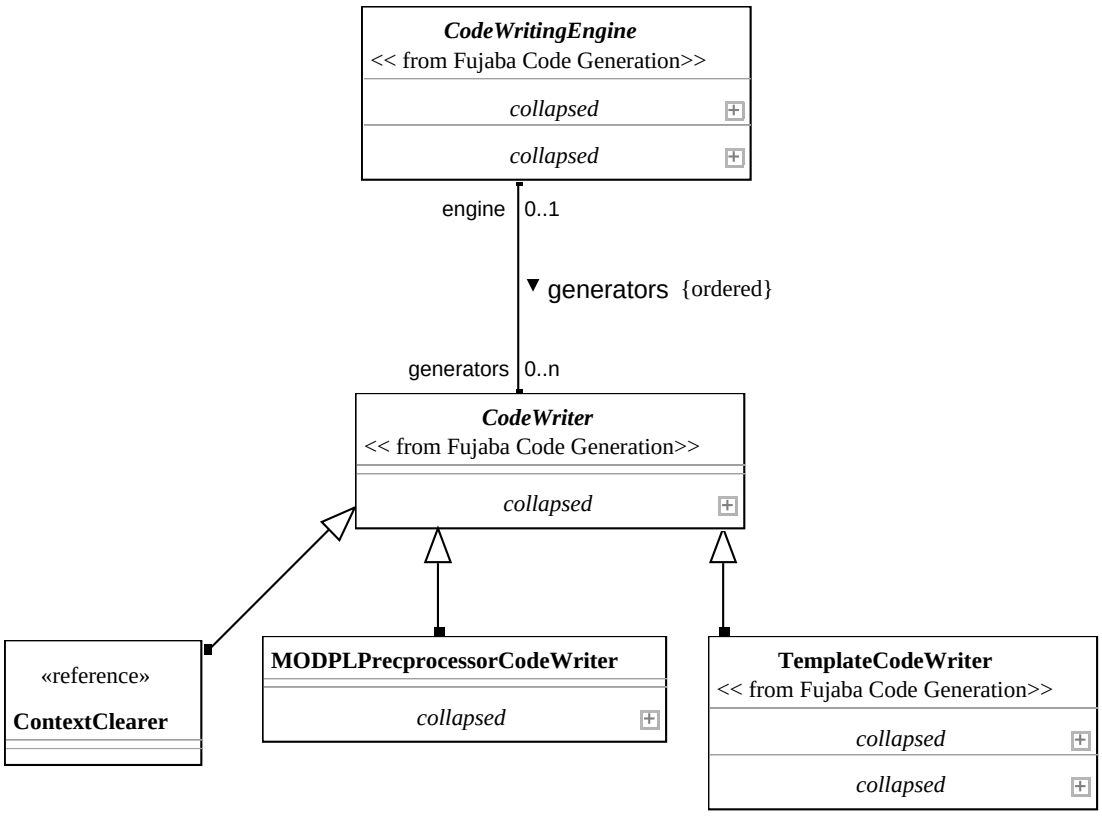

Abbildung 6.18: Architektur der Fujaba Codegenerierung.

nismus von Fujaba ist ebenfalls als Plugin konzipiert und verwendet Velocity Templates [5], um Elemente des abstrakten Syntaxgraphen auf Quelltextfragmente abzubilden. Hiermit ist es sehr einfach möglich, Quelltext für unterschiedliche Plattformen bzw. Sprachen zu erzeugen.

Das Velocity Rahmenwerk arbeitet dabei immer nach folgendem Prinzip: (1) Initialisierung, (2) Erzeugen eines Context Objekts, (3) Hinzufügen der Daten zum Context Objekt, (4) Auswahl des Templates, (5) Erzeugen der Ausgabe.
Mithilfe des Context Objekts, das als Container zwischen den verschiedenen Ebenen der Velocity Engine fungiert, können Objekte platziert werden, deren Methoden dann in den Templates ausgeführt werden können. Der Context wird wie eine Hashtabelle benutzt, d.h. die Objekte werden als Schlüssel-Wert-Paar darin abgelegt. Das Context Objekt wird von der Fujaba Codegenerierung beispielsweise dafür benutzt, um Klassen und Interfaces im Fujaba Modell zu unterscheiden (beide werden durch **UMLClass** realisiert, ein Interface wird durch den Stereotyp «**interface**» bestimmt), oder um eine Abbildung von den UML Basistypen auf die jeweiligen Basistypen der Zielsprache zu beschreiben.

Dabei sind für jede Plattform spezielle *CodeWriter* vorhanden, die nach dem *Chain-*

[5] http://velocity.apache.org/

of-Responsibility-Muster [GHJV94] organisiert sind. Bei diesem Verhaltensmuster gibt es eine Kette von Aktoren, die unterschiedliche Aufgabenbereiche besitzen. Bei der Abarbeitung wird die Aufgabe einfach dem ersten Glied der Kette zugewiesen. Jeder Bestandteil der Kette prüft dann den Zuständigkeitsbereich. Liegt er in der Verantwortung des aktuellen Elements, so wird die Aufgabe erledigt, ansonsten wird die Aufgabe an das nächste Glied der Kette weitergereicht. Dieses Design erleichtert die Implementierung eines Präprozessors, da dieser in der Kette lediglich vor den für die Codeerzeugung zuständigen Aktoren eingeklinkt werden muss. Abbildung 6.18 zeigt einen Ausschnitt aus dem Klassendiagramm der Fujaba Codegenerierung. Alle Glieder der Kette erben von **CodeWriter**, so auch der Präprozessor **MODPLPreprocessorCodeWriter**. Während der Initialisierung des Plugins, das für die Konfiguration des Domänenmodells zuständig ist, wird der Präprozessor an die entsprechende Stelle in der Verantwortlichkeitskette eingefügt. Die Methode, die am ersten Glied der Kette aufgerufen wird ist **generateCode**, die erwartete Rückgabe ist eine Zeichenkette, die das entsprechende Codefragment enthält. Die übergebene „Aufgabe“ ist ein **Token**, ein Element des abstrakten Syntaxgraphen. Ist das aktuelle Glied der Kette nun nicht für die Bearbeitung der Aufgabe zuständig, so wird **null** zurückgeliefert. Fällt das übergebene Objekt jedoch in den Zuständigkeitsbereich des CodeWriters, so wird das entsprechende Quellcodefragment erzeugt und zurückgeliefert. Für das Verhalten des Präprozessors bedeutet dies, dass für jedes Element des Syntaxgraphen zunächst geprüft wird, ob es sich in der aktuellen Konfiguration befindet. Ist dies der Fall, so wird das Token an das nächste Element der Kette weitergereicht (durch Rückgabe von **null**). Ist es allerdings nicht in der Konfiguration enthalten, so fällt es in den Zuständigkeitsbereich des Präprozessors. Es soll dafür kein Quelltext generiert werden, daher wird eine leere Zeichenkette zurückgegeben. Abbildung 6.19 zeigt die Implementierung der Methode **generateCode** des Präprozessors.

Nach Ermittlung des aktuellen Context Objekts, wird die Klasse **MODPLUtility**, die eine direkte Subklasse von **Utility** aus der Fujaba Codegenerierung ist, darin abgelegt. Sie wird benötigt, um in den Templates prüfen zu können, ob ein Element Teil der aktuellen Konfiguration ist. Dies ist beispielsweise nötig, um bei der Deklaration von Java Klassen die Ausgabe der Schlüsselwörter **extends** bzw. **implements** zu unterdrücken, falls die zugehörigen Vererbungsbeziehungen im Fujaba Modell mit Feature Tags ausgezeichnet sind. Aber auch, um zu verhindern, dass ausgezeichnete Typen als Attribute, Methodenparameter bzw. Rückgabewerte verwendet werden. In diesem Fall wird dann jeweils das Basisobjekt **java.lang.Object** verwendet. Die entsprechenden Templates der Fujaba Codegenerierung wurden dahingehend angepasst. Ist das übergebene Element des abstrakten Syntaxgraphen das Äquivalent für das Projekt, so werden zunächst die Propagationsregeln angewandt. Anschließend wird dann bei jedem weiteren Aufruf der Methode für das entsprechende Fujaba Modellelement geprüft, ob es sich in der aktuellen Konfiguration befindet. Einen Sonderfall stellt hier die Klasse **UMLAttr** dar, für die geprüft werden muss, ob sich außerdem auch der entsprechende Typ in der Konfiguration befindet. Ist dies der Fall, so wird das entsprechende Token an die nächste Instanz in der Codegenerator Kette weitergereicht. Falls nicht, so wird ein leerer String zurückgegeben und somit eine Codegenerierung für diese Elemente verhindert.

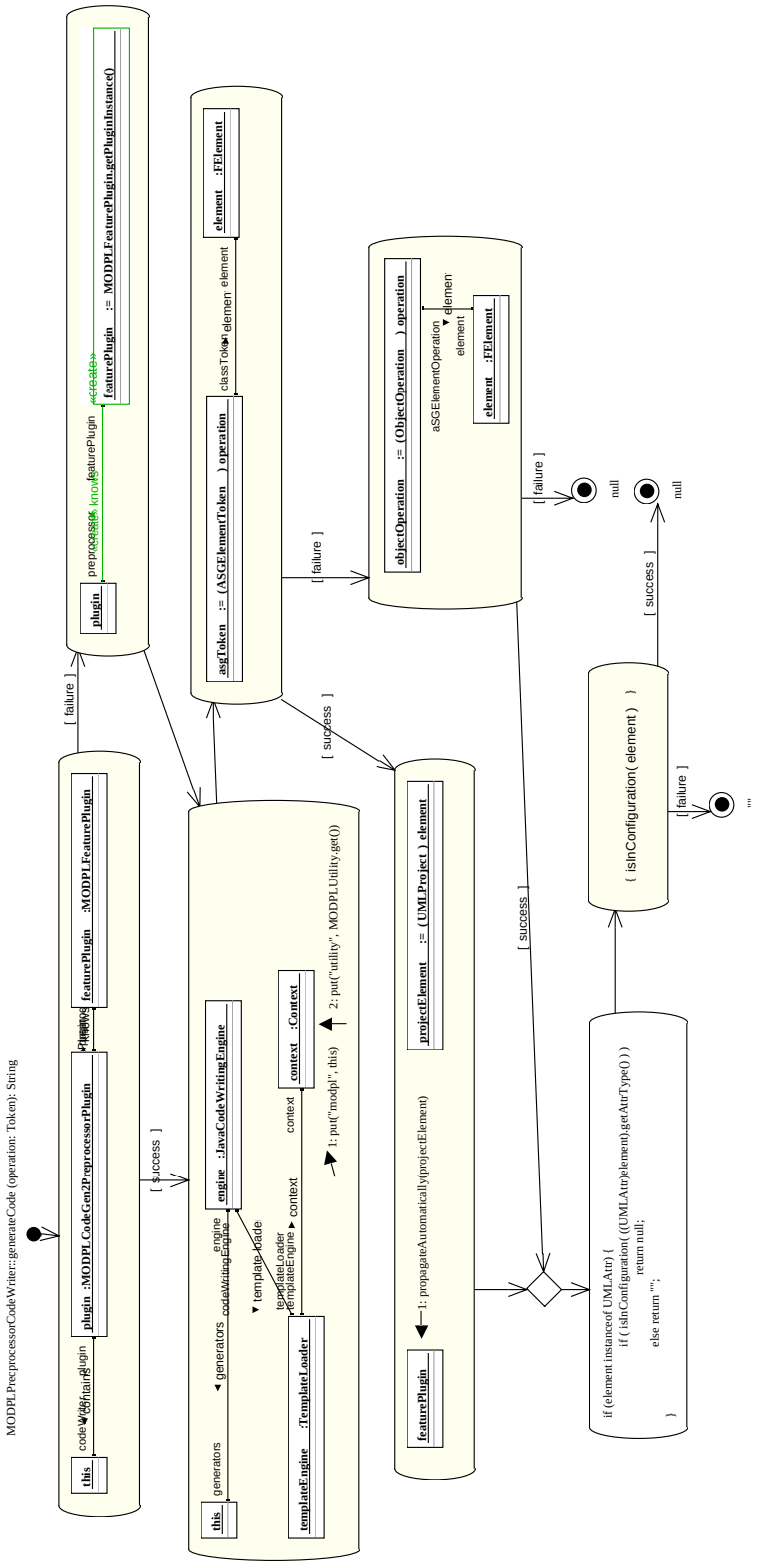

Abbildung 6.19: Implementierung der Methode **generateCode**.

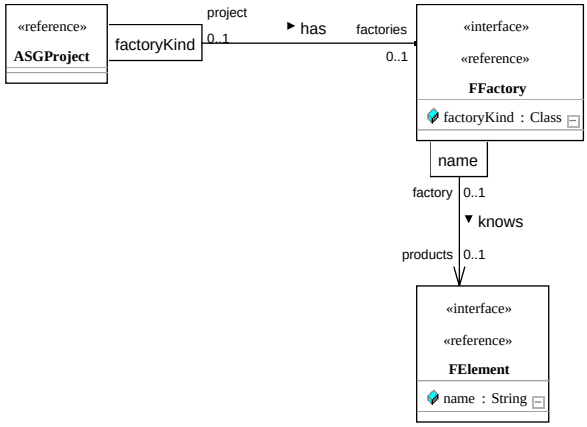

Abbildung 6.20: Ausschnitt aus dem Fujaba Metamodell.

Erzeugen eines konfigurierten Modells

Die dritte vom Werkzeug unterstützte Möglichkeit der Darstellung einer Konfiguration ist das Erzeugen eines neuen Fujaba Modells, das nur die in der Konfiguration enthaltenen Modellelemente enthält. Die dafür benötigten Modelltransformationsregeln sind in der Klasse **MODPLModelGenerator** implementiert. Letztlich muss dieser Generator, obwohl er eine ähnliche Aufgabe erfüllt wie das Konfigurieren von Quelltext, völlig neu realisiert werden. Die Templates zur Codegenerierung können dafür nicht verwendet werden, da das neue Modell über die Fujaba API als Speicherobjekt aufgebaut werden muss.

Zunächst wird programmatisch ein neues Fujaba Modell mit neuem Namen erzeugt. Da der Projektname einen eindeutigen Bezeichner darstellt, muss sich dieser zwingend vom Namen des Domänenmodells, das konfiguriert werden soll, unterscheiden. Auch bei diesem Verfahren ist es nötig, über alle Elemente des Fujaba Modells zu iterieren, und für jedes Modellelement zu prüfen, ob es Bestandteil der aktuellen Konfiguration ist, oder nicht. In Fujaba werden sämtliche Modellelemente nach dem *Factory*-Muster [FF04] erzeugt, wobei die einzelnen Fabriken Referenzen auf die erzeugten Elemente halten. Über das jeweilige Projekt kann auf die entsprechenden Fabriken zugegriffen werden. Abbildung 6.20 zeigt den entsprechenden Ausschnitt aus dem Fujaba Metamodell. Als sinnvolle Vorgehensweise bietet es sich an, zuerst über alle Klassen zu iterieren und so das grobe Gerüst des neuen Modells aus den in der Konfiguration enthaltenen Klassen und deren umschließenden Paketen zu bilden. Sämtliche Attribute, Methoden, Vererbungsbeziehungen und Assoziationen folgen dann in einem zweiten Schritt. Dies ist notwendig, da eine Klasse beispielsweise Attributtypen oder Oberklassen verwenden kann, die zu diesem Zeitpunkt im Zielmodell noch gar nicht existieren. Die Reihenfolge der Erzeugung der Modellelemente im Zielmodell ist wie folgt:

1. Klassen mit den zugehörigen Stereotypen, Paketen und Klassendiagrammen
2. Vererbungsbeziehungen

3. Attribute
4. Assoziationen
5. Methoden mit den zugehörigen Storydiagrammen
6. Hinzufügen von referenzierten Klassen zu Klassendiagrammen

Listing 6.11 zeigt den Quelltext der Methode, die Klassen mit den zugehörigen Stereotypen und Klassendiagrammen anlegt. Die Methoden zum Erzeugen von Attributen, Assoziationen, Methoden und Vererbungsbeziehungen sind analog dazu aufgebaut. In diesem Fall wurde einer manuellen Implementierung der Vorzug gegenüber der modellgetriebenen Entwicklung mit Fujaba gegeben, da diese zeitsparender war. Für die Verwendung von Fujaba müssten zunächst etliche notwendige Elemente aus dem Fujaba Metamodell als Referenzen importiert werden. Weitaus problematischer ist allerdings die Tatsache, dass die Fabriken zur Erzeugung von Fujaba Metamodellelementen *Generics* verwenden und dies nicht von Fujaba unterstützt wird. Es wären in jedem Fall Statement-Activities zusätzlich zu den Storymustern notwendig gewesen, was gegenüber einer rein modellgetriebenen Entwicklung schon einen Nachteil darstellt.

Komplexer hingegen ist das Erzeugen der zu den Methoden gehörenden Storydiagrammen. Ähnlich wie ein Aktivitätsdiagramm der UML besitzt ein Storydiagramm eine Startaktivität sowie beliebig viele Stopaktivitäten. Dazwischen befinden sich Aktivitäten, die mittels Transitionen verbunden sind. Diese Transitionen dienen der Steuerung des Kontrollflusses. Eine Transition hat immer genau eine Quelleaktivität und eine Zielaktivität. Eine Aktivität kann maximal zwei ausgehende Transitionen besitzen. Verzweigungen können über Diamanten erreicht werden. Jedes Storydiagramm hat Referenzen auf alle im Diagramm vorkommenden Aktivitäten und Transitionen. Aufgrund des von Fujaba verwendeten Persistenzmechanismus (siehe unten) ist die Reihenfolge der Elemente jedoch nicht analog zum Kontrollfluss, sondern sie ergibt sich aus der zeitlichen Abfolge der Erzeugung der Elemente im Editor. Daher bietet sich als geeignete Durchlaufstrategie für die Storydiagramme eine Iteration über alle Transitionen an. Da eine Transition immer genau zwei Aktivitäten miteinander verbindet, kann so sukzessive das Storydiagramm im Zielmodell aufgebaut werden. Neben Quell- und Zielaktivitäten besitzen Transitionen noch sogenannte *TransitionGuards* mit deren Hilfe eine Kontrollflusssteuerung erfolgen kann. Dies können sowohl boolsche Ausdrücke sein, als auch Bedingungen für Iterationen (wie etwa für jeden Iterationsschritt) und Ausnahmebehandlungen. Damit im Falle von Verzweigungen Aktivitäten und Transitionen nicht doppelt erzeugt werden, werden die bereits erzeugten Objekte mit ihren Pendants aus dem Quellmodell in einer Hashmap abgelegt. Bei den Aktivitäten unterscheidet Fujaba fünf verschiedene Typen:

Start Die Klasse **UMLStartAcitivity** repräsentiert den Startpunkt für den Kontrollfluss durch das Storydiagramm.

Stop Die Klasse **UMLStopAcitivity** hingegen repräsentiert den Endpunkt des Kontrollflusses.

```
1  private void createClasses() {
2    FFactory<UMLClass>   sClassFactory = source.getFromFactories(
         UMLClass.class);
3    FFactory<UMLClass>   tClassFactory = target.getFromFactories(
         UMLClass.class);
4    FFactory<UMLClassDiagram> tClassDiagFactory = target.
         getFromFactories(UMLClassDiagram.class);
5    FFactory<UMLStereotype> tStereotypeFactory = target.
         getFromFactories(UMLStereotype.class);
6
7    Iterator<UMLClass> it = sClassFactory.iteratorOfProducts();
8    while ( it.hasNext() ) {
9      UMLClass currentElem = it.next();
10     if (!writer.isInConfiguration(currentElem))
11       continue;
12     UMLPackage sourcePack = currentElem.getDeclaredInPackage();
13     createPackagePath(sourcePack);
14     UMLPackage targetPack = target.getRootPackage().findPackage(
           sourcePack.getFullPackageName());
15
16     UMLClass targetClass = tClassFactory.create();
17     targetClass.setDeclaredInPackage(targetPack);
18     targetClass.setName(currentElem.getName());
19     targetClass.setAbstract(currentElem.isAbstract());
20
21     Iterator sIter = currentElem.iteratorOfStereotypes();
22     while (sIter.hasNext()) {
23       String name = ((UMLStereotype)sIter.next()).getName();
24       UMLStereotype targetStereo = tStereotypeFactory.getFromProducts
             (name);
25       if (targetStereo == null) {
26         targetStereo = tStereotypeFactory.create();
27         targetStereo.setName(name);
28       }
29       targetClass.addToStereotypes(targetStereo);
30     }
31     UMLClassDiagram sDiag = (UMLClassDiagram) currentElem.
           getFirstFromDiagrams();
32     UMLClassDiagram tDiag = tClassDiagFactory.getFromProducts(sDiag.
           getName());
33     if (tDiag == null) {
34       tDiag = tClassDiagFactory.create();
35       tDiag.setName(sDiag.getName());
36     }
37     targetClass.addToDiagrams(tDiag);
38     if (currentElem.getComment() != null)
39       addComment(targetClass, tDiag, currentElem.getComment().getText
             ());
40   }
41 }
```

Listing 6.11: Quelltext der Methode zum Anlegen der Klassen für das konfigurierte Modell.

NOP Die Klasse **UMLNopAcitivity** repräsentiert den Diamanten, mit dessen Hilfe der Kontrollfluss verzweigt und auch wieder zusammengeführt werden kann.

Story Die Klasse **UMLStoryActivity** enthält ein **UMLStoryPattern** was seinerseits beliebige Instanzen von Modellelementen beinhalten kann. Veränderungen des Objektgraphen werden mit Graphtransformationsregeln ausgedrückt. Außerdem erlaubt es noch die Angabe von Bedingungen und den Aufruf von Methoden an den Objekten durch *Kollaborationsaufrufe*.

Statement Die Klasse **UMLStatementActivity** repräsentiert eine Aktivität, die beliebigen handgeschriebenen Java Quelltext aufnehmen kann. Dieser wird dann 1:1 in den generierten Code an der entsprechenden Stelle des Kontrollflusses eingefügt.

Ist die aktuelle Aktivität ein Storypattern, so müssen alle darin enthaltenen Bestandteile geprüft und bei Bedarf kopiert werden. Elemente von Storypattern sind: Objekte (**UMLObject**) als Instanzen von Klassen des Klassendiagramms, Verbindungen (**UMLLink**) zwischen diesen Objekten als Instanzen von Assoziationen, Bedingungen (**UMLConstraint**) und Kollaborationsaufrufe (**UMLCollabStat**). Zusätzlich können Objekte noch Attribute (**UMLAttrExprPair**) mit Wertzuweisungen bzw. vorgegebenen Werten als Bedingung besitzen.

Optional können auch noch alle Kommentare kopiert werden. Feature Tags hingegen werden für das konfigurierte Modell nicht mehr erzeugt, da es sich beim Ergebnis der Modelltransformation ja schon um das Modell eines konkreten Produkts handelt und somit eine weitere Konfigurierung nicht mehr notwendig ist.

Der einzige Nachteil des verwendeten Ansatzes ist, dass hiermit keine Layoutinformationen der vorhandenen Diagramme kopiert werden können. Die Trennung von Darstellung und Inhalt erfolgt in Fujaba analog dem *Model-View-Controller*-Muster [FF04], bei dem die zur Darstellung verantwortlichen Elemente die zugrunde liegenden Modellelemente referenzieren, aber nicht umgekehrt. Somit ist es nicht möglich von der Ebene des Fujaba Metamodells auf Darstellungselemente zuzugreifen. Eine mögliche Lösung wäre die Verwendung von *CoObRA* [SZN04, Sch07] zur Replikation des gesamten Modells und anschließend sukzessives Löschen aller nicht benötigten Modellelemente. Allerdings birgt dieser Ansatz eine Reihe von schwerwiegenden Nachteilen, die hauptsächlich durch Aufbau und Arbeitsweise von CoObRA begründet sind. Elementare Editieroperationen in Fujaba werden hier ähnlich einer Art Logdatei der Reihe nach gespeichert. Dies bedeutet, dass bei jedem Öffnen eines Fujaba Modells sämtliche Änderungen aus der CoObRA Datei wieder ausgeführt werden, um das Modell aufzubauen. Neben dem Nachteil der Größe der Logdatei besteht außerdem noch das Problem des eindeutigen Namens von Projekten in Fujaba. Wird ein Projekt mittels CoObRA repliziert, so muss zwingend der Name geändert werden. Hierfür müssen sämtliche CoObRA Änderungsoperationen überprüft werden. Überall, wo eine Änderung des Projektnamens stattfindet, muss der neue Projektname eingetragen werden. Eine über ein derartiges Verfahren erzeugte Datei eines konfigurierten Modells ist ebenfalls auch größer als die des Quellmodells, da ja erst

das komplette Modell aufgebaut und dann nicht benötigte Bestandteile wieder daraus entfernt werden und all diese elementaren Fujaba Operationen in der Reihenfolge ihrer Anwendung in der Datei gespeichert sind.

6.6.3 Integration in Fujaba

Die Integration der in den beiden vorigen Abschnitten vorgestellten Werkzeuge in Fujaba erfolgte über die Plugin Infrastruktur des Fujaba Rahmenwerks, sowie über die Plugin Infrastruktur von Eclipse. Um dem Benutzer einen komfortablen Zugriff auf die implementierten Methoden zu ermöglichen, wurden Aktionen definiert, die an der Benutzerschnittstelle aufgerufen werden können. So wurden etwa Dialoge zur Auswahl der zu importierenden XML Dateien mit dem Featuremodell bzw. der Konfiguration erstellt und ebenso ein neuer Dialog zur Vergabe von Feature Tags. Der Fujaba Dialog zur Vergabe einer Annotation birgt zu viele potentielle Fehlerquellen, da er die Eingabe von beliebigen Zeichenketten erlaubt. Um aber eine Zuordnung von Elementen des Feature Modells mit Elementen des Domänenmodells zu gewährleisten, ist es zwingend notwendig, dass das Schlüssel-Wert-Paar, das mit dem Feature Tag assoziiert ist als Schlüssel die Zeichenkette **id** und als Wert den eindeutigen Bezeichner des Features aus dem Feature Modell verwendet. Daher beschränkt sich der Dialog zur Vergabe von Feature Tags auf eine reine Auswahl aller Features des aktuell geladenen Modells. Somit wird ausgeschlossen, dass eine inkonsistente Zuordnung etwa durch Tippfehler entsteht.

Um dem Benutzer die Hierarchie des Featuremodells bzw. der Konfiguration anzuzeigen, wurde jeweils eine neue Ansicht in Eclipse definiert, die die entsprechende Objektstruktur als Baum anzeigt. Durch Selektieren von Baumeinträgen werden die entsprechenden Feature Tags an den Modellelementen als Hilfsmittel für den Modellierer farblich hervorgehoben, wie Abbildung 6.21 zeigt. Da die Vergabe von Feature Tags mit beliebiger Granularität erfolgen kann, können somit unter Umständen sehr viele Modellelemente für die Realisierung ein und desselben Features verantwortlich sein. Die farbliche Hervorhebung erleichtert dem Modellierer somit den Überblick. Die Darstellung von inkonsistenten Feature Annotationen erfolgt ebenfalls über farbliche Hervorhebung sowohl in der Baumansicht, als auch im Fujaba Editor. Auch durch Anwendung der Propagationsregeln automatisch hinzugefügte Feature Tags können bei Bedarf visualisiert werden. Neben den Aktionen, die für die Zuordnung von Features und Modellelementen zuständig sind, wurden auch die entsprechenden Aktionen zum Aufruf der für die Konfigurierung zuständigen Methoden implementiert.

6.6.4 Integration in den Paketdiagrammeditor

Um Elemente des Paketdiagramms mit Feature Annotationen auszeichnen zu können, musste das Paketdiagramm Metamodell erweitert werden. EMF sieht als Erweiterungsmechanismus für Modelle die Möglichkeit vor, Modellelemente mit Annotationen, sog. **EAnnotation**s auszuzeichnen. Da die einzelnen Bestandteile des Paketdiagramms aber Instanzen von **EClass** Objekten sind, ist es zur Laufzeit leider nicht möglich diese mit Hilfe des generierten Editors für die Baumsicht oder dem graphischen Editor (der mit GMF

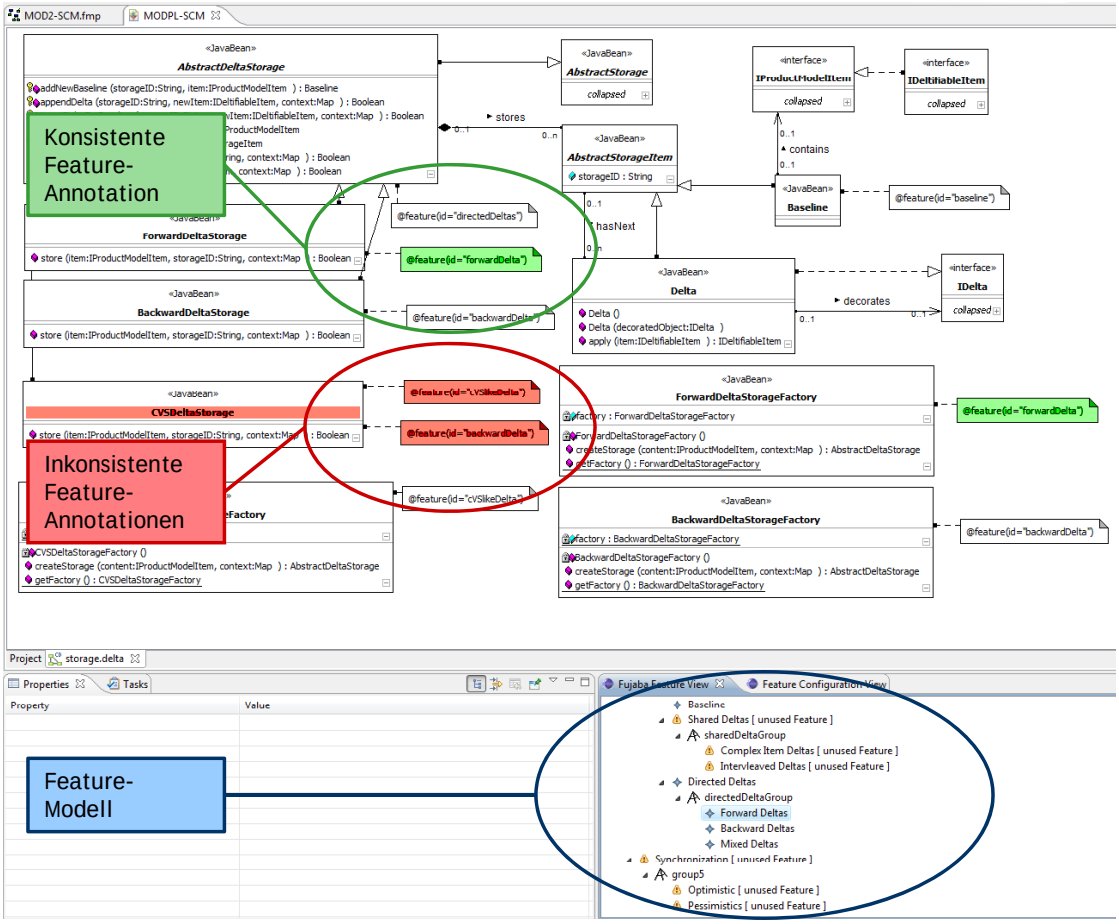

Abbildung 6.21: MODPLFeaturePlugin als Erweiterung des Fujaba Editors.

erstellt wurde) mit **EAnnotation**s zu versehen. Man könnte zwar das Paketdiagramm Modell dahingehend ändern, dass das Basiselement des Paketdiagramms, **Element**, von dem entsprechenden Ecore Interface **EModelElement** erbt, dies hätte dann aber zur Folge, dass sämtliche in diesem Interface definierten Methoden ebenfalls implementiert werden müssten. Die einfachste Art der Erweiterung des Paketdiagramm Modells war daher die Einführung einer Klasse **FeatureTag**. Jedes Element des Paketdiagramms kann nun ein oder mehrere dieser **FeatureTag**s enthalten. Auch hier gilt: sind mehrere **FeatureTag**s einem Modellelement zugeordnet, so sind diese konjunktiv verknüpft. Abbildung 6.22 zeigt einen Ausschnitt des Klassendiagramms mit der entsprechenden Erweiterung.

Auch hier erfolgt die Zuordnung zwischen Elementen des Feature Modells und den Elementen des Paketdiagramms durch Verwendung des eindeutigen Bezeichners des Features im Feld **id** der Klasse **FeatureTag**. Diese zusätzlichen Informationen werden ebenfalls in beiden Richtungen zwischen Paketdiagramm und Fujaba Modell propagiert und stehen somit beim *Forward-*, *Reverse-* und *inkrementellen Roundtrip Engineering* zur Verfügung.

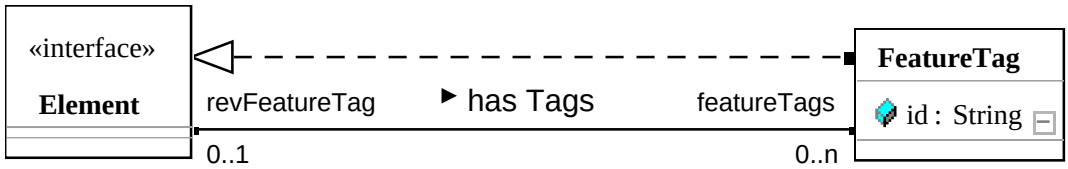

Abbildung 6.22: Erweiterung des Paketdiagramm Modells mit **FeatureTag**s.

6.7 Arbeiten mit dem Werkzeug

Analog zu dem Beispiel aus Abbildung 6.3 werden nun exemplarisch die Schritte aufgezeigt, um einerseits die Elemente des Feature Modells auf das Domänenmodell des MOD2-SCM Projekts abzubilden, als auch die unterschiedlichen Arten der Konfigurierung.

6.7.1 Hinzufügen von Feature Tags

Bevor Feature Tags mit Elementen des Domänenmodells verknüpft werden können, muss ein entsprechendes Feature Modell geladen werden. Dies geschieht durch Aufruf der entsprechenden Aktionen an der Fujaba Benutzerschnittstelle. Sobald das Feature Modell geladen ist, wird die entsprechende Hierarchie in der dazugehörigen Baumansicht angezeigt. Abbildung 6.21 zeigt die Baumansicht unten rechts im Bild. Um eine Zuordnung von Features mit Modellelementen zu erstellen, kann nun entweder die entsprechende Funktion im Kontextmenü des Fujaba Editors aufgerufen werden, es besteht aber auch die Möglichkeit, mehrere Modellelemente im Fujaba Editor zu selektieren und dann in der Baumansicht das Kontextmenü des zu vergebenden Features aufzurufen und die entsprechende Aktion auszuführen. Abbildung 6.23 zeigt dies. Ebenso hat der Benutzer nun die Option, sich mögliche Features für Modellelemente durch Analyse des Vererbungsbaums und der Featurehierarchie vorschlagen zu lassen.

6.7.2 Kontrolle der Konsistenz

Der Benutzer hat nun verschiedene Möglichkeiten zu prüfen, ob die vorgenommene Abbildung von Features auf Elemente des Domänenmodells zu inkonsistenten Konfigurationen führt. Inkonsistente Konfigurationen können beispielsweise das Resultat der Vergabe von sich gegenseitig ausschließenden Features an ein und dasselbe Modellelement sein. Durch die konjunktive Verknüpfung von Feature Tags bei der Konfiguration des Modells entsteht so die Situation, dass in keiner Konfiguration alle mit dem Modellelement verknüpften Features enthalten sein können. Für ein solches Modellelement würde also niemals Quelltext erzeugt werden. Abbildung 6.21 zeigt eine inkonsistente Feature Annotation. Dem Element **CVSDeltaStorage** wurde neben dem Feature für die gemischten Deltas (mit dem Bezeichner *cVSlikeDelta*) das Feature für die Rückwärtsdeltas (mit dem Bezeichner *backwardDelta*) zugeordnet. Da sich diese beiden Features laut dem Feature

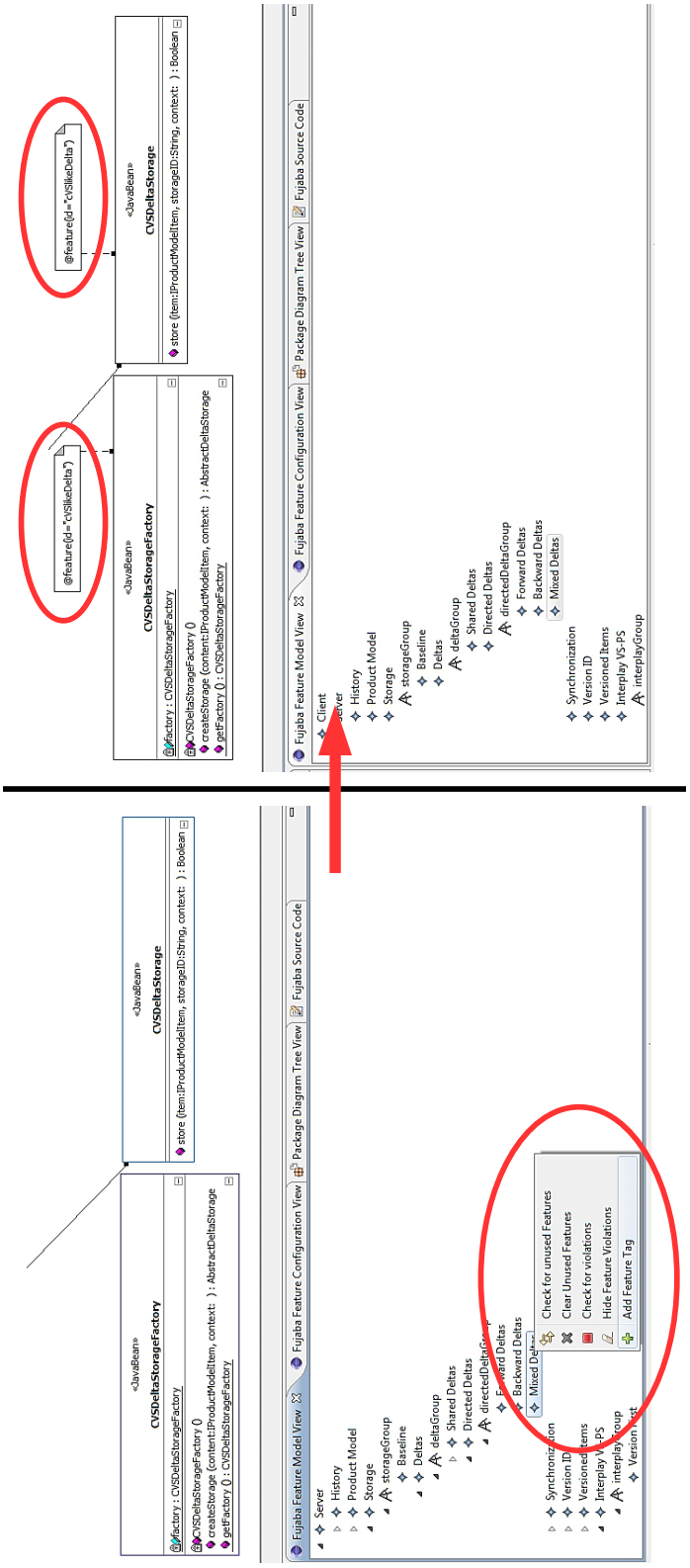

Abbildung 6.23: Hinzufügen eines Features zu Modellelementen über die Baumansicht.

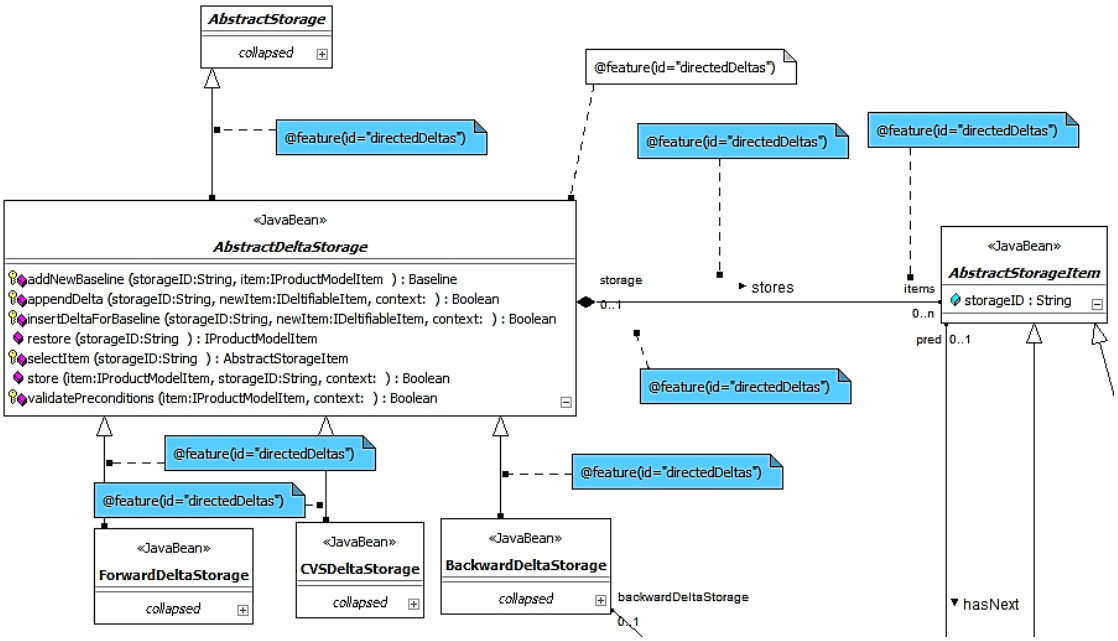

Abbildung 6.24: Anwendung und Visualisierung von Propagationsregeln.

Modell aber gegenseitig ausschließen, entstehen inkonsistente Konfigurationen. Ebenso werden Inkonsistenzen in der Hierarchie der Elemente entdeckt.

6.7.3 Anwendung und Visualisierung der Propagationsregeln

Der Benutzer hat jederzeit die Möglichkeit, die Propagationsregeln anzuwenden und sich das Ergebnis anzeigen zu lassen. Abbildung 6.24 zeigt einen Ausschnitt des MOD2-SCM Klassendiagramms nach Anwendung und Visualisierung der Propagationsregeln. Alle automatisch hinzugefügten Feature Tags sind in blauer Farbe dargestellt.

Ursprünglich wurde vom Modellierer nur die Klasse **AbstractDeltaStorage** mit dem Feature *Directed Deltas* (mit dem eindeutigen Bezeichner *directedDeltas*) ausgezeichnet. Dieser Feature Tag ist in der Abbildung mit weißer Hintergrundfarbe dargestellt. Nach der Anwendung der Propagationsregeln aus Abschnitt 6.3.4 wurden alle in blauer Farbe dargestellten Feature Tags automatisch hinzugefügt. Alle ein- und ausgehenden Vererbungsbeziehungen sind nun ausgezeichnet, sowie auch alle Attribute, Methoden (in der Abbildung zu Gunsten besserer Lesbarkeit ausgeblendet) und Rollenenden. Aufgrund der Regeln 4 und 5 sind bei ausgezeichneten Rollenenden dann auch die zugehörige Assoziation, sowie das gegenüberliegende Rollenende auszuzeichnen, wie in Abbildung 6.24 auch zu erkennen ist.

Auf diese Weise kann zwischen primären Annotationen (vom Benutzer angebracht) und sekundären Annotationen (durch Regelanwendung automatisch hinzugefügt) unterschieden werden. Mit diesem Mechanismus ist es dem Benutzer etwa möglich, ein Paket mit einem Feature Tag zu versehen. Alle in dem Paket enthaltenen Kindelemente sind

dann auf Grund von Regel 2 auch mit dem gleichen Feature ausgezeichnet. Auf Benutzerwunsch können die sekundären Annotationen ein- und wieder ausgeblendet werden. Der Benutzer hat somit bei der Vergabe der Feature Annotationen freie Wahl der Granularität. Die Annotationen können grob-granular auf Paketebene erfolgen, aber auch auf fein-granularer Ebene im Falle von Attributen oder Methoden.

6.7.4 Visualisierung einer Konfiguration

Wurden die Features auf die Elemente des Domänenmodells abgebildet und eine gültige Konfiguration geladen, so kann diese im Fujaba Editor visualisiert werden. Hierbei werden alle Modellelemente, die nicht zur gewünschten Konfiguration gehören „ausgeblendet“, wie in Abbildung 6.25 zu erkennen ist. Im dargestellten Beispiel wurde eine Konfiguration für ein System geladen, das ähnlich wie CVS arbeitet, also mit gemischten Deltas. Elemente, die die Funktionalität von Vorwärts- und Rückwärtsdeltas implementieren, sind somit nicht Teil der Konfiguration.

6.7.5 Erzeugen von Quellcode

Um konfigurierten Quelltext zu erzeugen, reicht es aus eine gültige Konfiguration zu laden und den Codegenerierungsmechanismus von Fujaba zu starten. Das Präprozessor Plugin erkennt die geladene Konfiguration und wendet automatisch die in Abschnitt 6.6.2 beschriebenen Funktionen an, um für nicht benötigte Modellelemente keinen Quelltext zu erzeugen.

6.7.6 Erzeugen eines konfigurierten Modells

Um ein konfiguriertes Modell zu erzeugen muss ebenfalls eine gültige Konfiguration geladen werden. Anschließend können bei Bedarf der Speicherort und Dateiname der neuen Fujaba Datei, sowie der Projektname noch angepasst werden. Die restliche Konfiguration des Modells erfolgt dann automatisch. Die Abbildungen 6.25 und 6.26 zeigen Ausschnitte aus dem kompletten Domänenmodell und dem neuen Modell, das anhand einer Konfiguration daraus erzeugt wurde. Die Anordnung der Elemente des Klassendiagramms im konfigurierten Modell erfolgte durch die automatische Layout-Funktion von Fujaba, da ja wie in Abschnitt 6.6.2 beschrieben, bei der Erzeugung des konfigurierten Modells keine Layoutinformationen mit übertragen werden.

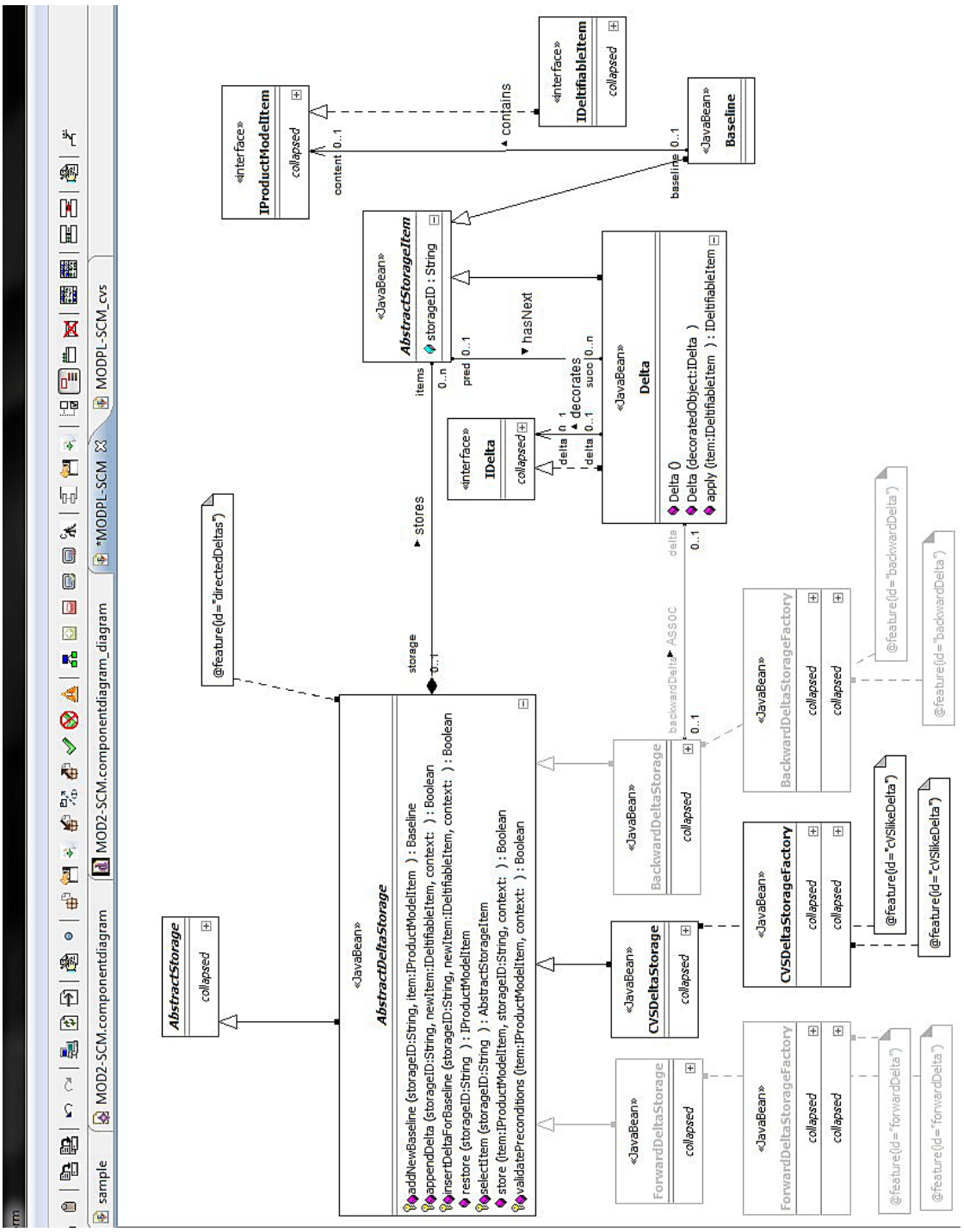

Abbildung 6.25: Visualisierung einer Konfiguration im Fujaba Editor.

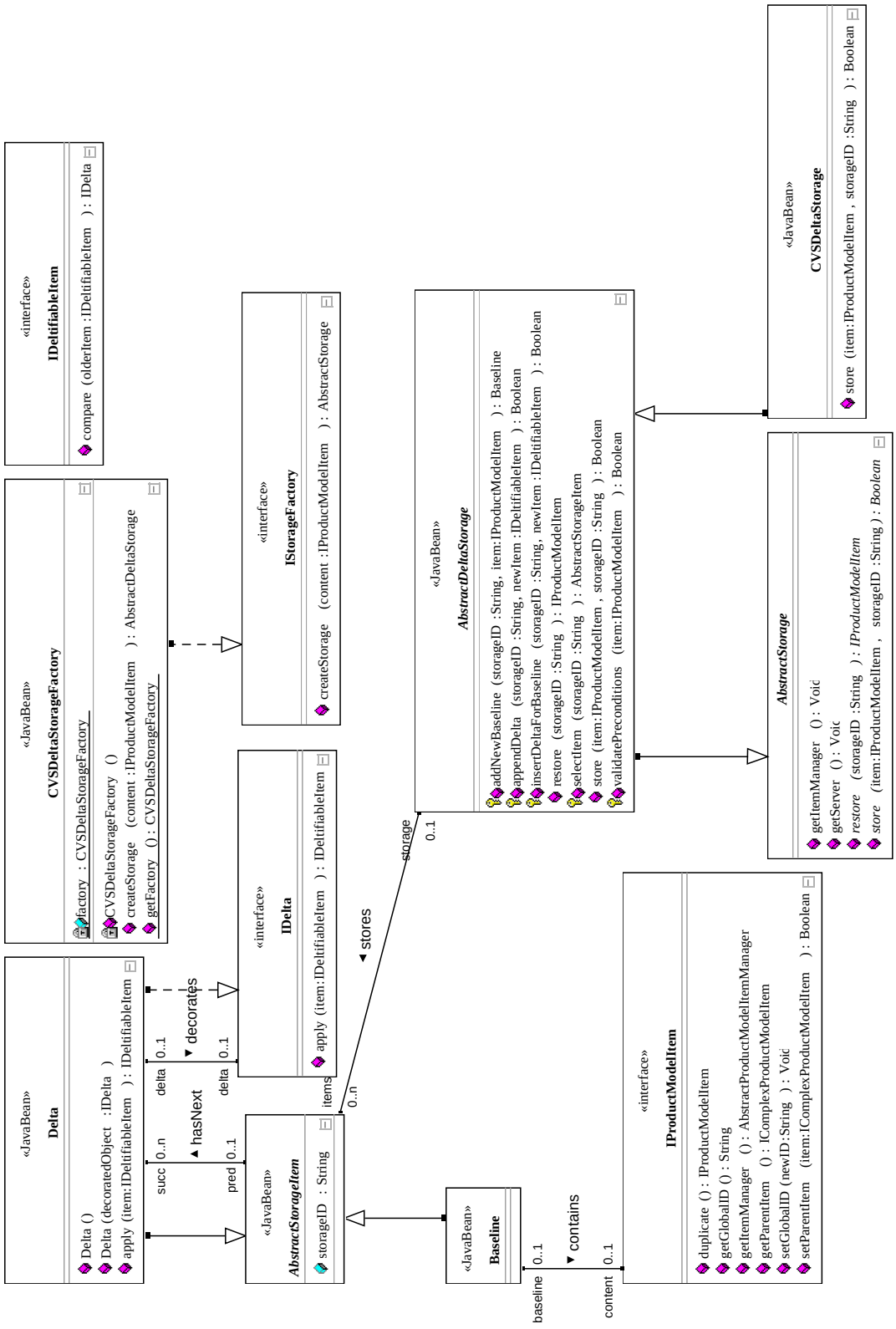

Abbildung 6.26: Ausschnitt aus dem anhand der Konfiguration erzeuten Modell.

6.8 Verwandte Ansätze

6.8.1 Ansatz von Czarnecki et. al

In [CA05] stellen Czarnecki und Antkiewicz einen theoretischen, auf Modellvorlagen basierten Ansatz vor, der es erlaubt, Featuremodelle auf prägnante Repräsentationen von Variabilität in verschiedenen Arten von anderen Modellen abzubilden. Eine prototypische Implementierung des Ansatzes für UML 2.0 Aktivitäts- und Klassenmodelle wird beschrieben. Die Einhaltung von Bedingungen zur Wohlgeformtheit mit OCL wird in [CP06] erläutert. Im Gegensatz zu Ansätzen zur Darstellung von Variabilität, bei denen einzelne Fragmente des Modells, die zu unterschiedlichen Features gehören, zusammengesetzt werden, präsentieren die Autoren einen Ansatz, der eine Überlagerung aller Varianten im Modell darstellt. Die Modellelemente sind mit den zugehörigen Features mittels Annotationen verbunden. Der Ansatz ist sehr allgemein, da er für alle Modelle anwendbar ist, deren Metamodell in MOF [OMG06a] oder einem vergleichbaren Formalismus spezifiziert ist und er kann leicht in bestehende Editoren integriert werden. Bei der prototypischen Umsetzung wird für jedes Feature aus dem Featuremodell ein eigener Stereotyp erzeugt, der dann den Elementen der Modellvorlage zugeordnet werden kann. Die Annotationen an den Modellelementen erfolgen durch sogenannte *presence conditions* (PCs), also Bedingungen, die erfüllt sein müssen, damit das Modellelement auch in der Konfiguration enthalten ist und *meta-expressions* (MEs). Meta-expressions werden dafür benutzt, um Attribute von Modellelementen, wie etwa den Namen eines Elements oder den Rückgabetyp einer Operation zu berechnen. Durch Erstellen einer Konfiguration kann dann eine konkrete Instanz der Modellvorlage mittels Modell-zu-Modell Transformation erzeugt werden. Die Angabe von Ein- und Ausgabemodell erfolgt hierbei in der Zielnotation (also z. B. in UML). Alle Modellelemente, deren PCs nicht erfüllt sind werden bei der Transformation entfernt. Die Interpretation der PCs erfolgt lokal in Abhängigkeit zu den Enthaltenseinsbeziehungen, die im Metamodell angegeben sind. Dies bedeutet insbesondere, dass Elemente, die aufgrund ihrer PC eigentlich in der konkreten Modellinstanz enthalten wären, entfernt werden, falls ihr übergeordnetes Element entfernt wird.

Im Gegensatz zu dem in dieser Arbeit vorgestellten Ansatz verwenden Czarnecki und Antkiewicz anstelle der Propagationsregeln sogenannte implizite PCs. Diese impliziten PCs werden auf Basis von PCs an anderen Modellelementen und der Syntax und Semantik der Zielnotation ermittelt. Im Falle einer Assoziation in UML wäre eine implizite PC so z. B. die Konjunktion der PCs der beiden Assoziationsenden. Analog zu den Propagationsregeln reduzieren diese impliziten PCs den Annotationsaufwand für den Benutzer. Für UML Klassendiagramme sind implizite PCs für Generalisierungen, Assoziationen und Abhängigkeiten angegeben. Allerdings umfassen diese impliziten PCs nicht alle Bedingungen, die in Abschnitt 6.3.4 formuliert wurden. Für Elemente, bei denen die Zuordnung der abhängigen Elemente mittels Assoziationen mit der Kardinalität „1“ erfolgt, werden keine Regeln angegeben. So gibt es in der Arbeit von Czarnecki et al. keine implizite PC, die Regel 3 entspricht. Außerdem werden noch implizite PCs

für Aktivitätsdiagramme genannt. Die in Abschnitt 6.3.4 vorgestellten Propagationsregeln behandeln außerdem noch unidirektionale Assoziationen und die Propagation auf Elemente von Storydiagrammen. Außer einer nicht mehr verfügbaren prototypischen Implementierung der Konzepte für Modelle, die mit Rational Software Modeler erstellt wurden, gibt es von den Autoren kein weiteres Werkzeug. Der vorgestellte Ansatz ist nur auf die Erzeugung von Modellinstanzen auf Basis einer Modellvorlage, die alle im Feature Modell vorkommenden Varianten abdeckt, ausgelegt. Der Ansatz von Czarnecki und Antkiewicz beinhaltet keine Generierung von konfiguriertem Quelltext oder eine Visualisierung durch Ausblenden im Editor.

6.8.2 Ecore.fmp

Eine Weiterentwicklung des zur Feature Modellierung verwendeten Werkzeugs *Feature-Plugin* (siehe Abschnitt 2.3.1) stellt *Ecore.fmp* [SA08] dar. Es bietet die Möglichkeit, Ecore Modelle als Featuremodelle zu betrachten und auch zu entwickeln. So erlaubt es auf einfache Art und Weise Ecore Metamodelle und deren entsprechende Instanzen zu bearbeiten. Ein Klassenmodell wird als Featuremodell und ein Objektmodell als Feature Konfiguration interpretiert. Die Darstellung von Variabilität beschränkt sich somit auf Instanzenbildung im Klassenmodell durch Verwendung von Kardinalitäten [CHE05, KC05] im Featuremodell. Dieser Ansatz setzt Konfigurierung des Featuremodells mit Instanziierung des Klassenmodells gleich, indem das Klassenmodell als Menge aller gültigen Objektmodelle und das Featuremodell als Beschreibung der Menge aller gültigen Feature Konfigurationen angesehen wird. Es bietet hierfür unterschiedliche Sichten an:

Feature Modellierung In dieser Ansicht können Ecore Modelle als ein Featuremodell betrachtet, erzeugt und verändert werden. Hierbei sind Ecore und Featuremodell durch eine bidirektionale Abbildung miteinander gekoppelt. Änderungen in den Modellen können so in beide Richtungen propagiert werden.

Konfiguration In der Konfigurationsansicht können beliebige Ecore Objekte ähnlich den Konfigurationseigenschaften des Feature Modeling Plugins (siehe Abschnitt 2.3.1) konfiguriert werden. Das EMF Metamodell wird hierbei mit den dafür erforderlichen Eigenschaften erweitert.

Um die Synchronisation zwischen den unterschiedlichen Modellen zu realisieren, wurde ein spezielles Metamodell mit Namen *Ecore.fm* eingeführt, das Referenzen auf die Ecore Modellelemente besitzt, die Elemente aus dem Featuremodell bzw. aus den Konfigurationen repräsentieren. Zusätzlich wurde eine bidirektionale Abbildung zwischen *Ecore* und *Ecore.fm* implementiert. Ein typischer Arbeitsablauf bei der Verwendung von *Ecore.fmp* beinhaltet somit die Erstellung eines Metamodells mit Hilfe der Feature Modellierungsansicht, die anschließende Erzeugung von dynamischen Instanzen von beliebigen Komponenten des Metamodells und deren Konfiguration mittels der Konfigurationsansicht. Das Ecore.fmp Projekt wurde 2008 gestoppt. Daher ist die Werkzeugunterstützung nur als experimentell anzusehen. So ist es beispielsweise nicht möglich, verschachtelte Pakethierarchien in Ecore durch ein Feature Modell darzustellen. Ebensowenig gelang es,

den in diesem Kapitel als Beispiel verwendeten Teil der Deltaspeicherung in der MOD2-SCM Produktlinie mit dem Werkzeug Ecore.fmp nachzubauen.

Klassen im Ecore Modell werden automatisch als Feature auf oberster Ebene im Featuremodell interpretiert. Enthalten diese Klassen Attribute, so wird im Featuremodell automatisch eine Feature Gruppe unterhalb des Features, das die Klasse repräsentiert, erstellt. In dieser Feature Gruppe ist nun für jedes Attribut der Klasse ein Feature vorhanden. Die Granularität lässt sich somit nicht kontrollieren. Ebensowenig können Features für Methoden oder Methodenparameter erstellt werden. Durch die Abbildung von Klassen auf Wurzel-Features ist es somit auch nicht möglich, diese Klassen als optional zu deklarieren, da die Kardinalität von Wurzel-Features immer **[1..1]** ist. Die Vererbungshierarchie von Elementen findet sich daher auch nicht im Featuremodell wieder. Außerdem bietet die Konfigurationssicht für ein solches Modell auch „Variabilität“ hinsichtlich der Eigenschaften von Klassen (so kann man beispielsweise in der Konfiguration festlegen, ob eine Klasse abstrakt ist oder nicht). Desweiteren erlaubt Ecore.fmp nur die Verwendung eines einzelnen Paketes im Ecore Modell. Sind darin weitere Pakete geschachtelt, die dann auch noch Elemenete enthalten, so werden diese für die Erstellung des Featuremodells nicht berücksichtigt. Eine Visualisierung von Konfigurationen im Modell bzw. die Erzeugung von konfiguriertem Quelltext ist nicht vorgesehen.

Durch die enge Kopplung der semantisch unterschiedlichen Modelle sind die Möglichkeiten der Modellierung stark eingeschränkt. Vor allem die 1:1 Abbildung von Klassen auf Features macht diesen Ansatz für die Entwicklung von komplexen Produktlinien, wie etwa der in dieser Arbeit als Beispiel verwendeten Produktlinie für Versionskontrollsysteme, komplett unbrauchbar.

6.8.3 Feature Mapper

Das Werkzeug *FeatureMapper*[6] [HW07, HcW08, HKW08] wurde entwickelt, um die Lücke zwischen Featuremodellen und auf Ecore basierten Domänenmodellen zu schließen. Zu diesen Modellen zählen sowohl Eclipse UML2 [7] Modelle, domänenspezifische Sprachen, die mit EMF [SBPM09] erstellt wurden, sowie textuelle Sprachen, die mit Hilfe von EMFText [HJK+09] beschrieben werden.

Zusätzlich zu dem eigenen proprietären Featuremodell erlaubt FeatureMapper auch die Verwendung von Feature- und Variantenmodellen aus pure::variants, sowie pure::variants Familienmodelle. Somit stellt FeatureMapper zur Zeit die einzige Möglichkeit dar, pure::variants im Zusammenhang mit Modellen zu verwenden (mit Ausnahme von pure::variants Matlab und pure::variants EA). Die Abbildung von Features auf das Domänenmodell (in den Papieren zu FeatureMapper immer als *solution-space model* bezeichnet), erfolgt durch ein ebenfalls Ecore basiertes *mapping-model*. Dieses Modell verbindet Domänenmodellelemente (Kindelemente des Ecore Basistyps **EObject**) mit Features

[6] http://www.featuremapper.org
[7] http://www.eclipse.org/modeling/mdt/?project=uml2

oder deren logischen Kombinationen aus dem Featuremodell [Hei09]. Die Erstellung des *mapping-model* kann auf mehrere Arten erfolgen:

Direkte Zuweisung Durch Selektieren des Features im Featuremodell und der zugehörigen Elemente des Domänenmodells und anschließende Zuweisung an das mapping-model wird eine Abbildung zwischen Feature und zugehörigen Modellelementen erstellt.

Aufzeichnung von Änderungen Die Abbildung kann aber auch automatisch erfolgen. Hierbei wird ebenfalls ein Feature ausgewählt, anschließend wird aber ein „Aufnahmemodus“ gestartet, der alle Änderungen im Domänenmodell erfasst und diese Änderungen mit dem Feature verknüpft. Dieser Ansatz bietet die Möglichkeit, Variabilität im Modell zu simulieren. Beispielsweise kann die Kardinalität von Assoziationsenden für verschiedene Features unterschiedlich spezifiziert werden.

FeatureMapper unterstützt mehrere unterschiedliche Visualisierungsmöglichkeiten, um dem Benutzer eine Analyse der erstellten Abbildungen zu ermöglichen [HcW08]:

Realisierungsfilter (Realisation View) Mit Hilfe des sog. Realisierungsfilters ist es möglich, alle Artefakte des Domänenmodells, die exakt zu dem gewählten Ausdruck in der Mapping View gehören, anzuzeigen.

Variantenfilter (Variant View) Der sog. Variantenfilter hebt alle Artefakte, die zu der ausgewählten Variante in der Mapping View gehören im graphischen Editor des Domänenmodells hervor. Hier werden auch alle Artefakte mit dargestellt, die zum nicht variablen Kern der Produktlinie gehören.

Kontextfilter (Context View) FeatureMapper erlaubt die Vergabe von Farben für Features. Hiermit ist es möglich, die zugehörigen graphischen Elemente im Editor des Domänenmodells in den den Features zugeordneten Farben darzustellen.

Eigenschaftenfilter (Property-Changes View) Manche Features erfordern eine Änderung der Struktur und der Eigenschaften des Domänenmodells, etwa wenn ein Feature die Kardinalität eines Assoziationsendes verändert. Diese oftmals schwer zu verstehenden und auch schwer darstellbaren featureabhängigen Änderungen werden in einer separaten Ansicht in markanter Farbe (rot) dargestellt und bieten somit ein Hilfsmittel für den Entwickler.

Neben einzelnen Features, erlaubt FeatureMapper auch die Vergabe von mehreren Features, die mit logischen Ausdrücken miteinander verknüpft sind.

Ein konfiguriertes Modell wird aus dem Domänenmodell durch Modelltransformation erstellt. Hierfür wird eine Konfiguration benötigt, die als Variantenmodell bezeichnet wird. Bei der Transformation wird allerdings nur das zugrunde liegende Ecore basierte Modell angepasst. Die graphische Darstellung dieses Modells in einem Editor (z. B.

Topcased[8]) bleibt unverändert, zeigt also nach wie vor das komplette Modell an.

Der in dieser Arbeit vorgestellte Ansatz unterscheidet sich von FeatureMapper im wesentlichen dadurch, dass umfangreiche Konsistenzbedingungen realisiert wurden, die die Konsistenz der resultierenden Modelle bzw. des erzeugten Quelltextes weitestgehend sicherstellen. FeatureMapper selbst beinhaltet keinerlei Regeln, die die syntaktische Korrektheit bzw. Wohlgeformtheit des konfigurierten Modells gewährleisten [Hei09]. Somit ist es sehr leicht möglich, syntaktisch inkorrekte Modelle zu erzeugen, etwa wenn das Beispiel aus Abschnitt 6.3.4 (Abbildung 6.7) verwendet wird. Dies bedeutet insbesondere für den Modellierer, dass bei der Zuweisung eines Features an ein Modellelement auch alle von dem Modellelement abhängigen Elemente manuell mit hinzugefügt werden müssen. Bei der Zuweisung einer Klasse an ein Feature müssten so beispielsweise auch alle Attribute, Methoden, ein- und ausgehenden Vererbungskanten und alle ein- und ausgehenden Assoziationen mit ausgewählt werden. Andernfalls entstehen bei der Transformation Modelle, die die Wohlgeformtheitsbedingungen der UML verletzen.

Durch die Tatsache, dass bei der Modelltransformation das zugrunde liegende Modell angepasst wird, die graphische Darstellung aber unverändert mitkopiert wird, entstehen ebenso inkonsistente Darstellungen. Im graphischen Modell sind Elemente vorhanden, die kein entsprechendes Gegenstück im zugrunde liegenden Modell besitzen. Bei dem hier vorgestellten Ansatz besitzt jedes graphische Element, das im konfigurierten Domänenmodell angezeigt wird, auch ein entsprechendes Modellelement.

FeatureMapper bietet zwar weiter reichende Möglichkeiten zur Visualisierung einer Konfiguration, dafür ist es aber nicht möglich, direkt konfigurierten Quelltext zu erzeugen.

6.8.4 pure::variants

Das kommerzielle Werkzeug *pure::variants*[9] wird in mehreren unterschiedlichen Versionen angeboten:

Professional & Enterprise Die beiden Versionen von pure::variants, die auch als kostenlose Community Edition verfügbar sind, erlauben die Erstellung von Featuremodellen, Variantenmodellen und Familienmodellen. Eine Abbildung von Features auf Domänenmodelle wird nicht unterstützt, sondern die Abbildung erfolgt hier auf Quelltextebene. Zusätzlich können noch Anforderungsdokumente oder auch Dokumentationen mit verwaltet werden. pure::variants bietet eine eigene auf Prolog basierende Regelsprache, namens *pvProlog*, um Bedingungen für Features oder komplexe Featureausdrücke zu definieren.

IBM Rational Rhapsody Ähnlich zu der Lösung für Sparx Enterprise Architect, bietet pure::variants eine Möglichkeit, IBM Rational Rhapsody Modelle zu konfigurieren.

[8] http://www.topcased.org
[9] http://www.pure-systems.com

Dafür muss in Rational Rhapsody ein neues Profil erstellt werden, das dann auch wiederum auf Rational Rhapsody Constraints angewendet werden kann. Auch hier muss der Benutzer manuell Text in den Rumpf des Constraints eintippen.

Matlab Simulink Diese Version von pure::variants erlaubt Variabilitätsmanagement für Simulink[10] Modelle, inklusive StateFlow und TargetLink Modelle. Alle Modellvarianten können innerhalb eines einzigen Modells verwaltet und konfiguriert werden. Diese Variante ist vor allem für Entwickler von eingebetteten Systemen, z. B. in der Automobilindustrie relevant.

Enterprise Architect Diese Version von pure::variants bietet eine Anbindung an Sparx Enterprise Architect[11], einem Werkzeug, das die Erstellung von UML und SysML Modellen anbietet. Um Elemente des Featuremodells mit Modellelementen zu verknüpfen, muss in Enterprise Architect ein neues Constraint erstellt werden. Mit Hilfe dieses Constraints können anschließend einfache Zuordnungen von einzelnen Features zu Modellelementen hergestellt werden. Die Identifikation der Features erfolgt über deren eindeutigen Bezeichner. Dieser muss allerdings vom Benutzer im Rumpf des Constraints eingetippt werden. Sparx Enterprise Architect unterstützt UML Klassendiagramme, Zustandsdiagramme, sowie Sequenz- und Aktivitätsdiagramme und ist in der Lage Quelltextfragmente aus diesen Diagrammarten zu erzeugen. Allerdings können keine ausführbaren Modelle, wie es in Fujaba möglich ist, erstellt werden. Eine Modellvariante wird durch Auswahl der Features mittels pure::variants und einer anschließenden Modelltransformation erzeugt.

Anbindung an FeatureMapper FeatureMapper (siehe voriger Unterabschnitt) bietet eine Anbindung an *pure::variants*. Das Featuremodell wird in diesem Fall mit pure::variants erstellt. Mit Hilfe der *Mapping View* von FeatureMapper ist es dann möglich, Features aus den pure::variants Modellen mit Ecore-basierten Domänenmodellelementen (z.B. einem UML Klassendiagramm, das mit Topcased erstellt wurde) zu verknüpfen. Anders als bei der Benutzung von FeatureMapper als eigenständiges Werkzeug können die Verknüpfungen zwischen FeatureModell und Domänenmodell bei dieser Art der Anbindung nicht graphisch erstellt werden. Stattdessen werden Regeln benötigt, die in der proprietären, auf Prolog basierten Regelsprache *pcSCL* von pure::variants erfolgen müssen. Bei der Erstellung dieser Regeln werden die Features über ihren eindeutigen Bezeichner angesprochen. Die Zuordnung von Modellelementen zu den neu erschaffenen Regeln erfolgt dann wieder graphisch. Um ein konkretes Produkt zu erstellen wird ebenfalls ein Variantenmodell benötigt, das wiederum mit pure::variants selbst erstellt wird. Die Konfiguration erfolgt ebenso wie in *FeaturePlugin* durch Auswahl der entsprechenden Features. Dieses neu erzeugte Variantenmodell wird nun dem Feature Mapping zugewiesen. In der Mapping View werden nun alle Regeln, die diese Konfiguration erfüllen in grün, alle anderen Regeln in roter Farbe dargestellt. Zur Visualisierung

[10] http://www.mathworks.com/products/simulink/
[11] http://www.sparxsystems.eu/

stehen dem Benutzer die oben beschriebenen *Realisierungsfilter*, *Variantenfilter* und *Kontextfilter* zur Verfügung.

Im Gegensatz zum hier vorgestellten Ansatz bietet *pure::variants Enterprise Architect* keine Möglichkeit, eine Konfiguration durch Visualisierung im Editor graphisch darzustellen. Auch eine direkte Erzeugung von konfiguriertem Quelltext ist nicht möglich. Lediglich die Erzeugung eines konfigurierten Modells, im *pure::variants* Sprachterminus *Projektvariante* genannt, kann mit Hilfe von Enterprise Architect erstellt werden. Auch werden keinerlei Konsistenzbedingungen bei der Zuordnung von Features zu Modellelementen genannt. Durch die Möglichkeit der freien Eingabe von Text besteht eine sehr große Fehleranfälligkeit (z. B. durch Tippfehler), die zu inkonsistenten Konfigurationen führen kann. Eine Propagation auf abhängige Modellelemente durch *pure::variants* findet nicht statt. Inkonsistente UML Modelle werden durch die Konsistenzbedingungen der UML, die in Enterprise Architect selbst realisiert sind, verhindert. Die FeatureMapper Anbindung von pure::variants hingegen bietet keine Möglichkeit, inkonsistente Modelle zu verhindern, da sie den gleichen Beschränkungen unterliegt, wie FeatureMapper selbst.

6.8.5 Apel et al.

Apel et al. beschreiben in [KAT+09, KAS10] einen Ansatz zur Abbildung von Elementen des Featuremodells auf Quellcodefragmente und die Sicherstellung der syntaktischen Korrektheit des konfigurierten Quelltextes. Die bedingte Übersetzung mit Präprozessoren stellt ein einfaches, aber wirksames Mittel zur Darstellung von Variabilität von Softwareproduktlinien im Quelltext dar. Durch Annotation von Quelltext-Fragmenten mit **#ifdef** und **#endif** lassen sich unterschiedliche Varianten des Programms mit oder ohne die eingeschlossenen Fragmente erzeugen. Die syntaktische Korrektheit wird dadurch erreicht, dass vom Benutzer die Feature Annotationen zwar auf Ebene des Quelltextes vergeben werden, der von Apel et al. implementierte Editor diese Annotationen dann aber auf die Elemente des abstrakten Syntaxgraphen abbildet und somit alle abhängigen Elemente ermittelt werden können. Wenn nun ein konkretes Produkt auf Basis einer Konfiguration erstellt wird, wird der abstrakte Syntaxgraph (AST) mittels Transformation in einen neuen AST überführt, bei dem alle Knoten mit Annotationen, die nicht Teil der Konfiguration sind, entfernt wurden [KAT+09]. In einer Referenzimplementierung für Java wurden zwei Regeln implementiert, die ausreichend sind, um die syntaktische Korrektheit des transformierten AST sicherzustellen: (1) Zum einen dürfen nur Knoten annotiert werden, die in der Java Syntax Spezifikation [GJSB05] als optional deklariert sind. (2) Zum anderen müssen alle Kinder eines AST Knotens entfernt werden, wenn der Knoten selbst entfernt wird. Dies bedeutet, dass obligatorische Knoten, wie z.B. der Name einer Klasse im Gegensatz zu Methoden nicht entfernt werden können. Die zweite Regel besagt, dass die Entfernung einer Methode, automatisch die Entfernung all ihrer Parameter und Anweisungen impliziert.

Ein großer Nachteil dieser Vorgehensweise sind die daraus resultierenden negativen Einflüsse auf die Qualität und Wartbarkeit des Quelltextes. Anders als bei modularen Implementierungen, die beispielsweise mit Hilfe von Komponenten [BKPS04] oder

Aspekten [KLM+97] realisiert werden können, findet bei der Verwendung von Präprozessoren keine Trennung von Belangen im Quelltext statt, da die Präprozessoranweisungen mit den Anweisungen der Programmiersprache vermischt sind. Die Lesbarkeit des Quelltextes wird durch die unter Umständen zahlreichen Präprozessoranweisungen massiv verschlechtert. Außerdem ist es sehr leicht möglich fehlerhaften Quelltext zu produzieren. Die Art der Fehler reicht von simplen Syntaxfehlern, über Typfehler bis hin zu Verhaltensfehlern.

Apel et al. zeigen in ihren Arbeiten eine Möglichkeit, diese Probleme teilweise zu beheben. Mittels geeigneter Werkzeugunterstützung ist es möglich, die Vorteile einer modularen Implementierung zu emulieren. Die Bedeutung der Verwendung des Präprozessors zur Darstellung von Variabilität in der industriellen Anwendung zeigt sich etwa auch dadurch, dass kommerzielle Produktlinienwerkzeuge, wie etwa *pure::variants* explizit Präprozessoren unterstützen.

Die vorgestellte Lösung umfasst die *virtuelle Trennung von Belangen* mittels editierbaren Sichten auf den Quelltext, die alle irrelevanten Features ausblenden und den Quelltext bzw. das Quelltextfragment im entsprechenden Kontext (d.h. in der entsprechenden Klasse bzw. Methode) anzeigen. Dieses Konzept sieht eine Erweiterung der Sichten auf Kombinationen von Features und sogar ganze Produktvarianten (Konfigurationen) vor.

Weiterhin werden Ansätze zur Vermeidung von Syntax- und Typfehlern vorgestellt. Verhaltensfehler werden nicht betrachtet, da diese nach Meinung der Autoren kein spezifisches Problem von Präprozessoren darstellen und auch genauso bei modularisierten Implementierungen auftreten können. Syntaxfehler können beispielsweise durch sogenannte *disziplinierte Annotationen* vermieden werden. Darunter versteht man Ansätze, die die Ausdrucksfähigkeit von Annotationen einschränken, ohne aber die Anwendbarkeit in der Praxis zu behindern [KAT+09]. Da Präprozessoren Quelltext als reine Zeichenfolgen interpretieren und es erlauben, beliebige einzelne Zeichen einschließlich Klammern zu annotieren, können dadurch leicht Syntaxfehler entstehen. Bei den disziplinierten Annotationen wird die zugrundeliegende Struktur des Quelltextes berücksichtigt und der Benutzer kann nur ganze Programmelemente, wie z. B. Klassen, Methoden oder Statements annotieren. Allerdings ist somit ein feingranulares Annotieren von Programmbestandteilen nicht mehr möglich. Dieser Ansatz ist nach Auffassung der Autoren aber immer noch deutlich ausdrucksfähiger im Bezug auf die Darstellung von Variabilität, als die Verwendung von Modularisierungsansätzen wie Komponenten, Plugins oder Aspekte [KAK08]. Durch die Analyse der zugrundeliegenden Quelltextstruktur sind aufwändigere Werkzeuge notwendig als bei der Verwendung von undisziplinierten Annotationen.

Um alle Produktvarianten einer Produktlinie auf Typsicherheit zu prüfen, ohne aber jede einzelne Variante übersetzen zu müssen, können *produktlinienorientierte Typsysteme* verwendet werden [KA08, CP06]. Mit Hilfe dieses Ansatzes lassen sich Probleme, wie etwa Methoden oder Klassen, deren Deklaration in bestimmten Produktvarianten entfernt wurde, die aber dennoch referenziert werden, erkennen. Das *produktlinienorien-*

tierte Typsystem prüft für jede Produktvariante, dass entweder die passende Deklaration vorhanden ist, oder auch die zugehörige Implementierung bzw. die Verwendung (z. B. Methodenaufruf) gelöscht wurde.

Dieser Ansatz, Quelltextfragmente mit Featureannotationen auszuzeichnen, entspricht dem in dieser Arbeit vorgestellten Verfahren. Um syntaktische Korrektheit sicherzustellen, werden Regeln spezifiziert, die auf dem abstrakten Syntaxgraphen der Programmiersprache operieren. Die in Abschnitt 6.3.4 vorgestellten Propagationsregeln dienen ebenfalls dazu, die syntaktische Korrektheit bzw. Wohlgeformtheit der resultierenden Modelle oder des erzeugten Quelltextes weitestgehend sicherzustellen. Auch sie arbeiten auf dem abstrakten Syntaxgraphen von Fujaba. Während die Granularität im Ansatz von Apel et al. beschränkt ist, erlaubt der in dieser Arbeit vorgestellte Mechanismus eine beliebige Granularität der Featureannotationen. Sie können auf grob-granularer Ebene der Paketdiagramme erfolgen, aber auch fein-granular für einzelne Methodenparameter. Durch die Verwendung von ausführbaren Modellen an Stelle des Quelltextes findet die Produktlinienentwicklung auf einer höheren Abstraktionsebene statt.

6.9 Diskussion

Im folgenden Abschnitt werden die Beschränkungen des aktuellen Ansatzes diskutiert und alternative Lösungen für einzelne Probleme bzw. zukünftige Verbesserungsmöglichkeiten aufgezeigt.

Werden für Attribute, Methodenparameter bzw. Methodenrückgabewerte Typen verwendet, die nicht Teil der gewählten Konfiguration sind, so wird in der aktuellen Implementierung des Präprozessor Plugins bzw. bei den Modelltransformationen der Basistyp der Zielsprache eingesetzt, im Falle von Fujaba ist dies das Wurzelobjekt von Java, **java.lang.Object**. Selbstverständlich sind hier auch andere Strategien denkbar und mit geringem Aufwand auch zu realisieren:

Propagation Die Propagationsregeln könnten dahingehend erweitert werden, dass in einem zweiten Durchlauf alle Typen von Attributen, Methodenparametern und Methodenrückgabewerten analysiert werden. Ist der entsprechende Typ ausgezeichnet, so werden auch die referenzierten Typen ausgezeichnet. Dies geschieht mit den bisherigen Regeln nicht, da in der UML jeder Typ nur Referenzen auf seine Instanzen hat, nicht aber auf die Verwendung als Attribut-, Parameter- oder Rückgabetyp. Wird diese Regel analog zu den bestehenden Regeln implementiert, so werden die entsprechenden Auftreten des Typs aus dem konfigurierten Modell herausgefiltert.

Nichtexistenten Typ belassen Eine weitere denkbare Möglichkeit wäre, den nicht existenten Typ im resultierenden Modell oder Quelltext zu belassen und statt dessen den Benutzer mit einem Warnhinweis auf den Konflikt aufmerksam zu machen. Im Falle von generiertem Quelltext übernimmt diese Aufgabe der Compiler, der Übersetzungsfehler meldet. In Fujaba könnten diese Modellelemente ebenfalls durch

farbliche Markierung an der Benutzerschnittstelle gekennzeichnet werden. Dem Benutzer können auf diese Art Inkonsistenzen schnell angezeigt werden.

Eine ähnliche Problematik ergibt sich ebenfalls durch die Auszeichnung eines Methodenrückgabewertes mit einem Feature Tag. Die aktuelle Implementierung des Werkzeuges setzt dann den entsprechenden Basistyp ein. Denkbar wäre hier aber auch beispielsweise eine Propagation des Feature Tags auf die Methode.

Die Propagationsregeln aus Abschnitt 6.3.4 sind nur notwendige, aber nicht hinreichende Bedingungen für die Konsistenz des konfigurierten Modells. Mit diesen Regeln kann ein Filtern von obligatorischen Elementen, wie etwa Startaktivitäten in Storydiagrammen nicht verhindert werden. In diesen Situationen ist eine Benutzerinteraktion erforderlich, was bedeutet, dass der Benutzer durch Visualisierung an der Benutzerschnittstelle auf die entsprechenden Konsistenzverletzungen aufmerksam gemacht werden muss.

Ebenso ist die Propagation von Feature Tags auf Modellelemente, die ausgezeichnete Typen in freien Textfragmenten verwenden, wie etwa in Constraints, Kollaborationsaufrufen, Statement Aktivitäten und als Bedingung an Transitionen nur eingeschränkt möglich. Mit Hilfe des Fujaba Textparsers [Opp06] ist es möglich, diese Auftreten zu entdecken und auch auszuzeichnen, was im Falle von Constraints und Kollaborationsaufrufen ausreichend ist, um die syntaktische Korrektheit des erzeugten Quelltextes sicherzustellen. Wird allerdings eine Bedingung an einer Transition ausgezeichnet und ist diese nicht Bestandteil der aktuellen Konfiguration, so wird sie herausgefiltert. Dies hat wiederum zur Folge, dass das Storypattern nun zwei ausgehende Transitionen besitzt, wovon eine ungültig ist, da eine Bedingung an einer Transition immer ein boolscher Ausdruck ist, der besagt, dass der Kontrollfluss über die vorhandene Transition weiterläuft, sofern diese Bedingung erfüllt ist, die zweite ausgehende Transition aus der Aktivität wird bei Nichterfüllung dieser Bedingung verwendet. Es muss ebenso untersucht werden, ob nur die Bedingung mit einem Feature Tag versehen wird, oder die gesamte Transition. Im Falle der Transition müssen dann selbstverständlich entweder alle nachfolgenden Aktivitäten und Transitionen bis zum nächsten Stop- oder Vereinigungsknoten ebenfalls ausgezeichnet werden, bzw. es muss sichergestellt sein, dass diese Elemente über einen alternativen Kontrollflusspfad erreichbar sind. Um dieses Problem zu lösen sind umfangreiche Eingriffe in den Sequencer der Fujaba Codegenerierung notwendig. Der Sequencer dient dazu, zu den einzelnen Elementen des abstrakten Syntaxgraphen atomare Operationen zu erzeugen, diese anschließend zu sortieren und zu strukturieren, bevor diese Struktur optimiert und an die einzelnen **CodeWriter** der Codegenerierung weitergereicht wird [GSR05]. Da der Präprozessor für die Codegenerierung erst im letzten Schritt angesprochen wird, die Transitionen aber bereits vorher in atomare Operationen umgewandelt werden, ist somit eine Anpassung des Sequencers unumgänglich.

Das Domänenmodell, das alle Varianten der Produktlinie umfasst, unterliegt im Bezug auf der Möglichkeit der Darstellung von Variabilität den Beschränkungen der UML. Auch die Option, Variabilität in den Methodenimplementierungen mit Hilfe von Storypattern

zu beschreiben, ist durch mehrere Umstände stark begrenzt: Zum einen erlaubt es Fujaba nicht, einem Objekt je nach gewähltem Feature unterschiedliche Typen zuzuordnen zum anderen erlaubt Fujaba maximal zwei ausgehende Transitionen pro Aktivität in Storydiagrammen. Der Modellgenerator (siehe Abschnitt 6.4) erlaubt die Erzeugung eines Modells auch, wenn das Quellmodell nicht gültig ist (weil es z.b. mehr als zwei ausgehende Transitionen pro Aktivität enthält), die Erzeugung von konfiguriertem Quelltext funktioniert allerdings nicht, da die Transitionen wie oben beschrieben vom Sequencer analysiert werden und dieser nur zwei ausgehende Transitionen pro Aktivität verarbeiten kann. Auch in diesem Falle ist zu untersuchen, ob Propagationsregeln dahingehend entwickelt werden können, dass der Benutzer nur die ausgehende Transition auszeichnen muss und sämtliche nachfolgenden Modellelemente werden automatisch analog ausgezeichnet. Auch hierfür sind Anpassungen am Sequencer von Nöten.

Die in Abschnit 6.3.4 aufgestellten Regeln sind notwendige, aber keine hinreichenden Bedingungen. So wäre es z. B. durchaus denkbar, dass in Storydiagrammen notwendige Elemente, wie etwa der Startknoten oder der Endknoten mit Features ausgezeichnet werden. Werden diese notwendigen Elemente dann herausgefiltert, da das zugehörige Feature nicht Teil der Konfiguration ist, so entsteht ein inkonsistentes Modell. Zur Lösung dieser Probleme sind unterschiedliche Strategien denkbar:

Verbot von Feature Tags an obligatorischen Elementen von Storydiagrammen, wie Start- bzw. Endknoten. Ein Storydiagramm muss genau einen Start- und mindestens einen Endknoten besitzen.

Propagation Werden obligatorische Elemente mit Feature Tags ausgezeichnet, so werden diese Tags auf alle Elemente des Storydiagramms propagiert, um so die Konsistenz zu gewährleisten.

In der derzeitigen Implementierung von *MODPLFeaturePlugin* ist keine Änderungspropagation zwischen Featuremodell und Domänenmodell vorhanden. Eine Propagation von Änderungen wie Umbenennen, Löschen oder auch das Zuweisen eines anderen Vaterknotens von Features sollte analog dem Mechanismus der Propagation von Änderungen zwischen Paketdiagramm und Fujaba Modell (siehe Abschnitt 5.6.7) implementiert werden. Dabei ist eine bidirektionale Synchronisation nicht erforderlich, da Änderungen am Featuremodell wegen der Möglichkeit zusätzliche Bedingungen zu spezifizieren, direkt im entsprechenden Werkzeug erfolgen sollten.

Die vorliegende Arbeit befasst sich im Bezug auf Featureausdrücke nur mit der Konjunktion von einzelnen Features, die einem Modellelement zugeordnet werden, was im Umkehrschluss bedeutet, dass das Modellelement nur im konfigurierten Modell bzw. im konfigurierten Quelltext vorhanden ist, sofern auch alle an der Konjunktion beteiligten Features in der Konfiguration enthalten sind. Dies führt zu Einschränkungen bei der Vergabe von Features an Modellelemente.

Ein Nebeneffekt der Benutzung von Fujaba und der Tatsache, dass sich Fujaba nicht vollständig an die in der UML Spezifikation beschriebenen Regeln hält, ist eine gegenüber dem UML Standard erweiterte Möglichkeit der Darstellung von Variabilität in Klassendiagrammen. So erlaubt Fujaba die Definition von Methoden mit identischer Signatur. Mittels Vergabe von Feature Tags kann so für eine bestimmte Produktvariante die entsprechende Methode mit der zugehörigen Implementierung ausgewählt werden. Sind diese Methoden mit alternativen Features ausgezeichnet, so ist gewährleistet, dass gültiger Quelltext erzeugt wird. Die UML Spezifikation fordert aber von **BehavioralFeatures**, der Oberklasse von **Operation** die Eindeutigkeit des Namens. Dieser Name ist die Signatur der Operation, die aus dem Namen des BehavioralFeatures, sowie aller über die **ownedParameters** Assoziation erreichbaren Parameter besteht (siehe [OMG09b], S. 48f).

6.10 Zusammenfassung und Ausblick

In diesem Abschnitt wurde ein Ansatz zur Abbildung von Elementen des Featuremodells auf Elemente des Domänenmodells vorgestellt. Das Domänenmodell ist hierbei ein Multi-Varianten-Modell. Ziel ist es, konkrete Produktvarianten anhand einer Konfiguration und dem annotierten Domänenmodell zu erzeugen. Dies kann entweder durch direkte Erzeugung von Quelltext geschehen, oder in einem Zwischenschritt durch die Generierung eines Modells, das nur die Modellelemente enthält, die der gewählten Featureauswahl entsprechen.

Durch die Verwendung eines UML Profils ist der vorgestellte Ansatz universell für alle UML Werkzeuge anwendbar. Eine leichte Modifikation des Konzeptes erlaubt es, auch Ecore basierte Modelle mit Feature Annotationen auszuzeichnen. Da das Ecore Metamodell für jedes **EObject** die Zuordnung von beliebigen **EAnnotation**s erlaubt, ist dieser Ansatz auch auf derartige Modelle übertragbar. Auch für Werkzeuge, deren technische Realisierung die Vergabe von Stereotypen, nicht aber von Attributen für diese Stereotypen erlaubt, wie es etwa bei Fujaba der Fall ist, kann durch Verwendung von *tagged values* oder Annotationen eine Unterstützung angeboten werden.

Zukünftige Forschungsarbeiten könnten weiterhin darauf abzielen, Erweiterungsmechanismen zur Darstellung von Variabilität in der statischen Struktur, sowie auch in den Storydiagrammen von Fujaba Modellen auszudrücken. Aktuell besteht beispielsweise keine Möglichkeit, Variabilität von Assoziationen auszudrücken. Diese Variabilität kann sich auf verschiedene Arten äußern: (1) in den Kardinalitäten der Assoziationsenden oder (2) im Quell- bzw. Zieltyp an dem die Assoziation beginnt bzw. endet. Auch bei den Fujaba Storydiagrammen besteht beispielsweise keine Möglichkeit, Variabilität durch Objektinstanziierungen auszudrücken. Sollen alternative Kontrollflusspfade verwendet werden, so ist dazu ebenfalls eine Anpassung des Sequencers unumgänglich. Weitere Forschungsarbeiten sind im Bereich der Verwendung von Featureausdrücken (Konjunktion, Disjunktion und Negation) an Modellelementen, sowie bei der Änderungspropagation zwischen Featuremodell und Domänenmodell denkbar. Zusätzliche Erweiterungen der bestehen-

den Implementierung könnten eine Erweiterung der Benutzerschnittstelle umfassen, die dem Anwender bei der Konfliktlösung Alternativen anbietet bzw. schon bei der Vergabe von Features vor möglichen Konsistenzverletzungen warnt.

7 Zusammenfassung und Ausblick

Softwareproduktlinien und modellgetriebene Entwicklung sind zwei moderne Herangehensweisen aus dem Gebiet der Softwaretechnik, um der immer noch aktuellen Problematik der Softwarekrise zu begegnen. Gerade jetzt in Zeiten der weltweiten Wirtschaftskrise sind moderne und vor allem effiziente Entwicklungsmethoden gefordert, die einerseits eine maßgebliche Reduktion des Entwicklungsaufwands bei einer gleichzeitigen Erhöhung der Softwarequalität versprechen.

Die Idee hinter Softwareproduktlinien ist *geplante Wiederverwendung* von bereits bestehenden (und somit getesteten und bewährten) Softwarekomponenten. Diese Komponenten werden zu einer gemeinsamen Plattform ausgebaut mit Hilfe derer kundenspezifische Produkte effizient mit nur geringen Anpassungen abgeleitet werden können. Die Entwicklung von Softwareproduktlinien erfolgt dabei immer in zwei unterschiedlichen Ebenen: (1) in der Domänenentwicklung werden die gemeinsamen und variablen Bestandteile der Plattform entwickelt. Durch *Variabilität* werden dabei die möglichen Unterschiede der einzelnen Produkte innerhalb der Produktlinie gekennzeichnet. Variabilität stellt die Grundvoraussetzung für eine gezielte Konstruktion von Produktartefakten und deren Wiederverwendung in der Anwendungsentwicklung dar. (2) Anwendungsentwicklung hingegen befasst sich mit der Entwicklung einzelner Produkte im Rahmen der Produktlinie. Die Wiederverwendung von Artefakten der gemeinsamen Plattform nimmt dabei eine zentrale Rolle ein. Diese Artefakte werden durch Zusammenfügen mit möglichst geringen produktspezifischen Weiterentwicklungen zu einem Produkt verschmolzen. Dieser Vorgang wird als *Konfigurierung* bezeichnet.

Modellgetriebene Entwicklung auf der anderen Seite zielt auf eine Erhöhung des Abstraktionsgrades bei der Spezifikation von Softwaresystemen ab. Formale Modelle dienen der Beschreibung von Problemen und Lösungen auf einer höheren Ebene als Quelltext. Dabei ist zu beachten, dass in der modellgetriebenen Entwicklung (im Gegensatz zur liberaleren *modellbasierten* Vorgehensweise) die Modelle mit dem Quelltext gleichzusetzen sind. Aus diesen formalen Modellen wird entweder mit Hilfe von Generatoren lauffähiger Quelltext erzeugt, der anschließend übersetzt und ausgeliefert werden kann, oder aber es werden spezielle Laufzeitumgebungen mit zugehörigen Interpretern bereitgestellt, um diese Modelle auch auszuführen.

7.1 Zielsetzung dieser Arbeit

Das Ziel der vorliegenden Arbeit war es, Modelle und Werkzeuge zur Kombination dieser beiden Techniken zu entwickeln. Wie bereits in Kapitel 1 dargestellt, verspricht diese

Herangehensweise die Ausnutzung der Vorteile von beiden Entwicklungsmethoden. Soweit als möglich wird hierbei auf bereits verfügbare und erprobte Werkzeuge gesetzt, wie etwa *FeaturePlugin* [AC04] zur Modellierung von Variabilität in der Produktlinie oder *Fujaba* [Zün02] zur Erstellung des ausführbaren Domänenmodells. Die Domäne Softwarekonfigurationsverwaltung dient hierbei als durchgängiges praxisbezogenes Anwendungsbeispiel. Sämtliche Modellausschnitte sind hierbei dem MOD2-SCM Projekt [BDW08b, BDW09, BD09c, Dot10] entnommen, das die Erstellung einer Produktlinie für Softwarekonfigurationsverwaltungssysteme zum Ziel hat.

Der im Rahmen dieser Arbeit verwendete Modellierungsansatz zur *modellgetriebenen Entwicklung von Softwareproduktlinien* unterscheidet hierbei, wie bei Produktlinien üblich, die Ebenen Domänenentwicklung und Anwendungsentwicklung. Zunächst wird in der Domänenentwicklung eine Analyse der Anwendungsdomäne durchgeführt. Dies geschieht mit Hilfe der Feature-orientierten Domänenanalyse (FODA) [KCH$^+$90], einer seit etwa 20 Jahren erprobten Vorgehensweise, die auch in industriellen Anwendungen auf hohe Akzeptanz stößt. Das Ergebnis dieser Analyse wird in einem sogenannten *Featuremodell* erfasst. Dieses Modell beschreibt variable und invariante Bestandteile der Produktlinie, alternative Ausprägungen der variablen Teile und deren Beziehungen untereinander. Dieses Featuremodell bildet die Basis für die Erstellung des ausführbaren Domänenmodells, das die einzelnen Features realisiert. Das ausführbare Domänenmodell enthält dabei sowohl strukturelle Modelle (ausgedrückt durch Paket- und Klassendiagramme), als auch Modelle, die das Verhalten beschreiben (ausgedrückt durch Storydiagramme).

Anders als beim klassischen Produktlinienansatz wird die Anwendungsentwicklung in diesem Falle aber auf einen reinen Konfigurierungsprozess reduziert. Dabei wird zuerst das Featuremodell konfiguriert, indem die gewünschten Eigenschaften ausgewählt werden. Das Ergebnis dieser Auswahl – die *Featurekonfiguration* – dient anschließend als Grundlage zur Konfigurierung des ausführbaren Domänenmodells. Abschluss der Anwendungsentwicklung stellt die Erzeugung von ausführbarem Quelltext aus dem konfigurierten Domänenmodell dar. Hierbei ist zu beachten, dass lediglich der erste Schritt, also die Konfigurierung des Featuremodells, interaktiv durchgeführt werden muss. Die übrigen Abläufe erfolgen vollständig automatisiert.

Die Verwendung von bereits existierenden Werkzeugen war aber bei weitem nicht ausreichend, um die Anforderungen für die modellgetriebene Entwicklung von Softwareproduktlinien abzudecken. Die zur Verfügung stehende Werkzeugkette zeigte gravierende Lücken, die nur durch Eigenentwicklungen zu schließen waren. So bietet etwa *FeaturePlugin* keine Möglichkeit der Anbindung an Werkzeuge zur Erstellung des Domänenmodells, sondern beschränkt sich einzig und allein auf die Featuremodellierung. Das im Rahmen dieser Arbeit entwickelte Werkzeug *MODPLFeaturePlugin* dient dazu, FeaturePlugin und Fujaba zu integrieren. Es besteht hierbei aus zwei Komponenten: (1) einer Erweiterung des Fujaba Editors und (2) einem Konfigurator für das Domänenmodell. Aber auch Fujaba selbst erfüllt nicht die Anforderungen, die die modellgetriebene Entwick-

lung von Softwareproduktlinien benötigt. So bietet Fujaba etwa keine Unterstützung für das Modellieren im Großen. Zu diesem Zweck wurde das *Paketdiagramm Werkzeug* entwickelt. Es unterstützt den Modellierer bei der Planung der Architektur und erlaubt eine explizite Darstellung von Abhängigkeiten zwischen den einzelnen Paketen. Diese Abhängigkeiten werden mit Hilfe eines Integrators während der Arbeit am Domänenmodell in Fujaba permanent überwacht und etwaige Verletzungen werden dem Benutzer angezeigt. Gerade dies ist essenziell für den Erfolg einer Produktlinie, in der die einzelnen variablen Bestandteile möglichst orthogonal kombinierbar sein sollen und daher über keinerlei Abhängigkeiten untereinander verfügen dürfen.

7.2 Erkenntnisse

Im MOD2-SCM Projekt, das in der Dissertation von Alexander Dotor [Dot10] genauer beschrieben wird, wurden die im Rahmen dieser Arbeit entwickelten Werkzeuge zur modellgetriebenen Entwicklung von Softwareproduktlinien bereits erfolgreich eingesetzt. Aufgrund der enormen Größe des Projektes wurde die Domänenanalyse mit einer Weiterentwicklung von FODA - FORM (**f**eature **o**riented **r**euse **m**ethod) [KKL$^+$98] durchgeführt. Ein wesentlicher Unterschied gegenüber von FODA ist die Klassifizierung von Features in verschiedene Bereiche. Diese Bereiche sind:

Capability Layer: Features auf dieser Ebene beschreiben die Produktlinie durch Dienste, Operationen und Qualitäten, die angeboten werden sollen.

Operating Environment Layer: Die Features, die hier spezifiziert werden, beschreiben die Produktlinie durch Eigenschaften der Hard- und Softwareumgebungen, in denen die konkreten Anwendungen später eingesetzt werden.

Domain Technology Layer: Die Techniken, die speziell in der Anwendungsdomäne verwendet werden, werden mit Features auf dieser Ebene beschrieben. In der MOD2-SCM Produktlinie werden hier etwa die Features zur Realisierung der unterschiedlichen Module (Produktraum, Versionsraum, etc.) definiert.

Implementation Technique Layer: Auf dieser Ebene beschreiben Features die Produktlinie mit Hilfe der verwendeten grundlegenden Techniken. Dabei handelt es sich nicht um domänenspezifische Techniken, sondern um Basistechniken der Datenübertragung und -verarbeitung.

Durch diese Klassifizierung ergibt sich eine höhere Anzahl an Features im Featuremodell. So besteht das MOD2-SCM Featuremodell aus 127 unterschiedlichen Features, die sich auf die oben genannten vier Ebenen verteilen. Der weitaus größte Anteil entfällt hierbei natürlich auf die Ebene, in der die Domänentechnologie beschrieben wird.

Basierend auf den Erkenntnissen der Domänenanalyse wurde mit Hilfe von Fujaba das Domänenmodell entwickelt, das die im Featuremodell spezifizierten Eigenschaften realisiert. Derzeit besteht das Domänenmodell aus 136 Klassen, die auf 36 verschiedene Pakete verteilt sind. Dabei ist zu beachten, dass das Domänenmodell aufgrund der

enormen Größe auf mehrere Fujaba Projekte aufgeteilt wurde, um so eine bessere Übersichtlichkeit zu erreichen. Den weitaus größten Teil stellt hierbei das Projekt dar, das die Kernmodule und die darauf aufbauenden Module zur Beschreibung des Versionsraums und der Speicherung von Deltas beinhaltet. Weitere Projekte dienen der Realisierung der unterschiedlichen Produktmodelle bzw. zur Implementierung von Client- und Serverschnittstellen.

Bei der Realisierung des Domänenmodells wurde explizit darauf geachtet, einen sehr hohen Grad an Entkopplung der einzelnen Module zu erreichen. Dies hatte zur Folge, dass die Granularität der Feature-Annotationen auf Paket- bzw. Klassenebene erfolgt. Eine Vergabe von Feature-Annotationen innerhalb von Storydiagrammen war nicht notwendig. Eine detaillierte Beschreibung der verwendeten Modellierungstechniken zur Erreichung der losen Kopplung erfolgt in der Dissertation von Alexander Dotor [Dot10].

Als sehr hilfreich bei dieser Aufgabe stellte sich die Verwendung des Paketdiagrammeditors heraus. Durch die Möglichkeit, Pakete und deren Abhängigkeiten zu visualisieren, konnten implizite und nicht gewünschte Kopplungen zwischen einzelnen Elementen des Domänenmodells leicht ermittelt und beseitigt werden. Dies wäre durch die alleinige Verwendung von Fujaba nicht möglich gewesen, da Fujaba hierfür keinerlei Unterstützung anbietet und eine Analyse der Abhängigkeiten nur auf Ebene der Klassendiagramme selbst bei kleineren Projektgrößen kaum mehr vom Modellierer durchgeführt werden kann.

Anschließend erfolgte die Zuordnung von Features aus dem Featuremodell an Elemente des Domänenmodells. Im Verlauf der Vergabe von Feature-Annotationen zeigte sich jedoch, dass die derzeit vorhandenen Möglichkeiten zur Bildung von Feature-Ausdrücken nicht ausreichen. Da derzeit lediglich eine Konjunktion von Features unterstützt wird, musste an einigen Stellen das Featuremodell selbst erweitert werden, um eine Negation von Features ausdrücken zu können. Dazu wurde an der entsprechenden Stelle im Featuremodell ein neues Feature eingeführt und die Features wurden als sich gegenseitig ausschließend deklariert. Somit können wieder gültige Konfigurationen erstellt werden.

Mit Hilfe des Propagationsmechanismus, der im Werkzeug *MODPLFeaturePlugin* realisiert und dessen Konzeption in Abschnitt 6.3.4 beschrieben ist, kann die Vergabe von Features an Elemente des Domänenmodells auf ein Minimum reduziert werden. Durch die automatische Propagation der Features auf abhängige Elemente ist sichergestellt, dass bei der anschließenden Konfigurierung ein gültiges Modell entsteht. Durch diesen Mechanismus wird der Modellierer einerseits entlastet, andererseits hat er jedoch die Möglichkeit der visuellen Kontrolle von automatisch hinzugefügten Feature-Annotationen und deren Auswirkungen auf das Domänenmodell.

Im Rahmen der Arbeit von Alexander Dotor wurden mit Hilfe des Featuremodells und des ausführbaren Domänenmodells einige Konfigurationen von Softwarekonfigurationsverwaltungssystemen erstellt und getestet. Darunter befanden sich Konfigurationen, die ähnlich den bekannten Versionskontrollsystemen CVS [Ber90] und Subversion [CSFP08]

arbeiten, aber auch sämtliche möglichen Kombinationen von Versions-, Deltaspeicherung und Synchronisation. Durch automatisierte und parametrisierte Testfälle werden 96 verschiedene Konfigurationen abgedeckt und auf ordnungsgemäße Funktionalität überprüft. Durch diese Tests wird sichergestellt, dass nicht vorhandene Abhängigkeiten im Paketdiagramm auch tatsächlich zu orthogonal kombinierbaren Modulen führen.

7.3 Ausblick

Zukünftige Arbeiten sind vor allem im Bereich der Featureausdrücke notwendig. Durch die Beschränkung auf die reine Konkatenation von Features an Modellelementen ist die Modellierung an dieser Stelle noch begrenzt. Hier wären logische Kombinationen von einzelnen Features zu komplexen Featureausdrücken wünschenswert und auch notwendig.

Auch wäre eine engere Integration des Werkzeugs *FeaturePlugin* erstrebenswert. Durch die Tatsache, dass der Datenaustausch zwischen dem Werkzeug zur Featuremodellierung und dem zur Erstellung des Domänenmodells nur in einer Richtung und auch nur dateibasiert mittels XML-Dateien realisiert ist, ist ein Abgleich von Änderungen im Featuremodell (vor allem im Bezug auf Umbenennung) nur schwer möglich. Eine engere Anbindung der beiden Modelle mit Hilfe eines Integrators, analog zur Anbindung des Paketdiagramm Editors an Fujaba, könnte hier Abhilfe schaffen.

Während der Ansatz zur Annotation von Domänenmodellelementen mit Elementen aus dem Featuremodell auf konzeptioneller Ebene allgemein für alle auf der UML-basierten Sprachen definiert wurde, beschränkt sich die derzeitige Implementierung rein auf den Einsatz mit dem Werkzeug Fujaba. Eine Portierung des Ansatzes auf Ecore-basierte Modelle ist problemlos möglich. Das EMF Metamodell sieht vor, dass jedes Modellelement mit zusätzlichen Informationen versehen werden kann. Über den Mechanismus der *EAnnotation*s können analog zu den FeatureTags diese Informationen zur Kopplung von Featuremodell und Domänenmodell gespeichert werden. Auch Mechanismen zur Transformation von annotierten Ecore Modellen in konfigurierte Ecore Modelle können problemlos durch geringfügige Anpassungen aus der bestehenden Lösung übernommen werden (zumindest für Klassendiagramme und Aktivitätsdiagramme). Da die EMF-Codegenerierung darüber hinaus auch noch Anpassungen erlaubt, sollte auch eine Erzeugung von konfiguriertem Quelltext direkt aus dem annotierten Ecore Modell heraus realisierbar sein.

Darüber hinaus wäre zu klären, ob der Paketdiagrammeditor und insbesondere die strikte Interpretation der Sichtbarkeitsregeln der UML auch für große Softwareprojekte, die nicht auf die Erstellung einer Produktlinie abzielen, von Nutzen sein kann. Die Anbindung des Integrators könnte zumindest auf technischer Ebene noch enger gestaltet werden, indem ein „echter“ Livemodus implementiert wird, der eine Benutzerinteraktion zum Abgleich beider Modelle obsolet macht. Desweiteren könnten sich zukünftige For-

schungsarbeiten mit der Rolle von Komponentendiagrammen im Rahmen der Disziplin Modellieren im Großen im Produktlinienkontext beschäftigen. Während die Verwendung von Paketdiagrammen gerade bei einer a priori Planung der Architektur unerlässlich ist, können Komponentendiagramme auch während der Verfeinerung der groben Struktur noch zur Analyse und Kontrolle von Abhängigkeiten innerhalb des Modells verwendet werden. Sie bieten eine Klassifizierung von Modellelementen in logische Einheiten, die durch Komponenten mit gleicher Funktionalität ersetzt werden können.

Da das in dieser Arbeit beschriebene Verfahren zur modellgetriebenen Entwicklung von Softwareproduktlinien und die dazu erstellten Werkzeuge nur im Rahmen von Forschungsprojekten eingesetzt wurden, ist eine Kooperation mit einem Industriepartner zur Evaluatierung der Vorgehensweise im produktiven Umfeld wünschenswert. Nur durch den Einsatz in der Praxis können wertvolle Rückschlüsse zur Akzeptanz des Ansatzes gewonnen werden und etwaige Verbesserungsvorschläge in die Weiterentwicklung der Werkzeuge einfließen.

Literaturverzeichnis

[AC04] Antkiewicz, Michal ; Czarnecki, Krzysztof: FeaturePlugin: feature modeling plug-in for Eclipse. In: *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA : ACM Press, 2004, S. 67–72

[AKRS06] Amelunxen, Carsten ; Königs, Alexander ; Rötschke, Tobias ; Schürr, Andy: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, Arend (Hrsg.) ; Warmer, J (Hrsg.): *Model Driven Architecture - Foundations and Applications: Second European Conference* Bd. LNCS 4066. Genova, Italy : Springer-Verlag, October 2006 2006, S. 361–375

[Baa03] Baar, Thomas: The Definition of Transitive Closure with OCL - Limitations and Applications. In: *Ershov Memorial Conference*, 2003, S. 358–365

[Bac98] Backus, John W.: The History of Fortran I, II, and III. In: *IEEE Annals of the History of Computing* 20 (1998), Nr. 4, S. 68–78

[Bat05] Batory, Don S.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, J. H. (Hrsg.) ; Pohl, Klaus (Hrsg.): *SPLC* Bd. 3714, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–28936–4, S. 7–20

[BD06] Buchmann, Thomas ; Dotor, Alexander: Building Graphical Editors with GEF and Fujaba. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.) ; University of Paderborn (Veranst.): *Proceedings of the Fujaba Days 2006*. Paderborn, Germany, 2006

[BD09a] Buchmann, Thomas ; Dotor, Alexander: Constraints for a fine-grained mapping of feature models and executable domain models, CTIT, Juni 2009 (CTIT Workshop Proceedings 1), S. 9–17

[BD09b] Buchmann, Thomas ; Dotor, Alexander: Mapping Features to Domain Models in Fujaba. In: Gorp, Pieter van (Hrsg.): *Proceedings of the 7th International Fujaba Days*. Eindhoven, The Netherlands, November 2009, S. 20–24

[BD09c] Buchmann, Thomas ; Dotor, Alexander: Towards a Model-Driven Product Line for SCM Systems. In: [JDM09], S. 174–181

[BDFW08] Buchmann, Thomas ; Dotor, Alexander ; Förtsch, Sabrina ; Westfechtel, Bernhard: Model-Driven Software Development with Graph Transformations: A Comparative Case Study. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Application of Graph Transformation with Industrial Relevance*. Berlin, Heidelberg : Springer-Verlag, 2008 (LNCS), S. 345–360

[BDG07] Buchmann, Thomas ; Dotor, Alexander ; Geiger, Leif: EMF Code Generation with Fujaba. In: Geiger, Leif (Hrsg.) ; Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proceedings of Fujaba Days 2007*. Kassel, Germany, 2007

[BDK09] Buchmann, Thomas ; Dotor, Alexander ; Klinke, Martin: Supporting Modeling in the Large in Fujaba. In: Gorp, Pieter van (Hrsg.): *Proceedings of the 7th International Fujaba Days*. Eindhoven, The Netherlands, November 2009, S. 59–63

[BDW07] Buchmann, Thomas ; Dotor, Alexander ; Westfechtel, Bernhard: Model-Driven Development of Graphical Tools - Fujaba meets GMF. In: Filipe, Joaquim (Hrsg.) ; Helfert, Markus (Hrsg.) ; Shishkov, Boris (Hrsg.): *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT 2007)*. Barcelona, Spain : INSTICC Press, Setubal, Portugal, July 2007, S. 425–430

[BDW08a] Buchmann, Thomas ; Dotor, Alexander ; Westfechtel, Bernhard: Experiences with Modeling in the Large in Fujaba. In: Assmann, Uwe (Hrsg.) ; Johannes, Jendrik (Hrsg.) ; Zündorf, Albert (Hrsg.) ; Technische Universität Dresden (Veranst.): *Proceedings of the 6th International Fujaba Days*. Dresden, Germany, 2008, S. 5–9

[BDW08b] Buchmann, Thomas ; Dotor, Alexander ; Westfechtel, Bernhard: MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system. In: Deridder, Dirk (Hrsg.) ; Gray, Jeff (Hrsg.) ; Pierantonio, Alfonso (Hrsg.) ; Schobbens, Pierre-Yves (Hrsg.): *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM08)* Bd. 1. Toulouse, France, September 2008, S. 50–65

[BDW09] Buchmann, Thomas ; Dotor, Alexander ; Westfechtel, Bernhard: Model-driven Development of Software Configuration Management Systems - A Case Study in Model-driven Engineering. In: *Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT 2009)* Bd. 1. Sofia, Bulgaria : INSTICC Press, July 2009, S. 309–316

[BDW10] Buchmann, Thomas ; Dotor, Alexander ; Westfechtel, Bernhard: Werkzeuge zur modellgetriebenen Entwicklung von Produktlinien: Ein Erfahrungsbericht am Beispiel von Versionskontrollsystemen. In: Birk, Andre-

as (Hrsg.) ; Schmid, Klaus (Hrsg.) ; Völter, Markus (Hrsg.): *Tagungsband der PIK 2010 Produktlinien im Kontext*, 2010

[Ber90] Berliner, Brian: *CVS II: Parallelizing Software Development.* 1990

[BFG+01] Bosch, Jan ; Florijn, Gert ; Greefhorst, Danny ; Kuusela, Juha ; Obbink, J. H. ; Pohl, Klaus: Variability Issues in Software Product Lines. In: Linden, Frank van d. (Hrsg.): *PFE* Bd. 2290, Springer, 2001 (Lecture Notes in Computer Science). – ISBN 3–540–43659–6, 13-21

[BGSZ08] Bork, Manuel ; Geiger, Leif ; Schneider, Christian ; Zündorf, Albert: Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In: Schieferdecker, Ina (Hrsg.) ; Hartman, Alan (Hrsg.): *ECMDA-FA* Bd. 5095, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–69095–5, S. 33–47

[BH89] Belina, F. ; Hogrefe, D.: The CCITT-specification and description language SDL. In: *Comput. Netw. ISDN Syst.* 16 (1989), Nr. 4, S. 311–341. – ISSN 0169–7552

[BHST04] Bontemps, Yves ; Heymans, Patrick ; Schobbens, Pierre-Yves ; Trigaux, Jean-Christophe: The semantics of FODA feature diagrams. In: *Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004

[BKPS04] Böckle, Günter (Hrsg.) ; Knauber, Peter (Hrsg.) ; Pohl, Klaus (Hrsg.) ; Schmid, Klaus (Hrsg.): *Software-Produktlinien: Methoden, Einführung und Praxis.* Dpunkt Verlag, 2004. – ISBN 3898642577

[Boe76] Boehm, Barry W.: Software Engineering. In: *IEEE Trans. Computers* 25 (1976), Nr. 12, S. 1226–1241

[Boo94] Booch, Grady: *Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen.* Bonn : Addison-Wesley, 1994. – ISBN 978–3–89319–673–9

[Béz05] Bézivin, Jean: On the unification power of models. In: *Software and System Modeling* 4 (2005), Nr. 2, S. 171–188

[CA05] Czarnecki, Krzysztof ; Antkiewicz, Michael: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *GPCE 05*, 2005

[CE00] Czarnecki, Krzysztof ; Eisenecker, Ulrich: *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional, 2000. – ISBN 0201309777

[CHE05] Czarnecki, Krzysztof ; Helsen, Simon ; Eisenecker, Ulrich W.: Formalizing cardinality-based feature models and their specialization. In: *Software Process: Improvement and Practice* 10 (2005), Nr. 1, S. 7–29

[CN01] Clements, Paul ; Northrop, Linda: *Software product lines: practices and patterns*. Bd. 0201703327. Boston, MA, USA : Addison-Wesley, 2001

[CP06] Czarnecki, Krzysztof ; Pietroszek, Krzysztof: Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In: *Proceedings of ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'06)*, 2006

[CSFP04] Collins-Sussman, Ben ; Fitzpatrick, Brian W. ; Pilato, C. M.: *Version Control with Subversion*. Sebastopol, California : O'Reilly & Associates, 2004

[CSFP08] Collins-Sussman, Ben ; Fitzpatrick, Brian W. ; Pilato, C. M.: *Version Control with Subversion*, 2008. – For Subversion 1.5

[CW98] Conradi, Reidar ; Westfechtel, Bernhard: Version models for software configuration management. In: *ACM Computing Surveys* 30 (1998), Nr. 2, S. 232–282. – ISSN 0360–0300

[Cza04] Czarnecki, Krzysztof: Overview of Generative Software Development. In: Banâtre, Jean-Pierre (Hrsg.) ; Fradet, Pascal (Hrsg.) ; Giavitto, Jean-Louis (Hrsg.) ; Michel, Olivier (Hrsg.): *UPP* Bd. 3566, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–27884–2, S. 326–341

[Dar91] Dart, Susan: Concepts in configuration management systems. In: *Proceedings of the 3rd international workshop on Software configuration management*. New York, NY, USA : ACM, 1991. – ISBN 0–897914–429–5, S. 1–18

[Dau08] Daubner, Bernhard: *Konzeption und prototypische Implementierung eines Frameworks zur automatisierten Softwaremessung*, Universität Bayreuth, Diss., 2008

[DeM78] DeMarco, Tom: Structured Analysis and System Specification. In: Broy, Manfred (Hrsg.) ; Denert, Ernst (Hrsg.): *Software Pioneers Contributions to Software Engineering*. Berlin : Springer, 1978. – ISBN 3–540–43081–4, S. 529–560. – zuerst veröffentlicht in: Tom DeMarco, Structured Analysis and System Specification, Yourdon, New York, pp. 1-7 and 37-44, 1978

[Dij72] Dijkstra, Edsger W.: The humble programmer. In: *Commun. ACM* 15 (1972), Nr. 10, S. 859–866. – ISSN 0001–0782

[Dot10] Dotor, Alexander: *Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen*. Bayreuth, Universität Bayreuth, Doktorarbeit, 2010

[EC94] Estublier, Jacky ; Casallas, Rubby: The Adele Configuration Manager. In: Tichy, Walter F. (Hrsg.): *Configuration Management* Bd. 2. New York : John Wiley & Sons, 1994. – ISBN 0–419–4245–6, S. 99–134

[ecl09] eclipse.org ; Eclipse Foundation (Hrsg.): *The Eclipse Modeling Framework (EMF) Overview.* Eclipse Foundation, 2009. – `http://www.eclipse.org/modeling/emf/` – last visited: 04/22/2010

[EF09] Eclipse-Foundation: *Graphical Modeling Framework.* Ottawa, 2009. – http://www.eclipse.org/modeling/gmf/

[FF04] Freeman, Eric ; Freeman, Elisabeth: *Head First Design Patterns.* Beijing, China : O'Reilly, 2004

[FNTZ99] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars ; Zündorf, Albert: Story Diagrams: A new Graph Grammar Language based in the Unified Modeling Language. In: *Proceedings of TAGT '98 - 6th International Workshop on Theory and Application of Graph Transformation* University of Paderborn, 1999 (Technical Report tr-ri-98-201)

[Fow05] Fowler, Martin: *Language Workbenches: The Killer-App for Domain Specific Languages?* `http://martinfowler.com/articles/languageWorkbench.html#ExternalDsl`. Version: 2005

[Fra03] Frankel, David S.: *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing, 2003

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification, Third Edition.* 3. Amsterdam : Addison-Wesley Longman, 2005. – 688 S. – ISBN 0321246780

[GPR06] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA Effektives Software-Engineering mit UML 2.0 und Eclipse.* Springer, 2006

[Gro09] Gronback, Richard C.: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* 1. Boston, MA : Addison-Wesley, 2009 (The Eclipse Series)

[GSR05] Geiger, Leif ; Schneider, Christian ; Reckord, Carsten: Template- and modelbased code generation for MDA-Tools. In: Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proceedings of the 3rd International Fujaba Days*, Universität Paderborn, 2005

[HcW08] Heidenreich, Florian ; Şavga, Ilie ; Wende, Christian: On Controlled Visualisations in Software Product Line Engineering. In: *Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), collocated with the 12th International Software Product Line Conference (SPLC 2008)*, 2008

[Hei09] Heidenreich, Florian: Towards Systematic Ensuring Well-Formedness of Software Product Lines. In: *In Proceedings of the 1st Workshop on Feature-Oriented Software Development*, ACM Press, Oktober 2009

[HJK+09] Heidenreich, Florian ; Johannes, Jendrik ; Karol, Sven ; Seifert, Mirko ; Wende, Christian: Derivation and Refinement of Textual Syntax for Models. In: Paige, Richard F. (Hrsg.) ; Hartman, Alan (Hrsg.) ; Rensink, Arend (Hrsg.): *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)* Bd. 5562, Springer, 2009 (LNCS). – ISBN 978–3–642–02673–7, S. 114–129

[HK05] Hitz, Martin ; Kappel, Gerti: *UML @ Work - Objektorientierte Modellierung mit UML 2.* 3. Auflage. 69115 Heidelberg : Dpunkt Verlag, 2005. – ISBN 3898642615

[HKW08] Heidenreich, Florian ; Kopcsek, Jan ; Wende, Christian: FeatureMapper: Mapping Features to Models. In: *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, ACM Press, Mai 2008. – ISBN 978–1–60558–079–1, S. 943–944

[HW07] Heidenreich, F ; Wende, C: Bridging the Gap Between Features and Models. In: *In Proceedings of the Second Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'07), Salzburg, Austria, October 2007*, 2007

[IEE05] IEEE: IEEE Standard for Software Configuration Management Plans (Revision of IEEE Std 828-1998) / IEEE Computer Society. Version: 2005. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1502775`. 2005. – Forschungsbericht. – 0_1–19 S.

[JCJv92] Jacobson, Ivar ; Christerson, Magnus ; Jonsson, Patrik ; Övergaard, Gunnar: *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992

[JDM09] John D. McGregor, Dirk M. (Hrsg.) ; Software Engineering Institute (Veranst.): *Proceedings of the 13th International Software Product Line Conference.* San Francsisco, CA, USA : Carnegie Mellon University, 2009

[JGJ97] Jacobson, I. ; Griss, M.L. ; Jonsson, P.: *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997

[JSW99] Jäger, Dirk ; Schleicher, Ansgar ; Westfechtel, Bernhard: Using UML for Software Process Modeling. In: Nierstrasz, Oscar (Hrsg.) ; Lemoine, Michel (Hrsg.): *Proceedings of the 7th European Software Engineering Conference (ESEC/FSE 1999)* Bd. 16870. Toulouse, France : Springer-Verlag, September 1999 (Lecture Notes in Computer Science), S. 91–108

[KA08] Kästner, Christian ; Apel, Sven: Type-Checking Software Product Lines - A Formal Approach. In: *ASE*, IEEE, 2008, S. 258–267

[KAK08] Kästner, Christian ; Apel, Sven ; Kuhlemann, Martin: Granularity in software product lines. In: Schäfer, Wilhelm (Hrsg.) ; Dwyer, Matthew B. (Hrsg.) ; Gruhn, Volker (Hrsg.): *ICSE*, ACM, 2008. – ISBN 978–1–60558–079–1, S. 311–320

[Kan09] Kang, Kyo C.: FODA: Twenty Years of Perspective on Feature Models. In: [JDM09]. – Keynote Speech

[KAS10] Kästner, Christian ; Apel, Sven ; Saake, Gunter: Virtuelle Trennung von Belangen (Präprozessor 2.0). In: *Software Engineering 2010 – Fachtagung des GI-Fachbereichs Softwaretechnik*, Gesellschaft für Informatik (GI), February 2010 (Lecture Notes in Informatics P-159). – ISBN 978–3–88579–253–6, 165-176

[KAT+09] Kästner, Christian ; Apel, Sven ; Trujillo, Salvador ; Kuhlemann, Martin ; Batory, Don S.: Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In: Oriol, Manuel (Hrsg.) ; Meyer, Bertrand (Hrsg.): *TOOLS (47)* Bd. 33, Springer, 2009 (Lecture Notes in Business Information Processing). – ISBN 978–3–642–02570–9, S. 175–194

[KC05] Kim, Chang Hwan P. ; Czarnecki, Krzysztof: Synchronizing Cardinality-Based Feature Models and Their Specializations. In: Hartman, Alan (Hrsg.) ; Kreische, David (Hrsg.): *ECMDA-FA* Bd. 3748, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–30026–0, S. 331–348

[KCH+90] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Novak, William E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. 1990. – Forschungsbericht

[KKL+98] Kang, Kyo C. ; Kim, Sajoong ; Lee, Jaejoon ; Kim, Kijoo ; Kim, Gerard J. ; Shin, Euiseob: FORM: A feature-oriented reuse method with domain-specific reference architectures. In: *Annals of Software Engineering* 5 (1998), S. 143–168

[KLD02] Kang, Kyo C. ; Lee, Jaejoon ; Donohoe, Patrick: Feature-Oriented Product Line Engineering. In: *IEEE Software* 19 (2002), Nr. 4, S. 58–65. – ISSN 0740–7459

[KLM+97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina V. ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *ECOOP*, 1997, S. 220–242

[Kru03] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321197704

[Kru06] Krueger, Charles W.: Software Mass Customization / BigLever Software. 2006. – Forschungsbericht

[Kru08] Krueger, Charles W.: The Software Product Line Lifecycle Framework / BigLever Software. Austin, TX, Dezember 2008 (#200805071r1). – Forschungsbericht

[KWB03] Kleppe, Anneke G. ; Warmer, Jos ; Bast, Wim: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 032119442X

[Leb95] Leblang, D.B.: The CM Challenge: Configuration Management that Works. In: Tichy, W.F. (Hrsg.): *Configuration Management*. 1. West Sussex, England : Wiley, 1995 (Trends in Software), Kapitel 1

[Lin02] Linden, Frank van d.: Software Product Families in Europe: The Esaps & Café Projects. In: *IEEE Software* 19 (2002), Nr. 4, S. 41–49

[MB02] Mellor, Stephen J. ; Balcer, Marc: *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA : Addison-Wesley, 2002 `http://www.executableumlbook.com/`. – ISBN 0201748045

[Nag90] Nagl, Manfred: *Softwaretechnik: Methodisches Programmieren im Großen*. Springer Verlag, Springer Compass, 1990

[OMG06a] OMG ; OMG (Hrsg.): *Meta Object Facility (MOF) Core Specification*. OMG, January 2006. – Version 2.0

[OMG06b] OMG ; OMG (Hrsg.): *Object Constraint Language*. OMG, May 2006. – Version 2.0

[OMG07] OMG ; OMG (Hrsg.): *MOF 2.0/XMI Mapping*. OMG, December 2007. – Version 2.1.1

[OMG09a] OMG ; OMG (Hrsg.): *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.2*. OMG, February 2009. – Version 2.2

[OMG09b] OMG ; OMG (Hrsg.): *OMG Unified Modeling Language (OMG UML), Superstructure V2.2*. 140 Kendrick Street, Building A, Suite 300 Needham, MA 02494 USA: OMG, Februar 2009. – Version 2.2

[Opp06] Oppermann, Patrick: *Parsing and Analysing UML Text language in CASE-Tools*, Universität Kassel, Diplomarbeit, 2006

[Par72] Parnas, D. L.: On the Criteria To Be Used in Decomposing Systems into Modules. In: *Communications of the ACM* 15 (1972), S. 1053–1058

[PBL05] Pohl, Klaus ; Böckle, Günter ; Linden, Frank van d.: *Software Product Line Engineering: Foundations, Principles and Techniques.* Berlin, Germany : Springer-Verlag, 2005

[RBP+91] Rumbaugh, James E. ; Blaha, Michael R. ; Premerlani, William J. ; Eddy, Frederick ; Lorensen, William E.: *Object-Oriented Modeling and Design.* Prentice-Hall, 1991. – ISBN 0–13–630054–5

[RJB99] Rumbaugh, James ; Jacobson, Ivar ; Booch, Grady: *The Unified Modeling Language User Guide.* Addison-Wesley, 1999

[SA08] Stephan, Matthew ; Antkiewicz, Michal: Ecore.fmp A tool for editing and instantiating class models as feature models / University of Waterloo. 2008. – Forschungsbericht. – 10 S.

[SB00] Svahnberg, Mikael ; Bosch, Jan: Issues Concerning Variability in Software Product Lines. In: Linden, Frank van d. (Hrsg.): *IW-SAPF* Bd. 1951, Springer, 2000 (Lecture Notes in Computer Science). – ISBN 3–540–41480–0, S. 146–157

[SBPM09] Steinberg, Dave ; Budinsky, Frank ; Paternostro, Marcelo ; Merks, Ed: *EMF Eclipse Modeling Framework.* 2. Boston, MA : Addison-Wesley, 2009 (The Eclipse Series)

[Sch02] Schleicher, Ansgar: *Management of Development Processes: An Evolutionary Approach.* Wiesbaden, Germany : Deutscher Universitäts-Verlag, 2002 (Informatik)

[Sch07] Schneider, Christian: *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten*, Diss., 2007

[SGW94] Selic, Bran ; Gullekson, Garth ; Ward, Paul T.: *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994. – ISBN 0471599174

[SVEH07] Stahl, Tom ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management.* 2. Auflage. dPunkt - Verlag, 2007

[SZN04] Schneider, Christian ; Zündorf, Albert ; Niere, Jörg: CoObRA - a small step for development tools to collaborative environments. In: *Workshop on Directions in Software Engineering Environments (WoDiSEE2004).* Edinburgh, Scotland, 2004, S. 21–28

[The05] The Fujaba Developer Teams from Paderborn, Kassel, Darmstadt, Siegen and Bayreuth: The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen and Bayreuth. In: Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proceedings of the 3rd international Fujaba Days*, 2005, S. 1–13

[Tic82] Tichy, Walter F.: Design, implementation, and evaluation of a Revision Control System. In: *ICSE '82: Proceedings of the 6th international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1982, S. 58–67

[Tic85] Tichy, Walter F.: RCS – A system for version control. In: *Software: Practice and Experience* 15 (1985), Juli, Nr. 7, S. 637–654

[Tic86] Tichy, W. F.: A data model for programming support environments and its application. (1986), S. 219–236. ISBN 0–444–87949–8

[Tol06] Tolvanen, Juha-Pekka: Krieg der Welten - Domänenspezifische Modellierung für vollständige Code-Generierung. In: *Java Spektrum* 1 (2006), S. 9–12

[Van06] Vangheluwe, Hans: *Multi-Paradigm Modeling, and the quest for tool support*. 3rd international Fujaba Days, September 2006. – Keynote speech

[Ves06] Vesperman, Jennifer: *Essential CVS*. Sebastopol, California : O'Reilly & Associates, 2006

[Whi03] White, Brian A.: *Software Configuration Management Strategies and Rational ClearCase*. Reading, Massachusetts : Addison-Wesley, 2003 (Object Technology Series)

[WK03] Warmer, Jos ; Kleppe, Anneke: *The Object Constraint Language: Getting your models ready for MDA*. 2. A. Addison Wesley, 2003. – ISBN 0321179366

[WL99] Weiss, David M. ; Lai, Chi Tau R.: *Software product-line engineering: a family-based software development process*. Boston, MA, USA : Addison-Wesley, 1999. – ISBN 0–201–69438–7

[Wor07] World Wide Web Consortium: *XML Path Language (XPath) 2.0*. http://www.w3.org/TR/xpath20/, 2007

[YC79] Yourdon, Edward ; Constantine, Larry L.: *Structured design*. Englewood Cliffs, NJ : Prentice-Hall, 1979. – ISBN 0138544719

[Zün02] Zündorf, Albert: *Rigorous Object Oriented Software Development*, 2002

[ZSW99] Zündorf, Alber ; Schürr, Andy ; Winter, Andreas J.: Story Driven Modeling / University of Paderborn. 1999 (tr-ri-99-211). – Technical Report