TECHNISCHE UNIVERSITÄT BERLIN

INSTITUT FÜR
SOFTWARETECHNIK UND THEORETISCHE INFORMATIK
FACHGEBIET SOFTWARETECHNIK
PROF. DR.-ING. STEFAN JÄHNICHEN

Diplomarbeit

im Studiengang Informatik mit dem Thema

Abbildung von produktlinienorientierten Featurediagrammen auf aspektorientierte Implementierungsmodule

Marko Feistkorn
(Matrikelnummer: 197581)

Betreut durch
Dipl.-Inform. Christine Hundt und
Dr. rer. nat. Ramin Tavakoli Kolagari

Berlin, 19. Dezember 2007

Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mir bei der Erstellung meiner Diplomarbeit zur Seite standen.

Ein besonderer Dank geht dabei an Dipl.-Inform. Christine Hundt und Dr. rer. nat. Ramin Tavakoli Kolagari. In vielen gemeinsamen Diskussionen konnten wir zahlreiche Probleme lösen und die Diplomarbeit so immer wieder ein Stück vorantreiben. Sie standen mir jederzeit zu Seite und hatten während der vielen Treffen immer wieder die Ausdauer, meine Diplomarbeit erneut zu lesen und mir konstruktive Verbesserungsvorschläge zu geben.

Ich bedanke mich ebenfalls bei Dipl.-Inform. Marco Mosconi, der mir besonders im praktischen Teil meiner Arbeit mit Rat und Tat zur Seite stand. Durch seine konstruktive Hilfe konnte ich zahlreiche größere und kleinere Probleme schnell und effizient lösen.

Ein weiterer Dank geht an meine ehemalige Grundschullehrerin Frau Graumann, die mir während der Diplomarbeit mit zahlreichen stilistischen Verbesserungsvorschlägen zur Seite stand.

Nicht zuletzt geht ein Dank an meine Eltern, die mir das Studium überhaupt erst ermöglicht haben.

Abschließend möchte ich mich bei meiner Ehefrau Aileen Feistkorn bedanken, die mich mit viel Geduld, Verständnis und Verbesserungsvorschlägen durch meine Diplomarbeit begleitet hat.

Marko Feistkorn, 19.12.2007

Kurzfassung

In der heutigen Zeit kommen komplexe Softwaresysteme in nahezu jedem Umfeld zum Einsatz. Softwaresysteme zur Steuerung und Verwaltung von Prozessen sind sowohl im privaten als auch im gewerblichen Bereich unverzichtbar geworden. Mit dem Fortschritt der technologischen Möglichkeiten erhöht sich auch die Komplexität von Softwaresystemen. Soll ein Produkt in einem heterogenen Umfeld gezielte Anwendungsbereiche abdecken, ist eine konfigurierbare Software unabdingbar. Gerade in einem heterogenen Umfeld entstehen enorme Kosten in der Softwareproduktion, wenn nicht bereits zu Beginn des Modellierungs- und Entwicklungsprozesses entsprechende Vorkehrungen getroffen werden. Auch die Wiederverwendbarkeit einzelner Komponenten kann große Ersparnisse in der Softwareproduktion erzielen. Es ist somit leicht nachvollziehbar, dass die moderne Softwareentwicklung einen noch größeren Stellenwert auf Variabilität und Wiederverwendbarkeit legen muss, um langfristig Kosten zu reduzieren und Entwicklungsprozesse zu beschleunigen. Ein viel versprechender Ansatz für diese Konzepte sind die *Software-Produktlinien*. Eine Software-Produktlinie konzentriert sich dabei nicht mehr nur auf ein einzelnes individuelles Produkt, sondern vielmehr auf ein System von Artefakten, aus denen eine Menge von individuellen Produkten zusammengesetzt werden kann. Eine Produktlinie beschreibt somit Gemeinsamkeiten ebenso wie Unterschiede zwischen den einzelnen Produkten. An dieser Stelle kann man von Eigenschaften sprechen, die verschiedene Produkte miteinander teilen, aber auch voneinander unterscheiden. Die Darstellung dieser Eigenschaften erfolgt in Form eines Feature-Modells. Eigenschaften werden im Rahmen von Feature-Modellen als *Features* bezeichnet. Diese Features werden innerhalb eines Feature-Modells in einer Baumstruktur hierarchisch angeordnet und mit Beziehungen untereinander versehen. Ganze Produktlinien können somit in Form von Feature-Modellen effizient dargestellt werden.
Ziel dieser Diplomarbeit ist es, ein Verfahren zu entwickeln, mittels dem Feature-Modelle automatisiert in Codegerüste beziehungsweise Designmodelle transformiert werden können. Dadurch werden alle Variabilitäten auf das Designmodell und somit die Implementierungsebene übernommen. Das Ziel der Transformation ist hierbei Object Teams. Bei Object Teams handelt es sich um eine aspektorientier-

te kollaborationsbasierte Sprache. Diese eignet sich besonders gut, um die aus dem Feature-Modell hervor gehenden Variabilitäten abzubilden. Neben der Entwicklung der benötigten Transformationsregeln wurde auch eine Fallstudie durchgeführt, anhand derer die Inhalte der Diplomarbeit gut veranschaulicht werden. Außerdem wurde ein Eclipse-Plugin implementiert, mit dem die in dieser Diplomarbeit erarbeiteten Transformationsregeln praktisch umgesetzt werden konnten. Anhand der Fallstudie und des implementierten Plugins konnte gezeigt werden, dass die Transformation von Feature-Modellen nach Object Teams sinnvolle Ergebnisse liefert und durchaus Potential für den Praxiseinsatz besitzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Einleitung und Motivation

In der heutigen Zeit kommen komplexe Softwaresysteme in nahezu jedem Umfeld zum Einsatz. Softwaresysteme zur Steuerung und Verwaltung von Prozessen sind sowohl im privaten als auch im gewerblichen Bereich unverzichtbar geworden. Mit dem Fortschritt der technologischen Möglichkeiten erhöht sich auch die Komplexität von Softwaresystemen. Soll ein Produkt in einem heterogenen Umfeld gezielte Anwendungsbereiche abdecken, ist eine konfigurierbare Software unabdingbar. Gerade in einem heterogenen Umfeld entstehen enorme Kosten in der Softwareproduktion, wenn nicht bereits zu Beginn des Modellierungs- und Entwicklungsprozesses entsprechende Vorkehrungen getroffen werden. Auch die Wiederverwendbarkeit einzelner Komponenten kann große Ersparnisse in der Softwareproduktion erzielen. Es ist somit leicht nachvollziehbar, dass die moderne Softwareentwicklung einen noch größeren Stellenwert auf Variabilität und Wiederverwendbarkeit legen muss, um langfristig Kosten zu reduzieren und Entwicklungsprozesse zu beschleunigen. So profitiert man durch die Variabilität von Software vor allem in heterogenen Einsatzgebieten, wohingegen sich die Wiederverwendbarkeit besonders bei standardisierten Komponenten positiv bemerkbar macht. Auch für die sich der Softwareentwicklung anschließenden Prozesse können durch die Variabilität und Wiederverwendung erhebliche Vorteile erzielt werden.

Ein viel versprechender Ansatz, der die genannten Konzepte aufgreift, sind die *Software-Produktlinien*. Die produktlinienorientierte Softwareentwicklung macht sich die oben genannten Probleme zur Aufgabenstellung und versucht, diese mittels der definierten Ansätze der Wiederverwendbarkeit und Variabilität zu lösen und somit den Softwareentwicklungsprozess effizienter zu gestalten. An dieser Stelle muss klar zwischen den in der Industrie bereits etablierten mechanischen Produktlinien und den Software-Produktlinien unterschieden werden. Das grundlegende

Konzept der Produktlinien ist für beide Arten zwar identisch, jedoch unterscheidet sich der Anwendungskontext. Gerade in der Automobilbranche zählen die Produktlinien in ihrer bestehenden Form bereits zu einem Standardhandwerkszeug. Um diese Grundlagen auch auf die Softwareentwicklung anwenden zu können, muss das bestehende Konzept der Produktlinien jedoch in einigen Bereichen angepasst und erweitert werden.
Bei Produktlinien wird sich nicht mehr nur auf ein einzelnes individuelles Produkt konzentriert, sondern vielmehr auf ein System von Artefakten, aus denen im Zusammenhang mit einer allgemeinen Basis (Architektur) eine Menge von individuellen Produkten abgeleitet werden kann. Eine Produktlinie beschreibt somit Gemeinsamkeiten ebenso wie Unterschiede zwischen den einzelnen Produkten. Ziel ist es, bestimmte Komponenten immer wieder zu verwenden und trotzdem individualisierte Produkte durch die vorhandene Variabilität zu schaffen. Man kann an dieser Stelle auch von Eigenschaften sprechen, die verschiedene Produkte miteinander teilen, aber auch voneinander unterscheiden können. Solche Eigenschaften werden als *Features* bezeichnet.
Für die Modellierung der Produktlinien sollen im Rahmen dieser Diplomarbeit Feature-Modelle zum Einsatz kommen. Hauptziel dieser Arbeit ist es, ein Verfahren zu entwickeln, mittels dem Software-Produktlinien, die durch Feature-Modelle repräsentiert werden, automatisiert in konkrete Codegerüste beziehungsweise Designmodelle zu transformieren.
Feature-Modelle werden erstmals im Rahmen von FODA (Feature Oriented Domain Analysis)[KCH$^+$90] vorgestellt. *FODA* bildet die Grundlage für die Feature-Modellierung und wurde bereits durch verschiedene Ansätze erweitert, zum Beispiel in den Veröffentlichungen [CUE04] und [KKL$^+$98]. Die *Feature-Modelle* nach FODA stellen die Features in Form eines Baumes dar, wobei die Wurzel das eigentliche Produkt und die einzelnen Knoten und Blätter des Baumes die einzelnen Features darstellen. Mittels weiterführender Notation lassen sich innerhalb des Feature-Baumes Abhängigkeiten zwischen Features und Kardinalitäten von einzelnen Features und Feature-Gruppen darstellen.

Da es mit den Bordmitteln von objektorientierten Sprachen oftmals nicht ohne weiteres möglich ist, Features auf Modulebene abzubilden und somit eine Lücke zwischen der Modellierungs- und Implementierungsebene entsteht und des Weiteren die automatisierte Transformation von eben dieser modellierten Features in Designmodelle beziehungsweise Codegerüste noch weitestgehend ungeklärt ist, werden in dieser Arbeit die sich durch die aspektorientierte Programmierung ergebenden Möglichkeiten geprüft und angewendet. Die *aspektorientierte Programmierung* ist ein relativ neues Programmierparadigma, das es ermöglicht, Software auch auf Implementierungsebene erheblich besser zu strukturieren. Ein Hauptproblem der etablierten Programmierparadigmen sind die *crosscutting concerns*.

Hiermit sind Anforderungen gemeint, die miteinander „verwoben“ sind und somit unter Umständen quer durch alle logischen Schichten eines Systems schneiden. Dieses wesentliche Problem kann mit den Strukturierungsmöglichkeiten der aspektorientierten Programmierung bequem umgangen werden, was in Hinblick auf die Software-Modellierung erhebliche Vorteile bringt. Eine wesentliche Säule der aspektorientierten Programmierung ist deshalb die mögliche Aufgliederung der sich schneidenden Anforderungen, dieser Grundsatz ist auch bekannt unter *seperation of crosscutting concerns*. Vor allem in Bezug auf die Wiederverwendbarkeit einzelner Module können hierdurch große Fortschritte im Bereich der Softwareentwicklung erzielt werden.

Um die Produktentwicklung beispielhaft darzustellen, wird der Modellierungsprozess mittels einer Fallstudie verdeutlicht. Bei der Fallstudie handelt es sich um eine Spiele-Produktlinie, die veranschaulicht, wie durch Variabilität und Wiederverwendbarkeit innerhalb einer Software-Produktlinie, basierend auf Grundfunktionalitäten, verschiedene Spieltypen erstellt werden können. Für die Umsetzung der Software-Produktlinie soll die aspektorientierte und kollaborationsbasierte Programmiersprache Object Teams zum Einsatz kommen.

Object Teams wird seit 2001 an der Technischen Universität Berlin entwickelt. Innerhalb von *Object Teams* werden neben den bereits bekannten Klassen zwei neue Arten von Modulen eingeführt: Teams und Rollen. Bei *Teams* und *Rollen* handelt es sich um eine besondere Art von Klassen, wobei ein Team zusammengehörige Rollen gruppiert. Zusätzlich zur Mitgliedschaft einer Rolle in einem Team kann es existierende Klassen erweitern beziehungsweise adaptieren. Diese Relation wird durch die *playedBy*-Beziehung ausgedrückt. Dadurch wird die Möglichkeit geschaffen, das Verhalten bestehender Klassen durch eine Rolle zu verändern. Aufgrund dieser Eigenschaften und der Vererbungsmöglichkeiten auf Teamebene, entstehen Vorteile in Bezug auf Flexibilität, Kapselung und Spezialisierung. So können durch die Verwendung von Klassen, Rollen und Teams die *core-level-concerns* (diese Bestandteile gehören zum logischen „Kern“ der Anwendung) klar von den *system-level-concerns* (hierbei handelt es sich um Anforderungen, die das gesamte System betreffen) getrennt werden. Außerdem können system-level-concerns durch Teams und Rollen sehr gut modularisiert werden, was innerhalb der Objektorientierung bisher nicht ohne Weiteres möglich war. Darüber hinaus wird durch das Teamkonzept ein weiterer wesentlicher Vorteil geschaffen: die zum Teil unabhängige Aktivierung einzelner Teams und somit einzelner Funktionalitäten. Werden also einzelne Features in Form von Teams abgebildet, so können diese zu einem späteren Zeitpunkt beliebig aktiviert beziehungsweise deaktiviert werden.

1.2 Zielsetzung der Diplomarbeit

Ziel dieser Arbeit ist es, die Transformation von erweiterten Feature-Modellen in Object Teams zu untersuchen. Die Feature-Modelle sind dabei als Ergebnis der Domänen-Analyse, die im Rahmen des Domain-Engineerings durchgeführt wird, zu betrachten. Hierbei soll weder die Darstellbarkeit von Software-Produktlinien mittels Feature-Diagrammen noch die Erstellung eines Feature-Modells auf Basis einer Domänen-Analyse untersucht, sondern vielmehr ein automatisiertes Verfahren entwickelt werden, welches es ermöglicht, Software-Produktlinien, die durch Feature-Diagramme dargestellt werden, nach Object Teams zu transformieren. Die Transformation soll hierbei den Kern der Arbeit darstellen. Mit dem Prozess der Transformation ist ein Verfahren gemeint, das es ermöglicht, aus Feature-Diagrammen Programmgerüste, basierend auf Teams[1], zu erstellen, welche die Struktur der Produktlinien darstellen und in einem darauf folgenden — nicht zu betrachtenden Schritt — nur noch mit der notwendigen Programmlogik befüllt werden muss. Hierbei sollen die gültigen Grundkonzepte des *Model Driven Engineerings (MDE)* berücksichtigt und angewendet werden. Eine automatisierte Transformation von

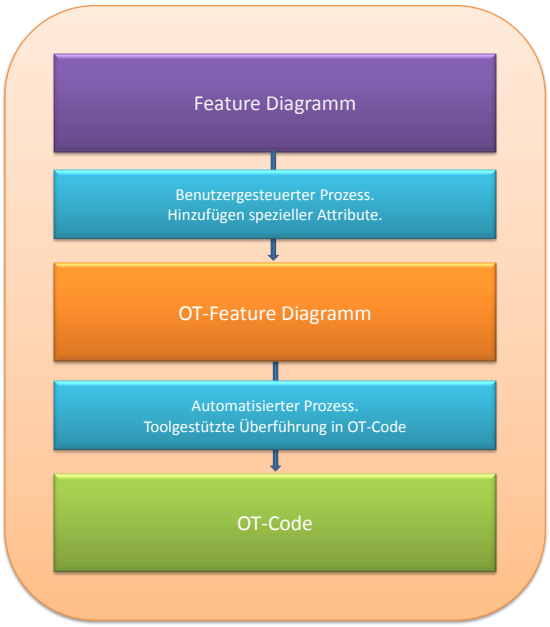

Abbildung 1.1: Schematische Darstellung der einzelnen Übersetzungsschritte

einfachen Feature-Diagrammen nach Object Teams ist nicht ohne weiteres mög-

[1] Teams sind eine Art erweitertes Klassenkonstrukt der Sprache ObjectTeams

lich. Aus diesem Grund soll im Rahmen dieser Arbeit eine weiterführende Notation entwickelt werden, die es ermöglicht, Features und deren Abhängigkeiten innerhalb eines Feature-Diagramms präzise zu definieren und durch zusätzliche Attribute die darauf folgende Transformation mit notwendigen zusätzlichen Informationen zu versehen. Neben der formalen Definition der Transformation soll ein Plugin entwickelt werden, welches die Transformation prototypisch umsetzt. In Abbildung 1.1 ist eine Übersicht der einzelnen Prozessschritte dargestellt. Ausgehend von einem Feature-Diagramm wird im Rahmen eines benutzergesteuerten Prozess ein erweitertes Feature-Diagramm erstellt. Aus diesem Feature-Diagramm kann dann der eigentliche Object Teams-Code erzeugt werden.

1.3 Gliederung der Arbeit

Zum besseren Verständnis wird nachfolgend eine kurze Übersicht der einzelnen Kapitel gegeben. In Kapitel 2 werden die in dieser Arbeit verwendeten Begriffe und Ansätze vorgestellt. Dazu gehören im Wesentlichen Software-Produktlinien, Feature-Modelle, die aspektorientierte Programmierung und die Konzepte des Model Driven Engineerings. Nachdem die grundlegenden Begriffe vorgestellt wurden, wird dann in Kapitel 3 die im Rahmen der Diplomarbeit angefertigte Fallstudie vorgestellt. Anschließend werden in Kapitel 4 die für die Abbildung von Feature-Modellen auf Object Teams notwendigen Erweiterung dargelegt und genauer beleuchtet. Kapitel 5 widmet sich dann der eigentlichen Transformation und stellt diese ausführlich vor. Im darauf folgenden Kapitel 6 werden die in Kapitel 4 und Kapitel 5 dargelegten Transformationsregeln auf die in Kapitel 3 vorgestellte Fallstudie angewendet. Im Anschluß folgt das Kapitel 7, in dem die Ergebnisse der Implementierung der Transformationsregeln vorgestellt werden sollen. Abschließend wird in Kapitel 8 eine Beurteilung des Ansatzes und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

Kapitel 2

Vorstellung der Begrifflichkeiten

In diesem Kapitel werden die Begrifflichkeiten vorgestellt, die im Rahmen dieser Arbeit Anwendung finden. Dafür werden zunächst die Software-Produktlinien vorgestellt. Im Anschluß daran wird eine Einführung in die Feature-Modellierung gegeben. Zum Ende des Kapitels wird neben der Beschreibung des aspektorientierten Programmierparadigmas auch noch die kollaborationsbasierte und aspektorientierte Sprache Object Teams vorgestellt, bevor im letzten Abschnitt des Kapitels auf das *Model Driven Engineering (MDE)* eingegangen wird, das im Rahmen dieser Arbeit dazu dient, das bestehende Feature-Modell anhand von definierten Regeln in ein Object Teams-Modell zu transformieren.

2.1 Software-Produktlinien

Eine Produktlinie setzt sich aus einer Reihe verschiedener Produkte zusammen, die eine Menge von Eigenschaften miteinander teilen und dadurch auf ein bestimmtes Marktsegment abzielen. Produktlinien sind allgegenwärtig, angefangen in der Automobilindustrie bis hin zu Gastronomiebetrieben begegnen uns ständig Produktlinien. Gerade in der Automobilindustrie können durch die Verwendung von Produktlinien enorme Kostenersparnisse erzielt werden. Man muss sich an dieser Stelle nur vorstellen, welche Kosten entstünden, wenn für jede Ausstattungskonfiguration, wie zum Beispiel Klimaanlage, Airbags, beheizbare Sitze, eines Mittelklassefahrzeugs ein komplett eigenständiger Autotyp entwickelt werden müsste.
Boeing zum Beispiel entwickelte die Flugzeuge der Typen 767 und 757 gleichzeitig. Obwohl es sich bei diesen beiden Modellen um komplett unterschiedliche Flugzeuge handelt, stimmen die Materiallisten zu 60% überein. Selbst bei normalen Tageszeitungen trifft man auf Produktlinien. So gibt es jeden Tag verschiedene Versionen einer Zeitung, die sich zum Beispiel regional nur geringfügig unterscheiden. So haben diese verschiedenen Versionen oftmals die gleichen Artikel, Bilder und

Rätsel und unterscheiden sich beispielsweise nur in gewissen regional abhängigen Artikeln. Würde man hierfür unabhängige Produkte definieren, würden sich der Aufwand und die daraus resultierenden Kosten entsprechend der Versionen multiplizieren. Sogar beim abendlichen Restaurantbesuch in seiner Lieblingspizzeria trifft man auf Produktlinien. Man wird feststellen, dass schon allein der Pizzaboden und die Grundzutaten, wie zum Beispiel Soße und Käse, bei nahezu allen Pizzavarianten übereinstimmen. Die soeben vorgestellten Produktlinien erscheinen uns als absolut selbstverständlich, lassen sich aber nicht ohne weiteres auf die Softwareentwicklung übertragen, da diese Art der Produktlinien eine andere technologische Herangehensweise erfordert. Verschiedene ökonomische Analysen für Softwareproduktivität haben jedoch ergeben, dass in der Wiederverwendung von Software eine der größten Möglichkeiten liegt, die Softwareproduktivität und -qualität zu verbessern.

2.1.1 Definition Software-Produktlinien

Eine Software-Produktlinie besteht aus einer Reihe von Softwaresystemen, die eine bestimmte Menge von Eigenschaften teilen, welche die Anforderungen einer bestimmten Domäne abdecken und auf Basis von gleichen Grundkomponenten entwickelt wurden (nach [CN01]).

> *„A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core software assets in a prescribed way.“[CN01]*

Bei Produktlinien wird sich nicht mehr nur auf ein einzelnes individuelles Produkt konzentriert, sondern vielmehr auf ein System von so genannten *Artefakten*, aus denen im Zusammenhang mit einer allgemeinen Basis eine Menge von individuellen Produkten abgeleitet werden kann. Die Software-Produktlinien-Entwicklung unterteilt sich dabei in drei Hauptprozesse:

- *Domain Engineering*[1] — Dieses Verfahren vereinigt die Analyse der Anwendungsdomäne und die Definition der *Artefakte*, die auch die eigentliche Produktlinien-Architektur beinhalten.
- *Application Engineering*[2] — In diesem Prozess wird aus der allgemeinen Produktlinien-Architektur das konkrete Produkt abgeleitet und erstellt.
- *Management* — Der Management-Prozess greift in die beiden anderen Hauptprozesse ein und koordiniert deren Zusammenspiel.

[1] In der Literatur auch als *Core Asset Development* bekannt
[2] In der Literatur auch als *Product Development* bekannt

Die beiden Hauptprozesse *Domain Engineering* und *Application Engineering* unterteilen sich wiederum in verschiedene Teilbereiche, die untereinander interagieren. Abbildung 2.1 enthält eine Darstellung der beiden genannten Hauptprozesse. Man sieht in dieser Abbildung sowohl die Trennung als auch die Interaktion der beiden Prozesse. Eine genauere Beschreibung der Prozesse erfolgt in den Abschnitten 2.1.4 und 2.1.5.

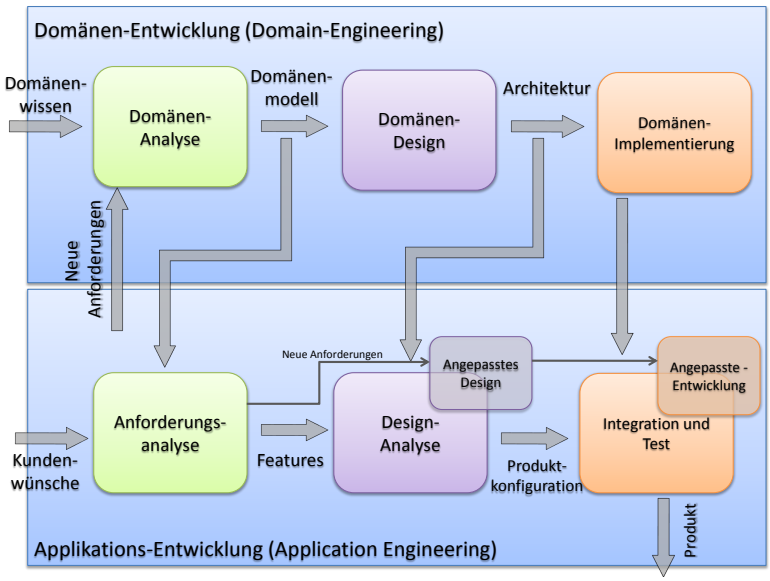

Abbildung 2.1: Hauptprozesse der Produktlinien-Entwicklung nach [CE00]

Artefakte — Core Assets Der Begriff *Artefakt* wird in der englischen Literatur teilweise auch als *core asset* bezeichnet. Im Rahmen dieser Arbeit wird jedoch ausschließlich der Begriff *Artefakt* stellvertretend für den Begriff *core asset* verwendet. Artefakte sind die Basis für die Produkterstellung innerhalb einer Produktlinie. Zu den Artefakten gehören Dinge wie Architektur, wiederverwendbare Softwarekomponenten, Domänen-Modelle, Anforderungen, Dokumentationen, Testfälle, Qualitätssicherungsmechanismen und so weiter. In den Artefakten spiegeln sich sowohl die Gemeinsamkeiten als auch die Unterschiede einzelner Produkte innerhalb einer Produktlinie wider. Hierbei können die Artefakte auf zwei unterschiedliche Arten konstruiert sein:

- Es gibt Artefakte, die nur Gemeinsamkeiten enthalten. Für die eigentlichen Variabilitäten existieren gesonderte Artefakte.
- Gemeinsamkeiten und Variabilitäten werden innerhalb eines Artefakts gehal-

ten. In diesem Fall müssen die Artefakte entsprechend parameterisiert werden.

Der große Vorteil in den einmalig definierten Artefakten besteht darin, dass diese nicht für jedes Produkt neu entwickelt werden müssen, sondern — sobald sie einmal umgesetzt wurden und die notwendigen Qualitätssicherungsprozesse durchlaufen haben — in einer qualitativ hochwertigen Form vorliegen und eingesetzt werden können. Die einzelnen Produkte der Produktlinie können dann durch das Zusammenfügen der Artefakte abgeleitet werden. Dieser Prozess wird als das Application Engineering bezeichnet. Auch diesem Prozess liegt eine sorgfältige Planung zu Grunde, da die für eine Produkt-Instanzierung notwendigen Modifikationen einzelner Komponenten bereits im Domain Engineering in Form von Variabilitäten definiert wurden. Das eigentliche Application Engineering sollte im Idealfall keine größeren Implementierungsarbeiten erfordern, da es sich in diesem Schritt nur um die Konfiguration und Kombination der bestehenden Artefakte handelt.
Es ist somit klar ersichtlich, dass eine umfassende und genaue Analyse der Anwendungsdomäne unabdingbar ist. Erst wenn genau evaluiert wurde, welche Anforderungen durch die einzelnen Produkte an eine Produktlinie gestellt werden, können die Artefakte, welche die Gemeinsamkeiten und Variabilitäten zwischen den Produkten einschließen, sinnvoll modelliert werden. Die Analyse- und Designphase einer Produktlinie stellt deshalb einen zentralen Bestandteil der Produktlinien-Entwicklung dar und ist im Gegensatz zur normalen Produktentwicklung weitaus komplexer. Produktlinien erfordern ein strategisches Denken, das über die Grenzen eines einzelnen Produkts hinausgeht. Neben dem technischen Verständnis erfordert die Produktlinien-Entwicklung auch ein ausgeprägtes Verständnis für die zugrunde liegenden Geschäftsprozesse.

Um Software-Produktlinien klar von den bisherigen Verfahren der Softwareentwicklung abzugrenzen, soll nachfolgend in Kurzform dargestellt werden, was Software-Produktlinien von anderen Ansätzen unterscheidet.

2.1.2 Abgrenzung gegenüber anderen Ansätzen nach [CN01]

Zufällige Wiederverwendung fein-granularer Softwarekomponenten

Die allgemeine Wiederverwendung von Software-Bestandteilen in der Softwareentwicklung ist kein neues Prinzip, jedoch handelt es sich bei dieser so genannten fein-granularen Wiederverwendung eher um Bibliotheken mit diversen Algorithmen und Hilfsmitteln, die im Laufe verschiedener Projekte immer weiter gewachsen sind. In darauf folgenden Projekten werden aus diesen, meist ungeordneten Bibliotheken, nur minimale Bestandteile wiederverwendet, die zudem noch auf vorhergegangene Projekte spezialisiert waren und zusätzliche Probleme bei der

Integration in das aktuelle Projekt mit sich bringen. Man kann an dieser Stelle also von Zufall sprechen, wenn eines der Bestandteile einer existierenden Bibliothek genau auf die Anforderung eines aktuellen Projekts passt. In einem Software-Produktlinien-Ansatz ist die Wiederverwendung von vornherein geplant und die Komponenten sind entsprechend darauf ausgerichtet. Die Artefakte beinhalten unter anderem die Bestandteile, die bei einer permanenten Neuentwicklung sehr kostenintensiv wären.

Produktentwicklung mit Wiederverwendung

Dieses Verfahren wird oftmals angewendet, wenn ein Produkt entwickelt werden soll, was einem anderen bereits bestehenden Produkt sehr ähnelt. Die wiederverwendbaren Bestandteile des bestehenden Produkts werden geklont und im neuen Produkt nahezu unverändert eingesetzt. An dieser Stelle können Kosten eingespart werden, da keine komplette Neuentwicklung stattfinden muss. Jedoch müssen von nun an zwei ähnliche Produkte getrennt voneinander gewartet werden, was sich in den verursachten Kosten stark niederschlägt. Bei Software-Produktlinien hingegen, werden die wiederverwendeten Komponenten von vornherein als solche konzipiert. Weiterhin wird eine Software-Produktlinie als Ganzes aufgefasst und nicht als eine Ansammlung von verschiedenen Produkten, die getrennt voneinander gewartet werden müssen. Die eigentlichen Produkte bestehen nur aus den einzelnen Komponenten der Produktlinie — aus diesem Grund zählen auch nicht die unterschiedlichen Produkte, sondern die sorgfältig entwickelten Artefakte zum wichtigsten Gut der Organisation.

Komponentenbasierte Entwicklung

Die komponentenbasierte Entwicklung steht der produktlinienorientierten Entwicklung relativ nahe, unterscheidet sich jedoch in verschieden Punkten. Bei der komponentenbasierten Entwicklung werden die einzelnen Komponenten aus einer bestehenden Bibliothek zusammengesetzt. Dies ist dem produktlinienorientierten Ansatz sehr ähnlich, der nur den Zusatz hat, dass innerhalb einer Produktlinie alle selektierbaren Komponenten durch die Produktlinien-Architektur definiert sind. Bei der produktlinienorientierten Entwicklung werden die Variabilitäten im Gegensatz zum komponentenbasierten Ansatz von vornherein in der Architektur berücksichtigt. Es existiert also eine generische Basis, die über das Auflösen von Variabilitäten zu einem speziellen Produkt umgesetzt wird. Beim komponentenbasierten Ansatz wird die Spezialisierung oftmals durch zusätzlichen Code erzielt, wodurch die daraus entstehenden spezialisierten Komponenten wieder getrennt voneinander gewartet werden müssen.

Konfigurierbare Architektur

Allgemeine Architekturen und bestehende Frameworks sind für die Verwendung in unterschiedlichen Kontexten konzipiert und können dementsprechend bis zu einem gewissen Grad frei konfiguriert werden. Allein durch diesen Ansatz können Kosten stark reduziert werden, da hierdurch die grundlegenden Strukturen eines jeden Systems abgedeckt werden. Die Architektur einer Produktlinie hingegen ist dafür konzipiert, die Abbildung der Variationen innerhalb einer Produktlinie zu unterstützen. In diesem Zusammenhang muss klar sein, dass die Architektur zwar eine wichtige Komponente ist, sich jedoch in die Sammlung der Artefakte einer Produktlinie eingliedert.

Releases und Versionen eines Produkts

Jedes Produkt unterliegt einer Evolution, innerhalb derer es zu verschiedenen Versionen und Releases kommt. Jedes dieser Versionen und Releases wird basierend auf den Bestandteilen der vorhergehenden Releases und Versionen erzeugt. Dies ist nicht mit Produktlinien gleichzusetzen. Bei Produktlinien handelt es sich zu jedem Zeitpunkt um unterschiedlich simultane Produkte, welche ihre eigenen Versionen und Releases besitzen. Die Versionen eines Produktes sind aufeinander folgend, wobei sich das eigentliche Produkt durch seine aktuelle Version charakterisiert.

2.1.3 Vorteile durch die Verwendung von Produktlinien

Nachdem im vorangegangenen Abschnitt die Produktlinien vorgestellt und gegenüber anderen Ansätzen abgegrenzt wurden, soll nun noch einmal kurz auf die aus der Anwendung von Produktlinien resultierenden Vorteile eingegangen werden.
Der wesentliche Vorteil der Produktlinien liegt in der Wiederverwendung der entwickelten Komponenten. Dadurch können Systeme schnell, effizient in Bezug auf die Kosten und mit hoher Qualität entwickelt werden. Die Qualität erhöht sich vor allem dadurch, dass alle nachfolgenden Systeme von der erzielten Qualität der vorhergehenden Systeme profitieren können, da alle Systeme auf Grundlage der gleichen Artefakte erstellt werden. Die Zeitspanne von der Beauftragung bis zur Marktreife (*time-to-market*) kann drastisch verkürzt werden. Auch die Kosten für die Wartung können durch die Verwendung von einheitlichen Komponenten stark reduziert werden. Durch die schnelle Auslieferung, die hohe Qualität und die geringen Wartungskosten wird nicht zuletzt auch die Kundenzufriedenheit stark erhöht.

2.1.4 Domain Engineering

Zu einem der beiden Hauptprozesse der Produktlinien-Entwicklung gehört das Domain Engineering (siehe Abbildung 2.1). Innerhalb dieses Prozesses wird die An-

wendungsdomäne der Produktlinie analysiert und eingegrenzt. Das Domain Engineering wurde entwickelt, um Systeme, die sich sowohl in Anforderungen als auch in ihrem Anwendungsgebiet überschneiden, zu vereinheitlichen und somit die entstehenden Synergie-Effekte nutzbar zu machen. Dieser Ansatz soll wiederverwendbare Bestandteile innerhalb einer Domäne sammeln, organisieren und für andere Systeme der gleichen Domäne nutzbar machen. Die Hauptmotivation für dieses Vorgehen liegt klar auf der Hand. Es sollen Softwaresysteme kostengünstig, schnell und mit hoher Qualität entwickelt werden. Das Domain-Engineering ist hierbei ein systematischer Ansatz, der sich in die drei Hauptbereiche *Domänen-Analyse (Domain Analysis)*, *Domänen-Design (Domain Design)* und *Domänen-Implementierung (Domain Implementation)* (siehe Abbildung 2.1) unterteilt.

Domänen-Analyse

In dieser Phase wird die Domäne zunächst klar eingegrenzt. Hierfür werden bereits bestehende Systeme beziehungsweise vorhandene Materialien evaluiert. Alle Systeme, die in diese Domäne fallen, werden auf ihre Anforderungen überprüft, um daraus einen Katalog mit den Gemeinsamkeiten und Unterschieden zu erstellen. Jedoch wird in dieser Phase nicht nur der *Ist-Zustand* analysiert, vielmehr sollen auch die zukünftigen Anforderungen ermittelt werden. Aus dieser Analyse wird das Domänen-Modell abgeleitet. Dieses Modell beinhaltet die gemeinsamen und variablen Eigenschaften der Systeme innerhalb der Domäne sowie Abhängigkeiten zwischen den variablen Eigenschaften.

Domänen-Design und Domänen-Implementierung

Das Ziel des Domänen-Designs und der Domänen-Implementierung ist es, eine allgemeine Architektur für alle Systeme die innerhalb der Domäne liegen zu entwickeln. Um die Architektur strukturiert darstellen zu können, werden häufig verschiedene Modelle benötigt, da somit unterschiedliche Ansichten auf die Architektur entstehen. Die eigentliche allgemeine Struktur soll dabei Komponenten und Subsysteme der zu implementierenden Softwaresysteme widerspiegeln. Neben den enthaltenen Komponenten muss die Architektur auch die Abhängigkeiten und Interaktionen zwischen den einzelnen Komponenten berücksichtigen. Die Architektur soll trotz dieser Vorgaben so flexibel wie möglich bleiben. Nach [Sof96] gibt es hierfür zwei verschiedene Ansätze, die Architektur zu konzipieren:

- *Generische Architekturen* — Eine generische Architektur bildet einen festen Rahmen mit verschiedenen Schnittstellen, an die alternative oder erweiternde Komponenten angedockt werden können. Hierbei ist es wichtig, dass die jeweiligen Schnittstellen über klare *Interfaces* verfügen, damit für die Komponenten klar definiert ist, welche Daten erwartet und zur Verfügung gestellt

werden. Die Topologie ist bei dieser Architektur fest vorgegeben und nicht änderbar.

- *Hochgradig flexible Architekturen* — Bei flexiblen Architekturen ist es möglich, deren Topologie zu verändern und gemäß vorhandener Struktur-Vorgaben anzupassen. Die generischen Architekturen bilden somit eine Teilmenge der hochgradig flexiblen Architekturen. Dies bedeutet, dass selbst der Rahmen einer solchen Architektur konfigurierbar ist und somit verändert werden kann. Dadurch besteht die Möglichkeit, Schnittstellen beliebig anzulegen und zu konfigurieren.

Es ist von hoher Wichtigkeit, dass die Architektur für eine Domäne die enthaltene Variabilität explizit darstellen kann. Hierfür können zum Beispiel Konfigurationssprachen verwendet werden, die es ermöglichen, die konfigurierbaren Komponenten auf eine einheitliche Art und Weise zu behandeln. Ein weiterer wichtiger Bestandteil des Domänen-Designs ist der *Produktionsplan (Production Plan)*. Durch diesen Plan wird beschrieben, wie aus der allgemeinen Architektur und der zugehörigen Komponenten, das konkrete Produkt erzeugt werden kann. Diese Informationen werden wiederum innerhalb des Application Engineerings benötigt. Hierzu gehört auch die kundenbezogene Anpassung der Systeme sowie der Prozess für die Umsetzung von allgemeinen *Änderungsanforderungen (Change Requests)*. Aus diesen drei Schritten wird der Unterschied zur normalen Systementwicklung klar. Während sich bei der konventionellen Softwareentwicklung auf die Anforderungen eines Systems konzentriert wird, wird beim Domain Engineering von vornherein eine Familie von ähnlichen Systemem betrachtet, aus der dann die einzelnen konkreten Systeme erzeugt werden können.

2.1.5 Application Engineering

Application Engineering (siehe Abbildung 2.1) bezeichnet den Prozess, der aus Kundenanforderungen und den Ergebnissen des Domain Engineerings die eigentlichen Applikationen ableitet. Hierfür werden in der Anforderungsanalyse die Kundenwünsche aufgenommen und mit dem Domänen-Modell abgeglichen, so dass im Anschluß die passenden Artefakte selektiert werden können. Sofern es Anforderungen seitens des Kunden gibt, die nicht durch eins der bestehenden Artefakte abgedeckt beziehungsweise nicht verallgemeinert werden können, müssen diese kundenspezifisch implementiert werden. Im letzten Schritt des Application Engineering werden die kundenspezifischen Anforderungen und die bestehenden Artefakte zusammengesetzt, um das konkrete Produkt zu erzeugen. Im Idealfall kann dieser Schritt automatisiert durchgeführt werden.

2.2 Feature-Modellierung

Die Domänen-Analyse und die Feature-Modellierung sind Teilbereiche des Domain Engineerings und gliedern sich somit auch in die Software-Produktlinien-Entwicklung ein. Da die Ergebnisse der Domänen-Analyse innerhalb dieser Arbeit in Form von Feature-Modellen dargestellt wird, soll nachfolgend das erweiterte Domänen-Analyse Verfahren *FODA* vorgestellt werden.

2.2.1 Feature Oriented Domain Analysis (FODA)

Eine Weiterentwicklung der ursprünglichen Domänen-Analyse ist die *Feature Oriented Domain Analysis*, die Feature-Diagramme zu einem zentralen Bestandteil des Domain Engineering macht. FODA unterteilt sich dabei in zwei Prozesse:

- *Kontextanalyse (Context Analysis)* — In diesem Prozess werden die Grenzen der zu analysierenden Domäne definiert.
- *Domänen-Modellierung (Domain Modeling)* — Während dieses Prozesses wird das Domänen-Modell erstellt.

Die Feature Oriented Domain Analysis stellt die Grundlage der heutigen Feature-Modellierung dar und wurde erstmals in [KCH+90] veröffentlicht. In FODA werden erstmalig Feature-Modelle verwendet, die dazu dienen, die Ergebnisse der Domänen-Analyse strukturiert darzustellen. Das neu eingeführte Konstrukt bietet Möglichkeiten, die Variabilitäten und Gemeinsamkeiten aller Systeme innerhalb einer Anwendungsdomäne übersichtlich und gut strukturiert darzustellen. Für weitergehende Informationen sei auf [KCH+90] verwiesen. Auch wenn die ursprüngliche Feature-Modellierung nach FODA ([KCH+90]) heutzutage an verschiedenen Stellen erweitert wurde, so sind doch die Grundkonzepte die gleichen geblieben. Diese werden im nachfolgenden kurz vorgestellt.

Feature

Features sind für den Benutzer sichtbare beziehungsweise benutzbare Eigenschaften des Systems. Diese können je nach Definition der Anforderungsanalyse nur funktionale oder funktionale und nicht funktionale Eigenschaften des Systems sein.

Feature-Modell

Ein Feature-Modell beinhaltet alle Features einer Domäne, ebenso wie deren Abhängigkeiten untereinander. Mit diesem Modell können sowohl Gemeinsamkeiten als auch Variabilitäten zwischen den verschiedenen Systemen einer Domäne dargestellt werden. Die Anordnung der Eigenschaften (Features) innerhalb eines Modells

erfolgt hierarchisch in einer Baumstruktur, dadurch können einzelne Eigenschaften hierarchisch gruppiert und verfeinert werden. Zwischen den einzelnen Features können auch querschneidende Beziehungen erstellt werden. Hierdurch können Abhängigkeiten zwischen Features gleicher oder verschiedener Ebenen innerhalb des Baums erstellt werden. Es gibt insgesamt vier verschiedene Bestandteile innerhalb eines Feature-Modells:

- *Feature-Diagramm* — Hierbei handelt es sich um die grafische Darstellung des Modells.
- *Feature-Definitionen* — Beschreibung der einzelnen Features.
- *Kompositionsregeln für Features* — Sie ermöglichen die Darstellung von Abhängigkeiten zwischen einzelnen Features.
- *Argumente (Rationales)* — Sie sollen die Auswahl von Features gewichten.

Diese vier Bestandteile werden im folgenden nun noch einmal genauer vorgestellt.

Feature-Diagramm Ein Feature-Diagramm ist die grafische Repräsentation eines Feature-Modells. Da nahezu alle Feature-Modelle eine baumartige Struktur besitzen, lassen sie sich auch in Form eines solchen darstellen. Die Wurzel eines Feature-Diagramms stellt die eigentliche Produktlinie dar. Die einzelnen Features werden durch Knoten oder Blätter abgebildet, wohingegen die Beziehungen zwischen den Features durch Kanten dargestellt sind. Das FODA Feature-Diagramm unterstützt drei verschiedene Feature-Typen:

- **Verpflichtende Features (Mandatory features)** — Bei diesen Features handelt es sich um solche, die in jedem System der Domäne enthalten sein müssen.
- **Alternative Features (Alternative features)** — Diese Art von Feature sagt aus, dass nur eins der verfügbaren Subfeatures zu einem Superfeature ausgewählt werden darf.
- **Optionale Features (Optional features)** — Hierbei handelt es sich um Features, die nicht zwangsweise für ein System ausgewählt werden müssen.

Da sich die grafischen Notationen zwischen den verschieden Ansätzen unterscheiden, soll an dieser Stelle eine allgemeine grafische Notation vorgestellt werden, die nicht vollständig konform zur ursprünglichen FODA-Notation ist. Ein gern verwendetes Beispiel eines Feature-Diagramms ist der Auszug aus dem Feature-Modell für ein Auto. Abbildung 2.2 zeigt ein Teil-Feature-Diagramm für das Feature-Modell Auto. Das *Auto* verfügt über die Features *Getriebe*, *Motorleistung* und *Klimaanlage*. Das Feature *Getriebe* verfügt zusätzlich über die *Subfeatures Automatik*

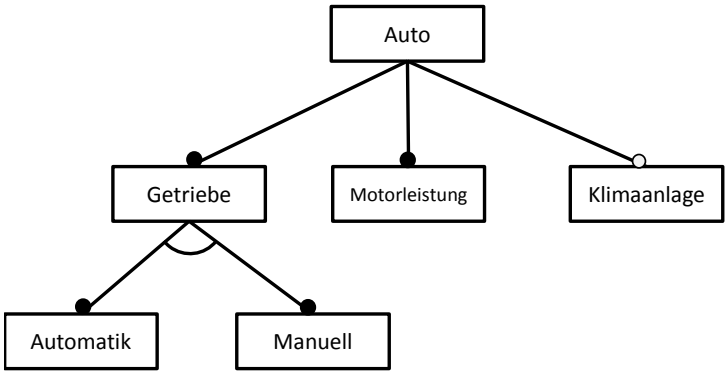

Abbildung 2.2: Beispiel Feature-Diagramm nach [KCH+90]

und *Manuell*. Durch die ausgefüllten Kreise wird symbolisiert, dass die Features *Getriebe* und *Motorleistung* verpflichtende Features sind. Aus dem Diagramm geht hervor, dass es sich bei den Features *Getriebe* und *Motorleistung* um verpflichtende Features handelt, dies ist durchaus nachvollziehbar, da ein Auto weder ohne Leistung noch ohne Getriebe vollständig wäre. Optional Features hingegen sind durch einen leeren Kreis gekennzeichnet. Im Diagramm der Abbildung 2.2 trifft dies auf das Feature *Klimaanlage* zu. Da es sich hierbei um kein zwingendes Bestandteil eines Autos handelt, ist klar. Die beiden verpflichtenden Subfeatures vom Feature *Getriebe* unterliegen einer alternativen Auswahl. Dies wird durch den Kreisbogen symbolisiert, der die beiden Kanten miteinander verbindet. Auch in diesem Fall ist klar verständlich, dass ein Auto entweder ein Automatikgetriebe oder ein Schaltgetriebe hat.

Feature-Definitionen Hierüber werden alle Features dokumentiert, die zum Modell gehören. Zusätzlich wird beschrieben, wann die Features gebunden werden sollen, hierfür stehen folgende Möglichkeiten zur Auswahl: *Kompilier-Zeit* oder *Laufzeit*. An dieser Stelle können auch benutzerdefinierte Bindungszeitpunkte festgelegt werden.

Kompositionsregeln für Features Um Abhängigkeiten und zusätzliche Beziehungen zwischen Features auszudrücken, gibt es die Möglichkeit, Kompositionsregeln innerhalb eines Feature-Modells zu definieren. Es werden zwei Regeltypen unterschieden:

- *Gerichtete Abhängigkeit (Require Rule)* — Sie gibt an, ob ein bestimmtes Features von einem anderen Feature abhängig ist. Im Beispieldiagramm *Auto*

können folgende Regeln definiert werden: *Eine Klimaanlage benötigt eine Motorleistung > 100 KW*

- *Gerichtete Beeinflussung (Influence Rule)*[3] – Durch diese Regel wird angegeben, dass ein Feature ein anderes beeinflusst. Es wird dabei jedoch keinerlei Aussage über eine Abhängigkeit (Require Rule) gemacht. Hätte man beispielsweise die beiden Features *Benzinverbrauch* und *Klimaanlage* könnte die Regel: *Die Klimaanlage beeinflusst den Benzinverbrauch* definiert werden.

- *Gegenseitiger Ausschluss (Mutual exclusive with Rule)* — Mit dieser Art von Regeln kann ausgesagt werden, dass sich bestimmte Features ausschließen. Im Beispieldiagramm *Auto* könnte die Regel auf die Features *Automatik* und *Manuell* angewendet werden, da sich diese Features aus dem Kontext heraus gegenseitig ausschließen. Da es sich bei diesen beiden Features aber um Subfeatures des gleichen Superfeatures handelt, kann dieser Ausschluß auch über eine Dekoration der Kanten angezeigt werden. Dieser Regeltyp wird deshalb normalerweise dann angewendet, wenn die sich ausschließenden Features auf verschiedenen Ebenen liegen.

Argumente (Rationales) Features können mit so genannten *rationales* annotiert werden. Diese *Argumente* sollen helfen, sich für bestimmte Features zu entscheiden. In unserem Beispiel könnte das Argument *Eine manuelle Schaltung ist kraftstoffsparender als eine Automatik* verwendet werden. Über dieses Werkzeug können mitunter komplexe Sachverhalte vereinfacht dargestellt werden. Für weitergehende Informationen zu rationales sei auf [KCH+90] verwiesen.

2.2.2 Kardinalitätsbasierte Feature-Diagramme

Sowohl in [CBUE02] als auch in [CUE04] werden kardinalitätsbasierte Feature-Diagramme vorgestellt. Hierbei handelt es sich um eine Erweiterung der bestehenden FODA-Notation. Mit dem bestehenden Modell konnten Features nur in die drei Gruppen *Verpflichtend*, *Optional* und *Alternativ* unterteilt werden, jedoch ohne die Angabe jeglicher Kardinalitäten. Mit den kardinalitätsbasierten Feature-Diagrammen ist es nun möglich, sowohl Features als auch Feature-Gruppen mit Kardinalitäten zu versehen, um neben der allgemeinen Aussage zur Selektion auch eine Aussage über die Anzahl der zu selektierenden Features machen zu können. Folgende Kardinalitäten werden für die bestehenden Feature-Typen definiert:

[3] Im weiteren Verlauf der Diplomarbeit wird diese Beziehung auch als «influence»-Beziehung bezeichnet

- *Verpflichtende Features* — Dieser Feature-Typ wird durch die Kardinalität $[1..1]$ oder die verkürzte Schreibweise $[1]$ repräsentiert.
- *Optionale Features* — Dieser Feature-Typ wird durch die Kardinalität $[0..1]$ definiert.
- *Alternative Features* — Diese Gruppierung wird weiterhin durch einen Bogen markiert, der zusätzlich mit der Kardinalität $[1..1]$ gekennzeichnet ist.

Neben der bisherigen Gruppierung für alternative Features können auch optionale Features mittels der Kardinalitäten gruppiert werden. Hierfür wird der Verbindungsbogen mit der Kardinalität $[0..1]$ annotiert. Neben der Abbildung der bestehenden Feature-Typen auf Kardinalitäten können über die Kardinalitäten auch Mengen ausgedrückt werden. So können Kardinalitäten wie $[0..1]$, $[1..5]$ oder $[1]$ ($\cong [1..1]$) definiert werden, wie es in der Beispielabbildung 2.3 dargestellt wird. Die Auswahlanzahl des annotierten Features muss sich dann jeweils in dem vorgegeben Intervall befinden. Das Beispiel aus Abbildung 2.2 wurde in Abbildung 2.3 nun in ein kardinalitätsbasiertes Feature-Diagramm transformiert. Zusätzlich ist das Feature *Sitz* hinzugekommen, um das Anwendungsgebiet der Kardinalitäten besser zu illustrieren. Da es in diesem Typ von Auto zwischen einem und fünf Sitzen geben kann, wurde dieses Feature mit der Kardinalität $[1..5]$ versehen. Dies bedeutet, dass das Auto mindestens einen Sitz, jedoch höchstens fünf Sitze haben darf. Für jeden einzelnen dieser Sitze muss zusätzlich ausgewählt werden, ob es sich um einen Sportsitz (Subfeature *Sport*) oder einen normalen Sitz (Subfeature *Normal*) handelt. Diese Auswahl ist mit einem Bogen annotiert, wodurch gekennzeichnet wird, dass nur eins der beiden Subfeatures *pro* Sitz gewählt werden kann. Die verpflichtenden Features *Getriebe* und *Motorleistung* sind nun durch die Kar-

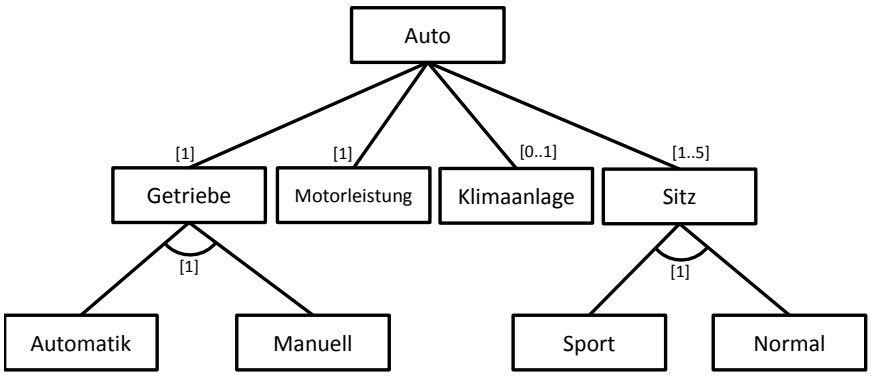

Abbildung 2.3: Beispiel kardinalitätsbasiertes Feature-Diagramm

dinalität $[1]$ gekennzeichnet, da sie in einem frontgetriebenen Auto genau einmal

vorkommen sollten. Das Feature *Klimaanlage* ist jedoch mit der Kardinalität versehen, bei der das Intervall bei 0 beginnt. Bei diesem Feature handelt es sich deshalb um optionales Feature, weil es genau 0 bis 1 mal vorkommen kann. Die Gruppierung der Subfeatures *Automatik* und *Manuell* mit der Annotation der Kardinalität $[1]$ ($\cong [1..1]$) sagt aus, dass genau eines der beiden Features selektiert werden muss.

2.2.3 Attributierte Features

Eine weitere sinnvolle Erweiterung für Feature-Modelle stellen die attributierten Features dar. Diese Erweiterung von Feature-Modellen wurde erstmals in [CBUE02] vorgestellt. Dadurch wird die Möglichkeit geschaffen, bestimmte Eigenschaften direkt innerhalb eine Features zu annotieren. Ein weiteres Subfeature ist dann an dieser Stelle nicht notwendig. Gerade für Eigenschaften wie zum Beispiel *Breite* oder *Höhe* sind keine gesonderten Features notwendig, da diese Eigenschaften über primitive Datentypen abgebildet werde können. Gerade bei umfangreichen Feature-Modellen kann dadurch ein wesentlicher Beitrag zur besseren Übersichtlichkeit geleistet werden, da nicht jede Eigenschaft als Feature abgebildet werden muss. Oftmals reicht die Annotation in Form eines Attributs. Die Anzahl von Attributen, die pro Features definiert werden können, ist nicht begrenzt. In Abbildung 2.4 ist die grafische Repräsentation dargestellt.

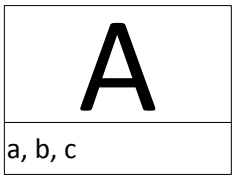

Abbildung 2.4: Attributiertes Feature

2.2.4 Werkzeugunterstützung

Für die Erstellung von Feature-Modellen und -Diagrammen stehen einige Editoren zur Verfügung, die es ermöglichen, auch komplexe Feature-Modelle zu erzeugen und zu verwalten. Im Rahmen dieser Arbeit wird der Feature-Modell-Editor *IO* verwendet. Eine genaue Vorstellung dieses Editors erfolgt in Abschnitt 7.2.2

2.3 Aspektorientierung

Ein bekanntes Hauptziel der Softwaretechnik ist es, das Niveau der Software-Qualität zu maximieren sowie die Entwicklungskosten zu minimieren. Um diese Ziele zu erreichen, muss ein einzelnes Softwaresystem unter anderem so strukturiert sein, dass

es ohne größere Eingriffe in das Gesamtsystem erweiterbar ist und die Wartung der bestehenden Komponenten kostensparend durchgeführt werden kann. Ebenso sollten Teile der Software wiederverwendbar sein, um in diesem Punkt eine weitere Kostenreduktion erzielen zu können. Ein wesentliches Strukturierungsmerkmal einer Software sollte deshalb die Modularisierung sein. Dies spielt gerade im Rahmen von Produktlinien-Architekturen eine große Rolle. Ein grundlegender Faktor der Modularisierung ist hierbei die *Trennung von Anforderungen (Seperations of concerns)*. Um diesem Problem zu begegnen gibt es verschiedene Lösungsansätze. Einer der vielversprechensten Ansätze ist hierbei jedoch die *Aspektorientierte Programmierung (AOP)*, da sie das Trennen von Anforderungen in umfassender Weise erlaubt.

2.3.1 Begriffe

Nachfolgend werden die wichtigsten Begriffe im Rahmen der AOP kurz vorgestellt.

Seperation of concerns

Bei *concerns* handelt es sich um Ziele, Anforderungen oder Konzepte, die bei der Softwarespezifikation getrennt voneinander definiert werden. Um diese Trennung bei der späteren Implementierung aufrecht zu erhalten, müssen diese concerns auch im eigentlichen Softwaresystem voneinander getrennt werden. Dies wird hauptsächlich durch die Modularisierung eines Softwaresystems erreicht. Hierbei werden die einzelnen Anforderungen und Konzepte (concerns), soweit es möglich ist, auf unterschiedliche Module verteilt. Der Begriff *seperation of concerns* wird auf *Edsger Wybe Dijkstra* zurückgeführt und gehört zu einem der wesentlichen Konzepte innerhalb der Informatik.

Crosscutting concern

In der objektorientierten Programmierung sind die Hauptstrukturierungsmerkmale Klassen und Pakete, wobei sich Pakete wiederum aus Klassen zusammensetzen. Ein Teil der *concerns*, der zum Spezifikationszeitpunkt noch getrennt voneinander definiert wurde, lässt sich mit den Mitteln der Objektorientierung während der Implementierung nicht mehr voneinander trennen, beziehungsweise müssen einzelne *concerns* auf verschiedene Klassen und Pakete verteilt werden. An dieser Stelle spricht man auch von *scattering* und *tangling*.
Scattering bezieht sich auf die Verstreuung einzelner concerns auf verschiedene Programmmodule, während *tangling* die Verflechtung verschiedener *concerns* meint (siehe Abbildung 2.5). Mit den Mitteln objektorientierter Programmierung lassen sich diese Probleme nur bis zu einem bestimmten Grad vermeiden. Ein prominentes Beispiel für ein *crosscutting concern* ist das Loggen von Anforderungen inner-

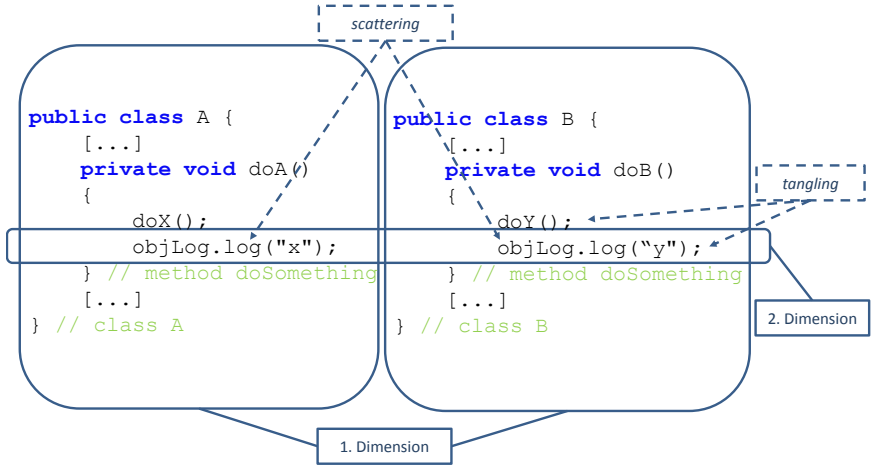

Abbildung 2.5: Beispiel *crosscutting*, *scattering* und *tangling*

halb eines Softwaresystems. Lässt sich die Anforderung „Bestimmte Ereignisse zur Laufzeit der Applikation müssen in eine Log-Datei geschrieben werden“ während der Spezifikation noch gut von den anderen Anforderungen trennen, entstehen bei der Implementierung Schwierigkeiten, da diese Anforderung aufgrund der betroffenen Module durch das gesamte Softwaresystem *schneiden* kann. Es existieren an dieser Stelle also zwei Dimensionen von Anforderungen, die sich überlagern: die erste Dimension, die Anforderungen beinhaltet, die mittels einer Klasse strukturiert werden können und die zweite Dimension, welche Anforderungen beinhaltet, die *quer* durch verschiedene Klassen schneidet (siehe Abbildung 2.5).

Aspekt

Das hierdurch entstehende Problem wird durch die AOP aufgegriffen [KLM+97]. Das Hauptziel der AOP ist es, Mechanismen zur Verfügung zu stellen, die es ermöglichen, die beiden vorgestellten Dimensionen (*crosscutting concens*) voneinander zu trennen. Um diese — sich überschneidenden — Anforderungen voneinander zu trennen, führt die AOP ein neues Konstrukt ein. Dieses Konstrukt wird als *Aspekt* bezeichnet. Ein Aspekt ist auf der gleichen Ebene wie eine Klasse angesiedelt und bietet die Möglichkeit, Anforderungen zu kapseln, die ursprünglich über verschiedene Klassen verteilt wurden. Ein Aspekt setzt sich aus zwei Teilen zusammen. Der erste Teil beinhaltet dabei die Angaben, an welchen Stellen der Aspekt im Basisprogramm angewendet werden soll und der zweite Teil beinhaltet die eigentliche Funktionalität des Aspekts.

Join Point

Die Stellen, an denen die Aspekte die bestehende Implementierung schneiden, werden *join points* genannt. Es handelt sich hierbei also um Punkte in der Programmausführung, an denen zusätzliche Funktionalität hinzugefügt werden soll. Join points können dabei auf verschiedenen Ebenen definiert werden. So können join points für Methoden und Felder definiert werden. Das bedeutet, dass zum Beispiel bei einem Methodenaufruf oder der Änderung eines Feldes Funktionalitäten des Aspekts ausgeführt werden. Ein anderer Ansatz bei der Bestimmung von join points sind Abfragesprachen, sogenannten *join point query languages*, bei denen join points nicht mehr einzeln definiert, sondern als eine Menge selektiert werden, die bestimmten vorgegebenen Kriterien entspricht.

Aspect Weaving

Sind sowohl die Aspekte als auch die join points definiert, müssen diese in einem weiteren Prozess zusammengeführt werden. Dieser Prozess wird als Weben (*Aspect-Weaving*) bezeichnet. Hierbei wird der Aspekt-Code an den durch die join points definierten Stellen in den ursprünglichen Code eingefügt. Für diesen Prozess kann bei der Definition von join points auf Methodenebene zusätzlich entschieden werden, ob der Aspekt-Code *vor*, *nach* oder *anstelle* der ursprünglichen Methode eingewoben werden soll. Neben dem statischen Weben auf Quellcode-Ebene zur Kompilierzeit gibt es auch die Möglichkeit des dynamischen Webens. Beim dynamischen Weben erfolgt die Adaption erst während des Ladens oder sogar erst zur Laufzeit des Programms.

2.3.2 Aspektorientierte Ansätze

Da es viele verschiedene Ansätze mit unterschiedlichen Merkmalen gibt, werden in diesem Abschnitt noch einmal zwei aspektorientierte Ansätze kurz vorgestellt.

- **AspectJ** [KHH+01][Asp01] ist eine aspektorientierte Erweiterung für Java. Ein eigener Compiler verwebt die Anwendungsklassen mit den Aspekten und erzeugt daraus normalen Java-Bytecode. Die Aspektimplementierung erfolgt in sogenannten *advices* und wird gemeinsam mit den join points (*pointcuts*) in einem Aspekt (*aspect*) definiert. AspectJ unterstützt sowohl das Weben auf Quellcode- als auch auf Bytecode-Ebene.

- **Composition Filters** [BA01][Com01] bildet die *concerns* in Form von Filtern ab. Das bedeutet, dass jeder Aspekt durch einen Filter dargestellt wird. Diese Filter können an Klassen angehangen werden, ohne dass sie in irgendeiner Form modifiziert werden müssen. Die Spezifikation der Filter ist hierbei deklarativ.

2.3.3 Object Teams

Da Object Teams für die Umsetzung dieser Arbeit ein wesentlicher Bestandteil ist, wird die Sprache im nachfolgenden Abschnitt noch einmal genauer vorgestellt.
Object Teams/Java (OT/J) ist eine Erweiterung für die Programmiersprache Java, welche sowohl aspektorientierte als auch kollaborationsbasierte Konzepte umsetzt. Hierbei wurden diverse bereits bestehende Ansätze und Konzepte berücksichtigt. Diese — vereinten — neuen Konzepte werden durch OT/J nahtlos in die bestehenden objektorientierten Konzepte von Java eingefügt. Es vereint somit verschiedene Konzepte und ermöglicht dadurch, ein weites Spektrum von Designkonzepten abzudecken. Um durch die Fülle der vereinten Konzepte die resultierende Sprache nicht zu verkomplizieren, wurde Object Teams einerseits klar strukturiert und andererseits wurden neue Konstrukte möglichst sparsam eingesetzt. Nachfolgend sollen die wichtigsten der Konzepte, die OT/J umsetzt, vorgestellt werden. Für eine vertiefende Vorstellung von Object Teams sei auf [Her02b] und [HHM07] verwiesen.

Kollaborationsbasiertes Design

Wie bereits dargestellt, können objektorientierte Applikationen oftmals in mindestens zwei Dimensionen unterteilt werden. Die erste Dimension ist hierbei von struktureller Natur und lässt sich durch Objekttypen (Klassen) begrenzen. Die zweite Dimension ist eine Kollaboration, in der die verschiedenen Objekttypen auf verschiedenen Ebenen in eine bestehende Kollaboration mit einbezogen werden (siehe Abbildung 2.6). Das bedeutet, dass sich jede Kollaboration über verschiedene Objekte erstrecken kann, wobei jedes Objekt in jeder Kollaboration eine andere Rolle spielt. Für die erste Dimension stellt die Objektorientierung Klassen zur Verfügung, für eine Kollaboration gibt es hingegen bisher kein Konstrukt. Ein wesentli-

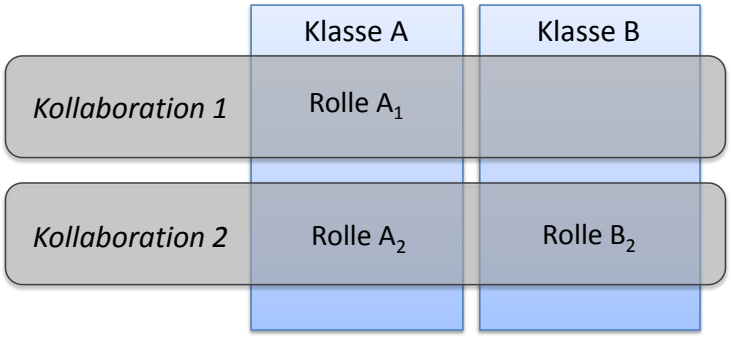

Abbildung 2.6: Kollaborationen und Klassen

cher Vorteil wäre es also, wenn es ein Konstrukt gäbe, mittels dessen Kollaboration

modularisiert werden könnten. Die Aspektorientierung stellt zwar für einzelne Rollen[4] Konstrukte zur Verfügung, jedoch fehlt es auch dort an der Möglichkeit, ganze Kollaborationen zu kapseln.

Rollen und Teams

Um Rollen und Kollaborationen abbilden zu können, führt Object Teams zwei neue Konstrukte ein. Zum einen handelt es sich dabei um *Teams*, welche die Kollaborationen repräsentieren und unabhängig vom Basissystem definiert werden können und zum anderen um Klassen innerhalb des Teams, so genannte *Rollen*[5], die wiederum die zuvor beschriebenen Rollen innerhalb einer Kollaboration repräsentieren. Um ein Team als solches zu kennzeichnen, muss die entsprechende Klasse mit dem *modifier* `team` versehen werden. Man spricht dann von einer *Team Klasse* oder kurz *Team*. Alle inneren Klassen eines Teams werden als Rollen bezeichnet. Neben Rollen können innerhalb eines Teams auch Methoden und Felder deklariert werden. Wie bei normalen inneren Klassen haben auch die Rollen eine implizite Referenz auf ihr umschließendes Team, diese Referenz ist unveränderbar. Auf diese Referenz kann über den Klassennamen unter Verwendung des Bezeichners `this` zugegriffen werden (`TeamA.this`). Das Team bildet zur Laufzeit den Kontext für die enthaltenen Rollen. In Listing 2.1 ist ein Beispiel für ein Team mit einer Rolle dargestellt.

```
1 public team class TeamA {
2    public class RoleA {
3       protected void print() {
4          System.out.println("Team: " + TeamA.this);
5       } // method print
6    } // role RoleA
7 } // team TeamA
```

Listing 2.1: Team mit einer Rolle

Die *playedBy*-Beziehung

Über die *playedBy*-Relation kann eine Rolle an eine Basisklasse gebunden werden. Diese Bindung resultiert in einer Verbindung zwischen der Rollen- und Basisklasseninstanz zur Laufzeit. Diese Verbindung wird hauptsächlich benötigt, um einen Kommunikationskanal zwischen Rollen- und Basisobjekt zu etablieren.

[4] Eine Rolle kann auch als Aspekt bezeichnet werden.

[5] Im Gegensatz zu den Grundkonzepten der aspektorientierten Programmierung (siehe Abschnitt 2.3.1) kann ein Aspekt in Object Teams sowohl eine Rolle als auch ein Teams sein

```
1 public team class TeamA {
2    public class RoleA playedBy BaseA {
3    } // role RoleA
4 } // team TeamA
```

Listing 2.2: Notation der playedBy-Relation

In Listing 2.2 ist ein Rolle dargestellt, die über die playedBy-Relation mit der Basisklasse `BaseA` verbunden ist.

lowering und *lifting*

Mit diesen beiden Begriffen wird die Umwandlung von einem Basisobjekt zu einem Rollenobjekt beschrieben. Jede Instanz einer an eine Basisklasse gebundenen Rolle hält eine Referenz auf das Basisobjekt. Diese Referenz bleibt während der gesamten Lebensdauer der Rolle erhalten. Will man das Basisobjekt einer Rolle erhalten, spricht man von *lowering*. Dieses Prinzip wird hauptsächlich verwendet, um Typkorrektheit zu erreichen. Durch das lowering kann eine Rolle zu ihrer Basis überführt werden. Dieses Verfahren wird zum Beispiel beim Aufruf einer Methode der Basisklasse angewendet.
Das *lifting* ist das Verfahren, welches genau in die andere Richtung läuft, das heißt von der Basisklasse zur Rolle. Da eine Basisklasse von verschiedenen Rollen annotiert werden kann, ist dieses Verfahren etwas anspruchsvoller. Beim lifting wird zu einem Basisobjekt das zugehörige Rollenobjekt ermittelt. Dieses Verfahren wird zum Beispiel bei einem von der Basisklasse ausgehenden Aufruf in die Rolle angewendet. Hierbei wird das Basisobjekt implizit mit seinem Rollenobjekt annotiert und ist fortan innerhalb des Teams nur noch als dieses Rollenobjekt sichtbar.

callin- und *callout*-Methodenbindung

callins können den Kontrollfluss an einem join point beeinflussen, in dem sie ihn unterbrechen und die passende Funktionalität der jeweiligen Rolle aufrufen. Dabei kann bestimmt werden, ob die aufgerufene Methode der Rolle additiv ist oder die bestehende Rolle die Basismethode ersetzt. Hierfür gibt es insgesamt drei Modifikatoren:

- *before* — Die Rollenmethode wird vor der Basismethode aufgerufen.
- *after* — Die Rollenmethode wird nach der Basismethode aufgerufen.
- *replace* — Die Rollenmethode ersetzt die ursprüngliche Basismethode komplett. An dieser Stelle muss darauf geachtet werden, dass etwaige Rückgabewerte von der Rollenmethode zur Verfügung gestellt werden.

Sowohl beim before als auch beim after Modifier werden keine Daten von der Rolle an die Basis zurückgegeben. Etwaige Rückgabewerte der Rollenmethode werden ignoriert.
Um eine callin-Bindung zu erstellen, muss es sich bei der umschließenden Klasse um eine Rolle handeln, welche über die playedBy-Relation an eine Klasse gebunden ist. Da sowohl callouts als auch callins immer aus der Perspektive der Rolle gesehen werden, bedeutet ein callin ein Aufruf in die Rolle hinein, wohingegen ein callout von der Rolle in Richtung der Basisklasse geht. Die Syntax für eine callin-Bindung ist wie folgt:

$$RollenMethode <- Modifikator\ BasisMethode$$

Auf die Übergabe von Parametern wird später genauer eingegangen. In Listing 2.3 ist ein Beispiel für eine callin-Bindung gegeben. Nach der Methode `doAction()` der Basisklasse `Action` wird die Rollenmethode `logAction()` der Rolle `WriteLog` aufgerufen.

```
1 public team class Logging {
2    public class WriteLog playedBy Action{
3       public void logAction(){...}
4       logAction <- after doAction;
5    } // role WriteLog
6 } // team Logging
```

Listing 2.3: Beispiel callin-Bindung

Ist das Team `Logging` aktiv (siehe Teamaktivierung in Abschnitt 2.3.3), wird nach jedem Aufruf der Methode `doAction` die Methode `logAction` aufgerufen. Methoden, die über den Replace-Modifikator an die Basismethode gebunden werden, müssen mit dem Modifikator `callin` versehen werden. Rollen, die mit einem `callin`-Modifikator gekennzeichnet sind, können über eine spezielle Syntax auf die ursprüngliche Basismethode vor ihrer Modifikation zugreifen. Die Syntax hierfür ist `base.m()`. Der Methodenname und die Übergabeparameter müssen hierbei mit der callin-Methode übereinstimmen.

callout Eine *callout*-Bindung geht genau in die entgegengesetzte Richtung, das heißt ausgehend von der Rolle in die Basisklasse hinein. Bei einer callout-Bindung wird deshalb der Aufruf einer Methode innerhalb einer Rolle an die gebundene Basismethode weitergeleitet. Alle Methoden-Weiterleitungen an die Basisklasse müssen explizit definiert werden. Bei einer callout-Bindung wird eine abstrakte Rollenmethode an eine Basismethode gebunden. Hierbei gilt folgende Syntax:

$$RollenMethode -> BasisMethode$$

Sobald eine Rollenmethode an eine Basismethode gebunden ist, wird die gebundene Basismethode bei jedem Aufruf der Rollenmethode ausgeführt. Bei einer callin-Bindung muss zuvor zusätzlich noch das zugehörige Team aktiviert werden. In Listing 2.4 ist ein Beispiel für eine callout-Bindung angegeben.

```
1 public team class Logging {
2    public class WriteLog playedBy Action{
3       abstract String getFilename();
4       String getFilename() -> String getName();
5    } // role WriteLog
6 } // team Logging
```

Listing 2.4: Beispiel callout-Bindung

Hierbei kann die Bindung sowohl bei callins als auch bei callouts nur über den Methodennamen oder auch über die komplette Signatur der Methode inklusive aller Übergabe- und Rückgabeparameter erfolgen. Im Listing 2.4 wurden die kompletten Signaturen der beiden Methoden angegeben. Eine callout-Bindung der Form `getFilename -> getActionName` wäre ebenfalls gültig. Sofern von einer Basis- oder Rollenmethode verschiedene *overloads* existieren, muss die Signatur explizit angegeben werden.
Sowohl bei callins als auch bei callouts können die Parameter von Basis- und Rollenmethode explizit aufeinander abgebildet werden, dies bezeichnet man als *Parameter Mapping*. Der einzige Unterschied zwischen dem Parameter Mapping bei callins und callouts besteht in der Übergaberichtung. Parameter zwischen Methoden der Basis- und Rollenmethoden können mittels der Klausel `with{...}` aufeinander abgebildet werden. Für die Abbildung einzelner Parameter bei einem callout gilt hierbei folgende Syntax:

$$Ausdruck -> BasisMethodenParameter$$
$$result <- Ausdruck$$

Für einen callin verläuft der Aufruf in umgekehrter Richtung.
In Listing 2.5 ist ein Beispiel für die Abbildung von Parametern gegeben. Die Parameter `cm` und `inch` werden bei der Abbildung zwischen den Methoden der Basis- und Rollenmethode zusätzlich noch konvertiert. Sollen die Parameter ohne eine explizite Abbildung mittels einer `with`-Klausel übergeben werden, kann diese weggelassen werden.

```
1  public team class TeamA {
2     public class RoleA playedBy BaseA{
3        abstract void setHeight(float cm};
4        abstract float getHeight();
5        void setHeight(float cm)
6           -> void setHeight(float inch) with {
7           cm * 0.3937008f -> inch
8        } // with
9        float getHeight() -> float getHeight() with {
10          result <- result * 2.54f
11       } // with
12    } // role RoleA
13 } // team TeamA
```

Listing 2.5: Beispiel Parameter-Abbildung

Teamaktivierung

Nur wenn ein Team aktiviert ist, sind die entsprechenden Rollen aktiv und die definierten callin-Bindungen werden ausgeführt, wenn die entsprechenden join points im Basisprogramm durchlaufen werden. Um ein Team zu aktivieren, gibt es verschiedene Möglichkeiten, es kann sowohl explizit als auch implizit aktiviert werden. Die Reihenfolge der Teamaktivierung bestimmt dabei die Reihenfolge und Priorisierung der Ausführung der einzelnen callins unterschiedlicher Teams. Ein Team kann auf zwei verschiedene Arten explizit aktiviert und deaktiviert werden. Zum einen über einen *Aktivierungsblock* und zum anderen über die Methoden `activate()` und `deactivate()`. Die Syntax für den Aktivierungsblock ist dabei wie folgt:

$$within\ (teamInstance)\ \{\ Anweisungen\ \}$$

Wird ein Team auf diese Art aktiviert, bleibt es solange aktiv, bis die Anweisungen innerhalb der geschweiften Klammern abgearbeitet sind, beziehungsweise aufgrund von Ausnahmebehandlungen aus diesem Scope-Bereich gesprungen wird.
Die Syntax für die Teamaktivierung mittels der imperativen Aktivierung ist wie folgt:

$$teamInstance.activate();\ Anweisungen;\ teamInstance.deactivate();$$

In diesem Fall ist das Team solange aktiv bis es durch die `deactivate`-Anweisung wieder deaktiviert wird.

Bei der impliziten Aktivierung wird ein Team immer dann aktiviert, wenn der Kontrollfluss auf ein Team oder einer der zugehörigen Rollen gelenkt wird. Eine weitere Möglichkeit der feingranularen Steuerung der Aktivierung von Teams, bieten die *Guard*-Prädikate. Diese ermöglichen eine bedingungsgeknüpfte Teamaktivierung auf verschiedenen Ebenen. Für weitergehende Informationen sei an dieser Stelle jedoch auf [HHM07] verwiesen.

Eine weitere — für diese Arbeit interessante — Möglichkeit der Teamaktivierung ist die externe Aktivierung über eine Konfigurationsdatei. Hierbei muss nicht in den Quellcode der Applikation eingegriffen werden, sondern nur eine entsprechende Konfigurationsdatei generiert werden. Diese wird beim Applikationsstart über ein spezielles VM-Argument der Virtuellen Maschine übergeben. Jede Zeile der Konfigurationsdatei enthält ein Team, dass aktiviert werden soll. Dieses Team muss unter Verwendung des kompletten Paketpfads angegeben werden. Des Weiteren muss dieses Team unter dem angegebenen Klassenpfad kompiliert und verfügbar sein. Beim Auslesen der Konfigurationsdatei wird für jedes enthaltene Team der Standard-Konstruktor aufgerufen. Die Teamaktivierungsreihenfolge hängt dabei von der Reihenfolge der Teams innerhalb der Konfigurationsdatei ab.

```
1 # Example configfile for OT Team activation
2 package1.TeamA
3 # ...
4 packageM.TeamN
```

Listing 2.6: Beispiel Konfigurationsdatei für die Teamaktivierung

2.4 Modell Driven Engineering (MDE)

Modelle werden heutzutage als wesentliche Bestandteile innerhalb von Softwareentwicklungsprozessen angesehen. Oftmals werden Modelle jedoch nur zu Dokumentationszwecken angefertigt und haben keinen direkten Einfluss auf den Entwicklungsprozess. Der Ansatz des *Model Driven Engineerings* hingegen ordnet der Verwendung von Modellen während des Entwicklungsprozesses einen wesentlich höheren Stellenwert zu. Sämtliche mit dem Entwicklungsprozess in Zusammenhang stehende Ressourcen können in Form von Modellen dargestellt werden. Um diesen Ansatz erfolgreich anwenden zu können, werden die verschiedensten Werkzeuge benötigt, die es ermöglichen Modelle bequem zu erstellen, anzusehen und zu verwalten. Damit die im Rahmen der Anforderungsdefinition angefertigten Modelle vollständig in die einzelnen Stufen des Entwicklungsprozess integriert werden können, muss zusätzlich die Möglichkeit bestehen, Modelle je nach benötigter Anforderung in verschiedene Formen zu transformieren. Dadurch können

die Modelle verfeinert und weiterentwickelt werden. Ziel dieser Arbeit ist es, ein Feature-Modell auf der Basis von verschiedenen Transformationsregeln in ein Object Teams-Modell zu transformieren. Hierfür wird zunächst das MDE-Konzept der *Model Driven Architecture (MDA)* und im Anschluß die Grundidee der Modell-Transformation vorgestellt.

2.4.1 Model Driven Architecture (MDA)

Die *Model Driven Architecture (MDA)* ist ein Standard der *Object Management Group (OMG)* mittels dessen der Softwareentwicklungsprozess mit Hilfe von Modellen und Codegeneratoren verbessert werden soll. Ziel dabei ist es, bestimmte Teile des Softwareentwicklungsprozesses zu formalisieren und zu automatisieren. Dabei soll keine hundertprozentige Formalisierung erfolgen, sondern es wird sich auf die sinnvoll zu formalisierenden Teilbereiche des Entwicklungsprozesses beschränkt. Durch den Einsatz der MDA soll in erster Linie der Entwicklungsprozess beschleunigt werden. Ein weiterer Vorteil dieses Ansatzes ist die zusätzliche Abstraktionsebene, die es ermöglicht auch komplexe Sachverhalte übersichtlich darzustellen. Die MDA strebt dabei eine klare Kategorisierung von Modellen an, die es ermöglicht zusammengehörige Anforderungen zu gruppieren. Ein Softwaresystem soll daher nicht durch ein einziges Modell repräsentiert werden, sondern durch verschiedene Modelle, die sich je nach Anforderungen unterscheiden können. Diese Modelle bilden dann die einzelnen Schichten des Gesamtmodells. Die MDA definiert hierfür folgende Modelle:

- *Computation Independent Model (CIM)* - Dieses Modell dient der umgangssprachlichen Beschreibung von Anforderungen.
- *Platform Independent Model (PIM)* - Durch dieses plattformunabhängige Modell werden die zu Grunde liegenden Geschäftsprozesse modelliert.
- *Platform Specific Model (PSM)* - Dieses plattformabhängige Modell beschreibt die System-Architektur und die benötigten Services.
- *Codemodell* - Hierbei handelt es sich um das Codemodell der Zielplattform.

Durch diese Form der Modellierung und der Unterteilung in verschiedene Schichten, wird sowohl eine gewisse Sprach- und Systemunabhängigkeit erreicht sowie das bekannte Konzept der *seperation of concerns* weiter verfolgt. Die MDA bezieht sich bei der Umsetzung der Konzepte auf verschiedene Standards, wie zum Beispiel die *Unified Modeling Language (UML)*, die *Meta-Object Facility (MOF)* und *XML Metadata Interchange (XMI)*, um nur einige der verwendeten Standards zu nennen. Neben der Erstellung von Modellen beschäftigt sich die MDA auch mit der Transformation von Modellen und unterscheidet dabei zwei verschiedene Transformationstypen:

- Die Modell-Transformation von einem Modell in ein anderes Modell (M2M)

- Die Transformation von Modellen in Code (M2C)

Im Rahmen dieser Arbeit soll die Modell zu Modell-Transformation verwendet werden. Sie wird deshalb im nächsten Abschnitt noch einmal genauer beleuchtet. Zunächst werden jedoch die zu Grunde liegenden Modelle vorgestellt.

Modelle

Modelle gehören zu den Grundbausteinen der MDA. Ein Modell wird dabei konform zu einer bestehenden Modellsemantik erzeugt. Man spricht an dieser Stelle von einem Modell eines Modells. Dieses zu Grunde liegende Modell wird allgemein auch als *Metamodell* bezeichnet. Modelle, die der Semantik des Metamodells entsprechen, werden als *konform* zum Metamodell bezeichnet. In Abbildung 2.7 ist das Metamodell für einen einfachen Baum dargestellt. Der Baum besteht aus Ele-

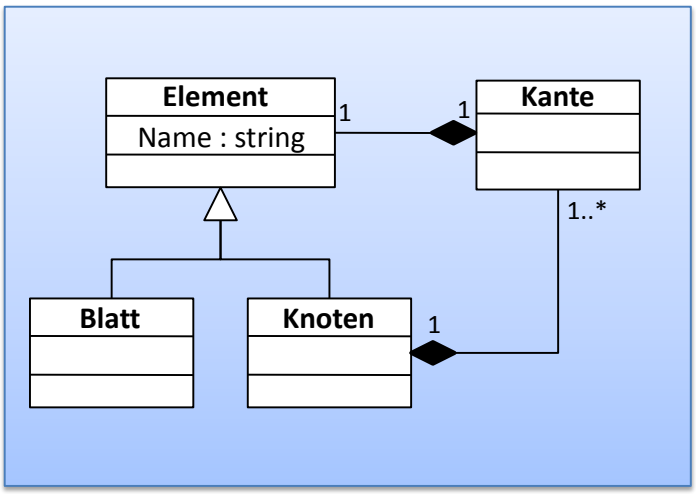

Abbildung 2.7: Metamodell für einen einfachen Baum

menten, die entweder *Knoten* oder *Blatt* sind. Über die *Kanten* können die Knoten und Blätter hierarchisch angeordnet werden. Das bedeutet, ein Knoten kann $[1..*]$ Kanten haben. Jede dieser Kanten wiederum ist mit genau einem weiteren Element verbunden. Blätter haben keine ausgehenden Kanten. In Abbildung 2.8 ist ein zum Metamodell des Baums konformes Modell dargestellt. Es definiert verschiedene Knoten (A, B, C), Kanten (Verbinden Knoten mit Knoten und Knoten mit Blättern)

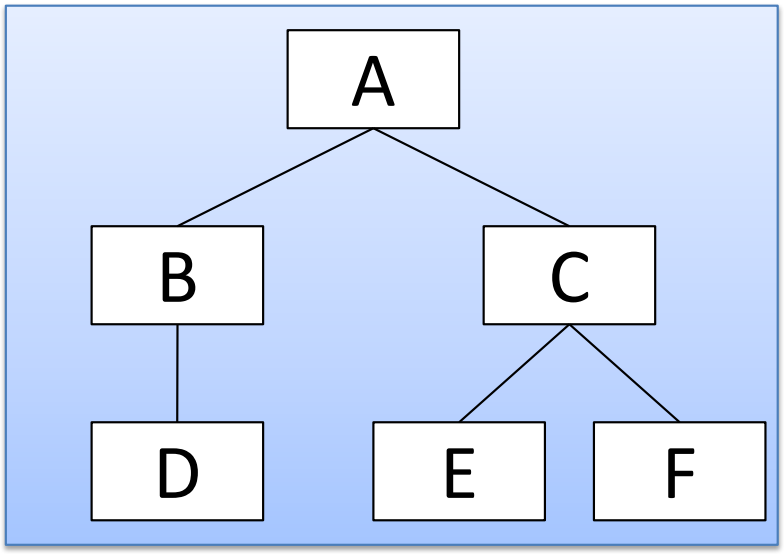

Abbildung 2.8: Zum Metamodell Baum konformes Modell

und Blätter (*D*, *E*, *F*). Das Baum-Modell setzt sich aus verschiedenen Elementen, die im Baum-Metamodell definiert wurden, zusammen. Ebenso wie die Elemente des Baums werden auch die Beziehungen untereinander im Baum-Metamodell definiert. An dieser Stelle muss berücksichtigt werden, dass jedes Metamodell zunächst auch ein Modell ist, dem ebenso ein Metamodell zu Grunde liegt. In Abbildung 2.9 ist noch einmal der Zusammenhang zwischen Modell und Metamodell dargestellt. Elemente eines Modells verweisen auf die Elemente des Metamodells, diese Beziehung wird als *meta* bezeichnet. Das Metamodell, zu dem das Metamodell des Baums konform ist, wird auch als Metametamodell bezeichnet. Oftmals sind diese Modelle konform zu sich selbst. Sie definieren sich also über ihre eigenen Elemente.

Queries/Views/Transformations (QVT)

QVT ist ein von der OMG im Rahmen der MDA entwickelter Standard für Modell-Transformationen. Bei der Modell-Transformation wird das zu einem Metamodell MM_a konforme Modell M_a in ein Modell M_b transformiert, wobei das Modell M_b konform zum Metamodell MM_b ist. Ist $MM_a = MM_b$ spricht man von einer endogenen andernfalls von einer exogenen Transformation. QVT definiert eine Standardlösung, um bestimmte Modelle in andere zu transformieren. Die hierfür in QVT verwendete Transformationssprache entspricht dem *Object Constraint Lan-*

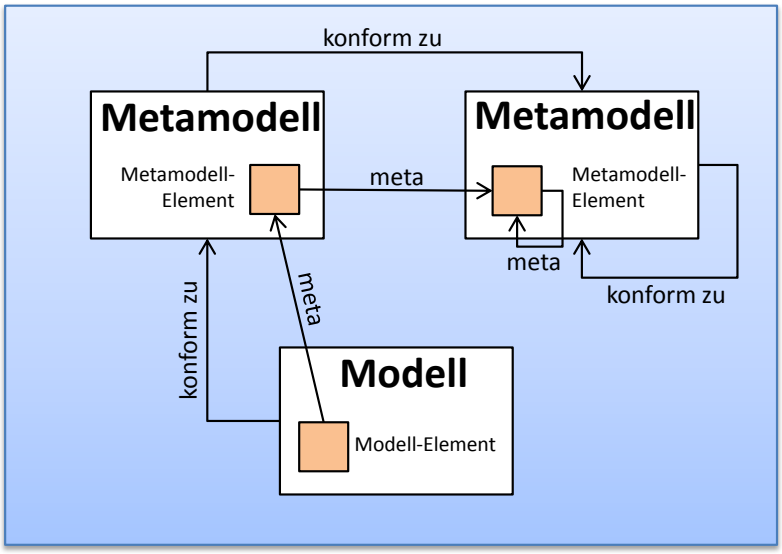

Abbildung 2.9: Zusammenhänge zwischen den einzelnen Modellen (nach [ATL06])

guage (OCL)-Standard. OCL ist eine deklarative Sprache mittels derer Regeln für UML-basierte Modelle definiert werden können. Des Weiteren definiert QVT drei Sprachen. Hierbei handelt es sich um zwei deklarativen Sprachen (*Relations*, *Core*) und eine imperative Sprache (*Operational Mappings*), welche die beiden deklarativen Sprachen um imperative Sprachkonstrukte erweitert.
Ein wichtiger Grundsatz der MDE ist es, alle verwendeten Bestandteile soweit wie möglich als Modelle zu beschreiben. Idealerweise sollten deshalb auch die Transformationen in Form von Modellen abbildbar sein. Analog den in Abschnitt 2.4.1 vorgestellten Prinzipien für Modelle muss dann also auch für eine Transformation ein Metamodell existieren, zu dem die eigentliche Transformation konform ist. Das Metamodell für die Transformation muss wiederum konform zu einem übergeordneten Metamodell sein. In Abbildung 2.10 ist ein Überblick zu den Zusammenhängen zwischen Modell und Transformation gegeben. In dieser Abbildung ist die Transformation des Modells M_a — das konform zum Metamodell MM_a — in das Modell M_b, welches wiederum konform zu MM_b ist, dargestellt. Die Modell-Transformation wird durch das Modell M_t definiert, welches konform zum Transformations-Metamodell MM_t ist. Alle drei Metamodelle (MM_a, MM_b und MM_t) sind konform zum Metametamodell MMM. Bei diesem Metametamodell handelt es sich in der Regel um das *Ecore*- oder *MOF*-Modell. Die eigentliche An-

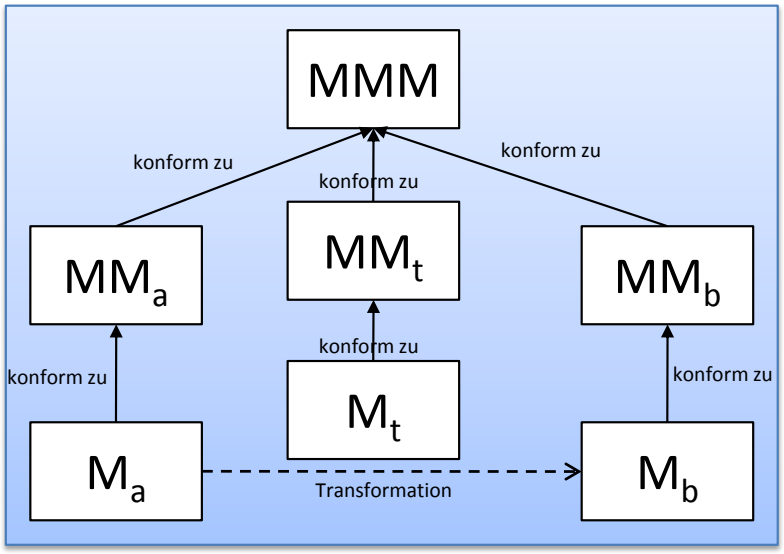

Abbildung 2.10: Überblick über die Modell-Transformation (nach [ATL06])

wendung des Konzepts und eine detaillierte Beispieltransformation wird in Kapitel 7 gegeben.

Kapitel 3

Fallstudie: Vorstellung

Im Rahmen der Diplomarbeit wird eine Fallstudie durchgeführt, um zwei wesentliche Vorteile zu erzielen. Die Fallstudie soll das Verständnis für die vorgestellten Ansätze verbessern, vor allem aber soll mit der Fallstudie überprüft werden, ob sich die entwickelten Lösungsansätze umsetzen und anwenden lassen. Um diese Ziele zu erreichen, wird die Vorstellung der Fallstudie in mehrere Teilbereiche unterteilt. Zunächst wird das für die Fallstudie benötigte Feature-Diagramm mit den bisher bekannten Modellierungstechniken erstellt (siehe Abbildung 2.2), in den nächsten Schritten soll das Feature-Diagramm dann gemäß den in Kapitel 4 vorgestellten Erweiterungen modifiziert werden, um es zum Schluss mit den in Kapitel 5 vorgestellten Transformationsregeln nach Object Teams zu transformieren. In diesem Kapitel wird zunächst die Fallstudie motiviert. Im darauf folgenden Abschnitt wird dann die Spiele-Produktlinie vorgestellt, bevor im letzten Abschnitt dieses Kapitels das Feature-Diagramm der Spiele-Produktlinie dargestellt wird.

3.1 Motivation

Für die eigentliche Fallstudie wurde eine Spiele-Produktlinie ausgewählt, da sich diese in vielerlei Hinsicht für die Darstellung eignet. Besonders Spiele lassen sich gut nachvollziehbar in verschiedene Domänen unterteilen und auf gewisse Grundfunktionalitäten reduzieren. Bei Spielen der aktuellen Generation lassen sich zum Beispiel folgende Kategorien unterscheiden:

- 3D-Spiele
  - Shooter
  - RPGs
  - Racing

- 2D-Spiele
  - Platformer(Jump'n'Runs)
  - Puzzle
  - Adventure

[IGN07]

Bei Spielen innerhalb einer Kategorie lassen sich viele Übereinstimmungen feststellen, so setzen heutige Shooter immer eine 3D- und Physik-Engine sowie bestimmte Sichtfeld-Eigenschaften voraus. Aufgrund der Komplexität von aktuellen 3D- und Physik-Engines werden diese auch nicht mehr exklusiv für ein Spiel entwickelt, sondern kommen, vor allem auch aus Kostengründen, in vielen Spielen zum Einsatz.
Aber auch für die Abbildung der Variabilität innerhalb eines Spiels eignen sich Produktlinien und Feature-Modelle. Angefangen bei verschiedenen Prozessortypen (PC, MAC) bis hin zu spezifischen Hardwareanforderungen (Grafikkarte, Prozessorleistung), lassen sich verschiedene Konfigurationen für ein Spiel unterscheiden.

3.2 Spiele-Produktlinie

Da die Komplexität der Fallstudie in einem überschaubaren Rahmen bleiben soll, wurde für die Spiele-Produktlinie eine Unterkategorie aus der Kategorie der 2D Arcade-Spiele ausgewählt. Es handelt sich hierbei um Spiele, die sich durch bewegte Avatare in jeglicher Repräsentation, Gegner und eine Spielfeldbegrenzung auszeichnen. Prominente Vertreter hierfür sind zum Beispiel Pong und PacMan, die nachfolgend kurz vorgestellt werden sollen.

3.2.1 Pong

Pong [Bus72] wurde 1972 von Atari veröffentlicht und zum weltweit ersten populären Computerspiel (siehe Abbildung 3.1). Bekannt wurde es vor allem durch Spielhallen. Obwohl bereits vorher Videospiele entwickelt worden waren, gilt Pong als Urvater der Videospiele. Das Prinzip des Videospiels ist recht einfach und ähnelt dem Spiel Tischtennis. Es gibt zwei Spieler, die durch *Schläger* repräsentiert werden, des Weiteren gibt es einen sich bewegenden Ball. Dieser Ball wird über eine Mittellinie zwischen den beiden Schlägern hin und her gespielt. Verfehlt ein Spieler den Ball, so erhält der Gegner einen Punkt. Im Laufe der Zeit gab es zahlreiche Erweiterungen der Ursprungsversion. Die Version, die in dieser Fallstudie betrachtet werden soll, wurde im Rahmen einer Lehrveranstaltung an der TU-Berlin entwickelt und trägt den Namen *OTPong*. In dieser Pongversion gibt es folgende Erweiterungen:

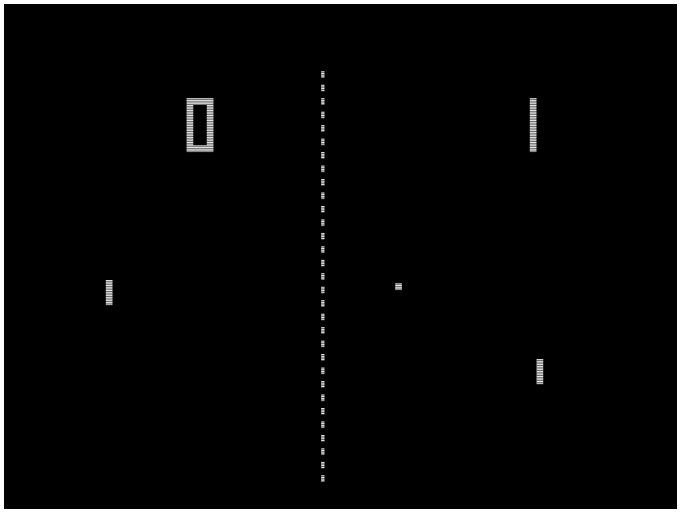

Abbildung 3.1: Screenshot der ersten Pong-Version

- **Spielfeldbegrenzung** — Das Spielfeld ist oben und unten durch eine Linie begrenzt.
- **Hindernisse** — Neben einer Spielfeldbegrenzung gibt es weitere Hindernisse innerhalb des Spielfeldes, an denne der Ball gemäß den Gesetzen der Physik abprallt.
- **Spezial-Gegenstände** — Auf dem Spielfeld werden willkürlich Gegenstände platziert. Werden diese durch den Ball getroffen, so erhält der Spieler, der den Ball zuletzt berührt hat, einen Bonus oder Nachteil.
- **KI** — Neben dem Mehrspielermodus gibt es auch einen Einzelspielermodus, bei dem der Computer einen der Schläger steuert.

3.2.2 Pac-Man

Ein weiteres Beispiel eines Videospiels, welches über die Spiele-Produktlinie abgebildet werden soll, ist Pac-Man [Tor80]. Pac-Man wurde am 12. Mai unter dem Namen *Puck-Man* in Japan veröffentlicht. Erst 1981 erschien das Spiel unter dem Namen „Pac-Man“ in den USA. Die Spielfigur muss durch ein Labyrinth von Gegenständen laufen und Punkte fressen, während es von Gespenstern verfolgt wird. Findet man einen bestimmten Gegenstand, so ändert sich die Farbe des Avatars und man kann für eine begrenzte Zeit auch die Gespenster fressen. Zusätzlich erscheinen während des Spielverlaufes auch verschiedene Spezialgegenstände, welche die Punktzahl zusätzlich erhöhen. In Abbildung 3.2 ist ein Screenshot der Ur-

sprungsversion dargestellt. Pac-Man eignet sich ebenso für die Spiele-Produktlinie,

Abbildung 3.2: Screenshot der ersten Pac-Man Version

da nahezu die gleichen Elemente wie bei Pong verwendet werden. Diese sind im Wesentlichen:

- Spielfeldbegrenzung
- Hindernisse
- Gegenstände
- Spielerrepräsentation
- KI

Die eigentlichen Unterschiede zwischen den Spielen lassen sich, wie sich später zeigen wird, gut über die Produktvariabilität abbilden.

3.3 Feature-Diagramm

Um die Spiele-Produktlinie übersichtlich darzustellen, soll in diesem Abschnitt das zugehörige Feature-Diagramm vorgestellt werden. Dieses Feature-Diagramm bildet die Grundlage für die weiteren Entwicklungs- und Anpassungsschritte, die in den nächsten Abschnitten vorgestellt werden. Das Feature-Diagramm entspricht weitgehend den bisher gültigen Konventionen. In Abbildung 3.3 ist das Feature-Diagramm für die Spiele-Produktlinie dargestellt. Das Wurzelfeature ist das allge-

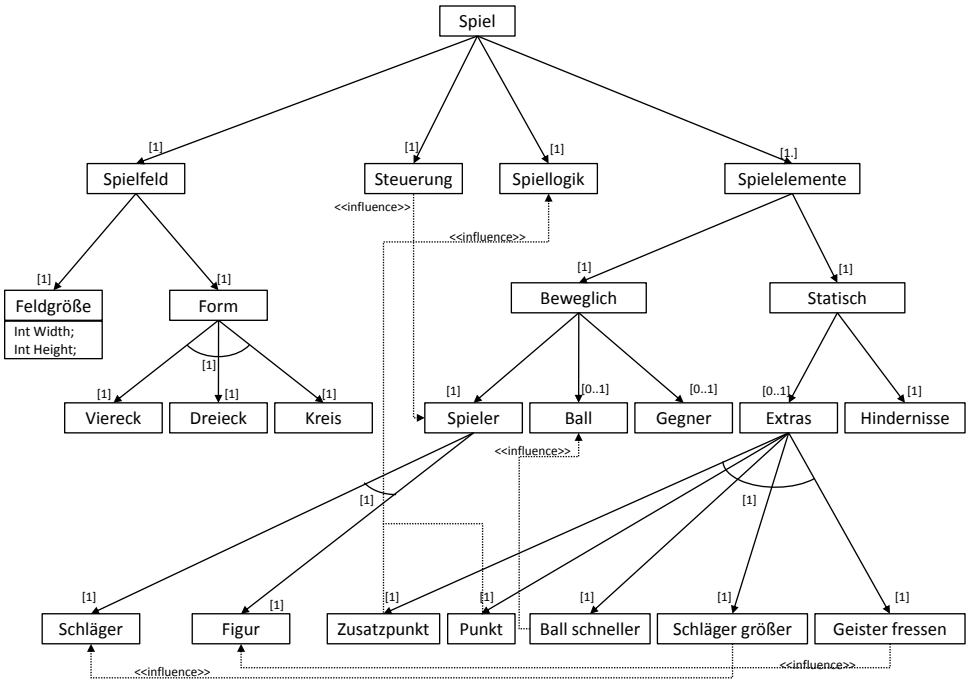

Abbildung 3.3: Feature-Diagramm der Spiele-Produktlinie

meine Feature *Spiel*. Ausgehend von diesem Feature gelangt man zu den Subfeatures *Spielfeld*, *Steuerung*, *Spiellogik* und *Spielelemente*. Das Subfeature *Spielfeld* beinhaltet die Eigenschaften des Spielfeldes, das sich sowohl durch die Feldgröße als auch durch eine Form charakterisieren lässt. Für diese Eigenschaften gibt es die entsprechenden Subfeatures *Feldgröße*, *Form*, *Viereck*, *Dreieck* und *Kreis*. Beim Feature *Feldgröße* handelt es sich um ein attributiertes Feature, da es zwei Attribute für die Parameter *Breite* und *Höhe* definiert. Das Feature *Steuerung* repräsentiert die Steuerung durch den Spieler, das Feature *Spiellogik* hingegen stellt die im Spiel implementierte Logik dar. Diese Features treten genau einmal auf. Des Weiteren gibt es noch das Feature *Spielelemente*. Dieses Feature repräsentiert die Gesamtheit der möglichen Spielelemente. Bei den Spielelementen gibt es eine weitere Unterscheidung nach beweglichen und statischen Spielelementen. Durch diese Unterteilung lassen sich die untergeordneten Features gut kategorisieren. *Spieler*, *Ball* und *Gegner* zählen zu den beweglichen Features, da sich diese später über das Spielfeld bewegen sollen. Das Feature *Spieler* steht mit dem Feature *Steuerung* in einer *«influence»*-Beziehung, da der Spieler durch die *Steuerung* gelenkt werden soll. Zu den statischen Spielelementen gehören die *Extras* und die *Hindernisse*. Die *Extras* beeinflussen zum Teil den *Spieler* (*Schläger größer*), den *Ball* (*Ball*

schneller) oder die *Spiellogik* (*Zusatzpunkt*) und die *Hindernisse* stellen die Spielfeldbegrenzung und die Wände innerhalb des Spielfeldes dar.
Soll aus diesem Feature-Diagramm nun Pong abgeleitet werden, müssen die entsprechenden Features ausgewählt werden. Durch diesen Schritt würde man dann auch die Produktvariabilität auflösen. Für den konkreten Fall Pong würde der Spieler zum Beispiel durch einen *Schläger* repräsentiert werden und es würden die Extras *Ball schneller* und *Schläger größer* zur Verfügung stehen.

Kapitel 4

Erweiterte Feature-Diagramm-Notation

In Abschnitt 2.2 wurden die verschiedenen Feature-Diagramm-Notationen ausführlich vorgestellt. In diesem Abschnitt wird sich nun mit den notwendigen Erweiterungen befasst. Diese werden hauptsächlich für die bessere Transformation von Feature-Modellen nach Object Teams benötigt, weiterhin lassen sich durch die Erweiterungen jedoch auch strukturelle Verbesserungen erzielen. Das Kapitel ist in zwei Abschnitte unterteilt. Im ersten Abschnitt werden die für die Transformation notwendigen Erweiterungen vorgestellt. Im zweiten Teilabschnitt werden dann die nicht unbedingt notwendigen, jedoch wünschenswerten den Übersetzungsprozess und das resultierende Modell verbessernden Spracherweiterungen dargestellt.

4.1 Stufe 1 - Erweiterungen

In diesem Abschnitt werden die grundlegenden Erweiterungen erläutert. Hierzu gehört zum einen die genaue Differenzierung zwischen der Aggregation und Generalisierung/Spezialisierung und zum anderen die daraus resultierenden Änderungen für die Kardinalitäten.

4.1.1 Aggregation und Generalisierung

Im Feature-Diagramm der Fallstudie lassen sich zwei Arten von Features unterscheiden. Dabei ist eine Art der Features eher von struktureller Natur und erinnert an die objektorientierten Konzepte der Super-/Subtypbeziehung. Bei der anderen Art handelt es sich um Features, die sich zu einem übergeordneten Feature kombinieren lassen und deshalb stark an das Konzept der Aggregation erinnern. Mit den bisherigen Mitteln der Featuremodellierung lassen sich diese Featurearten syn-

taktisch nicht eindeutig unterscheiden. Um Aggregationen und Generalisierungen/-Spezialisierungen eindeutig unterscheiden zu können, bedarf es deshalb einer Erweiterung — denn nur so können Feature-Diagramme sinnvoll auf objekt- und aspektorientierte Strukturen abgebildet werden. Es wird also ein zusätzlicher Strukturierungsmechanismus benötigt, der zum einen eine genaue Unterscheidung zwischen den beiden Featurearten und zum anderen die Kardinalitäten genauer spezifiziert. Da auch die Anpassung an den Kardinalitäten eine größere Erweiterung darstellt, soll diese in einem gesonderten Teilabschnitt betrachtet werden. Um eine Generalisierung/Spezialisierung genau von einer Aggregation unterscheiden zu können, wird in [KKL+98] eine zusätzliche Featurebeziehung eingeführt:

- - - - - Generalisierung/Spezialisierung
- ——— Aggregation

Eine genaue Aussage zur Semantik dieser Spracherweiterung wird in der Quelle jedoch nicht getroffen.
Durch diese Relationen kann jedoch eine genaue Unterscheidung zwischen einer Generalisierung/Spezialisierung und einer Aggregation getroffen werden.

Anpassung der Notation Durch die neu eingeführte Relation der Generalisierung/-Spezialisierung (- - - - -) kann die Notation an anderer Stelle vereinfacht werden. Bisher wurden Subfeatures — von denen immer nur eins selektiert werden durfte (Alternative-Auswahl) — mit einem Kreisbogen verbunden. Diese Notation soll bei der normalen Aggregation weiterhin beibehalten werden. Bei der Generalisierung/Spezialisierung ist jedoch davon auszugehen, dass es sich in allen Kombinationen, bei denen Subfeatures in einer Gen/Spec-Relation zu ihrem Superfeature stehen, immer um eine exklusive Auswahl handelt. In Abbildung 4.1 ist hierfür noch einmal ein Beispiel dargestellt. Da bei einer Generalisierung/Spezialisierung immer nur ein Subfeature ausgewählt wird, es sich also in jedem Fall um eine Einfach-Auswahl handelt, kann der Bogen entfernt werden. Abbildung 4.1 zeigt im oberen Bereich die ursprüngliche Form der Generalisierung/Spezialisierung mit Bogen. Im unteren Teil der Abbildung ist dieser Bogen nun nicht mehr enthalten, da durch ihn eine redundante Aussage zum Relationstyp gemacht wurde. Die beiden Beispiele in Abbildung 4.2 weisen strukturell den gleichen Aufbau auf und ließen sich deshalb bisher nur anhand des Kontextes in eine Generalisierung/Spezialisierung beziehungsweise Aggregation unterscheiden. Anhand des neuen Darstellungsmittels können diese Feature-Diagramme in ihrer Struktur nun klar unterschieden werden.

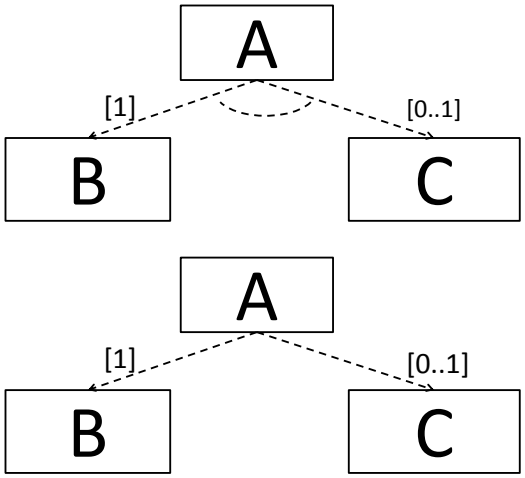

Abbildung 4.1: Reduktion des *Alternativ*-Bogens

Konsequenzen für die Kardinalitäten

Auch in der bisherigen Feature-Diagramm-Notation konnten bereits Kardinalitäten vergeben werden (siehe Abschnitt 2.2.2). Oftmals wurden jedoch nur die Kardinalitäten $[0..1]$ und $[1]$ verwendet, um optionale und verpflichtende Features zu unterscheiden. Wurden beliebige Kardinalitäten verwendet, so war das Kardinalitäten-Verhältnis zwischen Super- und Subfeature, das einer Vervielfältigung. Das bedeutet, dass sich innerhalb eines Feature-Diagramms die Kardinalitäten vervielfältigen, wenn man in Richtung der Blätter geht. Hat also ein Superfeature die Kardinalität n und ein Subfeature die Kardinalität m bedeutet dies, dass innerhalb des Feature-Diagramms das Superfeature n-fach mit jeweils m-facher Subfeaturemenge auftritt. Im Rahmen der Fallstudie hat sich jedoch gezeigt, dass die Kardinalitäten an einigen Stellen noch genauer differenziert werden müssen. Hierfür wurde ein weiteres Kardinalitätsverhältnis eingeführt, dass ausschließlich für die Generalisierung/Spezialisierung gültig ist.

Kardinalitäten bei der Generalisierung/Spezialisierung Im Rahmen der Fallstudie hat sich ein weiterer Kardinalitätentyp herauskristallisiert. An einigen Stellen des Featuremodells wird anstatt einer Vervielfältigung eine Einschränkung der Superfeature-Kardinalitäten auf Subfeature-Ebene benötigt. In Abbildung 4.3 ist ein Beispiel dargestellt, dass die Einschränkung der Kardinalitäten verdeutlichen soll. Es handelt sich bei diesem Beispiel um einen Auszug aus der Fallstudie. Die Features *Spielelement*, *Spieler* und *Ball* stehen in einer Generalisierungs/Spezialisierungs-Beziehung. Die Kardinalität über dem Feature *Spielelement* sagt aus, dass

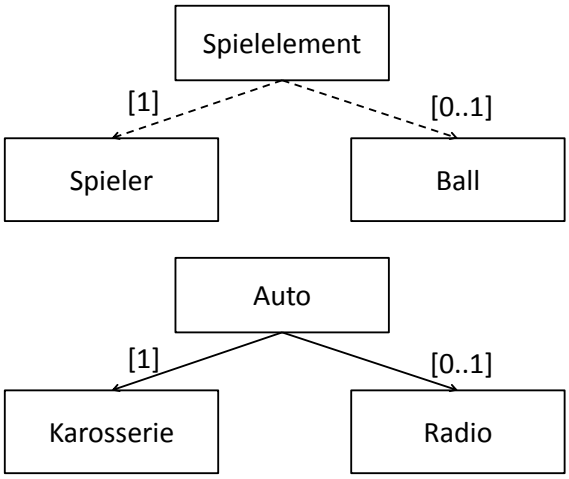

Abbildung 4.2: Beispiel für die Aggregation und Generalisierung

nur eine untere Grenze von mindestens 1 für alle Spielelemente festgelegt ist. Die beiden Subfeatures *Spieler* und *Ball* beschränken die Kardinalität des Superfeatures einmal auf $[1..2]$ und $[1..4]$. Hieran lässt sich erkennen, dass das Superfeature eine allgemeine *begrenzende* Kardinalität vorgibt, die durch die Subfeatures übernommen, spezialisiert und somit weiter eingeschränkt werden können.

Semantik der Subfeatures-Beziehungen

Aufgrund der Spezialisierung der Kardinalitäten bei einer Generalisierung/Spezialisierung in Richtung der Blätter bedeutet dies, dass sich die bisherige Vervielfältigungs-Semantik[1] nicht anwenden lässt. Für die Generalisierung/Spezialisierung gilt somit eine gesonderte Semantik, da ein Subfeature die Kardinalitäten seines Superfeatures weiter einschränkt und somit spezialisiert. Definiert ein Subfeature keine Kardinalität, so übernimmt es die Kardinalität seines Superfeatures.
Neben den bereits bekannten Kardinalitäten $[k..n]$, $[*](\cong [0..*])$ und $[k..*]$ wird zusätzlich die Kardinalität $[undef]$, die eine undefinierte Kardinalität darstellt, eingeführt. Die Kardinalität wird dann benötigt, wenn auf einer bestimmten Ebene des Feature-Modells noch keine konkrete Kardinalität festgelegt werden kann. Folgende Regeln gelten für diese Kardinalität innerhalb einer Generalisierung/Spezialisierung Super-/Subfeature-Beziehung:

- Die Kardinalität muss durch ein Subfeature überschrieben werden, so dass auf Blattebene keine Kardinalität von Typ $[undef]$ existiert.

[1] Hierbei handelt es sich um die herkömmliche Semantik bei der Featuremodellierung

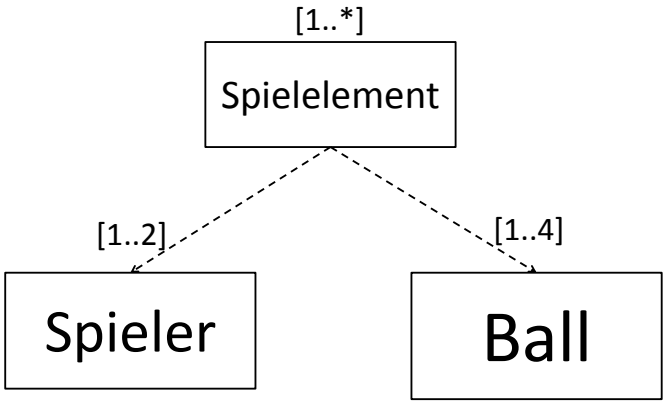

Abbildung 4.3: Beispiel für das Konzept der Einschränkung

- Die Kardinalität $[undef]$ eines Subfeatures, darf keine Kardinalität eines Superfeatures überschreiben, es sei denn, es handelt sich bei dieser Kardinalität auch um den Typ $[undef]$.

Ferner wird festgelegt, dass das Wurzelfeature eines jeden Featuremodells immer die implizite Kardinalität $[1]$ besitzt.

Konsistenzprüfung

Um Inkonsistenzen bei der Erstellung eines Featuremodells — gerade in Bezug auf die Generalisierungs/Spezialisierungs-Relation — von vornherein auszuschließen, werden nachfolgend Regeln zur Konsistenzprüfung definiert. Diese Regeln beziehen sich auf die in einer Super-/Subfeature-Beziehung stehenden Features. Folgendes soll gelten:
Sei $[k_P..n_P]$ die Kardinalität eines Superfeatures(P) und $[k_{C_i}..n_{C_i}]$ die Kardinalität für das i-te Subfeature(C_i), wobei ($1 \leq i \leq m$), dann lässt sich folgendes festlegen:

- $[k_P..n_P] \neq [undef] \rightarrow \forall i\ mit\ [1 \leq i \leq m] :\ [k_{C_i}..n_{C_i}] \neq [undef]$
  Hierdurch wird ausgesagt, dass kein Subfeature die Kardinalität $[undef]$ haben darf, wenn nicht auch für das Superfeature diese Kardinalität gesetzt ist.

- $[k_P..n_P] \neq [0..*] \rightarrow \forall i\ mit\ [1 \leq i \leq m] :\ [k_{C_i}..n_{C_i}] \neq [0..*]$
  Diese Regel besagt, die Kardinalität $[0..*]$ für ein Subfeature nur dann gesetzt sein darf, wenn auch das Superfeature über diese Kardinalität verfügt.

- $k_P \leq \sum_{i=1}^{m} n_{C_i}$, falls $k_P \neq [undef]$ und $k_P \neq *$
  Mit dieser Regel wird ausgedrückt, dass die Untergrenze des Intervalls eines Superfeatures, kleiner gleich der Summe der Obergrenzen aller Subfeatures sein muss. Dabei darf das Superfeature weder die Kardinalität $[undef]$ noch als untere Grenze $*$ haben.

- $\sum_{i=1}^{m} k_{C_i} \leq n_P \leq \sum_{i=1}^{m} n_{C_i}$, falls $n_P \neq [undef]$ und $n_P \neq *$
  Durch diese Regel wird beschrieben, dass die Summe der unteren Intervallbegrenzungen aller Subfeatures kleiner gleich der Intervallobergrenze des Superfeatures und die Summe der Intervallobergrenzen aller Subfeatures größer oder gleich der Intervallobergrenze des Superfeatures ist. Dabei darf das Superfeature weder die Kardinalität $[undef]$ noch als obere Grenze $*$ haben.

4.2 Stufe 2 - Erweiterungen

In diesem Abschnitt werden nun die weiteren Spracherweiterungen vorgestellt. Bei diesen Erweiterung handelt es sich nicht um Erweiterungen, die für den Transformationsprozess notwendig sind, sondern vielmehr um Zusätze, die den Transformationsprozess verbessern beziehungsweise um gewisse Funktionalitäten erweitern, so dass im Resultat ein bessere Strukturierung erzielt werden kann.

4.2.1 Elementar-Features

In einigen Fällen lassen sich Subfeatures eindeutig als elementare Features identifizieren. Elementare Features sind Features, die Eigenschaften mit geringer Komplexität darstellen. Diese können einerseits zu Attributen reduziert werden, andererseits kann durch eine entsprechende Kennzeichnung ein wichtiger Strukturierungsschritt für den späteren Transformationsprozess vorweggenommen werden. Die Entscheidung hierbei sollte im Hinblick auf die Komplexität der Eigenschaft getroffen werden. Handelt es sich um Eigenschaften, die zum Beispiel über Standard-Datentypen wie *Integer*, *String*, *Boolean* abgebildet werden können, empfiehlt es sich, diese über Attribute abzubilden. Bei Attributen, denen benutzerdefinierte oder komplexere Datentypen zugrunde liegen, sollten Elementar-Features verwendet werden. Um Elementar-Features gesondert kennzeichnen zu können, wird die Notation entsprechend erweitert. In Abbildung 4.4 ist die grafische Repräsentation dargestellt. Verfügt ein Feature über diese Kennzeichnung, wird es als ein Elementar-Feature behandelt. In Abbildung 4.5 ist ein Beispiel für die vorgestellte Notation gegeben. Dieses Beispiel zeigt einen Auszug aus der Fallstudie der Spieleproduktlinie. Bei den dargestellten Features handelt es sich um die Eigenschaften des Spielfeldes. Das Spielfeld setzt sich demnach aus Form und Größe zusammen. Für das Feature Form wären noch weitere Subfeatures wie Kreis, Viereck

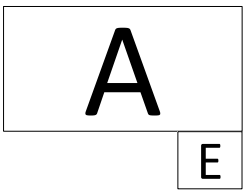

Abbildung 4.4: Notation Elementar-Feature

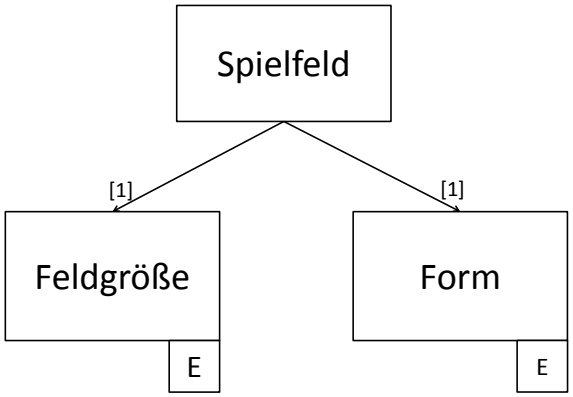

Abbildung 4.5: Beispiel für ein Elementar-Feature

oder Dreieck denkbar. Da die Features Größe und Form aufgrund ihrer geringen Komplexität als Elementar-Features betrachtet werden können, wurden diese mit der entsprechenden Notation versehen.

4.2.2 Design Patterns

Die Verwendung von wiederverwendbaren Entwurfsmustern, so genannten *Design-Patterns* [GHJV95], hat sich in der heutigen Softwareentwicklung erfolgreich etabliert. Ursprünglich stammte dieser Begriff aus der Architektur, wird aber heutzutage vielfältig in der Softwareentwicklung verwendet. Bei einem Entwurfsmuster handelt es sich um ein geprüftes und funktionierendes Konzept, das schematisch angewendet werden kann. Da auch in der aspektorientierten Programmierung von den Design-Patterns profitiert werden kann, sollen diese auch in der erweiterten Feature-Notation berücksichtigt werden. Es existieren eine Vielzahl verschiedener Design-Patterns, die sich mitunter stark in ihrer Komplexität unterscheiden. Inwiefern sich die einzelnen Patterns eignen, um in Feature-Modellen angewendet werden zu können, muss einzeln überprüft werden.

Observer Pattern

Da für den konkreten Anwendungsfall in der Fallstudie vor allem das *Observer Pattern* [GHJV95] [HK02] interessant ist, soll hierfür eine einfache Notation eingeführt werden. Bei dem Observer Pattern handelt es sich um ein Design Pattern, das zur Kategorie der Verhaltensmuster (Behavioural-Patterns) gehört. Dieses Muster gehört zu den *GoF*-Mustern [GHJV95], das vor allem für die grafische Darstellung verwendet wird. Das Observer-Pattern unterscheidet hierbei folgende zwei Rollen:

- Subject(Subjekt)
- Observer(Beobachter)

Die Subjekte nehmen hierbei die Rolle des *Beobachteten* ein. Die eigentlichen Beobachter sind die Objekte, die über Änderungen an den Subjekten benachrichtigt werden und im Anschluss daran bestimmte Aktionen durchführen. Für unseren konkreten Anwendungsfall (siehe auch Abbildung 4.7) soll der Beobachter die Aufgabe der grafischen Darstellung der Subjekte übernehmen. Das bedeutet: Sobald sich an einem Subjekt etwas Beobachtetes ändert, stellt der Beobachter die grafische Repräsentation des Subjekts neu dar. Um dieses Pattern im Rahmen der Spiele-Produktlinie anwenden zu können, müssen die betroffenen Features nur entsprechend gekennzeichnet werden. Hierfür werden in der Notation zwei weitere Attribute hinzugefügt. In Abbildung 4.6 ist die Darstellung eines Subjekts und ei-

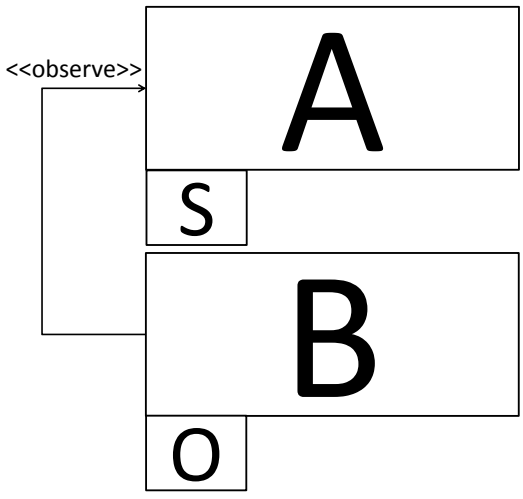

Abbildung 4.6: Feature mit *Subjektmarkierung*

nes Beobachters im Feature-Diagramm abgebildet. Die Kennzeichnung *S* oder *O* in der rechten unteren Ecke zeigt an, dass es sich entweder um ein **S**ubjekt oder

einen Beobachter (**O**bserver) handelt. Da es innerhalb eines Feature-Diagramms verschiedene Beobachter geben kann, die nur bestimmte Gruppen von Features beobachten, wird zwischen den beobachteten Subjekten und dem Beobachter noch eine neue Relation eingeführt. Die Relation *«observe»* zeigt an, dass ein Beobachter ein bestimmtes Subjekt beobachtet. In Abbildung 4.7 wurde ein Teilbereich der

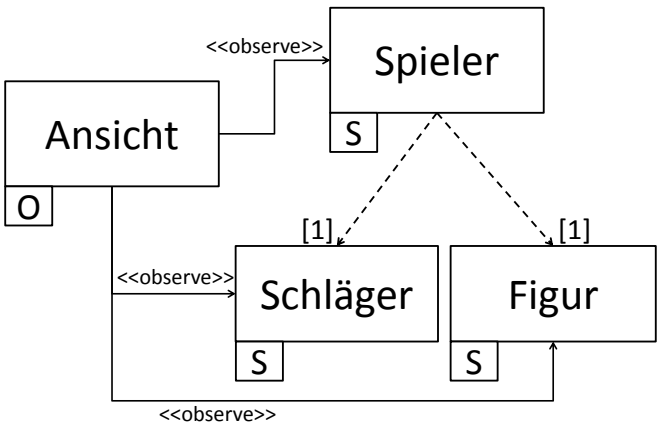

Abbildung 4.7: Beispiel für die Pattern Notation

Fallstudie aufgegriffen, der die zuvor vorgestellte Notation noch einmal illustrieren soll. Es handelt sich bei diesen drei Features um die Spielerdarstellung. Da der Spieler dargestellt werden muss — entweder in Form eines Schlägers oder einer Figur — sind alle drei Features als Subjekt gekennzeichnet. Zusätzlich existiert noch das Feature *Ansicht*, welches die eigentliche Beobachter Repräsentation ist. Durch die Relation zwischen diesem Feature und den anderen wird angezeigt, dass die Subjekte durch diesen Beobachter beobachtet werden.

Kapitel 5

Transformation von Feature-Modellen nach Object Teams

Nachdem im vorherigen Kapitel die für die Transformation nach Object Teams notwendigen Erweiterungen der Feature-Diagramm-Notation vorgestellt wurden, beschäftigt sich dieser Abschnitt nun mit der eigentlichen Transformation des Feature-Modells nach Object Teams. Dafür werden die in Abschnitt 2.4 vorgestellten Grundlagen des Model Driven Engineering angewendet. Ausgehend vom Eingabemodell, das durch das Feature-Modell repräsentiert wird, sollen die Transformationsregeln dazu verwendet werden, das Ausgabemodell in Form eines Object Teams-Modells zu erzeugen. Hierfür werden in den nachfolgenden Abschnitten Schritt für Schritt die einzelnen Transformationsregeln vorgestellt und ausführlich beschrieben. Mittels dieser Regeln kann dann eine Transformation des Feature-Modells in ein Object Teams-Modell erfolgen. Die grundlegenden Transformationsregeln wurden erstmalig in [HMPS07] vorgestellt und sollen in diesem Kapitel weiter ausgebaut werden. Zunächst soll hierfür die Notation von Rollen, Klassen und Teams vorgestellt werden. Anschließend werden in Abschnitt 5.2 die Stufe 1 - Transformationsregeln vorgestellt. Diese Regeln basieren auf den in Kapitel 4 vorgestellten *Stufe 1 - Erweiterungen*. Hierzu gehören die Transformationsregeln für die Aggregation und Generalisierung/Spezialisierung sowie die Transformation der «influence»-Beziehung. Im darauf folgenden Abschnitt werden dann die weiteren Transformationsregeln vorgestellt, die auf den Stufe 2 - Erweiterungen des 4. Kapitels basieren.

5.1 Grafische Notation von Rollen, Teams und Klassen

Da in den nachfolgenden Teilabschnitten die Transformation von Feature-Diagrammen nach Object Teams dargestellt wird, soll zuvor noch einmal auf die grafische Notation von Rollen, Teams und Klassen eingegangen werden. Die Notation der OT-Modelle erfolgt auf Grundlage von *UFA (UML for Aspects)* [Her02a]. Hierbei handelt es sich um eine Erweiterung der Modellierungssprache UML, in der die für die Aspektorientierung spezifischen Sprachbestandteile berücksichtigt wurden. In Abbildung 5.1 sind zwei Beispiele gegeben, anhand derer die in diesem Dokument verwendete grafische Object Teams Notation erklärt werden soll. Jedes Team und jede Rolle wird durch eine rechteckige Box dargestellt. Sofern es sich um Teams handelt beziehungsweise die OT-Paradigmen erfüllt werden, ist es möglich, die Boxen beliebig zu verschachteln. Im oberen Bereich der Box wird der Name

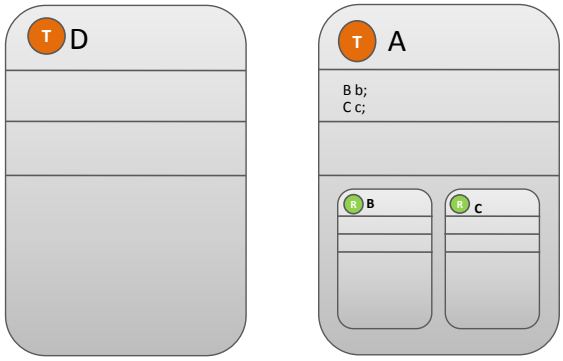

Abbildung 5.1: Beispiel-Notation Object Teams

der entsprechenden Rolle oder des Teams festgelegt. Weiterhin wird hier die Unterscheidung zwischen einem Team und einer Rolle getroffen. Teams sind mit dem Bezeichner *T* und Rollen mit dem Bezeichner *R*, gefolgt vom Namen, gekennzeichnet. Im mittleren Bereich der Box können Variablen deklariert werden. Im Beispiel in Abbildung 5.1 wurden zwei Variablen (*b* und *c*) mit vorangestelltem Typ (*B* und *C*) deklariert. Im unteren Bereich der Box können, sofern es sich um ein Team handelt, interne Rollen oder Teams deklariert werden. Im rechten Teil der Abbildung ist eine solche Konstellation dargestellt. Das Team *A* beinhaltet die Rollen *B* und *C*.

5.2 Stufe 1 - Transformationsregeln

5.2.1 Transformation von Aggregation und Generalisierung

Wie in Abschnitt 4.1.1 bereits festgestellt wurde, kann es mit der ursprünglichen FODA-Notation vorkommen, dass nicht eindeutig entscheidbar ist, ob es sich bei den vorliegenden Feature-Gruppen um eine Aggregation oder Generalisierung handelt. Hierfür wurde deshalb in Abschnitt 4.1.1 eine entsprechende Erweiterung vorgestellt, die diesem Problem begegnet. Durch die Erweiterung lassen sich die Features nun gut in passende Object Teams-Strukturen transformieren. Nachfolgend sollen die für die Aggregation und Generalisierung benötigten Transformationsregeln dargestellt werden.

Aggregation

In Abschnitt 4.1.1 wurde sowohl für die Aggregation als auch für die Generalisierung/Spezialisierung eine neue Relation festgelegt, eine Aggregation wird gemäß Definition durch eine durchgezogene Linie dargestellt. Folglich bedeutet dies, dass alle Features, die über eine durchgezogene Linie miteinander in einer Subfeature-Beziehung stehen, als Aggregationen betrachtet werden können. In Abbildung 5.2 ist die grafische Darstellung der Transformationsregel abgebildet. Aus dem Superfeature wird ein Team erzeugt, welches die beiden Subfeatures als innere Teams beinhaltet. Die Kardinalitäten sind in diesem Fall auf $[*..*]$ gesetzt, weil sie für die Transformationsregeln von untergeordneter Rolle sind. Handelt es sich bei dem Superfeature um das Wurzelfeature des Feature-Diagramms, so werden die Subfeatures nicht in ein Team, sondern in ein Paket eingegliedert.

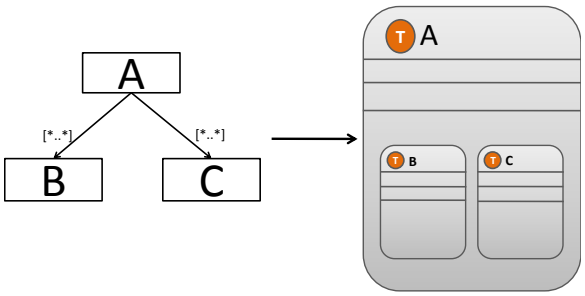

Abbildung 5.2: Abbildung von Aggregationen auf Object Teams

Generalisierung/Spezialisierung

Die Generalisierung/Spezialisierung wird im Gegensatz zur Aggregation durch eine gestrichelte Linie gekennzeichnet. Stehen also Features durch eine gestrichelte Linie in einer Subfeature-Beziehung, handelt es sich um eine Generalisierung/-Spezialisierung. Auch in diesem Modell sind die Kardinalitäten auf $[*..*]$ gesetzt, da diese für die Transformation keine Rolle spielen. Für die eigentlichen Kardinalitäten ist es jedoch wichtig, dass die Vorgaben aus Abschnitt 4.1.1 berücksichtigt und eingehalten werden. In Abbildung 5.3 ist die Transformationsregel dargestellt. Bei der Abbildung wird für das Superfeatures ein neues Team erzeugt. Für die bei-

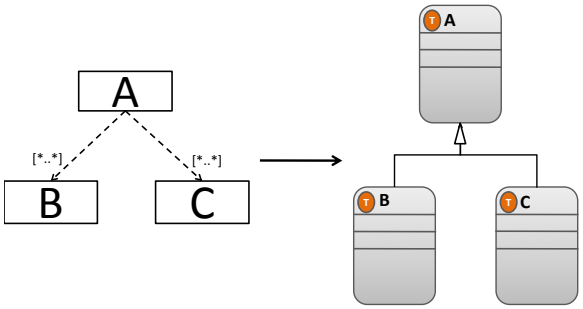

Abbildung 5.3: Transformation der Generalisierung/Spezialisierung nach Object Teams

den Subfeatures wird auch jeweils ein Team erzeugt. Im Anschluß werden die beiden Teams in eine Generalisierung/Spezialisierung-Beziehung mit dem Superteam gesetzt.

5.2.2 Transformationsregel für die *influence*-Beziehung

Ein weiteres Merkmal der Feature-Diagramme sind die Abhängigkeitsbeziehungen. Diese werden innerhalb eines Feature-Diagramms durch das Schlüsselwort *«influence»* gekennzeichnet. Sind zwei Features durch dieses Schlüsselwort in Relation gesetzt, unterliegen diese Features einer gerichteten Abhängigkeit. Diese Abhängigkeit lässt sich auf Object Teams durch die *playedBy*-Relation abbilden. In Abbildung 5.4 ist diese Transformationsregel grafisch dargestellt. Die Features *A* und *B* werden mittels dieser Transformationsregel über die *playedBy*-Relation in Beziehung gesetzt. Hierbei ist darauf zu achten, dass nur Rollen Klassen, Rollen oder Teams adaptieren können. Deshalb muss sichergestellt sein, dass ein Feature,

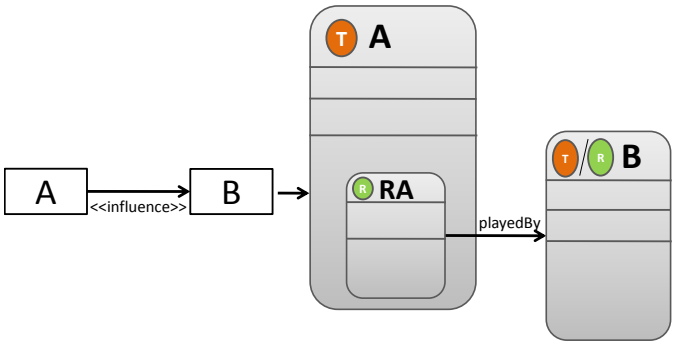

Abbildung 5.4: Transformationsregel für die *influence*-Beziehung

welches ein anderes Feature beeinflusst, entsprechend abgebildet wird. Um diesem Problem zu begegnen, erhalten Features, die in ein Team transformiert werden, das ein anderes Team beeinflusst[1], eine zusätzliche innere Rolle, die dann das beeinflusste Team oder die beeinflusste Rolle adaptiert.

5.3 Stufe 2 - Transformationsregeln

5.3.1 Transformation von attributierten Features

In Abschnitt 2.2.3 wurde die Attributierung von Features dargestellt. In diesem Abschnitt soll nun die Transformationsregel für die attributierten Features vorgestellt werden. Attribute eines Features können Eigenschaften sein, die nicht über ein separates Feature abgebildet werden müssen und deshalb nur als Attribut dargestellt werden. In Abbildung 5.5 ist die Transformationsregel dargestellt. Ausgehend vom Feature, welches die Attribute *a*, *b* und *c* enthält, wird ein Team erzeugt, das genau diese Attribute im Variablendeklarationsteil berücksichtigt. Die zugehörigen Typen können bereits im Vorfeld gesetzt oder müssen im Rahmen der späteren Implementierung genau definiert werden. Die Transformation des attributierten Features ergibt sich zudem aus dem Kontext. Es kann somit sein, dass das resultierende Konstrukt ein Team, eine Rolle oder eine Klasse ist. Hierfür werden in der Transfor-

[1] Beeinflusst bedeutet an dieser Stelle, dass eine Team beziehungsweise eine Rolle in einer *playedBy*-Relation mit einem anderen Team steht.

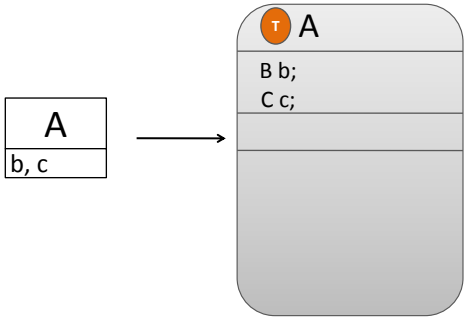

Abbildung 5.5: Transformation von attributierten Features

mationsregel keine besonderen Vorgaben gemacht, da sich dieser Zusammenhang aus den anderen Transformationsregeln ergibt.

5.3.2 Transformation von Elementarfeatures

In Abschnitt 4.2.1 wurden die Elementarfeatures vorgestellt. Diese sollen nun nach Object Teams transformiert werden. Abbildung 5.6 zeigt eine grafische Darstellung der Transformationsregel. Bei dieser Transformationsregel werden die Subfeatures als Attribute der Klasse beziehungsweise des Teams des Superfeatures abgebildet. Zunächst wird für das Superfeature ein Team erzeugt. Im Anschluss werden inner-

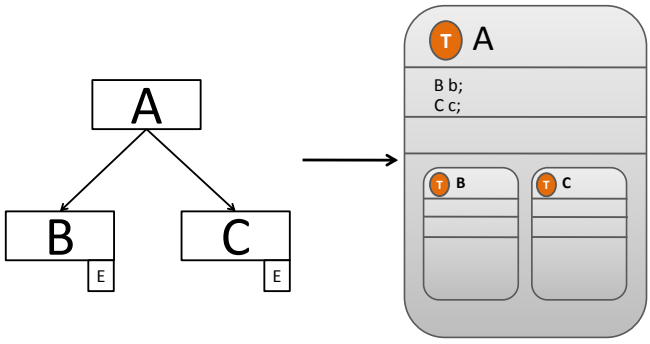

Abbildung 5.6: Transformation von Elementarfeatures nach Object Teams

halb des Teams zwei weitere Teams aus den als Elementarfeatures gekennzeichneten Subfeatures generiert. Diese Teams dienen im nächsten Schritt als Datentypen für die beiden Attribute *b* und *c*, die in diesem Schritt erzeugt werden

5.3.3 Transformation des Observer-Patterns

In Abschnitt 4.2.2 wurde die Feature-Diagramm-Erweiterung für das Observer-Pattern vorgestellt. Diese soll dazu dienen, Subjekte und Beobachter in Feature-Diagrammen zu annotieren. Um das Observer-Pattern korrekt nach Object Teams zu transformieren, wird eine spezielle Transformationsregel für die Transformation der Subjekte und Beobachter benötigt. Abbildung 5.7 stellt die Transformationsregel in grafischer Form dar. Im ersten Schritt werden Teams — gemäß den an-

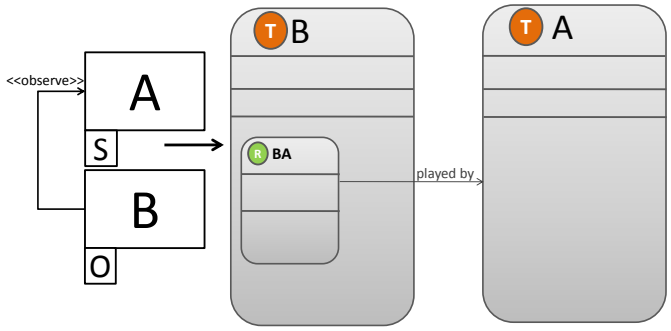

Abbildung 5.7: Abbildung von Patterns auf Object Teams

deren Transformationsregeln — für alle Features erzeugt, die als Beobachter oder Subjekt gekennzeichnet sind. Nun werden alle Features, die als Subjekte gekennzeichnet sind, iteriert. Steht ein Subjekt über die *«observer»*-Relation mit einem Beobachter in Verbindung, so wird für dieses Feature innerhalb des Beobachter-Teams eine Rolle erzeugt. Im Anschluß wird, ausgehend von der erzeugten Rolle, eine *playedBy*-Relation zum Team des Subjekts definiert.

Kapitel 6

Fallstudie: Anwendung der Transformationsregeln

In Abschnitt 3.3 wurde das Feature-Diagramm für die Spiele-Produktlinie vorgestellt. In diesem Kapitel wird zum einen das nach Kapitel 4 modifizierte Feature-Diagramm vorgestellt und zum anderen werden die in Kapitel 5 aufgezeigten Transformationsregeln auf das modifizierte Feature-Diagramm angewendet, um im Endresultat das Object Teams-Modell zu erhalten. Hierfür wird im nächsten Abschnitt noch einmal auf das ursprüngliche Feature-Diagramm eingegangen. Im darauf folgenden Abschnitt wird dann das modifizierte Feature-Diagramm vorgestellt, bevor dann im letzten Abschnitt die Transformationsregeln auf das modifizierte Feature-Diagramm angewendet werden.

6.1 Ursprüngliches Feature-Diagramm

In Abschnitt 3.3 wurde die Produktlinie in Form eines Feature-Diagramms vorgestellt. Dieses Feature-Diagramm ist noch einmal in Abbildung 6.1 dargestellt.

6.2 Modifiziertes Feature-Diagramm

In Abbildung 6.2 ist nun das mit den Erweiterungen versehene Feature-Diagramm dargestellt. Anhand der Beschriftungen sollen nachfolgend nun die einzelnen Anpassungsschritte noch einmal genau erläutert werden. Am oberen Teil des Feature-Diagramms hat sich nichts geändert. Das Spiel setzt sich nach wie vor aus den Features *Spielfeld*, *Steuerung*, *Spiellogik* und *Spielement* zusammen.

Modifikation Nr. 1 Die erste wesentliche Änderung ① am modifizierten Feature-Diagramm bezieht sich auf die Kardinalitäten. Waren die Kardinalitäten im ur-

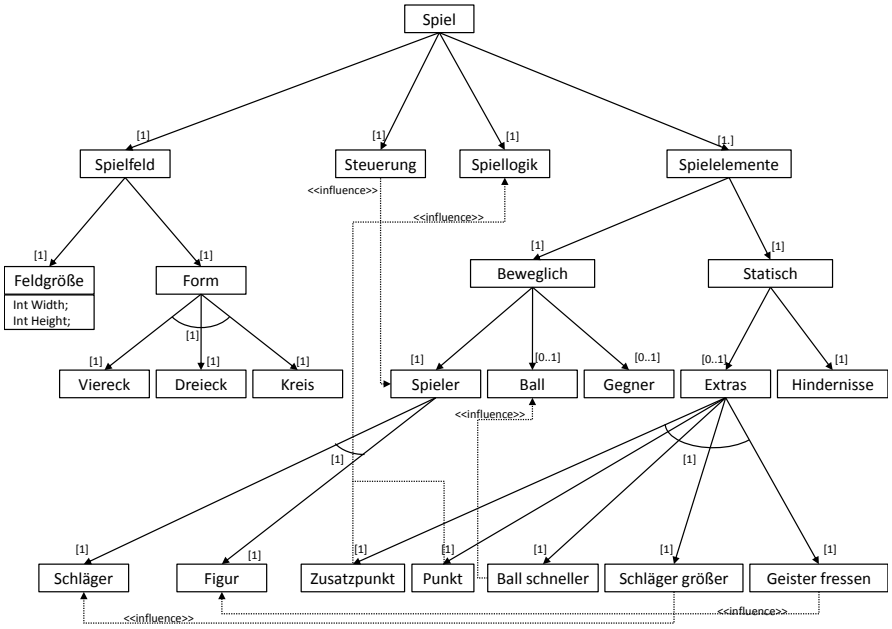

Abbildung 6.1: Feature-Diagramm der Spieleproduktlinie

sprünglichen Feature-Diagramm auf verpflichtende ($[1]$) und optionale ($[0..1]$) Features beschränkt und dadurch teilweise ungenau, so sind diese jetzt genauer und aussagekräftiger. Besonders geht dies aus dem Teilbereich *Spielelement* hervor. Ursprünglich hieß dieses Feature *Spielelemente*, um eine Aussage über die Kardinalität zu machen. Nun kann die Kardinalität des Features *Spielelement* jedoch wesentlich genauer mittels der entsprechenden Kennzeichnung dargestellt werden. Auch im Teilbereich der beweglichen Features lassen sich jetzt sehr genaue Aussagen über die Anzahl der Spieler ($[1..4]$) und die anderen Gegenstände (zum Beispiel $Ball$ $[0..5]$) innerhalb des Spiels machen. In diesem Zusammenhang findet auch die in Abschnitt 4.1.1 vorgestellte Unterscheidung der Kardinalitäten für Aggregation und Generalisierung/Spezialisierung ihre Anwendung. Während sich das *Spielfeld* aus den Features *Feldgröße* und *Form* mit ihren entsprechenden Kardinalitäten zusammensetzt und somit eine Vervielfältigung angewendet wird, erfolgt bei den Subfeatures des Features *Spielelement* eine Einschränkung der Kardinalitäten. Die Kardinalität des Spielelements ($[1..*]$) wird auf der Ebene des Features *Spieler* auf $[1..4]$ eingeschränkt.

Modifikation Nr. 2 Weiterhin lässt sich die in Abschnitt 4.1.1 vorgestellte *Vererbung* von Kardinalitäten im Diagramm unter ② erkennen. Da die Features *Be-*

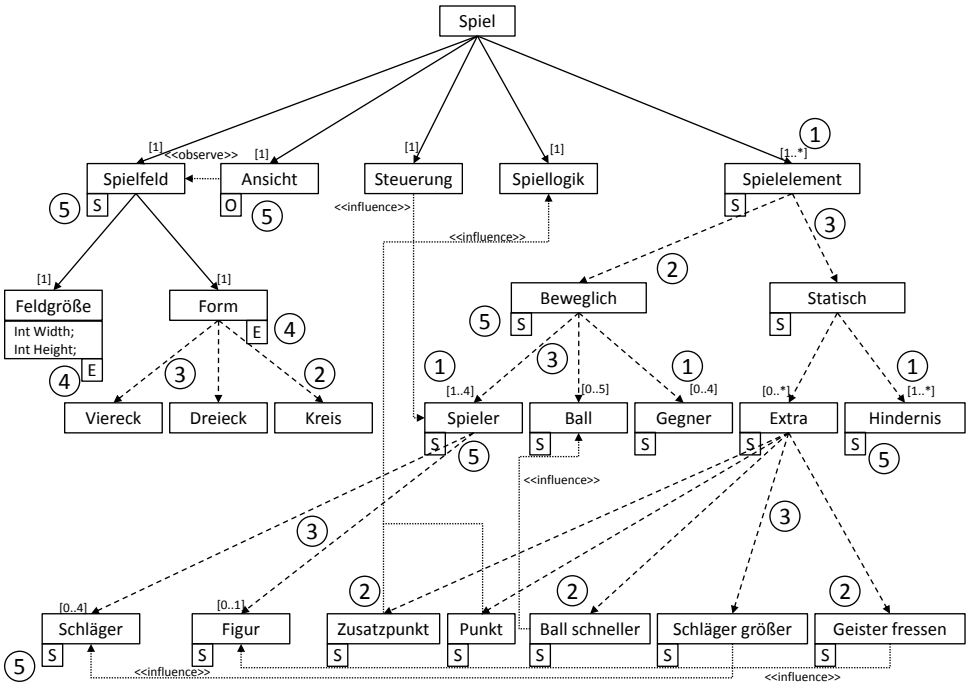

Abbildung 6.2: Modifiziertes Feature-Diagramm der Spiele-Produktlinie

weglich und *Statisch* die Kardinalitäten nicht weiter einschränken, werden sie bei diesen Features nicht mehr gesondert annotiert. Auf der nächsten Ebene kommt es jedoch zu einer Einschränkung, da zum Beispiel die Anzahl der maximal möglichen Spieler festgelegt werden soll. Deshalb sind die Features auf dieser Ebene wieder mit den entsprechenden Kardinalitäten versehen.

Modifikation Nr. 3 Eine der wesentlichsten Änderungen gegenüber dem ursprünglichen Feature-Diagramm ist die Unterscheidung zwischen *Aggregation* und *Generalisierung/Spezialisierung* ③. Durch die unterschiedliche Kennzeichnung (——/ - - - -) der Relationen können sowohl Aggregation als auch die Generalisierung/-Spezialisierung eindeutig differenziert werden. Es ist somit nun genau erkennbar, dass es sich beispielsweise bei dem Feature *Beweglich* um eine Spezialisierung des Features *Spielelement* handelt. Aus der Relation wird deutlich, dass dieses Feature in Richtung der Blätter noch weiter spezialisiert wird. Beim Feature *Spielfeld* lässt sich hingegen erkennen, dass es sich um eine Aggregation handelt, da sich das Spielfeld durch die Feldgröße und Form charakterisiert.

Modifikation Nr. 4 In Abschnitt 4.2.1 wurden die Elementarfeatures vorgestellt, auch diese sind bei der Modifikation des ursprünglichen Feature-Diagramms mit eingeflossen. An den Features *Feldgröße* und *Form* (④) lässt sich diese Modifikation erkennen. Da es sich sowohl bei Feldgröße als auch bei der Form um zwei elementare Features handelt, wurden diese mit einem $\boxed{E}$ gekennzeichnet und können somit beim späteren Abbildungsprozess berücksichtigt werden.

Modifikation Nr. 5 Die letzte Modifikation bezieht sich auf das Observer-Pattern (Abschnitt 4.2.2). Hierfür wurden, wie im entsprechenden Abschnitt beschrieben, gewisse Features ⑤ mit einem $\boxed{S}$ oder $\boxed{O}$ gekennzeichnet. Durch die Kennzeichnung als Subjekt werden diese bei der späteren Abbildung als durch den Beobachter beobachtete Objekte berücksichtigt. Auf eine Vererbung des Subjekt-Attributs wurde absichtlich verzichtet, da somit, unabhängig von der bestehenden Hierarchie, festgelegt werden kann, welches der Features bei der Anwendung des Observer-Patterns berücksichtigt werden soll. Aus Gründen der Übersichtlichkeit wurde die *«observe»*-Relation nur exemplarisch eingetragen. Es muss jedoch klar sein, dass zwischen jedem Subjekt und dem zugehörigen Beobachter im Normalfall eine solche Relation gesetzt werden muss.

In Abbildung 6.3 ist noch einmal das modifizierte Feature-Diagramm dargestellt. Dieses Diagramm soll für die nun folgende Anwendung der Transformationsregeln als Grundlage dienen. Auch in dieser Abbildung wurde zur besseren Übersicht die *«observe»*-Relation nur exemplarisch eingetragen. Im Normalfall ist jedes Subjekt mit einem Observer verbunden. Da es in diesem Diagramm nur einen Observer gibt, ist auch ohne die entsprechende Notation klar zu erkennen, zu welchem Observer das jeweilige Subjekt gehört. Ein Version dieser Abbildung im Querformat befindet sich im Anhang (Abbildung B.2).

6.3 Anwenden der Transformationsregeln

Im vorangegangenen Abschnitt wurde das modifizierte Feature-Diagramm vorgestellt. Ausgehend von diesem Feature-Diagramm und unter Zuhilfenahme der bereits vorgestellten Transformationsregeln soll nun das Object Teams-Modell erzeugt werden. Auch hierfür soll wieder die in Abschnitt 5.1 vorgestellte Notation verwendet werden. Aufgrund des relativ umfangreichen Modells wurde ein Teil der Relationen des Observer-Patterns nicht mit im Diagramm berücksichtigt, um so die Übersichtlichkeit der Darstellung zu wahren. In Abbildung 6.4 ist das OT-Diagramm dargestellt. Die angewendeten Transformationsregeln sind durch die Nummern markiert und sollen nachfolgend schrittweise betrachtet werden. Damit die Transformationsregeln leichter durchschaut werden können, wird in nachfol-

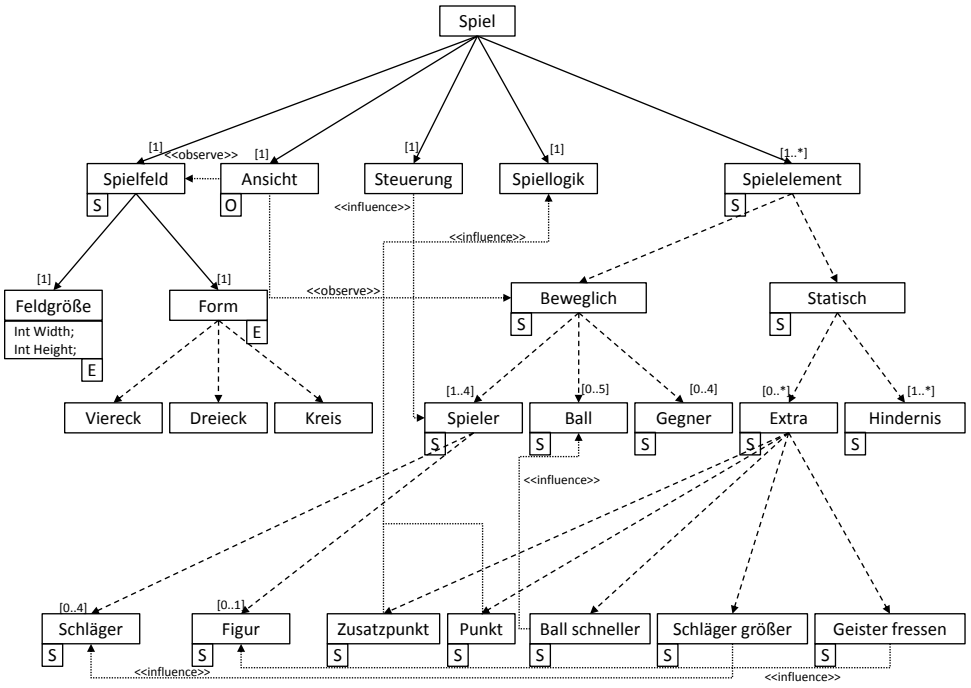

Abbildung 6.3: Modifiziertes Feature-Diagramm der Spiele-Produktlinie

gender Tabelle (Tabelle 6.1) noch einmal eine kurze Übersicht über die bestehenden Transformationsregeln gegeben.

Schritt Nr. 1 In diesem Schritt wurden die erste und zweite Ebene des Feature-Diagramms nach Object Teams transformiert. Auf die Features *Spiel*, *Spielfeld*, *Steuerung*, *Spiellogik* und *Spielelement* wurde aufgrund ihrer Struktur die Transformationsregel Ⓐ angewendet und somit wurden die zugehörigen Teams erzeugt. Aufgrund der in Abschnitt 5.2.1 beschriebenen Sonderregelung wurde das Wurzelfeature *Spiel* nicht auf ein Team, sondern auf ein Package abgebildet.

Schritt Nr. 2 In diesem Schritt wurden die Features *Feldgröße* und *Form* nach Object Teams transformiert. Da die betroffenen Features mit einem $\boxed{E}$ gekennzeichnet sind, wurde an dieser Stelle die Transformationsregel Ⓒ verwendet, da es sich bei diesen Features um Elementarfeatures handelt. Das heißt die Features *Form* und *Feldgröße* wurden in das Team *Spielfeld* gekapselt und anschließend als Typen für die Variablendeklaration verwendet.

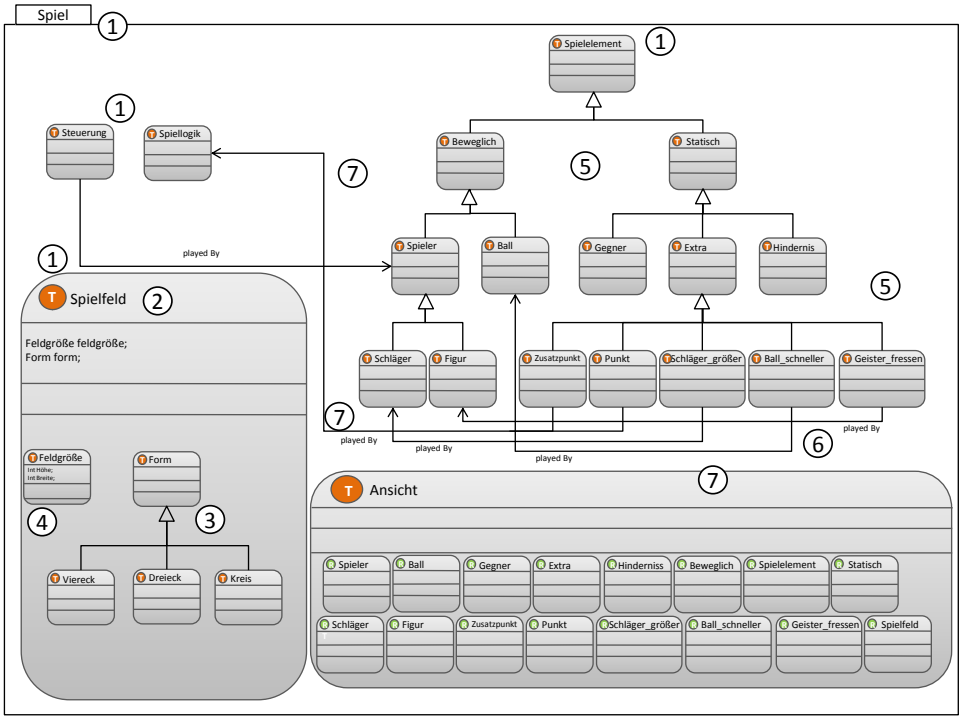

Abbildung 6.4: Object Teams-Diagramm

Schritt Nr. 3 In diesem Schritt wurden die Features *Viereck*, *Dreieck* und *Kreis* nach Object Teams transformiert. Hierfür wurde die Transformationsregel Ⓑ angewendet, da die Features über die entsprechende Relation miteinander verknüpft waren.

Schritt Nr. 4 In diesem Schritt wurde die Transformationsregel Ⓓ angewendet. Die Attribute Höhe und Breite waren als Parameter im Feature *Feldgröße* deklariert und wurden jetzt als Attribute in das Team *Feldgröße* übernommen.

Schritt Nr. 5 In diesen Schritten wurde erneut die Transformationsregel Ⓑ angewendet, um die markierten Features nach Object Teams zu transformieren. Die zuvor gesetzten Kardinalitäten spielen bei dieser Transformation keine Rolle und wurden deshalb auch nicht berücksichtigt.

Schritt Nr. 6 In diesem Schritt wurden die *playedBy*-Relationen erzeugt. Hierzu wurden die über die *«influence»*-Relation miteinander verbundenen Features auf Object Teams-Ebene mit der *playedBy*-Relation versehen. Hierfür wurde die

Bezeichner	Transformationsregel	Vorgestellt in Abschnitt
Ⓐ	Aggregation	5.2.1
Ⓑ	Generalisierung/Spezialisierung	5.2.1
Ⓒ	Elementarfeature	5.3.2
Ⓓ	Attributiertes Feature	5.3.1
Ⓔ	Observer-Pattern	5.3.3
Ⓕ	Influence/Played By	5.2.2

Tabelle 6.1: Übersicht der Abbildungsregeln

Transformationsregel Ⓕ angewendet. In diesem Schritt ist besonders auf die in Abschnitt 5.2.2 vorgestellte Sonderregelung zu achten. Diese Sonderregelung sagt aus, dass Features, die andere Features beeinflussen, bei der Transformation in ein Team mit einer zusätzlichen Rolle versehen werden müssen, welche die eigentliche Adaption implementiert. Dies resultiert aus dem Grundsatz, dass sich Teams untereinander nicht adaptieren können. Aus Gründen der Übersichtlichkeit sind die für die Adaption notwendigen Rollen in der Grafik nicht dargestellt. Das bedeutet, dass alle Teams, von denen eine playedBy-Relation ausgeht, im Normalfall über eine separate Rolle verfügen würden.

Schritt Nr. 7 In diesem Schritt wurde die Transformationsregel Ⓔ für das Observer-Pattern angewendet. Alle Features, die mit einem $\boxed{S}$ versehen sind und somit ein Subjekt darstellen, wurden bei der Transformation berücksichtigt. Für jedes dieser Subjekte wurde im Team *Observer* eine Rolle erzeugt. Jede dieser Rollen wird im Anschluß über die *playedBy*-Relation mit dem ursprünglichen Subjekt verknüpft. Zur besseren Übersicht wurden diese einzelnen *playedBy*-Relationen in der Darstellung nicht berücksichtigt. An dieser Stelle wäre es des Weiteren vorstellbar, die Feature-Hierarchie aus dem Diagramm zu übernehmen. Da jedoch nicht von vornherein sichergestellt ist, dass alle Features innerhalb einer Hierarchie als Subjekt gekennzeichnet sind, kann die Struktur nicht automatisiert übernommen werden.

In Abbildung 6.5 ist noch einmal das komplette Diagramm dargestellt. Dieser Abschnitt hat gezeigt, dass insgesamt nur zwei Schritte notwendig sind, um aus dem ursprünglichen Feature-Diagramm über die erweiterte Feature-Notation das fertige Object Teams Programmmodell zu erhalten. Das entstandene Programmmodell enthält noch die komplette Produktvariabilität aus dem ursprünglichen Feature-Diagramm. Um im letzten Schritt konkrete Produktinstanzen zu bilden, müssen die benötigten Features selektiert werden, damit die noch bestehende Produktvariabilität aufgelöst wird und die konkrete Produktinstanz entsteht.

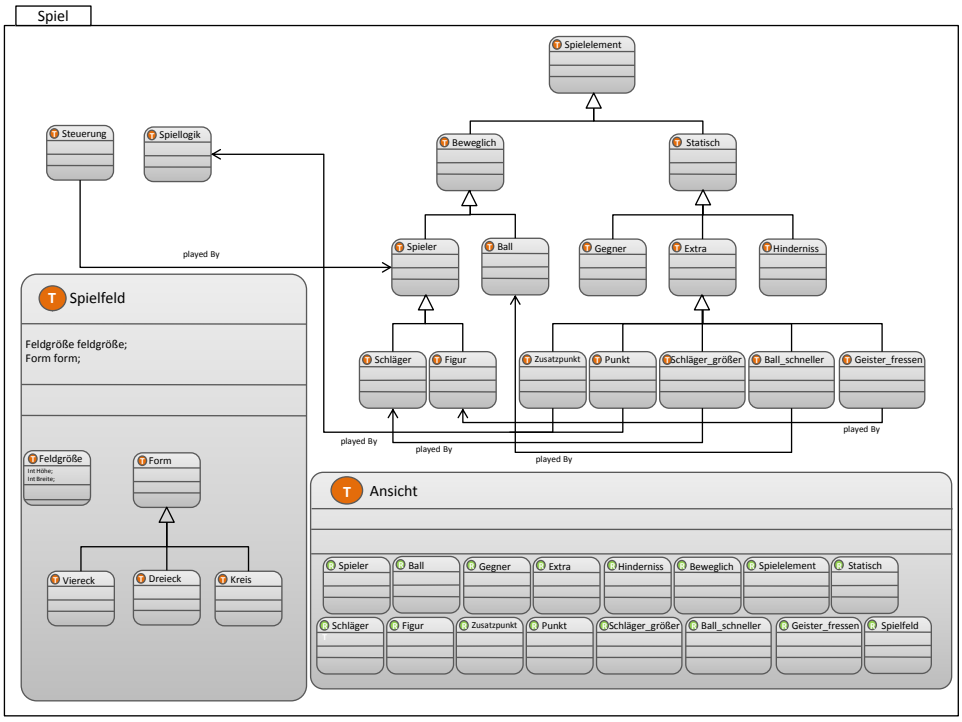

Abbildung 6.5: UFA Diagramm der Produktlinie (Querformat im Anhang)

Kapitel 7

Implementierung der Transformationsregeln

Dieses Kapitel befasst sich mit der Implementierung der in Kapitel 5 vorgestellten Transformationsregeln. Hierfür werden zunächst die technischen Grundlagen erläutert. Dazu gehören im Wesentlichen das *Eclipse Modeling Framework (EMF)*, die *ATLAS Transformation Language (ATL)* und die *Eclipse-Plugin-Entwicklung*. Auf eine nähere Vorstellung der Entwicklungsplattform *Eclipse* kann aufgrund ihrer allgemeinen Bekanntheit verzichtet werden. Für nähere Informationen sei auf [Ecl07] verwiesen. Nachdem die verwendeten Technologien und Frameworks vorgestellt wurden, soll im darauf folgenden Abschnitt die eigentliche Implementierung dargestellt werden.
Im Rahmen der Implementierung wurde ein Eclipse-Plugin programmiert, welches die in Kapitel 5 vorgestellten Transformationsregeln auf ein existierendes Feature-Modell anwendet, um daraus ein Object Teams-Modell zu erzeugen. Dabei wird ein mit dem Editor *IO* erzeugtes Feature-Modell in ein *OTUML*-Modell transformiert. Die eigentlichen Transformationsregeln wurden mit ATL implementiert und in das Plugin eingebettet. Ist das Plugin installiert, kann die Transformation über das Kontextmenü des Navigator-Fensters von Eclipse angestoßen werden.
Der zugehörige Implementierungsabschnitt (Abschnitt 7.2) ist dabei wie folgt aufgebaut. Zu Beginn des Abschnitts wird ein Gesamtüberblick gegeben. Im Anschluss wird dann das Eingabemodell vorgestellt. Hierbei handelt es sich um ein Feature-Modell. Im darauf folgenden Abschnitt wird das OTUML-Ausgabemodell dargestellt. Im letzten Abschnitt des Implementierungsteils wird dann die technische Umsetzung der Transformationsregeln genau beleuchtet.

7.1 Verwendete Technologien und Frameworks

In diesem Abschnitt werden zunächst die für die praktische Umsetzung notwendigen Technologien vorgestellt. Es wird mit dem *Eclipse Modeling Framework* begonnen. Anschließend wird ein Überblick über die Transformationssprache ATL gegeben, bevor im letzten Abschnitt ein kurzer Einblick in die Eclipse-Plugin-Entwicklung gegeben werden soll.

7.1.1 Eclipse Modeling Framework (EMF)

EMF ist ein Open-Source Java-Framework zur Erstellung und Bearbeitung von strukturellen Modellen, die im Rahmen der modell-getriebenen Software-Entwicklung zum Einsatz kommen. Für die interne Darstellung der Modelle verwendet EMF das *XMI (XML Metadata Interchange)*-Format. Das in EMF verwendete Metamodell basiert größtenteils auf dem *MOF (Meta Object Facility)*-Metamodell der OMG. Die Unterschiede zwischen den beiden Metamodellen sind hauptsächlich in der Benennung einzelner Elemente zu finden. Das in EMF verwendete Metamodell wird als *Ecore* bezeichnet.
EMF bietet die Möglichkeit der Code-Generierung, um aus bestehenden strukturellen Modellen Code für mit dem Modell in Zusammenhang stehende Tools und Applikationen zu erzeugen. Hierfür reicht es aus, ein Modell beispielsweise in XML zu definieren, damit mittels des EMF-Frameworks der zugehörige Code generiert werden kann. Der erzeugte Code kann modifiziert und erweitert werden, ohne dass das zu Grunde liegende Modell verändert werden muss.
EMF bietet weiterhin die Möglichkeit, basierend auf Metamodellen einfache Editoren zu erzeugen, mittels derer konforme Modelle erzeugt und bearbeitet werden können. EMF besteht dabei aus zwei fundamentalen Frameworks. Zum einen handelt es sich dabei um das *core framework* und zum anderen um das *EMF.Edit* Framework. Das core-Framework stellt die grundlegende Codegenerierung und die Laufzeitumgebung für die Entwicklung zur Verfügung. Das EMF.Edit Framework erweitert das core Framework um verschiedene Funktionalitäten, die es ermöglichen einfache Modell-Editoren automatisiert zu generieren.
Die in dieser Arbeit verwendeten Metamodelle basieren auf dem Ecore-Standard. Die zugehörigen Editoren wurden mit Hilfe von EMF erstellt.

7.1.2 ATLAS Transformation Language (ATL)

Die *ATLAS Transformation Language* ist eine von der *ATLAS-Group (INRIA und LINA)* entwickelte Modell-Transformationssprache. Im Bereich des *Model Driven Engineering (MDE)* (siehe Abschnitt 2.4) ermöglicht ATL, basierend auf Transformationsregeln, verschiedene Ausgabemodelle aus definierten Eingabemodellen zu erzeugen.

ATL ist eine hybride Sprache, die sowohl deklaratives als auch imperatives Programmieren unterstützt. Da es sich bei ATL jedoch um eine Transformationssprache handelt, sollten bevorzugt die deklarativen Sprachbestandteile eingesetzt werden, da sich dadurch Abbildungsregeln sehr übersichtlich und strukturiert darstellen lassen. Für die Fälle, in denen keine deklarativen Abbildungsregeln definiert werden können, stellt ATL imperative Sprachkonstrukte zur Verfügung.

Die einzelnen Transformationen für ein Modell werden in ATL in Form von Modulen gekapselt. Neben den Modulen unterstützt ATL auch die Entwicklung von unabhängigen Bibliotheken, die wiederum in ATL Module oder aber auch in andere Bibliotheken eingebunden werden können. Sowohl Module als auch Bibliotheken werden in Dateien mit der Endung .atl abgelegt. Nachfolgend sollen die Module etwas genauer vorgestellt werden.

ATL-Modul

Ein ATL-Modul kann mit einer Modell zu Modell Transformation gleichgesetzt werden. Sowohl die Eingabe- als auch das Ausgabemodelle müssen in einem ATL-Modul durch ihre Metamodelle typisiert werden. Die Anzahl der Eingabe- und Ausgabemodelle eines Moduls müssen im Vorfeld festgelegt werden. Ein ATL-Modul ist dabei folgendermaßen aufgebaut:

- *Header Sektion* - In diesem Bereich werden verschiedene Attribute sowie die Eingabe- und Ausgabemodelle definiert.
- *Import Sektion* - In diesem Bereich können optional zu verwendende ATL-Bibliotheken angegeben werden.
- *Helper* - Reihe von *Helpern*, die als eine Art von Methoden (ähnlich einer Java-Methode) angesehen werden können
- *Rules* - Reihe von Regeln, die definieren, wie Elemente des Eingabemodells auf Elemente des Ausgabemodells abgebildet werden sollen

Die Helper und Rules gehören zu keinem speziellen Bereich innerhalb eines Moduls. Sie können deshalb unter Berücksichtigung einiger Bedingungen beliebig innerhalb des Moduls definiert werden.

Header Sektion Im Header Bereich werden neben dem Modulnamen sowohl die Eingabe- als auch die Ausgabemodelle definiert. In Listing 7.1 ist ein Header Beispiel dargestellt.

```
1 module    InputModel2OutputModel
2 create    OUT    :  OutputMetamodel
3 from      IN     :  InputMetamodel;
```

Listing 7.1: Beispiel Header Sektion eines ATL-Moduls

Das Schlüsselwort `module` leitet den Modulnamen ein. Dieser Name muss mit dem Dateinamen übereinstimmen. Die Ausgabemodell-Definition wird durch das Schlüsselwort `create` eingeleitet. Das Eingabemodell wird durch das Schlüsselwort `from` deklariert. Die Ein- und Ausgabemodelle werden in der Form

$$Modellname : Metamodellname$$

annotiert. Es können mehrere Ein- und Ausgabemodelle definiert werden. Diese werden dann durch ein Komma voneinander getrennt.

Import Sektion In dieser optionale Sektion können benötigte Import deklariert werden. Die Import werden dabei folgendermaßen deklariert:

```
1 uses library1;
2 uses library2;
```

Listing 7.2: Beispiel Import Sektion eines ATL-Moduls

Helper Helper sind die Methoden innerhalb eines ATL-Moduls. Sie ähneln dabei Methoden in Java. Helper können an verschiedenen Stellen einer ATL-Transformation aufgerufen werden. Ein Helper setzt sich dabei aus folgenden Bestandteilen zusammen:

- *Kontexttyp* — gibt an in welchem Kontext die Methode verwendet werden soll (analog zum Klassenkontext einer objektorientierten Sprache)
- *Name* — Bezeichner des Helpers
- *Parameter* — Helper können optional mit Parameter versehen werden. Die Struktur eines Parameters ist hierbei $ParameterName : ParameterTyp$.
- *Rückgabetyp* — gibt an, welchen Rückgabetyp ein Helper hat. Alle Helper müssen über einen Rückgabetyp verfügen.
- *Code* — Die eigentliche Funktionalität des Helpers.

In Listing 7.3 ist ein Beispiel Helper dargestellt. Durch das Schlüsselwort `helper` wird der Helper deklariert.

```
1 helper context Integer def : add(value : Integer)
2    : Integer = self + value;
```

Listing 7.3: Beispiel Helper

Mit dem Schlüsselword `context` wird der Kontext angegeben, der in diesem Fall `Integer` ist. An dieser Stelle kann aber auch ein beliebiges Element eines Ein- oder Ausgabemetamodells angegeben werden. Nach dem Schlüsselwort `def` steht der Name gefolgt von den Parametern des Helpers. Vor dem eigentlichen Code des Helpers ist noch der Rückgabewert — in diesem Fall `Integer` — angegeben. Innerhalb eines Helpers wird das Schlüsselwort `self` zum Referenzieren des Kontexts verwendet (analog des `this`-Pointers).

Rules In ATL werden zwei verschiedene Arten von Regeln unterschieden. Zum einen sind dies die deklarativen *matched rules* und zum anderen die imperativen *called rules*.

Matched rules Die matched rules bilden den Kern einer ATL-Transformation. Anhand dieser Regeln kann deklariert werden, für welche Eingabeelemente welche Ausgabeelemente erzeugt werden sollen. Zusätzlich bietet diese Art der Regel die Möglichkeit, die Ausgabeelemente mit Werten (beispielsweise mit Werten des Eingabemodells) zu initialisieren. Eine matched rule wird über ihren Namen identifiziert und hat einen Typ der mit einem Typ des Eingabemodells übereinstimmt. Eine matched rule wird durch das Schlüsselwort `rule` eingeleitet. Des Weiteren verfügt sie über zwei verpflichtende und zwei optionale Sektionen. Zu den verpflichtenden Sektionen gehören die beiden Bereiche, in denen Ein- und Ausgabeelemente angegeben werden und zu den optionalen Sektionen gehören die Deklaration von lokalen Variablen sowie der imperative Bereich. In Listing 7.4 ist eine Beispielregel angegeben. Diese Regel erzeugt aus allen Features, die in ihrem Namen ein „a“ haben ein Team und weist diesem den Namen und eine Reihe von anderen Features zu.

```
1 rule Feature2Team{
2    from   s: UFM!Feature(s.name.indexOf('a') >= 0)
3    to     t: OT!Team(
4                 name <- s.name,
5                 ownedRole <- s.getChildFeatureSet())
6 }
```

Listing 7.4: Beispiel matched rule

Die Regel ist dabei wie folgt aufgebaut: Hinter dem Schlüsselwort `rule` steht der Bezeichner der Regel (`Feature2Team`). Daran schließt sich die Deklaration des Ein- und Ausgabeelements an. Das Eingabeelement wird durch das Schlüsselwort `from` und das Ausgabeelement durch das Schlüsselwort `to` gekennzeichnet, darauf folgt der Bezeichner und der Typ des Ein- bzw. Ausgabeelements. Der Typ setzt sich aus dem Metamodell und einem dort enthaltenen Element zusammen. In Listing 7.4 wird als Eingabeelement zum Beispiel ein *Feature* aus dem Metamodell UFM erwartet. Das bedeutet in diesem konkreten Fall, dass die Regel automatisch über den *matching Prozess* für alle Elemente vom Typ *Feature* des Eingabemodells aufgerufen wird. Um Regeln nur unter bestimmten Bedingungen auf einem Eingabeelement anzuwenden, kann ein Eingabeelement zusätzlich mit Bedingungen versehen werden. Die Regel wird in diesem Fall nur dann aufgerufen, wenn die angegebenen Bedingungen wahr sind. Die Bedingungen für die Eingabeelemente werden in Klammern hinter der Deklaration angegeben (zum Beispiel `s.name.indexOf('a') >= 0`). Für die Ausgabemodelle hingegen können Zuweisungen deklariert werden. So wird im Beispiel der konkreten Regel das Feld `name` des Eingabelements dem Feld `name` des Ausgabeelements zugewiesen. Innerhalb der Zuweisung können auch definierte Helper aufgerufen werden, wie in diesem Beispiel der Helper `getChildFeatureSet`.

Called rules Diese Art der Regeln werden für imperative Konstrukte verwendet und müssen im Gegensatz zu matched rules explizit aufgerufen werden. Sie haben deshalb große Ähnlichkeiten mit einem Helper. Im Gegensatz zu diesen können called rules allerdings Ausgabeelemente definieren. Called rules haben keinen Bereich für das Eingabeelement. Auch das Ausgabeelement ist nur optional, da diese Art von Regel nicht für bestimmte Eingabeelemente automatisch aufgerufen wird und sie auch nicht zwangsweise Ausgabeelemente erzeugt. Da dieser Regeltyp im Rahmen der Diplomarbeit nicht verwendet wurde, wird er an dieser Stelle auch nicht näher erläutert. Für genaue Informationen sei auf die weiterführende Literatur ([ATL06]) verwiesen.

Ausführen von Transformationen

Sobald eine Transformation in Form eines Moduls implementiert wurde, wird diese in ein ausführbares Kompilat umgewandelt. Das Kompilat ist ein Modell im XML-Format. Die Transformation kann dann auf zwei verschiedene Arten ausgeführt werden. Die erste Möglichkeit ist die Ausführung über eine *Eclipse Run Configuration*, die bequem über den *Run Dialog* der jeweiligen Eclipse-Installation erstellt werden kann. Die andere Möglichkeit ist die programmatische Ausführung einer Transformation. Hierfür stellt ATL eine *API (Application Programming Interface)* zur Verfügung mittels der Modelle transformiert werden können.

7.1.3 Eclipse Plugin Entwicklung

Eclipse ist eine erweiterbare Plattform, mit deren Hilfe *Integrated Development Environments (IDE)* erstellt werden können. Um Erweiterungen in Eclipse umzusetzen, stellt die Plattform das Konzept der *Eclipse Plugins* zur Verfügung. Durch dieses Konzept können Plugins die bestehende Plattform beliebig erweitern. Das Eclipse-Framework stellt dafür verschiedene Strukturen zur Verfügung, mit deren Hilfe Plugins in die bestehende IDE integriert werden können. Die eigentliche Integration eines Plugins in eine bestehende Eclipse-Instanz, sozusagen das *Deployment*, ist dabei relativ einfach. Hierfür müssen nur die entsprechenden Dateien in das *plugin*-Verzeichnis der Eclipse-Installation kopiert werden. Ein einzelnes Plugin kann dabei in zwei verschiedene Beziehungen mit anderen Plugins treten:

- *Dependency - Abhängigkeit* - In dieser Form der Beziehung gibt es ein *erforderliches* und ein *abhängiges* Plugin. Das abhängige Plugin erfordert in dieser Konstellation die Funktionalitäten des erforderlichen Plugins.
- *Extension - Erweiterung* - In dieser Konstellation gibt es ein *Host*- und ein *Extender*-Plugin[1], wobei das Extender-Plugin das Host-Plugin um verschiedene Funktionalitäten erweitert.

Um die Arten von Beziehungen zu unterscheiden, werden diese bei der Plugin Definition explizit festgelegt.

Extender-Plugin

Da im Rahmen dieser Diplomarbeit ein Plugin implementiert wurde, welches die Eclipse-UI (Plugin `org.eclipse.ui`) an einer Stelle erweitert, soll das Konzept eines Extender-Plugins nachfolgend kurz erläutert werden. Für genaue Informationen zur Eclipse Plugin-Entwicklung sei auf die weiterführende Literatur (zum Beispiel [Aza03]) verwiesen.
Für die Erweiterung eines bestehenden Eclipse-Plugins stellt Eclipse so genannte *Extension Points* zur Verfügung. Hierbei handelt es sich um frei definierbare Schnittstellen, die vom Host-Plugin definiert werden. Über diese Extension Points können dann Erweiterungen vorgenommen werden. Die Eclipse-UI stellt an dieser Stelle verschiedene Extension-Points zur Verfügung mittels derer das User Interface bis zu einem bestimmten Grad angepasst und erweitert werden kann. Zum Beispiel können in die bestehenden Kontextmenüs beliebige Einträge eingefügt werden, an die dann wiederum Aktionen gekoppelt sind. Um einen bestehenden Extension Point zu erweitern, muss das Extender-Plugin die vorgegebene Schnittstelle implementieren, dies erfolgt in der Regel über ein zuvor vom Host-Plugin

[1] Im Rahmen dieser Diplomarbeit wird der Begriff *Extender-Plugin* (in Anlehnung an [Aza03]) für Plugins verwendet, die ein anderes Plugin erweitern

definiertes Interface. Wird nun die das Extender-Plugin betreffende Aktion ausgeführt (zum Beispiel der Auswahl eines Menüeintrags im Kontextmenü), wird über die definierte Schnittstelle das angebundene Extender-Plugin aufgerufen. Dort können dann die für die Aktion definierten Funktionalitäten ausgeführt werden.

7.2 Implementierung der Transformationsregeln

In diesem Abschnitt soll die technische Umsetzung der Transformationsregeln erläutert werden. Hierfür wird im nächsten Abschnitt ein Gesamtüberblick über die verwendete Architektur gegeben, bevor im Anschluß die einzelnen Bereiche der Umsetzung näher beleuchtet werden.

7.2.1 Gesamtüberblick

Die Implementierung des Transformationsprozesses erfolgt in Form eines Eclipse-Plugins. Das entwickelte Plugin trägt den Namen *FTA (Features to Aspects)*. Das zu Grunde liegende Programm führt eine *model to model*-Transformation durch. Das bedeutet es bekommt ein Feature-Modell als Eingabe und erzeugt daraus ein *OTUML*[2] *(Object Teams UML)* konformes Modell. Das Plugin unterteilt sich dabei in insgesamt vier Pakete:

- *de.diploma.fta.plugin.actions* - beinhaltet die eigentliche Aktion des Plugins
- *de.diploma.fta.plugin.transformation* - beinhaltet die Transformation in Form eines ATL-Moduls
- *de.diploma.fta.plugin.model* - beinhaltet das Ein- und Ausgabemetamodell für die Transformation
- *de.diploma.fta.plugin.atl* - beinhaltet die Klasse zu programmatischen Ausführung einer Transformation

Nachdem das Plugin[3] in Eclipse installiert wurde, kann nach Auswahl einer kompatiblen Modell-Datei (mit .ufmIO Endung) aus dem Kontextmenü, über den Eintrag *Features to Aspects*, die Transformation angestoßen werden (siehe Abbildung 7.1). Das Plugin erzeugt dann — sofern es sich um ein gültiges Eingabemodell handelt — das passende Ausgabemodell im *Root*-Verzeichnis des aktuellen Projekts. Die Realisierung der eigentlichen Transformation erfolgt über ein ATL-Modul (siehe Abschnitt 7.1.2). Die Transformation verwendet als Eingabemodell ein mit dem Editor *IO* (siehe Abschnitt 7.2.2) erzeugtes Feature-Modell und erzeugt daraus ein

[2] OTUML basiert auf *UFA (UML for Aspects)* (siehe auch [Her02a])

[3] Eine aktuelle Version des Plugins kann über die Eclipse-Updatesite http://cs.tu-berlin.de/~marfei/fta/update bezogen werden

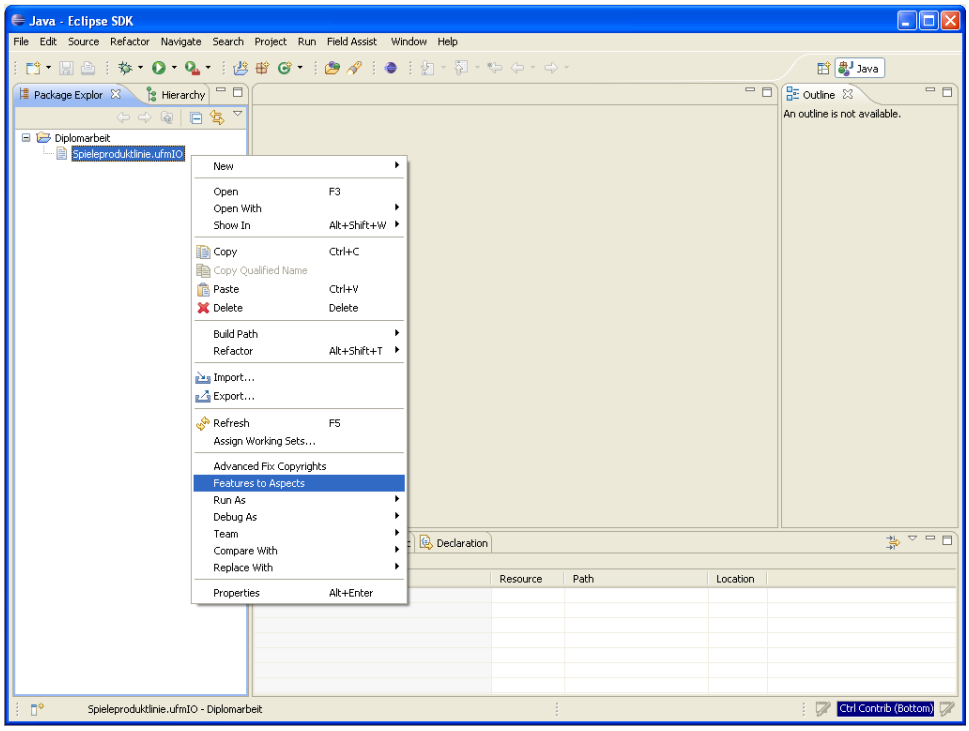

Abbildung 7.1: Plugin-Darstellung im Kontextmenü

OTUML konformes *OTuml*-Modell (siehe Abschnitt 7.2.3). Die eigentlichen Modelle und dessen zugehörige Metamodelle werden in den darauf folgenden Abschnitten noch einmal genauer beschrieben. In Abbildung 7.2 ist die Transformation mittels des Plugins schematisch dargestellt.

7.2.2 Feature-Modell (Eingabemodell)

In diesem Abschnitt wird zunächst das dem Feature-Modell zu Grunde liegende Metamodell vorgestellt. Im Anschluß soll der für die Erstellung der Modelle verwendete Editor *IO* erläutert werden. Zuletzt wird noch einmal auf das zum vorgestellten Metamodell konforme bereits aus der Fallstudie bekannte Feature-Modell eingegangen.

Eingabe-Metamodell

Bei dem verwendeten Metamodell handelt es sich um ein Modell mittels dessen Feature-Modelle konstruiert werden können. Das Metamodell wurde von Mark-Oliver Reiser im Rahmen des Feature-Modell-Editors *IO* entwickelt (siehe [Rei]).

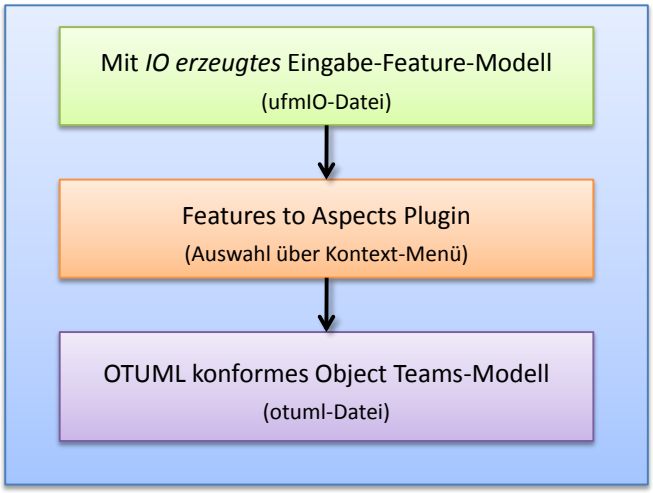

Abbildung 7.2: Schematische Darstellung der Transformation

Eine Modifikation des bestehenden Metamodells fand im Rahmen dieser Arbeit nicht statt. Da das Metamodell neben den für die Arbeit relevanten auch verschiedene Bestandteile enthält, die für die Transformation nicht interessant sind, sollen nachfolgend nur die verwendeten Bestandteile erläutert werden. Das komplette Metamodell ist auf der beiliegenden CD enthalten. In Abbildung 7.3 ist der benötigte Ausschnitt des Metamodells dargestellt, der nachfolgend vorgestellt werden soll. Die Grundlage eines Feature-Modells bildet die Entität *FeatureModel*. Die einzelnen Features werden durch die Entität *Feature* repräsentiert. Diese Entität ist eine Spezialisierung der Entität *FeatureTreeNode*. Die Entität *FeatureTreeNode* ordnet Subfeatures mittels der *childNodes* hierarchisch an. Die Verbindung der einzelnen Features, wie sie zum Beispiel für die *«influence»*-Beziehung benötigt werden, ist durch die Entität *FeatureLink* dargestellt. An dieser Stelle wurden ebenfalls einige der Assoziationen zwischen den Entitäten aus Gründen der Übersichtlichkeit weggelassen. Jedoch verfügt die Entität *FeatureLink* im Normalfall über verschiedene Referenzen zum eigentlichen *Feature*, über die die notwendigen Beziehungen abgebildet werden können. Die *FeatureGroup* ermöglicht das Gruppieren von Features und ist ebenfalls eine Spezialisierung des *FeatureTreeNode*, dadurch wird impliziert, dass ein *Feature* und *FeatureGroups* in einer Eltern-Kind Beziehung stehen können. Aufgrund der Verbindung zwischen den Entitäten *FeatureGroup* und *Feature* können auch *FeatureGroups* beliebige *Features* als Kinder haben. Alle bisher vorgestellten Elemente stellen Spezialisierungen der Entität *VariabilityModelElement* dar. Diese Entität ist wiederum mit zwei weiteren Entitäten *UserAttribute-*

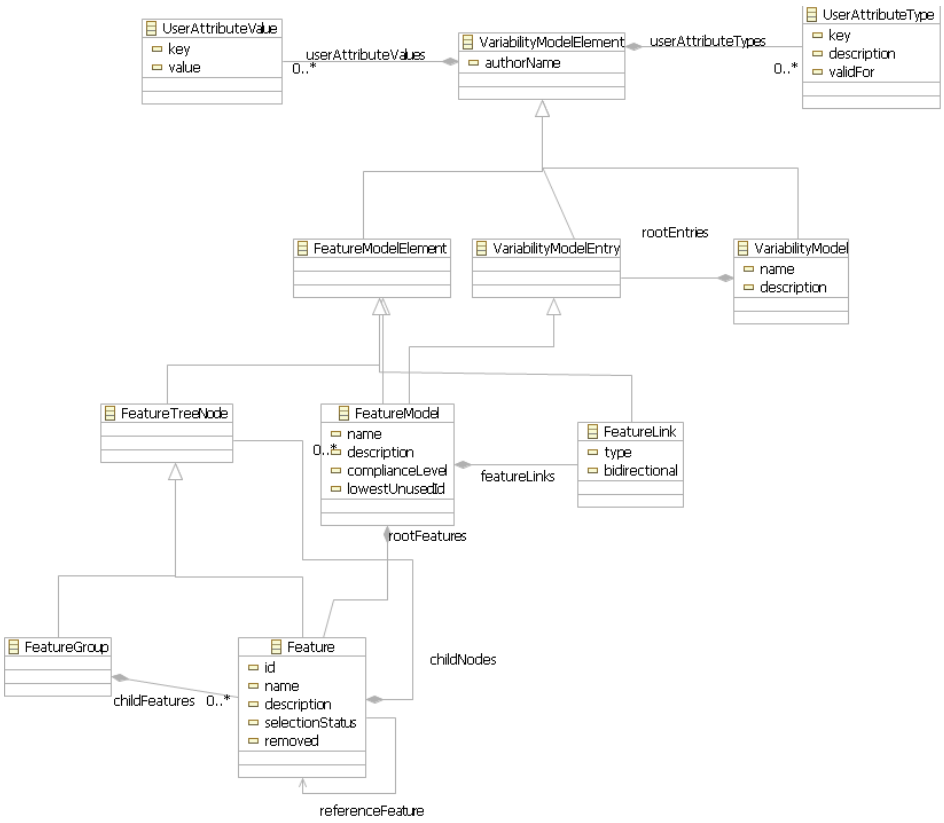

Abbildung 7.3: Ausschnitt des *IO*-Metamodells

Value und *UserAttributeType* verknüpft. Über diese Entitäten können für beliebige Elemente des Feature-Modells benutzerdefinierte Attribute erfasst werden.

Verwendetes Modellierungstool

Der *Unified Feature Model Editor (IO)* ist ein von Mark-Oliver Reiser im Rahmen seiner Dissertation[4] entwickelter Variabilitätsmodell-Editor. Der Editor wird im Rahmen dieser Diplomarbeit verwendet, um die in der technischen Umsetzung verwendeten Feature-Modelle zu erstellen. IO ist ein Eclipse Plugin, das auf EMF-Basis entwickelt wurde. Es unterstützt sowohl die grafische als auch die tabellarische Bearbeitung von Feature-Modellen. Der Editor unterstützt neben Feature-Modellen auch noch weitere Modelltypen, die aber im Rahmen dieser Diplomarbeit nicht relevant sind und deshalb nicht näher betrachtet werden. In Abbildung 7.4 ist die Entwicklungsumgebung des Features-Editors dargestellt. Jedes Feature-Modell

[4] Für weiterführende Informationen siehe [Rei]

ist Bestandteil eines Variabilitätsmodells, welches auch noch weitere Modelltypen enthalten kann. Das eigentliche Feature-Modell setzt sich aus den zugehörigen Features und einem Diagramm zur grafischen Darstellung zusammen.

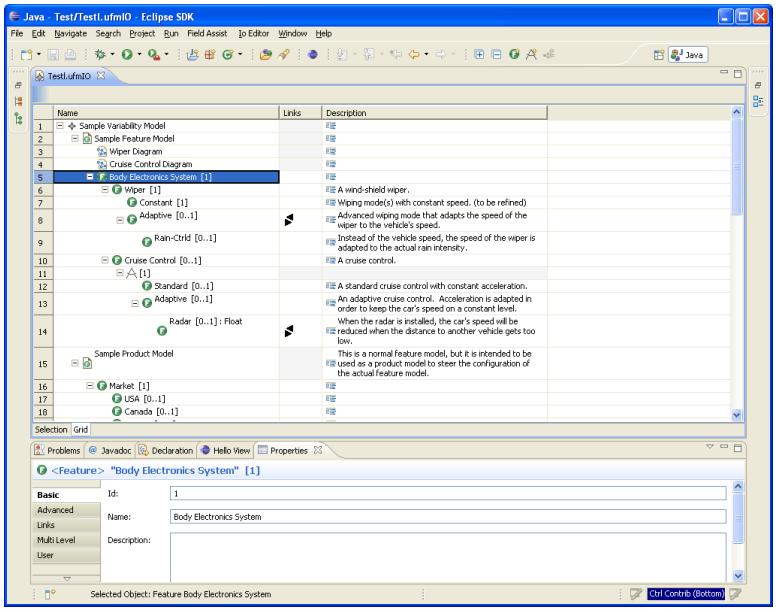

Abbildung 7.4: Tabellarischer Editor im IO

Anlegen von Feature-Modellen und Features Um ein neues Feature-Modell anzulegen, wählt man die entsprechende Funktion aus dem Kontextmenü. Im Anschluss kann man über eine *Eigenschaftsseite* die Grundattribute für das Feature-Modell setzen. Ist das Feature-Modell erstellt, können über das Kontextmenü neue Features oder Gruppen zum erzeugten Feature-Modell hinzugefügt werden. Hierfür muss nur das Superfeature des zu erzeugenden Features oder der zu erzeugenden Feature-Gruppe selektiert werden. Über einen *Rechts-Klick* gelangt man in das Kontextmenü und kann über die Funktionen *New Child -> New Feature* oder *New Child -> New Feature Group* ein neues Feature beziehungsweise eine neue Feature-Gruppe erzeugen. Dieses Element wird dann gemäß des zuvor selektierten Features in das Feature-Modell einsortiert. Für jedes Feature und jede Feature-Gruppe existiert ebenfalls eine Eigenschaftsseite, über welche die Grund- und Benutzer-Attribute definiert werden können. In Abbildung 7.5 ist die Eigenschaftsseite für ein Feature dargestellt. Neben den Basis-Eigenschaften (*Basic*) können auch erweiterte Eigenschaften festgelegt und Abhängigkeiten (*Links*) zwischen verschiedenen Features definiert werden. Benutzerdefinierte Attribute können über den Tab *User* zugewiesen werden.

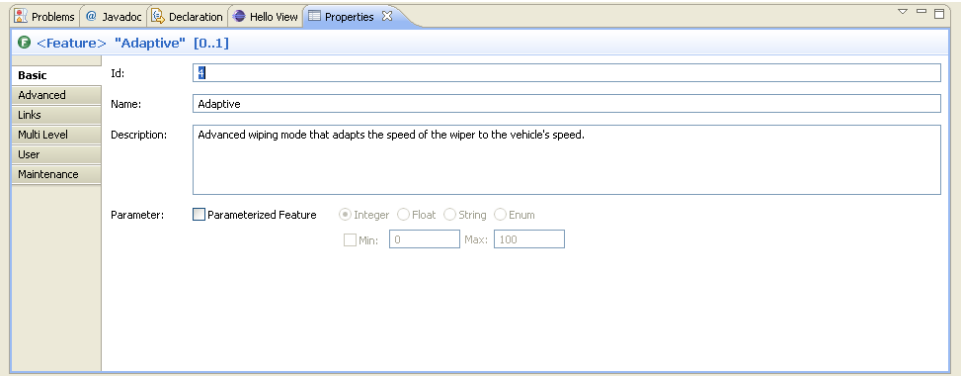

Abbildung 7.5: Eigenschaftsseite für ein Feature

Setzen der Kardinalitäten Wurde ein Feature oder eine Feature-Gruppe angelegt, muss noch der Typ festgelegt werden. Da es sich bei diesem Modell um ein kardinalitätsbasiertes Feature-Modell handelt, wird der Typ mit Hilfe der Kardinalitäten ausgedrückt. Um einem Feature oder einer Feature-Gruppe eine Kardinalität zuzuweisen, muss das entsprechende Element ausgewählt und das Kontextmenü aufgerufen werden. Unter dem Menüpunkt *Set cardinality* kann neben den Standardkardinalitäten ($[0..1]$, $[1]$, $[0..*]$) auch eine frei definierbare Kardinalität angegeben werden.

Definition von Kompositionsregeln In Abschnitt 2.2.1 wurden Kompositionsregeln für Features vorgestellt. Es handelt sich hierbei um Relationen, die unabhängig von der Feature-Hierarchie angelegt werden können. Für diesen Typ von Feature-Beziehung gibt es im IO die allgemeine Relation *Link*. Diese Relation erwartet zwei Features und einen Relationstyp, der frei definierbar ist. Zusätzlich kann für diese Relation bestimmt werden, ob sie einseitig oder bidirektional ist. Um eine Relation anzulegen, müssen die zu verbindenden Features selektiert werden. Standardmäßig ist die Relation einseitig. Um eine bidirektionale Abhängigkeit zu definieren, muss auf der Eigenschaftsseite für den Link nur die entsprechende Option ausgewählt werden. Für die aus 2.2.1 bekannten Kompositionsregeln *Gerichtete Abhängigkeit* und *Gegenseitiger Ausschluss* werden im IO die *Linktypen includes* und *excludes* verwendet.

Meta-Attribute Zusätzlich zu den Standardattributen für die einzelnen Elementtypen im IO, können des Weiteren noch beliebige Attribute frei definiert werden. Hierfür gibt es auf allen Ebenen die Möglichkeit, so genannte *User-Attributes* zu definieren. In Abbildung 7.6 ist der Dialog zum Anlegen von Benutzerattributen dargestellt. Neben der Bezeichnung des Attributes muss noch der Gültigkeitsbe-

reich und ein Datentyp selektiert werden. Als Datentypen stehen drei Standardda-

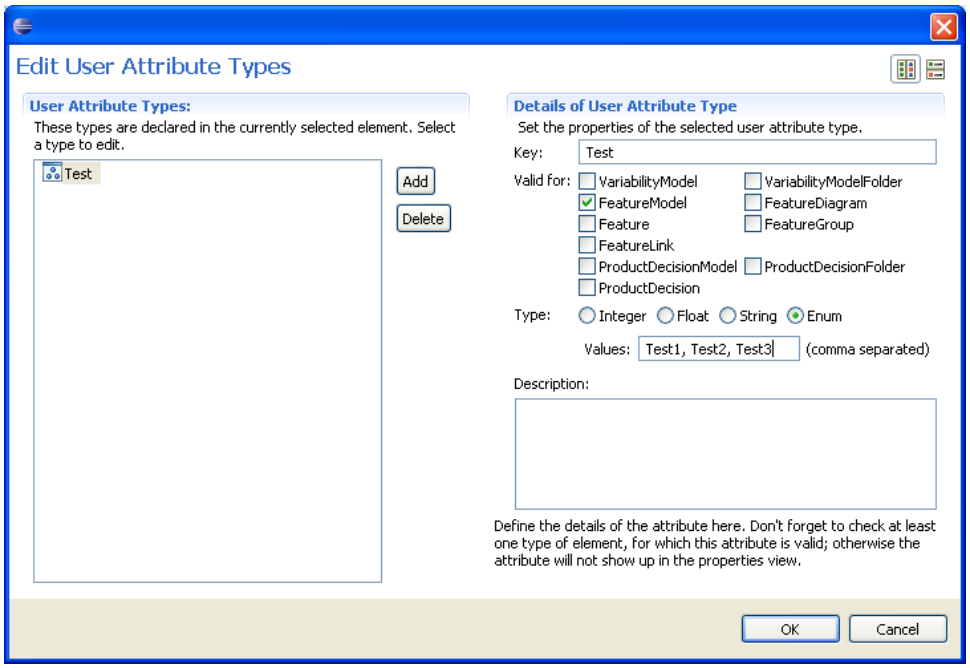

Abbildung 7.6: Anlegen eines User-Attributes in IO

tentypen und ein Aufzählungsdatentyp zur Verfügung. Sobald ein Attribut definiert wurde, kann im Anschluß für alle Elemente, die in den definierten Gültigkeitsbereich fallen, ein Wert für das Attribut selektiert werden. Die Zuweisung erfolgt in diesem Fall auch über den *User* Tab.

Feature-Modell

In diesem Abschnitt soll nun die auf dem vorgestellten Metamodell basierende Umsetzung des in der Fallstudie vorgestellten Feature-Modells erläutert werden. Da sich der Editor momentan in der Weiterentwicklung befindet und im Rahmen dieser Diplomarbeit nicht modifiziert wurde stehen einige der in Kapitel 4 vorgestellten Modellierungselemente nicht zur Verfügung. Die fehlenden Elemente können jedoch größtenteils über die Meta-Attribute des IO-Editors abgebildet werden und sollen nachfolgend kurz vorgestellt werden.

Aggregation und Generalisierung In Abschnitt 4.1.1 wurde die Unterscheidung zwischen einer Aggregation und einer Generalisierung vorgestellt. Diese sollten innerhalb eines Feature-Diagramms durch eine gestrichelte beziehungsweise durchgezogene Linie unterschieden werden. Da dieses Modellierungsmittel nicht durch

das vorhandene Metamodell abgedeckt wird, müssen die Relationen anders unterschieden werden. Um die Relationen unterscheiden zu können, wurde ein Meta-Attribut (siehe Abschnitt 7.2.2) mit der Bezeichnung *Relationstyp* eingeführt, das die Auswahlmöglichkeiten *Gen/Spec* und *Aggregation* hat. Dieses Attribut kann nur Feature-Gruppen zugewiesen werden. Will man also eine Generalisierung/-Spezialisierung darstellen, so muss eine Feature-Gruppe erstellt werden, bei der das Attribut auf *Gen/Spec* gesetzt wird. Alle anderen Relationen innerhalb der Feature-Hierarchie, die nicht explizit als Generalisierung/Spezialisierung gekennzeichnet sind, werden als Aggregationen betrachtet.

Attributierte Features In Abschnitt 2.2.3 wurden die attributierten Features vorgestellt. Bei diesen Features handelt es sich um solche für die verschiedene Attribute definiert werden können. Auch diese Modifikation wird über Features und Meta-Attribute abgebildet. Jedes Feature, das Attribute enthält, bekommt, entsprechend den definierten Attributen, Subfeatures zugeordnet, welche die eigentlichen Attribute darstellen. Die Subfeatures haben ein besonderes Meta-Attribut, dass sie als Attribute kennzeichnet. Dieses Attribut trägt den Namen *Parameter*. Es kann die Werte *Integer*, *Boolean*, *String* und *Float* annehmen, wodurch gleichzeitig der Datentyp festgelegt wird. Ist für ein Feature also eines der Werte gesetzt, handelt es sich um ein Attribut des jeweiligen Superfeatures.

Elementar-Features In Abschnitt 4.2.1 wurden die Elementar-Features vorgestellt. Für diese Art von Features wurde des Weiteren in Abschnitt 5.3.2 eine spezielle Transformationsregel definiert. Um ein Feature nun als Elementar-Feature kennzeichnen zu können, wurde ein weiteres Meta-Attribut eingeführt. Das Attribut trägt den Namen *Elementar* und hat den Typ *Boolean*. Für alle Features, die ein Elementar-Feature darstellen, muss das Attribut auf *True* gestellt werden. Alle anderen Features werden implizit als normale Features betrachtet, wenn sie nicht mit einem anderen Meta-Attribut gekennzeichnet sind.

Observer-Pattern In Abschnitt 4.2.2 wurde die Observer-Pattern Erweiterung vorgestellt. Diese ermöglicht es verschiedene Features als Subjekte oder Beobachter zu kennzeichnen, um dadurch das entsprechende Pattern anwenden zu können. Für die Kennzeichnung als Subjekt beziehungsweise Beobachter wurden zwei weitere Meta-Attribute eingefügt, zum einen das Attribut *Subjekt* und zum anderen das Attribut *Beobachter*. Beide Attribute sind vom Typ *Boolean* und können somit nur zwei verschiedene Zustände einnehmen. Steht eines der Attribute für ein Feature auf *True*, so wird dieses Feature, je nach dem welches der Attribute selektiert wurde, als Subjekt oder Beobachter angesehen. Des Weiteren gibt es für die Transformation des Observer-Patterns auch noch eine zusätzliche Verbindung,

die den Beobachtern Subjekte zuordnet. Hierfür wurde eine spezielle Verbindung (*FeatureLink*) eingeführt, die den Namen *observe* trägt.

Feature-Modell der Spiele-Produktlinie

Nachfolgend wird nun die auf dem erläuterten Metamodell (siehe Abschnitt 7.2.2) basierende Repräsentation der Fallstudie kurz vorgestellt. Dafür ist in Abbildung 7.7 die tabellarische Darstellung des Feature-Modells abgebildet, da die Diagrammdarstellung im aktuellen Release von *IO* nicht verfügbar ist. Die wesentlichen Unterschiede gegenüber dem Feature-Modell aus der Fallstudie (siehe Abschnitt 6.2) wurden aber bereits im vorangegangenen Abschnitt (Abschnitt Feature-Modell) genau beleuchtet, so dass hier nur noch einmal auf die Sonderfälle eingegangen werden soll.
Das attributierte Feature *Feldgroesse* hat in diesem Modell zwei Subfeatures, welche die entsprechenden Attribute darstellen. An dieser Stelle wurde die Anpassung gemäß des letzten Abschnitts vorgenommen. Alle Gen/Spec-Features (zum Beispiel die Subfeatures von *Form*) sind über eine Feature-Gruppe gruppiert. Diese Feature-Gruppe ist mit dem Attribut *Relationstyp = Gen/Spec* versehen. Die beiden Features *Feldgroesse* und *Form* sind mit dem Attribut *Elementar = True* versehen, da es sich bei dieser Auswahl um Elementar-Features handelt. Für das Feature *Ansicht* wurde das Attribut *Beobachter* auf *True* gesetzt, da es sich bei diesem Feature um den *Beobachter (Observer)* handelt. Alle zugehörigen Subjekte wurden ebenfalls mit dem entsprechenden Attribut gekennzeichnet. Über die *observe*-Verbindungen sind alle Subjekt-Features mit dem Beobachter-Feature verbunden (erkennbar an den schwarzen Pfeilen im Editor). Eine komplette Version des Modells ist auf der CD enthalten (*Spieleproduktlinie.ufmIO*).

7.2.3 OTUML-Modell (Ausgabemodell)

In diesem Abschnitt wird das Ausgabemodell vorgestellt. Hierfür wird zunächst das zugehörige Metamodell erläutert. Im Anschluß folgt ein kurzer Blick auf ein für OTUML existierendes Modellierungstool, bevor im letzten Abschnitt das eigentliche Ausgabemodell vorgestellt wird.

OTUML-Metamodell

Bei dem Metamodell handelt es sich um das *OTUML (Object Teams UML)*-Metamodell, das eine Erweiterung des UML-Metamodells ist. Das OTUML-Metamodell wurde von Marco Mosconi im Rahmen des OTUML-Editors an der TU-Berlin entwickelt. Da das UML-Metamodell [Obj07b] [Obj07a] hinlänglich bekannt ist, sollen nachfolgend nur die zusätzlichen Eigenschaften des OTUML-Modells beschrieben werden. In Abbildung 7.8 sind die Erweiterungen des UML-Metamodells dar-

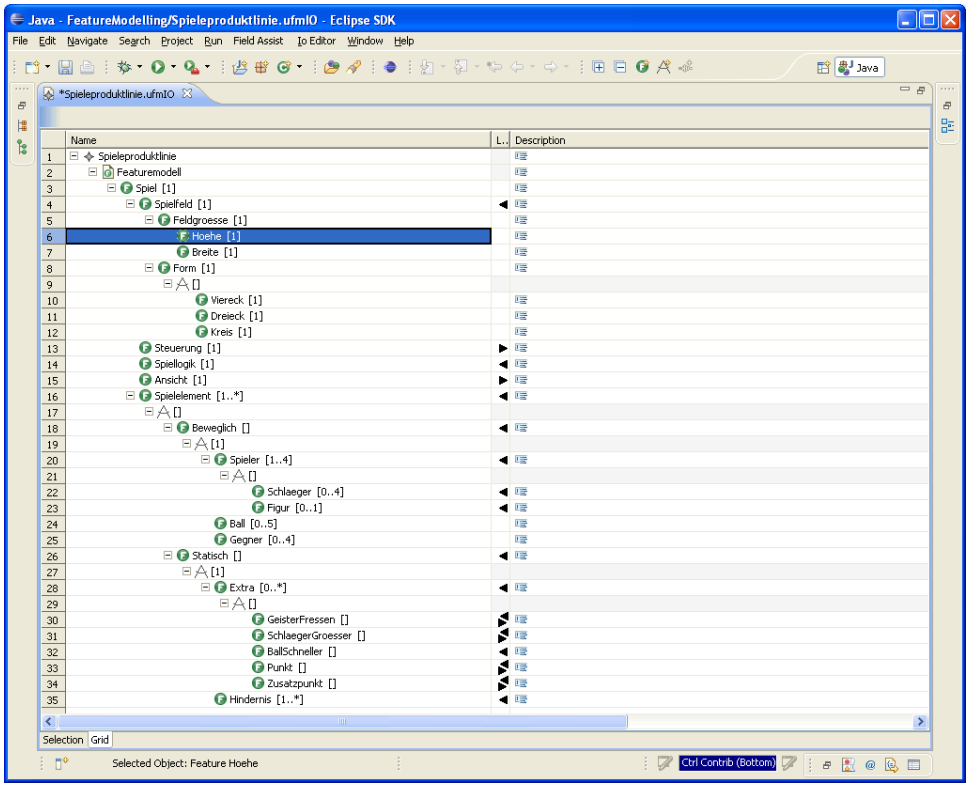

Abbildung 7.7: Editor mit Spiele-Produktlinie

gestellt. Auch in diesem Fall werden nur die für die Umsetzung relevanten Bestandteile des erweiterten UML-Metamodells beschrieben. Im Rahmen der Diplomarbeit sind vor allem die Entitäten *Team*, *Role*, *TeamRole* sowie die Entität *Classifier* interessant. Die Konstrukte *Team* und *Role* wurden ausführlich in Abschnitt 2.3.3 vorgestellt. Im Rahmen des Metamodells stellen die beiden Entitäten eine Spezialisierung der Entität *Class* dar. Die Entität *Team* kann eine beliebige Anzahl von Rollen über die Referenz *ownedRole* kapseln. Außerdem kann ein *Team* auch weitere *Teams* beinhalten, hierfür gibt es die zusätzliche Entität *TeamRole*. Diese Entität stellt eine Spezialisierung der Entitäten *Team* und *Role* dar. Die Adaption von Klassen, Teams und Rollen wird in OTUML über den *baseClassifier* abgebildet. Hierdurch kann die Entität *Classifier* adaptiert werden, die wiederum eine Generalisierung der Entitäten *Class*, *Role* und *Team* darstellt.

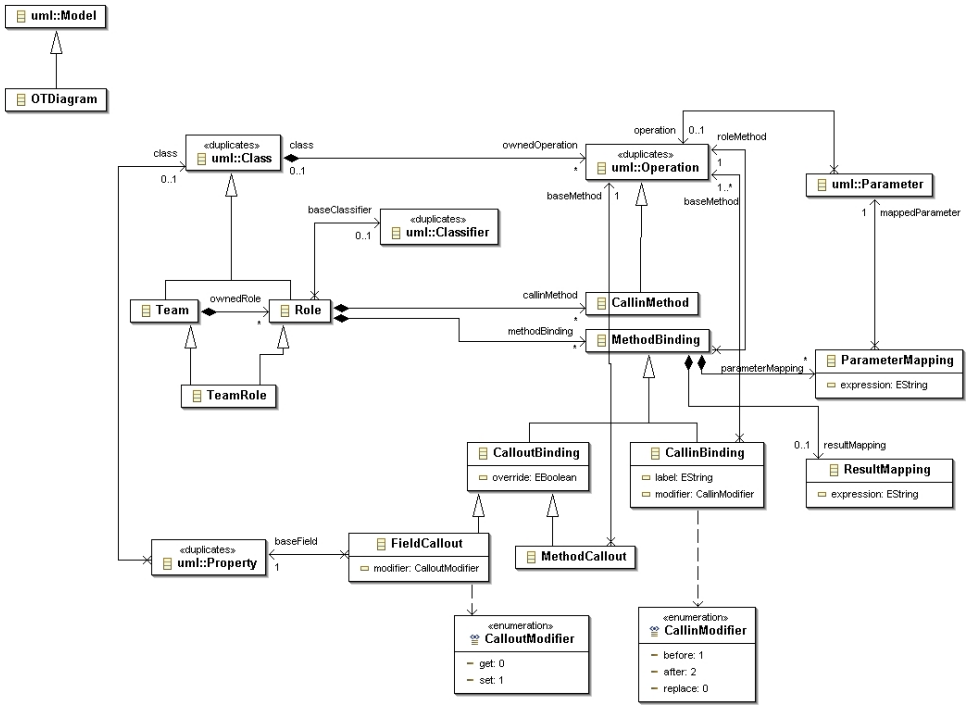

Abbildung 7.8: OTUML Erweiterungen des UML-Metamodells

OTUML-Modellierungstool

Für Darstellung und Modifikation des Ausgabemodells wurde im Rahmen dieser Diplomarbeit ein weiterer auf EMF basierender Editor verwendet. Der Editor wurde von Marco Mosconi an der TU-Berlin entwickelt und läuft momentan als Plugin unter Eclipse 3.2. Da sich der Editor zum aktuellen Zeitpunkt in der Überarbeitung befindet, sollen nur die wesentlichsten Bestandteile erläutert werden.
Der Editor basiert auf dem zuvor vorgestellten OTUML-Metamodell und wurde mit Hilfe des EMF-Frameworks implementiert. Der Editor kann neben dem Erstellen und Bearbeiten von OTUML-Modellen auch Diagramme sowie Object Teams Quellcode aus bestehenden Modellen erzeugen. Die Modelle können sowohl grafisch als auch in einer Baumstruktur erstellt und editiert werden.
Da das im Rahmen der Diplomarbeit erstellte Plugin (Features to Aspects — FTA) für die aktuellen Eclipse-Version, die UML in der Version 2.1.x verwendet, implementiert wurde und der Editor auf der UML Version 2.0.0 basiert, sind die erzeugten Modelle mit dieser Version des Editors inkompatibel. Durch eine einfache Mo-

difikation[5] an der Modelldatei kann das erzeugte Modell jedoch trotzdem über den bestehenden Editor betrachtet werden. In Abbildung 7.9 ist der Editor dargestellt. Über das Kontextmenü können beliebige Elemente hinzugefügt oder modifiziert werden. Eine aktuelle Version des Editors kann über eine Eclipse-Updatesite unter http://swt.cs.tu-berlin.de/ mosconi/otuml/update-site/ heruntergeladen werden.

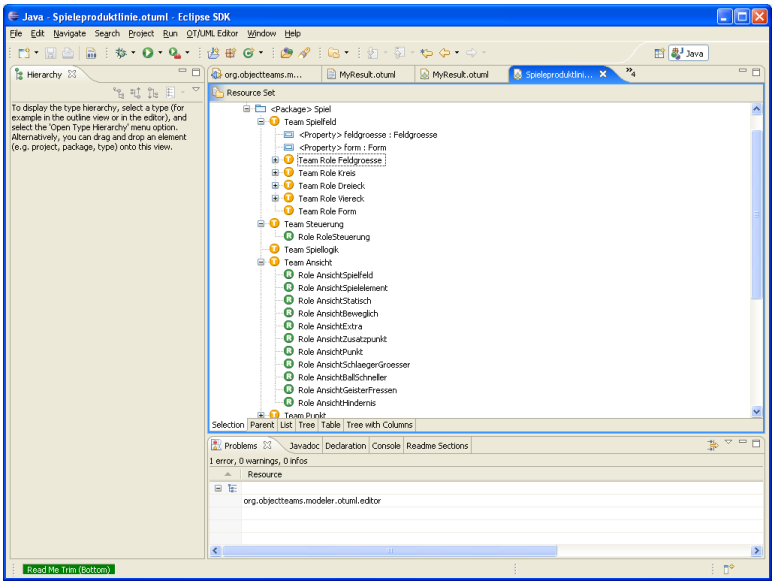

Abbildung 7.9: OTUML-Editor

OTUML-Modell der Spiele-Produktlinie

Obwohl das Modell eigentlich aus der im nächsten Abschnitt beschriebenen Transformation hervor geht, soll der chronologischen Reihenfolge trotzdem vorweg gegriffen und an dieser Stelle das resultierende Modell vorgestellt werden. Da die Struktur des resultierenden Modells mit dem Ergebnismodell der Fallstudie (siehe Abschnitt 6.3) weitgehend übereinstimmt, soll an dieser Stelle nur kurz darauf eingegangen werden. Das resultierende Modell basiert auf dem OTUML-Metamodell und repräsentiert das Gerüst einer Object Teams Applikation. In Abbildung 7.10 ist die Editor-Ansicht des resultierenden Modells dargestellt. Es wurden nur zum Teil die Bezeichner der Rollen und Teams angepasst und zusätzlich ein Modell-Element angelegt, dass das Paket *Spiel* umspannt. Im Gegensatz zur Darstellung der Fallstudie (Abschnitt 6.3) ist die Sonderregelung bezüglich der extra Rolle bei einer

[5] Um die mit FTA erzeugten Modelle im bestehenden OTUML-Editor zu betrachten, muss der UML Namensraum von http://www.eclipse.org/uml2/2.1.0/UML auf http://www.eclipse.org/uml2/2.**0**.0/UML angepasst werden.

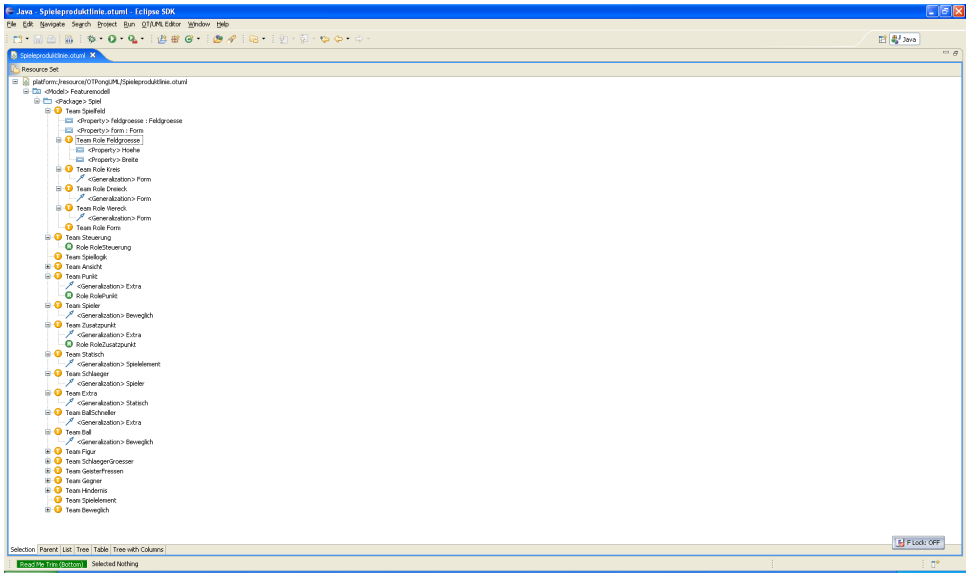

Abbildung 7.10: Editor Ansicht des resultierenden Modells

playedBy-Relation (siehe Abschnitt 5.2.2) in dem Modell berücksichtigt worden. Das Team *Steuerung* verfügt zum Beispiel über eine Rolle *RoleSteuerung*, durch die die Adaption erfolgt. Die attributierten Features wurden in Form von *Properties* auf die entsprechenden Teams und Rollen abgebildet. Auch die Elementar-Features finden sich mit den entsprechenden Typen im Modell wieder (zum Beispiel *Feldgroesse* im Team *Spielfeld*). Auch die Generalisierungen können in der Abbildung nachvollzogen werden. Hierfür wurden im Modell entsprechende Generalisierungsbeziehungen angelegt, die zum Beispiel am Team *Spieler*, das das Team *Beweglich* spezialisiert, nachvollzogen werden kann. Für eine genaue Beschreibung des resultierenden Modells der Spiele-Produktlinie sei auf das Kapitel 6 verwiesen. Des Weiteren ist eine vollständige Version des Modells auf der beiliegenden CD enthalten.

7.2.4 ATL-Transformation

Nachdem in den vorangegangenen Abschnitten sowohl die Technologien als auch die verwendeten Metamodelle genannt wurden, soll in diesem Abschnitt nun die eigentlichen Transformation vorgestellt werden. Die Transformation wurde in Form eines ATL-Moduls implementiert, die durch das entwickelte Plugin programmatisch aufgerufen wird. In Abbildung 7.11 ist noch einmal ein Überblick über die gesamte Transformation gegeben. Zum besseren Verständnis sind die einzelnen Modelle analog der Prozessschritte aus Abbildung 7.2 eingefärbt. In dieser Ab-

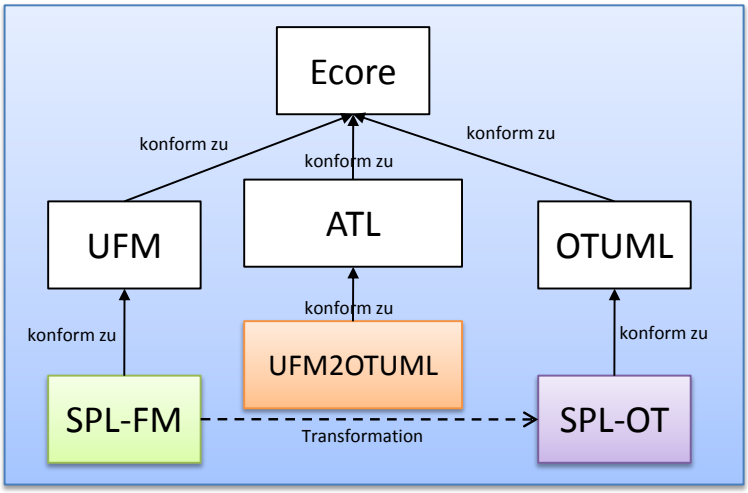

Abbildung 7.11: Der Transformationsprozess

bildung ist erkennbar, dass sich sowohl die Modelle als auch die Transformationen vom Ecore-Metametamodell ableiten. Das UFM-Metamodell ist hierbei das Eingabemetamodell und das OTUML-Metamodell das Ausgabemetamodell. Die eigentlichen Modelle *SPL-FM (Spiele-Produktlinie Feature-Modell)* und *SPL-OT (Spiele-Produktlinie Object Teams)* stellen dabei das Ein- und Ausgabemodell dar. Die Transformation wird durch das *UFM2OTUML*-Modell dargestellt, das wiederum vom *ATL*-Metamodell abgeleitet ist. Mittels dieser Transformation wird das SPL-FM Modell in das SPL-OTUML Modell transformiert. Nachfolgend soll die Umsetzung der Transformationsregeln strukturiert dargestellt werden. Dafür werden analog der in Kapitel 5 behandelten Transformationsregeln, die auf ATL basierenden Regeln vorgestellt. Hierbei werden die Regeln nur bis zu einem gewissen Detailgrad dargestellt, für das komplette ATL-Modul sei in diesem Zusammenhang auf den gut kommentierten Quellcode verwiesen. Zusätzlich wurde eine weitere Regel eingeführt, die das Element Feature-Modell in ein UML-Modell transformiert. Im nachfolgenden Abschnitt soll zunächst ein *Helper* vorgestellt werden, der für das Verständnis der Transformationen essentiell ist. Auf eine Vorstellung der anderen Helper soll verzichtet werden, da dies den Rahmen der Arbeit übersteigen würde. Sofern erforderlich erfolgt an den notwendigen Stellen eine kurze prosaische Erläuterung der verwendeten Helper.

Der `getChildFeatureSet`-Helper

Bei diesem Helper handelt es sich um eine imperative Methode, die alle zugehörigen Kinder eines *Features* ermittelt. Da an dieser Stelle verschiedene Sonderfälle berücksichtigt werden, wurde diese Aufgabe in einen separaten Helper ausgelagert. Folgende Fälle müssen unterschieden werden:

- alle in Form einer Aggregation gruppierten untergeordneten Features
- Features, die an einer dem Feature untergeordneten Gruppe hängen
- alle Spezialisierungen, die einem Feature in Form von Subfeatures zugeordnet sind

Der erste Fall ist leicht nachvollziehbar. Alle Subfeatures, die in einer Aggregation zu einem Superfeature stehen, gehören zu den Kindern, die innerhalb des zu erzeugenden Pakets oder Teams einsortiert werden müssen. Mit dem zweiten Fall sind alle Features gemeint, die in Form einer Feature-Gruppe angeordnet sind, die nicht als eine *Gen/Spec*-Gruppe (siehe Abschnitt 7.2.2) definiert wurde. Auch diese Features müssen in der späteren Form von Rollen oder Team-Rollen in das zugehörige übergeordnete Paket oder Team eingeordnet werden. Beim dritten Fall handelt es sich um einen Sonderfall, der ausschließlich bei der Gen/Spec-Relation auftritt. Auch an dieser Stelle muss eine Zuordnung zu einem übergeordneten Paket oder Team erfolgen. Das bedeutet, dass alle in einer Gen/Spec-Relation stehenden Features innerhalb der übergeordneten Aggregation eingeordnet werden müssen. In Richtung der Blätter bedeutet dies, dass alle in der Gen/Spec-Relation stehenden Features bis zur nächsten Aggregation zur übergeordneten Aggregation gehören. Hierfür wurde ein teilweise rekursiver Algorithmus entwickelt, der durch die Knoten und Blätter des Feature-Baums geht und die zu einem Feature zugehörigen Kinder ermittelt. In Listing 7.5 ist der entsprechende Helper dargestellt.

```
helper context UFM!Feature
   def : getChildFeatureSet() :
   Set(UFM!Feature) =
   self.childNodes->collect(childNode |
      if childNode.oclIsTypeOf(UFM!Feature) then
         if not childNode.isParameterFeature() then
            Set{childNode}.union(
               self.getGenFeatureSet(childNode))
         else
            Set{}
         endif
      else
         if childNode.oclIsTypeOf(
                     UFM!FeatureGroup) then
            if not childNode.isGenSpecGroup() then
               childNode.childFeatures->collect(
                     childFeature | childFeature)
            else
                  Set{}
            endif
         else
            Set{}
         endif
   endif)
   ->flatten().union(self.startingLinks);
```

Listing 7.5: Helper zum ermitteln der Kinder

Im Rahmen des Helpers werden zunächst alle `childNodes` über die `collect`-Methode iteriert. Handelt es sich bei dem Kindknoten um ein Feature, das kein attributiertes Feature ist, wird es in die Liste der Kind-Features einsortiert. Andernfalls wird eine leere Liste erzeugt. Bei einer Feature-Gruppe wird zunächst geprüft, ob es sich um eine Gen/Spec-Gruppe handelt, ist dies der Fall, wird ein leeres `Set` erzeugt. Anderfalls werden alle Subfeatures der Liste hinzugefügt (`collect(childFeature | childFeature)`). Der eigentliche Sonderfall (Fall 3), der bei einer Gen/Spec auftritt, ist bereits im ersten Teil des Helpers abgebildet. An dieser Stelle wird für jedes Feature der Helper `getGenFeatureSet` aufgerufen. Dieser Helper iteriert rekursiv alle Subfeatures bis zur nächsten Aggregation und fügt diese als Kinder hinzu. Da bei der Iteration ein `Set` von `Sets` erzeugt wird, wird am Ende des Helpers die Methode `flatten()` aufgerufen, die die einzelnen Sets zu einem vereint. Die Vereinigung mit den `startingLinks`

wird für die spätere Zuordnung der Subjekte zu den jeweiligen Beobachtern benötigt. Innerhalb des ATL-Moduls befindet sich eine kommentierte Version dieses Helpers.

Transformation des Feature-Modells

Bei der Transformation des Feature-Modells wird aus einem bestehenden Feature-Modell ein UML-Modell generiert. Hierfür wird die in Listing 7.6 abgebildete Transformationsregel verwendet.

```
1 rule UFMModel2OTModel {
2  from s : UFM!FeatureModel
3
4  to   t : OT!Model (
5             name <- s.name.format(),
6             packagedElement <- s.rootFeatures)
7 }
```

Listing 7.6: Modell-Transformation

Die Eingabe für diese Regel ist ein *FeatureModel*, welches auf ein *Model* abgebildet wird. Hierbei wird sowohl der Name als auch die Sammlung der Subfeatures übernommen.

Die Transformation der Aggregation

In Abschnitt 5.2.1 wurde die Transformationsregel für die Aggregation vorgestellt. In diesem Abschnitt wird die technische Umsetzung dieser Regel dargestellt. Für die Aggregation werden insgesamt drei Regeln benötigt, da an dieser Stelle zwischen insgesamt drei Fällen zu unterscheiden ist:

- das Wurzel-Feature
- Features der ersten Ebene
- alle restlichen Features

Die Abbildungsregeln verwenden alle die gleiche Eingabe, unterscheiden sich jedoch in der Ausgabe.

Das Wurzel-Feature Da es sich beim Wurzel-Feature um das oberste Feature der Baum-Hierarchie handelt, muss dieses auf ein Paket und nicht auf ein Team abgebildet werden (siehe hierzu auch Abschnitt 5.2.1). Hierfür wird eine spezielle Regel verwendet, die in Listing 7.7 dargestellt ist.

```
rule RootFeature2Package{
   from  s : UFM!Feature (
             s.isRootFeature())

   to    t : OT!Package(
               name  <- s.name.format(),
               packagedElement
                     <- s.getChildFeatureSet())
}
```

Listing 7.7: Transformation des Wurzel-Features

Die Regel transformiert ein *Feature* in ein *Package*. Hierfür werden verschiedene *Helper* verwendet. Einer der *Helper* ermittelt beispielsweise, ob es sich um das Wurzel-Feature handelt oder nicht (`isRootFeature`). Handelt es sich also um das Wurzel-Feature, wird ein neues *Package* erzeugt, das den Namen des Wurzel-Features erhält. Die Zuordnung der gekapselten Elemente erfolgt über den Helper `getChildFeatureSet()`, der alle zugehörigen Kinder ermittelt.

Features der ersten Ebene Alle Features der obersten Ebene müssen in Teams transformiert werden. Hierfür wird in Listing 7.8 die entsprechende Transformationsregel dargestellt.

```
rule Feature2TeamAggregation1stLevel{
   from  s: UFM!Feature(
             s.isFirstLevelFeature()
             and not  s.isParameterFeature())

   to      t: OT!Team(
             name <- s.name.format(),
             ownedRole    <- s.getChildFeatureSet(),
             ownedAttribute <- s.getAttributeSet())
}
```

Listing 7.8: Transformation der Features der ersten Ebene

Innerhalb der Regel wird zunächst der Name auf das zu erzeugende Team übernommen. Des Weiteren werden alle Kinder, die über den Helper `getChildFeatureSet()` ermittelt werden, dem Team als Rollen zugeordnet. Da durch die attributierten und elementaren Features auch bestimmte Attribute innerhalb der Rollen und Teams erzeugt werden, müssen diese als `ownedAttributes` zugeordnet werden. Der Helper `getAttributeSet()` ermittelt hierfür die in Frage kommenden Features.

Die restlichen Aggregations-Features Die Features, die im Rahmen der Aggregation nicht zu einem der oben angegebenen Fälle gehören, zählen zu den restlichen Features. Alle diese Features werden auf *TeamRole*s abgebildet. Hierfür wird nachfolgende Regel verwendet.

```
1  rule Feature2TeamAggregation{
2     from   s: UFM!Feature(
3                not s.isFirstLevelFeature()
4                and not s.isParameterFeature()
5                and not s.isRootFeature()
6                and not s.isGenSpecFeature())
7
8     to       t: OT!TeamRole(
9                name <- s.name.format(),
10               ownedRole <- s.getChildFeatureSet(),
11               ownedAttribute <- s.getAttributeSet())
12 }
```

Listing 7.9: Transformation eines Features in eine Team Rolle - Aggregation

Diese Regel bildet alle Features, die den definierten Bedingungen entsprechen, auf *TeamRoles* ab. Es werden dabei nur die Features berücksichtigt, die zu keinem der vorher genannten Gruppen gehören und zusätzlich kein Parameter- und kein Gen/Spec-Feature darstellen. Die Zuweisungen entsprechen den Zuweisungen für die Features der ersten Ebene. Aufgrund der bisher nur unzureichend dokumentierten Vererbungskonzepte für Regeln von ATL wird die dreizeilige Redundanz bewusst in Kauf genommen.

Die Transformation der Generalisierung/Spezialisierung

Auch die Transformation der Generalisierung/Spezialisierung unterscheidet sich in verschiedene Regeln. Es gibt eine Regel für die Features erster Ebene und eine Regel für die restlichen Features. Der einzige Unterschied liegt in der Transformation auf ein *Team* oder eine *TeamRole*. Aus diesem Grund soll nur eine der Regeln vorgestellt werden.

```
1  rule Feature2TeamGenSpec{
2     from  s: UFM!Feature(
3               not s.isRootFeature()
4               and s.isGenSpecFeature()
5               and not  s.isParameterFeature()
6               and not  s.isFirstLevelGenFeature())
7        to t: OT!TeamRole(
8                  name  <- s.name.format(),
9                  ownedRole <- s.getChildFeatureSet(),
10                 generalization     <- g),
11
12             g: OT!Generalization(
13                 general  <-
14                 s.parentGroup.parentFeature)
15
16 }
```

Listing 7.10: Transformation eines Features in eine Team Rolle - Generalisierung

Mit diesen Regeln werden alle Features, die in einer Gen/Spec-Beziehung stehen, in eine *TeamRole* oder ein *Team* transformiert. In Listing 7.10 ist die Transformation auf eine *TeamRole* dargestellt. Diese Regel erzeugt aus einem *Feature* neben einer *TeamRole* auch noch eine *Generalization*, durch die die Gen/Spec-Relation repräsentiert wird. Als Generalisierung wird das Superfeature der übergeordneten Feature-Gruppe angegeben, da diese eine Gen/Spec-Relation anzeigt. Diese Regel berücksichtigt aufgrund der definierten Bedingungen nur die in einer Gen/Spec-Relation stehenden Features. Die erzeugte *Generalization* wird der erzeugten *TeamRole* über das Feld `generalization` zugeordnet.

Die Transformation der Elementar-Features

In Abschnitt 5.3.2 wurden die Elementar-Features vorgestellt. Auch für diese Features wurde eine separate Regel deklariert. Diese Regel ist in Listing 7.11 dargestellt. Da die eigentlichen Elementar-Features bereits durch eine der anderen Re-

geln erzeugt werden, muss an dieser Stelle nur noch die entsprechende *Property* in dem zugehörigen Team beziehungsweise der zugehörigen Rolle gesetzt werden.

```
1 rule UserAttributeValue2PropertyElemental{
2    from   s: UFM!UserAttributeValue(
3                    s.isElementalAttribute())
4    to     t: OT!Property(
5                  name   <- s.parent.name.format()
6                                        .toLower(),
7                  type   <- s.parent)
8
9 }
```

Listing 7.11: Transformation eines Elementar-Features

Hierfür wird ein *UserAttributeValue*[6], das angibt, dass es sich beim zugeordneten Feature um ein Elementar-Feature handelt, in eine *Property* umgewandelt. Die Zuordnung der *Property* zum entsprechenden Team beziehungsweise zur entsprechenden Rolle ist bereits über den Helper `getAttributeSet()` in einer der vorangegangenen Regeln erfolgt. Deshalb muss an dieser Stelle nur noch der Typ und der Name der *Property* definiert werden. Hierfür wird der Name des übergeordneten Features verwendet.

Die Transformation der attributierten Features

Neben den Elementar-Features wurden in Abschnitt 5.3.1 die attributierten Features vorgestellt. Die Zuordnung der Attribute zu den zugehörigen Rollen und Teams erfolgt über eine weitere Regel. Diese Regel ist in Listing 7.12 dargestellt. Auch an dieser Stelle wird ein *UserAttributeValue* auf eine *Property* abgebildet. Hierbei wird über eine Bedingung geprüft, ob es sich beim zugeordneten Feature um ein Attribut handelt.

```
1 rule UserAttributeValue2PropertyParameter{
2    from   s: UFM!UserAttributeValue(
3                       s.isParameterAttribute())
4    to     t: OT!Property(
5                  name   <- s.parent.name.format())
6
7 }
```

Listing 7.12: Transformation eines attributierten Features

[6] vorgestellt in Abschnitt 7.2.2

Sofern dies der Fall ist, wird eine *Property* angelegt, die den Namen des übergeordneten Features erhält. Die eigentliche Zuordnung der Property ist bereits in einer der vorangegangen Regeln über den Helper `getAttributeSet()` erfolgt.

Die Transformation des Observer-Patterns

In Abschnitt 5.3.3 wurde die Transformation des Observer-Patterns vorgestellt. Im Rahmen der Umsetzung konnte die Abbildung des Observer-Patterns mit nur einer Regel realisiert werden. Die eigentliche Hauptfunktionalität wird bereits durch eine der allgemeinen Regeln implementiert. In Listing 7.13 ist die Regel zur Transformation der noch fehlenden Bestandteile dargestellt. Da sowohl die *Beobachter* und *Subjekte* als normale Features behandelt werden, wurden diese bereits im Rahmen der Aggregations- und Gen/Spec-Regeln abgebildet. Durch diese Regel müssen deshalb nur noch die eigentlichen Rollen innerhalb des *Beobachter*-Teams erzeugt werden, die dann die Rollen und Teams der eigentlichen Subjekte adaptieren. Hierfür wird in der Transformationsregel eine Transformation von einem *FeatureLink* in eine *Role* durchgeführt. Die eigentliche Zuordnung der Ansichtsrollen zum Beobachter wurde ebenfalls im Rahmen der anderen Regeln durchgeführt (hierfür wurde der Helper `getChildFeatureSet()` verwendet).

```
1 rule FeatureLink2RoleObserve{
2    from   s: UFM!FeatureLink(s.isObserverLink())
3
4    to     t: OT!Role(
5           name <- s.start.name.format()
6                   + s.end.name.format(),
7           baseClassifier <- s.end
8       )
9 }
```

Listing 7.13: Transformation des Observer-Patterns

Innerhalb dieser Regel muss deshalb nur noch die Rolle erzeugt werden, welche das eigentliche Team bzw. die Rolle adaptiert. Hierfür wird eine Rolle mit dem Namen *[BeobachterName][SubjektName]* erzeugt, die als *baseClassifier*[7] das zu überwachende Subjekt erhält.

Die Transformation der influence-Beziehung

Auch die in Abschnitt 5.2.2 vorgestellte Transformation der influence-Beziehung kann über eine einzelne Regel abgebildet werden. Da alle Transformationsregeln

[7] Wie in 7.2.3 beschrieben, wird über den `baseClassifier` die Adaption abgebildet

ausschließlich auf Teams basieren, wurde in Abschnitt 5.2.2 beschrieben, dass für jede *playedBy*-Relation eine zusätzliche Rolle generiert werden muss, da nur diese ein andere Rolle oder Klasse adaptieren kann. Für dieses Verfahren wird in Listing 7.14 eine entsprechende Regel vorgestellt.

```
1 rule FeatureLink2RoleInfluence{
2    from  s: UFM!FeatureLink(s.isInfluenceLink())
3
4    to    t: OT!Role(
5          name <- 'Role' + s.start.name.format(),
6          baseClassifier    <- s.end
7       )
8 }
```

Listing 7.14: Transformation der influence-Beziehung

Diese Regel erzeugt aus einem *FeatureLink* eine *Role*, wenn es sich bei dem *FeatureLink* um eine influence-Beziehung handelt. Die Zuordnung der Rolle zum entsprechenden Team ist bereits in den allgemeinen Regeln erfolgt, so dass an dieser Stelle neben der Festlegung des Namens für die Rolle nur noch die Rolle oder das Team, das adaptiert wird, angegeben werden muss. Der Name der Rolle setzt sich dabei aus dem übergeordneten Team und dem Prefix *Role* zusammen. Die playedBy-Relation wird durch das Setzen des `baseClassifiers` abgebildet.

Kapitel 8

Schlussbetrachtung

In diesem Kapitel soll neben einer Zusammenfassung der Arbeit auch eine Bewertung des vorgestellten Ansatzes und ein Ausblick auf zukünftige Arbeiten gegeben werden. Hierfür wird im nächsten Abschnitt zunächst die Arbeit zusammengefasst. Im darauf folgenden Abschnitt wird das aus der Transformation hervorgegangene Design für Pong mit einer bereits existierenden Implementierung von Pong verglichen, bevor im letzten Abschnitt noch ein Ausblick auf eine mögliche Erweiterung gegeben wird.

8.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde ein Verfahren zur Abbildung von produktlinienorientierten Feature-Modellen auf Object Teams vorgestellt. Hierfür wurden in Kapitel 2 die grundlegenden Begriffe wie *Software-Produktlinien*, *Feature-Modelle*, *aspektorientierte Programmierung*, *Object Teams* und die Grundlagen des *Model Driven Engineerings* vorgestellt. Nachdem in Kapitel 3 die betrachtete Fallstudie behandelt wurde, wurde in Kapitel 4 eine erweiterte Feature-Notation vorgestellt. Hierbei wurde besonderes Augenmerk auf die Unterscheidung zwischen Aggregation und Generalisierung/Spezialisierung gelegt. Im Rahmen der Generalisierung/Spezialisierung wurde zusätzlich eine neue Kardinalitätsklassifikation eingeführt. Diese dient dazu, Spezialisierung in Richtung der Blätter auch auf die Kardinalitäten anwenden zu können. Neben der genauen Differenzierung zwischen Aggregation und Generalisierung/Spezialisierung wurden zwei weitere Erweiterungen vorgestellt. Zum einen waren das die *Elementar-Features* und zum anderen die *Pattern-Notation*. In Kapitel 5 wurden dann die eigentlichen Transformationsregeln definiert, um die Feature-Modelle nach Object Teams zu transformieren. Hierbei wurden insgesamt sechs Transformationsregeln benötigt. Auch hier wurde ein besonderes Augenmerk auf die unterschiedliche Transformation von Aggregation und

Generalisierung/Spezialisierung gelegt, da in den bisherigen Feature-Modellen nur die Aggregation berücksichtigt wurde. In Kapitel 6 wurden dann schließlich sowohl die erweiterte Feature-Notation als auch die Transformationsregeln auf das in der Fallstudie erarbeitete Feature-Modell angewendet. In Kapitel 7 wurde die Umsetzung der in Kapitel 4 und 5 dargelegten Regeln vorgestellt. Hierbei wurden zunächst die verwendeten Technologien erläutert. Im Anschluß wurden dann die zu Grunde liegenden Metamodelle betrachtet, bevor im letzten Abschnitt die eigentlichen technischen Transformationsregeln dargestellt wurden.

8.2 Beurteilung

In diesem Abschnitt soll der in dieser Diplomarbeit bearbeitete Ansatz beurteilt werden. Bei der Beurteilung wird zunächst das Ergebnis der Fallstudie mit der ursprünglichen OTPong Version (siehe 3.2.1) verglichen. Im Anschluss wird dann das Gesamtkonzept kurz bewertet.

8.2.1 Vergleich OTPong

Wie in Abschnitt 3.2.1 bereits erwähnt, wurde OTPong im Rahmen einer Lehrveranstaltung an der TU Berlin im Fachbereich Softwaretechnik entwickelt. Ausgehend von der bestehenden OTPong-Implementierung, wurde im Rahmen dieser Diplomarbeit eine Spiele-Produktlinie entwickelt, aus der sich unter anderem auch Pong ableiten lassen soll. In diesem Abschnitt werden die beiden existierenden Architekturen verglichen, um daraus Rückschlüsse auf die Qualität der automatisch erzeugten Pong-Architektur ziehen zu können. Hierbei ist zu berücksichtigen, dass OTPong „von Hand“ modelliert wurde, wohingegen die Architektur der allgemeinen Spiele-Produktlinie aus einem Feature-Diagramm abgeleitet ist.
Zunächst soll jedoch die OTPong-Architektur vorgestellt werden. In Abbildung 8.1 ist dafür das zugehörige Architektur-Modell dargestellt. Bei der Abbildung wurde aus Gründen der Übersichtlichkeit ein Teil der playedBy-Relation weggelassen. Eine Abbildung im Querformat befindet sich im Anhang (siehe Abbildung B.4), eine vollständige Version des Architektur-Modells befindet sich auf der beiliegenden *CD-ROM*. Das OTPong-Modell unterteilt sich im wesentlichen in folgende fünf Pakete:

- *games* — allgemeine Module für ein Spiel
- *games.model* — Klassen für die verwendeten Elemente im Spiel
- *otpong* — von *games* abgeleitete Spezialisierungen für Pong
- *otpong.model* — für OTPong spezialisierte von *games.model*-Klassen abgeleitete Spielelemente

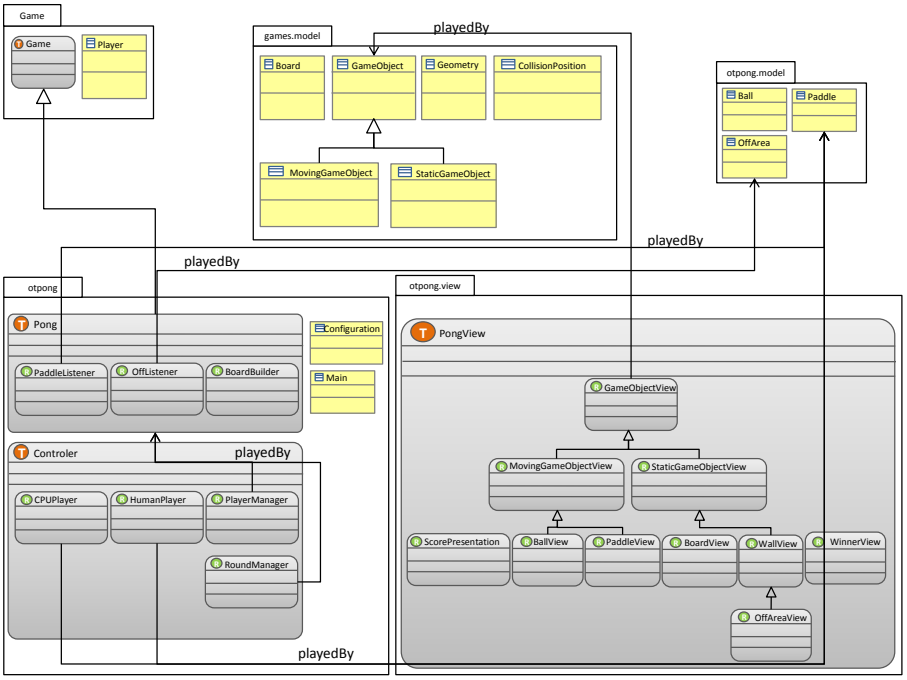

Abbildung 8.1: OTPong-Ursprungsmodell

- *otpong.view* — Paket für die grafische Repräsentation der einzelnen Spielelemente (*Observer-Pattern*)

Paket *games* Das Paket *Game* beinhaltet das allgemeine Team *Game* und die Datenstruktur für den Spieler (*Player*).

Paket *games.model* Das Paket *games.model* beinhaltet die einzelnen Spielelemente in allgemeiner Form. Hierzu gehören zum Beispiel das Spielfeld (*Board*), das allgemeine Spielobjekt (*GameObject*), von dem sich die beweglichen (*MovingGameObject*) und die statischen (*StaticGameObject*) Spielobjekte ableiten.

Paket *otpong* Ausgehend von den Paketen *games* und *games.model* gibt es OTPong spezifische Pakete. Hier gibt es zunächst das *otpong*-Paket, welches das Team *Pong* und *Controler* sowie die Klassen *Configuration* und *Main* enthält. Das Team *Pong* stellt hierbei einen zentralen Bestandteil der Implementierung dar. Es enthält zwei Rollen, die Bewegungen verschiedener Spielobjekte überwachen (*PaddleListner* und *OffListener*). Weiterhin beinhaltet das Team die Rolle *BoardBuilder*, welche das eigentliche Spielfeld initialisiert. Neben dem Team *Pong* beinhaltet das

Paket zusätzlich das Team *Controler*. Dieses Team vereint die Rollen *CPUPlayer*, *HumanPlayer*, *PlayerManager* und *RoundManager*. Diese Rollen sind für die eigentliche Spiellogik verantwortlich.

Paket *otpong.model* Dieses Paket beinhaltet die auf OTPong spezialisierten Spielelemente. Hierzu gehören der Schläger (*Paddle*), der Ball (*Ball*), die Wand (*Wall*) und der Aus-Bereich (*OffArea*). Die Klassen *Paddle* und *Ball* sind von der allgemeinen Klasse *MovingGameObject* des Paketes *games.model* abgeleitet. Die Klasse *Wall* ist wiederum von der Klasse *StaticGameObject* abgeleitet. Die Klasse *Paddle* wird zusätzlich unter anderem durch die Rollen *CPUPlayer* und *HumanPlayer* des Teams *Pong* adaptiert. Somit spielt der Schläger einmal die Rolle eines Spielers und einmal die des Computergegners.

Paket *otpong.view* Das letzte verbleibende Paket im Rahmen von OTPong ist das Paket *otpong.view*. Bei diesem Paket handelt es sich um die grafische Darstellung der einzelnen Spielelemente. Das Team *PongView* ist hierfür mit den entsprechenden Rollen versehen, die die einzelnen Klassen der Spielelemente adaptieren[1]. Das bedeutet, dass für jedes Spielelement eine grafische Repräsentation existiert, die in Form einer Rolle an die entsprechende Klasse gebunden wird. Die *Views* stellen hierbei die *Observer* und die einzelnen Spielelemente die *Subjects* des *Observer-Patterns* (siehe 4.2.2) dar.

Architektur der Spiele-Produktlinie

Die Architektur der Spiele-Produktlinie wurde bereits in den Kapiteln 3 und 6 ausführlich vorgestellt. Im Rahmen dieses Abschnittes soll nur noch einmal die grafische Darstellung aus Kapitel 6 in Abbildung 8.2 wiederholt werden. Für eine genaue Beschreibung des Modells sei auf das Kapitel 6 verwiesen.

8.2.2 Gemeinsamkeiten und Unterschiede

Nachdem nun sowohl die ursprüngliche OTPong-Architektur, als auch die Architektur der Spiele-Produktlinie vorgestellt wurde, sollen diese nun abschließend verglichen werden. Auf den ersten Blick fällt auf, dass sich die beiden Architekturen strukturell sehr ähnlich sind. Nur die Granularität der beiden Architekturen unterscheidet sich in einigen Bereichen. Beide Version setzen sich im Wesentlichen aus den Spielelementen, der Ansicht und der Spiellogik zusammen. Bei OTPong wird an dieser Stelle jedoch noch etwas mehr generalisiert. So leiten sich Spielelemente

[1] Diese playedBy-Relationen sind, wie bereits erwähnt, nicht vollständig im vorliegendem Diagramm enthalten. Jedoch können die einzelnen playedBy-Relation gut nachvollzogen werden, da die Zugehörigkeit der einzelnen Rollen aus dem Namen hervor geht (...*View*).

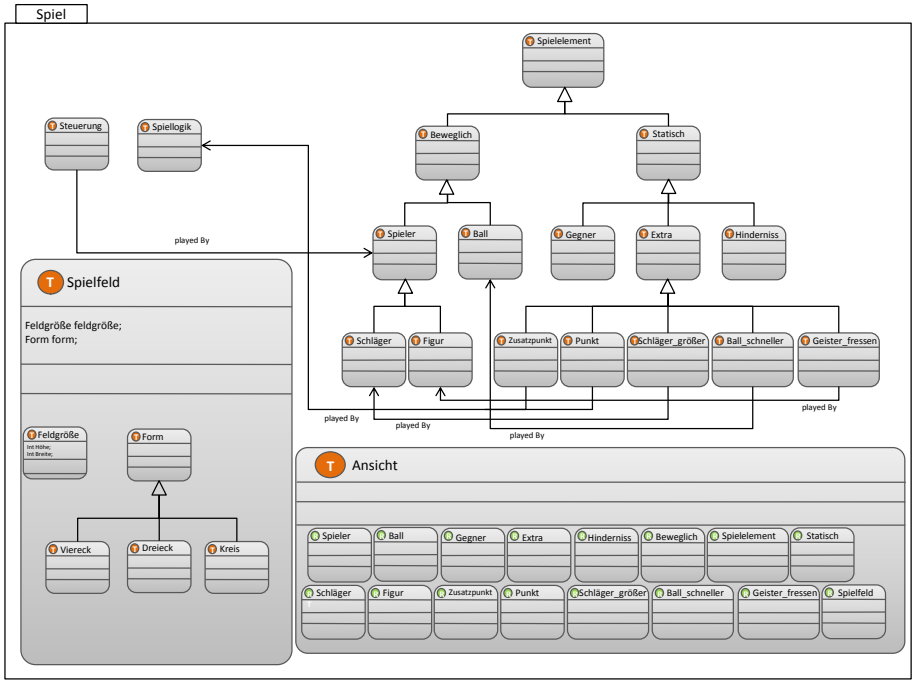

Abbildung 8.2: UFA-Diagramm der Spiele-Produktlinie

und Spiellogik von allgemeinen Klassen ab. Dies könnte in der Spiele-Produktlinie ebenso berücksichtigt werden, jedoch müsste hierfür das Feature-Modell entsprechend angepasst werden, was zu einem gewissen verzichtbaren Overhead innerhalb des Modells führen würde. Des Weiteren fällt auf, dass innerhalb der Spiele-Produktlinie ausschließlich Teams verwendet werden, wohingegen in der OTPong-Architektur Klassen, Rollen und Teams zum Einsatz kommen. Diese reduzierte Darstellung ist auf die relativ allgemein gehaltenen Transformationen zurückzuführen. An dieser Stelle könnten die Transformationsregeln noch weiter spezialisiert werden, was einerseits die Komplexität der Regeln erhöhen, andererseits jedoch die durch OT/J gegebenen Strukturierungsmöglichkeiten besser ausnutzen würde. Wie bereits erwähnt, macht das OTPong-Modell einen vollständigeren Eindruck als die Spiele-Produktlinie. Das liegt hauptsächlich daran, dass es sich bei OTPong um eine vollständig implementierte Version und bei der Spiele-Produktlinie nur um ein automatisch erzeugtes Programmgerüst handelt. Diese automatisch erzeugte Version stellt nur die Grundlage dar und muss im Rahmen des Implementierungsprozesses sicherlich noch um einige Elemente erweitert werden. So ist beispielsweise die Spiellogik in OTPong bereits relativ feingrunalar, wohingegen die Spiellogik und Steuerung in der Spiele-Produktlinie noch relativ grobgranular repräsentiert ist. Ein weiterer Unterschied ist die Strukturierung der einzelnen Elemente. In OT-

Pong werden verschiedene Pakete verwendet, um die einzelnen Teilbereiche des Spiels besser zu strukturieren. So gibt es für die allgemeinen Klassen, die Ansicht und die spezialisierten Elemente einzelne Pakete, währenddessen es in der Spiele-Produktlinie nur ein Paket gibt. Dies kommt dadurch zu Stande, dass durch die momentanen Abbildungsregeln nur aus dem Wurzel-Feature ein Paket erzeugt wird. Weitere Pakete lassen sich aus den Informationen des Feature-Modells vorerst nicht ableiten. An dieser Stelle könnten Abbildungsregeln eingeführt werden, die bestimmte Subfeature-Bäume in ein separates Paket einsortieren. Jedoch müsste geprüft werden, welche Indikatoren hierfür in Frage kämen. Von zusätzlichen Attributen auf Feature-Ebene wäre auf den ersten Blick abzusehen, da sich das Feature-Modell dadurch nur unnötig verkompliziert und eine spätere Strukturierung auf Object Teams-Ebene relativ unkompliziert durchgeführt werden kann, ohne den Bezug zum Feature-Modell zu verlieren. Der letzte große Unterschied lässt sich in der Abbildung des Observer-Patterns erkennen. Während in der OTPong-Implementierung auch für die einzelnen Observer eine Vererbungshirarchie aufgebaut wird, ist die Struktur der einzelnen Observer in der Spiele-Produktlinie flach. Auch hier werden momentan möglichst unkomplizierte Abbildungsregeln verwendet, die nach genauerer Prüfung möglicherweise noch weiter spezialisiert werden können. Jedoch ist momentan noch nicht klar, ob in allen Fällen die Vererbungsstruktur der Subjekte auf die Observer übernommen werden kann (siehe hierzu auch Abschnitte 4.2.2 und 6).

8.2.3 Bewertung des Gesamtkonzeptes

Zusammenfassend kann gesagt werden, dass die bestehenden Ansätze, die im Rahmen dieser Diplomarbeit erweitert wurden, funktionieren. Zum Teil müssen die Ansätze an verschiedenen Stellen zwar noch weiter konkretisiert beziehungsweise angepasst werden, jedoch konnte gezeigt werden, dass die Erweiterungen grundlegend funktionieren. Nicht nur im vorangegangen Vergleich wurde gezeigt, dass der entwickelte Ansatz im Rahmen der Fallstudie durchaus Potential für eine Praxistauglichkeit besitzt. Gerade durch den angefertigten Prototypen konnte nachgewiesen werden, dass die technische Umsetzung der definierten Transformationsregeln gut funktioniert. Vor allem durch die Verwendung der deklarativen Transformationssprache ATL, konnte die Regeln übersichtlich und teilweise ohne großen Overhead technisch umgesetzt werden.

8.3 Ausblick

Im Rahmen dieser Arbeit wurden die in [HMPS07] grundlegend definierten Transformationsregeln erweitert. Somit wurde die Lücke zwischen der Modellierung und der allgemeinen Implementierung geschlossen. Das fehlende Glied, das im Rah-

men des Ausblicks ansatzweise betrachtet werden soll, ist die eigentliche Produkt-Instanzierung. Die Produkt-Instanzierung ist Bestandteil des bereits vorgestellten *Application Engineerings* (siehe 2.1.5). Da es sich bei der Produkt-Instanzierung um ein durchaus komplexes Thema handelt, welches den Umfang dieser Arbeit übersteigen würde, sollen an dieser Stelle nur technische Ansätze für die Produkt-Instanzierung vorgestellt werden. Diese können dann im Rahmen einer gesonderten Arbeit konkretisiert werden. In Abbildung 8.3 ist noch einmal der Produktlinienzyklus im Rahmen von Object Teams und Feature-Modellen dargestellt. Ausgehend

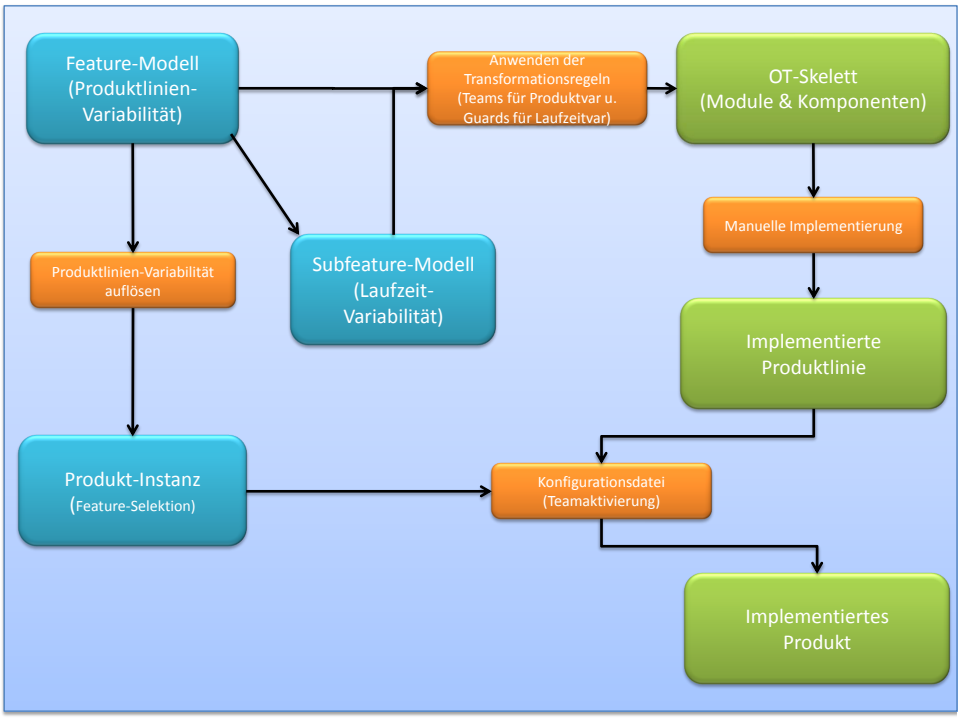

Abbildung 8.3: Produkterstellungs-Zyklus

vom Feature-Modell wird über die Transformationsregeln das OT-Skelett erstellt (siehe Abbildung 8.3). Dieses Object Teams-Gerüst enthält vorerst die gesamte Variabilität. In einem weiteren Schritt werden die erzeugten Teams, Rollen und Klassen implementiert. In diesem Schritt können noch weitere Klassen, Rollen und Teams hinzukommen, die nicht in Form von Features definiert wurden. Nach dem die Implementierung abgeschlossen ist, liegt eine fertige Produktlinie auf Object Teams-Ebene vor. Die Variabilitäten sowie die Gemeinsamkeiten sind durch Rollen und Teams abgebildet und können im darauf folgenden Instanzierungs-Schritt beliebig aktiviert beziehungsweise deaktiviert werden. Das in der Abbildung 8.3

dargestellte Auflösen der Produktlinien-Variabilität wird im folgenden Abschnitt genauer beschrieben.

8.3.1 Auflösen der Variabilitäten

Die im Feature-Modell enthalten Variabilitäten können grob in zwei Klassen unterschieden werden. Zum einen sind dies die *Produktvariabilitäten*, hierbei handelt es sich um Variabilitäten, die einzelne Produkte der Produktlinie unterscheiden und zum anderen sind dies die *Laufzeitvariabilitäten*, hierbei handelt es sich um Variabilitäten, die bei einem Produkt zu bestimmten Zeitpunkten der Laufzeit variieren können. Für die Umsetzung der Darstellung und der Konfiguration wären folgende Möglichkeiten denkbar. Das Ziel der Produkt-Instanzierung ist das Auflösen der Variabilitäten.

Produktvariabilität

Bei der Produktvariabilität handelt es sich um die Variabilität, die einzelne Produkte einer Produktlinie unterscheidet. Diese Variabilität wird deshalb bereits bei der Produkt-Instanzierung aufgelöst. Um diesen Prozess im Rahmen der Feature-Modellierung so einfach wie möglich zu gestalten, wäre es denkbar, die zu verwendenden Features — unter Berücksichtigung der bestehenden Abhängigkeiten — bereits im Feature-Diagramm zu selektieren. Hierfür müsste der bestehende Feature-Editor (*IO* siehe auch Abschnitt 7.2.2) entsprechend angepasst werden. Denkbar wäre an dieser Stelle eine Auswahl der Features per *Point & Click*, das bedeutet, dass Features, die in ein konkretes Produkt einfließen sollen, per Klick direkt im Feature-Diagramm ausgewählt werden. Während der Auswahl wird permanent eine Konsistenzprüfung durchgeführt, so dass sofort erkannt wird, wenn eine ungültige Feature-Kombination vorliegt. Nachdem die für das Produkt gewünschten Features im Feature-Diagramm ausgewählt wurden, müssen diese noch auf die bestehende Object Teams-Implementierung übertragen werden.

Zuordnung von Features zu Teams Für diesen Schritt ist eine Zuordnung von Features auf Teams unabdingbar. Vorstellbar wäre an dieser Stelle eine Datenstruktur, die während der Abbildung der Features auf Object Teams erzeugt wird und die entsprechenden Zuordnungen sowie Abhängigkeiten beinhaltet. Diese Datenstruktur müsste dann während des Implementierungsprozesses kontinuierlich mitgepflegt werden, so dass auch die im Rahmen der Implementierung hinzugekommenen Klassen, Rollen und Teams berücksichtigt werden können. Für die Datenstruktur wären zwei grundsätzliche Möglichkeiten denkbar:

Kommentare Die zugehörigen Features und Abhängigkeiten könnten in Form von Kommentaren direkt in den Klassen annotiert werden. Hierfür müssten entsprechende Schlüsselwörter eingeführt werden, die dann während des Produkt-Instanzierungsprozesses berücksichtigt werden können. In Listing 8.1 ist eine mögliche Notation von Abhängigkeiten dargestellt.

```
1  /**
2   * @AssociatedFeature A
3   * @DependantFeatures B, C, D, E
4   * @MutualExcludedFeatures F
5   */
6  public team class TeamA {
7      /**
8       * @AssociatedFeature G
9       */
10     public class RoleG {
11         protected void print() {
12             System.out.println("Team: " + TeamA.this);
13         } // method print
14     } // role RoleG
15 } // team TeamA
```

Listing 8.1: Beispiel für die Darstellung von Abhängigkeiten

Bei der Abbildung der Features auf Object Teams werden zusätzlich Kommentare mit folgendem Inhalt erzeugt:

- `AssociatedFeature` — gibt den Namen des zugeordneten Features an
- `DependantFeatures` — gibt alle Features an, von denen das aktuelle Team abhängig ist
  Hierbei handelt es sich also um die Features, die zum Beispiel in einer *«include»* Beziehung zueinander stehen.
- `MutualExcludedFeatures` — gibt die Features an, die nicht gleichzeitig mit dem aktuellen Feature aktiv sein dürfen
  Es handelt sich also um die Features, die sich innerhalb eine Feature-Modells gegenseitig ausschließen.

Der große Vorteil bei dieser Methode ist der relativ geringe Aufwand für die Wartung der Zuordnung, da die Kommentare während des Implementierungsprozesses problemlos mitgepflegt werden können. Des Weiteren können auch problemlos neue Klassen hinzugefügt werden, die dann einfach über das `@AssociatedFeature` dem passenden Feature zugeordnet werden können.

XML-Datei Die Zuordnung von Features zu Teams könnte alternativ auch über eine XML-Datei erfolgen. Die XML-Datei würde dann alle Features und deren zugehörige Teams beinhalten. Ebenso könnten Abhängigkeiten zwischen einzelnen Features innerhalb dieser Datei gepflegt werden. Der Vorteil dieser Datenhaltung ist es, dass die Daten an einer Stelle gehalten werden können, was sowohl die Generierung als auch die feature-seitige Wartung erleichtert. Die Nachteile hingegen sind die erschwerte Wartung in Bezug auf die Teams und die teilweise redundante Datenhaltung gegenüber dem Feature-Diagramm.

Zusammenfassend wäre zu sagen, dass die dezentrale Haltung mittels Kommentaren höchstwahrscheinlich vorzuziehen ist.

Abbildung der Konfiguration Nachdem sowohl die Abbildung als auch die spätere Zuordnung von Features auf Teams definiert sind, muss nun noch die eigentliche Produkt-Instanzierung vorgenommen werden. Hierfür soll die Möglichkeit der separaten Teamaktivierung in Object Teams verwendet werden (siehe Abschnitt 2.3.3). Da einzelne Features während der Transformation hauptsächlich auf Teams abgebildet werden, besteht die Möglichkeit, diese auch getrennt voneinander zu aktivieren. Für die Konfiguration und Aktivierung der konkreten Produkt-Instanz auf Object Teams-Ebene soll die zur Verfügung stehende Konfigurationsdatei genutzt werden. Hierbei handelt es sich um eine Datei mittels derer, die zu aktivierenden Teams im Vorfeld festgelegt werden können. Während der Kompilierung wird diese Datei dann ausgewertet und das entsprechende Produkt erzeugt. Die Aktivierung per Konfigurationsdatei wurde in Abschnitt 2.3.3 genauer vorgestellt.

In diesem konkreten Anwendungsfall würden, über die in *IO* ausgewählten Features die zugehörigen Teams selektiert werden. Diese Teams würden dann mit ihrem vollständigen Paketpfad und Namen in die Konfigurationsdatei geschrieben werden. Im anschließenden Kompilierungsprozess würden die entsprechenden Teams aktiviert werden und das Produkt wäre instanziert. In Listing 8.2 ist noch einmal ein Beispiel für die Konfigurationsdatei gegeben.

```
1 # Team Activation
2 package1.TeamA
3 package1.TeamB
4 package2.TeamV
```

Listing 8.2: Beispiel für die Teamaktivierung

Laufzeitvariabilität

Das Pendant zur Produktvariabilität ist die Laufzeitvariabilität. Auch diese Variabilitätsklasse kann in Object Teams umgesetzt werden. Da es sich hierbei jedoch um ein relativ komplexes Thema handelt, sollen an dieser Stelle nur kurz die Grundzüge vorgestellt werden. Die Laufzeitvariabilität unterscheidet im Gegensatz zur Produktvariabilität nicht die einzelnen Produkte, sondern Merkmale innerhalb eines Produkts. Das bedeutet, dass Produkte aufgrund der Laufzeitvariabilität während des Programmverlaufes verschiedene Zustände durchlaufen, an die wiederum verschiedene Variabilitäten geknüpft sind. Im Rahmen der Fallstudie könnte die unterschiedliche Darstellung eines Levels beispielsweise als eine Laufzeitvariabilität angesehen werden. Bei Pong könnte das Spielfeld in *Level 1* zum Beispiel viereckig und in *Level 2* zum Beispiel dreieckig sein. Die Laufzeitvariabilität wird ebenso wie die Produktvariabilität im Feature-Modell dargestellt. Abhängig von Produkt und Zustand kann sich die Laufzeitvariabilität ändern oder auch bereits einschränken. Um diesen Sachverhalt darzustellen, könnten zum Beispiel Subfeature-Modelle eingesetzt werden. Diese können dann produkt- oder zustandsbezogen einzelne Teile des Feature-Modells darstellen und verfeinern. Hierfür würde ausgehend vom Subfeature-Modell eine Referenz auf das Ursprungs-Feature-Modell gesetzt werden, um beide Feature-Modell in Beziehung zu bringen.

Abbildung der Laufzeitvariabilität Da sich das Auflösen der Laufzeitvariabilitäten nach bisherigem Kenntnisstand höchstwahrscheinlich an bestimmte Zustände im Programm knüpft, können die Laufzeitvariabilitäten bereits bei der Abbildung der Produktlinie auf Object Teams mit übertragen werden (siehe hierzu auch Abbildung 8.3). Die Form der Abbildung ist noch weitgehend offen, es wäre jedoch vorstellbar, die Variabilitäten über das Konzept der *Guards*-Prädikate abzubilden (siehe hierzu auch 2.3.3). Diese ermöglichen die bedingungsgeknüpfte Teamaktivierung.

Zur Laufzeit werden diese Prädikate dann geprüft. Sofern sie erfüllt sind, wird das entsprechende Team aktiviert beziehungsweise die entsprechende Stelle im Programm ausgeführt. Hierüber können dann die Laufzeitvariabilitäten zustandsgebunden aufgelöst werden. Die eigentliche Umsetzbarkeit der Laufzeitvariabilitäten auf diesem Wege ist bisher noch weitgehend ungeklärt und muss erst noch genau evaluiert werden.

Anhang A

Inhalt der CD

Die der Diplomarbeit beiliegenden CD beinhaltet sowohl den praktischen als auch den theoretischen Teil der Diplomarbeit. Nachfolgend soll der Inhalt der CD kurz erläutert werden.

A.1 Verzeichnis: *Praktischer Teil*

Das Verzeichnis *Praktischer Teil* beinhaltet diverse Unterverzeichnisse in denen die einzelnen, für den praktischen Teil der Arbeit relevanten, Daten angeordnet sind.

A.1.1 Verzeichnis: *Quellcode*

Dieses Verzeichnis beinhaltet den Quellcode des im Rahmen der Diplomarbeit entwickelten Plugins.

A.1.2 Verzeichnis: *Metamodelle*

In diesem Verzeichnis ist sowohl das UFM- als auch das OTUML-Metamodell enthalten.

A.1.3 Verzeichnis: *Editoren*

Dieses Verzeichnis beinhaltet die im Rahmen dieser Arbeit verwendeten Editoren. Dazu gehört der Editor *IO* mit dem das Eingabemodell bearbeitet werden kann und der *OTUML*-Editor der zum editieren das Ausgabemodells geeignet ist.

A.1.4 Verzeichnis: *Bin*

In diesem Verzeichnis befindet sich sowohl das Feature als auch die Eclipse Update Site des entwickelte Plugins.

A.1.5 Verzeichnis: *Beispielmodell*

In diesem Verzeichnis befindet sich ein Beispieleingabemodell. Bei diesem Modell handelt es sich um das Feature-Modell der vorgestellten Spiele-Produktlinie.

A.1.6 Verzeichnis: *Ausgabemodell*

Dieses Verzeichnis beinhaltet, ein mit dem FTA-Plugin erzeugtes Ausgabemodell (Spiele-Produktlinie).

A.1.7 Verzeichnis : *OTPong-Modell*

In diesem Verzeichnis ist das OTUML-Modell von OTPong hinterlegt.

A.2 Verzeichnis: *Theoretischer Teil*

In diesem Verzeichnis befinden sich zwei verschiedene Versionen der Diplomarbeit in Form von PDFs. Zum einen handelt es sich um eine komprimierte Online-Version und zum anderen um eine unkomprimierte Druck-Version der Diplomarbeit.

Anhang B

Abbildungen im Querformat

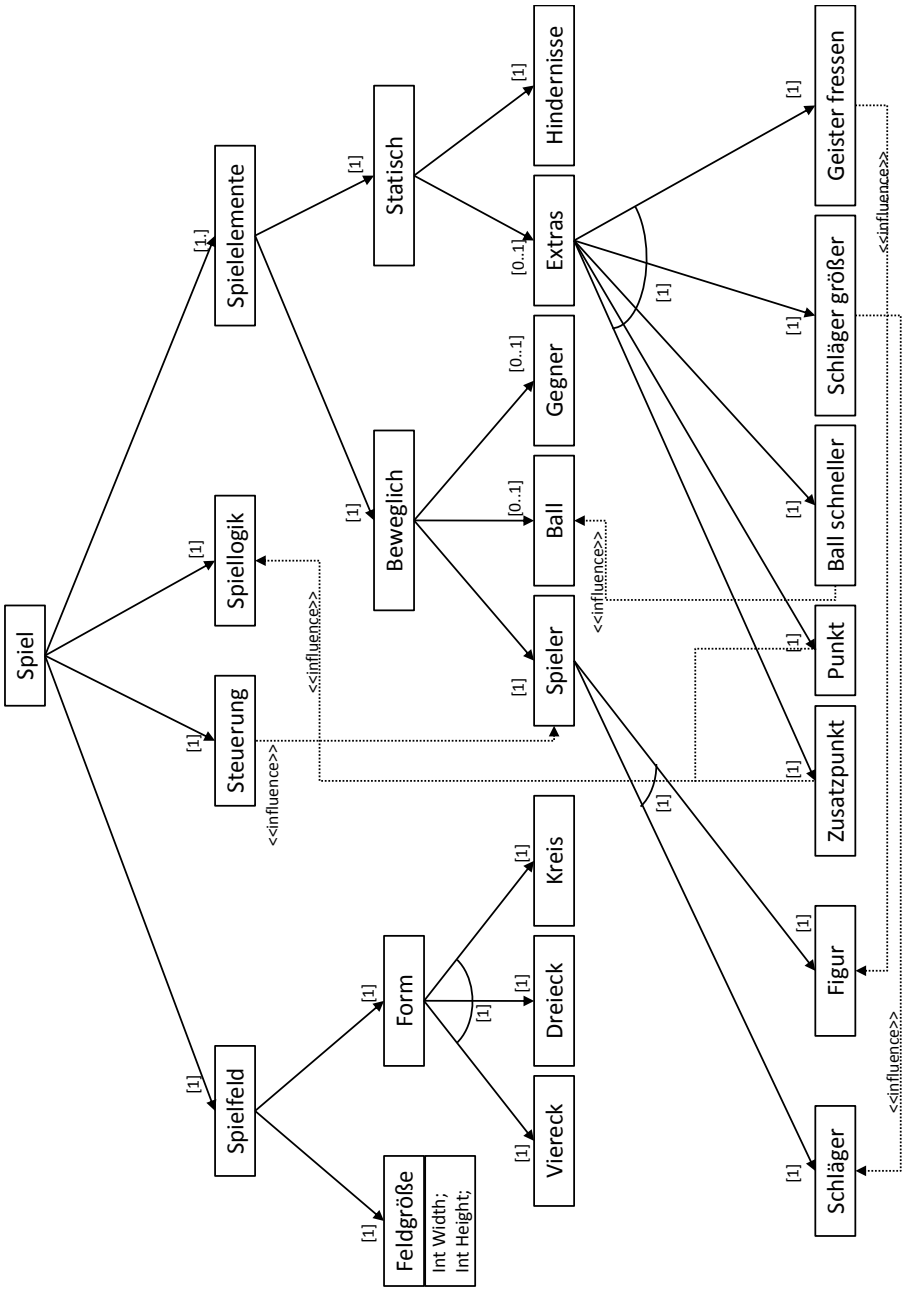

Abbildung B.1: Feature Diagramm vor Modifikation - Querformat

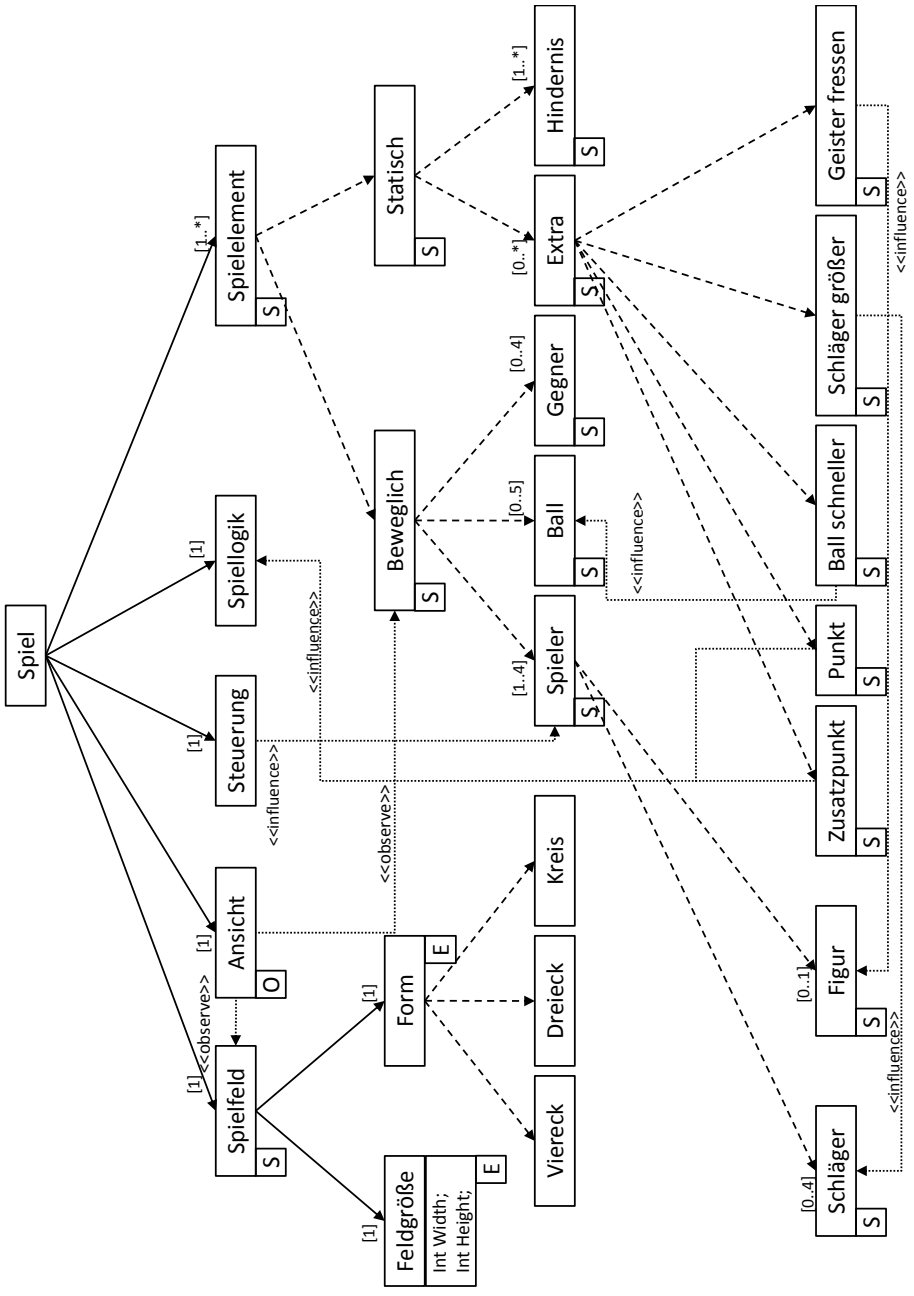

Abbildung B.2: Feature Diagramm nach Modifikation - Querformat

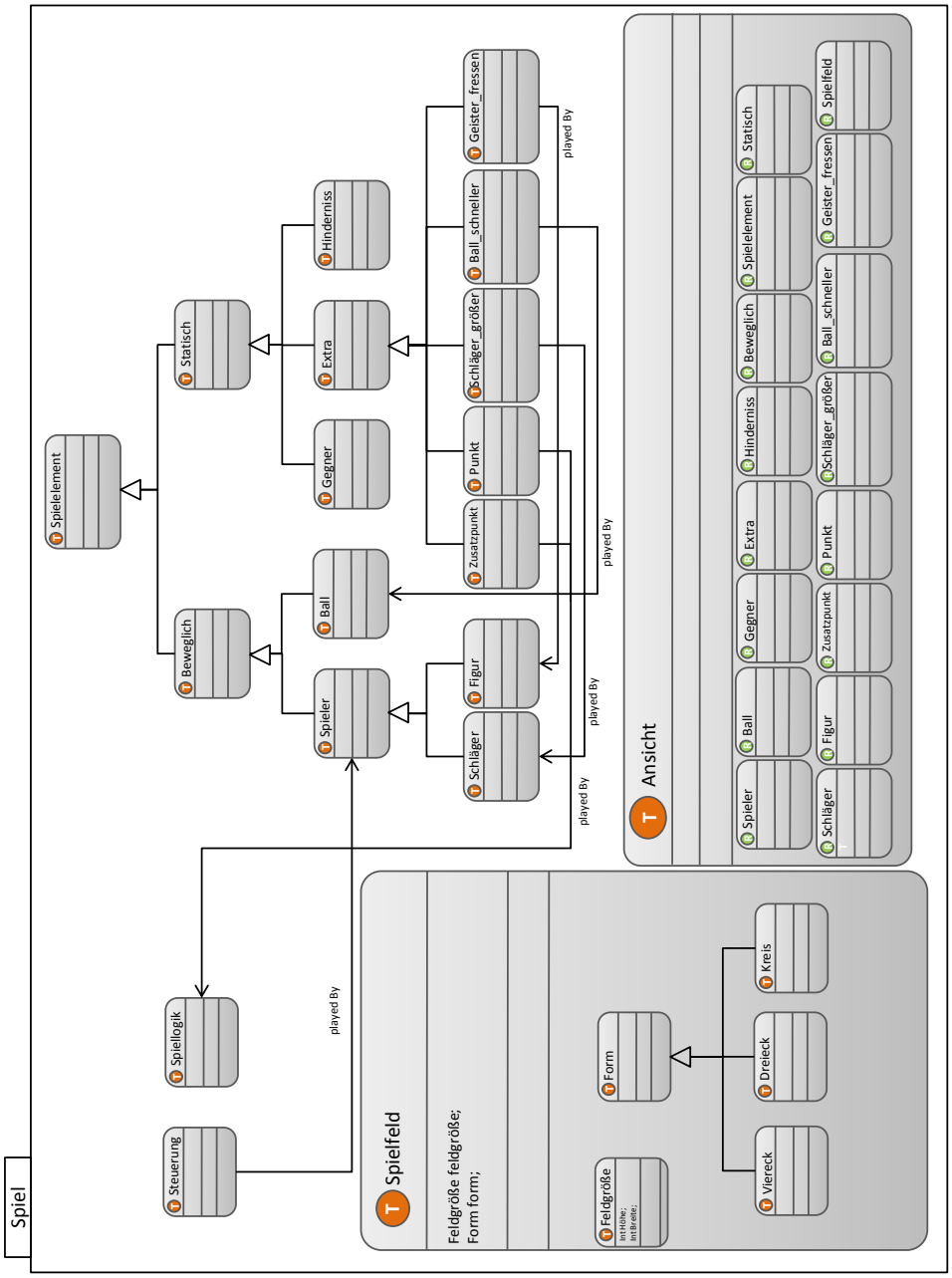

Abbildung B.3: UFA Diagramm der Spiele-Produktlinie

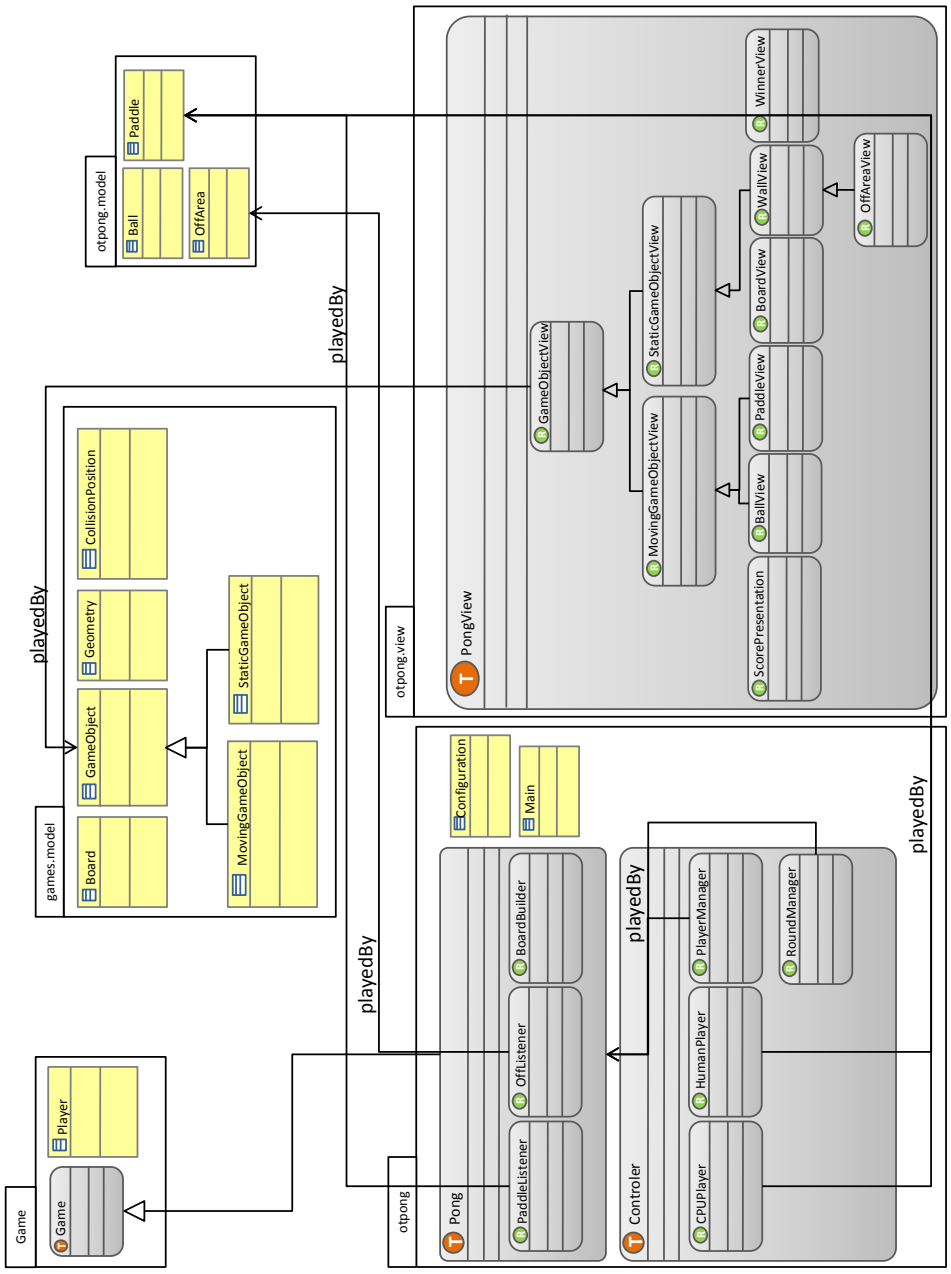

Abbildung B.4: OTPong-Ursprungsmodell

Abkürzungsverzeichnis

AOP	Aspektorientierte Programmierung
API	Application Programming Interface
ATL	ATLAS Transformation Language
CIM	Computation Independent Model
EMF	Eclipse Modeling Framework
FODA	Feature Oriented Domain Analysis
FTA	Features to Aspects
M2C	Model to Code
M2M	Model to Model
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OT	Object Teams
OT/J	Object Teams/Java
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Queries/Views/Transformations
UML	Unified Modeling Language
XMI	XML Metadata Interchange

[]

Literaturverzeichnis

[Asp01] *AspectJ*. Website. 2001. – http://www.eclipse.org/aspectj/; besucht am 18. Oktober 2007. 25

[ATL06] ATLAS GROUP(LINA & INRIA) NANTES: ATL: ATLAS Transformation Language - User Manual / INRIA & LINA. 2006. – Forschungsbericht xi, 36, 37, 76

[Aza03] AZAD BOLOUR: *Notes on the Eclipse Plug-in Architecture*. Website. 2003. – http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html; besucht am 05. Dezember 2007. 77

[BA01] BERGMANS, L. ; AKŞIT, M.: Composing multiple concerns using composition filters / University of Twente, The Netherlands. 2001. – Forschungsbericht 25

[Bus72] BUSHNELL, Nolan: *Pong*. Website. 1972. – http://de.wikipedia.org/wiki/Pong; besucht am 03. Oktober 2007. 40

[CBUE02] CZARNECKI, Krzysztof ; BEDNASCH, Thomas ; UNGER, Peter ; EISENECKER, Ulrich W.: Generative Programming for Embedded Software: An Industrial Experience Report. In: BATORY, Don S. (Hrsg.) ; CONSEL, Charles (Hrsg.) ; TAHA, Walid (Hrsg.): *GPCE* Bd. 2487, Springer, 2002. – ISBN 3–540–44284–7, S. 156–172 20, 22

[CE00] CZARNECKI, Krzysztof ; EISENECKER, Ulrich W.: *Generative Programming: Methods, Tools, and Applications*. Boston : Addison-Wesley, 2000 xi, 11

[CN01] CLEMENTS, Paul ; NORTHROP, Linda M.: *Software Product Lines : Practices and Patterns*. Addison-Wesley, 2001 (Professional) vii, 10, 12

[Com01] *Composition Filters*. Website. 2001. – http://trese.cs.utwente.nl/oldhtml/composition_filters/; besucht am 18. Oktober 2007. 25

[CUE04] CZARNECKI, Krzysztof ; AD ULRICH EISENECKER, Simon H.: Staged Configuration Using Feature Models. In: NORD, Robert L. (Hrsg.): *Proceedings of the Third Software Product Line Conference*. Boston, MA : Springer, September 2004 (LNCS 3154), S. 266–283 4, 20

[Ecl07] ECLIPSE FOUNDATION, INC: *Eclipse - an open development platform*. Website. 2007. – `www.eclipse.org`; besucht am 01. Dezember 2007. 71

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995 51, 52

[Her02a] HERRMANN, Stephan: Composable Designs with UFA. In: ALDAWUD, Omar (Hrsg.) ; BOOCH, Grady (Hrsg.) ; CLARKE, Siobhán (Hrsg.) ; ELRAD, Tzilla (Hrsg.) ; HARRISON, Bill (Hrsg.) ; KANDI, Mohamed (Hrsg.) ; STROHMEIER, Alfred (Hrsg.): *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*, 2002 56, 78

[Her02b] HERRMANN, Stephan: Object Teams: Improving Modularity for Crosscutting Collaborations. In: AKSIT, Mehmet (Hrsg.) ; MEZINI, Mira (Hrsg.) ; UNLAND, Rainer (Hrsg.): *NetObjectDays* Bd. 2591, Springer, 2002. – ISBN 3–540–00737–7, S. 248–264 26

[HHM+06] HERRMANN, Stephan ; HUNDT, Christine ; MOSCONI, Marco ; WLOKA, Jan ; PFEIFFER, Carsten: Das Object Teams Development Tooling / Gesellschaft für Informatik. Universität Siegen, Deutschland : Gesellschaft für Informatik, November 2006. – Technical Report

[HHM07] HERRMANN, Stephan ; HUNDT, Christine ; MOSCONI, Marco: Object-Teams/Java Language Definition — version 1.0 / Institut für Softwaretechnik und Theoretische Informatik. 2007 (2007/03). – Forschungsbericht. – ISSN 1436–9915 26, 32

[HK02] HANNEMANN, Jan ; KICZALES, Gregor: Design pattern implementation in Java and aspectJ. In: *OOPSLA*, 2002, S. 161–173 52

[HMPS07] HUNDT, Christine ; MEHNER, Katharina ; PFEIFER, Carsten ; SOKENOU, Dehla: Improving Alignment of Crosscutting Features with Code in Product Line Engineering. In: *Journal of Object Technology* 2 (2007), Nr. 2 55, 106

[Hun03] HUNDT, Christine: *Bytecode-Transformation zur Laufzeitunterstützung von Aspekt-Orientierter Modularisierung mit Object-Teams/Java*, Technische Universität Berlin, Diplomarbeit, Februar 2003

[IGN07] IGN ENTERTAINMENT: *IGN.com Games, Cheats, Movies and more*. Website. 2007. – `http://xbox360.ign.com/index/reviews.html`; besucht am 03. Oktober 2007. 40

[JMa01] *The JMangler Project*. Website. 2001. – `http://roots.iai.uni-bonn.de/research/jmangler/`; besucht am 18. Oktober 2007.

[KCA01] KNIESEL, Günter ; COSTANZA, Pascal ; AUSTERMANN, Michael: Jmangler-A Framework for Load-Time Transformation of Java Class Files. In: *SCAM*, IEEE Computer Society, 2001. – ISBN 0–7695–1387–5, S. 100–110

[KCH+90] KANG, K. ; COHEN, S. ; HESS, J. ; NOVAK, W. ; PETERSON, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Software Engineering Institute, Carnegie Mellon University. 1990 (CMU/SEI-90-TR-21). – Forschungsbericht xi, 4, 17, 19, 20

[KHH+01] KICZALES, Gregor ; HILSDALE, Erik ; HUGUNIN, Jim ; KERSTEN, Mik ; PALM, Jeffrey ; GRISWOLD, William G.: An Overview of AspectJ. In: KNUDSEN, J. L. (Hrsg.): *ECOOP 2001 — Object-Oriented Programming 15th European Conference* Bd. 2072. Budapest, Hungary : Springer-Verlag, Juni 2001, S. 327–353 25

[KKL+98] KANG, Kyo C. ; KIM, Sajoong ; LEE, Jaejoon ; KIM, Kijoo ; SHIN, Euiseob ; HUH, Moonhang: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. In: *Ann. Software Eng* 5 (1998), S. 143–168 4, 46

[KLM+97] KICZALES, G. ; LAMPING, J. ; MENDHEKAR, A. ; MAEDA, C. ; LOPES, C. ; LOINGTIER, J.-M. ; IRWIN, J.: Aspect-Oriented Programming / Xerox PARC. 1997 (SPL97-008 P9710042). – Forschungsbericht 24

[Mos03] MOSCONI, Marco: *Modularisierung und Adaptierung von Komponenteninteraktionen mit Object Teams und dem CORBA Komponentenmodell*, Technische Universität Berlin, Diplomarbeit, Mai 2003

[Obj04] OBJECT MANAGEMENT GROUP: *MOF 2.0 Query / Views / Transformations ad/2002-04-10*. Website. October 2004. – `http://www.omg.org/cgi-bin/apps/doc?ad/2002-04-10`; besucht am 02. Dezember 2007.

[Obj05] OBJECT MANAGEMENT GROUP: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Website. November

2005. – http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01; besucht am 02. Dezember 2007.

[Obj06] OBJECT MANAGEMENT GROUP: *OMG Model Driven Architecture*. http://www.omg.org/mda/. Mai 2006. – besucht am 02. Dezember 2007.

[Obj07a] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Infrastructure*. PDF. Februar 2007. – http://www.omg.org/docs/formal/07-02-04.pdf; besucht am 18. Dezember 2007. 86

[Obj07b] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Superstructure*. PDF. Februar 2007. – http://www.omg.org/docs/formal/07-02-03.pdf; besucht am 18. Dezember 2007. 86

[Rei] REISER, Mark-Oliver: *Product Line Engineering of Automotive Software Systems*. Berlin, Technische Universität Berlin, Diss.. – erscheint voraussichtlich im Mai 2008 79, 81

[Sof96] Software Technology for Adaptable, Reliable Systems (STARS): *Organization Domain Modeling (ODM) Guidebook, Version 2.0*. Juni 1996 15

[Sri99] SRINIVASAN, S.: Design Patterns in Object-Oriented Frameworks. In: *IEEE Computer* 32 (1999), Nr. 2, S. 24–32

[Tor80] TORU, Iwatani: *PacMan*. Website. 1980. – http://de.wikipedia.org/wiki/Pacman; besucht am 03. Oktober 2007. 41

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

_____

Berlin, den / Unterschrift