Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von Dipl.-Wirtschaftsinformatiker (FH) Kai Böllert
geboren am 20. Juni 1973 in Borkum

vorgelegt am 29. Januar 2002

Gutachter:

1. Prof. Dr.-Ing. habil. Ilka Philippow
2. Prof. Dr. Andreas Weber
3. Dr.-Ing. Krzysztof Czarnecki, M.S.

wissenschaftliche Aussprache am 29. November 2002

Verf.-Nr.: IA 101

Kurzfassung

Eine Software-Produktlinie umfaßt eine Gruppe von Software-Systemen, die auf Basis gemeinsam genutzter Komponenten entwickelt werden. Die Systeme entstehen aufgrund der Wiederverwendung in kurzer Zeit. Ihre Entwicklung verursacht verhältnismäßig geringe Kosten, und das Ergebnis ist von hoher Qualität. Vorteile, die eine an einzelnen Systemen ausgerichtete Softwareentwicklung nicht in vergleichbarem Maßstab erzielt. Noch einen Schritt weiter geht die generative Programmierung. Mit ihr können Systeme automatisiert aus Produktlinien generiert werden. Dazu überträgt die generative Programmierung Konzepte der aus der Automobilindustrie bekannten Serienfertigung in die Welt des Software-Engineerings. Die Automatisierung verkürzt erneut Entwicklungszeit und -kosten für Systeme aus der Produktlinie.

Die bisherigen Methoden zur Entwicklung von Produktlinien berücksichtigen die generative Programmierung nicht. Statt dessen werden aus den Produktlinien manuell Systeme entwickelt. Zur Behebung dieses Umstands leistet die vorliegende Arbeit einen Beitrag. Sie stellt HyperFeatuRSEB vor, eine Methode zur objektorientierten Entwicklung von Produktlinien, aus denen Systeme automatisiert in Serie gefertigt werden können. Dazu nutzt die Methode den Hyperspace-Ansatz, eine Technik der generativen Programmierung. Bevor mit diesem Ansatz entwickelt werden kann, müssen seine abstrakten Konzepte und Begriffe auf Modellierungs- und Programmiersprachen abgebildet werden. Bisher existiert eine solche Abbildung für Java. Um mit dem Ansatz Produktlinien nicht nur zu implementieren, sondern auch zu modellieren, entwickelt diese Arbeit Hyper/UML: eine Abbildung des Hyperspace-Ansatzes auf die UML (Unified Modeling Language). Die Serienfertigung übernimmt ein eigens entworfener, für beliebige mit HyperFeatuRSEB entwickelte Produktlinien einsetzbarer Generator. Die Praxistauglichkeit der Methode untermauert eine größere empirische Fallstudie, in der eine Produktlinie für das Gesellschaftsspiel „Die Siedler von Catan“ entwickelt worden ist.

Abstract

A software product line comprises a group of software systems that are developed from a common set of reusable components. Due to reusing the components, the time to develop the systems is short, development costs are relatively low, and the systems exhibit a high quality. Advantages that are not achieved on a comparable scale without product lines. Another step forward is the Generative Programming paradigm. It defines how to automatically generate systems from product lines. To accomplish this task, Generative Programming takes concepts from the batch production of cars in the automobile industry and transfers them to the field of software engineering. The automated generation again reduces development time and costs for the systems of a product line.

Today's methods for developing product lines do not take the Generative Programming paradigm into account. Instead, they describe how to manually develop systems from product lines. This circumstance is alleviated by the dissertation. The dissertation introduces HyperFeatuRSEB, an object-oriented method to develop product lines, from which systems can be produced automatically. The method uses the Hyperspace approach, a Generative Programming technique. However, before the Hyperspace approach can be used for development, its abstract concepts and terms have to be mapped onto modeling and programming languages. Up to now a mapping for Java exists. To use the approach not only for implementing product lines, but also for modeling them, the dissertation introduces Hyper/UML: a mapping of the Hyperspace approach onto the UML (Unified Modeling Language). Batch production is performed by a generator, which was designed to work with every product line developed with HyperFeatuRSEB. To demonstrate the usefulness of the method in practice, an extensive empirical case study was done in the context of the dissertation. In the case study a product line for the parlor game "Die Siedler von Catan" (Settlers of Catan) was developed.

Vorwort

Diese Dissertation entstand am Fachgebiet Prozeßinformatik unter der Leitung von Frau Prof. Dr.-Ing. habil. Ilka Philippow innerhalb des Forschungsprojekts Alexandria, geleitet von Dr.-Ing. Matthias Riebisch. Ziel des Projekts ist die Entwicklung von Methoden für Software-Produktlinien.

Ich möchte mich bei Ilka Philippow und Matthias Riebisch für die Möglichkeit zur Promotion und die Betreuung ihrer Durchführung bedanken; bei Peter Gmilkowsky für die Unterstützung während des Eignungsfeststellungsverfahrens für Fachhochschulabsolventen; bei Martin Halle und Dirk Ulrich für die Mitarbeit in der Entwicklung der Fallstudie „Die Siedler von Catan“; bei Detlef Streitferdt, Ulrich Eisenecker, Krzysztof Czarnecki, Hans Wegener und Sven Macke für interessante Diskussionen, die wertvolle Hinweise für die Dissertation lieferten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Objekte sind wiederverwendbar. Einmal entwickelt sind sie auch in anderen Software-Systemen einsetzbar. Das reduziert die Zeit für Neuentwicklungen, senkt die Kosten und führt zu qualitativ besserer Software. Argumente, mit denen die objektorientierte Gemeinde für eine Abkehr von der strukturierten Programmierung warb. Viele glaubten den Versprechungen und entwickeln fortan ihre Systeme objektorientiert. Allein, Wiederverwendung findet praktisch nicht statt, jedenfalls nicht in einer Größenordnung, ab der die erwähnten Vorteile spürbar werden.

Damit Objekte in einem anderen System als dem ursprünglich vorgesehenen wiederverwendet werden können, müssen sie ein gewisses Maß an Flexibilität besitzen. Wie bestimmt man dieses Maß? Und wer trägt die Kosten für den Aufwand, das Objekt flexibel zu gestalten? Fragen, die *nicht* beantworten kann, wer an der an *einzelnen* Systemen orientierten Vorgehensweise in der Softwareentwicklung festhält – wohl aber jemand, der seinen Entwicklungsstandpunkt verlagert.

Solch ein neuer Standpunkt sind Produktlinien und, damit verbunden, die Möglichkeit zur Serienfertigung von Software-Systemen. Dieser Standpunkt bildet die Grundlage der vorliegenden Arbeit und wird im folgenden motiviert. Im Anschluß werden Probleme der Produktlinien-Entwicklung und Serienfertigung erarbeitet, die zu lösen Ziel der Arbeit ist. In welchen Schritten dies geschieht, beschreibt der Überblick am Ende des Kapitels.

1.1 Motivation

Die heutige Softwareentwicklung bringt zwei Arten von Software-Systemen hervor: Individualsoftware und Standardsoftware. Individualsoftware entsteht in Einzelfertigung und ist maßgeschneidert auf die Anforderungen genau eines Kunden. Standardsoftware hingegen deckt die (vermeintlichen) Anforderungen vieler Kunden gleichzeitig ab. Einmal entwickelt läuft die Produktion von Standardsoftware als Massenfertigung durch Vervielfältigung der Datenträger ab.

Die Konsequenzen: Individualsoftware verursacht hohe Kosten durch eine lange, personalintensive Entwicklung. Standardsoftware ist preiswerter, da die Entwicklungskosten auf alle Kunden umgelegt werden. Allerdings muß der Kunde einer Standardsoftware in Kauf nehmen, daß die Software seine Anforderungen nicht zu 100 Prozent erfüllt und individuelle Anpassungen notwendig sind – sofern die Software dafür ausgelegt ist.

Produktlinien und Serienfertigung

Als Mittelweg zwischen Einzelfertigung und Massenfertigung bietet sich die Serienfertigung an, wie sie z.B. die Automobilindustrie seit einigen Jahren erfolgreich betreibt. Die täglich vom Band rollenden Autos basieren auf einer standardisierten Modell-Plattform, sind aber (innerhalb gewisser Grenzen) gemäß den Wünschen der Kunden individuell ausgestattet.

Den wirtschaftlichen und organisatorischen Rahmen für eine Serienfertigung bildet eine Produktlinie (vgl. Kapitel 2). Aus Sicht des Marketings faßt eine Produktlinie eine Reihe von Produkten zusammen, die über vorher definierte Kriterien zueinander in Beziehung stehen. Solche Kriterien können die Funktion der Produkte oder der von den Produkten angesprochene Kundenkreis sein. Aus technischer Sicht basieren die Produkte einer Linie auf einer gemeinsamen Architektur und werden aus den gleichen Bausteinen gefertigt. Die Automobilindustrie spricht von einer Plattformstrategie.

Produktlinien und die Serienfertigung von Gütern lassen sich auf Software übertragen. Dazu ist eine Software-Produktlinie zu entwickeln. Sie legt die Architektur für alle Software-Systeme der Linie fest, implementiert die Komponenten, aus denen sich die Systeme zusammensetzen, und beinhaltet die Infrastruktur für die Serienfertigung.

Die Entwicklung einer Software-Produktlinie beginnt mit der Beschreibung der Anforderungen für die einzelnen Systeme der Linie. Da die Systeme einander in ihrer Funktion ähneln, kristallisieren sich zwei Arten von Anforderungen heraus: gemeinsame und variable Anforderungen. Erstere sollen alle Systeme erfüllen, letztere nur einzelne oder wenige bestimmte Systeme. In der Entwurfs- und Implementierungsphase entstehen aus den gemeinsamen Anforderungen Komponenten, die in mehr als nur einem System Verwendung finden. Die für die Wiederverwendbarkeit der Komponenten notwendige Flexibilität ergibt sich aus den variablen Anforderungen. Der zusätzliche Aufwand für die Flexibilisierung wird über die Produktlinie abgerechnet und so auf alle Systeme verteilt. Die Infrastruktur für die Serienfertigung besteht aus einem Generator, der ausgehend von Kunden-Anforderungen passende Systeme (teil-)automatisiert aus den Komponenten fertigt.

Die Serienfertigung von Software-Systemen bietet für die Kunden Vorteile hinsichtlich Zeit bis zur Inbetriebnahme der Systeme, Kosten, Qualität, Übereinstimmung von Kunden-Anforderungen mit der Funktionalität der Systeme und Einarbeitungsaufwand für Anwender. Letzterer verringert sich, je stärker Systeme auf die Bedürfnisse der Kunden zugeschnitten sind. Nach diesen fünf Kriterien bewertet Abbildung 1.1 die Entwicklung von Individualsoftware, den Kauf und die Anpassung von Standardsoftware sowie die Serienfertigung von

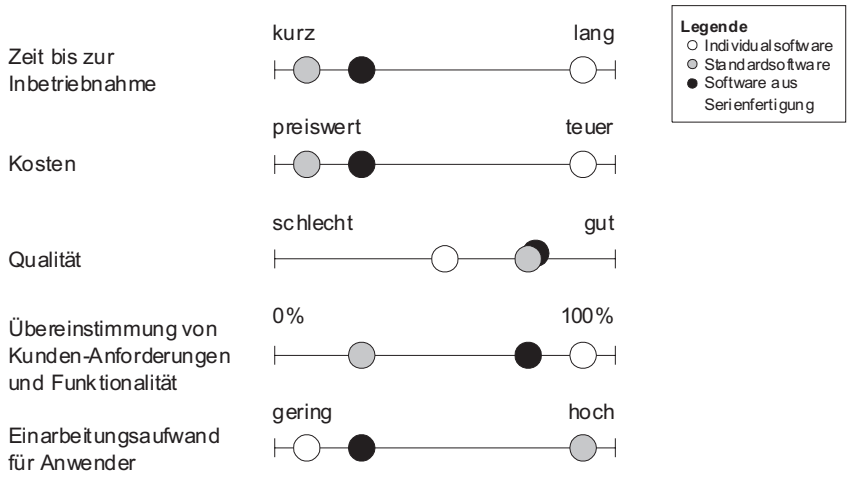

Abbildung 1.1: Kunden-Vorteile bei Serienfertigung von Software-Systemen

Software. Dabei soll die Abbildung verdeutlichen, in welchem Verhältnis die drei Arten von Software zueinander stehen; konkrete Zahlen fehlen deshalb. Aus der Grafik läßt sich ablesen, daß Software aus der Serienfertigung bei keinem einzelnen Kriterium besser abschneidet als eine der beiden anderen Arten von Software. Insgesamt gesehen kommt die Serienfertigung jedoch in allen Kriterien dem Optimum am nächsten und stellt damit wirtschaftlich einen guten Kompromiß dar.

Wer kann von der Serienfertigung von Software-Systemen profitieren? Zwei Szenarien sollen den Nutzen verdeutlichen: für einen Hersteller von Mobiltelefonen und für ein Software-Haus, das betriebswirtschaftliche Lösungen für mittelständische Unternehmen entwickelt.

Szenario 1: Produktlinie für Mobiltelefon-Software

Mobiltelefone weisen je nach Modell unterschiedliche Funktionen auf, z.B. Internetzugang, Kalender, Konferenzschaltung usw., intern realisiert durch Hardware- und Software-Komponenten. Um Zeit und Kosten bei der Entwicklung der Modelle einzusparen und die Produktion der Telefone zu vereinfachen, könnte der Hersteller alle Modelle mit derselben Hardware und Software ausstatten. Funktionen, die ein Modell dem Kunden nicht bietet, deaktiviert die Benutzeroberfläche.

Allerdings entscheiden Kunden sich nicht allein aufgrund der Funktionen für oder gegen ein Modell. Relevant sind auch Größe, Gewicht und Akku-Laufzeit eines Mobiltelefons. Werte, die jede zusätzliche, deaktivierte Funktion negativ beeinflußt. Deshalb strebt der Hersteller an, nur die von einem Modell vorgesehenen Funktionen in die Telefone einzubauen. Dies kann er erreichen, indem er für jedes Modell Hardware und Software individuell entwickelt. Die Folge wären lange Entwicklungszeiten, hohe Kosten und ein komplexer Produktionsprozeß.

Die Lösung liegt in einer Produktlinie, die alle Mobiltelefonmodelle zusammenfaßt. Die Wiederverwendung von Hardware- und Software-Komponenten über Modellgrenzen hinweg spart Entwicklungskosten ein und versetzt den Hersteller in die Lage, neue Modelle schneller als die Konkurrenz auf den Markt zu bringen. Damit die Mobiltelefone möglichst wenig Ressourcen verbrauchen, steuert der Hersteller den Produktionsprozeß so, daß nur benötigte Komponenten in die Telefone eingebaut werden.

Szenario 2: Produktlinie für betriebswirtschaftliche Software

Wettbewerb und Konkurrenzdruck zwingen inzwischen auch kleine und mittlere Unternehmen dazu, ihre Geschäftsprozesse durch die Einführung von IT-Technologie effizienter abzuwickeln. Standardsoftware wie SAP kommt für diese Unternehmen selten in Frage, da die damit einhergehende Umstrukturierung der über Jahre gewachsenen Aufbau- und Ablauforganisation die Unternehmen überfordert. Darüber hinaus ist SAP ein sehr komplexes System, so daß trotz Schulung der Mitarbeiter anfangs Produktivitätsverluste auftreten. Ebensowenig gefragt sind individuelle Lösungen, da die Unternehmen die Kosten für die teure Entwicklung kaum aufbringen können.

Hier bietet sich Software-Häusern ein interessantes Geschäftsfeld. Sie entwickeln eine Produktlinie für betriebswirtschaftliche Software-Systeme und fertigen darauf aufbauend in kurzer Zeit preiswerte, maßgeschneiderte, schlanke IT-Lösungen für mittelständische Unternehmen. Der Fertigungsprozeß läuft in diesem Szenario zumeist teilautomatisiert ab, da die Produktlinie die Vielfalt an Kunden-Anforderungen nicht vorsehen kann und das Software-Haus die eine oder andere Komponente doch individuell entwickeln muß. Ist diese spezielle Komponente für weitere Kunden von Interesse, sollte das Software-Haus die Komponente im nachhinein in die Produktlinie integrieren. Die Produktlinie wächst und deckt im Laufe ihres Lebens immer mehr Anforderungen ab.

Zusammenfassend läßt sich sagen, daß für viele Unternehmen die Entwicklung von Produktlinien verstärkt anzustreben ist. Bei der Umsetzung in die Praxis zeigen sich aber noch Probleme.

1.2 Problemstellung

Die Entwicklung einer Software-Produktlinie ist eine Aufgabe mit vielen Facetten: Wie läuft der Entwicklungsprozeß ab? Welche Team-Organisation eignet sich am besten? Wie beschreibt man die Produktlinien-Architektur? Wie modelliert und implementiert man ihre Komponenten? Wie bleibt die Komplexität der Produktlinie beherrschbar? Wie baut man den Generator für die Serienfertigung der Systeme? Was ist bei der Wartung zu beachten? Damit nicht jeder, der Produktlinien entwickeln will, diese Fragen von neuem stellen muß, sollten die besten Antworten zusammengestellt und in eine Entwicklungsmethode für Produktlinien überführt werden.

Das Aufstellen einer völlig neuen Produktlinien-Methode wäre jedoch eine sehr aufwendige Angelegenheit. Deshalb ist es besser, aus der Menge bereits be-

stehender Software-Entwicklungsmethoden eine passende Methode aufzugreifen und derart zu erweitern beziehungsweise abzuwandeln, daß sie auch für die Entwicklung von Produktlinien einsetzbar ist. Ein weiterer Vorteil dieser Vorgehensweise ist die Akzeptanz der Produktlinien-Methode bei Projektleitern und Entwicklern: Wer die Original-Methode kennt, muß beim Umstieg auf die Produktlinien-Entwicklung nicht gänzlich umlernen.

FeatuRSEB (sprich: *featured RSEB*) ist eine Produktlinien-Methode, die auf vorhandenem aufsetzt. Sie basiert schwerpunktmäßig auf der OOSE-Methode (Object-Oriented Software Engineering) und unterstützt dadurch die objektorientierte Entwicklung von Produktlinien [JCJÖ92, GFA98]. Leider beantwortet FeatuRSEB nicht alle aufgeworfenen Fragen zufriedenstellend (vgl. Kapitel 4):

- *Serienfertigung von Systemen.* FeatuRSEB sieht nicht vor, daß auf Basis von Produktlinien Systeme automatisiert in Serie gefertigt werden sollen. Statt dessen werden die Komponenten einer Produktlinie manuell zu Systemen zusammengesetzt. Entsprechend höher ist der Aufwand an Zeit und Kosten. Das in Abbildung 1.1 auf Seite 3 gezeigte Verhältnis von Software aus Serienfertigung zu Individualsoftware verschlechtert sich.

- *Komplexität entwickelter Produktlinien.* Die Serienfertigung erfordert kleine Komponenten, damit der Generator bei der Fertigung wirklich nur die benötigte, vom Kunden gewünschte Funktionalität in ein System übernehmen kann. Die Modellierung und Implementierung derart feingranularer Komponenten führt in FeatuRSEB jedoch zu Produktlinien, deren Komplexität kaum mehr handhabbar ist.

- *Wartbarkeit entwickelter Produktlinien.* In der Wartungsphase verhindert die hohe Komplexität den Ausbau einer Produktlinie. Neue Kundenkreise bleiben verschlossen, und die Produktlinie wird unter Umständen zu einem wirtschaftlichen Risiko für das Unternehmen.

Eine Lösung für die Probleme FeatuRSEBs bietet die generative Programmierung. Sie beschäftigt sich mit der Generierung von Software-Systemen auf Basis von Produktlinien und stellt dafür einige Techniken bereit, unter anderem den Hyperspace-Ansatz. Er definiert ein allgemeines Modell, wie Komponenten in allen Phasen eines Softwareentwicklungsprozesses feingranular modelliert und implementiert werden können. Darüber hinaus verfügt der Ansatz über einen Mechanismus, der die Komponenten wieder zu Systemen zusammensetzt [CE00, OT01]. Dennoch bleiben auch hier Antworten offen (vgl. Kapitel 5):

- *Entwicklung von Produktlinien-Komponenten.* Der Hyperspace-Ansatz legt nur abstrakte Konzepte zur Entwicklung feingranularer Komponenten fest. Eine konkrete Modellierungs- oder Programmiersprache schreibt er nicht vor. Um den Hyperspace-Ansatz in FeatuRSEB nutzen zu können, müssen seine Konzepte auf die dort verwendeten Sprachen abgebildet werden. Die Modellierung geschieht in FeatuRSEB mit der UML (Unified Modeling Language) [OMG01], die Implementierung mit objektorientierten Programmiersprachen.

- *Integration in eine Methode.* Der Hyperspace-Ansatz verfolgt bei der Zerlegung von Kunden-Anforderungen in Komponenten eine etwas andere Vorgehensweise als FeatuRSEB. Die Nutzung des Hyperspace-Ansatzes in FeatuRSEB erfordert deswegen die Integration beider Vorgehensweisen im Rahmen einer neuen Methode.

Eine Abbildung des Hyperspace-Ansatzes auf Java, genannt Hyper/J, existiert bereits [TO01]. Die Implementierung feingranularer Komponenten ist also, zumindest in Java, möglich. Die weiteren, noch fehlenden Antworten zu geben, ist Ziel dieser Arbeit.

1.3 Zielsetzung

Diese Arbeit liefert einen Beitrag zur objektorientierten Entwicklung von Software-Produktlinien. Der Beitrag besteht aus zwei Teilen:

1. *HyperFeatuRSEB* (sprich: *hyper-featured RSEB*) ist eine Produktlinien-Methode, die, wie der Name andeutet, auf die FeatuRSEB-Methode aufsetzt. HyperFeatuRSEB löst die Probleme FeatuRSEBs durch Integration des Hyperspace-Ansatzes (vgl. Kapitel 7).

2. *Hyper/UML* ist eine Abbildung des Hyperspace-Ansatzes auf die UML. Mit ihr gelingt es, Produktlinien-Komponenten feingranular mit der UML zu modellieren und zu Systemen zusammenzusetzen (vgl. Kapitel 6).

Die Implementierung von Produktlinien-Komponenten übernimmt in HyperFeatuRSEB Hyper/J oder eine zukünftige Abbildung des Hyperspace-Ansatzes auf eine beliebige andere objektorientierte Programmiersprache. Insgesamt soll der Beitrag die folgenden Ziele erfüllen:

- *Serienfertigung schlanker Systeme.* Die mit HyperFeatuRSEB entwickelten Produktlinien eignen sich zur Serienfertigung von Systemen. Ein Generator setzt die Produktlinien-Komponenten automatisiert zu Systemen zusammen. Die Systeme sind schlank. Sie enthalten nur genau die Funktionalität beziehungsweise Komponenten, die die Kunden gefordert haben.

- *Komplexität entwickelter Produktlinien.* Die Komplexität der mit HyperFeatuRSEB entwickelten Produktlinien bleibt trotz feingranular modellierter und implementierter Komponenten handhabbar und damit für Entwickler beherrschbar.

- *Wartbarkeit entwickelter Produktlinien.* Die Wartbarkeit der mit HyperFeatuRSEB entwickelten Produktlinien, d.h. ihr Ausbau, wird nicht durch die hohe Komplexität der Produktlinien verhindert, sondern ist in vielen Fällen sogar vergleichsweise einfach möglich.

- *Skalierbarkeit der Methode.* Entwicklungsmethoden eignen sich nicht für Produktlinien jeder Größe, nicht für jede Organisationsstruktur und Programmiertechnik. Mit HyperFeatuRSEB können Teams mit bis zu 3 Personen Produktlinien objektorientiert mindestens bis zu einem Umfang von 80 Use-Cases und 300 Klassen entwickeln. Damit skaliert die Methode besser als FeatuRSEB.

Ob die hier vereinbarten Ziele für die HyperFeatuRSEB-Methode in Theorie und Praxis tatsächlich eintreten, überprüft Kapitel 8.

1.4 Aufbau der Arbeit

Die Arbeit ist in zwei Teile gegliedert. Der erste Teil umfaßt die Kapitel 2 bis 5, in denen der Stand der Technik analysiert und mit Blick auf die Zielsetzung der Arbeit bewertet wird. Kapitel 2 vertieft die bisherigen Erläuterungen zu Produktlinien, Entwicklungsmethoden und der Serienfertigung von Software-Systemen. Das Kapitel begründet außerdem, warum diese Arbeit gerade auf die FeatuRSEB-Methode und den Hyperspace-Ansatz aufbaut und nicht auf eine andere Entwicklungsmethode beziehungsweise eine andere Technik der generativen Programmierung. Kapitel 3 gibt einen Überblick zu einer Fallstudie, die im Rahmen der Arbeit durchgeführt wurde. Inhalt der Studie war die Entwicklung einer Produktlinie mit HyperFeatuRSEB für das Gesellschaftsspiel „Die Siedler von Catan“. Kapitel 4 und 5 beschließen den ersten Teil mit einer ausführlichen Beschreibung und Bewertung von FeatuRSEB und dem Hyperspace-Ansatz.

Der zweite Teil besteht aus den Kapiteln 6 bis 9. Sie enthalten den eigentlichen Beitrag dieser Arbeit zum Gebiet der objektorientierten Entwicklung von Produktlinien. Kapitel 6 stellt Hyper/UML vor, Kapitel 7 HyperFeatuRSEB. Illustriert werden beide Kapitel durch Auszüge aus der Fallstudie „Die Siedler von Catan“. In Kapitel 8 folgt die Überprüfung der Zielsetzung. Hierfür wird erneut die Fallstudie herangezogen. Schlußfolgerungen und ein Ausblick auf weiterführende, zukünftige Themen runden die Arbeit in Kapitel 9 ab.

Wer wenig Zeit zur Verfügung hat und sich schnell, aber dennoch umfassend über den Inhalt der Arbeit informieren möchte, dem sei das Lesen zumindest folgender Kapitel und Unterkapitel empfohlen: 2, 3, 4.5, 5.2, 5.4 und 7. Kennt der Leser das Spiel „Die Siedler von Catan“ schon, kann er zusätzlich Kapitel 3 überspringen.

Zur Verwendung von Begriffen: Die dominierende Sprache in der Informatik und im Software-Engineering ist Englisch. Demzufolge tragen viele Konzepte englische Bezeichnungen. Eine passende deutsche Übersetzung fehlt häufig, insbesondere für neuartige Konzepte. Autoren, die ihre Arbeiten in deutsch verfassen, stehen somit vor dem Problem, wie sie mit englischen Begriffen umgehen sollen. Das eine Extrem ist, alle Begriffe einzudeutschen. Dies schafft künstliche, nicht unbedingt gebräuchliche Übersetzungen und erschwert das Verständnis beim Leser. Das andere Extrem ist ein Text voller Anglizismen, worunter der Lesefluß leidet. Beim Schreiben dieser Arbeit wurde ein Mittelweg zwischen beiden Extremen beschritten. Wo immer sich eine Übersetzung für einen englischen Begriff in Büchern und Zeitschriften bereits durchgesetzt hat, wird sie

verwendet. Beispiele sind Entwurfsmuster für *design pattern* oder deutsche Begriffe für Elemente der UML wie sie [UAD] und [HK99] vorschlagen. Gibt es keine gebräuchliche Übersetzung, wurde von Fall zu Fall entschieden, ob die Arbeit den englischen Begriff beibehält oder eine Übersetzung neu einführt. So bleibt z.B. Framework erhalten, wohingegen Begriffe für neuere Konzepte aus der Forschung, beispielsweise aus dem Hyperspace-Ansatz, übersetzt wurden.

Zum Schriftsatz: Wird im Fließtext auf Elemente in Abbildungen oder auf Elemente des Metamodells der UML Bezug genommen, sind die Namen der Elemente in einer serifenlosen Schriftart gesetzt. Dieselbe Schriftart wird für Quellcode verwendet, der in den Fließtext eingebettet ist.

Kapitel 2

Produktlinien, Entwicklungsmethoden, Serienfertigung

Dieses Kapitel führt in die Thematik der Arbeit ein. Nach einem Überblick zum Stand der Technik in der objektorientierten Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen trifft das Kapitel eine Vorauswahl bestimmter Entwicklungsmethoden und Implementierungsansätze, die dann die Kapitel 4 und 5 weiter vertiefen.

2.1 Produktlinien

Wozu überhaupt Produktlinien als Software? Welche Probleme lösen sie, welche bringen sie mit sich? Woraus besteht eine Software-Produktlinie? Die Antworten gibt dieses Unterkapitel.

2.1.1 Probleme der heutigen Softwareentwicklung

Ein Blick auf die gängige Praxis in der Softwareentwicklung zeigt: Software-Systeme werden im Rahmen von Projekten realisiert. Ziel jedes Projekts ist die Entwicklung und Auslieferung eines *einzelnen* Systems – innerhalb der vorgesehenen Zeit und unter Einhaltung der geplanten Kosten. Alle Entscheidungen, die Projektleiter und Entwickler während des Projekts treffen, werden durch diese Zeit- und Kosten-Restriktion stark beeinflußt. Zusätzlicher Nutzen, der für andere Systeme oder zukünftige Versionen oder Varianten desselben Systems relevant ist (d.h. deren Entwicklung beschleunigt oder verbessert), wird nicht erwirtschaftet, weil dies jetzt Zeit braucht, jetzt Kosten verursacht und das aktuelle Projekt so Gefahr läuft, zu spät und zu teuer ausgeliefert zu werden. Die Fokussierung auf kurzfristige Projektziele ist Ursache für viele Probleme der heutigen Softwareentwicklung:

- *Langsame, teure Neuentwicklung.* Die Entwicklung eines Systems beginnt im wesentlichen ständig bei Null. Zwar kann ein geringer Teil der Funktionalität durch Einbinden von Bibliotheken abgedeckt werden, aber für den verbleibenden, größeren Rest werden jedesmal wieder die gleichen oder ähnliche Algorithmen wie im vorherigen Projekt implementiert. Die Folge sind lange Entwicklungszeiten und hohe Kosten.

- *Ungenügende Qualität.* Der Beginn bei Null führt auch dazu, daß bei der Implementierung der Algorithmen immer wieder die gleichen Fehler gemacht werden. Diese Fehler alle zu finden und auszubessern würde soviel Zeit verbrauchen, daß Projekte ihren Auslieferungstermin verpassen. Um den Termin dennoch zu halten, wird die erste Version eines Systems häufig fehlerhaft ausgeliefert. Erst in der zweiten oder dritten Version erreicht die Qualität des Systems dann einen akzeptablen Stand.

- *Schlechte Wartbarkeit.* Der Lebenszyklus eines Software-Systems erstreckt sich über einen langen Zeitraum, währenddessen das System gewartet und weiterentwickelt werden muß, weil sich das Umfeld des Systems verändert. Bei der Neuentwicklung eines Systems sollte also dessen leichte Wartbarkeit ein wichtiges Kriterium sein. In der Praxis fällt Wartbarkeit häufig den oben genannten kurzfristigen Projektzielen zum Opfer, wodurch später hohe Wartungskosten anfallen.

Am Ende zahlt der Kunde die „Zeche“. Er bekommt seine Software spät und zu einem hohen Preis. Und bis die Software endlich reif ist für den Produktiveinsatz, geht weitere Zeit für Nachbesserungen verloren. Bei der Wartung zahlt der Kunde erneut für Versäumnisse in der Vergangenheit. Entschärfen kann diese Probleme die Wiederverwendung von Ergebnissen aus bisher durchgeführten Projekten.

2.1.2 Probleme der Wiederverwendung

Wiederverwendung heißt, einmal entwickelte und getestete Projektergebnisse in zukünftigen Projekten ebenfalls zu nutzen. Jede Wartung eines mehrfach genutzten Ergebnisses erfolgt nur einmal für alle Nutzer. Dadurch lassen sich qualitativ gute Systeme schneller und kostengünstiger entwickeln und warten.

Allerdings sind Projektergebnisse selten per se wiederverwendbar. Kein System, kein Einsatzkontext für wiederzuverwendende Ergebnisse stimmt mit einem anderen *genau* überein. Deshalb müssen wiederverwendbare Ergebnisse flexibel an ihren jeweiligen Einsatzkontext anpaßbar sein. Beispiel: Eine Komponente, die Löhne und Gehälter von Arbeitnehmern berechnet, soll in mehreren Unternehmen eingesetzt werden. Da die Unternehmen ihre Arbeitsentgelte unterschiedlich berechnen, muß die Komponente einen Mechanismus (Parameter) zur flexiblen Einstellung des gewünschten Berechnungsverfahrens anbieten.

Projektergebnisse flexibel und parametrisierbar zu halten, braucht Zeit und verursacht Kosten. Um Ergebnisse überhaupt wiederverwenden zu können, muß also zunächst einmal zusätzlicher Aufwand investiert werden. Aufwand, den jenes Projekt zu tragen hat, das morgen auszuliefern ist und aus dem Aufwand keinen

Gewinn zieht. Wiederverwendung hat etwas paradoxes an sich: Jeder möchte gerne Ergebnisse wiederverwenden. Aber niemand ist aufgrund des Zeit- und Kostendrucks in der Lage, in seinem aktuellen Projekt Mittel bereitzustellen, damit wiederverwendbare Ergebnisse entstehen.

Angenommen, es existiert ein organisatorischer Rahmen, innerhalb dessen solche Mittel zur Verfügung stehen. Dann gilt es, technische Probleme bei der Nutzung und Entwicklung wiederverwendbarer Ergebnisse zu meistern. Welche Probleme dies sind, beschreiben die folgenden zwei Abschnitte am Beispiel von Komponenten und Frameworks.

Nutzung von Komponenten und Frameworks

Eine Komponente ist immer mit Hinblick auf einen bestimmten Einsatzkontext implementiert. Damit die Komponente ihre Aufgaben erledigen kann, erwartet sie von ihrem Umfeld: die Existenz weiterer Komponenten und ein bestimmtes Protokoll, um mit diesen zu kommunizieren; eine Technik zur Speicherverwaltung; eine Strategie zur Behandlung von Fehlern; einen Mechanismus zur Speicherung ihres Zustands usw. Erfüllt nun ein System, das die Komponente nutzen möchte, nicht genau die Erwartungen der Komponente, kann die Komponente gar nicht oder nur nach Anpassungen genutzt werden. System-Architektur und Komponenten-Architektur harmonieren nicht miteinander. Dieses Phänomen wird *architectural mismatch* genannt [GAO95].

Ein Framework besteht aus einer Architektur sowie abstrakten und konkreten Komponenten. Ein System, das ein Framework nutzt, verwendet Architektur und Komponenten wieder. Die Abstraktionen werden bei einem White-Box-Framework um systemspezifisches Verhalten verfeinert, bei einem Black-Box-Framework mit einer passenden, vom Framework angebotenen konkreten Komponente belegt. Der Steuerfluß in einem solchen System ist durch die Framework-Architektur vorgegeben und wird zur Laufzeit vom Framework kontrolliert. Dies ist solange unproblematisch, wie das System nur ein Framework nutzt. Bei Nutzung mehrerer Frameworks mit unterschiedlichen Architekturen konkurrieren die Frameworks um den Steuerfluß, was in der Regel aufwendige Anpassungen erfordert – sofern der Quellcode der Frameworks verfügbar ist [Bos00, Unterkapitel 11.3].

Entwicklung von Komponenten und Frameworks

Wer Komponenten und Frameworks entwickeln will, dem stellen sich eine Reihe von Fragen: Welches Maß an Flexibilität müssen die Komponenten aufweisen, damit sie in möglichst vielen anderen Systemen sinnvoll genutzt werden können? Über welche Parameter sollen die Nutzer die Flexibilität der Komponenten einstellen? Wie ist die Architektur der Frameworks zu wählen, damit die Frameworks einen möglichst großen Kreis potentieller Nutzer ansprechen? Welche Abstraktionen soll das Framework zur Verfeinerung anbieten?

Die Fragen sind nicht leicht zu beantworten, da bei der Entwicklung von Komponenten und Frameworks meist noch nicht bekannt ist, welche Systeme diese einmal nutzen werden. Zwei Strategien, um mit dieser Unsicherheit umzugehen,

sind: (a) Parameter und Abstraktionen nach bestem Wissen beziehungsweise spekulativ in Komponenten und Frameworks einbauen; (b) Parametrisierbarkeit und Abstraktionen erst bei Bedarf vornehmen (Motto: *„Frameworks are grown, not born“*). Die Gefahr bei Strategie (a) ist, daß wichtige Parameter und Abstraktionen vergessen oder zu viele unwichtige in Komponenten und Frameworks eingebaut werden. Beides wirkt sich negativ auf die Wiederverwendbarkeit aus: Ersteres schränkt den Kreis der Nutzer ein; letzteres erhöht die Komplexität, wodurch Komponenten und Frameworks für die Nutzer schwerer zu verstehen und einzusetzen sind [Fow99, Seite 83]. Bei Strategie (b) ziehen neue Parameter und Abstraktionen häufig umfangreiche Änderungen in der Architektur von Komponenten und Frameworks nach sich, was zur raschen Degeneration der Architektur führen kann.

Komponenten und Frameworks für Geschäftsprozesse und -objekte

Trotz aller technischen Probleme gibt es erfolgreiche Beispiele für die Wiederverwendung. Der Einsatz von großen Komponenten wie Betriebssystemen, Datenbanksystemen und Web-Servern oder von Frameworks für grafische Benutzeroberflächen ist heutzutage selbstverständlich. Damit ist das Potential der Wiederverwendung jedoch längst nicht ausgeschöpft. Der Bereich, in dem sie bisher versagt, sind Komponenten und Frameworks für Geschäftsprozesse und -objekte. Warum ist es so schwierig, hier Wiederverwendung zu erreichen?

Ein Grund ist die Vielfalt an Anforderungen. Abgesehen von Abläufen, die gesetzlich vorgeschrieben sind (z.B. in der Buchführung), gibt es zwischen den Unternehmen erhebliche Unterschiede, was ihre Geschäftsprozesse und die darin verarbeiteten Daten betrifft. Der Versuch externer Anbieter, möglichst viele dieser Unterschiede zu berücksichtigen, führt zu sehr komplexen, schwer verständlichen Komponenten und Frameworks. Ein bekanntes Beispiel ist IBMs San-Francisco-Framework [MCD00]. Aber auch innerhalb von Unternehmen sind Geschäftsobjekte meist ungenügend standardisiert, so daß sich die Entwicklung von Komponenten und Frameworks für IT-Abteilungen nicht lohnt.

Ein weiterer Grund für die fehlende Wiederverwendung ist das externe Angebot wiederverwendbaren Materials. Je stärker intern entwickelte Komponenten und Frameworks die Kerngeschäftsfelder und Kernkompetenzen eines Unternehmens betreffen, desto weniger ist das Unternehmen daran interessiert, die Komponenten auf einem Marktplatz seiner Konkurrenz anzubieten. Denn es sind ja genau diese Komponenten, die dem Unternehmen den entscheidenden Vorsprung gegenüber den Wettbewerbern sichern. Was bleibt, ist die interne Entwicklung solcher Komponenten und Frameworks, vorausgesetzt man löst sich von der an einzelnen Systemen orientierten Herangehensweise in der heutigen Softwareentwicklung und meistert die technischen Probleme der Wiederverwendung.

2.1.3 Produktlinien als Lösung

Der Begriff der Produktlinie stammt aus der Betriebswirtschaftslehre. Dort findet er vor allem im Bereich Marketing Verwendung, was sich in seiner Definition widerspiegelt:

> „Eine *Produktlinie* ist eine Gruppe von Produkten, die in enger Beziehung zueinander stehen, da sie eine ähnliche Funktion erfüllen, an dieselben Zielgruppen verkauft werden, über dieselben Arten von Distributionspunkten verteilt werden oder in eine bestimmte Preisklasse fallen." [KB99, Seite 680]

Die Software-Engineering-Gemeinde übernimmt diese Definition, bereinigt sie aber um die vertriebsspezifischen Aspekte und reduziert sie so auf die Ähnlichkeit der Produkte bezüglich ihrer Funktion. Die Ähnlichkeit der Produkte weist darauf hin, daß die Produkte der gleichen Domäne angehören:

> Eine *Domäne* ist „an area of knowledge (...) characterized by a set of concepts and terminology understood by practitioners in that area" [CN99, Seite 156].

Beispiele für geschäftliche Domänen sind das Bank- und Versicherungswesen, für softwaretechnische Domänen Betriebs- und Datenbanksysteme sowie für mathematische Domänen Graphentheorie und lineare Algebra. Jede Domäne kann hierarchisch weiter aufgegliedert werden. Untergeordnete Domänen des Versicherungswesens sind Krankenversicherung, Lebensversicherung, Sachversicherung usw.

Technisch läßt sich die Ähnlichkeit der Produkte einer Produktlinie in wiederverwendbaren Komponenten und Frameworks implementieren, die dann bei der Entwicklung der Produkte genutzt werden. Diese Erkenntnis bringt die Definition des Begriffs der Software-Produktlinie zum Ausdruck:

> Eine *Software-Produktlinie* ist eine Gruppe von Software-Produkten beziehungsweise -Systemen, die in enger Beziehung zueinander stehen, da sie der gleichen Domäne angehören, eine ähnliche Funktion erfüllen, die gleiche Architektur aufweisen und auf Basis gemeinsam genutzter Komponenten entwickelt werden.

Die Definition gilt lediglich für die vorliegende Arbeit, da der Begriff Software-Produktlinie in der Literatur nicht einheitlich gebraucht wird. Als Synonyme finden sich Produktfamilie oder Systemfamilie, z.B. in [CN99] beziehungsweise [CE00]. *Hinweis zur Terminologie: Ab hier bezeichnet der Begriff Produktlinie immer eine Software-Produktlinie. Die Produkte einer Produktlinie sind demnach immer Software-Systeme.*

Im Gegensatz zur gängigen Praxis in der Softwareentwicklung steht bei der Entwicklung von Produktlinien nicht mehr das Wohl einzelner Projekte beziehungsweise Systeme im Vordergrund, sondern das der gesamten Produktlinie beziehungsweise aller daraus entwickelten Systeme. Dadurch können Produktlinien, anders als einzelne Projekte, einen Zeit- und Kosten-Rahmen bereitstellen, innerhalb dessen wiederverwendbare Komponenten und Frameworks für die Geschäftsprozesse und -objekte eines Unternehmens systematisch entwickelt und genutzt werden können. Wiederverwendung bekommt einen wirtschaftlichen Kontext, kann dem Management gegenüber als strategische Investition

verständlich gemacht werden und wird in der Organisationsstruktur des Unternehmens verankert. Produktlinien lösen auch die technischen Probleme bei der Wiederverwendung von Komponenten und Frameworks. Mehr dazu im nächsten Abschnitt 2.1.4.

Zusammenfassend die Vorteile von Produktlinien: Durch den hohen Grad an Wiederverwendung können Systeme schnell, kostengünstig und mit hoher Qualität entwickelt werden. Die Zeit bis zur Marktreife (engl.: *time-to-market*), also bis ein System entwickelt, ausgeliefert und beim Kunden in Betrieb genommen ist, verkürzt sich. Die Wartungskosten sinken, weil Änderungsanforderungen nur in die Produktlinie eingearbeitet werden müssen. Die Zufriedenheit der Kunden steigt.

Keineswegs sind Produktlinien jedoch die Lösung für alle Probleme in der Softwareentwicklung. Neben bestimmten Voraussetzungen, ohne deren Vorliegen der Umstieg auf Produktlinien wirtschaftlich riskant ist (vgl. Abschnitt 2.2.1), ist am Anfang der Produktlinien-Entwicklung mit folgenden Nachteilen zu rechnen: Der Aufbau einer Produktlinie (Spezifikation der Architektur, Entwicklung der gemeinsamen Komponenten) nimmt Zeit in Anspruch und erfordert hohe Vorabinvestitionen. Weiterhin wirft die Produktlinie während der Aufbauphase keine Erträge ab. Erst müssen Architektur und Komponenten vorhanden sein, bevor auf Basis der Produktlinie Systeme entwickelt und ausgeliefert werden können. Diese Nachteile können durch eine bestimmte Strategie beim Umstieg abgemildert werden (vgl. Abschnitt 2.2.2). Ist der Umstieg erfolgt, treten die Nachteile zunehmend in den Hintergrund, so daß die Vorteile auf lange Sicht gesehen überwiegen.

2.1.4 Produktlinien-Architektur und -Komponenten

Woher kommen eigentlich Architektur und Komponenten einer Produktlinie? Sie ergeben sich aus den Anforderungen an die Produktlinie. Wer definiert diese Anforderungen? Der Kreis potentieller Kunden, die Systeme auf Basis der Produktlinie entwickelt haben möchten. Wie kann ein Unternehmen den Umfang dieses Kreises bestimmen? Durch Marktforschung, also Identifikation der Nachfragerwünsche und Beobachtung der Konkurrenten.

Nicht alle (potentiellen) Kunden stellen die gleichen Anforderungen. Deshalb bilden sich in der Menge der Produktlinien-Anforderungen verschiedene Schnittmengen, aus denen sich zwei Kategorien von Anforderungen ableiten lassen [WL99]:

1. *Gemeinsame Anforderungen* müssen alle Systeme einer Produktlinie erfüllen. Sie machen die *Gemeinsamkeit* beziehungsweise Kernfunktionalität der Produktlinie aus.

2. *Variable Anforderungen* muß nicht jedes System einer Produktlinie erfüllen. Sie verdeutlichen, welche Unterschiede die Systeme in der Produktlinie aufweisen können. Deshalb werden sie auch als Veränderlichkeit oder *Variabilität* der Produktlinie bezeichnet.

Ausgehend von diesen Anforderungen können Architektur und Komponenten der Produktlinie abgeleitet werden. Die Architektur richtet sich primär an der Gemeinsamkeit der Produktlinie aus. Sie setzt die Komponenten zueinander in Beziehung und legt ihre Schnittstellen fest. Systeme, die auf Basis der Produktlinie entwickelt werden, instanziieren die Architektur und nutzen die benötigten Komponenten.

Nicht immer können jedoch gemeinsame und variable Anforderungen auf klar voneinander getrennte Komponenten abgebildet werden, d.h. auf gemeinsame *oder* variable Komponenten, von denen letztere je nach zu entwickelndem System weggelassen werden. Aus Gründen der Modularisierung entstehen Mischformen, also Komponenten, die beide Arten von Anforderungen enthalten. Diese Misch-Komponenten müssen flexibel für ihre Nutzung in den Systemen parametrisierbar sein. Im Gegensatz zu der in Abschnitt 2.1.2 geschilderten Situation ist jedoch klar, welches Maß an Flexibilität und welche Parameter notwendig sind. Beides ergibt sich aus den variablen Anforderungen.

Desweiteren besteht bei den Komponenten einer Produktlinie nicht die Gefahr des *architectural mismatch*. Der Einsatzkontext der Komponenten ist immer ein aus der Produktlinie entwickeltes System. Dieses folgt derselben Architektur, in die auch die Komponenten eingeordnet sind.

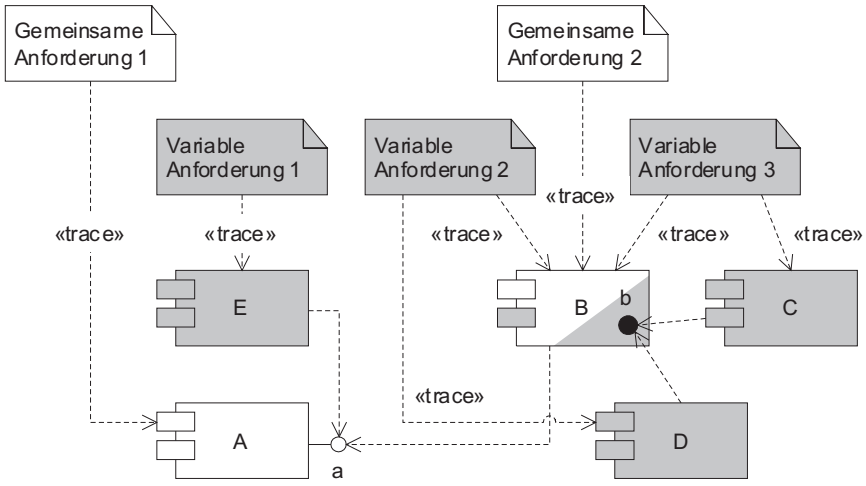

Abbildung 2.1: Produktlinien-Architektur und -Komponenten

Abbildung 2.1 zeigt beispielhaft eine Produktlinien-Architektur mit fünf Komponenten (A bis E) in UML-Notation[1]. Die Gemeinsamkeit der Produktlinie bilden die zwei aus den gemeinsamen Anforderungen hervorgegangenen Komponenten A (mit der Schnittstelle a) und B (mit dem Parameter b). Zur Variabilität zählen die mit den variablen Anforderungen verknüpften Komponenten C, D und E. B ist eine Misch-Komponente, deren Parameter b ein System entweder

[1] Die Notation der Komponente B mit ihrem Parameter b (gefüllter Kreis) ist an die Notation von Variationspunkten in der Produktlinien-Entwicklungsmethode FeatuRSEB angelehnt (vgl. [JGJ97, Unterkapitel 4.9] und Abschnitt 4.2.3).

mit C oder D belegen muß. E ist eine variable Komponente; sie nehmen nur jene Systeme auf, die die variable Anforderung 1 erfüllen sollen.

Zur Implementierung einer Produktlinien-Architektur eignet sich auch ein Framework, dessen Abstraktionen die Variabilität der Produktlinie realisieren. Bei der Entwicklung des Frameworks tritt nicht das in Abschnitt 2.1.2 angesprochene Problem des Findens der richtigen Architektur und Abstraktionen auf. Beide können systematisch aus den gemeinsamen und variablen Anforderungen der Produktlinie abgeleitet werden. Keine Abstraktion wird vergessen, keine zuviel eingebaut. Eine frühzeitige Degeneration der Framework-Architektur wird verhindert, da die Abstraktionen im voraus bekannt sind und nicht erst bei Bedarf eingebaut werden.

2.2 Entwicklung von Produktlinien

Produktlinien entstehen nicht über Nacht. Sie müssen entwickelt werden wie jedes andere Stück Software auch. Unter welchen Voraussetzungen kann die Entwicklung wie ablaufen?

2.2.1 Voraussetzungen

Der Umstieg von der Entwicklung einzelner Software-Systeme auf die produktlinienbasierte Entwicklung von Systemen birgt wirtschaftliche Risiken. Deshalb sollten den Umstieg nur Unternehmen wagen, in derem internen und externen Umfeld bestimmte wirtschaftliche und technische Voraussetzungen vorliegen.

Wirtschaftliche Voraussetzungen

Wenn in dem Geschäftsfeld, in dem das Unternehmen tätig ist, ein hoher Konkurrenzdruck herrscht, dann ist die Zeit bis zur Marktreife für neue Software-Systeme meist ein kritischer Erfolgsfaktor. Um wettbewerbsfähig zu bleiben beziehungsweise zu werden, empfiehlt sich die Einführung einer Produktlinie als Basis für neue Systeme.

Die hohen Vorabinvestitionen beim Aufbau einer Produktlinie führen erfahrungsgemäß dazu, daß die Produktlinie frühestens nach drei Jahren beziehungsweise zwei bis drei Produktzyklen den Kostendeckungspunkt erreicht. Die geplante Lebensdauer der Produktlinie sollte also länger als drei Jahre betragen, und aus der Produktlinie sollten mindestens drei bis fünf Systeme entwickelt werden [JGJ97, Seite 23] [WL99, Unterkapitel 4.1].

Aufgrund der Langfristigkeit von Produktlinien muß das Management eine Produktlinie als strategische Entscheidung zur Zukunftssicherung des Unternehmens betrachten und bereit sein, die notwendigen Ressourcen zur Verfügung zu stellen.

Technische Voraussetzungen

Eine Produktlinie muß einer stabilen, etablierten Domäne angehören. Stabil heißt, daß die Anforderungen an Systeme in der Domäne bekannt sind und sich während der Lebensdauer der Produktlinie nicht grundlegend ändern. Ist dies nicht der Fall, besteht die Gefahr, daß in der Wartungsphase häufiger tiefgreifende Änderungen an Produktlinien-Architektur und -Komponenten vorgenommen werden müssen. Die Folge: Die Wartungskosten der Produktlinie fressen die bei der Produkt-Entwicklung eingesparten Kosten auf. Produktlinien sind somit nicht für Domänen geeignet, deren Geschäftsprozesse und -objekte sich noch in der Entwicklung befinden. Beispielsweise wäre ein hohes wirtschaftliches Risiko eingegangen, wer bereits in der ersten Hälfte der 90er Jahre eine Produktlinie für e-Commerce-Systeme aufgebaut hätte.

2.2.2 Strategien für den Einstieg

Sind die Voraussetzungen für den Umstieg auf beziehungsweise Einstieg in die produktlinienbasierte Entwicklung von Systemen erfüllt, kann ein Unternehmen zwischen zwei grundsätzlich verschiedenen Einstiegsstrategien wählen: evolutionär oder revolutionär [Bos00, Unterkapitel 7.3].

Evolutionäre Strategie

Bei der evolutionären Strategie erfolgt der Umstieg auf Produktlinien System für System. Ausgangspunkt ist ein bereits existierendes, konventionell entwickeltes System, zu dem zunächst ein zweites ähnliches System und später ein drittes, viertes usw. neu hinzukommen soll. Der Umstieg läuft in folgenden Schritten ab [PR01, Abschnitt 3]:

1. Anforderungsbeschreibung des existierenden Systems mit den Anforderungen an das neue, zweite System abgleichen. Daraus ergibt sich die Gemeinsamkeit und Variabilität der Produktlinie.

2. Produktlinien-Architektur und Komponenten spezifizieren und implementieren. Basis hierfür sind Entwurfsdokumentation (sofern verfügbar) und Quellcode des existierenden Systems.

3. Existierendes System refaktorieren (unter Beibehaltung der Funktionalität umstrukturieren [Opd92, Fow99]), so daß es auf die Produktlinie aufsetzt

4. Neues System auf Basis der Produktlinie entwickeln

Für den weiteren Ausbau der Produktlinie um ein drittes, viertes usw. System sind die Schritte zu wiederholen.

Revolutionäre Strategie

Bei der revolutionären Strategie erfolgt der Einstieg in Produktlinien nicht in vielen kleinen Ausbaustufen, sondern als ein großer Schritt. Ausgangspunkt sind drei oder mehr geplante, noch nicht existierende Systeme, aus deren Anforderungen die Produktlinien-Architektur und -Komponenten abgeleitet werden.

Hilfreich bei dieser Strategie ist es, wenn das Unternehmen bereits früher Systeme entwickelt hat, die der gleichen Domäne angehören, in der auch die Produktlinie angesiedelt ist. Die dabei gewonnenen Erfahrungen sind bei der Spezifikation der Produktlinien-Architektur und -Komponenten nützlich. Im Unterschied zur evolutionären Strategie werden solche existierenden Systeme aus Kostengründen jedoch nicht refaktoriert und auf die Produktlinie umgestellt. Ziel ist vielmehr eine Neuentwicklung der Systeme auf einer gemeinsamen Basis. Die neuen Systeme lösen später die alten ab.

Vergleich beider Strategien

Vorteilhaft an der evolutionären Strategie ist der stufenweise Ausbau der Produktlinie. Er reduziert die Vorabinvestitionen und minimiert so die wirtschaftlichen Risiken von Produktlinien. Das zweite System kann schnell entwickelt und ausgeliefert werden. Return on Investment (Rückfluß des investierten Kapitals) setzt früh ein.

Nachteilig an der evolutionären Strategie ist, daß Refaktorierungen immer nur mit Blick auf das nächste in die Linie zu integrierende System geschehen. Anforderungen, die weiter in der Zukunft liegen, werden nicht mit einbezogen. Diese Kurzsichtigkeit, sprich Fokussierung auf morgen ohne Berücksichtigung von übermorgen, ähnelt der heutigen Softwareentwicklung und kann große Änderungen in Produktlinien-Architektur und -Komponenten zur Folge haben. Werden diese Änderungen nicht sorgfältig durchgeführt, degeneriert die Architektur der Produktlinie. Die Integration weiterer Systeme wird im Laufe des Lebenszyklus' der Produktlinie zunehmend schwieriger.

Natürlich müssen auch revolutionär gestartete Produktlinien ausgebaut werden. Aber da die anfängliche Architektur und Komponenten unter gleichzeitiger Einbeziehung von deutlich mehr Anforderungen als bei der evolutionären Strategie entwickelt worden sind, setzt die Degeneration erst später ein. Der Preis dafür sind höhere Vorabinvestitionen und ein zunächst langsamer Return on Investment.

2.2.3 Vorgehensmodell

Der Entwicklungsprozeß für ein einzelnes Software-System läuft in mehreren aufeinanderfolgenden Phasen ab. Jede Phase führt zu einem Ergebnis, das der nächsten Phase als Eingabe dient und von ihr weiterverarbeitet wird. Dieses Vorgehen illustriert Abbildung 2.2. Demnach sind die einzelnen Phasen: (1) Beschreibung von Anforderungen, (2) Analyse und Entwurf sowie (3) Implementierung und Auslieferung. Wie oft jede Phase bei der Entwicklung eines Systems

durchlaufen wird, variiert je nach Vorgehensmodell, z.B. Wasserfallmodell mit Rückkopplung [Roy70, DW99], inkrementelles Modell [Kru00, Bec99] oder Spiralmodell [Boe88]. Nach der Auslieferung geht das System in die Nutzungs- beziehungsweise Wartungsphase über, in der für jede Änderungsanforderung alle drei Phasen erneut durchlaufen werden und die Phasenergebnisse entsprechend der neuen Anforderung anzupassen sind.

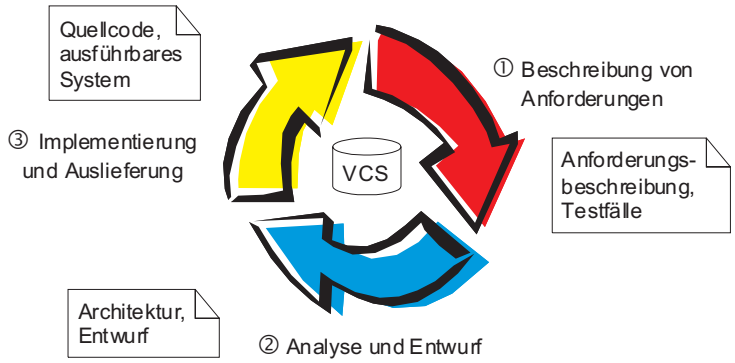

Abbildung 2.2: Vorgehensmodell zur Entwicklung einzelner Systeme

Das Vorgehensmodell für Produktlinien sieht etwas anders aus. Hier sind zunächst die Produktlinien-Architektur und die gemeinsamen Komponenten zu entwickeln. Zeitlich versetzt folgt die Entwicklung von Systemen auf Basis der Linie. Parallel dazu setzt die Wartung der Produktlinie ein, bei der Änderungsanforderungen, die während der Produkt-Entwicklung erkannt worden sind, in die Linie eingearbeitet werden. Im Produktlinien-Vorgehensmodell gibt es also zwei Typen von Entwicklungsprozessen, die jeweils ein anderes Ziel verfolgen:

- Das *Produktlinien-Engineering* entwickelt und wartet die Produktlinien-Architektur und -Komponenten.
- Das *Produkt-Engineering* entwickelt ausgehend von Kunden-Anforderungen konkrete Systeme auf Basis der Ergebnisse des Produktlinien-Engineerings.

Rückmeldungen des Produkt-Engineerings an das Produktlinien-Engineering führen zum Ausbau der Produktlinie. Abbildung 2.3 verdeutlicht die Zusammenhänge zwischen beiden Prozeßtypen.

Sowohl Produktlinien- als auch Produkt-Engineering-Prozesse gliedern sich in die gleichen Phasen wie beim Entwicklungsprozeß für einzelne Systeme. Allerdings paßt jeder Prozeßtyp die Phasen inhaltlich an, damit am Ende das gewünschte Ziel erreicht wird. Wie die Anpassungen konkret aussehen, beschreiben die folgenden zwei Abschnitte.

Produktlinien-Engineering

Ziel des Produktlinien-Engineerings ist die Festlegung der gemeinsamen Architektur für alle Systeme, die aus einer Produktlinie entwickelt werden sollen,

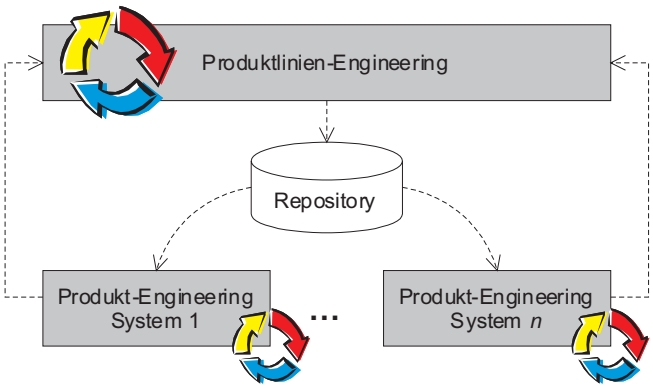

Abbildung 2.3: Vorgehensmodell zur Produktlinien- und Produkt-Entwicklung

und die Bereitstellung von Komponenten, die diese Architektur ausfüllen. Wie dies geschieht, daß beeinflußt vor allem die gewählte Einstiegsstrategie (vgl. Abschnitt 2.2.2). Pro Produktlinie gibt es genau einen Produktlinien-Engineering-Prozeß.

Phase 1: Beschreibung von Anforderungen. In dieser Phase sind die Anforderungen an die Produktlinie aufzunehmen und entweder der Gemeinsamkeit oder der Variabilität der Produktlinie zuzuordnen. Bei der evolutionären Einstiegsstrategie ergeben sich die Anforderungen aus dem Ausgangssystem und dem neu hinzukommenden System. Bei der revolutionären Strategie stammen die Anforderungen aus den geplanten Systemen und, wenn vorhanden, aus existierenden Systemen im eigenen Unternehmen oder bei Wettbewerbern.

Phase 2: Analyse und Entwurf. Hier wird zunächst die Architektur der Produktlinie aus den Anforderungen abgeleitet. Daraufhin werden die Komponenten unter Berücksichtigung der variablen Anforderungen entworfen und modelliert (vgl. Abschnitt 2.1.4). Bei der evolutionären Strategie fließen Architektur und Entwurf des existierenden Systems in die Produktlinie ein. Ist keine Entwurfsdokumentation zu dem System verfügbar, kann sie mit Mitteln des Reverse-Engineerings erstellt werden. In jedem Fall ist das existierende System am Ende dieser Phase so zu refaktorieren, daß es auf die Produktlinie aufsetzt.

Phase 3: Implementierung und Auslieferung. Zum Schluß folgen Implementierung, Test und Auslieferung der Architektur und Komponenten in das Repository.

Produkt-Engineering

Ziel des Produkt-Engineerings ist die Entwicklung von Systemen für Kunden auf Basis einer Produktlinie. Pro Produktlinie gibt es für jedes aus der Linie zu entwickelnde System einen Produkt-Engineering-Prozeß.

Phase 1: Beschreibung von Anforderungen. In dieser Phase sind die Anforderungen des Kunden aufzunehmen und mit den Anforderungen, die die Produktlinie bereits erfüllt, abzugleichen. Deckt die Produktlinie nicht alle Anforderungen des Kunden ab, gibt es mehrere Varianten, mit den *nicht erfüllten* Anforderungen umzugehen: (a) Anforderungen weiterleiten an das Produktlinien-Engineering zwecks Einarbeitung in die Produktlinie; (b) Anforderungen individuell nur für den Kunden realisieren; (c) Kunde mit dem Argument „schnellere und kostengünstigere Auslieferung des Systems“ zum Verzicht auf die nicht erfüllten Anforderungen bewegen.

Phase 2: Analyse und Entwurf. Hier wird zunächst die Produktlinien-Architektur instanziiert. Dabei werden variable Komponenten je nach Kunden-Anforderungen in das System aufgenommen oder weggelassen. Parametrisierbare Komponenten werden gemäß den Anforderungen eingestellt. Gegebenenfalls sind neue Komponenten für individuelle Anforderungen des Kunden zu entwerfen und in die instanziierte Architektur zu integrieren.

Phase 3: Implementierung und Auslieferung. Zum Schluß folgen Test und Auslieferung des Systems an den Kunden. Gegebenenfalls sind vorher individuelle Komponenten zu implementieren.

2.2.4 Methoden

Das Vorgehensmodell bestehend aus Produktlinien-Engineering und Produkt-Engineering legt einen allgemeinen Rahmen zur Entwicklung von Produktlinien fest. Gesucht ist jetzt noch eine methodische Unterstützung für die einzelnen Phasen beider Prozeßtypen, die Wege vorgibt, wie man Schritt für Schritt zu qualitativ guten Phasenergebnissen kommt.

Objektorientierte Methoden für Produktlinien?

Objektorientierte Methoden, Notationen und Programmiersprachen liegen im Trend und finden breite Akzeptanz in der Praxis. Beispiele sind die OOSE-Methode (Object-Oriented Software Engineering) [JCJÖ92], der Rational Unified Process [Kru00], die UML, C++, Java und Smalltalk. Könnten also objektorientierte Methoden das Produktlinien-Vorgehensmodell mit Leben, sprich der notwendigen methodischen Unterstützung füllen? Ja und nein. Die folgenden Ausführungen sagen warum, aufgeschlüsselt nach Prozeßtyp und Phase:

1. *Produktlinien-Engineering – Beschreibung von Anforderungen.* Objektorientierte Methoden sind auf die Entwicklung einzelner Systeme ausgerichtet. Sie erlauben die Beschreibung von Anforderungen für ein System zur Zeit. Für das Produktlinien-Engineering wäre die gleichzeitige Beschreibung von Anforderungen für alle Systeme einer Produktlinie sowie die Einordnung der Anforderungen als Gemeinsamkeit oder Variabilität der Produktlinie zu ergänzen.

2. *Produktlinien-Engineering – Analyse und Entwurf.* Die Zerlegung von Anforderungen in Klassen, Attribute und Operationen hat sich für viele Domänen als praxistauglich erwiesen. Diese Art der Zerlegung ist für Analyse und Entwurf von Produktlinien-Architektur und -Komponenten ebenfalls nützlich. Eine Produktlinien-Architektur kann als Framework implementiert werden (vgl. Abschnitt 2.1.4). Produktlinien-Komponenten können durch Einsatz von Entwurfsmustern wie Abstrakte Fabrik, Besucher, Strategie usw. [GHJV95] flexibel und parametrisierbar entworfen werden.

3. *Produktlinien-Engineering – Implementierung und Auslieferung.* Objektorientierte Programmiersprachen bieten durch Kapselung, Vererbung und Polymorphismus Möglichkeiten, Frameworks und Entwurfsmuster zu implementieren.

4. *Produkt-Engineering.* Das Produkt-Engineering ähnelt der Entwicklung einzelner Systeme, mit Ausnahme der Wiederverwendung von Architektur und Komponenten aus der Produktlinie. Da die Einzelsystem-Entwicklung das klassische Einsatzgebiet für objektorientierte Methoden ist, eignet sich die Objektorientierung für das Produkt-Engineering, wenn zeitgleich das Produktlinien-Engineering wiederverwendbare Frameworks und Komponenten liefert.

Fazit: Objektorientierte Methoden, Notationen und Programmiersprachen besitzen viele brauchbare Konzepte für die Prozesse und Phasen im Produktlinien-Vorgehensmodell. Was ihnen jedoch fehlt, ist die adäquate Betrachtung von Gemeinsamkeit und Variabilität im Produktlinien-Engineering.

Domänen-Engineering-Methoden für Produktlinien?

Wenn es darum geht, Gemeinsamkeit und Variabilität von Systemen in einer Domäne zu beschreiben, bieten sich Domänen-Engineering-Methoden an. Sie liefern Verfahren, um Domänen vollständig zu analysieren und Beschreibungsmittel, um die Ergebnisse, das Domänenwissen, zu dokumentieren und letztendlich zu implementieren. Das Domänen-Engineering berücksichtigt sowohl existierende als auch geplante zukünftige Systeme. Nach [CE00, Unterkapitel 2.8] sind die bekanntesten Methoden des Domänen-Engineerings (vgl. auch Abbildung 2.4 auf Seite 24):

- Draco, als Vorreiter aller Domänen-Engineering-Methoden [Nei80]
- Feature-Oriented Domain Analysis (FODA) [K+90]
- Organization Domain Modeling (ODM) [S+96]

Warum also nicht Domänen-Engineering-Methoden für die Entwicklung von Produktlinien nutzen? Vier Gründe, die dagegen sprechen:

1. *Aufwand für Domänenanalyse.* Der Aufwand für eine vollständige Analyse und Beschreibung der Domäne und für die Implementierung des gesamten

Domänenwissens ist sehr hoch und wäre im Rahmen des Produktlinien-Engineerings wirtschaftlich kaum zu rechtfertigen. Schließlich sollen nur eine Reihe ganz bestimmter Systeme aus der Produktlinie entwickelt werden und nicht beliebige Systeme. Neuere Methoden des Domänen-Engineerings wie ODM umgehen dieses Problem, indem sie nach der Analyse ein sogenanntes *scoping* durchführen. Dabei wird das Domänenwissen auf die tatsächlich zu entwickelnden Systeme eingeschränkt und so der Aufwand für Entwurf und Implementierung des Domänenwissens reduziert.

2. *Methode und Notation für Domänenentwurf.* Alle Domänen-Engineering-Methoden beschreiben einen Prozeß und bieten methodische Unterstützung, um die Domäne zu analysieren. Nur wenige gehen darüber hinaus und unterstützen, z.B. mittels einer Notation, die Modellierung der Domäne in der Entwurfsphase. Und diese wenigen Methoden setzen auf veraltete Modellierungstechniken wie strukturierten Entwurf.

3. *Implementierung der Domäne.* Obwohl die Implementierung des Domänenwissens Bestandteil des Domänen-Engineerings ist, trifft kaum eine Methode hierzu eine konkrete Aussage. Bei den wenigen Ausnahmen beherrschen strukturierte Programmiersprachen das Bild.

4. *Standardisierung und Akzeptanz.* Domänen-Engineering-Methoden sind proprietär, wenig bekannt, nicht standardisiert und meist nur in kleineren Fallstudien erprobt. Ihre industrielle Akzeptanz ist deshalb gering.

Fazit: Die Stärke von Domänen-Engineering-Methoden liegt in der Analyse und Beschreibung der Gemeinsamkeit und Variabilität von Systemen in einer Domäne. Damit gleichen sie die Schwäche objektorientierter Methoden aus. Zusammengenommen ergeben beide methodischen Ansätze eine gute Ausgangsbasis für Methoden zur Entwicklung von Produktlinien im Rahmen des eingeführten Vorgehensmodells.

Produktlinien-Methoden

Abbildung 2.4 gibt einen Überblick zu den heute verfügbaren Produktlinien-Methoden (grau hinterlegt) und ihren Vorgängern beziehungsweise Ideenlieferanten. Die aktuellsten Methoden (fett markiert) sind demnach:

FeatuRSEB (sprich: *featured RSEB*) ist der Nachfolger des Reuse-Driven Software Engineering Business (RSEB). RSEB ist eine aus der OOSE-Methode hervorgegangene Vorgehensweise zur objektorientierten, UML-basierten Entwicklung von Produktlinien. FeatuRSEB integriert Konzepte aus FODA in RSEB und beseitigt damit Schwächen, die bei der industriellen Erprobung von RSEB in der Telekommunikationsdomäne auftraten [K+90, JCJÖ92, JGJ97, GFA98].

KobrA basiert auf PuLSE, der Product-Line Software Engineering Methodology. PuLSE beschreibt einen Entwicklungsprozeß für Produktlinien auf abstraktem Niveau. Phasen, deren Abläufe und Ergebnisse werden nur

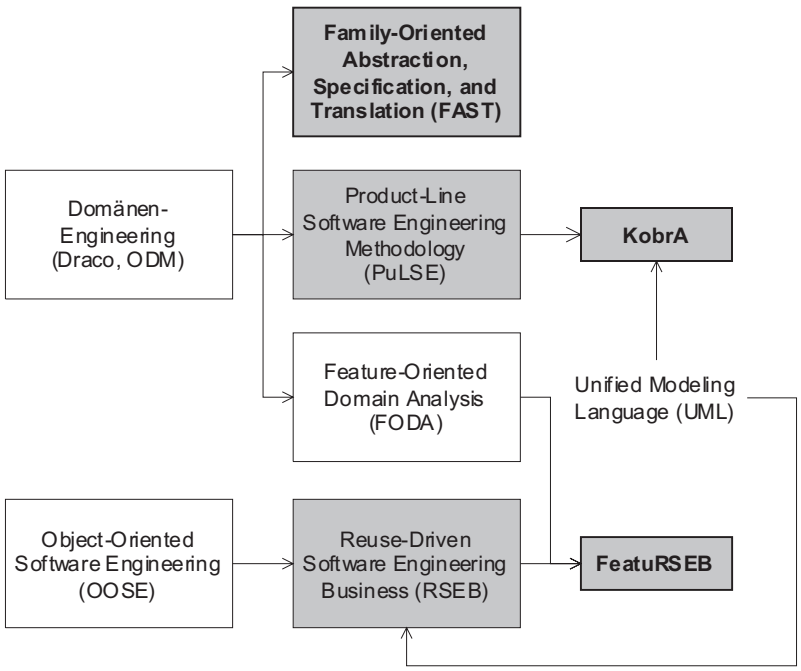

Abbildung 2.4: Historische Entwicklung von Produktlinien-Methoden

skizziert, ohne eine konkrete Modellierungstechnik und Notation vorzugeben. KobrA konkretisiert die Phasen von PuLSE, so daß Produktlinien objektorientiert mit der UML entwickelt werden können. Sowohl PuLSE als auch KobrA sind Produkte des Fraunhofer-Instituts für Experimentelles Software-Engineering und als solche im Rahmen von Beratungsleistung käuflich zu erwerben [B+99, ABM00].

FAST (Family-Oriented Abstraction, Specification, and Translation) legt ähnlich wie PuLSE Abläufe und Ergebnisse von Prozessen zur Entwicklung von Produktlinien fest. Abbildung und Anpassung dieser Prozesse auf eine konkrete, z.B. objektorientierte Methode, Notation und Programmiersprache überläßt FAST jedoch dem Anwender der Methode; ein KobrA für FAST ist nicht verfügbar. AT&T entwickelte FAST und setzt es mit Erfolg ein [WL99].

Im Rahmen der Arbeit werden weder FAST noch KobrA weiterverfolgt. FAST, weil dort nur eine Prozeßbeschreibung erfolgt; KobrA, weil die während des Zeitraums der Entstehung dieser Arbeit *öffentlich* verfügbare Dokumentation zu KobrA sich auf wenige Artikel beschränkte, aufgrund derer die Eignung der Methode für die Produktlinien-Entwicklung nicht stichhaltig bewertet werden konnte. Durch das Erscheinen von [A+01] hat sich diese Situation mittlerweile geändert.

FeatuRSEB hingegen ist ausführlich in den oben zitierten Referenzen dokumentiert. Eine detaillierte Betrachtung und Bewertung von FeatuRSEB folgt in

Kapitel 4. Daran anschließend erweitert diese Arbeit FeatuRSEB, um die im folgenden Unterkapitel angesprochenen Nachteile der Methode auszugleichen.

2.3 Serienfertigung mit Produktlinien

Die Entwicklung von Software-Systemen im Produkt-Engineering auf Basis einer Produktlinie hat Vorteile gegenüber der Entwicklung von Einzelsystemen (vgl. in Abschnitt 2.1.3 auf Seite 14), bringt aber immer noch Nachteile mit sich:

- *Zeit bis zur Marktreife.* Durch die Wiederverwendung ist die Zeit bis zur Marktreife produktlinienbasierter Systeme zwar kürzer als die von Einzelsystemen. Trotzdem braucht es Zeit und kostet Geld, bis die Anforderungen eines Kunden aufgenommen und mit der Funktionalität der Produktlinie abgeglichen sind und bis das passende System manuell aus den Produktlinien-Komponenten zusammengesetzt ist.

- *Fette Systeme.* Die Komponenten einer Produktlinie sind von grober Granularität, weil das manuelle Zusammensetzen von Produktlinien-Systemen aus feingranularen Komponenten zu zeitaufwendig und zu fehlerträchtig wäre. Die grobe Granularität der Komponenten führt dazu, daß die Systeme Funktionalität enthalten, die der Kunde nicht gefordert hat, die sich aber wegen der groben Granularität nicht aus den Systemen entfernen läßt. Der Bedarf an Ressourcen ist bei produktlinienbasierten Systemen somit größer als bei Einzelsystemen, die Effizienz entsprechend schlechter. Je nach Art des Umfelds von Produktlinien-Systemen kann diese mangelnde „Schlankheit“ nicht vernachlässigt werden – beispielsweise bei eingebetteten Systemen (auf Chipkarten, in Mobiltelefonen oder Autos) oder bei Systemen, die über Netzwerke bezogen werden (Applets).

Die Nachteile kann ausräumen beziehungsweise abschwächen, wer die manuellen Tätigkeiten im Produkt-Engineering automatisiert beziehungsweise teilautomatisiert.

2.3.1 Produktion von Autos

Wie kann dieses Ziel erreicht werden? Dazu lohnt ein Vergleich der Software-Branche mit der Automobilindustrie: Die Entwicklung einer Software-Produktlinie ist vergleichbar mit der Entwicklung eines Auto-Modells. Aber, Autos werden nicht auf Basis solcher Modelle entwickelt (wie Software-Systeme im Produkt-Engineering), sondern produziert beziehungsweise in Serie gefertigt:

> In der *Serienfertigung* „werden mehrere Produkte, die sich aus vielen Einzelteilen zusammensetzen und die auf Grund ihrer unterschiedlichen Konstruktion einen unterschiedlichen Fertigungsgang haben, in begrenzter Menge hergestellt“ [Wöh93, Seite 508].

Jedes Auto-Modell ist unterschiedlich konstruiert. Deshalb benötigt ein Automobilhersteller für jedes Modell eine eigene Fertigungsstraße, in der Autos nach einem modellspezifischen, fest vorgegebenen Ablauf gefertigt, d.h. aus Einzelteilen zusammengesetzt werden. Allerdings sind die Autos, die aus einer Fertigungsstraße kommen, selten genau gleich ausgestattet. Wäre dies der Fall, liegt Massenfertigung und nicht Serienfertigung vor. Statt dessen legt der Kunde bei der Bestellung eines Autos dessen Ausstattung – innerhalb gewisser, vom Hersteller vorgegebener Grenzen – fest. Der Kunde *konfiguriert* sozusagen das Auto nach seinen Wünschen.

Ein anschauliches Beispiel zur Festlegung der Konfiguration eines Autos liefert der Volkswagen-Konzern in seinem Internet-Auftritt mit dem Volkswagen-Konfigurator [VW]. Dort läuft der Konfigurationsvorgang für das Wunschauto in folgenden Schritten ab:

1. Modell und Motorisierung wählen
2. Außen- und Innenfarbe wählen
3. Sonderausstattung ankreuzen
4. Finanzierungsangebot kalkulieren
5. Konfiguration drucken oder an Händler senden

Abbildung 2.5 zeigt Teile des Vorgangs anhand von Ausrissen aus Bildschirmfotos. Auffällig sind die zwei Arten, wie der Kunde bestimmte Entscheidungen treffen kann: aus einer Menge von Alternativen genau eine auswählen (Modell und Motorisierung) sowie aus einer Menge von optionalen Eigenschaften beliebig viele oder gar keine ankreuzen (Sonderausstattung). Interessant ist desweiteren, daß der Konfigurator nach jedem Schritt prüft, ob basierend auf der aktuellen Konfiguration tatsächlich ein Auto produziert werden kann. Ist dies nicht der Fall, gibt der Konfigurator Hinweise, welche Korrekturen der Kunde vornehmen muß. Die eigentliche Bestellung des Wunschautos gibt der Kunde dann beim Händler in Auftrag, woraufhin Volkswagen das Auto fertigt und liefert.

2.3.2 Produktion von Software-Systemen

Der Serienfertigungsgedanke läßt sich auf Software-Systeme übertragen. Software wird dann nicht mehr entwickelt, sondern von einem Generator (Fertigungsstraße) ausgehend von Kunden-Anforderungen (Ausstattung, Konfiguration) aus Komponenten (Einzelteilen) automatisiert zusammengesetzt. Dazu müssen Architektur (Konstruktion) und Komponenten der zu fertigenden Systeme im voraus bekannt sein – was eine Software-Produktlinie zur idealen Basis für die Serienfertigung macht. Abbildung 2.6 visualisiert den Fertigungsablauf einer Software anhand eines UML-Aktivitätsdiagramms.

Zwischen welchen Ausstattungsmöglichkeiten Kunden wählen können, ergibt sich aus der Variabilität der Produktlinie. Die Konfiguration enthält folglich eine Teilmenge der variablen Anforderungen der Produktlinie. Konsequenterweise gilt: Je breiter die Variabilität einer Produktlinie ist, desto vielfältiger sind

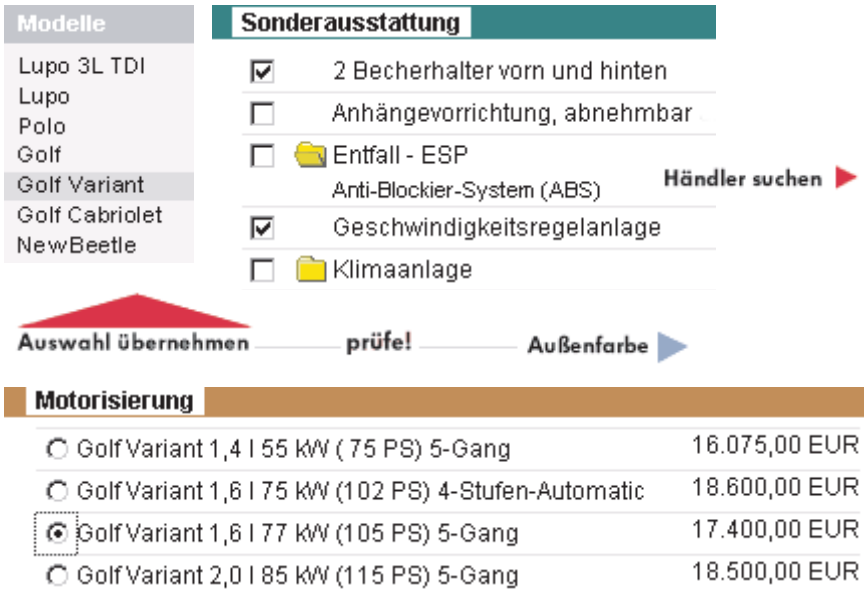

Abbildung 2.5: Volkswagen-Konfigurator

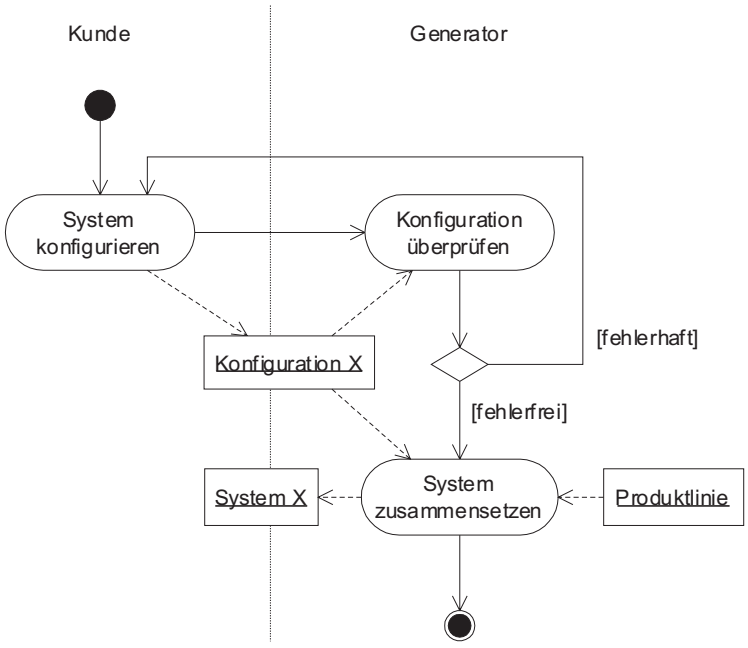

Abbildung 2.6: Serienfertigung von Software-Systemen

die aus ihr generierbaren Systeme, was wiederum den Kreis potentieller Kunden vergrößert. Aber: Mehr Variabilität bedeutet auch mehr Komplexität bei Produktlinien-Architektur und -Komponenten und damit höhere Entwicklungs- und Wartungskosten. Gesucht ist ein wirtschaftlich begründeter Mittelweg.

Zusammenfassend die Vorteile der produktlinienbasierten Serienfertigung von Software-Systemen: Das Produkt-Engineering fällt weg beziehungsweise kann stark verkürzt werden. Das reduziert erneut die Zeit bis zur Marktreife für Software-Systeme. Durch das automatisierte Zusammensetzen der Systeme sind feingranulare Komponenten in der Produktlinie möglich, wodurch sich schlanke Systeme fertigen lassen. Damit eignen sich Produktlinien auch als Basis z.B. für eingebettete Systeme.

Die Nachteile eines automatisierten Produkt-Engineerings: Kunden-Anforderungen, die die Produktlinie nicht abdeckt, können bei der Serienfertigung, anders als noch beim manuellen Produkt-Engineering, nicht mehr berücksichtigt werden. Der Grad an Individualität bei den generierten Systemen ist durch die Variabilität der Produktlinie festgelegt. Schließlich funktioniert die Serienfertigung bei Autos auch nur, weil die Kunden immer Autos mit vier Rädern bestellen und nicht heute welche mit zwei Rädern und morgen welche mit sechs. Auf der anderen Seite sind individuelle Anforderungen bei Software-Systemen nie auszuschließen. Eine Lösung ist die teilautomatisierte Fertigung eines Systems: Der Generator setzt das System soweit wie möglich zusammen; fehlende Teile werden dann manuell ergänzt. Wird die gleiche Anforderung mehrfach nachgefragt, ist ein Ausbau der Produktlinie zu überlegen.

Ein weiterer Nachteil erwächst aus den feingranularen Komponenten. Sie machen das Produktlinien-Engineering zeitaufwendiger und teurer. Es besteht die Gefahr, daß Produktlinien-Architektur und -Komponenten sehr komplex werden und nur noch schwer zu warten sind.

2.3.3 Voraussetzungen

Welche technischen Voraussetzungen sind notwendig, damit Serienfertigung von Software-Systemen auf Basis einer Produktlinie implementiert werden kann?

Der Fertigungsablauf muß als Algorithmus vorliegen und in einem Generator implementiert sein.

Die Konfiguration des zu fertigenden Systems muß maschinenverarbeitbar vorliegen und vor der Fertigung vom Generator auf ihre Korrektheit hin überprüfbar sein (vgl. die Bemerkungen zum Volkswagen-Konfigurator in Abschnitt 2.3.1 auf Seite 26). Das wiederum setzt voraus, daß die Konfigurationsmöglichkeiten in der Produktlinie in maschinenauswertbarer Form hinterlegt sind.

Beim Abarbeiten der Konfiguration – also der dort aufgeführten variablen Anforderungen – muß der Generator wissen, (a) welche Komponente eine Anforderung implementiert, damit der Generator die Komponente in das System aufnehmen kann; (b) bei welcher Komponente er welche Parameter aufgrund einer Anforderung wie einstellen muß. Anders ausgedrückt: Das Produktlinien-Engineering muß dokumentieren, welche Komponenten oder Komponenten-Parameter aus welcher variablen Anforderung hervorgehen (Verfolgbarkeit, engl.: *traceability*).

Die Komponenten müssen feingranular modularisiert sein, damit tatsächlich schlanke Systeme gefertigt werden können. Eine Komponente darf also immer nur genau eine variable Anforderung implementieren.

2.3.4 Implementierungsansätze

Nachdem klar ist, was eine Serienfertigung technisch voraussetzt, stellt sich die Frage, welche existierenden Ansätze diese Voraussetzungen erfüllen und welche nicht. Brauchbare Ansätze müssen beantworten, wie der Generator zu bauen ist; wie Konfigurationsmöglichkeiten in einer Produktlinie zu hinterlegen sind; wie die Konfiguration eines Systems anzugeben ist; wie Anforderungen und Komponenten zusammenhängen; wie Komponenten feingranular zu modularisieren sind.

FeatuRSEB für Serienfertigung?

Die in Abschnitt 2.2.4 ausgewählte Produktlinien-Methode FeatuRSEB setzt bei Modellierung und Implementierung von Produktlinien auf das objektorientierte Paradigma. Aber eignet sich FeatuRSEB, speziell Objektorientierung und die UML auch für die Entwicklung solcher Produktlinien, auf deren Basis einmal Software-Systeme in Serie gefertigt werden sollen? Ja und nein. Die folgenden Ausführungen begründen diese Antwort:

1. *Generator.* Der Bau eines Generators ist mit objektorientierten Techniken möglich. Er wird weder besonders gefördert noch unterbunden (vgl. Abschnitt 4.5.1).

2. *Konfiguration.* FeatuRSEB hinterlegt die Konfigurationsmöglichkeiten einer Produktlinie in einem Merkmalmodell. Aufbauend auf dem Modell wird die Konfiguration eines Systems angegeben. Ihre Korrektheit kann der Generator überprüfen, weil das Merkmalmodell die hierfür notwendigen Informationen in maschinenauswertbarer Form enthält (vgl. Abschnitte 4.2.2 und 4.5.1).

3. *Verfolgbarkeit.* Die Verfolgbarkeit von Anforderungen zu Komponenten stellt FeatuRSEB durch *trace*-Abhängigkeitsbeziehungen im UML-Modell der Produktlinie sicher (vgl. Abschnitte 4.2.3, 4.2.4 und 4.5.1).

4. *Granularität der Komponenten.* Die Modellierung und Implementierung feingranularer Komponenten ist mit objektorientierten Techniken möglich, aber nur durch vermehrten Einsatz von Vererbung und Entwurfsmustern – und den damit verbundenen negativen Konsequenzen in Bezug auf Komplexität, Verständlichkeit und Wartbarkeit von Modell und Quellcode der Produktlinie (vgl. Abschnitte 4.2.4, 4.5.2 und 4.5.3).

Fazit: Das Problem in FeatuRSEB sind die unzureichenden Möglichkeiten zur Modellierung und Implementierung feingranularer Komponenten. Eine Lösung hierfür bietet die generative Programmierung an.

Generative Programmierung

Die generative Programmierung beschäftigt sich mit der Generierung von Software-Systemen auf Basis von Produktlinien [CE00, GP]. Um dieses Ziel zu erreichen, definiert die generative Programmierung eine Reihe von Prinzipien. Das wichtigste ist die „Trennung von Belangen“ (engl.: *separation of concerns*). Dijkstra prägte diesen Ausdruck und wollte damit betonen, daß man bei der Entwicklung von Systemen immer nur einen Belang auf einmal betrachten und bearbeiten sollte, damit man sich nicht im Geflecht der Belange verliert [Dij76, Seite 211]. Die generative Programmierung sieht die Variabilität in Produktlinien-Komponenten als Belange an und möchte sie getrennt voneinander modellieren und implementieren. Dadurch können Komponenten von grober Granularität sehr fein zerlegt werden. Bei der Generierung eines Systems werden dann nur diejenigen Komponenten in das System aufgenommen, deren Funktionalität das System tatsächlich nutzt. Die generierten Systeme sind schlank, belegen keine unnötigen Ressourcen und sind entsprechend effizienter. Die Umsetzung des Prinzips der Trennung von Belangen ist mit einer Reihe von Techniken möglich, die in den letzten Jahren entwickelt worden sind:

Aspektorientierte Programmierung verwirklicht die Trennung von Belangen, indem sie aus den Klassen eines Systems die Belange herauszieht und getrennt in sogenannten Aspekt-Modulen implementiert. Ein Aspekt-Modul (kurz Aspekt) nimmt in seiner Implementierung immer Bezug auf ein oder mehrere Klassen und beschreibt, wie die dort definierte Struktur und das Verhalten um belangspezifisches zu ergänzen ist. Die Ergänzungen wirken sich auf Instanzen der Klassen jedoch nur dann aus, wenn der Aspekt beim Übersetzen des Systems an die Klassen gebunden wird. Bei der Zerlegung von Systemen in Klassen und Aspekte ist die Orthogonalität von Aspekten zu beachten: Ein Aspekt kann nur Klassen, nie andere Aspekte ergänzen. AspectJ ist ein Werkzeug, das aspektorientiertes Programmieren mit Java ermöglicht [K+97, K+01, AJ].

Der **Hyperspace-Ansatz** dehnt die in der aspektorientierten Programmierung vorgenommene Zerlegung von Systemen auf zwei Arten (Klassen und Aspekte) auf beliebig viele Arten aus. Eine Zerlegungsart wird auch als Dimension bezeichnet, so daß mit dem Hyperspace-Ansatz eine mehrdimensionale Trennung von Belangen möglich wird. Jeder Belang wird getrennt in einem Modul, sogenannte Hyperebene, modelliert und implementiert. Im Gegensatz zu Aspekten sind Hyperebenen nicht zwingend orthogonal zueinander. Sie können beliebig zu Systemen zusammengesetzt werden. Hyper/J bildet den Hyperspace-Ansatz auf Java ab [OT01, TO01, HS].

GenVoca faßt Belange (GenVoca-Komponenten genannt) in *realms* (dt.: Königreich) zusammen. *Realms* wiederum sind in eine Schichtenarchitektur eingeordnet, wobei die oberen Schichten die Komponenten aus den unteren Schichten um weitere Funktionalität ergänzen. Die Schichtenarchitektur läßt sich mit einer GenVoca-Grammatik formal beschreiben. Ein Generator, der diese Grammatik verarbeitet, setzt *realms* und Komponenten zu Systemen zusammen. Der Bau eines solchen Generators und die Implementierung von GenVoca-Komponenten in Java ist mit der Jakarta Tool Suite möglich [BG97, JTS].

Nicht alle diese Techniken sind gleichermaßen gut geeignet, um das Problem von FeatuRSEB zu lösen. Eine geeignete Technik muß (a) die Modellierung und Implementierung feingranularer Produktlinien-Komponenten erlauben, ohne daß die entwickelten Produktlinien zu komplex und unwartbar werden; (b) ein Werkzeug bereitstellen, das die Komponenten wieder zu Systemen zusammensetzt und damit den letzten Schritt in der Serienfertigung vollzieht.

Die aspektorientierte Programmierung ist vornehmlich auf die Implementierung von Aspekten ausgerichtet. Zwar existiert ein Vorschlag, wie die strukturelle Sicht auf Aspekte in UML-Klassendiagrammen modelliert werden kann [SY99], eine Modellierung des Aspekt-Verhaltens wird aber nicht unterstützt. Problematisch ist auch die Orthogonalität von Aspekten: Sie führt zu Aspekt-Modulen, die nicht immer leicht verständlich und wartungsfreundlich sind [Mac01, Unterkapitel 8.2]. Desweiteren ist noch nicht geklärt, ob und wie eine Beschreibung der Anforderungen von Systemen getrennt nach Aspekten geschehen kann. Positiv fällt hingegen das bei AspectJ mitgelieferte Werkzeug auf. Es hat die Beta-Phase hinter sich, läuft stabil und ist gut dokumentiert.

GenVoca ist ähnlich implementierungsnah gehalten wie die aspektorientierte Programmierung. Die Modellierung von *realms* und GenVoca-Komponenten unterstützt GenVoca bislang nicht, abgesehen von der formalen Beschreibung der Schichtenarchitektur. Auf der Werkzeugseite enttäuscht die Jakarta Tool Suite. Sie ist sehr spärlich dokumentiert, so daß weder die Implementierung von *realms* und GenVoca-Komponenten noch der Bau eines Generators gelang [Kle02].

Diese Arbeit verfolgt deshalb nur den Hyperspace-Ansatz weiter. Durch das von ihm definierte allgemeingültige Modell, wie die Belange eines Systems getrennt werden können, unterstützt der Ansatz als einzige der drei vorgestellten Techniken alle Phasen der Softwareentwicklung: Beschreibung von Anforderungen, Analyse und Entwurf sowie Implementierung. Zudem können die Belange beliebig zueinander in Beziehung stehen; Orthogonalität ist nicht gefordert. Das Werkzeug von Hyper/J befindet sich noch im Beta-Stadium, arbeitet aber hinreichend stabil (vgl. Kapitel 5).

2.4 Zusammenfassung

Ziel dieser Arbeit ist es, Software-Produktlinien objektorientiert so zu entwickeln, daß auf ihrer Basis Software-Systeme in Serie gefertigt werden können. Grundlage für die Entwicklung solcher Produktlinien ist die objektorientierte, UML-basierte Methode FeatuRSEB. Grundlage für die Implementierung der Serienfertigung ist der Hyperspace-Ansatz. Bevor diese Grundlagen in Kapitel 4 und 5 detaillierter vorgestellt und bewertet werden, folgt im nächsten Kapitel die Beschreibung einer Fallstudie, die im Rahmen dieser Arbeit zur Überprüfung der Zielsetzung und zur Illustration der erarbeiteten Konzepte durchgeführt worden ist.

Kapitel 3

Fallstudie: Die Siedler von Catan

Software-Engineering ist Wissenschaft und Praxis zugleich. Es ist deshalb wenig zweckmäßig, Entwicklungsmethoden allein auf einer rein theoretischen Basis aufzustellen und zu bewerten. Nur ein praktischer Einsatz im Rahmen empirischer Fallstudien bringt ans Licht, welcher Nutzen aus einer Methode für die Software-Engineering-Gemeinde erwächst.

Diese Arbeit bewertet im nächsten Kapitel 4 die Produktlinienmethode FeatuRSEB (vgl. Abschnitt 2.2.4) und stellt anschließend eine Weiterentwicklung der Methode vor. Um die bei FeatuRSEB gefundenen Probleme und die zu ihrer Lösung erarbeiteten Konzepte in der Praxis zu überprüfen, wurde im Umfeld der Arbeit eine Fallstudie durchgeführt. Die Studie bestand aus der Entwicklung einer Produktlinie für das Gesellschaftsspiel „Die Siedler von Catan“. Im folgenden wird die Auswahl des Spiels begründet und das Spiel selbst näher vorgestellt.

3.1 Vorüberlegungen

Warum wurde gerade diese und nicht eine andere Produktlinie entwickelt? Das erste Entscheidungskriterium war, daß die Fallstudie keine fiktive, extra für die Arbeit entworfene Produktlinie zum Inhalt haben sollte. Damit wurde der Gefahr begegnet, die Beispiel-Produktlinie speziell auf die gewünschten Ergebnisse der Arbeit hin abzustimmen, was wiederum die Verallgemeinerbarkeit der Ergebnisse, d.h. ihre Übertragbarkeit auf andere Produktlinien, in Frage gestellt hätte. Das Spiel „Die Siedler von Catan“ bildet schon von Haus aus eine „natürliche“ Produktlinie. Sie besteht aus vier Produkten, deren Spielregeln einen gemeinsamen Kern bilden und die darüber hinaus einen hohen Grad an Variabilität aufweisen.

Das zweite Entscheidungskriterium war die Größe der Produktlinie. Sie mußte einerseits wenigstens so umfangreich sein, daß an ihr die in der Zielsetzung

getroffenen Aussagen hinreichend gut zu überprüfen waren. Andererseits durfte die Produktlinie höchstens so groß sein, daß ihre Entwicklung nicht das zur Verfügung stehende Zeitkontingent überschritt. Letzteres betrug etwa 8 Personen-Monate und ergab sich aus der an der Fallstudie beteiligten Zahl an Personen. Die Problemstellung der „Siedler von Catan“ ist überschaubar und dennoch nicht trivial. Aufgrund der Natur eines Gesellschaftsspiels liegt die Lösung in einem interaktiven System, das intern einen hohen Anteil Verhalten aufweist, der sich nicht wie bei anderen, eher datenlastigen Systemen auf bloßes Einfügen, Ändern und Löschen beschränkt. Das führte während der Entwicklung zu aufschlußreichen Beobachtungen hinsichtlich der bei FeatuRSEB zu verbessernden Punkte.

Drittens mußte die Produktlinie aus einer Domäne stammen, für die FeatuRSEB die Entwicklung von Produktlinien unterstützt. Das trifft auf Domänen zu, deren Wissen sich mit Use-Cases beschreiben und in Objekte zerlegen läßt. Solch eine Domäne ist die der Gesellschaftsspiele.

Schließlich sollten die Ergebnisse der Fallstudie veröffentlicht werden. Industrielle Produktlinien unterliegen in der Regel Vereinbarungen zur Geheimhaltung und dürfen somit nicht oder nur eingeschränkt der Öffentlichkeit zugänglich gemacht werden. Dagegen lief die Entwicklung der Produktlinie „Die Siedler von Catan“ vollständig hochschulintern ab, so daß eine Darstellung von Modellen und Quellcodes keine geschäftlichen Interessen eines Unternehmens verletzt.

3.2 Ausbaustufen

Die Produktlinie „Die Siedler von Catan“ besteht, wie oben bereits erwähnt, aus vier Produkten [Teu]:

- Basisspiel
- 5 und 6 Spieler
- Die Seefahrer
- Städte & Ritter

Das Basisspiel legt die grundlegenden Spielregeln fest, die anderen drei Produkte, auch Ergänzungssets genannt, fügen weitere Regeln hinzu oder ändern vorhandene ab. Somit definiert das Basisspiel vorwiegend die Gemeinsamkeit oder Kernfunktionalität der Produktlinie, die Ergänzungssets ihre Variabilität.

Die Produkte können beliebig miteinander kombiniert werden, vorausgesetzt das Basisspiel ist mit dabei. Daraus ergeben sich insgesamt 8 mögliche Kombinationen. Ein Ziel bei der Entwicklung der Software-Produktlinie war es, die Spielregeln der Produkte so in Komponenten zu modularisieren, daß zusätzliche Kombinationen, also Mischformen der verkauften Produkte, gefertigt werden können. Das Ziel wurde erreicht: Über 38 Millionen Varianten der „Siedler von Catan“ fertigt ein Generator aus den Komponenten der Produktlinie.

Die Entwicklung der Produktlinie lief evolutionär ab (vgl. Abschnitt 2.2.2). Die Ausbaustufen entsprachen denen der Produkte, wobei das Basisspiel den Anfang machte. Jede Ausbaustufe unterteilte sich wiederum in einzelne Mikro-Stufen. Bei den mehr als 30 Ausbauten wurden vor allem Daten gesammelt, um das Ziel Wartbarkeit überprüfen zu können.

Die folgenden Abschnitte geben einen kurzen Abriß über die Spielregeln der einzelnen Produkte. Bei den Ergänzungssets werden die gegenüber dem Basisspiel hinzugefügten und geänderten Regeln gesondert ausgewiesen. Eine detaillierte Analyse der Spielregeln, insbesondere hinsichtlich ihrer Zuordnung zur Gemeinsamkeit und Variabilität der Produktlinie, wird hier nicht vorgenommen. Wer daran interessiert ist, sei auf [Hal01, Kapitel 4 und 5] und [Ulr02, Kapitel 3] verwiesen.

3.2.1 Basisspiel

Ziel des Spiels ist die Besiedlung der Insel Catan durch 3 oder 4 Spieler. Wer als erster 10 Siegpunkte erreicht, gewinnt das Spiel. Die Insel wird durch einen Spielplan repräsentiert, der aus einzelnen Spielfeldern zusammengesetzt ist. Abbildung 3.1 zeigt einen Ausschnitt des Spielplans in einem laufenden Spiel. Dort sind eine Reihe Sechsecke, die Spielfelder, zu sehen. Es gibt verschiedene Arten von Feldern: Landfelder (Ackerland, Gebirge, Hügelland, Wald, Weideland oder Wüste) und Meerfelder (mit oder ohne Hafen).

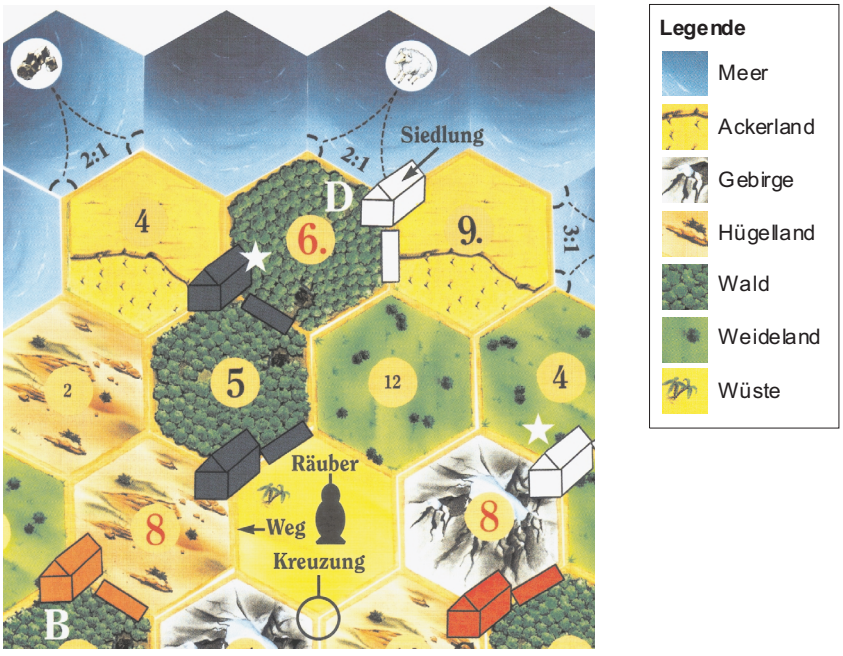

Abbildung 3.1: Spielplan „Die Siedler von Catan“

Durch das Aneinanderlegen der Spielfelder ergeben sich Kreuzungen und Wege. Eine Kreuzung ist eine Ecke eines Spielfelds, ein Weg eine Kante. Diese gilt es im Spiel zu besiedeln. Dafür besitzt jeder Spieler ein Kontingent an Spielfiguren: Straßen, Siedlungen und Städte. Die Besiedlung beginnt in der Gründungsphase eines Spiels und setzt sich in der Besiedlungsphase fort.

Gründungsphase

Diese Phase besteht aus zwei Runden. In der Hinrunde gründen die Spieler reihum zunächst eine Siedlung auf einer beliebigen Kreuzung und bauen dann eine Straße auf einem der an die Siedlung angrenzenden Wege. Die Rückrunde läuft ebenso ab, nur in umgekehrter Reihenfolge. Zusätzlich erhält jeder Spieler für seine in der Rückrunde gegründete Siedlung ein Startkapital an Rohstoffen. Rohstoffe sind die Währung Catans, ohne die die Spieler in der Besiedlungsphase keine Aktionen ausführen können.

Besiedlungsphase

In der Besiedlungsphase kommen die Spieler reihum zum Zug. Als erstes muß der Spieler am Zug die Rohstofferträge auswürfeln. Dazu würfelt er mit zwei Würfeln. Die Summe der Augenzahlen bestimmt die Rohstoffe, die die Spieler in diesem Zug erhalten. Auf jedem Landfeld, mit Ausnahme der Wüste, liegt ein Zahlenchip, z.B. die 8 auf dem Gebirgsfeld in der unteren Zeile von Abbildung 3.1. Angenommen, der Spieler am Zug hat eine 8 gewürfelt. Dann bekommen alle Spieler, die auf den Kreuzungen des Gebirgsfeldes eine Siedlung oder Stadt erbaut (oder gegründet) haben, hier Weiß und Rot, den Rohstoff, den Gebirge abwerfen: Erz. Eine Stadt verdoppelt den Ertrag. Andere Landfelder liefern folgende Rohstoffe: Getreide vom Ackerland, Bausteine (Hügelland), Holz (Wald), Wolle (Weideland). Die Wüste wirft keine Rohstoffe ab.

Der Spieler am Zug kann jetzt mit den erhaltenen Rohstoffen Aktionen ausführen, während seine Mitspieler mit dem Einsatz ihrer Rohstoffe warten müssen, bis sie wieder am Zug sind. Es gibt drei Kategorien von Aktionen:

- *Handel.* Beim Handeln tauscht der Spieler am Zug mit seinen Mitspielern, der Bank oder in einem Hafen Rohstoffe. Das ist nützlich, wenn ihm ein bestimmter Rohstoff zum Bauen fehlt und er, um diesen zu bekommen, andere im Moment nicht benötigte Rohstoffe eintauschen möchte.
- *Bau.* Hier besiedelt der Spieler mit seinen Spielfiguren die Insel Catan. Mit Straßen bebaut er die Wege; Siedlungen und Städte errichtet er auf den Kreuzungen. Das hat seinen Preis: So kostet beispielsweise das Verlegen einer Straße 1 Rohstoff Holz und 1 Rohstoff Bausteine, der Bau einer Stadt 2 Getreide und 3 Erz. Aber mit jeder Besiedlung gewinnt der Spieler wichtige Siegpunkte hinzu. Für den längsten zusammenhängenden Straßenzug wird die Sonderkarte „Längste Handelsstraße“ vergeben (2 Siegpunkte). Eine Siedlung bringt 1 Siegpunkt, eine Stadt 2 – und die Chancen steigen, beim nächsten Auswürfeln der Rohstofferträge noch mehr Rohstoffe zu bekommen.

- *Entwicklung.* Beim Entwickeln kauft der Spieler Entwicklungskarten, von denen er pro Zug eine aufdecken kann. An die Karten sind Ereignisse geknüpft, die dem Spieler Vorteile verschaffen, wie z.B. kostenloser Bau von zwei Straßen.

Natürlich läuft die Besiedlung Catans nicht ohne Zwischenfälle ab. Ein Bösewicht, der Räuber, treibt auf der Insel sein Unwesen. Er wird immer dann aktiv, wenn ein Spieler beim Auswürfeln der Rohstofferträge eine 7 würfelt. Dann bestraft der Räuber alle Spieler, die sich mit mehr als sieben Rohstoffen bevorratet haben: Sie müssen die Hälfte davon abgeben. Zusätzlich muß der Spieler am Zug den Räuber auf ein Landfeld versetzen und darf bei Mitspielern, die auf den Kreuzungen des Feldes gebaut haben, einen Rohstoff stehlen. Solange der Räuber das Landfeld besetzt, wirft es keine Rohstofferträge mehr ab.

Der Zug eines Spielers endet, wenn der Spieler ihn für abgeschlossen erklärt. Der nächste Spieler darf zu den Würfeln greifen.

3.2.2 5 und 6 Spieler

Das Ergänzungsset „5 und 6 Spieler“ erweitert das Spiel, so daß auch 5 oder 6 Spieler an der Besiedlung Catans teilhaben können. Abgesehen von einem größeren Spielplan fügt das Set eine dritte Phase zum Spielablauf hinzu: die außerordentliche Bauphase. Diese Phase schließt sich an das Ende eines Zugs von einem Spieler an und gibt allen Mitspielern Gelegenheit, zu bauen oder Entwicklungskarten zu kaufen. Andere Aktionen, z.B. Handeln, sind verboten. Möchte niemand mehr bauen, wird die Besiedlungsphase und damit der reguläre Spielablauf wieder fortgesetzt. Die Siegpunkte zum Gewinnen eines Spiels erhöhen sich auf 13.

3.2.3 Die Seefahrer

Das Ergänzungsset „Die Seefahrer“ stattet die Spieler mit neuen Spielfiguren aus: Schiffen. Mit ihnen können die Spieler zu benachbarten Inseln übersetzen und so auch diese besiedeln. Zu entdecken gibt es dort unter anderem Goldflüsse – Spielfelder, die den Rohstoff Gold abwerfen. Erhält ein Spieler Gold, muß er dieses sofort gegen einen beliebigen anderen Rohstoff seiner Wahl eintauschen.

Da der Spielplan des Basisspiels nur aus der Insel Catan besteht, bringen die Seefahrer eine Reihe weiterer Spielpläne mit unterschiedlich vielen, kleinen und großen Inseln mit sich. Jeder der neuen Spielpläne erfordert eine bestimmte Anzahl Spieler und legt fest, wieviele Siegpunkte zum Gewinn erreicht werden müssen. Eine solche Festlegung (Spielplan, Spieleranzahl und Siegpunkte) nennt sich auch Szenario. Eine Besonderheit stellen die Entdecker-Szenarien dar. Auf ihren Spielplänen sind zu Beginn eines Spiels noch nicht alle Felder sichtbar. Die unbekannten Gegenden werden erst im Laufe des Spiels von den Schiffen der Spieler entdeckt.

Die Bedrohung Catans durch Fremde wächst. Zu dem Räuber gesellt sich der Seeräuber, der das umliegende Meer unsicher macht. Beim Würfeln einer 7 kann

der Spieler am Zug den Seeräuber anstelle des Räubers versetzen und einen Mitspieler berauben, dessen Schiffe an dem vom Seeräuber besetzten Meerfeld entlangsegeln.

3.2.4 Städte & Ritter

Das Ergänzungsset „Städte & Ritter“ beinhaltet von allen Sets die umfangreichsten Erweiterungen zum Basisspiel. Die erste Erweiterung betrifft die Gründungsphase, in der die Spieler statt einer zweiten Siedlung eine Stadt gründen. Dadurch starten sie nicht nur mit dem doppelten Kapital an Rohstoffen ins Spiel, sondern haben auch die Chance, gleich am Anfang des Spiels in den Besitz der begehrten Handelswaren zu gelangen.

Handelswaren sind, genau wie Rohstoffe, eine Währung auf Catan. Es gibt Münzen, Papier und Tuch. Der Erhalt von Handelswaren ist an den Bau (oder die Gründung) von Städten geknüpft. Städte, die am Gebirge, Wald oder Weideland erbaut sind, werfen beim Auswürfeln der Rohstofferträge keinen doppelten Ertrag ab, sondern liefern einen Rohstoff und eine Handelsware: Erz und Münzen vom Gebirge, Holz und Papier aus dem Wald, Wolle und Tuch von der Weide.

Zusätzlich zu den zwei Augenwürfeln würfelt ein Spieler zu Beginn seines Zugs mit dem Ereigniswürfel. Er zeigt entweder Schiff oder Stadttor. Die Bedeutung des Schiffs wird unten erläutert. Das Stadttor führt dazu, daß bestimmte Spieler Fortschrittskarten erhalten. Fortschrittskarten ersetzen die Entwicklungskarten des Basisspiels und ermöglichen im Vergleich zu diesen tiefere Eingriffe in das Spielgeschehen. Spielt ein Spieler eine Fortschrittskarte aus, kann er beispielsweise Mitspieler ausspionieren, ihre Städte sabotieren oder mit einem fahrenden Händler, der auf Catan Station macht, Handelsbeziehungen knüpfen. Letzteres bringt zudem 1 Siegpunkt. Niemand darf mehr als vier Fortschrittskarten besitzen; überzählige sind deshalb abzugeben. Ausgegeben werden Fortschrittskarten nur an Spieler, die Städte errichtet und ausgebaut haben.

Der Ausbau von Städten geschieht in drei Richtungen: Handel, Politik und Wissenschaft. Pro Richtung gibt es fünf Ausbaustufen, für die ein Spieler entsprechend viele Handelswaren bezahlen muß. Tuch ermöglicht den Ausbau Richtung Handel, Münzen steht für Politik und Papier für Wissenschaft. Ab der dritten Ausbaustufe ergibt sich für den Spieler ein Zusatznutzen. So kann er zum Beispiel nach dem dritten Ausbau Richtung Handel Rohstoffe und Handelswaren auf einem weiteren Handelsplatz, der Gilde, günstig tauschen. Darüber hinaus ringen die Spieler ab der vierten Ausbaustufe um die Auszeichnung ihrer Städte als Metropolen. Der Besitz einer Metropole zählt 2 Siegpunkte.

Das Ergänzungsset führt weitere Spielfiguren ein: Stadtmauern und Ritter. Der Bau einer Stadtmauer dient der Abwehr des Räubers, das Aufstellen von Rittern der Verteidigung der Insel. Denn die Bedrohung Catans erreicht hier das volle Ausmaß ihres Schreckens. Barbaren, angelockt vom Reichtum Catans, segeln heran. Zeigt der Ereigniswürfel ein Schiff, rücken sie der Insel wieder eine Seemeile näher, bis sie schließlich auf Catan landen und die Spieler überfallen. Wenn es den Rittern gelingt, die Barbaren wieder ins Meer zurückzuwerfen, winken Siegpunkte. Andernfalls brandschatzen die Barbaren eine Stadt des schwächsten Spielers. Das ist derjenige, der am wenigsten Ritter aufgestellt hat.

Alles in allem wird bei „Städte & Ritter“ das Leben auf Catan härter. Gewinner ist, wer als erster 13 Siegpunkte erreicht.

3.3 Zusammenfassung

Die Problemstellung dieser Arbeit und ihre Lösung untermauert und illustriert eine Fallstudie, in der eine Produktlinie für das Gesellschaftsspiel „Die Siedler von Catan“ in mehreren Iterationen entwickelt worden ist. Die ersten zwei Iterationen, „Basisspiel“ und „5 und 6 Spieler“, nutzten die FeatuRSEB-Methode und deckten dort Probleme auf. Die Methode und ihre Bewertung beschreibt das nächste Kapitel. Der Lösungsansatz folgt ab Kapitel 5.

Kapitel 4

Die FeatuRSEB-Methode

Dieses Kapitel beschreibt in den Unterkapiteln 4.1 bis 4.4 die in Abschnitt 2.2.4 ausgewählte Produktlinien-Entwicklungsmethode FeatuRSEB (sprich: *featured RSEB*). Die Beschreibung konzentriert sich auf die im Zusammenhang der Arbeit wichtigen Teile FeatuRSEBs. Aufbauend darauf bewertet Unterkapitel 4.5, inwieweit sich die Methode zur Entwicklung solcher Produktlinien eignet, auf deren Basis Software-Systeme in Serie gefertigt werden sollen. Illustriert werden beide Teile durch Auszüge aus der im vorangegangenen Kapitel vorgestellten Fallstudie „Die Siedler von Catan“.

4.1 Überblick

Die FeatuRSEB-Methode ist durch ihre Vorgänger geprägt (vgl. Abbildung 2.4 auf Seite 24): Feature-Oriented Domain Analysis (FODA), Object-Oriented Software Engineering (OOSE), Reuse-Driven Software Engineering Business (RSEB) und die UML [K+90, JCJÖ92, JGJ97, GFA98]. Die charakteristischen Eigenschaften von FeatuRSEB sind demzufolge:

- Beschreibung von Anforderungen an eine Produktlinie und ihre Produkte durch ein Use-Case-Modell
- Strukturierung der Produktlinien-Anforderungen in Gemeinsamkeit und Variabilität durch ein Merkmalmodell
- Objektorientierte Analyse und Entwurf von Architektur und Komponenten der Produktlinie und ihrer Produkte; Dokumentation der Ergebnisse durch ein Objektmodell
- Objektorientierte Implementierung der Produktlinie und ihrer Produkte

Aus den Eigenschaften leitet sich das Einsatzgebiet von FeatuRSEB ab: Die Methode ist zur Entwicklung solcher Produktlinien und Produkte geeignet, die einer Domäne angehören, deren Wissen mit Use-Cases beschrieben und in Objekte zerlegt werden kann. Dies ist in nahezu allen Domänen möglich, wenn auch

noch nicht überall üblich. So werden in technischen Domänen, z.B. Steuerung von Auto-Motoren, Systeme weiterhin mit strukturierten Methoden und Programmiersprachen entwickelt. FeatuRSEB ist folglich keine domänenspezifische, sondern eine allgemein anwendbare Methode.

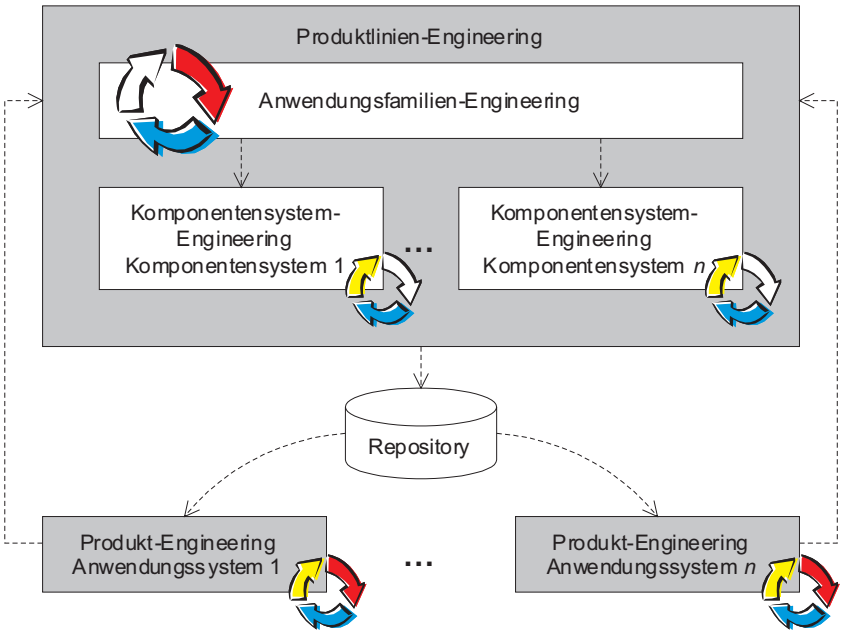

Abbildung 4.1: Vorgehensmodell in FeatuRSEB

Das Vorgehensmodell zur Entwicklung von Produktlinien und Produkten in FeatuRSEB zeigt Abbildung 4.1, in Anlehnung an [JGJ97, Abbildung 2.4] und das in Abbildung 2.3 auf Seite 20 vorgestellte allgemeine Vorgehensmodell. FeatuRSEB unterscheidet drei Typen von Entwicklungsprozessen:

- Das Produktlinien-Engineering zerfällt in *Anwendungsfamilien-Engineering* und *Komponentensystem-Engineering*. Ersteres entwickelt und wartet die Anforderungsbeschreibung der Produktlinie und ihre Architektur, letzteres die Produktlinien-Komponenten.
- Das *Produkt-Engineering* entwickelt ausgehend von Kunden-Anforderungen konkrete Anwendungssysteme auf Basis der Ergebnisse der Prozesse im Produktlinien-Engineering.

Wie schon die Namen der Prozesse andeuten, verwendet FeatuRSEB eine etwas andere Terminologie als sie Kapitel 2 eingeführt hat. Produktlinien heißen in FeatuRSEB auch Anwendungsfamilien, Produkte Anwendungssysteme. Obwohl die Zuordnung der Begriffe eindeutig ist, werden im weiteren Verlauf der Arbeit nur die Begriffe Produkt und Anwendungssystem synonym verwendet. Der Begriff Anwendungsfamilie taucht hingegen nur in dem zusammengesetzten Wort Anwendungsfamilien-Engineering auf. Damit soll verhindert werden, daß das

Produktlinien-Engineering mit seinem Subprozeß, dem Anwendungsfamilien-Engineering, gleichgesetzt wird.

Warum teilt FeatuRSEB Produktlinien-Engineering in Anwendungsfamilien-Engineering und Komponentensystem-Engineering auf? Die Architektur einer mit FeatuRSEB entwickelten Produktlinie besteht aus einzelnen Subsystemen, genannt Komponentensysteme. Das Anwendungsfamilien-Engineering legt die Architektur der Produktlinie fest, gliedert sie in Komponentensysteme und definiert deren externe Schnittstellen. Das Komponentensystem-Engineering entwirft und implementiert dann die Interna der Komponentensysteme. Pro Komponentensystem einer Produktlinie gibt es einen Komponentensystem-Engineering-Prozeß. Diese Prozeßaufteilung hilft vor allem dem Management bei der Bildung der Teams für die einzelnen Prozesse. Im Anwendungsfamilien-Engineering werden Architekten und Entwickler mit langjähriger Berufserfahrung sowie umfangreichem Domänenwissen eingesetzt, während im Komponentensystem-Engineering jüngere Entwickler und Berufsanfänger ihren Platz finden. Ein Nachteil der Prozeßaufteilung ist der erhöhte Kommunikationsbedarf zwischen den Teams. Deshalb sieht FeatuRSEB vor, daß bei kleinen Teams alle Prozesse (auch das Produkt-Engineering) in einem Prozeß vereint werden können. Kleine Teams treten typischerweise im Rahmen der Pilotphase auf, wenn ein Unternehmen auf die Produktlinien-Entwicklung umsteigt.

Die folgenden Unterkapitel 4.2 und 4.3 beschreiben die drei Prozeßtypen, ihre Phasen und deren Ergebnisse näher. Dabei werden Anwendungsfamilien-Engineering und Komponentensystem-Engineering in einem Unterkapitel „Produktlinien-Engineering“ zusammengefaßt. Anschließend diskutiert Unterkapitel 4.4 das Thema Werkzeugunterstützung für FeatuRSEB.

4.2 Produktlinien-Engineering

Ziel des Produktlinien-Engineerings (beziehungsweise Anwendungsfamilien- und Komponentensystem-Engineerings) in FeatuRSEB ist die Festlegung der gemeinsamen Architektur für alle Anwendungssysteme, die aus einer Produktlinie entwickelt werden sollen, und die Bereitstellung wiederverwendbarer Komponenten, die die Architektur ausfüllen.

4.2.1 Ablauf

Das Produktlinien-Engineering gliedert sich in drei Phasen, die zu entsprechenden Ergebnissen führen (vgl. Abbildung 4.2).

Phase 1: Beschreibung von Anforderungen. Das Anwendungsfamilien-Engineering nimmt in dieser Phase die Anforderungen an die Produktlinie auf und legt sie im Repository als Use-Case-Modell bestehend aus Glossar, Akteuren, Use-Cases und Aktivitätsgraphen ab. Parallel dazu ordnet es die Anforderungen entweder der Gemeinsamkeit oder der Variabilität der Produktlinie zu. Die Zuordnung geschieht über ein Merkmalmodell.

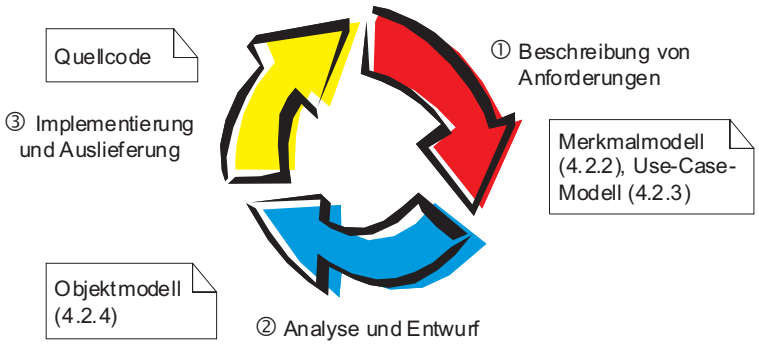

Abbildung 4.2: Ablauf Produktlinien-Engineering

Phase 2: Analyse und Entwurf. Hier leitet das Anwendungsfamilien-Engineering zunächst die Architektur der Produktlinie aus dem Use-Case-Modell ab. Die Architektur besteht aus Schichten, die in Komponentensysteme untergliedert sind. Nachdem die externen Schnittstellen der Komponentensysteme definiert sind, übernimmt das Komponentensystem-Engineering. Es entwirft die Komponentensysteme unter besonderer Berücksichtigung der Variabilität und spezifiziert die Systeme in einem Objektmodell mit Klassen und den Interaktionen ihrer Instanzen. Pro Komponentensystem gibt es einen Komponentensystem-Engineering-Prozeß.

Phase 3: Implementierung und Auslieferung. Zum Schluß folgt die Implementierung der Komponentensysteme in einer objektorientierten Programmiersprache. Nach erfolgreichem Test liefert das Komponentensystem-Engineering Objektmodell und Quellcode in das Repository aus.

4.2.2 Merkmalmodell

Das Merkmalmodell ist das zentrale Modell einer mit FeatuRSEB entwickelten Produktlinie. Es wird deswegen vor dem Use-Case-Modell vorgestellt, obwohl es zeitlich versetzt beziehungsweise parallel zu diesem erstellt wird. Das Merkmalmodell erlaubt einen schnellen Überblick zur Gemeinsamkeit und Variabilität der Produktlinie, ohne daß Entwickler und Kunden sich mit Details wie Anforderungen, Architektur oder Komponenten beschäftigen und auskennen müssen. Insofern liefert das Merkmalmodell einen wesentlichen Beitrag zum Produkt-Engineering (vgl. Unterkapitel 4.3).

Ursprünglich verwendet FeatuRSEB die Merkmalmodelle aus FODA [K+90]. Czarnecki und Eisenecker entwickelten jedoch in [CE00, Kapitel 4] eine Reihe nützlicher Erweiterungen für Merkmalmodelle, die sich problemlos in FeatuRSEB übernehmen lassen. Dieser Abschnitt stellt deshalb die erweiterte Version der Merkmalmodelle vor.

Merkmale

Ein Merkmalmodell besteht aus einer Menge von Merkmalen mit ihren Eigenschaften und ihren Beziehungen untereinander. Was sind Merkmale? Dazu zunächst eine Definition aus dem Domänen-Engineering:

> Ein *Merkmal* ist „a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances“ [CE00, Seite 755].

Bezieht man diese Definition auf die Produktlinien-Entwicklung, bezeichnet „domain concept“ die Produktlinie selbst und „concept instances“ die Produkte der Produktlinie. Danach ist ein Merkmal eine Eigenschaft, worin sich die Produkte einer Produktlinie voneinander unterscheiden können. Nun nehmen nicht alle Produktlinien-Interessenten (engl.: *stakeholder*) die gleichen Unterschiede wahr. Aus Sicht der Kunden unterscheiden sich Produkte hinsichtlich ihrer Funktionalität; aus Sicht der Entwickler zusätzlich durch die in einem Produkt enthaltenen Komponenten. Ein Merkmal bezeichnet also einmal ein Stück Funktionalität, ein anderes Mal eine Komponente usw. Im Rahmen dieser Arbeit spielen nur Merkmale aus Kundensicht eine Rolle. Darüber hinaus müssen Merkmale als gemeinsam oder variabel klassifiziert sein, damit das Merkmalmodell einen Überblick zur Gemeinsamkeit und Variabilität einer Produktlinie geben kann. Aufgrund der getroffenen Aussagen kann obige Definition jetzt konkreter gefaßt werden:

> Ein *Merkmal* ist eine Eigenschaft einer Produktlinie aus Sicht der Kunden. Ein gemeinsames Merkmal ist eine Eigenschaft, die alle Produkte einer Produktlinie aufweisen. Ein variables Merkmal ist eine Eigenschaft, worin sich Produkte einer Produktlinie voneinander unterscheiden können.

Ein Merkmalmodell wird in einem Merkmaldiagramm visualisiert. Abbildung 4.3 zeigt einen Auszug aus dem Merkmalmodell der Produktlinie „Die Siedler von Catan“. Ein Merkmaldiagramm ordnet die Merkmale eines Merkmalmodells in einem Graphen aus Knoten und Kanten an. Die Quelle des Graphen bildet der Konzeptknoten, der die Produktlinie (beziehungsweise „domain concept“ im Domänen-Engineering) repräsentiert: Die Siedler von Catan. Alle anderen Knoten stellen Merkmale der Produktlinie dar. Die Knoten sind durch Kanten miteinander verbunden, die die Merkmale zueinander in Beziehung setzen. Es gibt zwei Arten von Beziehungen: Hierarchie und Abhängigkeit.

Hierarchiebeziehungen zwischen Merkmalen

Eine Hierarchiebeziehung zeigt an, welches Merkmal welchem anderen Merkmal (beziehungsweise dem Konzeptknoten) untergeordnet ist. Die Beziehung wird durch eine gerichtete Kante mit einer durchgezogenen Linie dargestellt. Der Kreis am Ende der Linie zeigt die Richtung der Beziehung an: Das Merkmal Schiffe ist das übergeordnete Merkmal von Seeräuber; Seeräuber ist ein untergeordnetes Merkmal von Schiffe. Der Kreis klassifiziert das untergeordnete Merkmal als:

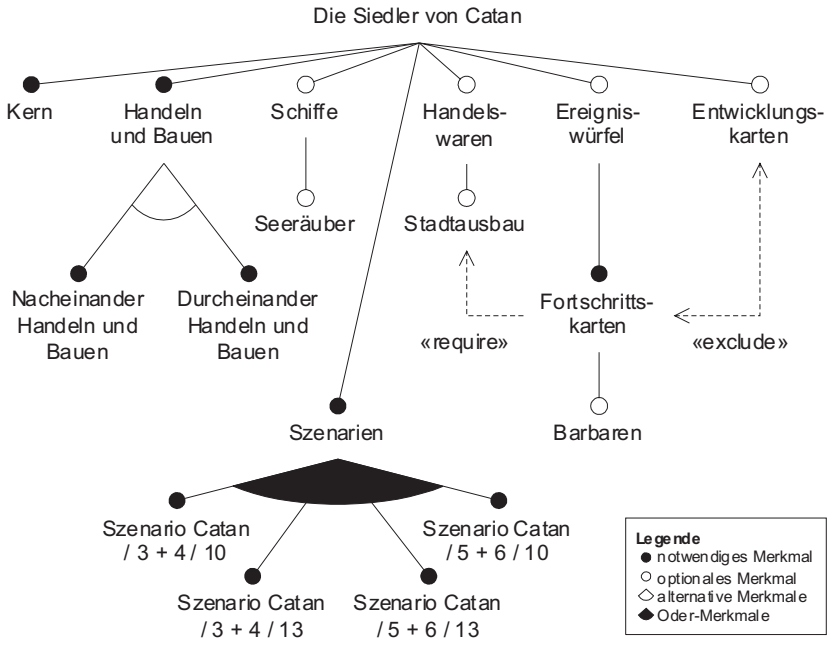

Abbildung 4.3: Beispiel Merkmalmodell

- notwendiges Merkmal (Kreis gefüllt; Kern, Fortschrittskarten usw.)
- optionales Merkmal (Kreis leer; Schiffe, Stadtausbau usw.)

Notwendige Merkmale können gruppiert werden[1], sofern sie demselben Merkmal (beziehungsweise dem Konzeptknoten) untergeordnet sind. Die Gruppierung wird durch einen Kreisbogen dargestellt, der die Kanten der Merkmale verbindet. Der Kreisbogen klassifiziert die Merkmale in der Gruppe als:

- alternative Merkmale (Kreisbogen leer; Handeln und Bauen)
- Oder-Merkmale (Kreisbogen gefüllt; Szenarien)

Was bedeuten die Klassifizierungen? Im späteren Produkt-Engineering treffen Kunden eine Vorentscheidung für die Funktionalität ihres Systems, indem sie Merkmale aus dem Merkmalmodell der Produktlinie auswählen. Die Auswahl ist jedoch nicht beliebig, sondern unterliegt Regeln. Letztere ergeben sich zum einen aus den weiter unten beschriebenen Abhängigkeitsbeziehungen zwischen Merkmalen, zum anderen aus der Hierarchie der Merkmale und ihrer Klassifizierung:

[1] In [CE00] werden auch optionale Merkmale gruppiert sowie gemischte Gruppen aus notwendigen und optionalen Merkmalen gebildet. Diese Spezialfälle traten in der Fallstudie nicht auf und werden deshalb hier aus Gründen der Vereinfachung übergangen. Ihre Berücksichtigung würde an den Aussagen der Arbeit nichts ändern.

- *Notwendige Merkmale.* Ein notwendiges, nicht gruppiertes Merkmal *muß* ein Kunde immer auswählen, sobald er das übergeordnete Merkmal auswählt oder wenn das notwendige Merkmal dem Konzeptknoten untergeordnet ist.

- *Optionale Merkmale.* Ein optionales Merkmal *kann* ein Kunde auswählen, sobald er das übergeordnete Merkmal auswählt oder wenn das optionale Merkmal dem Konzeptknoten untergeordnet ist.

- *Alternative Merkmale.* Aus einer Gruppe alternativer Merkmale *muß* ein Kunde *genau ein* Merkmal auswählen, sobald er das übergeordnete Merkmal auswählt oder wenn die Gruppe dem Konzeptknoten untergeordnet ist.

- *Oder-Merkmale.* Aus einer Gruppe von Oder-Merkmalen *muß* ein Kunde *mindestens ein* Merkmal auswählen, sobald er das übergeordnete Merkmal auswählt oder wenn die Gruppe dem Konzeptknoten untergeordnet ist. Zusätzlich kann der Kunde aus der Gruppe beliebig viele weitere Merkmale auswählen.

Aus diesen Wahlmöglichkeiten ergibt sich jetzt automatisch die Einteilung der Merkmale in gemeinsame oder variable Merkmale. Gemeinsame Merkmale sind alle notwendigen, nicht gruppierten Merkmale, die dem Konzeptknoten untergeordnet sind oder in deren übergeordneter Hierarchie sich bis hinauf zum Konzeptknoten nur notwendige, nicht gruppierte Merkmale befinden. Alle anderen Merkmale sind variabel. Im Beispiel stellen die Merkmale Kern, Handeln und Bauen und Szenarien die Gemeinsamkeit der Produktlinie dar, alle anderen Merkmale die Variabilität.

Die Hierarchiebeziehungen eines Merkmalmodells müssen widerspruchsfrei sein. So darf ein Merkmal nur genau einem übergeordneten Merkmal oder dem Konzeptknoten zugeordnet sein. Daraus folgt, daß ein Merkmal höchstens in einer Gruppe von Merkmalen enthalten sein kann. Desweiteren muß die Hierarchie der Merkmale frei von Zyklen sein.

Abhängigkeitsbeziehungen zwischen Merkmalen

Merkmale stehen meist nicht nur in einer streng hierarchischen Beziehung zueinander. Deshalb zeigen Abhängigkeitsbeziehungen an, welche Merkmale unterschiedlicher Hierarchiezweige oder -ebenen voneinander abhängen. Eine Abhängigkeitsbeziehung wird durch eine gestrichelte Linie dargestellt. Der annotierte Stereotyp weist auf die Art der Abhängigkeit hin: *«require»* oder *«exclude»*. Pfeile am Anfang und/oder Ende der Linie geben die Richtung der Beziehung an. Eine *require*-Beziehung ist uni- oder bidirektional, eine *exclude*-Beziehung immer bidirektional.

Abhängigkeitsbeziehungen beeinflussen die Auswahl von Merkmalen durch die Kunden im negativen Sinne. Sie schränken die Wahlmöglichkeiten weiter ein:

- *Require-Beziehungen.* Eine *require*-Beziehung vom Merkmal Fortschrittskarten zu Stadtausbau bedeutet, die Auswahl von Fortschrittskarten *erfordert* die Auswahl von Stadtausbau. Ein Kunde muß das Merkmal Stadtausbau mitsamt seinem übergeordneten Merkmal Handelswaren auswählen, sobald er Fortschrittskarten auswählt.
- *Exclude-Beziehungen.* Eine *exclude*-Beziehung zwischen den Merkmalen Entwicklungskarten und Fortschrittskarten bedeutet, die Auswahl von Entwicklungskarten *schließt* die Auswahl von Fortschrittskarten *aus*. Weil die Beziehung immer birektional ist, gilt umgekehrt das gleiche: Ein Kunde kann zusätzlich zu den Fortschrittskarten nicht auch noch die Entwicklungskarten wählen.

Die Abhängigkeitsbeziehungen eines Merkmalmodells müssen widerspruchsfrei sein. So dürfen beispielsweise gemeinsame Merkmale nicht von gemeinsamen oder variablen Merkmalen abhängen und umgekehrt. Ein Merkmal darf keine *require*-Beziehungen zu verschiedenen Merkmalen derselben Gruppe alternativer Merkmale unterhalten.

4.2.3 Use-Case-Modell

Das Use-Case-Modell beschreibt die Anforderungen an eine mit FeatuRSEB entwickelte Produktlinie. Es beinhaltet Glossar, Akteure, Use-Cases und Aktivitätsgraphen. Das Glossar definiert Begriffe aus der Domäne der Produktlinie. Die Akteure stehen stellvertretend für Rollen, in die Anwender schlüpfen, während sie mit einem aus der Produktlinie entwickelten System interagieren. Die Abläufe solcher Interaktionen beschreiben Use-Cases, entweder formal in UML durch Aktivitätsgraphen oder informal in Prosa. Für die informale Beschreibung legt FeatuRSEB kein Format fest. Die Literatur zum Themengebiet „Anforderungsbeschreibung mit Use-Cases“ definiert jedoch verschiedene Formate (z.B. [Coc97] und [Oes99, Seite 205ff]). Danach besteht eine Use-Case-Beschreibung wenigstens aus den folgenden vier Abschnitten:

1. *Vorbedingungen:* Zustand des Systems, damit der Use-Case durchgeführt werden kann
2. *Durchführung:* Ablauf des Use-Case, gegliedert in numerierte Schritte
3. *Nachbedingungen:* Zustand des Systems, nachdem der Use-Case erfolgreich durchgeführt wurde
4. *Ausnahmen:* Abweichungen von der normalen Durchführung, z.B. wenn Fehler fachlicher Natur auftreten

Die Zuordnung der Use-Cases und damit der Produktlinien-Anforderungen zur Gemeinsamkeit und Variabilität der Produktlinie geschieht über das Merkmalmodell (vgl. Abbildung 4.4). Ein Use-Case, der eine gemeinsame Anforderung beschreibt (Straße bauen, Sonderkarte Längste Handelsstraße vergeben), ist über eine *trace*-Abhängigkeitsbeziehung mit einem gemeinsamen Merkmal (Kern) verknüpft. Ein Use-Case, der eine variable Anforderung beschreibt (Schiff bauen,

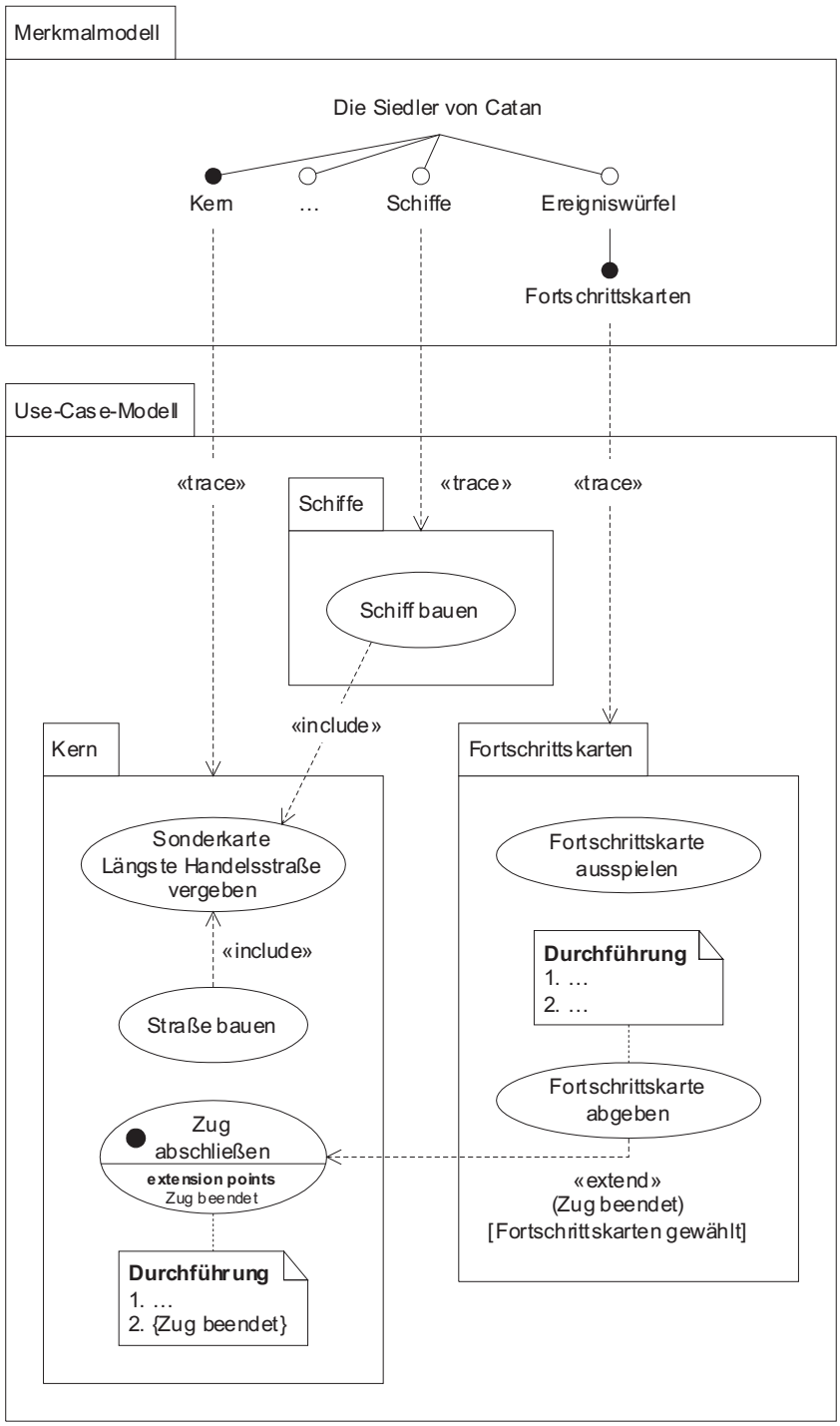

Abbildung 4.4: Beispiel Use-Case-Modell

Fortschrittskarte ausspielen usw.), ist mit einem variablen Merkmal (Schiffe beziehungsweise Fortschrittskarten) verknüpft.

Gemeinsame und variable Anforderungen können jedoch nicht immer durch klar voneinander getrennte gemeinsame *oder* variable Use-Cases beschrieben werden (vgl. Abschnitt 2.1.4). Neben *include*-Beziehungen treten aus Gründen der Modularisierung häufig auch Use-Cases auf, die an speziellen Stellen von anderen Use-Cases erweitert werden können. Solche parametrisierbaren Use-Cases kennzeichnet FeatuRSEB mit einem gefüllten Kreis, Variationspunkt genannt. Ein Beispiel ist der Use-Case Zug abschließen. Er besitzt eine Erweiterungsstelle Zug beendet, auf die sich der Use-Case Fortschrittskarte abgeben mittels *extend*-Beziehung bezieht. Wählt ein Kunde das Merkmal Fortschrittskarten aus, ergänzt Fortschrittskarte abgeben die in Zug abschließen beschriebene Interaktion. Dazu hängt der Use-Case Fortschrittskarte abgeben seine Durchführung in die Durchführung von Zug abschließen an der Stelle Zug beendet ein. Wählt der Kunde das Merkmal Fortschrittskarten nicht aus, bleibt die Stelle Zug beendet leer. Wie die Erweiterung vonstatten geht, wenn Use-Cases durch Aktivitätsgraphen spezifiziert sind, darüber trifft FeatuRSEB keine Aussage.

4.2.4 Objektmodell

Das Objektmodell spezifiziert Architektur und Komponentensysteme einer mit FeatuRSEB entwickelten Produktlinie. Die Architektur folgt immer dem Architekturmuster *Layers* [B+96, Seite 31ff]. Sie besteht also aus ein oder mehreren übereinanderliegenden Schichten, die wiederum in ein oder mehrere miteinander kommunizierende Komponentensysteme zerfallen. Ein Komponentensystem wird als Paket mit dem Stereotyp *«component system»* modelliert. Abbildung 4.5 zeigt die Architektur der Produktlinie „Die Siedler von Catan“. Wer mit wem kommuniziert, ist aus den Abhängigkeitsbeziehungen ersichtlich.

Komponentensysteme beinhalten Schnittstellen und Klassen. Sie beschreiben, welche Zustände die Instanzen der Klassen annehmen und wie die Instanzen interagieren, sprich welche Nachrichten sie sich in welcher Reihenfolge zuschicken. Modelliert werden Komponentensysteme durch Klassen-, Zustands- und Interaktionsdiagramme, implementiert als Quellcode in einer objektorientierten Programmiersprache. Als Beispiel ist in Abbildung 4.6 ein Ausschnitt des Komponentensystems catan::spiel zu sehen.

Das Überschneidungsproblem

Wie spiegelt sich das Use-Case-Modell im Komponentensystem wider? Die Abbildung von Use-Cases auf Klassen gelingt in FeatuRSEB, wie in der objektorientierten Softwareentwicklung allgemein üblich, selten eins zu eins. Im Komponentensystem catan::spiel entstand z.B. das Attribut fortschrittskarten der Klasse Spieler aus den Use-Cases Fortschrittskarte ausspielen und Fortschrittskarte abgeben, wohingegen die Operation getErbauteSchiffe() in derselben Klasse aus dem Use-Case Schiff bauen entstand. Eine Ausnahme, bei der die Abbildung eins zu eins gelang, ist die Klasse FortschrittskarteAbgeben, die allein

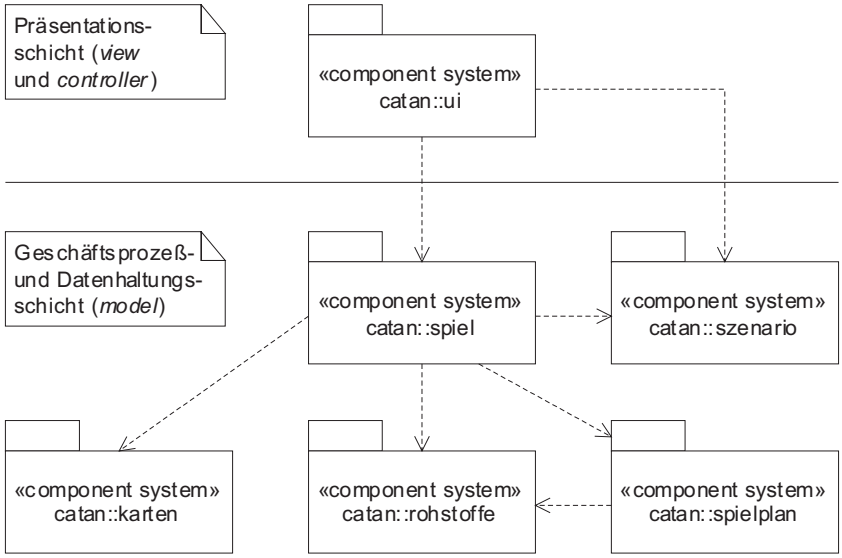

Abbildung 4.5: Beispiel Architektur

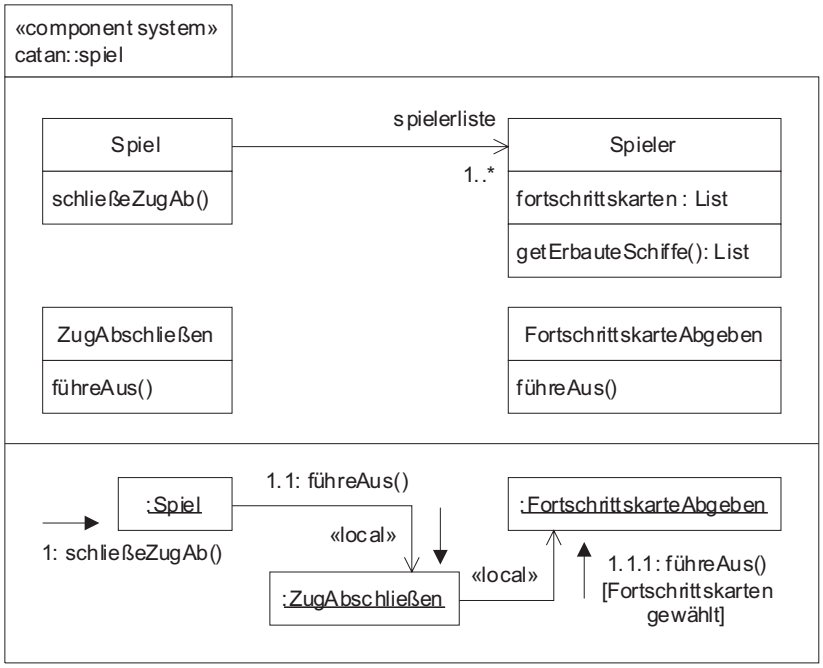

Abbildung 4.6: Beispiel Komponentensystem

aus dem Use-Case Fortschrittskarte abgeben entstand. Ebenfalls aus diesem Use-Case entstand die Interaktion der Klasse ZugAbschließen mit Fortschrittskarte-Abgeben im Rahmen der Operation ZugAbschließen::führeAus() – die Operation entstand hingegen, nebst ihrer Klasse, aus dem Use-Case Zug abschließen. FeatuRSEB empfiehlt, solche „Was ist woraus entstanden"-Beziehungen durch *trace*-Abhängigkeitsbeziehungen zwischen Use-Case- und Objektmodell zu dokumentieren. Sie grafisch darzustellen, führt zu unübersichtlichen, schwer zu pflegenden Diagrammen und wurde deshalb in Abbildung 4.6 unterlassen.

Die Überschneidung von Use-Cases in den Klassen des Objektmodells ist eine Folge der unterschiedlichen Arten, auf die Produktlinien in den Phasen des Produktlinien-Engineerings zerlegt werden. So erfolgt die Zerlegung während der Beschreibung von Anforderungen funktional in Use-Cases, ab Analyse und Entwurf aber objektorientiert in Klassen. Der Bruch in der Zerlegung ist unproblematisch, solange es gelingt, Klassen beziehungsweise Komponentensysteme bereitzustellen, die das Produkt-Engineering leicht wiederverwenden kann. Dies trifft auf das Komponentensystem catan::spiel, so wie es in Abbildung 4.6 modelliert ist, *nicht* zu.

Angenommen, ein Kunde gibt ein System in Auftrag und wählt zusätzlich zum gemeinsamen Merkmal Kern das variable Merkmal Schiffe aus. Der Kunde will also ein System ohne das Merkmal Fortschrittskarten, sprich ohne die zwei Use-Cases Fortschrittskarte ausspielen und Fortschrittskarte abgeben. Die Entwickler im Produkt-Engineering stehen jetzt vor einem Problem: Einerseits bietet sich eine Wiederverwendung der Klasse Spieler an, weil der Kunde das Merkmal Schiffe gewählt hat und die Klasse hierfür die Operation getErbauteSchiffe() bereitstellt. Andererseits verbietet sich die Wiederverwendung, weil der Kunde das Merkmal Fortschrittskarten nicht gewählt hat, die Klasse Spieler aber das Attribut fortschrittskarten definiert. Eine ähnliche Situation ergibt sich, möchten die Entwickler die Klasse ZugAbschließen wiederverwenden. Einerseits implementiert sie Teile des vom Kunden gewählten Merkmals Kern, andererseits enthält sie aber auch Teile des nicht gewählten Merkmals Fortschrittskarten. Im Endeffekt müssen die Entwickler die Klassen kopieren und entsprechend abändern. Das ist fehleranfällig; eine Wiederverwendung findet nicht statt. Die Vorteile von Produktlinien wie schnelle Marktreife schwinden.

... und eine Lösung

FeatuRSEB löst das Überschneidungsproblem, indem es variable Anteile aus Klassen herauslöst und in andere, neue Klassen auslagert. Die Integration der ausgelagerten Teile in die alten Klassen geschieht durch Vererbung, Einsatz von Entwurfsmustern oder eine Kombination aus beidem. Von den in [GHJV95] vorgestellten Mustern sind insbesondere geeignet: Abstrakte Fabrik, Befehl, Besucher, Brücke, Dekorierer, Erbauer, Fabrikmethode, Schablonenmethode und Strategie. Setzt man Vererbung und eine Abstrakte Fabrik ein, um das Komponentensystem catan::spiel zu refaktorieren, ändert sich das Komponentensystem wie in Abbildung 4.7 gezeigt. Die Quellcodes sind in der Syntax von Java formuliert. Ein weiteres Beispiel, diesmal für den Einsatz des Strategie-Musters, liefert Abschnitt 4.5.3.

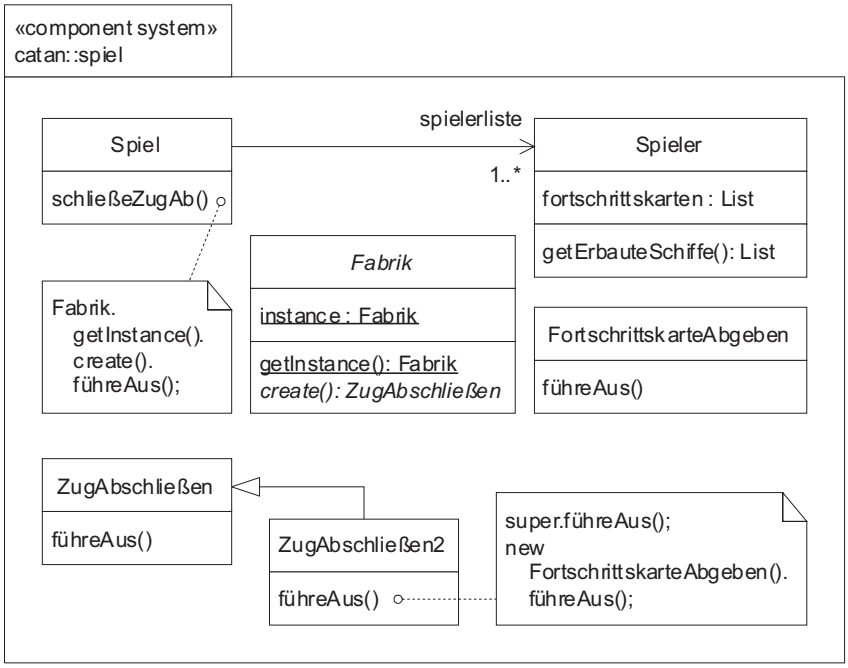

Abbildung 4.7: Beispiel Komponentensystem (refaktoriert)

Die refaktorierte Version des Komponentensystems definiert eine neue Unterklasse von ZugAbschließen. Die Unterklasse kapselt die variable Interaktion mit der Klasse FortschrittskarteAbgeben und überschreibt die Operation führeAus(). Damit ein Aufruf der Operation Spiel::schließeZugAb() den gewünschten Effekt hat, instanziiert ein Spiel die Klasse ZugAbschließen beziehungsweise ihre Unterklasse über den Umweg der abstrakten Klasse Fabrik. Entwickler können die Klasse ZugAbschließen jetzt für das System des oben erwähnten fiktiven Kunden wiederverwenden, ohne daß Teile des vom Kunden nicht gewählten Merkmals Fortschrittskarten in das System einfließen. Dazu müssen sie lediglich eine konkrete Unterklasse der Fabrik definieren, dort die Operation create() überschreiben und zur Laufzeit des Systems eine Instanz der konkreten Fabrik im Klassenattribut Fabrik::instance registrieren. Die Operation create() instanziiert ZugAbschließen oder, wenn der Kunde sich doch für das Merkmal Fortschrittskarten entscheidet, ZugAbschließen2 (vgl. Abschnitt 4.3.2).

Letztlich ist durch die Refaktorierung ein Framework entstanden, das Entwickler im Produkt-Engineering spezialisieren müssen, um es wiederverwenden zu können. Damit ist das Überschneidungsproblem aber noch nicht vollständig gelöst. Die Klasse Spieler ist nach wie vor davon betroffen. Hinweise auf weiterführende Refaktorierungen und eine Analyse der dadurch entstehenden Konsequenzen auf Komplexität, Verständlichkeit und Wartbarkeit der Produktlinie folgen in Abschnitt 4.5.2.

4.3 Produkt-Engineering

Ziel des Produkt-Engineerings in FeatuRSEB ist die Entwicklung von Anwendungssystemen für Kunden auf Basis einer Produktlinie.

4.3.1 Ablauf

Das Produkt-Engineering gliedert sich in drei Phasen, die zu entsprechenden Ergebnissen führen (vgl. Abbildung 4.8).

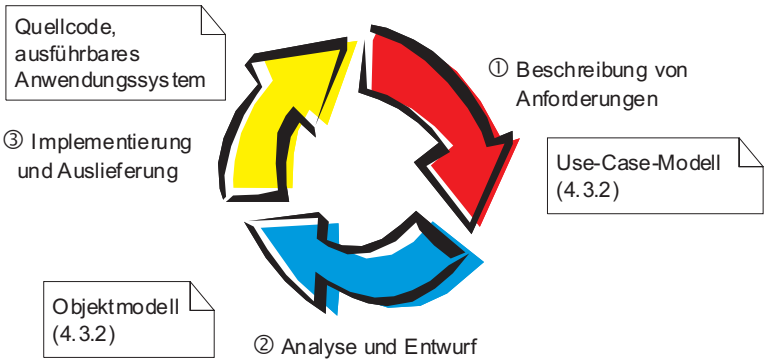

Abbildung 4.8: Ablauf Produkt-Engineering

Phase 1: Beschreibung von Anforderungen. In dieser Phase sind die Anforderungen des Kunden aufzunehmen und mit den Anforderungen, die die Produktlinie bereits erfüllt, abzugleichen. Dabei hilft das Merkmalmodell. Bevor die Anforderungen des Kunden detailliert aufgenommen werden, verschafft sich der Kunde, unter Anleitung durch die Entwickler, anhand der Merkmale im Merkmalmodell einen ersten Überblick über die Funktionalität, die er für sein Anwendungssystem auswählen kann. Unter Einhaltung der im Modell hinterlegten Regeln entscheidet der Kunde sich dann für bestimmte Merkmale, aus denen die Entwickler durch Verfolgen der *trace*-Abhängigkeitsbeziehungen eine Reihe von Use-Cases ableiten. Diese Use-Cases bilden die Ausgangsbasis, um die Anforderungen für das System des Kunden zu beschreiben. Stimmt die Ausgangsbasis nicht genau mit den Anforderungen des Kunden überein, gibt es, wie in Abschnitt 2.2.3 „Produkt-Engineering“ erläutert, mehrere Varianten, damit umzugehen: Produktlinie ausbauen, Anforderungen individuell realisieren oder Kunde zum Verzicht bewegen. Ergebnis der ersten Phase ist das Use-Case-Modell des Anwendungssystems.

Phase 2: Analyse und Entwurf. Hier wird der Umstand genutzt, daß Use-Cases der Produktlinie mit den aus ihnen hervorgegangenen Klassen im Produktlinien-Objektmodell durch *trace*-Abhängigkeitsbeziehungen verknüpft

sind. Dadurch können die Entwickler für jeden Use-Case, den das Anwendungssystem wiederverwendet, auch dessen Klassen im Objektmodell des Anwendungssystems wiederverwenden – vorausgesetzt, im Produktlinien-Engineering wurde beim Entwurf der Klassen das Überschneidungsproblem vermieden. Erfordern wiederverwendete Klassen eine Spezialisierung beziehungsweise Parametrisierung, ist sie in dieser Phase vorzunehmen. Gegebenenfalls sind neue Klassen für individuelle Anforderungen des Kunden zu entwerfen und in das Objektmodell des Anwendungssystems zu integrieren.

Phase 3: Implementierung und Auslieferung. Zum Schluß folgen Test und Auslieferung des Anwendungssystems an den Kunden. Gegebenenfalls sind vorher individuell entworfene Klassen zu implementieren.

4.3.2 Anwendungssysteme

Ein Anwendungssystem bezeichnet in FeatuRSEB ein aus einer Produktlinie entwickeltes Produkt. Es beinhaltet ein Use-Case-Modell und Objektmodell, deren Use-Cases und Klassen zu einem großen Teil aus den entsprechenden Modellen der Produktlinie wiederverwendet wurden. Ein Anwendungssystem wird als Paket mit dem Stereotyp *«application system»* modelliert. Die im Anwendungssystem wiederverwendeten Use-Cases und Klassen sind durch *import*-Beziehungen mit ihrem Original in der Produktlinie verknüpft.

Abbildung 4.9 zeigt, wie Anwendungssysteme und Produktlinien zusammenhängen. Dazu wird auf den fiktiven Kunden aus Abschnitt 4.2.4 zurückgegriffen, der für sein Anwendungssystem die Merkmale Kern und Schiffe auswählt. Dies führt zur Wiederverwendung der mit den Merkmalen verknüpften Use-Cases, von Zug abschließen bis hin zu Schiff bauen. Nicht genutzt werden hingegen die zwei Use-Cases des Merkmals Fortschrittskarten: Fortschrittskarte ausspielen und Fortschrittskarte abgeben. Damit bleibt im Use-Case Zug abschließen die Erweiterungsstelle Zug beendet leer.

Im Objektmodell des Anwendungssystems wurden die aus den wiederverwendeten Use-Cases hervorgegangenen Klassen ebenfalls wiederverwendet, namentlich Spiel, Spieler, Fabrik und ZugAbschließen. Nicht benötigt wurde die Unterklasse von ZugAbschließen und die Klasse FortschrittskarteAbgeben, weil beide indirekt dem Merkmal Fortschrittskarten zugeordnet sind. Diese Zuordnung gilt auch für das Attribut Spieler::fortschrittskarten, so daß das Attribut im Anwendungssystem eigentlich nicht benötigt wird. Dennoch wurde die Klasse Spieler mitsamt fortschrittskarten in das Anwendungssystem importiert, da das Attribut aufgrund der schlechten Modularisierung nicht entfernt werden konnte. Die Alternative wäre eine Re-Implementierung der in Spieler enthaltenen, benötigten Funktionalität gewesen, in diesem Fall der Operation getErbauteSchiffe(). Um den Aufwand für die Re-Implementierung zu sparen, wird bewußt in Kauf genommen, daß das Anwendungssystem unnötige Funktionalität enthält.

Über die importierten Klassen hinaus spezialisiert das Anwendungssystem noch die Fabrik durch die konkrete Unterklasse MeineFabrik. Dort ist die Operation create() so überschrieben, daß sie eine Instanz von ZugAbschließen zurückliefert.

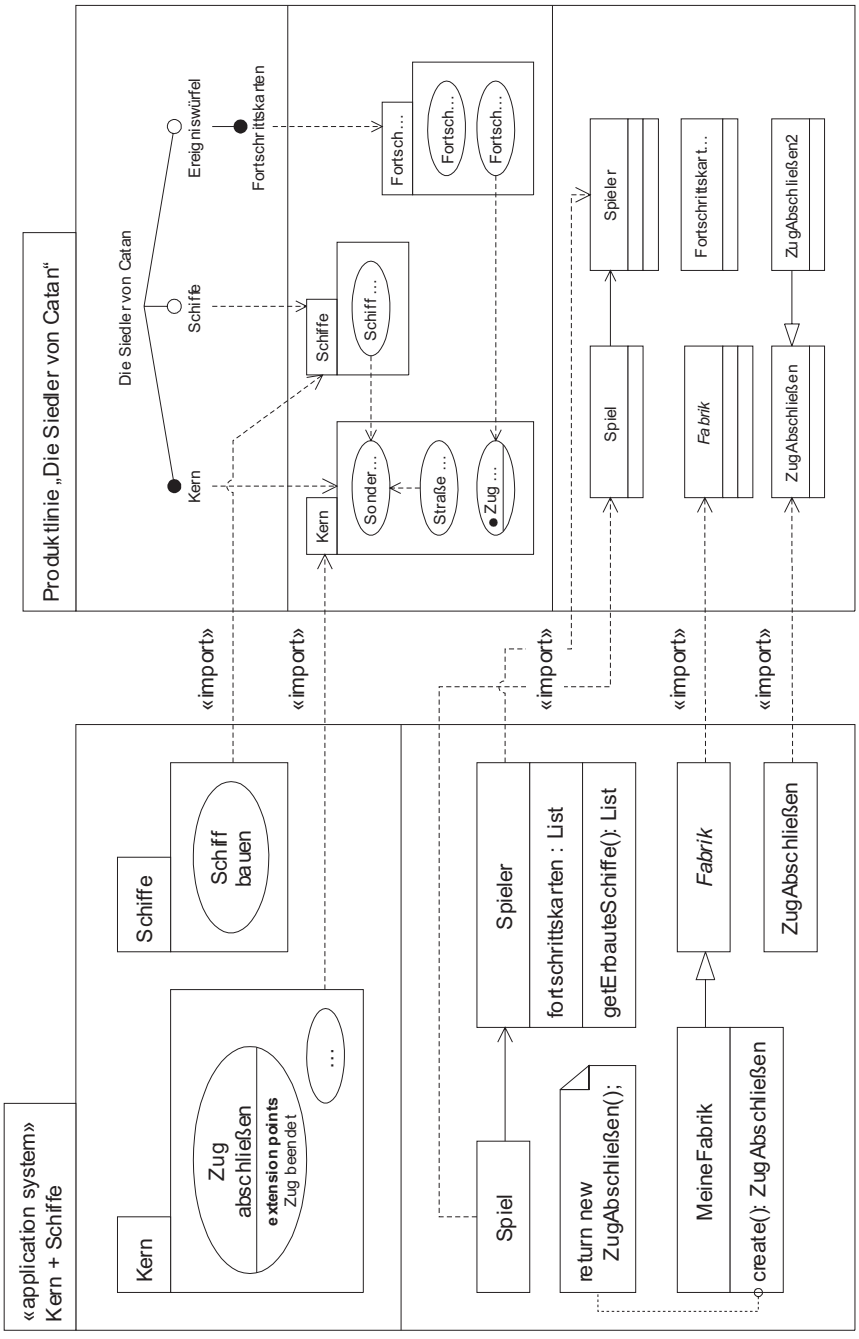

Abbildung 4.9: Beispiel Anwendungssystem

Damit wird der Abschluß eines Zugs, wie vom Kunden gewünscht und im Use-Case-Modell beschrieben, ohne das Abgeben einer Fortschrittskarte ausgeführt.

4.4 Werkzeuge

Sowohl das Produktlinien-Engineering als auch das Produkt-Engineering von FeatuRSEB müssen durch Werkzeuge unterstützt werden. Dazu reichen herkömmliche CASE-Werkzeuge und Entwicklungsumgebungen nur bedingt aus. Das Hauptproblem stellt das Merkmalmodell dar, dessen Merkmale mit ihren Beziehungen zu verwalten und in Merkmaldiagrammen zu visualisieren sind. Weiterhin muß ein Werkzeug die Auswahl von Merkmalen für das Produkt-Engineering zulassen und prüfen, ob die Auswahl den Regeln im Merkmalmodell genügt. Kommerzielle Werkzeuge zur Modellierung und Auswahl von Merkmalen sind zum Zeitpunkt dieser Arbeit nicht auf dem Markt. Selbig und Blinn entwickelten jedoch ein freies Werkzeug namens AmiEddi, das zumindest die Modellierung von Merkmalen erlaubt [Sel00, Bli01, AE]. Was AmiEddi für den Einsatz im Rahmen von FeatuRSEB fehlt, ist (a) eine Anbindung an CASE-Werkzeuge, um *trace*-Abhängigkeitsbeziehungen vom Merkmalmodell zum Use-Case-Modell zu verwalten und (b) die Auswahl von Merkmalen.

Ein weiteres Problem bestehender CASE-Werkzeuge ist die rein visuelle Verwaltung von Beziehungen zwischen Modellelementen. FeatuRSEB macht regen Gebrauch von *trace*-Beziehungen. Sie alle visuell zu verwalten, führt schnell zu unübersichtlichen Modellen und ist deshalb kaum praktikabel.

Weniger problematisch gestaltet sich die Übernahme der UML-Erweiterungen von FeatuRSEB in CASE-Werkzeuge. Zu den Erweiterungen zählen die neuen Stereotype für Pakete, *«component system»* und *«application system»*, sowie der Variationspunkt. Stereotype können in nahezu allen CASE-Werkzeugen hinzugefügt und anschließend in Modellen verwendet werden. Anders der Variationspunkt: Ihn hinzuzufügen bedeutet, das Metamodell und die Notation des Werkzeugs zu erweitern. Dies lassen nur wenige Werkzeuge zu, z.B. MetaEdit+ [ME] oder ArgoUML [AU]. Bei allen übrigen Werkzeugen muß auf den Variationspunkt verzichtet werden. Seine Semantik läßt sich jedoch nachbilden, beispielsweise indem parametrisierbare Use-Cases farblich besonders gekennzeichnet werden.

4.5 Bewertung

Die Bewertung der FeatuRSEB-Methode ist nicht allgemein gehalten, sondern an der Zielsetzung der Arbeit ausgerichtet: die objektorientierte, UML-basierte Entwicklung von Produktlinien (a) zur Serienfertigung schlanker Systeme und (b) unter Abschwächung des Nachteils komplexer, unwartbarer Produktlinien, der aus der Forderung nach schlanken Systemen erwächst (vgl. Unterkapitel 1.3 sowie 2.3, insbesondere Abschnitt 2.3.2). Grundlage für die Bewertung von FeatuRSEB sind zwei Fallstudien:

1. *Bibliothekssysteme:* Entwicklung einer Produktlinie für Bibliothekssysteme, z.B. für Hochschulbibliotheken, Stadtbüchereien und ihre ländlichen Ableger, die Fahrbüchereien beziehungsweise Bücherbusse

2. *Die Siedler von Catan:* Entwicklung einer Produktlinie für das Gesellschaftsspiel „Die Siedler von Catan"

Die erste Fallstudie wurde zeitlich vor der zweiten durchgeführt. Ihre Ergebnisse sind in [BP00, PBSR02] veröffentlicht. Die bei der Bibliothekssystem-Produktlinie gewonnene erste, negative Einschätzung FeatuRSEBs bezüglich des oben genannten Ziels (b) erhärtete sich während der Durchführung der zweiten Fallstudie und führte nach Erreichen der Ausbaustufe „5 und 6 Spieler" zum vorzeitigen Abbruch der Studie. Die zwei weiteren Ausbaustufen „Die Seefahrer" und „Städte & Ritter" wurden statt dessen mit einer verbesserten, im Rahmen der vorliegenden Arbeit aufgestellten Methode verwirklicht (vgl. Kapitel 7).

Somit ist die Bewertung der FeatuRSEB-Methode, so wie sie dieses Unterkapitel vorstellt, die Summe der Erkenntnisse aus beiden Fallstudien. Beispiele aus der Studie „Die Siedler von Catan" illustrieren die Bewertung.

4.5.1 Serienfertigung schlanker Systeme

Serienfertigung bedeutet automatisiertes Produkt-Engineering. Mit FeatuRSEB müßten folglich Anwendungssysteme nicht mehr manuell entwickelt, sondern automatisiert von einem Generator zusammengesetzt werden. Schlanke Systeme erfordern, daß die Produktlinien-Komponenten so feingranular modularisiert sind, daß der Generator nur die von den Kunden gewünschte Funktionalität in die Anwendungssysteme aufnimmt. In FeatuRSEB müßte also im Objektmodell das Überschneidungsproblem durch Einsatz von Vererbung und Entwurfsmustern vermieden werden, wie am Beispiel der Klasse ZugAbschließen gezeigt. Ansonsten landet unnötige Funktionalität im Anwendungssystem, wie es die Klasse Spieler demonstrierte.

Der generelle Ablauf einer Serienfertigung war in Abbildung 2.6 auf Seite 27 zu sehen. Er besteht aus drei Schritten: System konfigurieren, Konfiguration überprüfen und System zusammensetzen. Dabei sind eine Reihe technischer Voraussetzungen zu berücksichtigen, die Abschnitt 2.3.3 zusammenstellte. Im folgenden wird nun untersucht, wie die Schritte inhaltlich aussehen, wenn die der Fertigung zugrundeliegende Produktlinie mit FeatuRSEB entwickelt wurde.

Schritt 1: System konfigurieren

Ein Kunde konfiguriert sein Anwendungssystem auf Basis des Merkmalmodells, das das Produktlinien-Engineering aufgestellt hat. Dazu empfiehlt es sich, die Wahlmöglichkeiten, die sich aus den Beziehungen zwischen den Merkmalen ergeben, in eine grafische Benutzeroberfläche zu verpacken, ähnlich wie dies beispielsweise beim Volkswagen-Konfigurator erfolgt ist (vgl. Abbildung 2.5 auf Seite 27). Eine *checkbox* repräsentiert ein optionales Merkmal, eine Liste mit *radio buttons* eine Gruppe alternativer Merkmale. Weitere Vorschläge für die

Umwandlung von Merkmalmodellen in grafische Benutzeroberflächen beschreibt [Mac01, Unterkapitel 6.4]. Dort wird auch diskutiert, inwieweit die Umwandlung automatisiert werden kann.

Nachdem der Kunde seine Auswahl abgeschlossen hat, werden die ausgewählten Merkmale als Konfiguration gespeichert. Eine Konfiguration beinhaltet in FeatuRSEB also lediglich eine Liste mit Merkmalen aus dem Merkmalmodell der Produktlinie.

Schritt 2: Konfiguration überprüfen

Hier prüft der Generator die im ersten Schritt gespeicherte Konfiguration auf Fehler. Er muß sicherstellen, daß die Konfiguration alle gemeinsamen Merkmale enthält und daß sie keine Hierarchie- und Abhängigkeitsbeziehung verletzt. Viele Fehlermöglichkeiten können bereits durch Vorkehrungen in der grafischen Benutzeroberfläche ausgeschlossen werden. So erzwingt eine Liste mit *radio buttons*, daß Kunden immer genau einen *button* und damit nur ein alternatives Merkmal aus einer Gruppe wählen.

Ist die Konfiguration fehlerhaft, wäre es wünschenswert, daß der Generator den Kunden darauf hinweist, warum der Fehler entsteht und wie er zu beheben ist. Dann kann der Kunde selbständig die Korrektur vornehmen. Solche Erläuterungen könnten in FeatuRSEB im Merkmalmodell der Produktlinie hinterlegt werden.

Schritt 3: System zusammensetzen

Der letzte Schritt im Fertigungsablauf beginnt, indem der Generator ein neues, zunächst leeres Anwendungssystem anlegt. In dieses importiert er aus dem Use-Case-Modell der Produktlinie alle Use-Cases, die über *trace*-Abhängigkeitsbeziehungen mit den Merkmalen aus der Konfiguration verknüpft sind. Dann verfolgt der Generator die *trace*-Beziehungen von den importierten Use-Cases zum Objektmodell der Produktlinie und ermittelt darüber, welche Klassen er in das Anwendungssystem importieren muß. Am Ende enthält das Anwendungssystem ein Use-Case-Modell und ein Objektmodell, genau so, als wäre es manuell entwickelt worden.

Wie das Beispiel der Klasse Fabrik demonstrierte, führt in FeatuRSEB die Modularisierung im Objektmodell von Produktlinien häufig dazu, daß Frameworks entstehen, die das Produkt-Engineering spezialisieren muß. Da bei einer Serienfertigung die manuelle Spezialisierung solcher Frameworks ausscheidet, müssen Frameworks entweder durch eine andere Modularisierung vermieden werden oder so einfach zu spezialisieren sein, daß der Generator, ausgestattet mit entsprechendem Wissen, die Spezialisierung nach dem Zusammenstellen des Objektmodells des Anwendungssystems selbst durchführen kann. Ein Ansatz, um solches Wissen zu formulieren, findet sich in [Iva99].

Sind noch individuelle Anforderungen des Kunden zu realisieren, könnte dies jetzt geschehen. Dazu ändern oder erweitern die Entwickler das vom Generator zusammengesetzte Anwendungssystem.

Die Fertigung endet, indem der Generator den Quellcode des Anwendungssystems übersetzt und das System an den Kunden ausliefert. Damit der Generator unabhängig von der bei der Produktlinien-Entwicklung verwendeten Programmiersprache ist, sollte er die Übersetzung des Systems einem Compiler übertragen.

Bewertung

FeatuRSEB sah ursprünglich nicht die Entwicklung von Produktlinien vor, aus denen Anwendungssysteme in Serie gefertigt werden sollen. Dennoch, wie oben erläutert, bietet die Methode Voraussetzungen auch zur Entwicklung genau solcher Produktlinien. Dabei ist unbedingt sicherzustellen, daß die Produktlinie alle für den Generator notwendigen *trace*-Beziehungen definiert. Sonst sind die generierten Anwendungssysteme unter Umständen fehlerhaft und können nicht übersetzt und ausgeführt werden. Darüber hinaus ist der in Unterkapitel 4.4 aufgestellte Forderungskatalog für Werkzeugunterstützung in FeatuRSEB um den Bau des skizzierten Generators zu ergänzen.

Aber genügen die gefertigten Anwendungssysteme dem Kriterium „Schlanke Systeme“? Ja, vorausgesetzt die Produktlinie ist so feingranular modelliert und implementiert, daß das Überschneidungsproblem nicht auftritt. Denn dann kann der Generator wirklich nur die benötigten Schnittstellen, Klassen, Attribute, Assoziationen und Operationen aus der Produktlinie in das Anwendungssystem importieren. Andernfalls wird das Kriterium nicht oder nur zum Teil erreicht. Zu welchen Konsequenzen die theoretische Forderung nach einer Lösung des Überschneidungsproblems mittels Vererbung und Entwurfsmustern führt, erläutert der nächste Abschnitt.

4.5.2 Komplexität entwickelter Produktlinien

Vergleicht man die Komplexität einzelner Software-Systeme mit der von Produktlinien, so ist die Komplexität bei Produktlinien in der Regel höher. Das liegt daran, daß eine Produktlinie mehrere einzelne Systeme quasi beinhaltet und sich ihre Komplexität aus der Summe der Komplexität der enthaltenen Systeme errechnet, abzüglich eines Faktors für die Gemeinsamkeit, die die Systeme aufweisen, zuzüglich eines Faktors für die Abstraktionen, die zur Modellierung und Implementierung der Variabilität notwendig sind. Weil die Komplexität von Produktlinien kaum reduziert werden kann, zielt die Produktlinien-Entwicklung vor allem darauf ab, die Komplexität wenigstens handhabbar zu halten, um so die Verständlichkeit und spätere Wartbarkeit von Produktlinien zu erleichtern (vgl. nächster Abschnitt 4.5.3).

In einer mit FeatuRSEB entwickelten Produktlinie wird die Komplexität am stärksten durch das Objektmodell beeinflußt. Je mehr Klassen, Assoziationen und andere Beziehungen dort existieren, desto unüberschaubarer und komplexer ist die Produktlinie für Entwickler.

Die Lösung des Überschneidungsproblems

Eines der grundlegenden Probleme bei der Entwicklung von Produktlinien mit FeatuRSEB ist das Überschneidungsproblem. Solange es nicht gelöst wird, sind Klassen und Komponentensysteme im Objektmodell nur eingeschränkt wiederverwendbar. Das wiederum hat zur Folge, daß die Vorteile von Produktlinien nicht voll zum Tragen kommen. Wer FeatuRSEB zur Entwicklung von Produktlinien nutzt, aus denen schlanke Systeme in Serie gefertigt werden sollen, für den ist eine Lösung des Überschneidungsproblems nicht nur wegen der Wiederverwendbarkeit wichtig, sondern auch um fette, ineffiziente Systeme zu verhindern.

Offenbar ist eine erfolgreiche Lösung des Überschneidungsproblems von essentieller Bedeutung. FeatuRSEB bietet als Lösung an: Auslagerung variabler Anteile aus Klassen und Re-Integration durch Vererbung und Entwurfsmuster (vgl. Abschnitt 4.2.4). Leider birgt die Lösung zwei Nachteile: Sie bewirkt einen erheblichen Anstieg der Komplexität der Produktlinie, was dem Ziel handhabbarer Komplexität zuwiderläuft; und sie funktioniert nur bis zu einem gewissen Umfang der Produktlinie, sprich sie skaliert nicht.

... erhöht die Komplexität

Mehr Vererbung heißt tiefere Klassenhierarchien sowie mehr abstrakte Klassen und Operationen. Mehr Entwurfsmuster bringen mehr Klassen mit sich, mehr Attribute, Assoziationen und Operationen sowie komplexere Interaktionen zwischen den Instanzen der Klassen. In beiden Fällen entstehen viele künstliche Abstraktionen. Diese dienen allein dem Zweck, die variablen Anteile voneinander zu trennen. Einen Beitrag zur eigentlichen Funktionalität der Produktlinie leisten sie nicht. Statt dessen wird die Funktionalität in viele kleine Fragmente aufgetrennt und verteilt – entweder über die Vererbungshierarchie oder auf die an einem Muster beteiligten Klassen. Das führt unter anderem zum Jo-Jo-Effekt (Verständnis einer Klasse erfordert Analyse der Oberklassen) [TGP89] und mündet letzten Endes in einer gestiegenen Komplexität der Produktlinie.

Als Beispiel sei erneut auf die Refaktorierung des Komponentensystems catan::spiel im Produktlinien-Engineering verwiesen (vgl. Abbildungen 4.6 auf Seite 51 und 4.7 auf Seite 53). Durch den Umbau kamen zu den bestehenden vier Klassen zwei weitere hinzu. Zusätzlich muß jedes Anwendungssystem noch eine Klasse zur Spezialisierung von Fabrik ergänzen. Der Aufruf von ZugAbschliessen::führeAus() in Spiel::schließeZugAb() erfolgt nicht mehr direkt, sondern über den Umweg der Fabrik. Weitere Beispiele für die zunehmende Komplexität liefert der nächste Abschnitt 4.5.3.

... skaliert nicht

Je mehr Merkmale zu einer Produktlinie hinzukommen, desto schwieriger gestaltet sich im Produkt-Engineering die Re-Integration ausgelagerter variabler Anteile. Ein Beispiel soll dies veranschaulichen. Abbildung 4.10 zeigt die Klasse Spiel, mit ein paar mehr Attributen und Operationen als in den Diagrammen bisher zu sehen waren. Die Bestandteile der Klasse sind nach Merkmalen gruppiert.

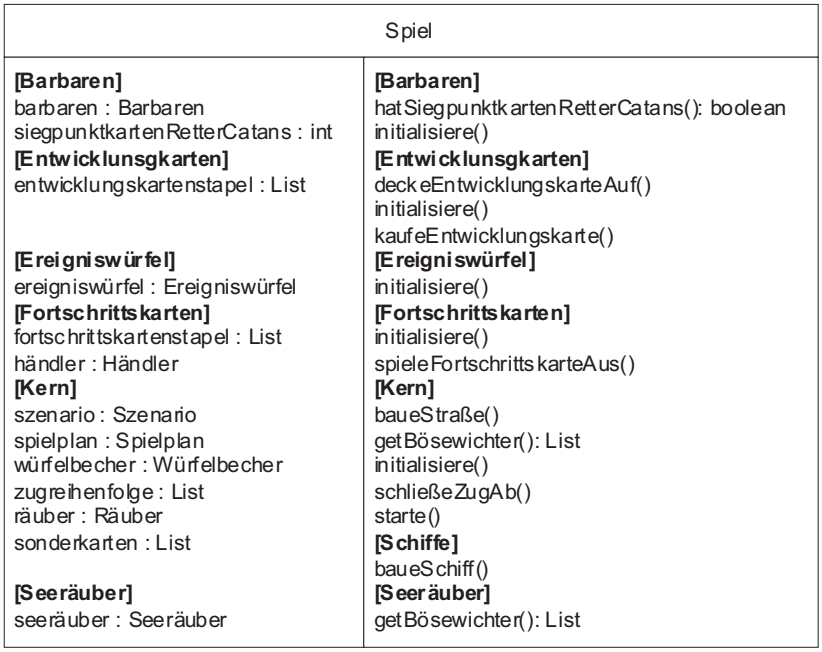

Abbildung 4.10: Komplexität entwickelter Produktlinien

Es gibt keinen praktikablen Weg, die Überschneidungen innerhalb dieser Klasse so aufzulösen, daß ein Anwendungssystem mit bestimmten Merkmalen die aufgetrennten Teile wieder integrieren kann. Das wird vor allem auch durch die Interaktion der Merkmale verhindert. Viele Merkmale greifen auf dieselben Attribute zu und steuern Quellcode zu denselben Operationen bei. So wird die Operation getBösewichter() von den Merkmalen Kern und Seeräuber beeinflußt. Die Operation initialisiere() ergänzen nahezu alle aufgeführten Merkmale um einige Anweisungen. Ein Lösungsversuch wäre die Nutzung von Mehrfachvererbung. Das würde allerdings wegen der Kombinationsvielfalt der Merkmale zu einer exponentiellen, unwartbaren Zunahme an Unterklassen von Spiel führen. FeatuRSEBs Lösung des Überschneidungsproblems skaliert somit nur bis zu Produktlinien kleineren Umfangs. Die Fallstudie „Die Siedler von Catan“ überschritt diese Grenze nach Erreichen der Ausbaustufe „5 und 6 Spieler“ und mußte abgebrochen werden, weil die Komplexität nicht mehr zu überblicken war.

Bewertung

Die Komplexität der mit FeatuRSEB entwickelten Produktlinien ist aufgrund der gewählten Lösung des Überschneidungsproblems zu hoch und nicht mehr zu handhaben. Dies schränkt schon bei kleinen Produktlinien die Entwicklung oder Serienfertigung von Produkten erheblich ein und verhindert einen Ausbau der Produktlinie. Neue Kundenkreise bleiben verschlossen, und die Produktlinie verfehlt unter Umständen den Kostendeckungspunkt.

Damit die Entwicklung von Produktlinien mit FeatuRSEB wirtschaftlich nicht zu einem Risiko wird, muß das Überschneidungsproblem anders gelöst werden. Solche Lösungen bietet die generative Programmierung an. Eine davon, der Hyperspace-Ansatz, wird in Kapitel 5 vorgestellt und anschließend in Kapitel 7 in FeatuRSEB integriert.

4.5.3 Wartbarkeit entwickelter Produktlinien

Wie ist der Begriff Wartbarkeit im Rahmen der Bewertung von FeatuRSEB zu verstehen? Wartbarkeit heißt, wie aufwendig ist der Ausbau von Produktlinien. Ausbau wiederum bedeutet Erweiterung von Produktlinien um neue Merkmale, damit Produktlinien zusätzliche Anforderungen abdecken. Dadurch verbreitert sich die Variabilität der Produktlinie und der Kreis potentieller Kunden, für die ein System aus einer Produktlinie entwickelt oder gefertigt werden kann, vergrößert sich. Der Return on Investment steigt, die Investitionen in die Produktlinien-Entwicklung amortisieren sich schneller.

Aufgrund der Definition zählen eine Reihe typischer Wartungsaufgaben, die bei Produktlinien anfallen, nicht zum Begriff der Wartbarkeit. Inwieweit die mit FeatuRSEB entwickelten Produktlinien die Bearbeitung solcher Aufgaben erleichtern oder erschweren, wird hier folglich nicht bewertet. Die Aufgaben sind im einzelnen:

- *Änderungen am Merkmalmodell und der Produktlinien-Architektur*, die über das bloße Hinzufügen neuer Merkmale oder Komponentensysteme hinausgehen. Die Notwendigkeit derartiger Änderungen ist bei Produktlinien allgemein und insbesondere bei Wahl der revolutionären Einstiegsstrategie (vgl. Abschnitt 2.2.2) gering, weil eine Produktlinie einer stabilen, etablierten Domäne angehört – dies ist eine der Voraussetzungen für den erfolgreichen Einstieg in die produktlinienbasierte Entwicklung von Systemen (vgl. Abschnitt 2.2.1).

- *Einarbeitung neuer volks- und betriebswirtschaftlicher Rahmenbedingungen*, wie z.B. Umstellung auf geänderte Steuergesetze oder auf fusionsbedingte Änderungen der Aufbau- und Ablauforganisation

- *Beseitigung von Fehlern*

Die Erweiterung einer mit FeatuRSEB entwickelten Produktlinie um neue Merkmale läuft in folgenden Schritten ab:

1. Neues Merkmal zum Merkmalmodell hinzufügen und die Beziehungen des neuen Merkmals zu den bestehenden Merkmalen definieren

2. Use-Cases des neuen Merkmals beschreiben und in das Use-Case-Modell der Produktlinie integrieren. Wenn die neuen Use-Cases bestehende Use-Cases erweitern, sind letztere um neue Erweiterungsstellen zu ergänzen.

3. Klassen für die neuen Use-Cases entwerfen, modellieren, implementieren und in das Objektmodell der Produktlinie integrieren. Überschneiden sich neue und bestehende Use-Cases im Objektmodell, so sind die betroffenen Klassen zu refaktorieren und ihre *trace*-Abhängigkeitsbeziehungen zum Use-Case-Modell zu aktualisieren.

Wie diese Tätigkeiten praktisch aussehen und welche Kritikpunkte dabei zu beobachten sind, erläutert der Rest des Abschnitts anhand eines Beispiels.

Ausbau von Produktlinien: Merkmalmodell

Die Produktlinie „Die Siedler von Catan“ soll um die Merkmale Goldfluß und Gründung erweitert werden. Das erste Merkmal ist der Ausbaustufe „Die Seefahrer“ entnommen, das zweite der Stufe „Städte & Ritter“. Abbildung 4.11 zeigt, wie die neuen Merkmale sich in das Merkmalmodell der Produktlinie einfügen: Goldfluß ist ein optionales Merkmal, wohingegen Gründung die alternativen Merkmale Siedlungsgründung und Stadtgründung gruppiert. Kein Merkmal hängt von einem der in Abbildung 4.3 auf Seite 46 dargestellten Merkmalen ab, so daß keine neuen Abhängigkeitsbeziehungen definiert werden müssen.

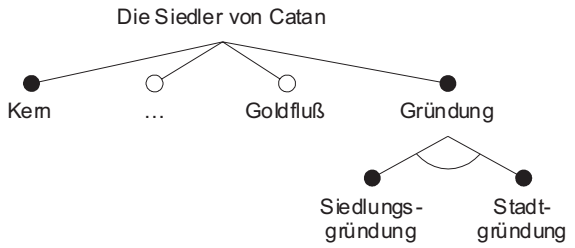

Abbildung 4.11: Wartung Merkmalmodell (Ergebnis)

... Use-Case-Modell

Sowohl Goldfluß als auch Gründung beeinflussen die Gründungsphase zu Beginn eines Spiels, genauer gesagt die Rückrunde der Gründungsphase. Im Use-Case-Modell der Produktlinie ist der Ablauf der Rückrunde durch einen Aktivitätsgraphen spezifiziert. Die Aktivitäten der einzelnen Aktivitätszustände im Graphen sind wiederum durch entsprechende Use-Cases beschrieben. Abbildung 4.12 zeigt, wie das Use-Case-Modell vor dem Ausbau der Produktlinie ausgesehen hat.

Die neuen Merkmale erweitern und verändern den Ablauf der Rückrunde (vgl. Abbildung 4.13). Goldfluß erweitert den Use-Case Rohstoffstartkapital vergeben (und damit die Aktivität des gleichnamigen Aktivitätszustands) um die im Use-Case Rohstoff Gold eintauschen beschriebene Durchführung: Wann immer ein Spieler als Startkapital den Rohstoff Gold erhält, muß er ihn gegen einen beliebigen anderen Rohstoff seiner Wahl umtauschen. Diese Erweiterung wurde durch

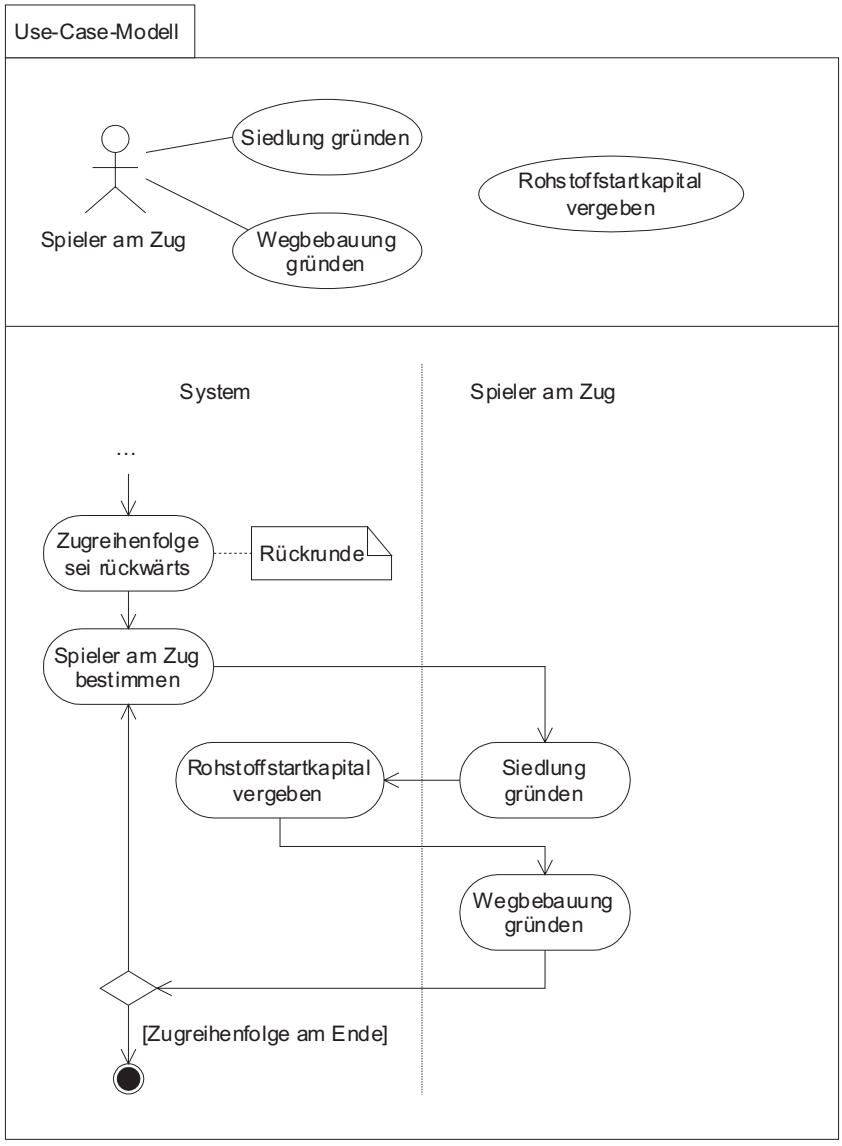

Abbildung 4.12: Wartung Use-Case-Modell (Ausgangssituation)

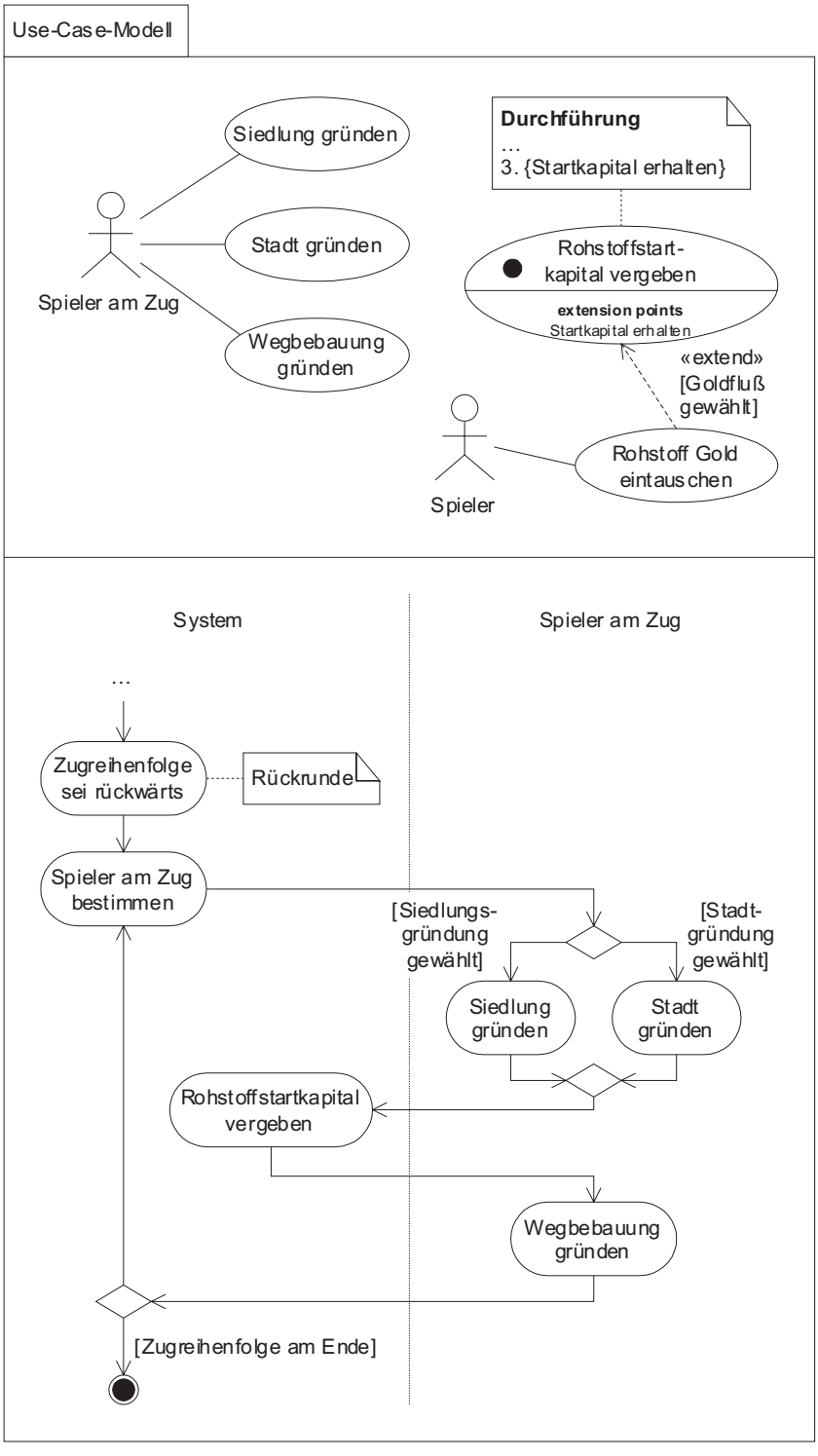

Abbildung 4.13: Wartung Use-Case-Modell (Ergebnis)

Hinzufügen der Erweiterungsstelle Startkapital erhalten zum Use-Case Rohstoffstartkapital vergeben in das bestehende Use-Case-Modell integriert.

Das Merkmal Gründung verändert die erste Aktivität des Spielers in der Rückrunde der Gründungsphase: Je nach dem, welches Merkmal der Kunde wählt, muß der Spieler in der Rückrunde als erstes eine Siedlung *oder* eine Stadt gründen. Dazu wurde in den Aktivitätsgraph ein Entscheidungsknoten eingefügt, der entweder zu dem bestehenden Aktivitätszustand Siedlung gründen oder zu dem neuen Zustand Stadt gründen verzweigt. Die Aktivität, die beim Erreichen des letztgenannten Zustands ausgeführt wird, beschreibt der neue Use-Case Stadt gründen.

... Objektmodell

Im Objektmodell der Produktlinie ist der Ablauf der Gründungsphase in der Klasse Gründungsphase modelliert (vgl. Abbildung 4.14 auf der folgenden Doppelseite). Hinsichtlich der Rückrunde definiert die Klasse die Operation führeRückrundenzugDurch(), die ein Spiel nach dem Start für jeden teilnehmenden Spieler aufruft. Die Operation führt nacheinander die drei Aktivitäten der Rückrunde aus, modelliert in den Klassen SiedlungGründen, RohstoffstartkapitalVergeben und WegbebauungGründen.

Die Veränderungen und Erweiterungen im Objektmodell durch die neuen Merkmale faßt Abbildung 4.15 zusammen. Das Merkmal Goldfluß, genauer gesagt die Durchführung des daraus hervorgegangenen Use-Case Rohstoff Gold eintauschen, kapselt die Klasse RohstoffGoldEintauschen in der Operation führeAus() und ergänzt damit das von der Oberklasse RohstoffstartkapitalVergeben geerbte Verhalten. Wählt der Kunde das Merkmal Goldfluß, so muß das Anwendungssystem in der Rückrunde die Klasse RohstoffGoldEintauschen instanziieren, andernfalls die Klasse RohstoffstartkapitalVergeben. Dazu wurde wieder das Entwurfsmuster Abstrakte Fabrik angewendet (vgl. Abschnitt 4.2.4). Folglich müssen Anwendungssysteme die Operation Fabrik:create1() gemäß dem Wunsch der Kunden spezialisieren.

Das Merkmal Gründung stellt die Rückrunde vor die Entscheidung, entweder eine Siedlung oder eine Stadt zu gründen. Im Objektmodell muß folglich die Operation führeRückrundenzugDurch() je nach gewähltem Merkmal derart parametrisiert werden können, daß sie entweder wie bisher SiedlungGründen::führeAus() oder neu StadtGründen::führeAus() aufruft. Die neue Klasse StadtGründen kapselt die Durchführung des Use-Case Stadt gründen. Um Überschneidungen zu vermeiden, werden solche Alternativen häufig durch das Entwurfsmuster Strategie [GHJV95, Seite 315ff] modelliert. Dabei wird der parametrisierbare Algorithmus wie folgt ausgelagert:

1. Eine abstrakte Klasse mit einer abstrakten Operation einführen, hier Gründungsstrategie und gründeGebäude()
2. Für jede Alternative eine konkrete Unterklasse definieren, die die abstrakte Operation entsprechend spezialisiert: Siedlungsgründung ruft in gründeGebäude() SiedlungGründen::führeAus() auf, Stadtgründung StadtGründen::führeAus().

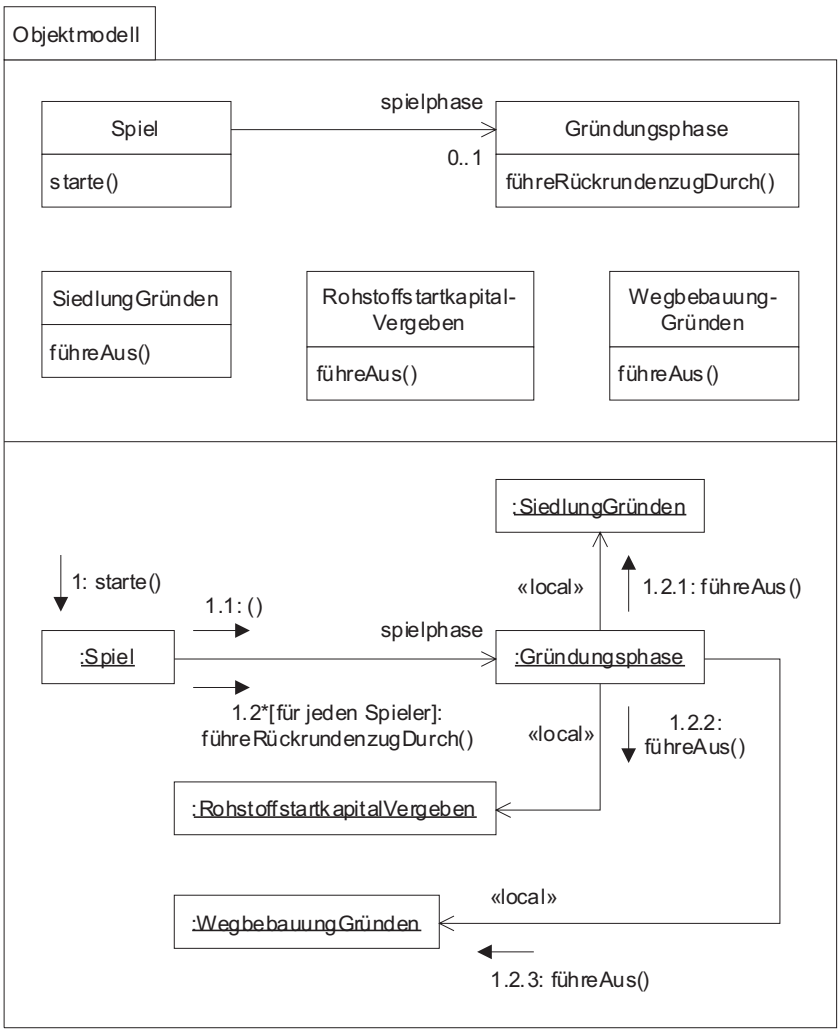

Abbildung 4.14: Wartung Objektmodell (Ausgangssituation)

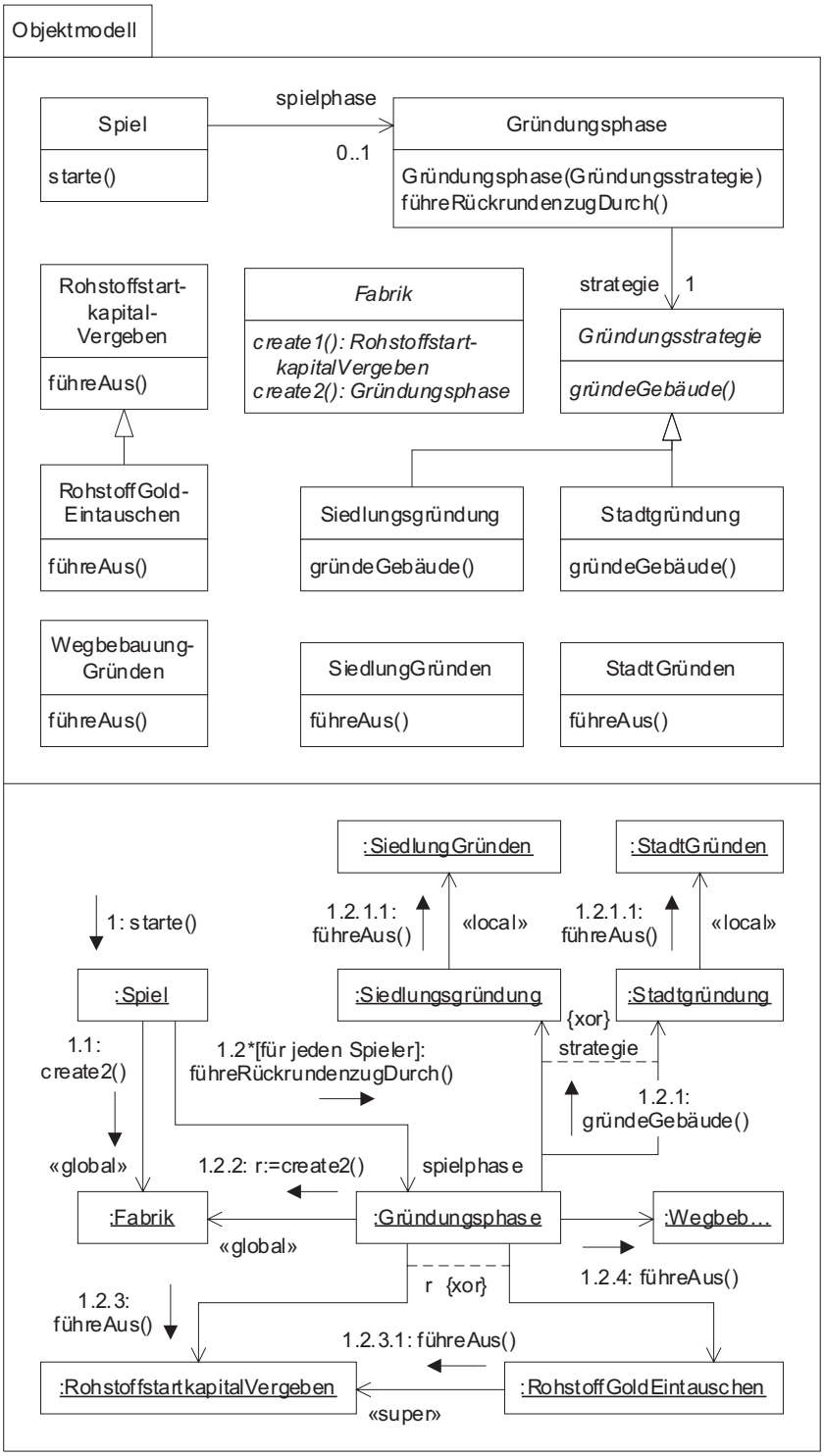

Abbildung 4.15: Wartung Objektmodell (Ergebnis)

3. Dem parametrisierbaren Objekt zur Laufzeit des Systems den gewünschten Algorithmus, eine der konkreten Unterklassen, mitteilen. Hier wird der Gründungsphase bei Instanziierung eine Gründungsstrategie übergeben. An diese Strategie delegiert dann die Operation führeRückrundenzugDurch() die Gründung einer Siedlung beziehungsweise Stadt.

Zum Schluß stellt sich noch die Frage, wer die Parametrisierung vornimmt. Da diese von der Auswahl einer der Alternativen durch die Kunden abhängt, muß die Parametrisierung im Anwendungssystem geschehen. Dazu wird die bereits für das Merkmal Goldfluß eingeführte Fabrik genutzt. Ein Spiel instanziiert die Gründungsphase nicht mehr selbst, sondern über den Umweg der Fabrik durch Aufruf von create2(). Anwendungssysteme müssen eine Unterklasse von Fabrik definieren, die create2() so überschreibt, daß die Operation eine Gründungsphase mit der vom Kunden gewünschten Strategie instanziiert und zurückliefert.

Bewertung

Der Ausbau einer mit FeatuRSEB entwickelten Produktlinie erfordert tiefe Eingriffe in das Use-Case-Modell und Objektmodell der Produktlinie, wie das umfangreiche, im Verhältnis zu anderen Ausbauten in der Fallstudie „Die Siedler von Catan“ jedoch fast triviale Beispiel demonstrierte. Die Art der Eingriffe hängt nicht davon ab, ob aus der Produktlinie Systeme entwickelt oder in Serie gefertigt werden sollen, genausowenig wie die Notwendigkeit, nach jedem Eingriff die *trace*-Abhängigkeitsbeziehungen zwischen den Modellen zu aktualisieren. Dies ist insbesondere bei den zahlreichen Beziehungen von Use-Cases zu Klassen eine fehleranfällige, da manuell durchzuführende Tätigkeit. Im Beispiel wurden die Aktualisierungen der Beziehungen der Übersichtlichkeit wegen nicht gezeigt.

Darüber hinaus veranschaulicht das Wartbarkeitsbeispiel einen Aspekt der Produktlinien-Entwicklung mit FeatuRSEB, der bisher noch nicht in dieser Deutlichkeit auftrat: Das Überschneidungsproblem, das zunächst auf das Objektmodell beschränkt schien, betrifft ebenfalls das Use-Case-Modell. Dort überschneiden sich die Merkmale aus dem Merkmalmodell in Use-Cases und Aktivitätsgraphen. So gehört der Use-Case Rohstoffstartkapitel erhalten mitsamt seiner Erweiterungsstelle zum Merkmal Kern, obwohl die Erweiterungsstelle aufgrund des neuen Merkmals Goldfluß entstand und nur von diesem genutzt wird. Eine zweite Überschneidung zeigt der Aktivitätsgraph, dessen Ablauf die Merkmale Kern, Siedlungsgründung und Stadtgründung bestimmen. Das Überschneidungsproblem beeinflußt im Use-Case-Modell die Komplexität und Wartbarkeit von Produktlinien ähnlich negativ wie im Objektmodell.

Die Wartbarkeit der mit FeatuRSEB entwickelten Produktlinien muß insgesamt mit unzureichend bewertet werden. Sie kann sowohl bezüglich Use-Case-Modell als auch Objektmodell durch eine andere Lösung des Überschneidungsproblems verbessert werden. Eine solche Lösung bietet der bereits bei der Bewertung der Komplexität angesprochene Hyperspace-Ansatz (vgl. Kapitel 5).

4.5.4 Skalierbarkeit der Methode

Im Software-Engineering werden Systeme in allen denkbaren Größen mit unterschiedlichen Organisationsstrukturen und Techniken entwickelt. Methoden zur Softwareentwicklung eignen sich in der Regel nicht für Systeme jeder Größenordnung, nicht für jede in Unternehmen vorzufindende Aufbau- und Ablauforganisation und nicht für jede Programmiertechnik. Statt dessen treffen Methoden Aussagen wie: „Hiermit können Teams bis zu 50 Personen objektorientierte Systeme mittlerer Größe entwickeln." Eine solche Aussage soll im Rahmen der Bewertung FeatuRSEBs als Skalierbarkeit einer Methode bezeichnet werden.

Bis wohin skaliert FeatuRSEB? Abgesehen von der Vorgabe einer an den Entwicklungsprozessen ausgerichteten Organisationsstruktur und der Fokussierung auf objektorientierte Systeme (vgl. Unterkapitel 4.1) finden sich in der Literatur keine Angaben zur Skalierbarkeit von FeatuRSEB, etwa in Form veröffentlichter Fallstudien. Anhand der zwei eingangs erwähnten Fallstudien soll trotzdem versucht werden, die Skalierbarkeit einzuschätzen. Dazu stellt Tabelle 4.1 einige Metriken der beiden Fallstudien gegenüber. Ein Fragezeichen bedeutet, daß die entsprechende Metrik in der Fallstudie nicht erhoben wurde. Die Berechnung der Anzahl möglicher Produkt-Konfigurationen geschah über die Formeln aus [ESBC01]. Grundlage für die Berechnung war das Merkmalmodell der jeweiligen Produktlinie: [BP00, Abbildung 1] beziehungsweise [Hal01, Abbildung 4.1]. Die Klassen und Anweisungen zählte das Werkzeug JavaNCSS [JSS].

Metrik	Bibliotheks-systeme	Die Siedler von Catan bis 5 und 6 Spieler
Projektteam	1 Entwickler	1 Entwickler
Entwicklungszeit [Personen-Monate]	0,5	2
Merkmale	9	6
Produkt-Konfigurationen	8	6
Use-Cases	20	24
UML-Diagramme	10	20
Klassen	50	122
Anweisungen (ohne Kommentare)	?	9.100

Tabelle 4.1: Skalierbarkeit von FeatuRSEB

Demnach kann mit FeatuRSEB eine einzelne Person objektorientierte Produktlinien bis zu einem Umfang von etwa 25 Use-Cases und 120 Klassen entwickeln. Diese Aussage ist natürlich vor dem Hintergrund der geringen Zahl an Fallstudien zu sehen und darf nicht überinterpretiert werden. Wenn man allerdings die obigen Bewertungen zu Komplexität und Wartbarkeit mit einbezieht, ist der Eindruck nicht von der Hand zu weisen, daß die Skalierbarkeit von FeatuRSEB bezogen auf die Größe der entwickelbaren Produktlinien begrenzt ist.

4.6 Zusammenfassung

Die FeatuRSEB-Methode eignet sich nur eingeschränkt zur Entwicklung von Produktlinien, da die dabei entstehenden Produktlinien zu komplex und wartungsaufwendig sind. Die Probleme verschärfen sich weiter, wenn aus solchen Produktlinien Systeme automatisiert in Serie gefertigt werden sollen.

Damit ist FeatuRSEB als Grundlage für das Erreichen der Zielsetzung dieser Arbeit zwar grundsätzlich geeignet, erfordert aber noch Verbesserungen. Dazu zählt die Integration des Hyperspace-Ansatzes, der die Modularisierung feingranularer Produktlinien-Komponenten erlaubt und so die Komplexität und Wartbarkeit der mit FeatuRSEB entwickelten Produktlinien entscheidend verbessert. Den Ansatz stellt Kapitel 5 vor und zeigt, wie sich feingranulare Komponenten implementieren lassen. Eine Ausdehnung des Hyperspace-Ansatzes auf die Modellierung feingranularer Komponenten folgt in Kapitel 6, bevor schließlich Kapitel 7 die verbesserte FeatuRSEB-Methode beschreibt.

Kapitel 5

Der Hyperspace-Ansatz

Die im vorangegangenen Kapitel erarbeiteten Probleme der FeatuRSEB-Methode können durch Nutzung des Hyperspace-Ansatzes bei der Entwicklung von Produktlinien gelöst werden. Deshalb wurde der Hyperspace-Ansatz in Abschnitt 2.3.4 als generative Technik für die Implementierung der Serienfertigung von Software-Systemen ausgewählt. Dieses Kapitel beschreibt zunächst den Hyperspace-Ansatz, bevor es in Unterkapitel 5.4 zeigt, wie der Ansatz die Probleme von FeatuRSEB löst. Die Illustrationen entstammen wieder der Fallstudie „Die Siedler von Catan“, vorgestellt in Kapitel 3.

5.1 Überblick

Software-Systeme werden bei ihrer Entwicklung in kleine, überschaubare Einheiten zerlegt, häufig auch Belange (engl.: *concerns*) genannt. Das hilft bei der Strukturierung komplexer Systeme und verbessert deren Verständlichkeit, weil ein Entwickler nicht ständig das gesamte System im Blick haben muß, sondern sich bei seiner Arbeit weitgehend getrennt auf einzelne Belange des Systems konzentrieren kann. In der Wartungsphase soll die Trennung von Systemen in Belange das Einarbeiten von Änderungsanforderungen erleichtern, sofern die Änderungen sich auf einen identifizierten Belang beschränken [Dij76].

Die Zerlegung von Systemen kann auf verschiedene Art und Weise geschehen, z.B. funktional, datenorientiert oder objektorientiert. Die Zerlegungsart ist jedoch nicht frei wählbar. Sie ist vielmehr durch die eingesetzte Entwicklungsmethode, Modellierungs- und Programmiersprache vorgegeben. Dabei verfolgen heutige Methoden und Sprachen keine einheitliche Zerlegungsart für alle Phasen und Ergebnisse des Entwicklungsprozesses. So werden Systeme in objektorientierten Methoden zunächst funktional in Use-Cases zerlegt und erst später objektorientiert in Klassen. Dies spiegelt die Denkansätze der Personen wider, die sich hauptsächlich für die bei der Zerlegung gefundenen Belange interessieren: Kunden denken in Funktionen, die Systeme erfüllen sollen; Entwickler denken in Objekten und ihrem Beitrag zur Verwirklichung der Funktionen.

Der Bruch in der Zerlegung von Systemen, sprich der Übergang von einer Zerlegungsart zur nächsten während der Entwicklung, bleibt nicht ohne Folgen. So

gelingt es objektorientierten Methoden selten, einen Use-Case auf ein oder mehrere isolierte Klassen abzubilden. Statt dessen überschneiden sich in einer Klasse oft mehrere Use-Cases. Jeder Use-Case fügt zur Klasse bestimmte Attribute, Assoziationen, Operationen oder sogar nur einzelne Anweisungen und Bedingungen hinzu. Dieser Effekt wurde bereits im Zusammenhang der FeatuRSEB-Methode in Abschnitt 4.2.4 erkannt und als Überschneidungsproblem bezeichnet. Das Überschneidungsproblem führt schon in der Entwicklung, spätestens aber in der Wartungsphase eines Systems zu Konsequenzen, die die oben erwähnten Ziele der Trennung von Belangen teilweise zunichte machen. Ein typischer Fall ist die Änderung eines Use-Case. Dann sind alle daraus hervorgegangenen (Teile von) Klassen entsprechend anzupassen. Diese Teile liegen aber nicht getrennt als Belang vor, sondern sind über mehrere Belange (Klassen) verstreut. Die Änderungen beschränken sich nicht auf einen einzigen Belang und sind deshalb schwerer durchzuführen [TOHS99].

Der Hyperspace-Ansatz löst das Überschneidungsproblem. Er definiert ein allgemeines, abstraktes Modell, wie Software-Systeme in allen Phasen des Entwicklungsprozesses *simultan* auf mehrere Arten zerlegt werden können. Damit unterscheidet sich der Ansatz von den bisherigen Methoden, die Systeme zwar auch auf mehrere Arten zerlegen, dies aber *nacheinander* tun statt simultan. Der Hyperspace-Ansatz schreibt dabei weder die Zerlegungsarten vor noch hängt er von bestimmten Methoden, Modellierungs- oder Programmiersprachen ab. Das bedeutet aber auch, daß vor dem praktischen Einsatz des Ansatzes dessen Konzepte und Begriffe auf die zu verwendenden Sprachen abgebildet werden müssen [OT01, HS].

Das folgende Unterkapitel 5.2 erläutert, welche Konzepte und Begriffe der Hyperspace-Ansatz definiert und wie sie abzubilden, sprich für Sprachen zu instanziieren sind. Darauf aufbauend beschreibt Unterkapitel 5.3 eine existierende Abbildung des Ansatzes auf die Programmiersprache Java, genannt Hyper/J [TO01]. Eine weitere, im Rahmen dieser Arbeit entwickelte Abbildung auf die UML folgt in Kapitel 6.

5.2 Konzepte und Begriffe

Der Hyperspace-Ansatz weist einen hohen Abstraktionsgrad auf, so daß eine rein theoretische Erläuterung der Konzepte nur schwer zu verstehen wäre. Deshalb greift das Unterkapitel erstens der Beschreibung von Hyper/J ein wenig vor und unterlegt zweitens die Erläuterungen mit einem fortlaufenden Beispiel. Das Beispiel zeigt, wie ein objektorientiert zerlegtes System simultan auf eine weitere Art zerlegt wird. Die Ausgangssituation des Systems ist in Abbildung 5.1 zu sehen.

Weiterhin wurde versucht, die Begriffe des Hyperspace-Ansatzes sinnvoll ins deutsche zu übersetzen. Dennoch wird bei der Einführung eines neuen Begriffs auch die englische Bezeichnung angegeben. So können Leser, die mit der Primärliteratur zum Hyperspace-Ansatz vertraut sind, die Übersetzungen leichter zuordnen. Zudem unterstützen die englischen Bezeichnungen das Verständnis der Syntax von Hyper/J.

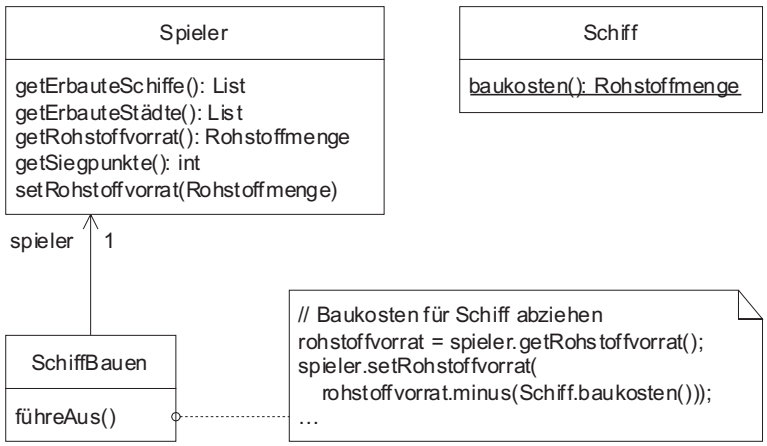

Abbildung 5.1: Objektorientiert zerlegtes System

5.2.1 Identifikation von Belangen: Hyperraum

Bevor ein Software-System mit dem Hyperspace-Ansatz beschrieben werden kann, ist zu klären, auf welche Arten das System zerlegt werden soll und welche Belange innerhalb der Zerlegungsarten von Interesse sind, sprich voneinander zu trennen sind. In der objektorientierten Zerlegung sind die zu trennenden Belange die Klassen, im obigen Beispiel Spieler, Schiff und SchiffBauen. Eine weitere Zerlegungsart, die speziell für Produktlinien relevant ist, ist die merkmalorientierte Zerlegung. Sie trennt Anforderungen und Komponenten von Produktlinien nach Merkmalen. Das Beispiel enthält zwei Merkmale aus der Produktlinie „Die Siedler von Catan“: Kern und Schiffe (vgl. Abschnitt 4.2.2 für die Definition von Merkmal und das Merkmalmodell der Produktlinie). Wenn alle Belange identifiziert sind, kann der Hyperraum für das System aufgestellt werden (vgl. Abbildung 5.2).

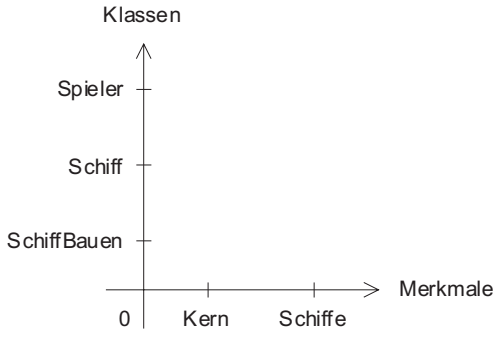

Abbildung 5.2: Beispiel Hyperraum

Ein Hyperraum (engl.: *hyperspace*) beschreibt ein Software-System als mehrdimensionalen Raum. Jede Dimension des Raums stellt eine Art dar, auf die das System zu zerlegen ist. Eine Dimension ist in diskrete (nicht kontinuierliche) Einheiten unterteilt. Jede Einheit repräsentiert einen identifizierten Belange innerhalb der Zerlegungsart der Dimension. Die Anordnung der Einheiten in der Dimension ist beliebig. Der Nullpunkt einer Dimension heißt 0-Belang (sprich: Null-Belang, ohne Belang; engl.: *none concern*). Die Abbildung zeigt den Hyperraum für das Beispiel-System mit seinen zwei Dimensionen (Achsen), auf denen die erwähnten Belange, Klassen und Merkmale, aufgetragen sind.

Eine Dimension verwirklicht also die Trennung von Belangen bezüglich einer Zerlegungsart und entspricht damit dem Stand heutiger Entwicklungsmethoden. Alle Dimensionen zusammen verwirklichen die Trennung von Belangen bezüglich der verschiedenen, auf ein System simultan anwendbaren Zerlegungsarten.

Nachdem der Hyperraum aufgestellt ist, folgt die Zerlegung des Systems. Die geschieht simultan auf alle gewählten Arten. Das Ergebnis ist eine Menge von Elementen (engl.: *units*). Welche Elemente im einzelnen möglich sind, definieren konkrete Abbildungen des Hyperspace-Ansatzes auf Modellierungs- und Programmiersprachen. Elemente in Hyper/J sind z.B. Pakete, Klassen, Felder und Methoden. Bei den Elementen wird noch unterschieden zwischen atomaren und zusammengesetzten Elementen (engl.: *primitive and compound units*). Letztere enthalten sowohl atomare als auch zusammengesetzte Elemente. In Hyper/J sind z.B. Felder und Methoden atomar. Aus diesen setzen sich Klassen zusammen, die wiederum in Paketen enthalten sind.

Um zu dokumentieren, welches Element für welchen Belang von Interesse ist, werden die Elemente den Punkten im Hyperraum zugeordnet. Die Koordinaten der Punkte geben Auskunft über die Zuordnung der Elemente zu den Belangen. Ist ein Element in einer Dimension für keinen Belang von Interesse (sozusagen belanglos), befindet sich das Element im 0-Belang der Dimension. Jedes Element muß genau einem Punkt im Hyperraum zugeordnet sein. Demnach können einem Punkt mehrere Elemente zugeordnet werden, er kann aber auch ohne Zuordnung bleiben.

Wird ein zusammengesetztes Element einem Punkt im Hyperraum zugeordnet, so sind auch alle darin enthaltenen (atomaren und zusammengesetzten) Elemente implizit dem Punkt zugeordnet, es sei denn, ein enthaltenes Element wird noch einmal explizit einem anderen Punkt zugeordnet. Erfolgt für ein Element keine Zuordnung, liegt dieses automatisch im Nullpunkt des Hyperraums. Diese Regeln vereinfachen in der Praxis die Zuordnung, wie die Fortsetzung des Beispiels in Abbildung 5.3 zeigt. Dort wurden im Hyperraum den Punkten p_1 bis p_4 und damit den entsprechenden Belangen die aufgeführten Elemente zugeordnet. Die Klassen Schiff und SchiffBauen gehören zum Merkmal Schiffe. Da keine ihrer Operationen explizit einem anderen Merkmal zugeordnet sind, gehören sie implizit ebenfalls zum Merkmal Schiffe. Anders bei der Klasse Spieler. Ihre Operation getErbauteSchiffe() gehört zum Merkmal Schiffe und nicht zum Kern, so daß die Operation explizit umgeordnet wurde.

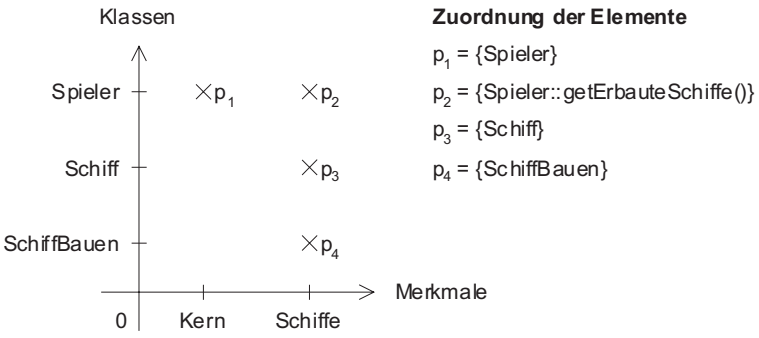

Abbildung 5.3: Beispiel Hyperraum mit zugeordneten Elementen

5.2.2 Kapselung von Belangen: Hyperebenen

Ein Hyperraum definiert Dimensionen und Belange und ordnet die Elemente eines Systems den Belangen zu. Was fehlt, ist eine Möglichkeit, Belange (d.h. ihre Elemente) in Module zu kapseln, damit Entwickler die Belange getrennt voneinander betrachten und bearbeiten können. Dadurch wird das Überschneidungsproblem vermieden, und die eingangs angeführten Ziele der Trennung von Belangen treten wieder ein. Die Kapselung von Belangen übernehmen Hyperebenen.

Eine Hyperebene[1] (engl.: *hyperslice*) kapselt die Elemente aus einem oder mehreren Belangen einer oder mehrerer Dimensionen. In Hyper/J wird eine Hyperebene als Paket implementiert, wie Abbildung 5.4 zeigt. Das Hyperebenen-Paket catan::kern kapselt die Elemente des Merkmals Kern, catan::schiffe die von Schiffe.

Eine wichtige Eigenschaft von Hyperebenen ist ihre deklarative Vollständigkeit (engl.: *declarative completeness*). Was verbirgt sich dahinter? Die Elemente eines Systems stehen in Beziehung zueinander: Eine Klasse erbt von einer anderen, eine Operation ruft eine andere auf usw. In der Ausgangssituation des Beispiels in Abbildung 5.1 auf Seite 75 ruft SchiffBauen::führeAus() Spieler::getRohstoffvorrat() und setRohstoffvorrat() sowie Schiff::baukosten() auf. Solche Beziehungen zwischen Elementen sind unproblematisch, solange voneinander abhängige Elemente in derselben Hyperebene gekapselt sind, wie dies bei SchiffBauen::führeAus() und Schiff::baukosten() der Fall ist. Beide Operationen sind über ihre Klasse dem Merkmal Schiffe zugeordnet und in der Hyperebene catan::schiffe implementiert. Anders verhält sich die Beziehung zwischen SchiffBauen::führeAus() und Spieler::getRohstoffvorrat() beziehungsweise setRohstoffvorrat(). Die erste Operation ist dem Merkmal Schiffe zugeordnet und dementsprechend in der Hyperebene catan::schiffe implementiert. Die anderen zwei Operationen hingegen gehören zum Kern und damit zu catan::kern.

[1] Der Begriff Ebene wird hier nicht in seiner eigentlichen Bedeutung gebraucht, da eine Hyperebene durchaus den gesamten Hyperraum (alle Belange) umfassen kann.

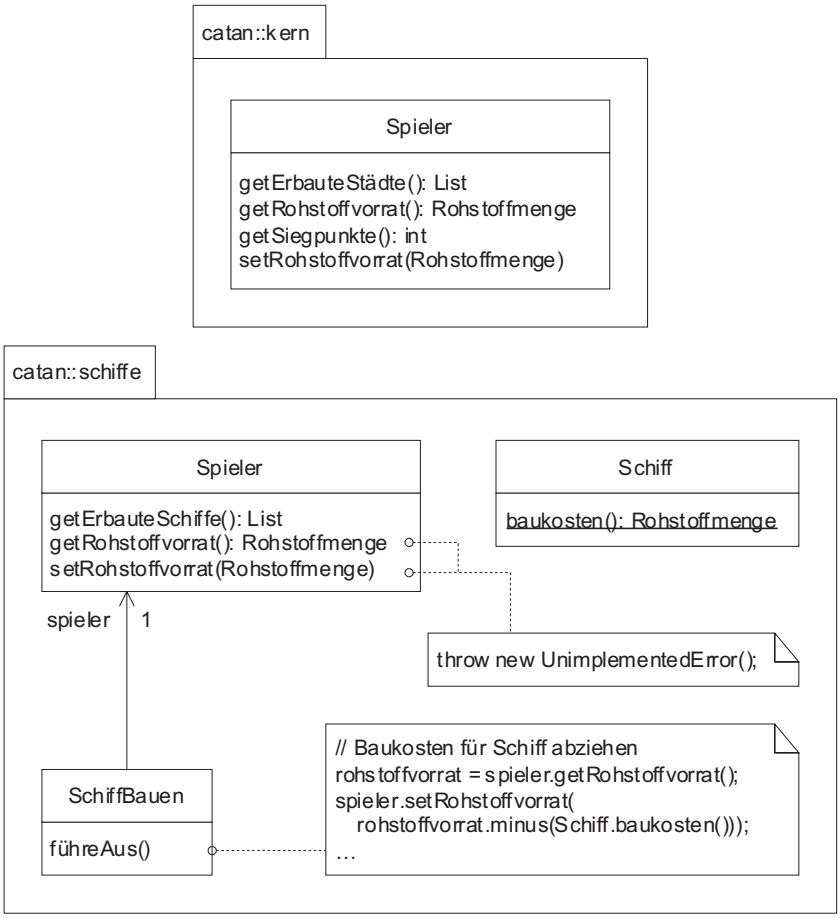

Abbildung 5.4: Beispiel Hyperebenen

Ohne deklarative Vollständigkeit würden die beiden Hyperebenen jetzt direkt voneinander abhängen. Dies ist im Hyperspace-Ansatz unerwünscht, weil solche Abhängigkeiten die Freiheit, Hyperebenen später beliebig zu Systemen integrieren zu können, einschränken (vgl. nächster Abschnitt 5.2.3). *Mit* deklarativer Vollständigkeit importiert beziehungsweise referenziert eine Hyperebene keine aus anderen Ebenen benötigten Elemente. Statt dessen deklariert sie benötigte Elemente genau bis zu dem Umfang, in dem sie sie nutzt. Details benötigter Elemente, die nicht von Interesse sind, werden ausgespart. Angewandt auf das Beispiel sehen die Hyperebenen für die Merkmale Kern und Schiffe so aus, wie sie bereits Abbildung 5.4 vorweggenommen hat. Die Hyperebene catan::schiffe deklariert die zwei benötigten Operationen Spieler::getRohstoffvorrat() und setRohstoffvorrat(), implementiert sie aber nicht. In Hyper/J wird dies technisch durch Auslösen der Ausnahme UnimplementedError realisiert. Damit hängt die Hyperebene catan::schiffe nicht mehr direkt von catan::kern ab und gilt als vollständig bezüglich ihrer Deklaration. Sie kann von einem Compiler übersetzt werden, ist aber alleine nicht lauffähig, da sie nicht implementierte Operationen definiert.

Hyperebenen lösen das Überschneidungsproblem. In der Ausgangssituation des Beispiels überschneiden sich die Merkmale Kern und Schiffe in der Klasse Spieler. Kern steuert vier Operationen zur Klasse bei, Schiffe eine. Durch die – zusätzlich zur objektorientierten Zerlegung – simultan durchgeführte merkmalorientierte Zerlegung des Beispiels entstanden die zwei Hyperebenen catan::kern und catan::schiffe. Dort tritt die Überschneidung nicht mehr auf, weil die Operationen von Spieler auf die jeweiligen Hyperebenen verteilt wurden.

5.2.3 Integration von Belangen: Hypermodule

Hyperebenen ermöglichen die getrennte Betrachtung und Bearbeitung von Belangen (d.h. ihrer Elemente). Damit aus den solchermaßen getrennten Belangen ein lauffähiges System wird, müssen die Elemente der Belange wieder zusammengefügt, sprich miteinander integriert werden. Welche Belange wie zu integrieren sind, beschreiben Hypermodule.

Ein Hypermodul (engl.: *hypermodule*) faßt eine Menge von Belangen zusammen und definiert Integrationsbeziehungen (engl.: *integration relationships*) für die Elemente der Belange. Im einfachsten Fall stehen alle Elemente orthogonal zueinander, so daß keine Integration notwendig ist; die Angabe von Integrationsbeziehungen entfällt.

Meist stimmen jedoch einige Elemente semantisch überein (unter anderem durch die Forderung nach deklarativer Vollständigkeit), wie im Beispiel die Klasse Spieler, die sowohl im Merkmal Kern als auch Schiffe vorhanden ist. Dann muß eine Integrationsbeziehung definiert werden. Sie beinhaltet Angaben darüber: (a) welche Elemente übereinstimmen (engl.: *correspondence*) und (b) wie sie zu integrieren sind. Ein Beispiel für eine Integrationsbeziehung in Hyper/J ist das Binden einer deklarierten, aber nicht implementierten Operation an eine konkrete Implementierung: catan::schiffe::Spieler::getRohstoffvorrat() an catan::kern::Spieler::getRohstoffvorrat() und ebenso setRohstoffvorrat(). Je nach Abbildung des Hyperspace-Ansatzes auf Modellierungs- und Programmiersprachen sind beliebig viele andere Arten von Integrationsbeziehungen denkbar, weshalb der Hyperspace-Ansatz an diesem Punkt keine weitergehenden Festlegungen mehr trifft.

Hypermodule sind ebenso wie Hyperebenen deklarativ vollständig, d.h. sie sind unabhängig von anderen Hypermodulen und -ebenen. Ein Hypermodul muß auch nicht immer ein lauffähiges System darstellen. Hyper/J erzwingt die Bindung nicht implementierter Operationen im Rahmen der Integration eines Hypermoduls nicht.

5.2.4 Instanziierung des Hyperspace-Ansatzes

Hier noch einmal eine Zusammenstellung, welche Konzepte und Begriffe des Hyperspace-Ansatzes auf Modellierungs- und Programmiersprachen abzubilden sind, damit der Hyperspace-Ansatz die Entwicklung von Systemen mit diesen Sprachen unterstützen kann:

- *Elemente.* Welche Elemente gibt es? Welche davon sind atomar, welche zusammengesetzt?
- *Hyperraum.* Wie wird der Hyperraum für ein System beschrieben? Wie werden die Elemente des Systems den identifizierten Belangen in den Dimensionen zugeordnet?
- *Hyperebenen.* Wie werden Hyperebenen gekapselt und beschrieben? Wie deklariert eine Hyperebene die aus anderen Ebenen benötigten Elemente?
- *Hypermodule.* Wie werden Hypermodule und Integrationsbeziehungen beschrieben?
- *Integrationsbeziehungen.* Welche Integrationsbeziehungen sind sinnvoll?
- *Werkzeuge.* Wie sieht das Werkzeug aus, das die durch Hypermodule beschriebenen Integrationen durchführt?

Das nächste Unterkapitel beantwortet die Fragen für die Programmiersprache Java.

5.3 Hyper/J

Hyper/J bildet den Hyperspace-Ansatz auf Java ab. Die folgende Beschreibung beschränkt sich auf die für diese Arbeit wichtigen Eigenschaften von Hyper/J. Sie nimmt dort zulässige Vereinfachungen vor, wo es angebracht erscheint und das Verständnis der Arbeit fördert, und sie übergeht eine Reihe von Limitierungen der aktuellen Version. Wer an den Details interessiert ist, sei auf das Handbuch [TO01] verwiesen.

5.3.1 Elemente

Elemente in Hyper/J sind Pakete, Schnittstellen, Klassen, Felder, Klasseninitialisierungen, Konstruktoren und Methoden. Alle Elemente sind benannt[2] und innerhalb ihres Namensraums (Paket, Schnittstelle oder Klasse) eindeutig referenzierbar. Darüber hinaus ist für jedes Element ein Typ definiert: package, interface, class, field sowie operation für Klasseninitialisierungen, Konstruktoren und Methoden.

Als zusammengesetzte Elemente definiert Hyper/J Pakete, Schnittstellen und Klassen. Alle anderen Elemente sind atomar. Darüber hinaus könnten Methoden als aus einzelnen Anweisungen zusammengesetzte Elemente angesehen werden. Dieser Schritt wurde (bisher) bewußt vermieden, da unklar ist, ob der damit einhergehende Anstieg an Komplexität den gewonnenen Nutzen rechtfertigt [OT98].

[2] Klasseninitialisierungen erhalten den Pseudonamen <clinit> (von engl.: *class initializer*).

5.3.2 Hyperraum

Ein Hyperraum in Hyper/J wird mit seinen Dimensionen, Belangen und der Zuordnung der Elemente zu den Belangen in folgender EBNF-Syntax[3] beschrieben:

$$
\begin{aligned}
\mathit{hyperspace} &\rightarrow \text{'hyperspace'}\ \mathit{hyperspaceName}\ (\ \mathit{mapping}\ )^* \\
\mathit{mapping} &\rightarrow \mathit{unitType}\ \mathit{unitName}\ \text{':'}\ \mathit{concern} \\
\mathit{unitType} &\rightarrow \text{'package'} \mid \text{'interface'} \mid \text{'class'} \mid \text{'field'} \mid \text{'operation'} \\
\mathit{concern} &\rightarrow \mathit{dimensionName}\ \text{'.'}\ (\ \mathit{concernName} \mid \text{'None'}\ )
\end{aligned}
$$

Der Name eines Elements ist vollständig anzugeben, d.h. einschließlich des Namensraums, in dem das Element liegt. Bei Konstruktoren und Methoden ist zusätzlich ihre Signatur anzugeben. None bezeichnet den 0-Belang einer Dimension. Demnach kann der Hyperraum mit Elementzuordnungen aus Abbildung 5.3 auf Seite 77 in Hyper/J wie folgt definiert werden:

```
hyperspace Die_Siedler_von_Catan
    class Spieler : Klasse.Spieler
    class Spieler : Merkmal.Kern
    operation Spieler::getErbauteSchiffe() : Merkmal.Schiffe
    class Schiff : Klasse.Schiff
    class Schiff : Merkmal.Schiffe
    class SchiffBauen : Klasse.SchiffBauen
    class SchiffBauen : Merkmal.Schiffe
```

5.3.3 Hyperebenen

Hyperebenen in Hyper/J sind gewöhnliche Pakete. Benötigt eine Hyperebene ein Element aus einer anderen Ebene, so deklariert sie das Element nur bis zu dem Umfang, in dem sie es nutzt beziehungsweise es die Syntax von Java erforderlich macht, so daß die Hyperebene von einem Compiler übersetzt werden kann. Benötigte Schnittstellen oder Klassen werden als leere Schnittstellen beziehungsweise Klassen deklariert. Benötigte Felder werden mit ihrem Typ deklariert und bleiben ohne Initialisierung. Klasseninitialisierungen können in Java nicht explizit aufgerufen werden. Sie können also von keiner Hyperebene benötigt werden; eine Deklaration entfällt. Benötigte Methoden einer Schnittstelle werden wie gewohnt abstrakt deklariert. Benötigte (aufgerufene) Konstruktoren und Methoden einer Klasse werden pseudo-implementiert. Sie bestehen aus der einzigen Anweisung throw new UnimplementedError().

[3] | = oder, ? = optional, + = mindestens einmal, * = beliebig oft oder nie

5.3.4 Hypermodule

Hypermodule in Hyper/J werden mit den zu integrierenden Belangen und, sofern notwendig, speziellen Integrationsbeziehungen in folgender EBNF-Syntax beschrieben:

$$
\begin{array}{rcl}
\mathit{hypermodule} & \rightarrow & \text{'hypermodule'}\ \mathit{hypermoduleName} \\
 & & \quad \mathit{hyperslices}\ \mathit{relationships}^{?} \\
 & & \text{'end hypermodule;'} \\
\mathit{hyperslices} & \rightarrow & \text{'hyperslices:'}\ \mathit{concern}\ (\ \text{','}\ \mathit{concern}\ )^{*} \\
\mathit{relationships} & \rightarrow & \text{'relationships:'}\ \mathit{relationship}\ (\ \mathit{relationship}\ )^{*} \\
\mathit{concern} & \rightarrow & \mathit{dimensionName}\ \text{'.'}\ (\ \mathit{concernName} \mid \text{'None'}\ ) \\
\mathit{relationship} & \rightarrow & (\ \mathit{mergeRelationship} \mid \mathit{orderRelationship} \mid \\
 & & \mathit{summaryFunctionRelationship} \mid \\
 & & \mathit{overrideRelationship}\ )\ \text{';'}
\end{array}
$$

Die wichtigsten Integrationsbeziehungen in Hyper/J sind: Verschmelzen, Ordnen, Summieren und Ersetzen. Ihre Semantik erläutern die folgenden Abschnitte zusammen mit Beispielen für Hypermodul-Spezifikationen. Ergebnis einer solchen Spezifikation ist immer ein neues Paket mit dem Namen des Hypermoduls. Dieses Paket enthält die miteinander integrierten Elemente.

5.3.5 Integrationsbeziehung Verschmelzen

Die Integrationsbeziehung Verschmelzen (engl.: *merge relationship*) verschmilzt zwei oder mehr Elemente gleichen Typs miteinander. Handelt es sich um zusammengesetzte Elemente, verschmelzen darin enthaltene Elemente gleichen Namens und Typs ebenfalls. Die Beziehung wird in folgender EBNF-Syntax im Hypermodul beschrieben:

$$
\begin{array}{rcl}
\mathit{mergeRelationship} & \rightarrow & \text{'merge'}\ \mathit{unitType}\ \mathit{unitName}\ (\ \text{','}\ \mathit{unitName}\ )^{+} \\
\mathit{unitType} & \rightarrow & \text{'package'} \mid \text{'interface'} \mid \text{'class'} \mid \text{'field'} \mid \text{'operation'}
\end{array}
$$

Abbildung 5.5 zeigt links die Hyperebenen der Merkmale Kern, Schiffe und Fortschrittskarten. Rechts oben ist die Hypermodul-Spezifikation Beispiel angegeben. Sie integriert die drei Merkmale und verschmilzt dabei die in den Hyperebenen jeweils vorhandene Variation der Klasse Spieler. Das Ergebnis ist ein neues Paket Beispiel, das die Klasse Spieler enthält. In der Klasse sind alle Felder und Methoden aus den Variationen vereinigt. Gleichnamige Felder und Methoden wurden zu einem Feld beziehungsweise einer Methode verschmolzen. Im Beispiel trifft dies auf die Methoden getRohstoffvorrat(), getSiegpunkte() und initialisiere() zu.

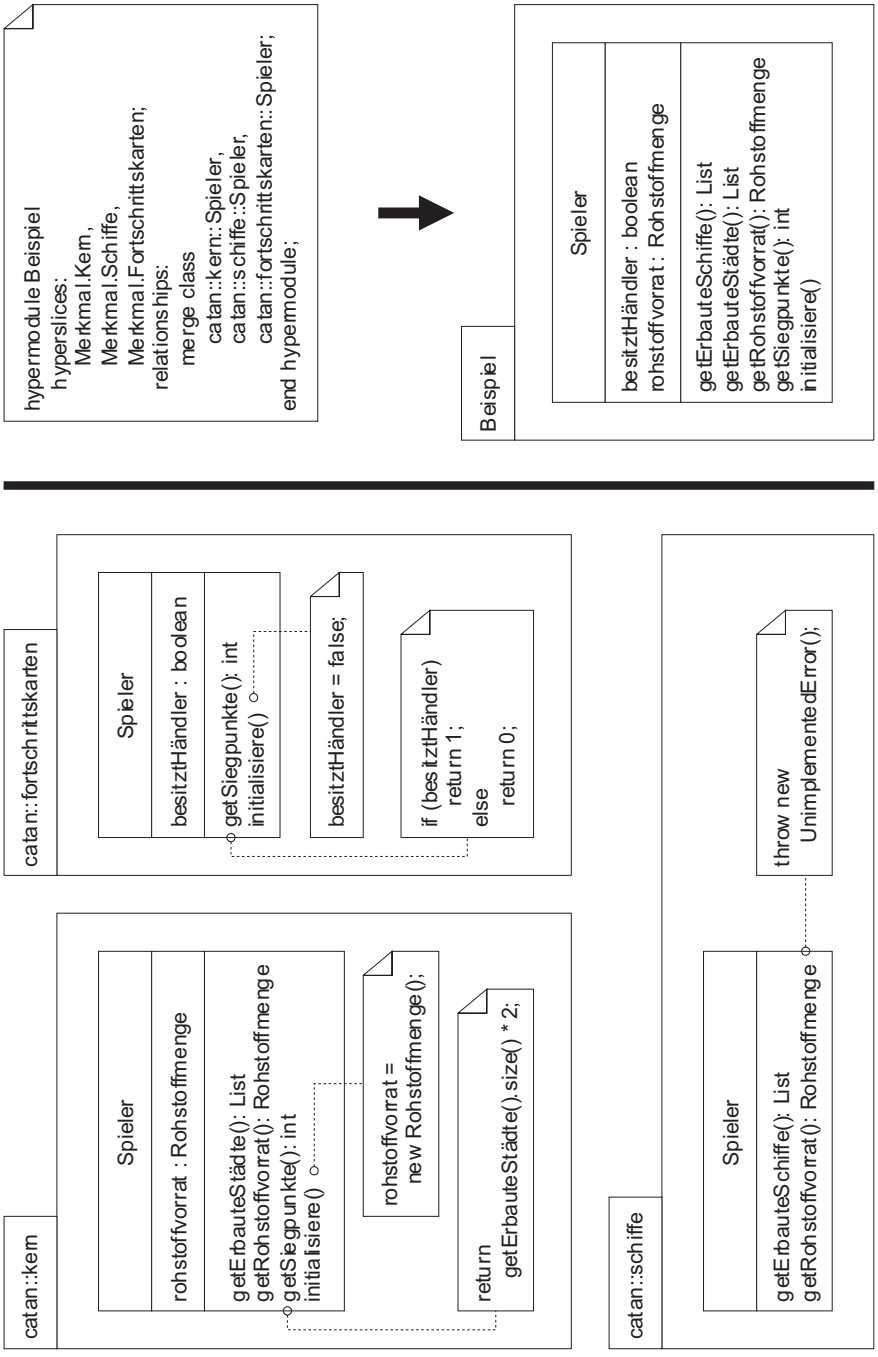

Abbildung 5.5: Beispiel Integrationsbeziehung Verschmelzen

Beim Verschmelzen von Methoden sind einige Besonderheiten zu beobachten. So führt die Methode Beispiel::Spieler::initialisiere() beim Aufruf nacheinander die Implementierungen aller in ihr verschmolzenen Variationen von initialisiere() aus. Der Quellcode von initialisiere() setzt sich im Hypermodul also aus zwei Anweisungen zusammen:

```
rohstoffvorrat = new Rohstoffmenge();    // Merkmal Kern
besitztHändler = false;    // Merkmal Fortschrittskarten
```

Die Reihenfolge, in der Implementierungen verschmelzen, ist undefiniert, d.h. die beiden Anweisungen könnten auch in der umgekehrten Reihenfolge zur Ausführung kommen. Wenn eine bestimmte Reihenfolge einzuhalten ist, muß im Hypermodul zusätzlich eine Integrationsbeziehung Ordnen angegeben werden (vgl. nächster Abschnitt 5.3.6).

Liefern verschmolzene Methoden einen Wert zurück, so hängt dieser ebenfalls von der Reihenfolge ab. Beispiel::Spieler::getSiegpunkte() gibt entweder das Ergebnis von getErbauteStädte().size() * 2 (Merkmal Kern) oder 1 beziehungsweise 0 (Merkmal Fortschrittskarten) zurück. Wenn statt dessen aus den Einzelergebnissen ein Gesamtergebnis zu bilden ist, muß im Hypermodul zusätzlich eine Integrationsbeziehung Summieren angegeben werden (vgl. Abschnitt 5.3.7).

Ist eine der verschmolzenen Methoden nur pseudo-implementiert, wie z.B. catan::schiffe::Spieler::getRohstoffvorrat(), übernimmt das Hypermodul die Pseudo-Implementierung nicht. Statt dessen gilt die nicht implementierte Methode mit dem Verschmelzen als gebunden. Die Methode Beispiel::Spieler::getRohstoffvorrat() enthält also lediglich die Implementierung aus dem Merkmal Kern.

5.3.6 Integrationsbeziehung Ordnen

Die Integrationsbeziehung Ordnen (engl.: *order relationship*) gibt an, in welcher Reihenfolge die Implementierungen von Methoden verschmolzen werden. Die Beziehung wird in folgender EBNF-Syntax im Hypermodul beschrieben:

$$
\begin{aligned}
orderRelationship &\rightarrow \text{'order'}\ unitType\ unitName\ (\ \text{'before'} \mid \text{'after'}\ ) \\
&\qquad unitName\ (\ \text{','}\ unitName\ )^{+} \\
unitType &\rightarrow \text{'operation'}
\end{aligned}
$$

Ergänzt man beispielsweise die Integrationsbeziehungen des Hypermoduls Beispiel aus Abschnitt 5.3.5 um

```
order operation catan::kern::Spieler::initialisiere() before
   catan::fortschrittskarten::Spieler::initialisiere();
```

so ist garantiert, daß Spieler::initialisiere() im Hypermodul die Anweisungen des Merkmals Kern immer vor denjenigen des Merkmals Fortschrittskarten ausführt.

5.3.7 Integrationsbeziehung Summieren

Die Integrationsbeziehung Summieren (engl.: *summary function relationship*) gibt an, wie aus den Rückgabewerten der in einer Methode verschmolzenen Implementierungen ein gemeinsamer Wert zu bilden ist, den Aufrufer der Methode erhalten. Die Beziehung wird in folgender EBNF-Syntax im Hypermodul beschrieben:

$$
\begin{aligned}
summaryFunctionRelationship \quad &\rightarrow \quad \text{'set summary function for'} \\
&\qquad\quad unitType\ unitName \\
&\qquad\quad \text{'to'}\ summaryFunction \\
unitType \quad &\rightarrow \quad \text{'operation'}
\end{aligned}
$$

Die Summenfunktion muß eine statische Methode sein. Sie bekommt ein Array mit einzelnen Rückgabewerten übergeben, bildet daraus einen einheitlichen Wert und gibt diesen zurück. Hyper/J beinhaltet in der Klasse SummaryFunctions bereits einige allgemeine Summenfunktionen wie intSum(int[]), booleanAnd(boolean[]) und booleanOr(boolean[]). Die erste Funktion summiert die int-Werte, die zwei anderen verknüpfen die boolean-Werte UND beziehungsweise ODER.

Das Hypermodul Beispiel aus Abschnitt 5.3.5 ergänzt um die Integrationsbeziehung

```
set summary function for operation Spieler::getSiegpunkte()
   to SummaryFunctions::intSum(int[]);
```

führt zu folgender Implementierung der Methode Spieler::getSiegpunkte() im Hypermodul:

```
int[] values = new int[2];
values[0] = getErbauteStädte().size() * 2;    // Merkmal Kern
if (besitztHändler) values[1] = 1;    // Merkmal Fortschrittskarten
else values[1] = 0;
return SummaryFunctions.intSum(values);
```

Die Methode liefert jetzt die Summe der Einzelergebnisse zurück und nicht wie zuvor nur ein Ergebnis, das zudem noch von der zufälligen Reihenfolge der Implementierungen abhing.

5.3.8 Integrationsbeziehung Ersetzen

Die Integrationsbeziehung Ersetzen (engl.: *override relationship*) ersetzt ein Element vollständig durch ein anderes Element gleichen Typs. Die Beziehung wird in folgender EBNF-Syntax im Hypermodul beschrieben:

$$
\begin{aligned}
overrideRelationship &\rightarrow \text{'override'}\ unitType\ unitName\ \text{'with'} \\
&\qquad\qquad unitName \\
unitType &\rightarrow \text{'operation'}
\end{aligned}
$$

Ergänzt man beispielsweise die Integrationsbeziehungen des Hypermoduls Beispiel aus Abschnitt 5.3.5 um

```
override operation catan::fortschrittskarten::Spieler::getSiegpunkte()
  with catan::kern::Spieler::getSiegpunkte();
```

so führt Spieler::getSiegpunkte() im Hypermodul nur die Anweisungen des Merkmals Kern aus und gibt das Ergebnis von getErbauteStädte().size() * 2 zurück.

Der vermehrte Einsatz der Integrationsbeziehung Ersetzen birgt in der Praxis einige Gefahren. Diese diskutiert Abschnitt 6.9.3 im Rahmen der Abbildung des Hyperspace-Ansatzes auf die UML. Die dort getroffenen Aussagen sind auf Hyper/J übertragbar.

5.3.9 Werkzeuge

Bei Hyper/J wird ein Werkzeug mitgeliefert, das Hypermodul-Spezifikationen verarbeitet und die Integration durchführt. Dazu benötigt das Werkzeug zusätzlich eine Hyperraum-Spezifikation und die Hyperebenen in Form übersetzter Java-Klassen (*class*-Dateien). Als Ausgabe erzeugt das Werkzeug ein Hypermodul-Paket, das den Bytecode der integrierten Klassen enthält. Der Bytecode ist konform zum Java-Standard, kann also mit jeder virtuellen Maschine ausgeführt werden.

Um die Entwicklung mit Hyper/J zu erleichtern, ist eine Einbettung des Werkzeugs in integrierte Entwicklungsumgebungen erforderlich. Weiterhin fehlt noch jegliche Art von Unterstützung für die Fehlersuche in Hypermodulen [Hal01, Unterkapitel 7.3].

5.4 Bewertung

Die Bewertung des Hyperspace-Ansatzes ist nicht allgemein gehalten. Statt dessen untersucht sie, inwiefern der Hyperspace-Ansatz einen Beitrag für das Erreichen der Zielsetzung dieser Arbeit erbringt. Zielsetzung ist die Serienfertigung schlanker Systeme auf Basis von Produktlinien. Grundlage hierfür ist die FeatuRSEB-Methode, deren Nutzung allerdings zu Problemen führt. FeatuRSEB zerlegt Produktlinien erst funktional in Merkmale und Use-Cases, danach objektorientiert in Klassen. Deswegen überschneiden sich Merkmale im Use-Case-Modell und Use-Cases im Objektmodell. Die Folge sind komplexe und wartungsaufwendige Produktlinien (vgl. Unterkapitel 4.5).

Der Schlüssel zur Lösung des Überschneidungsproblems in FeatuRSEB ist die durch den Hyperspace-Ansatz mögliche simultane Zerlegung von Systemen auf mehrere Arten. Übertragen auf Produktlinien erfolgt die funktionale Zerlegung in Merkmale jetzt nicht mehr als Vorstufe der Zerlegung in Use-Cases und Klassen, sondern simultan dazu. Es entsteht ein Hyperraum mit zwei Dimensionen. Die Belange der ersten Dimension sind die Use-Cases und Klassen; die zweite Dimension enthält die Merkmale (vgl. Abschnitt 7.2.3). Die Hyperebenen kapseln jeweils ein Merkmal, d.h. eine Hyperebene definiert nur den für das jeweilige Merkmal relevanten Teil des Use-Case- und Objektmodells der Produktlinie (vgl. Abschnitt 7.2.4). Die Forderung nach deklarativ vollständigen Hyperebenen fördert eine relativ unabhängige Modellierung und Implementierung der Merkmale, so daß ein Generator sie beliebig (gemäß den im Merkmalmodell hinterlegten Konfigurationsmöglichkeiten) zu Hypermodulen, sprich Anwendungssystemen zusammensetzen kann. Die Anwendungssysteme sind schlank, weil sie nur den Quellcode der von den Kunden gewünschten Merkmale enthalten (vgl. Unterkapitel 7.3 und 8.1).

Die simultane merkmalorientierte Zerlegung von Produktlinien vermeidet das Überschneidungsproblem im Use-Case-Modell vollständig, im Objektmodell hingegen nur zum Teil. Dort können sich Use-Cases weiterhin in den Klassen überschneiden. Diese verbliebenen Überschneidungen ließen sich auslöschen, wenn man Produktlinien simultan auf alle drei Arten zerlegt: funktional in Merkmale, funktional in Use-Cases und objektorientiert in Klassen. Das Ergebnis wäre ein Hyperraum mit drei Dimensionen, deren dritte Dimension das Überschneidungsproblem auch im Objektmodell vollständig löst – zum Preis eines deutlich komplexeren Hyperraums. Ob der gewonnene Nutzen den zu zahlenden Preis rechtfertigt, ist ein Thema für zukünftige Arbeiten. Denn für die Zielsetzung dieser Arbeit trägt die zusätzliche Dimension nichts bei. Warum nicht?

Die Use-Cases einer Produktlinie werden merkmalorientiert zerlegt. Aus den nach Merkmalen getrennten Use-Cases entstehen im nächsten Schritt Klassen, die ebenfalls nach Merkmalen getrennt sind. Folglich können sich in einer Klasse nur noch Use-Cases überschneiden, die zu *demselben* Merkmal gehören, während sich vorher in einer Klasse auch Use-Cases *verschiedener* Merkmale überschnitten haben. Da die Wiederverwendung von Klassen bei der Entwicklung oder Fertigung von Systemen primär über die Merkmale stattfindet, haben Überschneidungen innerhalb eines Merkmals keine negativen Auswirkungen auf die Wiederverwendbarkeit der Klassen beziehungsweise auf die Serienfertigung schlanker Systeme. Somit ist eine dritte Dimension im Hyperraum nicht zwingend erforderlich.

Die Lösung des Überschneidungsproblems durch die Auftrennung der Modelle nach Merkmalen erlaubt eine andere Art der Modellierung von Variabilität, die ohne Vererbung und Entwurfsmuster auskommt. Künstliche Abstraktionen werden reduziert. Als Beispiel greift Abbildung 5.6 wieder das in Abschnitt 4.2.4 eingeführte Komponentensystem catan::spiel auf und stellt die Ergebnisse zweier Refaktorierungen gegenüber: (a) durch Einsatz von Vererbung und Entwurfsmustern (wie bereits in Abbildung 4.7 auf Seite 53 gezeigt) und (b) durch Einsatz von Hyperebenen. Im Vergleich zu (a) benötigt (b) weder eine Abstrakte Fabrik noch die Unterklasse von ZugAbschließen, um die Überschneidungen im ursprünglichen Komponentensystem aufzulösen. Zusätzlich hebt (b) die Über-

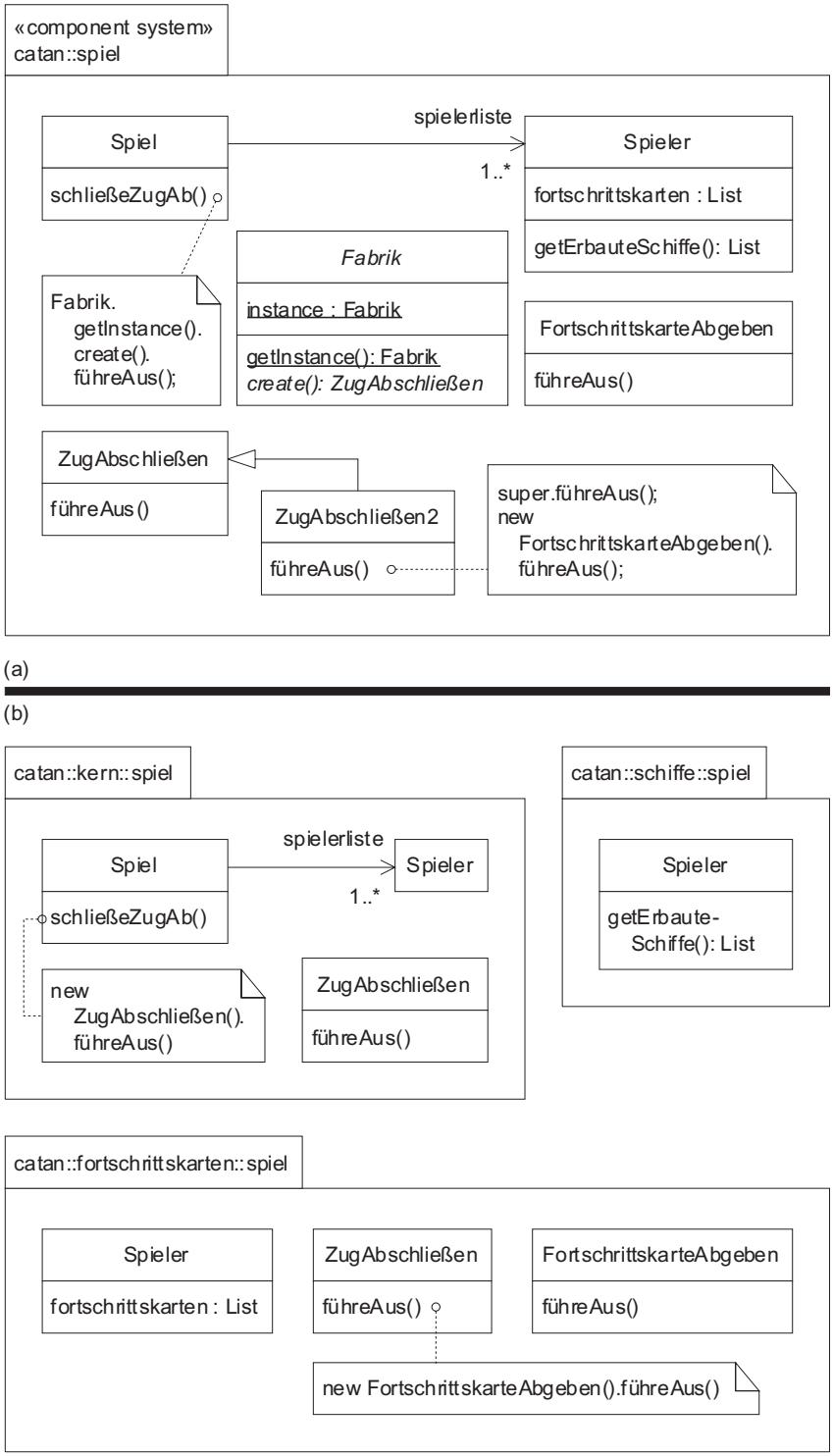

Abbildung 5.6: Gegenüberstellung Variabilität in Komponentensystemen

schneidung in der Klasse Spieler auf, für die (a) um weitere künstliche Abstraktionen hätte ergänzt werden müssen. Die Trennung nach Merkmalen lichtet zudem das Geflecht an *trace*-Abhängigkeitsbeziehungen zwischen den Modellen. Nur eine einzige Beziehung bleibt pro Merkmal übrig. Sie verbindet das Merkmal mit seiner Hyperebene. Insgesamt reduziert sich damit die Komplexität der mit FeatuRSEB entwickelten Produktlinien wieder auf ein handhabbares Maß (vgl. Unterkapitel 8.2).

Darüber hinaus verbessert die Auftrennung der Modelle die Wartbarkeit von Produktlinien, die mit FeatuRSEB entwickelt werden. Der Ausbau einer solchen Produktlinie erfordert seltener tiefe Eingriffe in das Use-Case- und Objektmodell (vgl. Unterkapitel 8.3). Und handhabbarere Komplexität sowie bessere Wartbarkeit zusammen erhöhen die Skalierbarkeit FeatuRSEBs (vgl. Unterkapitel 8.4).

Allerdings ist der Hyperspace-Ansatz so allgemein gehalten, daß er das Überschneidungsproblem in FeatuRSEB nur aus konzeptioneller Sicht löst. Damit Produktlinien-Komponenten mit dem Hyperspace-Ansatz in der Praxis feingranular modelliert und implementiert werden können, muß der Ansatz auf die zu verwendende Modellierungs- und Programmiersprache abgebildet werden. Für die Implementierung in Java ist bereits Hyper/J definiert. Zukünftige Arbeiten könnten Abbildungen auf weitere Programmiersprachen vornehmen. Für Modellierungssprachen fehlt bisher eine entsprechende Abbildung des Hyperspace-Ansatzes. Dies holt das nächste Kapitel für die UML nach.

5.5 Zusammenfassung

Eine Integration des Hyperspace-Ansatzes in die FeatuRSEB-Methode senkt die Komplexität entwickelter Produktlinien und vereinfacht ihren Ausbau. Dadurch leistet der Ansatz einen wesentlichen Beitrag für das Erreichen der Zielsetzung dieser Arbeit. Bevor Kapitel 7 und 8 die Integration von FeatuRSEB und dem Hyperspace-Ansatz vorstellen und bewerten, bildet Kapitel 6 den Ansatz auf die UML ab.

Kapitel 6

Hyper/UML

Das vorangegangene Kapitel stellte den Hyperspace-Ansatz vor, der sich zur Modularisierung feingranularer Komponenten von Produktlinien eignet. Der Ansatz ist unabhängig von Modellierungs- und Programmiersprachen gehalten. Er muß also vor der Anwendung noch auf diejenigen Sprachen abgebildet werden, die bei der Entwicklung von Produktlinien zum Einsatz kommen. Derzeit ist eine Abbildung für Java definiert, genannt Hyper/J. Damit ist zumindest schon einmal (in Java) die Implementierung gemeinsamer und variabler Produktlinien-Komponenten möglich. Um Gemeinsamkeit und Variabilität auch getrennt voneinander (mit der UML) modellieren zu können, wurde im Rahmen der vorliegenden Arbeit Hyper/UML entwickelt: eine Abbildung des Hyperspace-Ansatzes auf die UML in der Version 1.4 [OMG01].

Hyper/UML stellt einen Kernbestandteil dieser Arbeit dar. Die Abbildung beschränkt sich nicht auf einige wenige Ausdrucksmittel der UML, sondern bezieht nahezu alle Diagrammarten der UML ein. Außen vor bleiben lediglich das Komponenten- und das Verteilungsdiagramm. Damit schafft Hyper/UML die Voraussetzungen zur Aufstellung einer Produktlinien-Entwicklungsmethode, die eine vollständige objektorientierte Modellierung von Produktlinien erlaubt, ohne daß die Komplexität der Modelle unbeherrschbar wird und deren Wartbarkeit leidet. Darüber hinaus liefern der Hyperspace-Ansatz beziehungsweise Hyper/UML und Hyper/J die Grundlage für die automatisierte Serienfertigung von Software-Systemen aus Produktlinien: Das Konzept der Hypermodule ermöglicht die Integration der getrennt modellierten und implementierten Gemeinsamkeit und Variabilität einer Produktlinie zu ausführbaren Systemen. Die entsprechende Produktlinien-Methode wird im nächsten Kapitel 7 beschrieben.

6.1 Vorüberlegungen

Wie ist die Herangehensweise an eine Abbildung des Hyperspace-Ansatzes? In Abschnitt 5.2.4 sind sechs Punkte zusammengestellt, die eine vollständige Abbildung beantworten muß: Elemente, Hyperraum, Hyperebenen, Hypermodule, Integrationsbeziehungen und Werkzeuge. Diese sechs Punkte sind also in Bezug auf die UML zu klären.

Am Anfang steht die Überlegung, welche der Elemente, die in UML-Modellen vorkommen, einem Hyperraum zugeordnet werden sollen. Dazu ist ein Blick auf das Metamodell der UML in [OMG01, Kapitel 2] hilfreich. Dort sind alle Elemente, aus denen ein UML-Modell bestehen kann, in Form von Klassen definiert. Jedes Element eines Modells ist eine Instanz einer im Metamodell definierten Klasse. Damit ist jedes Modell eine Instanz des Metamodells.

Eine Metamodellklasse besteht aus Attributen, Assoziationen, Regeln und Operationen. Attribute definieren, welche Eigenschaften Entwickler bei den Instanzen der Klasse (Modellelementen) einstellen können. Assoziationen definieren, wie Modellelemente zueinander in Beziehung stehen, also wie Entwickler Elemente miteinander verknüpfen können. Operationen liefern Informationen über den Zustand der Elemente. Diese Informationen rufen die Regeln ab, um die Gültigkeit der Elemente und damit der Modelle sicherzustellen. Die meisten Regeln sind formal in OCL (Object Constraint Language [OMG01, Kapitel 6]) formuliert.

Damit die Übersicht bei der Vielzahl an Metamodellklassen erhalten bleibt, haben die Autoren der UML die Klassen in Pakete gegliedert. Die Pakete bauen aufeinander auf. So entsteht die Architektur des Metamodells, zu sehen in Abbildung 6.1 (nach [OMG01, Abbildung 1]).

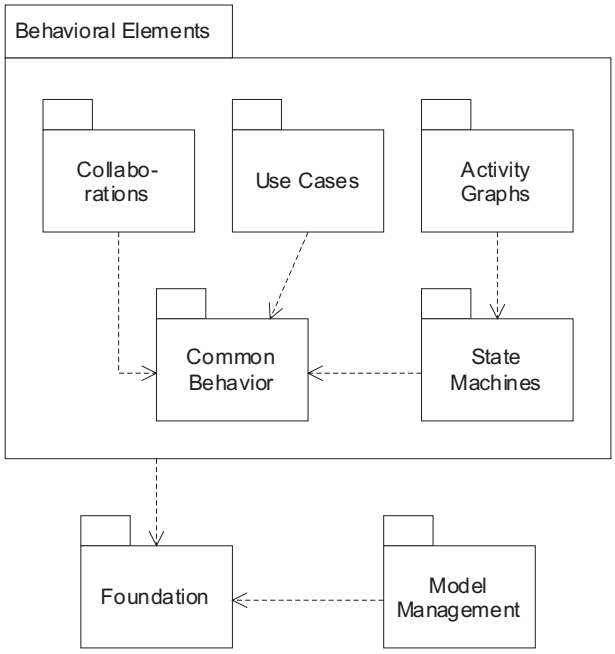

Abbildung 6.1: Architektur des UML-Metamodells

Der nächste Schritt hin zur Beantwortung der eingangs genannten Punkte ist eine Untersuchung der Pakete im Metamodell daraufhin, welche der dort definierten Modellelemente in Hyper/UML als atomare oder zusammengesetzte Elemente angesehen werden können. Unterkapitel 6.2 nimmt diese Untersuchung vor. Dann ist festzulegen, wie Modellelemente einem Hyperraum zugeordnet

werden (vgl. Unterkapitel 6.3) und wie sie in Hyperebenen gekapselt werden (vgl. Unterkapitel 6.4). Die Integration der solchermaßen getrennten Modellelemente erfordert Hypermodule (vgl. Unterkapitel 6.5). Wie die Integration abläuft, d.h. welche Semantik die Integrationsbeziehungen für die Modellelemente haben, beschreiben die Unterkapitel 6.6 bis 6.9 in einer einheitlichen Struktur, die Abschnitt 6.5.3 zuvor erläutert. Ein Ausblick in Unterkapitel 6.10 zur Werkzeugunterstützung für Hyper/UML beschließt das Kapitel.

Die hier vorgenommene Abbildung des Hyperspace-Ansatzes auf die UML führt letztlich zu einer Erweiterung der UML, die, wie in der UML üblich, ein sogenanntes Profil zusammenfaßt. Das Profil startet mit einem leeren Paket namens Hyper/UML (vgl. Abbildung 6.2) und wird im folgenden schrittweise mit Leben erfüllt. Ein Gesamtüberblick zum Profil ist Anhang A zu entnehmen.

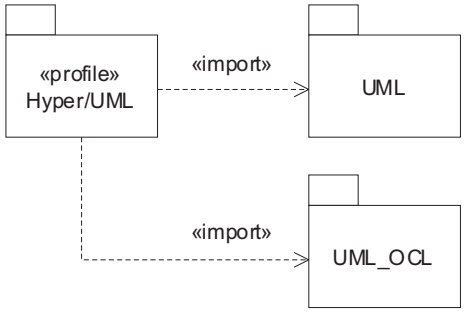

Abbildung 6.2: Profil Hyper/UML

Im weiteren Verlauf des Kapitels wird intensiv auf die UML Bezug genommen, ohne jedoch die UML selbst näher zu erläutern. Dies wurde aus Gründen der Lesbarkeit unterlassen. Der Leser sei statt dessen auf die Primärliteratur [OMG01] oder Sekundärliteratur, z.B. [HK99], verwiesen.

6.2 Elemente

Im Hyperspace-Ansatz werden Software-Systeme simultan auf verschiedene Arten zerlegt. Das Ergebnis der Zerlegung ist eine Menge atomarer und zusammengesetzter Elemente (vgl. Abschnitt 5.2.1).

6.2.1 Syntax

Die Umsetzung atomarer und zusammengesetzter Elemente in eine grafische Syntax zeigt Abbildung 6.3. Elemente in Hyper/UML sind Instanzen konkreter Unterklassen entweder von PrimitiveUnit oder CompoundUnit, je nach dem, ob das Element atomar ist oder sich aus anderen Elementen zusammensetzt. Die Oberklasse dieser beiden Klassen ist HyperspaceUnit. Sie bereitet die Zuordnung der Elemente zu einem Hyperraum vor (vgl. nächstes Unterkapitel 6.3).

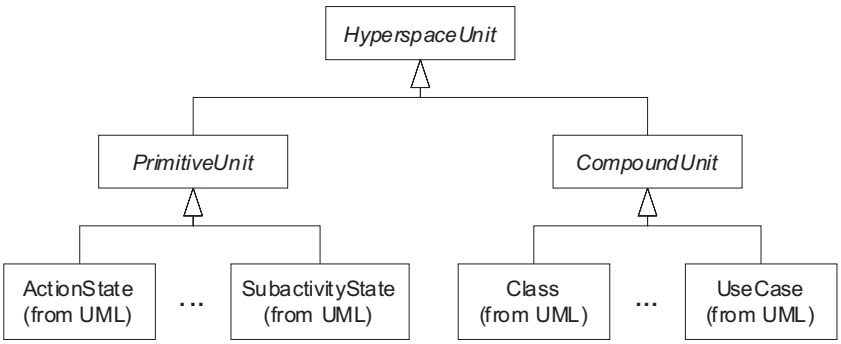

Abbildung 6.3: Profil Hyper/UML – Elemente

Welche Modellelemente der UML sind Elemente in Hyper/UML? Grundsätzlich alle, da nur eine vollständige Abbildung des Hyperspace-Ansatzes auf sämtliche Ausdrucksmittel, sprich Diagrammarten der UML die bestmögliche Unterstützung bei der simultanen Zerlegung von Systemen auf verschiedene Arten bietet. Tabelle 6.1 orientiert sich an den Paketen des UML-Metamodells und schlüsselt von den dort definierten Modellelementen beziehungsweise Metamodellklassen diejenigen auf, die zu den Hyper/UML-Elementen zählen.

ELEMENTE	METAMODELLKLASSE	PAKET
Pakete Modelle	Package Model	Model Management
Klassen Schnittstellen Attribute Assoziationen (zwischen Klassen) Operationen	Class Interface Attribute AssociationEnd Operation	Foundation
Akteure Use-Cases	Actor UseCase	Behavioral Elements:: Use Cases
Zustandsautomaten Zustände	StateMachine SimpleState CompositeState	Behavioral Elements:: State Machines
Signale	Signal	Behavioral Elements:: Common Behavior
Aktivitätsgraphen Aktivitätszustände	ActivityGraph ActionState SubactivityState	Behavioral Elements:: Activity Graphs

Tabelle 6.1: Elemente in Hyper/UML

In der Tabelle fehlen einige Modellelemente der UML. Ihr Ausschluß aus der Liste der Hyper/UML-Elemente wird im folgenden begründet:

- *Beziehungen.* Eine Beziehung verbindet Modellelemente miteinander und legt die Semantik der Verbindung fest, z.B. Generalisierung oder Abhängigkeit. Solche Beziehungen gehören in Hyper/UML zum Zustand eines Modellelements und sind deshalb keine eigenständigen Hyper/UML-Elemente. Eine Ausnahme bilden Assoziationen zwischen Klassen. Ihre Enden sind Elemente in Hyper/UML, weil ein Assoziationsende wie ein Attribut der am anderen Ende der Assoziation befindlichen Klasse aufgefaßt werden kann.

- *Subsysteme.* Ein Subsystem spezifiziert die Schnittstelle eines physischen Systems. Subsysteme fanden in der Fallstudie zu dieser Arbeit (vgl. Kapitel 3) keine Verwendung. Eine Abbildung von Subsystemen im Rahmen von Hyper/UML wäre somit theoretischer Natur und nicht im praktischen Modellierungseinsatz überprüft. Deshalb wird auf die Abbildung zunächst verzichtet und statt dessen als zukünftige Arbeit vermerkt.

- *Komponenten und Knoten.* In der UML bezeichnet eine Komponente einen physisch vorhandenen, implementierten Teil eines Systems, z.B. Quellcode oder ausführbarer Code. Eine Komponente steht in Abhängigkeit zu anderen Komponenten und ist einem Knoten in einer Hardwaretopologie zugeordnet. Komponenten und Knoten werden in Komponenten- und Verteilungsdiagrammen modelliert. Diese Diagramme fanden in der Fallstudie ebenfalls keine Verwendung, so daß sich hier die gleiche Argumentation wie bei den Subsystemen anschließt: Die Abbildung von Komponenten und Knoten in Hyper/UML wird als zukünftige Arbeit vermerkt.

- *Instanzen.* Eine Instanz repräsentiert eine konkrete Ausprägung eines Modellelements (Use-Case, Klasse, Subsystem, Komponente, Knoten usw.) zur Laufzeit des modellierten Systems. Beispielsweise ordnet eine Instanz einer Klasse den Attributen der Klasse Werte zu und ist gemäß den in der Klasse definierten Assoziationen mit anderen Instanzen verbunden. Solche Instanzen werden in Objektdiagrammen, Interaktionsdiagrammen und Aktivitätsdiagrammen (als Objektflüsse) modelliert. Instanzen stellen kein direktes Ergebnis der Zerlegung eines Systems dar, sondern entstehen erst aus den Zerlegungsergebnissen heraus, wenn diese zur Laufzeit des Systems instanziiert werden. Die Beschreibung laufender Systeme ist nicht Bestandteil des Hyperspace-Ansatzes, weshalb Instanzen nicht zu den Hyper/UML-Elementen gehören.

Die Einordnung der Hyper/UML-Elemente in atomar und zusammengesetzt nimmt Abbildung 6.4 vor. Sie zeigt die Hierarchie der Elemente. Ein Element, das kein weiteres enthält, ist atomar. Dies trifft auf Attribute, Assoziationen, Operationen, Akteure und Aktivitätszustände zu. Die entsprechenden Metamodellklassen erben zusätzlich von PrimitiveUnit. Unterklassen von CompoundUnit sind die Metamodellklassen der zusammengesetzten Elemente (vgl. Beispiele in Abbildung 6.3). Hinweis: Die Hierarchie der Hyper/UML-Elemente entspricht nur zum Teil den tatsächlichen Besitzverhältnissen der Elemente in UML-Modellen. Beispielsweise enthalten Zustandsautomaten zunächst einen verfeinerten Zustand (CompositeState), in dem dann einzelne einfache Zustände (SimpleState) oder erneut verfeinerte Zustände auftreten. Laut Hierarchie der Hyper/UML-Elemente enthalten Zustandsautomaten hingegen beide Arten von

Zuständen beliebig ineinander geschachtelt. Diese Abstraktion minimiert die Abhängigkeit Hyper/UMLs von den Interna des UML-Metamodells. Dadurch ist Hyper/UML robuster gegenüber Änderungen am Metamodell, wie sie für die kommende Version 2.0 der UML bereits angekündigt sind.

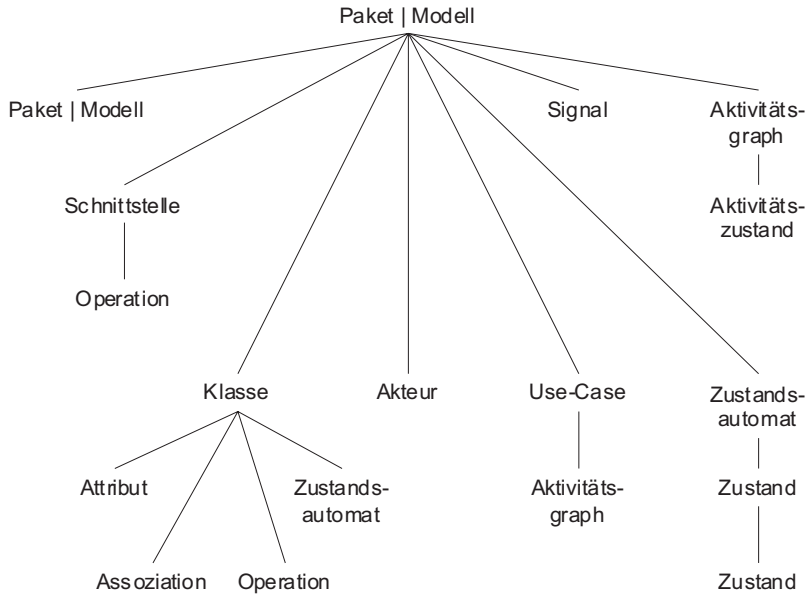

Abbildung 6.4: Hierarchie der Elemente in Hyper/UML

6.2.2 Regeln

Die Syntax-Definition atomarer und zusammengesetzter Elemente benötigt keine zusätzlichen Regeln. Damit nachfolgende Syntax-Definitionen durch die Hierarchie der Hyper/UML-Elemente navigieren können, ist jedoch die Bereitstellung einer Reihe von Operationen nützlich.

CompoundUnit

Operation 1: units() liefert alle in dem zusammengesetzten Element enthaltenen Elemente zurück.

```
units(): Set(HyperspaceUnit);
```

HyperspaceUnit

Operation 2: allParents() liefert die Elementhierarchie zurück, in der sich das Element befindet. Die Liste beginnt mit dem zusammengesetzten Element, in der das Element enthalten ist und endet mit dem obersten zusammengesetzten

Element. Die Liste ist leer, wenn das Element in keinem zusammengesetzten Element enthalten ist.

```
allParents(): Sequence(CompoundUnit);
allParents =
   if (self.parent()->isEmpty())
      then Sequence{}
      else self.parent()->asSequence()->union(self.parent().allParents())
   endif
```

Operation 3: parent() liefert für ein Element, das in einem zusammengesetzten Element enthalten ist, dieses zusammengesetzte Element zurück.

```
parent(): CompoundUnit;
```

6.3 Hyperraum

Ein Hyperraum ordnet die bei der Zerlegung eines Software-Systems identifizierten Elemente den Belangen seiner Dimensionen zu. Die Dimensionen repräsentieren jeweils eine Zerlegungsart des Systems (vgl. Abschnitt 5.2.1).

6.3.1 Syntax

Die Umsetzung eines Hyperraums in eine grafische Syntax zeigt Abbildung 6.5. Ein Hyperraum in Hyper/UML ist eine Instanz der Klasse Hyperspace, die mit ihren Dimensionen assoziiert ist. Die Dimensionen wiederum zerfallen in ihre Belange beziehungsweise Concern-Objekte. Zusätzlich besitzt jede Dimension einen 0-Belang (NoneConcern). Hyper/UML-Elemente sind indirekt Instanzen von HyperspaceUnit und über diesen Weg den Belangen eines Hyperraums zugeordnet.

6.3.2 Regeln

Die folgenden Regeln und Operationen vervollständigen die Syntax-Definition eines Hyperraums.

AbstractConcern

Operation 4: includesUnit() liefert true zurück, wenn das übergebene Element dem Belang explizit oder implizit zugeordnet ist. Die Operation wird bei der Integration von Belangen zu Hypermodulen genutzt. Mit ihr kann festgestellt werden, ob ein Element an einer Integration teilnimmt.

```
includesUnit(u : HyperspaceUnit): Boolean;
includesUnit =
   u.concernInDimension(self.dimension) = self
```

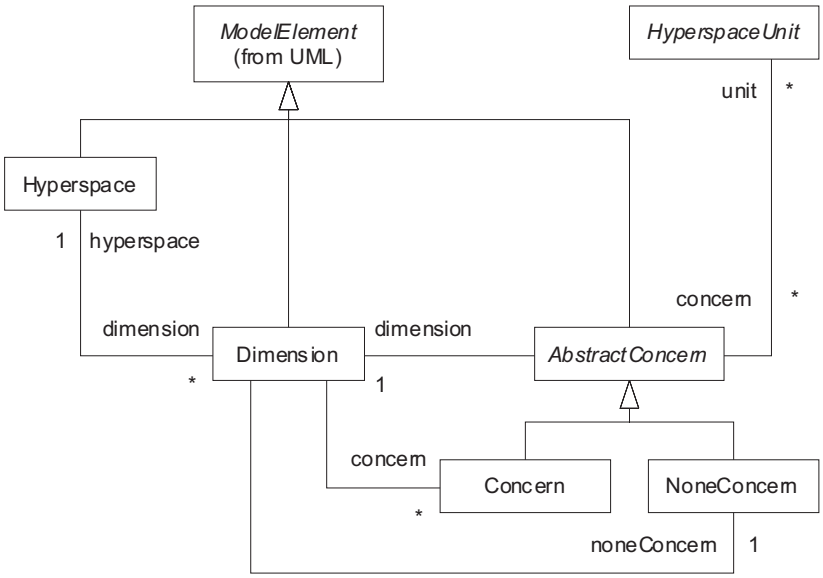

Abbildung 6.5: Profil Hyper/UML – Hyperraum

Dimension

Regel 1: Die Belange einer Dimension sind verschieden benannt.

```
self.allConcerns()->isUnique( c | c.name)
```

Operation 5: allConcerns() liefert die Belange einer Dimension zurück.

```
allConcerns(): Set(AbstractConcern);
allConcerns =
    self.concern->including(noneConcern)
```

Hyperspace

Regel 2: Pro UML-Modell, d.h. pro System existiert nur ein Hyperraum, weil der Hyperspace-Ansatz nicht mehrere Hyperräume für ein System vorsieht. Die Klasse Hyperspace ist folglich ein Singleton [GHJV95, Seite 127ff].

```
Hyperspace.allInstances->size() = 1
```

Regel 3: Die Dimensionen eines Hyperraums sind verschieden benannt.

```
self.dimension->isUnique( dim | dim.name)
```

HyperspaceUnit

Regel 4: Jedes Element ist in jeder Dimension nur einem Belang explizit zugeordnet – eine Forderung des Hyperspace-Ansatzes.

```
self.concern->isUnique( c | c.dimension)
```

Operation 6: concernInDimension() liefert den Belang zurück, dem das Element in der übergebenen Dimension zugeordnet ist. Diese Zuordnung muß nicht explizit geschehen, sondern kann auch implizit über die Elementhierarchie gegeben sein. Ist auch dies nicht der Fall, ist das Element automatisch dem 0-Belang zugeordnet.

```
concernInDimension(dim : Dimension): AbstractConcern;
concernInDimension =
   if (self.concern->exists( c | c.dimension = dim))
      then self.concern->any( c | c.dimension = dim)
      else if (self.parent()->notEmpty())
         then self.parent().concernInDimension(dim)
         else dim.noneConcern
      endif
   endif
```

NoneConcern

Regel 5: Der 0-Belang heißt in Hyper/UML analog zu Hyper/J „None“.

```
self.name = 'None'
```

6.4 Hyperebenen

Hyperebenen kapseln die Belange (d.h. ihre Elemente) eines Hyperraums und ermöglichen so die getrennte Betrachtung und Bearbeitung von Belangen (vgl. Abschnitt 5.2.2).

6.4.1 Syntax

Die Umsetzung von Hyperebenen in eine grafische Syntax zeigt Abbildung 6.6. Hyperebenen in Hyper/UML sind Instanzen der Klasse Hyperslice, einer Unterklasse von Package. Ein Hyperebenen-Paket enthält Hyper/UML-Elemente und ihre zugehörigen UML-Modellelemente, z.B. Beziehungen.

Hyperebenen sind deklarativ vollständig. Benötigt eine Hyperebene ein Element aus einer anderen Ebene, so deklariert sie das Element nur bis zu dem Umfang, in dem sie es nutzt beziehungsweise es die im UML-Metamodell formulierten Regeln erforderlich machen. Letzteres ist eine notwendige Forderung, um die Gültigkeit des Modells nicht zu verletzen.

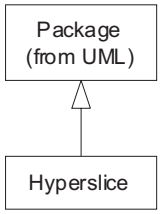

Abbildung 6.6: Profil Hyper/UML – Hyperebenen

6.4.2 Regeln

Die folgende Regel vervollständigt die Syntax-Definition von Hyperebenen.

Hyperslice

Regel 6: Hyperebenen können nicht ineinander geschachtelt werden, weil sie Module darstellen, die voneinander unabhängig sein sollen, was durch die deklarative Vollständigkeit erreicht wird. Eine Schachtelung von Hyperebenen würde die Ebenen in eine hierarchische Beziehung zueinander setzen und führt zu einer Abhängigkeit der untergeordneten von den übergeordneten Ebenen.

```
self.allSurroundingNamespaces()->forAll( ns |
    not ns.isOclKindOf(Hyperslice))
```

6.5 Hypermodule

Hypermodule beschreiben wie die in den Hyperebenen getrennt gekapselten Belange (d.h. ihre Elemente) wieder miteinander zu Systemen integriert werden (vgl. Abschnitt 5.2.3).

6.5.1 Syntax

Die Umsetzung von Hypermodulen in eine grafische Syntax zeigt Abbildung 6.7. Hypermodule in Hyper/UML sind Instanzen der Klasse Hypermodule, einer Unterklasse von Hyperslice. Dadurch erben Hypermodule die für Hyperebenen definierten Regeln, z.B. die deklarative Vollständigkeit. Ein Hypermodul ist mit den zu integrierenden Belangen assoziiert. Erfordert die Integration spezielle Integrationsbeziehungen, sind diese von den Entwicklern im Hypermodul in einer Liste anzugeben. Hyper/UML unterstützt vier Arten von Integrationsbeziehungen, die jeweils die konkreten Unterklassen von IntegrationRelationship umsetzen:

Integrationsbeziehung	Klasse
Verschmelzen	MergeRelationship
Ordnen	OrderRelationship
Summieren	SummaryFunctionRelationship
Ersetzen	OverrideRelationship

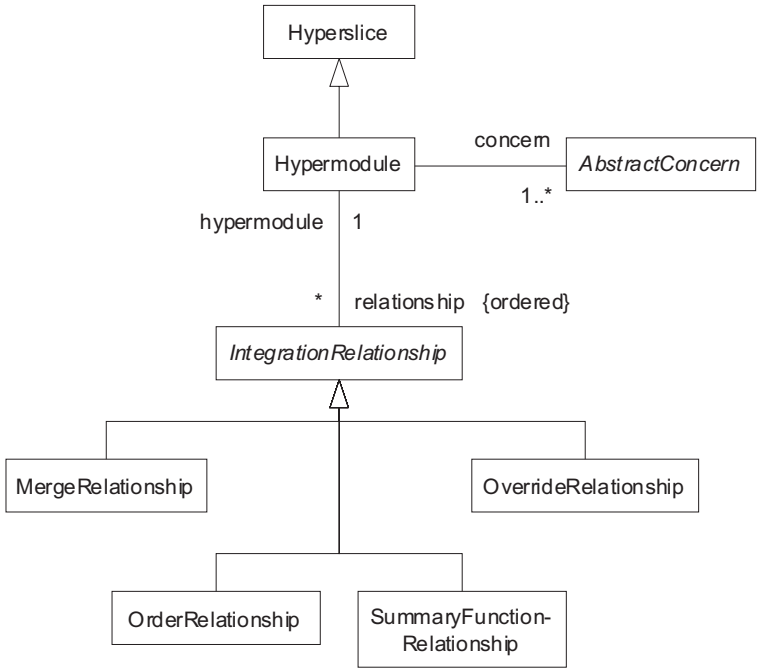

Abbildung 6.7: Profil Hyper/UML – Hypermodule

Die vier Arten wurden aus Hyper/J übernommen (vgl. Abschnitte 5.3.5 bis 5.3.8). Warum? Erstens reichten in der Fallstudie „Die Siedler von Catan“ die genannten Integrationsbeziehungen vollkommen aus, um die Integration von Belangen zu verwirklichen. Dies muß natürlich auf andere Systeme nicht unbedingt zutreffen, obwohl der Umfang der Fallstudie von über 80 Use-Cases und 300 Klassen (vgl. Unterkapitel 8.4) zumindest die Wahrscheinlichkeit dafür erhöht. Zweitens erleichtert die Verwendung der gleichen Arten von Integrationsbeziehungen den Übergang vom Entwurf zur Implementierung, da nicht erst Abbildungsvorschriften zwischen den Arten zu definieren sind. Drittens stellen die Integrationsbeziehungen einen während der Durchführung der Fallstudie als gelungen empfundenen Kompromiß dar: Sie sind von Entwicklern leicht, beinahe intuitiv zu verstehen, aber auch mächtig genug, um die notwendigen Integrationen realisieren zu können. Trotzdem soll damit die Suche nach weiteren nützlichen Integrationsbeziehungen sowohl für Hyper/UML als auch für Hyper/J nicht als abgeschlossen betrachtet werden. Sie wird vielmehr Bestandteil zukünftiger Arbeiten sein.

Die Liste der Integrationsbeziehungen eines Hypermoduls ist geordnet, weil die Reihenfolge, in der Elemente miteinander integriert werden, für die Integrationsbeziehungen Verschmelzen und Ersetzen von Bedeutung ist. Angenommen, in einem Hypermodul seien zwei Integrationsbeziehungen angegeben:

```
override B with C    // (1)
merge A,B    // (2)
```

Die erste Beziehung ersetzt B vollständig durch C. Dann will die zweite Beziehung A und B verschmelzen. B wurde aber ersetzt und existiert nicht mehr. Statt dessen könnte Hyper/UML A und C verschmelzen, da C der „Nachfahre" von B ist. Ein solcher Automatismus ist problematisch, denn Hyper/UML verfügt nur über das Wissen, wie Elemente zu integrieren sind, nicht aber, warum sie integriert werden. Deshalb bricht die Integration mit einem Fehler ab, wenn eines der zu integrierenden Elemente nicht mehr existiert. Die Entwickler müssen also Integrationsbeziehungen entsprechend anordnen oder umformulieren. Bezüglich des Beispiels hieße das, (2) in merge A,C ändern.

Nicht von Bedeutung ist die Reihenfolge, in der Integrationsbeziehungen vom Typ Ordnen und Summieren im Hypermodul angegeben werden. Beide Typen liefern zusätzliche Informationen, die beim Verschmelzen von Elementen berücksichtigt werden. An welcher Stelle die Informationen im Hypermodul stehen ist dabei unerheblich, weil beim Verschmelzen grundsätzlich alle Ordnen- und Summieren-Beziehungen des Hypermoduls beachtet werden.

6.5.2 Regeln

Die Syntax-Definition von Hypermodulen benötigt keine zusätzlichen Regeln und Operationen.

6.5.3 Ablauf der Integration

Jedes Element ist im Hyperraum eindeutig einem Belang in einer Dimension zugeordnet. Deshalb steht durch die Angabe der zu integrierenden Belange im Hypermodul fest, welches die zu integrierenden Elemente sind: Elemente, die den angegebenen Belangen zugeordnet sind. Die Zuordnung kann explizit oder implizit über das übergeordnete Element gegeben sein. Ziel der Integration ist es, ein Hypermodul-Paket zu erstellen, das keine Regel des UML-Metamodells und Hyper/UML-Profils verletzt. Ob das Hypermodul semantisch korrekt ist, kann Hyper/UML nicht überprüfen. Diese Aufgabe müssen Entwickler durch Testen erledigen. Der Ablauf der Integration besteht aus zwei Schritten.

Schritt 1: Elemente ins Hypermodul kopieren

Hyper/UML kopiert alle zu integrierenden Elemente in das Hypermodul-Paket und ordnet sie dort in die gleiche Namensraumhierarchie ein, in der sie sonst auch liegen. Kopierte Hyperebenen werden zu gewöhnlichen Paketen. Andernfalls wäre Regel 6 auf Seite 100 verletzt, die eine Schachtelung von Hyperebenen und Hypermodulen untersagt.

Die UML verwaltet für jedes Element eines Namensraums die Sichtbarkeit des Elements. Sie bestimmt, welche anderen Elemente im Modell Zugriff auf ein Element haben. Vordefinierte Sichtbarkeiten sind public, protected, package und private. Die im nächsten Schritt anstehende Integration der kopierten Elemente durch die im Hypermodul angegebenen Integrationsbeziehungen kann dazu führen, daß die Sichtbarkeitsschranke nicht mehr eingehalten wird. Um zu verhindern, daß durch die Integration ein ungültiges Hypermodul entsteht, wird

die Sichtbarkeit aller kopierten Elemente auf public gesetzt. Diese Aufweichung des Sichtbarkeitskonzeptes der UML birgt die Gefahr, daß unerlaubte Zugriffe beim Testen des Hypermoduls nicht erkannt werden. Zukünftige Arbeiten sollten deshalb diskutieren, inwieweit die Aufweichung ganz oder teilweise zurückzunehmen ist.

Schritt 2: Integrationsbeziehungen abarbeiten

Die Integrationsbeziehungen führen weitere Integrationen mit den in das Hypermodul kopierten Elementen durch. Wie das genau geschieht, sagen die folgenden Unterkapitel. Ein Unterkapitel beschreibt jeweils eine Art von Integrationsbeziehung. Es beginnt mit der Festlegung von Syntax und Regeln für die Angabe der Integrationsbeziehung und beschreibt danach, welche Integrationssemantik die Beziehung für die einzelnen Hyper/UML-Elemente besitzt. Zur Vermeidung von Wiederholungen erfolgt die Beschreibung nicht getrennt für jedes Hyper/UML-Element, sondern für Gruppen zusammengehöriger Elemente. Alle Beschreibungen folgen einem einheitlichen Format:

1. *Beispiel.* Eine Beschreibung führt zunächst ein Beispiel in Form von UML-Diagrammen aus der Fallstudie „Die Siedler von Catan“ ein. Die Diagramme zeigen Ausgangssituation und Ergebnis einer Integration, sprich die zu integrierenden Belange und das Hypermodul.

2. *Vorbedingungen.* Vorbedingungen legen fest, welchen Bedingungen die zu integrierenden Elemente genügen müssen. Eine häufige Vorbedingung ist die Forderung, daß bestimmte Eigenschaften der zu integrierenden Elemente gleich sein müssen. Solche Forderungen ergeben sich daraus, daß im Fall ungleicher Eigenschaften die Integration semantisch nicht sinnvoll ist oder zu Problemen führen würde. Ist eine Vorbedingung nicht erfüllt, bricht die Integration mit einem Fehler ab. Der Fehler ist dann von Entwicklern durch Ändern der Elemente oder des Hypermoduls zu beheben.

3. *Nachbedingungen.* Der Abschnitt Nachbedingungen erläutert, teilweise anhand des Beispiels, wie die Elemente integriert werden, sprich wie die Eigenschaften der zu integrierenden Elemente zusammengeführt werden. Um die Auflistung der Eigenschaften zu gliedern und ein Nachschlagen zu vereinfachen, werden Randnotizen verwendet. Eine Randnotiz nennt eine Eigenschaft, deren Integration in den nebenstehenden Absätzen beschrieben ist.

4. *Ausnahmen.* Die Nachbedingungen decken nur den idealen, fehlerfreien Verlauf der Integration ab. Mögliche Fehlerquellen, die die Vorbedingungen nicht abfangen und deren Konsequenz ein ungültiges, die Regeln des UML-Metamodells verletzendes Hypermodul wäre, faßt deshalb der Abschnitt Ausnahmen zusammen. Ein solcher Fehler bricht die Integration ab und muß von Entwicklern durch Ändern der Elemente oder des Hypermoduls behoben werden.

Die Abschnitte Beispiel, Vorbedingungen und Ausnahmen sind optional.

6.6 Integrationsbeziehung Verschmelzen

Die Integrationsbeziehung Verschmelzen (engl.: *merge relationship*) verschmilzt zwei oder mehr Elemente gleichen Typs miteinander.

6.6.1 Syntax

Die Umsetzung der Integrationsbeziehung Verschmelzen in eine grafische Syntax zeigt Abbildung 6.8. Eine Integrationsbeziehung Verschmelzen ist mit den zu verschmelzenden Elementen assoziiert. Diese Elemente zerfallen in zwei Kategorien: ein Ziel-Element (targetUnit) und ein oder mehrere Quell-Elemente (sourceUnit). Im Ziel-Element sind nach dem Verschmelzen die Informationen aus den zu verschmelzenden Elementen vereinigt, wohingegen die Quell-Elemente (ihre Kopien im Hypermodul) entfernt wurden. Referenziert eine nachfolgende Integrationsbeziehung Verschmelzen oder Ersetzen ein verschmolzenes, entferntes Quell-Element, bricht die Integration mit einem Fehler ab.

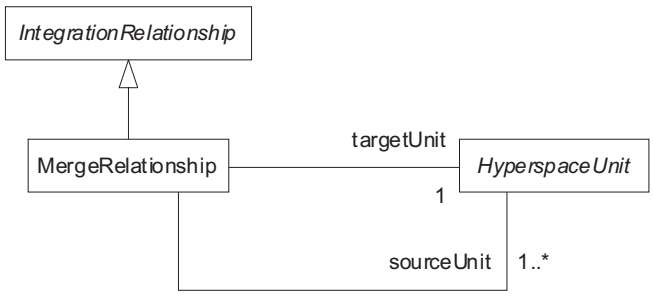

Abbildung 6.8: Profil Hyper/UML – Integrationsbeziehung Verschmelzen

Handelt es sich bei den zu verschmelzenden Elementen um zusammengesetzte Elemente, enthält das Ziel-Element nach dem Verschmelzen zusätzlich die in den Quell-Elementen (ihren Kopien im Hypermodul) enthaltenen Elemente. Übereinstimmende enthaltene Elemente (vgl. die zum Teil elementspezifischen Implementierungen der Operation correspondsTo() im nächsten Abschnitt) wurden zu einem Element verschmolzen. Dieser Automatismus kann für einzelne enthaltene Elemente bestimmter Typen durch Angabe einer Integrationsbeziehung Ersetzen im Hypermodul abgeschaltet werden (vgl. Unterkapitel 6.9).

6.6.2 Regeln

Die folgenden Regeln und Operationen vervollständigen die Syntax-Definition der Integrationsbeziehung Verschmelzen. Sie ergänzen auch einige im UML-Metamodell definierte Klassen. Weitere Regeln für die zu verschmelzenden Elemente legen die nachfolgenden Abschnitte zur Semantik in Form von Vorbedingungen fest.

MergeRelationship

Regel 7: Das Ziel-Element ist nicht zugleich Quell-Element. Das verhindert, daß ein Element mit sich selbst verschmolzen wird.

```
self.targetUnit->intersection(self.sourceUnit)->isEmpty()
```

Regel 8: Die zu verschmelzenden Elemente sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.units()->forAll( u |
   self.hypermodule.concern->exists( c |
      c.includesUnit(u)))
```

Regel 9: Die zu verschmelzenden Elemente sind vom aufgezählten Typ. Hier sind alle Elementtypen von Hyper/UML erlaubt, weil die Semantik des Verschmelzens für jeden Typ festgelegt ist.

```
self.units()->forAll( u |
   if (u.oclIsTypeOf(AssociationEnd))
      then u.participant.oclIsTypeOf(Class)
      else Set{
         Package, Model, Hyperslice,
         Interface, Class, Attribute, Operation,
         Actor, UseCase,
         StateMachine, SimpleState, CompositeState,
         Signal,
         ActivityGraph, ActionState, SubactivityState
      }->includes(u.oclType()))
   endif
```

Regel 10: Die zu verschmelzenden Elemente sind vom selben Typ. Elemente verschiedenen Typs, z.B. Use-Cases und Zustände, können semantisch nicht verschmolzen werden.

```
self.units()->forAll( u1, u2 |
   (u1.oclType() = u2.oclType()) and
   (u1.stereotype = u2.stereotype))
```

Operation 7: units() liefert die zu verschmelzenden Elemente zurück. Die obigen Regeln verwenden diese Operation.

```
units(): Set(HyperspaceUnit);
units =
   self.targetUnit->union(self.sourceUnit)
```

ModelElement (Ergänzung)

Operation 8: correspondsTo() liefert true zurück, wenn das Modellelement mit dem übergebenen Element übereinstimmt. Unterklassen, bei denen die Übereinstimmung weiteren Bedingungen unterliegt, überschreiben diese Operation.

```
correspondsTo(me : ModelElement): Boolean;
correspondsTo =
   (self <> me) and
   (self.oclType() = me.oclType()) and
   (self.stereotype = me.stereotype) and
   (self.name <> '') and (self.name = me.name)
```

Operation (Ergänzung)

Operation 9: correspondsTo() liefert true zurück, wenn die Operation mit der übergebenen Operation übereinstimmt. Zusätzlich zu den Prüfungen in ModelElement wird bei Operationen die Signatur verglichen.

```
correspondsTo(op : Operation): Boolean;
correspondsTo =
   self.oclAsType(ModelElement).correspondsTo(op) and
   self.hasSameSignature(op)
```

6.6.3 Semantik

Die Integrationsbeziehung Verschmelzen ist die mächtigste Beziehung in Hyper/UML. Ihre Semantik ist für alle Hyper/UML-Elemente spezifiziert und unterscheidet sich wesentlich vom Typ der zu verschmelzenden Elemente. Dennoch gibt es einen kleineren Teil der Semantik, der über alle Elementtypen hinweg gleich ist, weil die Metamodellklasse jedes UML-Modellelements und damit jedes Hyper/UML-Elements von der abstrakten Oberklasse ModelElement erbt. Wie die dort definierten Attribute und Assoziationen verschmelzen, beschreibt dieser Abschnitt.

Abbildung 6.9 zeigt im oberen Teil die zu integrierenden Belange Kern und Stadtausbau mit einigen Klassen. Der untere Teil präsentiert das Ergebnis der Integration. Die Ziel-Klasse Kern::Spieler wurde mit der Quell-Klasse Stadtausbau::Spieler verschmolzen. Interessant ist dabei vor allem, wie Einschränkungen, Eigenschaftswerte und Abhängigkeiten der Klassen beziehungsweise von Elementen allgemein verschmelzen. Weitere Erläuterungen zum Verschmelzen von Klassen folgen in Abschnitt 6.6.5.

Vorbedingungen

Den zu verschmelzenden Elementen ist kein Eigenschaftswert gleichen Namens und unterschiedlichem Wert zugeordnet. Begründung: Im Beispiel ist der Klasse Kern::Spieler der Eigenschaftswert persistence=true zugeordnet. Würde die

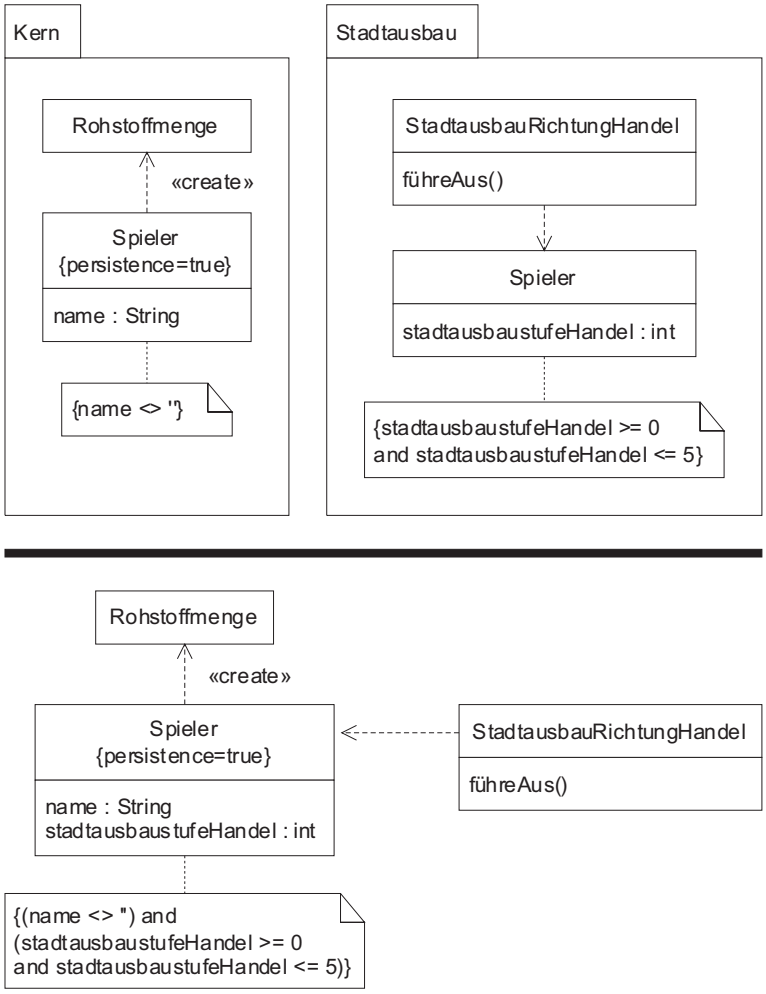

Abbildung 6.9: Verschmelzen von Elementen

Klasse Stadtausbau::Spieler jetzt ebenfalls die Eigenschaft persistence besitzen, allerdings mit dem Wert false, wäre unklar, welchen Wert persistence in der integrierten Klasse haben soll: true oder false?

Nachbedingungen

Name

Das Ziel-Element behält seinen Namen, sofern vorhanden. Die Namen der Quell-Elemente werden ignoriert. Textuelle Referenzen auf die Quell-Elemente (über ihren Namen), z.B. in Einschränkungen und Kommentaren, sind an den Namen des Ziel-Elements angepaßt.

Einschränkungen

Dem Ziel-Element sind zusätzlich die Einschränkungen der Quell-Elemente zugeordnet. Dies entspricht einer UND-Verknüpfung der Einschränkungen. Eine zukünftige Erweiterung von Hyper/UML ist die wahlweise ODER-Verknüpfung

von Einschränkungen. Im Ergebnis des Beispiels sind die Einschränkungen der Klasse Spieler aus Kern und Stadtausbau UND-verknüpft.

Eigenschaftswerte Dem Ziel-Element sind zusätzlich die Eigenschaftswerte der Quell-Elemente zugeordnet. Redundante Eigenschaftswerte wurden entfernt.

Abhängigkeitsbeziehungen Das Ziel-Element steht zusätzlich in Abhängigkeit zu den Modellelementen, von denen die Quellelemente abhängen. Modellelemente, die von Quellelementen abhängen, stehen statt dessen in Abhängigkeit zum Ziel-Element. Redundante Abhängigkeitsbeziehungen wurden entfernt. Im Ergebnis des Beispiels hängt StadtausbauRichtungHandel von der Ziel-Klasse Spieler aus Kern ab.

Ausnahmen

UML-Modelle sind ungültig, wenn Einschränkungen sich widersprechen. Stehen nach dem Verschmelzen die Einschränkungen des Ziel-Elements im Widerspruch zueinander, bricht die Integration mit einem Fehler ab. Die Widerspruchsfreiheit von Einschränkungen kann nur geprüft werden, wenn die Einschränkungen in einer formalen Sprache, z.B. OCL, formuliert sind.

6.6.4 Semantik für Pakete, Modelle und Hyperebenen

Abbildung 6.10 zeigt im oberen Teil die zu integrierenden Belange Kern und Schiffe mit ihren Paketen catan::spiel und catan::spielplan. Der untere Teil präsentiert das Ergebnis der Integration. Das Ziel-Paket Kern::catan::spiel wurde mit dem gleichnamigen Quell-Paket aus Schiffe verschmolzen.

Vorbedingungen

Ein Paket befindet sich in einer Hierarchie von Paketen. Das Verschmelzen von Paketen über verschiedene Hierarchieebenen hinweg ist semantisch problematisch. Deshalb müssen die zu verschmelzenden Pakete alle in einer identischen Hierarchieebene stehen. Diese Bedingung wird unter Einbeziehung aller Integrationsbeziehungen des Hypermoduls geprüft. Sie ist erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt, z.B. indem sie die übergeordneten Pakete der zu verschmelzenden Pakete verschmilzt. Die gleiche Vorbedingung gilt für Modelle und Hyperebenen.

Nachbedingungen

Das Ziel-Paket enthält zusätzlich die Elemente der Quell-Pakete. Übereinstimmende Elemente (vgl. die Implementierungen der Operation correspondsTo() in Abschnitt 6.6.2) verschmelzen miteinander. Die gleiche Semantik gilt für Modelle und Hyperebenen. Im Ergebnis des Beispiels enthält das Paket catan::spiel zusätzlich die Klasse SchiffBauen aus Schiffe, während Spieler aus der Übereinstimmung zwischen Kern und Schiffe hervorgegangen ist.

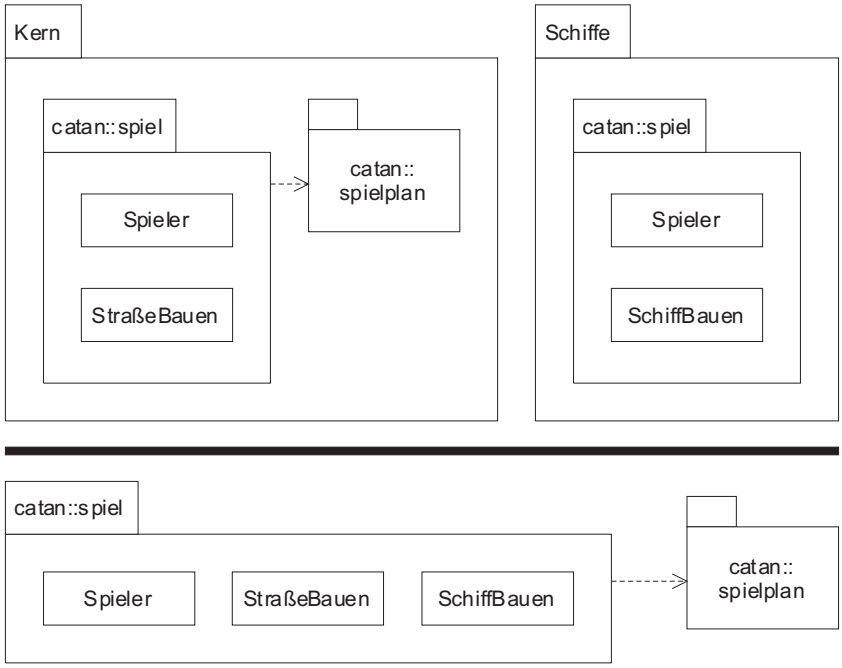

Abbildung 6.10: Verschmelzen von Paketen, Modellen und Hyperebenen

6.6.5 Semantik für Klassen und Schnittstellen

Abbildung 6.11 zeigt im oberen Teil die zu integrierenden Belange Kern und Stadtausbau mit einigen Klassen und Schnittstellen. Der untere Teil präsentiert das Ergebnis der Integration. Die Ziel-Klassen Spieler, Aktion und RohstoffeAufHandelsplätzenHandeln aus Kern wurden jeweils mit der gleichnamigen Quell-Klasse aus Stadtausbau verschmolzen. Außerdem wurde die Ziel-Schnittstelle Kern::Handelsplatz mit der gleichnamigen Quell-Schnittstelle aus Stadtausbau verschmolzen.

Vorbedingungen

Die zu verschmelzenden Klassen sind alle abstrakt oder alle konkret. Sie sind alle passiv oder alle aktiv. Aktiv bedeutet, daß jede Instanz einer Klasse in ihrem eigenen Kontrollfluß (Thread) läuft.

Nachbedingungen

Klassen: Attribute, Assoziationen

Die Ziel-Klasse enthält zusätzlich die in den Quell-Klassen definierten Attribute und Assoziationen. Übereinstimmende Attribute und Assoziationen (vgl. Operation 8 auf Seite 106) verschmelzen miteinander (vgl. Abschnitt 6.6.6). Im Ergebnis des Beispiels definiert Spieler zusätzlich das Attribut stadtausbaustufeHandel, weil Stadtausbau dies spezifiziert.

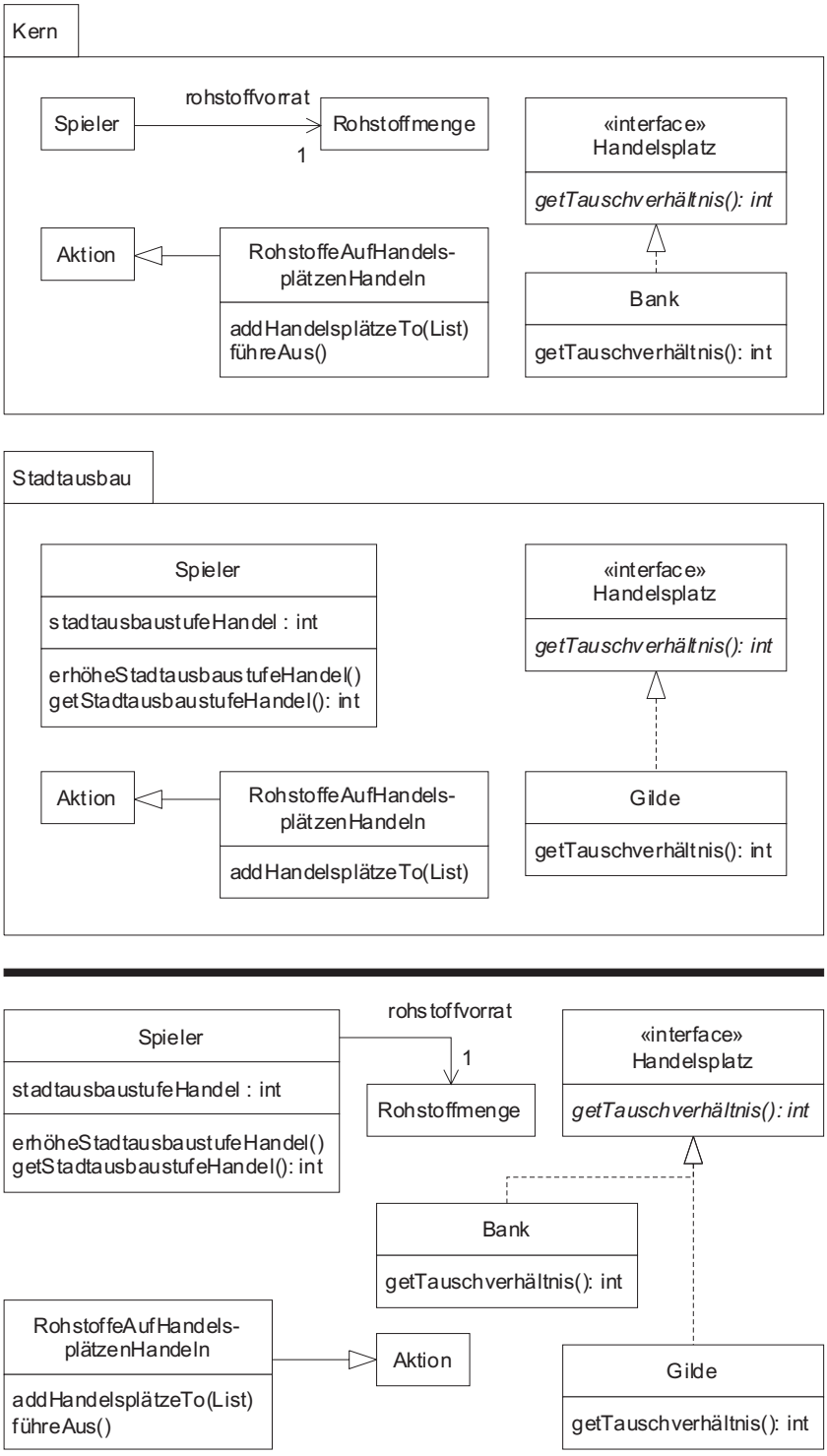

Abbildung 6.11: Verschmelzen von Klassen und Schnittstellen

Klassen/ Schnittstellen: Operationen

Die Ziel-Klasse enthält zusätzlich die in den Quell-Klassen definierten Operationen. Übereinstimmende Operationen (vgl. Operation 9 auf Seite 106) verschmelzen miteinander (vgl. Abschnitt 6.6.6). Die gleiche Semantik gilt für Schnittstellen. Im Ergebnis des Beispiels definiert die Klasse Spieler zusätzlich die Operationen erhöheStadtausbaustufeHandel() und getStadtausbaustufeHandel() aus Stadtausbau. In RohstoffeAufHandelsplätzenHandeln und Handelsplatz wurden übereinstimmende Operationen verschmolzen: addHandelsplätzeTo() beziehungsweise getTauschverhältnis().

... Vererbungsbeziehungen

Die Ziel-Klasse erbt zusätzlich von denjenigen Klassen, von denen die Quell-Klassen erben. Klassen, die von den Quell-Klassen erben, erben statt dessen von der Ziel-Klasse. Redundante Generalisierungsbeziehungen wurden entfernt. Die gleiche Semantik gilt für Schnittstellen, mit dem Zusatz, daß Operationen entfernt wurden, die über die gesamte Vererbungshierarchie der Ziel-Schnittstelle gesehen redundant definiert waren. Im Ergebnis des Beispiels wurde die Vererbungshierarchie von Aktion zusammengeführt.

Klassen: Realisierungsbeziehungen

Die Ziel-Klasse realisiert zusätzlich diejenigen Schnittstellen, die die Quell-Klassen realisieren. Andere Klassen, die eine Quell-Schnittstelle realisieren, realisieren statt dessen die Ziel-Schnittstelle. Redundante Realisierungsbeziehungen wurden entfernt. Im Beispiel realisiert Gilde die Quell-Schnittstelle Handelsplatz. Entsprechend realisiert die Klasse im Ergebnis die Ziel-Schnittstelle Handelsplatz.

... Zustandsautomaten

Die Ziel-Klasse enthält zusätzlich die Zustandsautomaten der Quell-Klassen. Übereinstimmende Zustandsautomaten (vgl. Operation 8 auf Seite 106) verschmelzen miteinander (vgl. Abschnitt 6.6.8).

... Instanzen

Instanzen von Klassen finden sich in der UML in Interaktionen und in Aktivitätsgraphen (als Objektflüsse). Die Instanzen einer Quell-Klasse sind statt dessen Instanzen der Ziel-Klasse.

Ausnahmen

Die UML verbietet Zyklen in den modellierten Vererbungshierarchien [OMG01, Abschnitt 2.5.3.20, Regel 3]. Entstehen durch das Verschmelzen von Klassen oder Schnittstellen Zyklen in der Vererbungshierarchie, bricht die Integration mit einem Fehler ab.

Die UML erlaubt nicht, daß Klassen Attribute oder Assoziationen definieren, die unter dem gleichen Namen bereits eine Oberklasse definiert [OMG01, Abschnitt 2.5.4.4]. Tritt dieser Fall durch das Verschmelzen ein, bricht die Integration mit einem Fehler ab.

Die UML fordert, daß Klassen, die Schnittstellen realisieren, alle dort vereinbarten Operationen implementieren [OMG01, Abschnitt 2.5.3.8, Regel 6]. Realisiert eine Klasse nach dem Verschmelzen durch die Umordnung der Realisierungsbeziehungen nicht mehr alle notwendigen Operationen und beheben weitere Integrationsbeziehungen des Hypermoduls dieses Problem nicht, bricht die Integration mit einem Fehler ab.

6.6.6 Semantik für Attribute, Assoziationen und Operationen

Abbildung 6.12 zeigt im oberen Teil die zu integrierenden Belange Kern und Stadtausbau mit den Implementierungen der Operationen RohstoffeAufHandelsplätzenHandeln::führeAus() und addHandelsplätzeTo() aus dem Beispiel im vorangegangenen Abschnitt 6.6.5. Der untere Teil präsentiert das Ergebnis der Integration. Die Ziel-Operation addHandelsplätzeTo() aus Kern wurde mit der gleichnamigen Quell-Operation aus Stadtausbau verschmolzen.

Vorbedingungen

Attribute, Assoziationen und Operationen dürfen nie losgelöst von ihren Klassen verschmolzen werden. D.h. sollen Attribute verschmelzen, muß im Hypermodul eine Integrationsbeziehung definiert sein, die direkt oder indirekt (über die Übereinstimmung in Paketen) die Klassen der Attribute verschmilzt. Begründung: Attribute, Assoziationen und Operationen werden über ihre Klasse von anderen Modellelementen referenziert. So wird eine Operation immer über eine Instanz der Klasse, in der sie definiert ist, aufgerufen. Würde die Operation durch das Verschmelzen aus einer Klasse entfernt werden, wäre der Aufruf der Operation ungültig. Bei Attributen und Assoziationen würden Zugriffe auf sie ungültig werden.

Die zu verschmelzenden Attribute müssen von identischem Typ sein. Sie sind alle Klassenattribute oder alle Instanzattribute. Weitere Eigenschaften, die gleich sein müssen: Änderbarkeit (changeable, frozen, addOnly), Initialwert (sofern angegeben), Multiplizität und Ordnung.

Die zu verschmelzenden Assoziationen müssen auf identische Teilnehmer (Typ) verweisen. Sie sind entweder alle allgemein, definieren alle eine Aggregation oder alle eine Komposition. Weitere Eigenschaften, die gleich sein müssen: Änderbarkeit, Multiplizität, Navigierbarkeit und Ordnung.

Die zu verschmelzenden Operationen müssen die gleiche Parameterliste haben, d.h. identische Parametertypen und gleiche Datenflußrichtung (in, out, inout, result). Außerdem muß der Defaultwert pro Parameter gleich sein, sofern angegeben. Die Operationen sind alle Klassenoperationen oder alle Instanzoperationen, alle konkret oder alle abstrakt, alle (nicht) abgeleitet. Eine weitere Eigenschaften, die gleich sein muß, ist concurrency.

Die Bedingung nach Identität von Attributtypen, Assoziationsteilnehmern oder Parametertypen wird unter Einbeziehung aller Integrationsbeziehungen des Hypermoduls geprüft. Die Bedingung ist erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt, z.B. indem sie die Typen verschmilzt.

Nachbedingungen

Attribute: Initialwert Das Ziel-Attribut bleibt ohne Initialwert, wenn kein zu verschmelzendes Attribut einen Wert festlegt. Ansonsten übernimmt das Ziel-Attribut einen der (gleichen) Initialwerte.

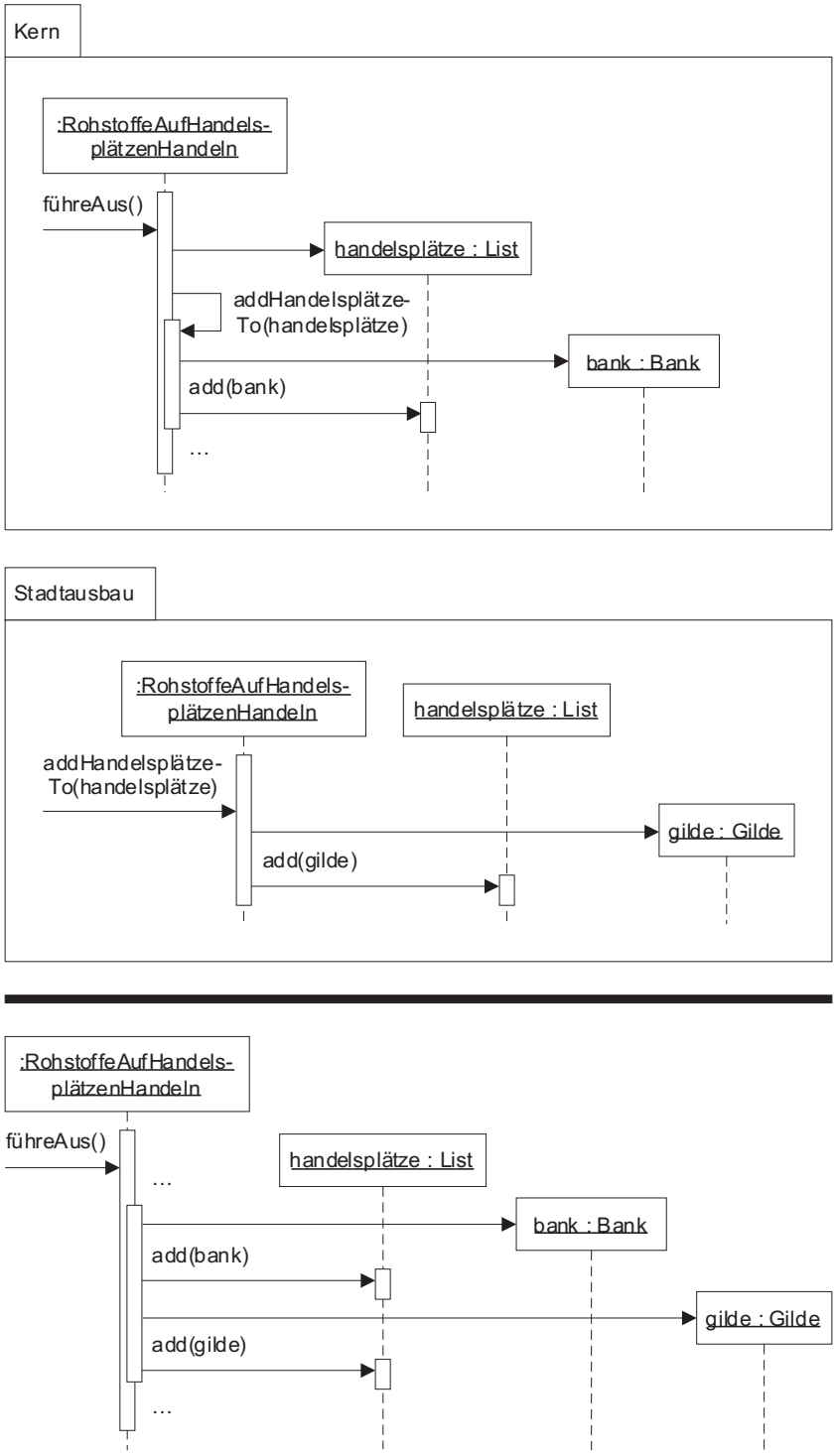

Abbildung 6.12: Verschmelzen von Attributen, Assoziationen und Operationen

Attribute/Assoziationen: Zugriffe Lesende oder schreibende Zugriffe auf Quell-Attribute greifen statt dessen auf das Ziel-Attribut zu. Die gleiche Semantik gilt für Zugriffe auf Quell-Assoziationen.

Operationen: Parameter Ein Parameter der Ziel-Operation bleibt ohne Defaultwert, wenn keine zu verschmelzende Operation für den Parameter einen Wert festlegt. Ansonsten übernimmt die Ziel-Operation einen der (gleichen) Defaultwerte.

... Implementierung Die Ziel-Operation enthält die Implementierungen der zu verschmelzenden Operationen, sofern vorhanden. Die Reihenfolge, in der Implementierungen beim Aufruf der Ziel-Operation ausgeführt werden, ist undefiniert beziehungsweise durch eine im Hypermodul angegebene Integrationsbeziehung Ordnen festgelegt (vgl. Unterkapitel 6.7). Die Reihenfolge wirkt sich auch auf die an jede Implementierung übergebenen Parameter aus. Die erste Implementierung erhält die Parameter so, wie sie der Aufrufer an die Operation übergibt. Die zweite Implementierung erhält die Parameter so, wie sie die erste Implementierung „hinterläßt". Und der Aufrufer der Operation bekommt immer den letzten Wert eines (in)out- beziehungsweise Rückgabeparameters zurück. Dieses Verhalten kann für Rückgabeparameter durch Angabe einer Integrationsbeziehung Summieren im Hypermodul geändert werden (vgl. Unterkapitel 6.8). Im Ergebnis des Beispiels führt addHandelsplätzeTo() zunächst die Implementierung von Kern aus, dann die von Stadtausbau.

... Aktionen Der Aufruf einer Operation findet immer dann statt, wenn zur Laufzeit des modellierten Systems eine entsprechende Aktion ausgeführt wird (CallAction). Aktionen treten (a) in Interaktionen als Nachrichten zwischen Objekten auf und (b) in Zustandsautomaten als *entry*-Aktion, *exit*-Aktion und Aktivität von Zuständen oder als Aktion von Transitionen. Aktionen, die eine Quell-Operation aufrufen, rufen statt dessen die Ziel-Operation auf.

... Ereignisse Der Aufruf einer Operation löst ein Ereignis aus (CallEvent), aufgrund dessen Transitionen in Zustandsautomaten schalten können. Ereignisse, die ein Aufruf einer Quell-Operation auslöst, löst statt dessen ein Aufruf der Ziel-Operation aus. Transitionen schalten entsprechend beim Aufruf der Ziel-Operation.

6.6.7 Semantik für Akteure und Use-Cases

Abbildung 6.13 zeigt die zu integrierenden Belange Kern und Entwicklungskarten mit ihren Akteuren und Use-Cases. Das Ergebnis der Integration präsentiert Abbildung 6.14. Der Ziel-Akteur Kern::Spieler am Zug wurde mit dem gleichnamigen Quell-Akteur aus Entwicklungskarten verschmolzen. Außerdem wurden die Ziel-Use-Cases Bösewicht versetzen und Spiel gewinnen aus Kern jeweils mit dem gleichnamigen Use-Case aus Entwicklungskarten verschmolzen.

Vorbedingungen

Die zu verschmelzenden Use-Cases sind alle abstrakt oder alle konkret.

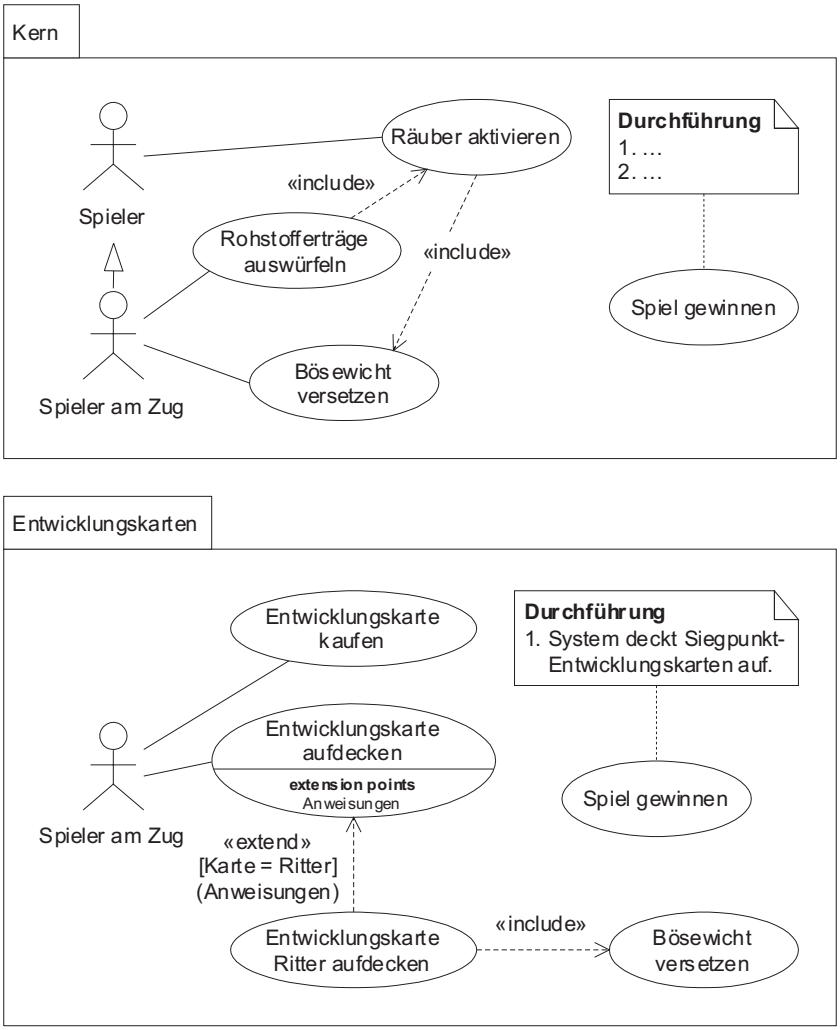

Abbildung 6.13: Verschmelzen von Akteuren und Use-Cases (i)

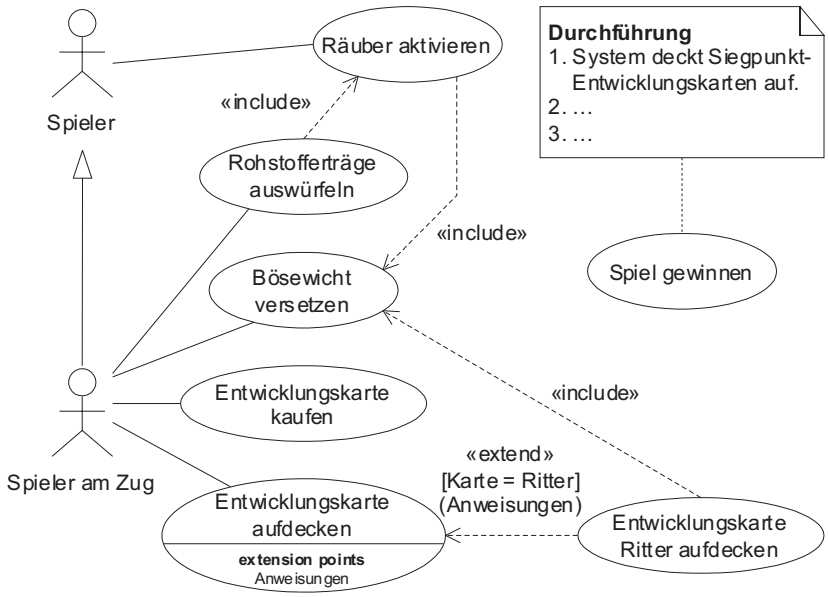

Abbildung 6.14: Verschmelzen von Akteuren und Use-Cases (ii)

Nachbedingungen

Akteure: Kommunikationsbeziehungen

Der Ziel-Akteur kommuniziert zusätzlich mit den Use-Cases, mit denen die Quell-Akteure kommunizieren. Use-Cases, die mit einem Quell-Akteur kommunizieren, kommunizieren statt dessen mit dem Ziel-Akteur. Redundante Kommunikationsbeziehungen wurden entfernt. Im Ergebnis des Beispiels kommunziert Spieler am Zug mit Entwicklungskarte kaufen und aufdecken, weil Entwicklungskarten dies spezifiziert.

Akteure/Use-Cases: Vererbungsbeziehungen

Der Ziel-Akteur erbt zusätzlich von den übergeordneten Akteuren der Quell-Akteure. Untergeordnete Akteure der Quell-Akteure erben statt dessen vom Ziel-Akteur. Redundante Vererbungsbeziehungen wurden entfernt. Die gleiche Semantik gilt für Use-Cases.

Use-Cases: Use-Case-Beschreibung

Der Ziel-Use-Case enthält zusätzlich die Use-Case-Beschreibungen der Quell-Use-Cases, sofern vorhanden. Die UML standardisiert das Format von Use-Case-Beschreibungen nicht. In der Sekundärliteratur werden jedoch verschiedene Formate vorgeschlagen (z.B. [Coc97] und [Oes99, Seite 205ff]). Danach besteht eine Use-Case-Beschreibung wenigstens aus (vgl. Abschnitt 4.2.3):

1. *Vorbedingungen.* Der Ziel-Use-Case enthält zusätzlich die Vorbedingungen der Quell-Use-Cases. Dies entspricht einer UND-Verknüpfung der Vorbedingungen.
2. *Durchführung.* Der Ziel-Use-Case enthält zusätzlich die Durchführungen der Quell-Use-Cases. Die Reihenfolge der Durchführungen ist undefiniert beziehungsweise durch eine im Hypermodul angegebene Integrationsbeziehung Ordnen festgelegt (vgl. Unterkapitel 6.7). Die Reihenfolge der

Schritte innerhalb der Durchführungen bleibt erhalten. Im Ergebnis des Beispiels stammt der erste Schritt in der Durchführung von Spiel gewinnen aus Entwicklungskarten.

3. *Nachbedingungen.* Für die Nachbedingungen gilt die gleiche Semantik wie für die Vorbedingungen.

4. *Ausnahmen.* Der Ziel-Use-Case enthält zusätzlich alle Ausnahmen der Quell-Use-Cases. Die Verweise der Ausnahmen auf bestimmte Schritte in der Durchführung bleiben erhalten. Sie sind an die neue Numerierung der Schritte angepaßt.

... *Erweiterungsstellen*

Der Ziel-Use-Case enthält zusätzlich alle Erweiterungsstellen der Quell-Use-Cases. Redundante Erweiterungsstellen wurden entfernt. Erweiterungsstellen sind redundant, wenn sie den gleichen Namen tragen und auf identische Lokationen (oder keine Lokation) innerhalb des Ziel-Use-Case verweisen. Die Identität von Lokationen ist in der vorliegenden Version der UML jedoch nicht prüfbar, weil die Spezifikation nur eine vage Aussage dazu trifft, wie Lokationen anzugeben sind: „May be a line or range of lines in code, or a state or set of states in a state machine, or some other means in a different kind of specification." [OMG01, Abschnitt 2.7.2.12] Deshalb sind im Ziel-Use-Case die Lokationen der Erweiterungsstellen gelöscht, und die Redundanzprüfung reduziert sich auf das Vergleichen der Namen der Erweiterungsstellen. Bei Erscheinen der nächsten Version der UML-Spezifikation ist diese Entscheidung zu überprüfen.

... *extend-Beziehungen*

Der Ziel-Use-Case erweitert zusätzlich diejenigen Use-Cases, zu denen die Quell-Use-Cases *extend*-Beziehungen unterhalten. Use-Cases, die Quell-Use-Cases erweitern, erweitern statt dessen den Ziel-Use-Case. Entstehen dadurch gleiche *extend*-Beziehungen (erweiterter Use-Case, erweiternder Use-Case und Erweiterungsstelle sind identisch), verschmelzen diese zu einer Beziehung. Dabei werden die Bedingungen, unter denen die Erweiterungen stattfinden, UND-verknüpft. Eine zukünftige Erweiterung von Hyper/UML ist die wahlweise ODER-Verknüpfung der Bedingungen.

... *include-Beziehungen*

Der Ziel-Use-Case schließt zusätzlich diejenigen Use-Cases ein, zu denen die Quell-Use-Cases *include*-Beziehungen unterhalten. Use-Cases, die Quell-Use-Cases einschließen, schließen statt dessen den Ziel-Use-Case ein. Redundante *include*-Beziehungen wurden entfernt. Im Ergebnis des Beispiels schließt der Use-Case Entwicklungskarte Ritter aufdecken den Ziel-Use-Case Bösewicht versetzen aus Kern ein.

... *Aktivitätsgraphen*

Der Ziel-Use-Case enthält zusätzlich die Aktivitätsgraphen der Quell-Use-Cases. Übereinstimmende Aktivitätsgraphen (vgl. Operation 8 auf Seite 106) werden miteinander verschmolzen (vgl. Abschnitt 6.6.10).

Ausnahmen

Die UML verbietet Zyklen in den modellierten Vererbungshierarchien [OMG01, Abschnitt 2.5.3.20, Regel 3]. Entstehen durch das Verschmelzen von Akteuren oder Use-Cases Zyklen in der Vererbungshierarchie, bricht die Integration mit einem Fehler ab.

Stehen die Vorbedingungen oder Nachbedingungen einer Use-Case-Beschreibung nach dem Verschmelzen im Widerspruch zueinander, kann dies wegen des informalen Charakters der Vor- und Nachbedingungen nicht erkannt werden.

Die UML erlaubt nicht, daß Use-Cases Erweiterungsstellen definieren, die unter dem gleichen Namen bereits ein übergeordneter Use-Case definiert [OMG01, Abschnitt 2.11.3.5, Regel 4]. Tritt dieser Fall durch das Verschmelzen ein, bricht die Integration mit einem Fehler ab.

UML-Modelle sind ungültig, wenn Bedingungen von *extend*-Beziehungen sich widersprechen. Stehen nach dem Verschmelzen die Bedingungen von *extend*-Beziehungen im Widerspruch zueinander, bricht die Integration mit einem Fehler ab. Die Widerspruchsfreiheit von Bedingungen kann nur geprüft werden, wenn die Bedingungen in einer formalen Sprache, z.B. OCL, formuliert sind.

6.6.8 Semantik für Zustandsautomaten und Zustände

Abbildung 6.15 zeigt die zu integrierenden Belange Kern, Nacheinander Handeln und Bauen, Schiffe und Fortschrittskarten mit ihrem Zustandsautomat Besiedlungsphasenzug. Das Ergebnis der Integration präsentiert Abbildung 6.16. Der Ziel-Zustandsautomat aus Kern wurde mit den Quell-Zustandsautomaten aus den anderen drei Belangen zu einem einzigen Automaten verschmolzen.

Vorbedingungen

Ein Zustandsautomat definiert Zustände, die Instanzen seines Kontextes (eine Klasse) annehmen können. Damit ist, von der Semantik her, das Verschmelzen von Zustandsautomaten über verschiedene Kontexte hinweg problematisch. Deshalb müssen die zu verschmelzenden Automaten alle einem identischen Kontext oder alle keinem Kontext zugeordnet werden. Diese Bedingung wird unter Einbeziehung aller Integrationsbeziehungen des Hypermoduls geprüft. Sie ist erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt, z.B. indem sie die Kontexte der Zustandsautomaten verschmilzt.

Ein Zustand befindet sich in einer Hierarchie von Zuständen, deren oberste Ebene ein Zustandsautomat einnimmt. Ähnlich wie bei Zustandsautomaten ist das Verschmelzen von Zuständen über verschiedene Hierarchieebenen hinweg semantisch problematisch. Deshalb auch hier die Konsequenz: Die zu verschmelzenden Zustände müssen in einer identischen Hierarchieebene stehen. Diese Bedingung wird ebenfalls in Bezug zum Hypermodul geprüft.

Den zu verschmelzenden Zuständen ist die gleiche *entry*-Aktion, *exit*-Aktion und Aktivität zugeordnet (oder keine).

Nachbedingungen

Zustands-automaten

Der Ziel-Zustandsautomat enthält zusätzlich die in den Quell-Zustandsautomaten definierten Zustände. Übereinstimmende Zustände (vgl. Operation 8 auf Seite 106) verschmelzen miteinander. Im Beispiel tritt in allen Zustandsautomaten der Zustand Besiedlungsphasenzug läuft auf. Diese Zustände verschmelzen

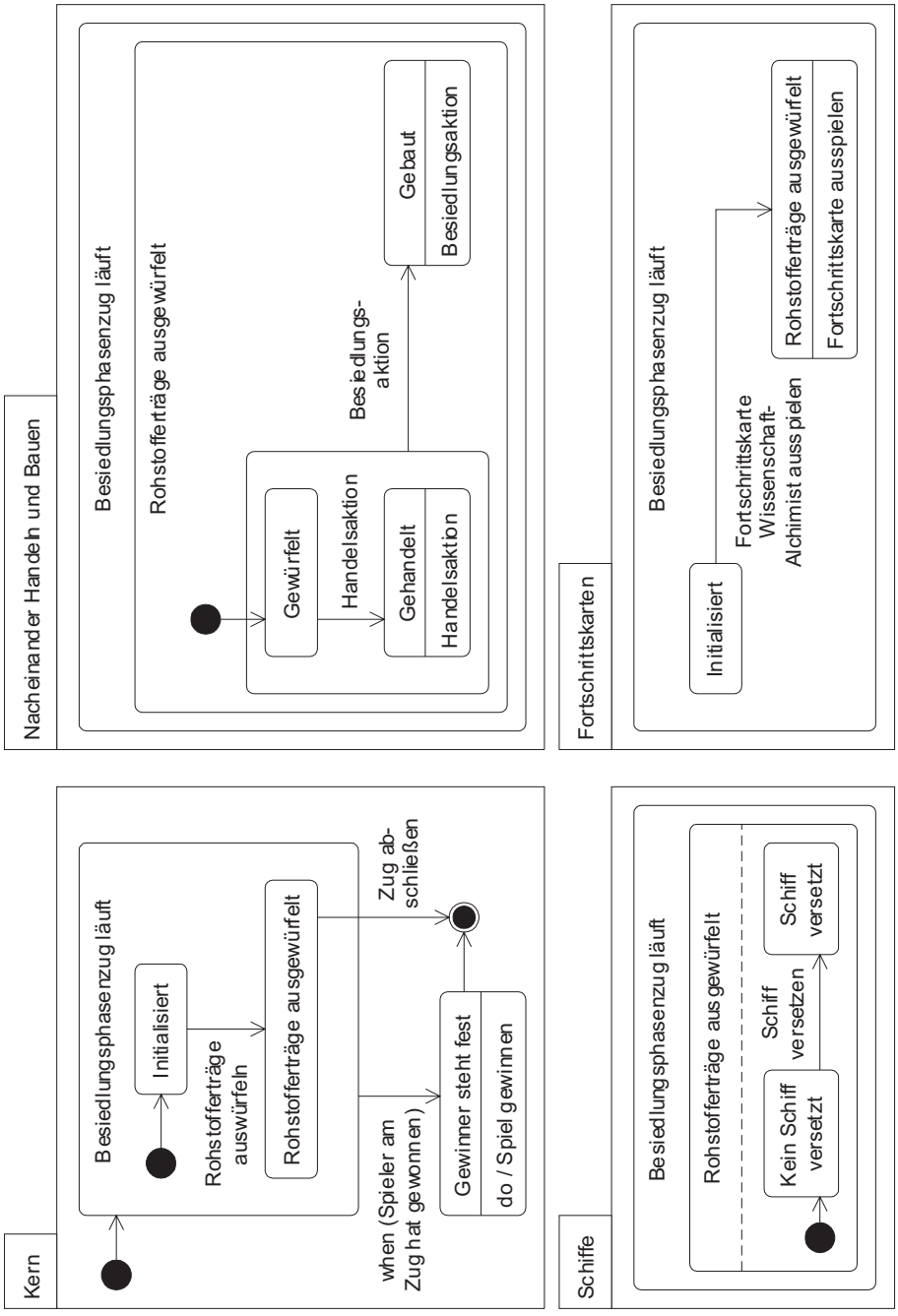

Abbildung 6.15: Verschmelzen von Zustandsautomaten und Zuständen (i)

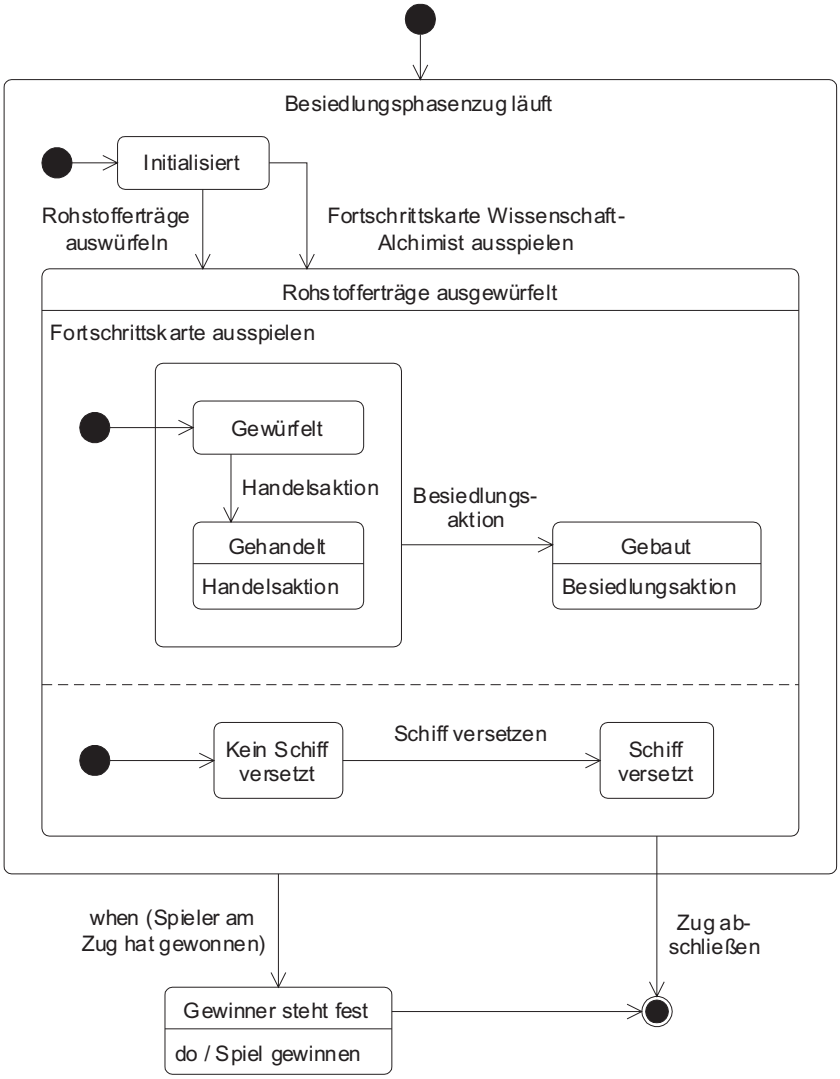

Abbildung 6.16: Verschmelzen von Zustandsautomaten und Zuständen (ii)

im Ergebnis zu einem Zustand. Dann verschmelzen ihre enthaltenen Zustände Initialisiert (Kern und Fortschrittskarten) sowie Rohstofferträge ausgewürfelt (alle Belange).

Zustände: entry-/exit-Aktion, Aktivität

Dem Ziel-Zustand ist die (gleiche) *entry*-Aktion der zu verschmelzenden Zustände zugeordnet, sofern dort eine definiert ist. Die gleiche Semantik gilt für die *exit*-Aktion und die Aktivität.

... aufschiebbare Ereignisse

Der Ziel-Zustand enthält zusätzlich die Liste der aufschiebbaren Ereignisse der Quell-Zustände. Redundante Ereignisse wurden aus der Liste entfernt.

... Transitionen

Aus „äußeren" eingehenden Transitionen der Quell-Zustände sind eingehende Transitionen des Ziel-Zustands geworden. Aus ausgehenden Transitionen der Quell-Zustände sind ausgehende Transitionen des Ziel-Zustands geworden. Der Ziel-Zustand enthält zusätzlich die inneren Transitionen der Quell-Zustände. Im Ergebnis des Beispiels führt die Transition Fortschrittskarte Wissenschaft-Alchimist ausspielen vom Zustand Initialisiert zu Rohstofferträge ausgewürfelt, weil Fortschrittskarten dies spezifiziert. Ebenfalls aus Fortschrittskarten stammt die innere Transition Fortschrittskarte ausspielen in Rohstofferträge ausgewürfelt.

... History-Zustände

Der Ziel-Zustand ist mit einem flachen History-Zustand ausgestattet, wenn mindestens ein zu verschmelzender Zustand mit einem flachen History-Zustand ausgestattet ist. Transitionen, die zu flachen History-Zuständen der zu verschmelzenden Zustände führen beziehungsweise von ihnen ausgehen, führen in den flachen History-Zustand des Ziel-Zustands beziehungsweise gehen von ihm aus. Die gleiche Semantik gilt für tiefe History-Zustände.

... ODER-Verfeinerung

Der Ziel-Zustand ist ODER-verfeinert, wenn mindestens ein zu verschmelzender Zustand ODER-verfeinert, aber keiner UND-verfeinert ist. Der Ziel-Zustand enthält zusätzlich die in den ODER-verfeinerten Quell-Zuständen enthaltenen Zustände. Übereinstimmende enthaltene Zustände verschmelzen miteinander.

... UND-Verfeinerung

Der Ziel-Zustand ist UND-verfeinert, wenn mindestens ein zu verschmelzender Zustand UND-verfeinert ist. Der Ziel-Zustand enthält die UND-Verfeinerungen der zu verschmelzenden Zustände. Übereinstimmende UND-Verfeinerungen verschmelzen miteinander. Sind ein oder mehrere zu verschmelzende Zustände ODER-verfeinert, enthält der Ziel-Zustand zusätzlich eine weitere UND-Verfeinerung mit den Zuständen, die die zu verschmelzenden ODER-verfeinerten Zustände enthalten. Übereinstimmende enthaltene Zustände verschmelzen miteinander. Im Ergebnis des Beispiels ist der Zustand Rohstofferträge ausgewürfelt UND-verfeinert, weil Schiffe eine UND-Verfeinerung spezifiziert. Die obere Region von Rohstofferträge ausgewürfelt ergibt sich aus der ODER-Verfeinerung von Nacheinander Handeln und Bauen. Die untere Region aus der UND-Verfeinerung von Schiffe.

... Start- und Endzustand

Der Ziel-Zustand besitzt einen Startzustand, wenn ein zu verschmelzender Zustand einen Startzustand besitzt. Die von den Startzuständen der zu verschmelzenden Zustände ausgehenden Transitionen gehen vom Startzustand des Ziel-Zustands aus. Die gleiche Semantik gilt für den Endzustand, nur führen die Transitionen zum Endzustand des Ziel-Zustands hin.

... Pseudozustände

Der Ziel-Zustand enthält zusätzlich die Entscheidungsknoten, Verbindungsstellen, Gabelungen, Vereinigungen und Synch-Zustände, die die Quell-Zustände enthalten. Diese sogenannten Pseudozustände verschmelzen nicht miteinander.

... Referenzen Zustände stehen in UML-Modellen nicht isoliert dar, sondern werden von anderen Modellelementen referenziert. Beispielsweise können Aktivitätsgraphen in Objektflüssen Instanzen von Klassen enthalten, die einen Quell-Zustand angenommen haben. Solche Instanzen befinden sich nach dem Verschmelzen im Ziel-Zustand.

Ausnahmen

Entstehen durch das Verschmelzen redundante (innere) Transitionen, werden sie entfernt. Eine Transition ist redundant, wenn eine zweite Transition dieselben Zustände verbindet (ist bei inneren Transitionen immer der Fall), durch das gleiche Ereignis ausgelöst wird und beim Auslösen die gleiche Aktion ausführt. Die Überwachungsbedingungen der entfernten Transitionen werden mit der Überwachungsbedingung der übrig gebliebenen Transition UND-verknüpft. Ist die verknüpfte Überwachungsbedingung widersprüchlich, dann ist das UML-Modell im Hypermodul ungültig, und die Integration bricht mit einem Fehler ab. Die Widerspruchsfreiheit von Überwachungsbedingungen kann nur geprüft werden, wenn die Bedingungen in einer formalen Sprache, z.B. OCL, formuliert sind. Eine zukünftige Erweiterung von Hyper/UML ist die wahlweise ODER-Verknüpfung von Überwachungsbedingungen.

6.6.9 Semantik für Signale

Abbildung 6.17 zeigt im oberen Teil die zu integrierenden Belange Kern und Stadtausbau mit ihren Signalen. Der untere Teil präsentiert das Ergebnis der Integration. Das Ziel-Signal Kern:Besiedlungsaktion wurde mit dem gleichnamigen Quell-Signal aus Stadtausbau verschmolzen.

Nachbedingungen

Parameter Das Ziel-Signal enthält zusätzlich die in den Quell-Signalen definierten Parameter. Redundante Parameter wurden entfernt. Transitionen in Zustandsautomaten, die beim Auslösen eines Quell-Signals schalten (SignalEvent), schalten statt dessen beim Auslösen des Ziel-Signals.

Vererbungsbeziehungen Das Ziel-Signal erbt zusätzlich von den übergeordneten Signalen der Quell-Signale. Untergeordnete Signale der Quell-Signale erben statt dessen vom Ziel-Signal. Redundante Vererbungsbeziehungen wurden entfernt. Parameter, die über die gesamte Vererbungshierarchie des Ziel-Signals gesehen redundant definiert waren, wurden (aus den untergeordneten Signalen) entfernt. Im Ergebnis des Beispiels wurde die Signalhierarchie unterhalb von Kern::Besiedlungsaktion mit der Hierarchie von Stadtausbau::Besiedlungsaktion zusammengeführt.

Ausnahmen

Die UML verbietet Zyklen in den modellierten Vererbungshierarchien [OMG01, Abschnitt 2.5.3.20, Regel 3]. Entstehen durch das Verschmelzen von Signalen Zyklen in der Vererbungshierarchie, bricht die Integration mit einem Fehler ab.

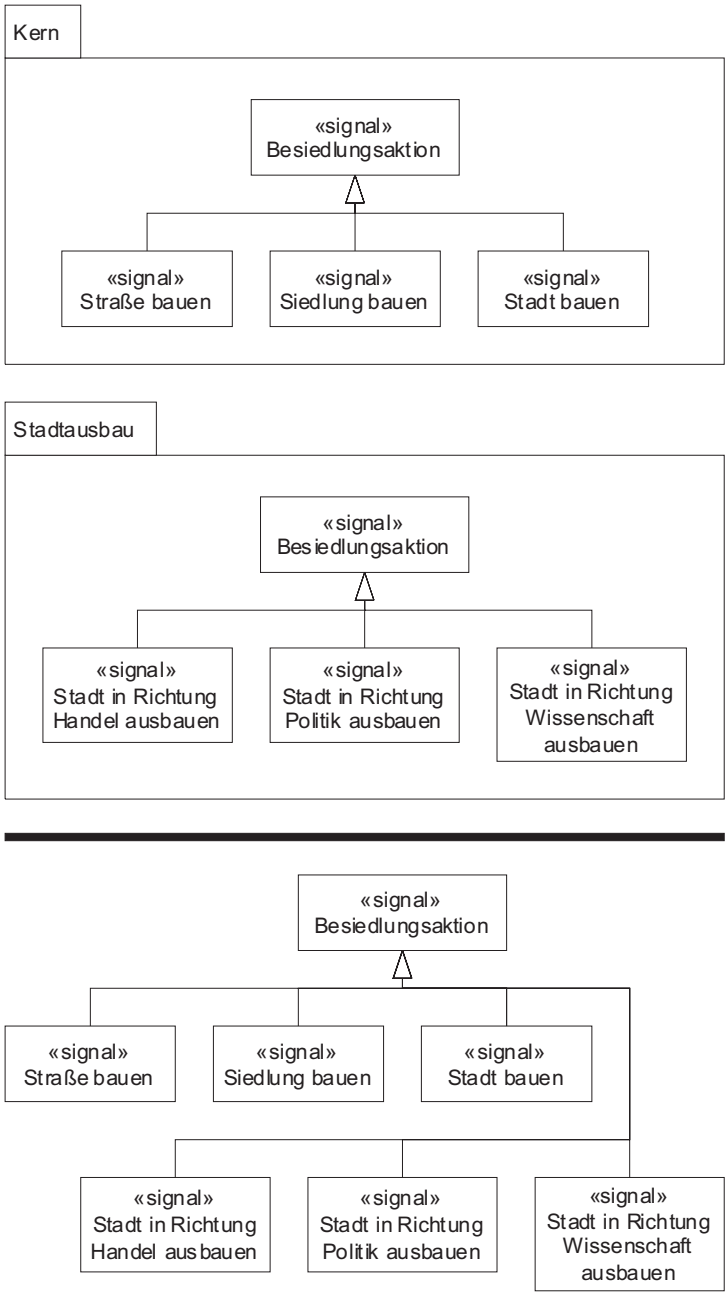

Abbildung 6.17: Verschmelzen von Signalen

6.6.10 Semantik für Aktivitätsgraphen und -zustände

Abbildung 6.18 zeigt die zu integrierenden Belange Kern, Ereigniswürfel und Barbaren mit ihrem Aktivitätsgraph Rohstofferträge auswürfeln. Das Ergebnis der Integration präsentiert Abbildung 6.19. Der Ziel-Aktivitätsgraph aus Kern wurde mit dem Quell-Aktivitätsgraph aus Ereigniswürfel verschmolzen. Außerdem wurde der Ziel-Aktivitätszustand Schiff gewürfelt mit dem Quell-Aktivitätszustand Barbaren vorrücken verschmolzen.

Vorbedingungen

Ein Aktivitätsgraph beschreibt den Ablauf seines Kontextes (eines Use-Case) als Folge von Aktivitätszuständen. Damit ist, von der Semantik her, das Verschmelzen von Aktivitätsgraphen beziehungsweise Aktivitätszuständen über verschiedene Kontexte hinweg problematisch. Deshalb müssen die zu verschmelzenden Graphen alle einem identischen Kontext oder alle keinem Kontext zugeordnet werden; zu verschmelzende Aktivitätszustände müssen alle zu einem identischen Graphen gehören. Diese Bedingung wird unter Einbeziehung aller Integrationsbeziehungen des Hypermoduls geprüft. Sie ist erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt, z.B. indem sie die Kontexte der Aktivitätsgraphen verschmilzt.

Nachbedingungen

Aktivitätsgraphen: Abläufe

Der Ziel-Aktivitätsgraph beschreibt zusätzlich die Abläufe, die die Quell-Aktivitätsgraphen beschreiben. Deshalb enthält der Ziel-Aktivitätsgraph zusätzlich die Aktivitätszustände, Transitionen, Entscheidungsknoten, Verbindungsstellen, Gabelungen, Vereinigungen, Synch-Zustände und Objektflüsse der Quell-Aktivitätsgraphen. Die Abläufe sind über ihren Startzustand und ihre Endzustände verknüpft: Der Ziel-Aktivitätsgraph beginnt mit dem Startzustand des ersten Ablaufs. Transitionen, die zu Endzuständen des ersten Ablaufs hinführen, führen statt dessen zum Startaktivitätszustand des zweiten Ablaufs, also zu dem Aktivitätszustand, zu dem der Startzustand des zweiten Ablaufs hinführt ... Der Ziel-Aktivitätsgraph endet mit den Endzuständen des letzten Ablaufs. Die Reihenfolge der Abläufe ist undefiniert beziehungsweise durch eine im Hypermodul angegebene Integrationsbeziehung Ordnen festgelegt (vgl. Unterkapitel 6.7). Im Ergebnis des Beispiels wurde der Ablauf des Aktivitätsgraphen aus Ereigniswürfel vor den aus Kern gehängt. Die Endzustände in Ereigniswürfel führen deshalb zum Startaktivitätszustand Kern::1. und 2. Augenwürfel würfeln.

... Verantwortlichkeitsbereiche

Der Ziel-Aktivitätsgraph enthält zusätzlich die Verantwortlichkeitsbereiche beziehungsweise Schwimmbahnen der Quell-Aktivitätsgraphen. Redundante Bereiche wurden entfernt. Die Zuordnung der Aktivitätszustände zu den Bereichen bleibt erhalten.

... Referenzen

Aktivitätszustände können anstelle ihrer *entry*-Aktion auch einen Aktivitätsgraphen referenzieren, der ihre Aktivität beschreibt. Aktivitätszustände mit einer Referenz auf einen Quell-Aktivitätsgraph referenzieren statt dessen den Ziel-Aktivitätsgraph.

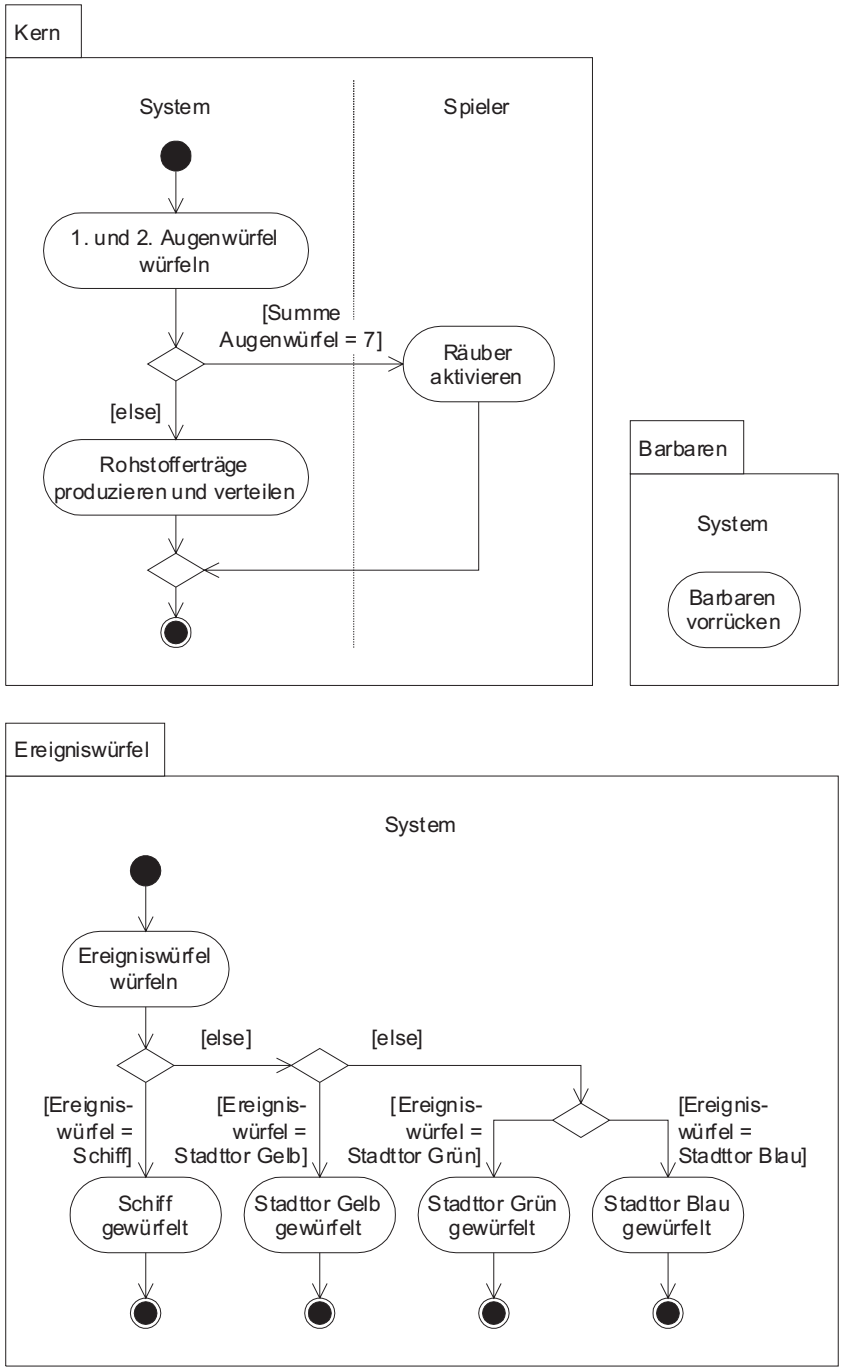

Abbildung 6.18: Verschmelzen von Aktivitätsgraphen und -zuständen (i)

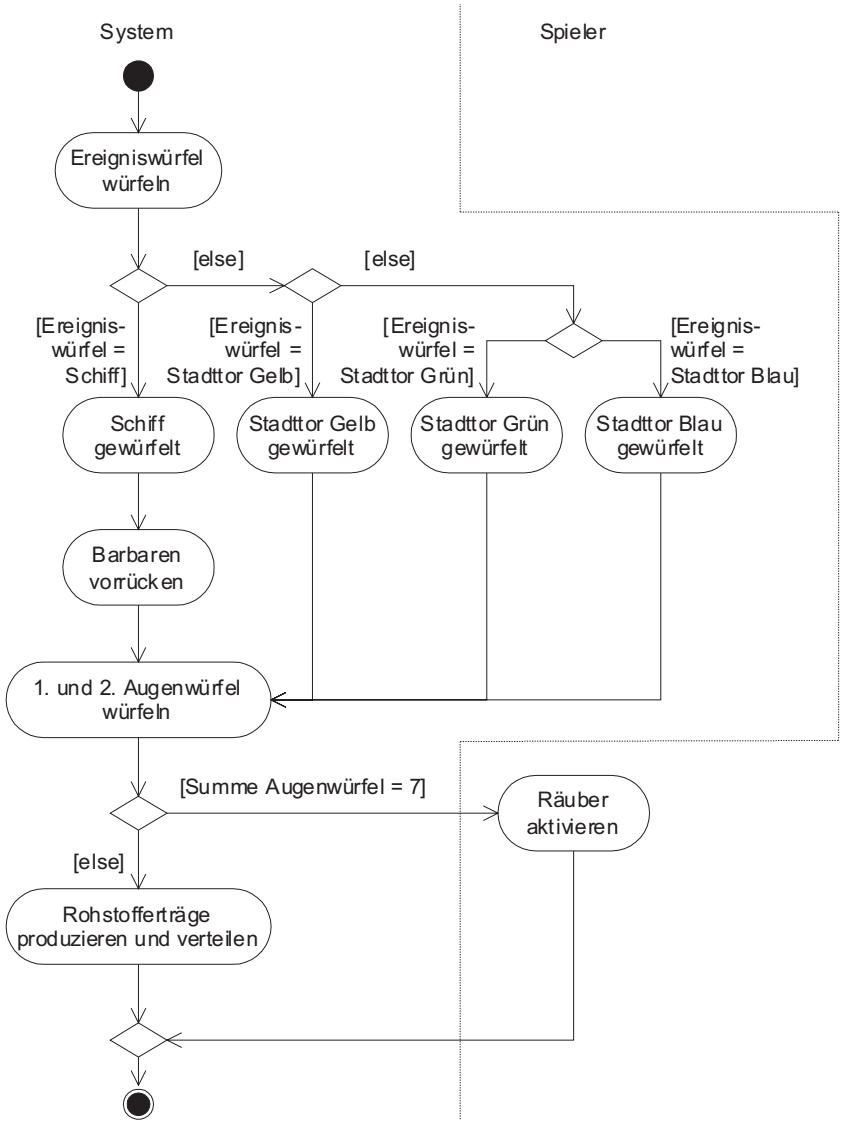

Abbildung 6.19: Verschmelzen von Aktivitätsgraphen und -zuständen (ii)

Aktivitäts-zustände

Die zu verschmelzenden Aktivitätszustände werden zunächst in undefinierter Reihenfolge im Aktivitätsgraphen angeordnet und über Transitionen verknüpft. Transitionen, die in die zu verschmelzenden Aktivitätszustände eingehen, führen statt dessen zum ersten Aktivitätszustand in der Reihenfolge hin. Transitionen, die von den zu verschmelzenden Aktivitätszuständen ausgehen, gehen vom letzten Aktivitätszustand in der Reihenfolge aus. Wenn notwendig, kann die Reihenfolge der Aktivitätszustände im Hypermodul durch Angabe einer Integrationsbeziehung Ordnen festgelegt werden (vgl. Unterkapitel 6.7). Die Objektflüsse der zu verschmelzenden Aktivitätszustände bleiben unverändert. Im Ergebnis des Beispiels wurde der Aktivitätszustand Barbaren::Barbaren vorrücken vor den Zustand Ereigniswürfel::Schiff gewürfelt gehängt. Die eingehende Transition von Schiff gewürfelt führt deshalb zu Barbaren vorrücken, die ausgehende Transition von Schiff gewürfelt geht jetzt von Barbaren vorrücken aus.

Ausnahmen

Definiert ein zu verschmelzender Aktivitätsgraph keinen Start- oder Endzustand, werden seine Aktivitätszustände (sowie Entscheidungsknoten, Verbindungsstellen, Gabelungen, Vereinigungen, Synch-Zustände und Objektflüsse) zwar in den Ziel-Aktivitätsgraph übernommen, aber nicht mit den Abläufen der anderen zu verschmelzenden Graphen verknüpft. Die Verknüpfung muß dann durch weitere Integrationsbeziehungen erfolgen.

Enthält der Ziel-Aktivitätsgraph nach dem Verschmelzen Aktivitätszustände mit gleichem Namen, so werden überzählige entfernt. Dieser Automatismus kann durch Angabe einer Integrationsbeziehung Ersetzen abgeschaltet werden (vgl. Unterkapitel 6.9). Die in die entfernten Aktivitätszustände ein- und ausgehenden Transitionen werden zu ein- beziehungsweise ausgehenden Transitionen des verbliebenen Aktivitätszustands. Die folgenden Eigenschaften müssen bei gleichnamigen Aktivitätszuständen ebenfalls gleich sein: Verantwortlichkeitsbereich, *entry*-Aktion oder Referenz auf (identischen) Aktivitätsgraphen, isDynamic, dynamicMultiplicity und dynamicArguments. Andernfalls bricht die Integration mit einem Fehler ab.

Entstehen durch das Verschmelzen redundante Transitionen, werden sie entfernt. Eine Transition ist redundant, wenn eine zweite Transition dieselben (Aktivitäts-)Zustände verbindet und beim Auslösen die gleiche Aktion ausführt. Die Überwachungsbedingungen der entfernten Transitionen werden mit der Überwachungsbedingung der übrig gebliebenen Transition UND-verknüpft. Ist die verknüpfte Überwachungsbedingung widersprüchlich, dann ist das UML-Modell im Hypermodul ungültig, und die Integration bricht mit einem Fehler ab. Die Widerspruchsfreiheit von Überwachungsbedingungen kann nur geprüft werden, wenn die Bedingungen in einer formalen Sprache, z.B. OCL, formuliert sind. Eine zukünftige Erweiterung von Hyper/UML ist die wahlweise ODER-Verknüpfung von Überwachungsbedingungen.

6.7 Integrationsbeziehung Ordnen

Die Integrationsbeziehung Ordnen (engl.: *order relationship*) gibt an, in welcher Reihenfolge Elemente gleichen Typs verschmolzen werden.

6.7.1 Syntax

Die Umsetzung der Integrationsbeziehung Ordnen in eine grafische Syntax zeigt Abbildung 6.20. Eine Integrationsbeziehung Ordnen ist mit den zu ordnenden Elementen assoziiert. Die Assoziation ist geordnet, legt also gleichzeitig die Reihenfolge der Elemente beim Verschmelzen fest. Widersprechen sich die im Hypermodul angegebenen Integrationsbeziehungen Ordnen bezüglich der Reihenfolge von Elementen, bricht die Integration mit einem Fehler ab.

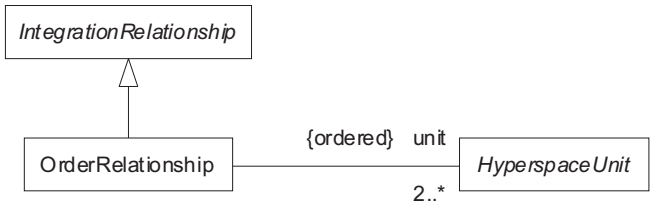

Abbildung 6.20: Profil Hyper/UML – Integrationsbeziehung Ordnen

6.7.2 Regeln

Die folgenden Regeln vervollständigen die Syntax-Definition der Integrationsbeziehung Ordnen.

OrderRelationship

Regel 11: Die zu ordnenden Elemente sind nicht mehrfach in der Integrationsbeziehung angegeben, da dann die Reihenfolge der Elemente nicht eindeutig bestimmbar wäre.

```
self.unit = self.unit->asSet()
```

Regel 12: Die zu ordnenden Elemente sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.unit->forAll( u |
   self.hypermodule.concern->exists( c |
      c.includesUnit(u)))
```

Regel 13: Die zu ordnenden Elemente sind vom aufgezählten Typ. Hier sind nur Operationen, Use-Cases, Aktivitätsgraphen und -zustände erlaubt. Das Ordnen anderer Elemente hätte keine semantische Bedeutung.

```
self.unit->forAll( u |
   Set{
      Operation,
      UseCase,
      ActivityGraph, ActionState, SubactivityState
   }->includes(u.oclType()))
```

Regel 14: Die zu ordnenden Elemente sind vom selben Typ. Elemente verschiedenen Typs können semantisch nicht angeordnet werden.

```
self.unit->forAll( u1, u2 |
   (u1.oclType() = u2.oclType()) and
   (u1.stereotype = u2.stereotype))
```

6.7.3 Semantik

Die Semantik der Integrationsbeziehung Ordnen hängt vom Typ der zu ordnenden Elemente ab.

Nachbedingungen

Operationen

Verschmelzen zu ordnende Operationen in einer Ziel-Operation (vgl. Abschnitt 6.6.6), führt die Ziel-Operation die Implementierungen der zu ordnenden Operationen in der angegebenen Reihenfolge aus.

Use-Cases

Verschmelzen zu ordnende Use-Cases in einem Ziel-Use-Case (vgl. Abschnitt 6.6.7), enthält der Ziel-Use-Case die Durchführungen aus der Use-Case-Beschreibung der zu ordnenden Use-Cases in der angegebenen Reihenfolge. Verschmelzen die Aktivitätsgraphen zu ordnender Use-Cases, überträgt sich die Ordnung der Use-Cases auf die Aktivitätsgraphen.

Aktivitätsgraphen

Verschmelzen zu ordnende Aktivitätsgraphen in einem Ziel-Aktivitätsgraph (vgl. Abschnitt 6.6.10), laufen im Ziel-Aktivitätsgraph die Abläufe der zu ordnenden Graphen in der angegebenen Reihenfolge ab.

Aktivitätszustände

Verschmelzen zu ordnende Aktivitätszustände (vgl. Abschnitt 6.6.10), sind sie in der angegebenen Reihenfolge verknüpft.

6.8 Integrationsbeziehung Summieren

Die Integrationsbeziehung Summieren (engl.: *summary function relationship*) gibt an, wie aus den Rückgabewerten der in einer Operation verschmolzenen Implementierungen ein gemeinsamer Wert zu bilden ist, den Aufrufer der Operation erhalten.

6.8.1 Syntax

Die Umsetzung der Integrationsbeziehung Summieren in eine grafische Syntax zeigt Abbildung 6.21. Das zu summierende Element (summarizedUnit) bezeichnet die Operation mit den verschmolzenen Implementierungen. Die Summenfunktion (summaryFunction) bildet den gemeinsamen Rückgabewert.

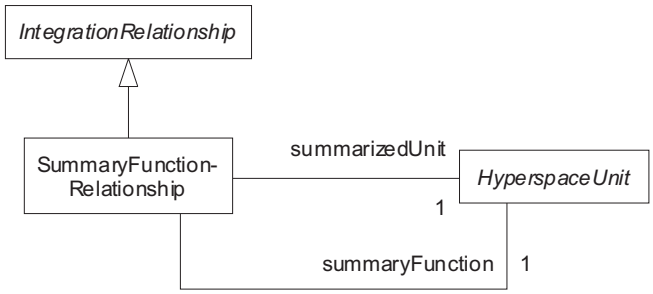

Abbildung 6.21: Profil Hyper/UML – Integrationsbeziehung Summieren

6.8.2 Regeln

Die folgenden Regeln und Operationen vervollständigen die Syntax-Definition der Integrationsbeziehung Summieren. Weitere Regeln für die zu verschmelzenden Elemente legen die nachfolgenden Abschnitte zur Semantik in Form von Vorbedingungen fest.

SummaryFunctionRelationship

Regel 15: Das zu summierende Element und die Summenfunktion sind nicht identisch. Das verhindert eine Endlosschleife ausgelöst durch rekursives Summieren.

```
self.summarizedUnit <> self.summaryFunction
```

Regel 16: Das zu summierende Element und die Summenfunktion sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.units()->forAll( u |
   self.hypermodule.concern->exists( c |
      c.includesUnit(u)))
```

Regel 17: Das zu summierende Element und die Summenfunktion sind Operationen. Das Summieren anderer Elemente hätte keine semantische Bedeutung.

```
self.units()->forAll( u | u.isOclTypeOf(Operation))
```

Operation 10: units() liefert das zu summierende Element und die Summenfunktion zurück. Die obigen Regeln verwenden diese Operation.

```
units(): Set(HyperspaceUnit);
units =
   self.summarizedUnit->union(self.summaryFunction)
```

6.8.3 Semantik

Abbildung 6.22 zeigt die zu integrierenden Belange Kern und Seeräuber mit ihren Klassen und den Implementierungen der Operation Spiel::getBösewichter(). Das Ergebnis der Integration präsentiert Abbildung 6.23. Die Ziel-Klasse Kern::Spieler wurde mit der gleichnamigen Quell-Klasse aus Seeräuber verschmolzen. Dabei wurden die übereinstimmenden getBösewichter()-Operationen verschmolzen, und zwar so, daß die Summenfunktion SummaryFunctions::listAddAll() die Rückgabewerte der Operationen summiert.

Vorbedingungen

Die zu summierende Operation und die Summenfunktion sind konkrete Operationen mit einem Rückgabeparameter von identischem Typ. Die Summenfunktion ist eine Klassenoperation; andernfalls müßte zur Laufzeit erst die Klasse mit der Summenfunktion instanziiert werden, damit die Summenfunktion aufgerufen werden kann.

Die Summenfunktion hat einen in-Parameter, dessen Typ identisch ist mit dem des Rückgabeparameters. Die Multiplizität des in-Parameters ist „*“, damit ihm eine Liste mit den Rückgabewerten der verschmolzenen Implementierungen übergeben werden kann.

Die Bedingung nach Identität des Typs der Rückgabeparameter wird unter Einbeziehung aller Integrationsbeziehungen des Hypermoduls geprüft. Die Bedingung ist erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt, z.B. indem sie die Typen verschmilzt.

Nachbedingungen

Während ihrer Ausführung sammelt die zu summierende Operation die Rückgabewerte der in ihr verschmolzenen Implementierungen (vgl. Abschnitt 6.6.6) in einer Liste. Bevor der Rücksprung an den Aufrufer erfolgt, ruft die zu summierende Operation die Summenfunktion auf und übergibt ihr die Rückgabewerte. Die Summenfunktion bildet daraus einen gemeinsamen Rückgabewert, den die zu summierende Operation dann an ihren Aufrufer zurückliefert. Im Ergebnis des Beispiels bildet die Summenfunktion listAddAll() aus den zwei Listen b1 und b2 eine Liste bösewichter, die alle Elemente aus b1 und b2 enthält (hier räuber und seeräuber) und an Aufrufer von getBösewichter() zurückgegeben wird.

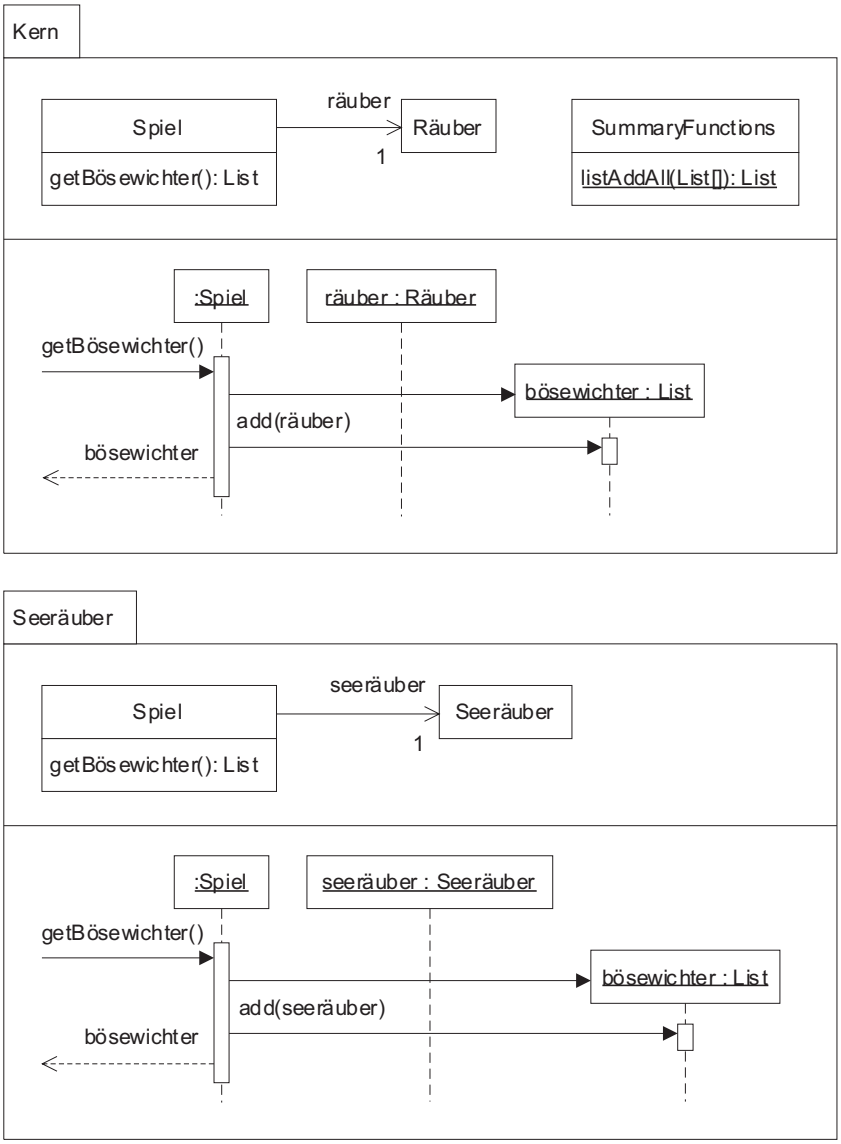

Abbildung 6.22: Summieren von Rückgabewerten von Operationen (i)

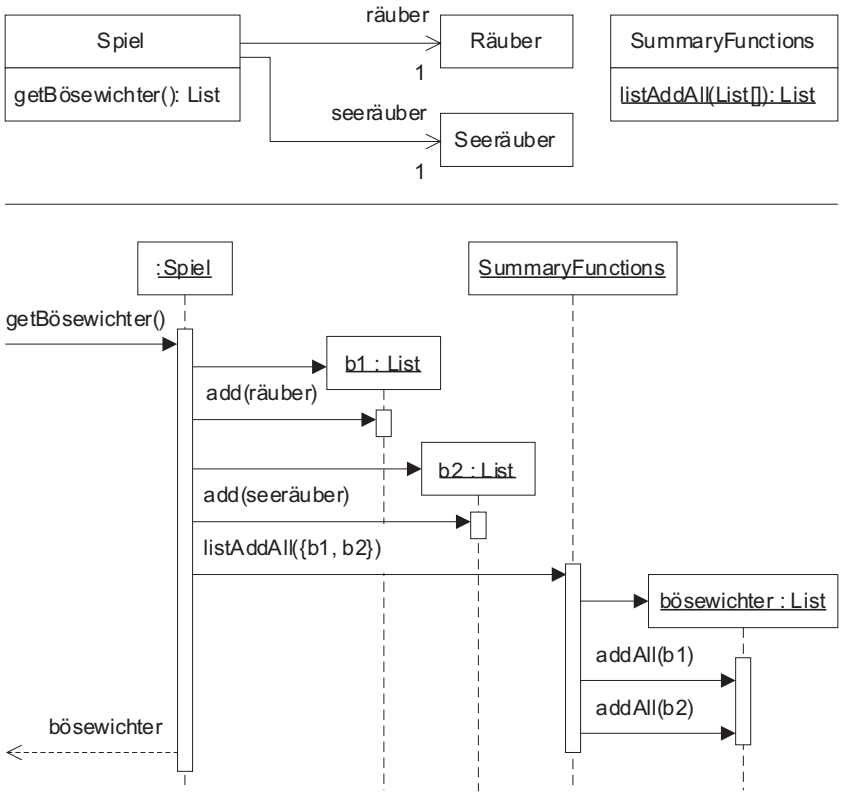

Abbildung 6.23: Summieren von Rückgabewerten von Operationen (ii)

6.9 Integrationsbeziehung Ersetzen

Die Integrationsbeziehung Ersetzen (engl.: *override relationship*) ersetzt ein Element vollständig durch ein anderes Element gleichen Typs.

6.9.1 Syntax

Die Umsetzung der Integrationsbeziehung Ersetzen in eine grafische Syntax zeigt Abbildung 6.24. Eine Integrationsbeziehung Ersetzen ist mit dem zu ersetzenden Element (overridenUnit) und dem Ersatz-Element (overridingUnit) assoziiert. Nach dem Ersetzen existiert das zu ersetzende Element (seine Kopie im Hypermodul) nicht mehr. Referenziert eine nachfolgende Integrationsbeziehung Verschmelzen oder Ersetzen das ersetzte Element, bricht die Integration mit einem Fehler ab.

Gibt man für Elemente eine Integrationsbeziehung Verschmelzen an, so verschmelzen auch die in den Elementen enthaltenen, übereinstimmenden Elemente miteinander (vgl. Unterkapitel 6.6). Dieser Automatismus kann mit Hilfe

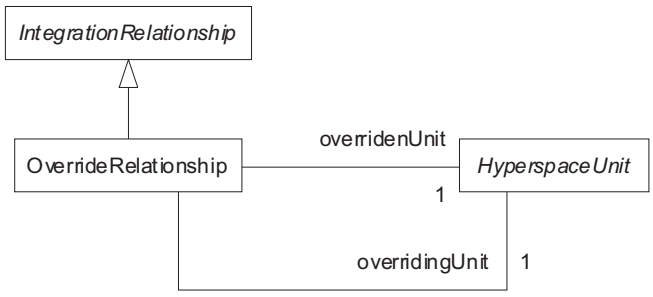

Abbildung 6.24: Profil Hyper/UML – Integrationsbeziehung Ersetzen

der Integrationsbeziehung Ersetzen abgeschaltet werden. Ein übereinstimmendes Element ersetzt dann ein anderes übereinstimmendes Element. Damit bleibt die Regel aus dem UML-Metamodell gewahrt, das ein Element kein übereinstimmendes Element enthalten darf.

6.9.2 Regeln

Die folgenden Regeln und Operationen vervollständigen die Syntax-Definition der Integrationsbeziehung Ersetzen. Weitere Regeln für die zu verschmelzenden Elemente legen die nachfolgenden Abschnitte zur Semantik in Form von Vorbedingungen fest.

OverrideRelationship

Regel 18: Das zu ersetzende Element und das Ersatz-Element sind nicht identisch, weil sonst das Ersetzen wirkungslos bliebe.

```
self.overridenUnit <> self.overridingUnit
```

Regel 19: Das zu ersetzende Element und das Ersatz-Element sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.units()->forAll( u |
   self.hypermodule.concern->exists( c |
      c.includesUnit(u)))
```

Regel 20: Das zu ersetzende Element und das Ersatz-Element sind vom aufgezählten Typ (vgl. Abschnitt 6.9.3).

```
self.units()->forAll( u |
   Set{
      Operation,
      ActionState, SubactivityState
   }->includes(u.oclType()))
```

Regel 21: Das zu ersetzende Element und das Ersatz-Element sind vom selben Typ. Elemente verschiedenen Typs, z.B. Operationen und Aktivitätszustände, können semantisch nicht ersetzt werden.

```
self.units()->forAll( u1, u2 |
   (u1.oclType() = u2.oclType()) and
   (u1.stereotype = u2.stereotype))
```

Operation 11: units() liefert das zu ersetzende Element und das Ersatz-Element zurück. Die obigen Regeln verwenden diese Operation.

```
units(): Set(HyperspaceUnit);
units =
   self.overridenUnit->union(self.overridingUnit)
```

6.9.3 Semantik

Jedes Element eines modellierten Systems besitzt eine Schnittstelle, über die Elemente miteinander interagieren. Die Schnittstelle einer Klasse sind ihre Attribute, Assoziationen und Operationen. Die Schnittstelle eines verfeinerten Zustands sind die enthaltenen Zustände, zu denen von außen Transitionen hinführen oder von denen Transitionen zu Zuständen außerhalb des verfeinerten Zustands erfolgen. Ersetzt ein Element ein anderes, muß es mindestens die gleiche Schnittstelle wie das ersetzte Element aufweisen, damit die von der Schnittstelle abhängigen Elemente semantisch gültig bleiben. Alternativ müssen die fehlenden Teile der Schnittstelle durch andere Integrationsbeziehungen wieder ergänzt werden. Oder abhängige, ungültige Elemente müssen ebenfalls ersetzt werden, wodurch sich die Schnittstellen-Problematik aber lediglich verlagert.

Im Vergleich zum Verschmelzen von Elementen ist das Ersetzen von Elementen also eine sehr viel invasivere Art der Integration. Sie erhöht nicht nur aus Sicht der Entwickler die Komplexität der Hypermodule, sondern führt auch zu weitreichenden Konsequenzen in Bezug auf die semantische Gültigkeit der Hypermodule. Diese Konsequenzen für alle Hyper/UML-Elemente zu untersuchen, hätte den Rahmen der Arbeit gesprengt. Deshalb ist der Einsatz der Integrationsbeziehung Ersetzen in Hyper/UML zunächst auf wenige Elementtypen beschränkt, die zusätzlich weiteren Bedingungen genügen müssen. Die Diskussion, ob und inwieweit die Einschränkungen aufzuheben sind und welcher praktische Nutzen verglichen mit der gestiegenen Komplexität daraus für die Modellierung erwächst, ist Thema zukünftiger Arbeiten.

6.9.4 Semantik für Operationen

Vorbedingungen

Die zu ersetzende Operation und die Ersatz-Operation müssen in einer identischen Klasse definiert werden. Diese Bedingung wird unter Einbeziehung aller

Integrationsbeziehungen des Hypermoduls geprüft. Sie ist erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt, z.B. indem sie Klassen oder Typen verschmilzt.

Die zu ersetzende Operation und die Ersatz-Operation tragen den gleichen Namen und haben die gleiche Parameterliste, d.h. identische Parametertypen und gleiche Datenflußrichtung (in, out, inout, result). Außerdem muß der Defaultwert pro Parameter gleich sein, sofern angegeben. Diese Bedingungen stellen sicher, daß Aufrufer der zu ersetzenden Operation nach dem Ersetzen die gleiche Schnittstelle wie vorher vorfinden.

Die zu ersetzende Operation und die Ersatz-Operation sind beide Klassenoperationen oder beide Instanzoperationen, beide konkret oder beide abstrakt, beide abgeleitet oder beide nicht abgeleitet. Eine weitere Eigenschaft, die gleich sein muß, ist concurrency.

Nachbedingungen

Eigenschaften Die Klasse der zu ersetzenden Operation definiert anstelle dieser die Ersatz-Operation mitsamt ihrer Implementierung, ihren Einschränkungen und Eigenschaftswerten.

Aktionen, Ereignisse Aktionen, die die zu ersetzende Operation aufrufen, rufen statt dessen die Ersatz-Operation auf. Ereignisse, die ein Aufruf der zu ersetzenden Operation auslöst, löst statt dessen ein Aufruf der Ersatz-Operation aus (vgl. Erläuterungen zu Aktionen und Ereignissen in Abschnitt 6.6.6 „Nachbedingungen“).

6.9.5 Semantik für Aktivitätszustände

Abbildung 6.25 zeigt im linken Teil die zu integrierenden Belange Kern und Barbaren mit Ausschnitten aus dem bereits in Abschnitt 6.6.10 vorgestellten Aktivitätsgraph. Der rechte Teil präsentiert das Ergebnis der Integration, nachdem die beiden Graphen verschmolzen und der Aktivitätszustand Kern::Schiff gewürfelt durch Barbaren::Barbaren vorrücken ersetzt wurde.

Vorbedingungen

Der zu ersetzende Aktivitätszustand und der Ersatz-Aktivitätszustand gehören zu einem identischen Aktivitätsgraphen. Beide sind keinem oder einem identischen Verantwortlichkeitsbereich zugeordnet. Diese Bedingungen werden unter Einbeziehung aller Integrationsbeziehungen des Hypermoduls geprüft. Sie sind erfüllt, wenn eine beliebige Integrationsbeziehung die Identität herstellt.

Nachbedingungen

Eigenschaften Der Aktivitätsgraph des zu ersetzenden Aktivitätszustands enthält anstelle diesem den Ersatz-Aktivitätszustand mit seiner *entry*-Aktion, seinen Einschränkungen und Eigenschaftswerten. Die Eigenschaften isDynamic, dynamicMultiplicity und dynamicArguments des Ersatz-Aktivitätszustands bleiben erhalten.

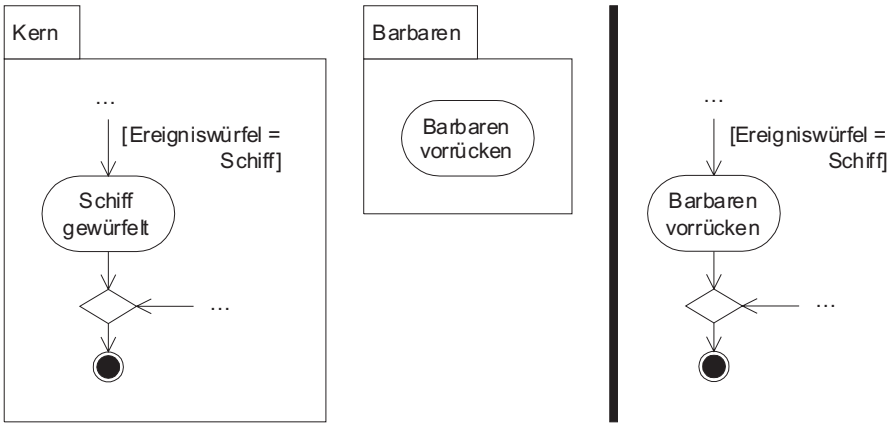

Abbildung 6.25: Ersetzen von Aktivitätszuständen

Transitionen An den Transitionen zum und vom Ersatz-Aktivitätszustand ändert sich nichts. Eingehende Transitionen des zu ersetzenden Aktivitätszustands führen statt dessen zum Ersatz-Aktivitätszustand hin. Ausgehende Transitionen des zu ersetzenden Aktivitätszustands gehen statt dessen vom Ersatz-Aktivitätszustand aus. Die gleiche Semantik gilt für ein- und ausgehende Objektflüsse.

Ausnahmen

Entstehen durch das Ersetzen von Aktivitätszuständen redundante Transitionen, werden sie entfernt. Eine Transition ist redundant, wenn eine zweite Transition dieselben (Aktivitäts-)Zustände verbindet und beim Auslösen die gleiche Aktion ausführt. Die Überwachungsbedingungen der entfernten Transitionen werden mit der Überwachungsbedingung der übrig gebliebenen Transition UND-verknüpft. Ist die verknüpfte Überwachungsbedingung widersprüchlich, dann ist das UML-Modell im Hypermodul ungültig, und die Integration bricht mit einem Fehler ab. Die Widerspruchsfreiheit von Überwachungsbedingungen kann nur geprüft werden, wenn die Bedingungen in einer formalen Sprache, z.B. OCL, formuliert sind. Eine zukünftige Erweiterung von Hyper/UML ist die wahlweise ODER-Verknüpfung von Überwachungsbedingungen.

6.10 Werkzeuge

Die Modellierung von Software-Systemen mit Hyper/UML ist mit vertretbarem Aufwand nur werkzeugunterstützt möglich. Ein passendes Werkzeug sollte die folgenden Funktionen anbieten:

- *Hyperraum verwalten.* Hierunter fällt die Definition des Hyperraums und die Zuordnung der Elemente zu den Punkten im Hyperraum. Zum Zeitpunkt dieser Arbeit existiert keine Notation für Hyperräume. Eine grafische Notation, wie sie Abbildung 5.3 auf Seite 77 nutzt, scheidet aber aus,

weil die Übersichtlichkeit dieser Darstellungsform bereits bei drei Dimensionen ihre Grenze erreicht. Etwas erfolgversprechender wäre eine textuelle Notation, wie sie Hyper/J nutzt.

- *Hyperebenen modellieren.* Diese Funktion entspricht der Modellierung von Systemen über UML-Modelle und -Diagramme und ist deshalb verhältnismäßig einfach zu realisieren.

- *Hypermodule verwalten.* Dazu gehört zum einen die Spezifikation von Hypermodulen mit Angabe der zu integrierenden Belange und der speziellen Integrationsbeziehungen. Die Spezifikation sollte textuell über Dialoge geschehen; eine grafische Notation ist nicht sinnvoll. Zum anderen muß das Werkzeug die durch ein Hypermodul beschriebene Integration durchführen und das Ergebnis in Form von UML-Diagrammen anzeigen können. Die Durchführung der Integration ist unproblematisch, sobald die in den vorangegangenen Unterkapiteln definierten Vor- und Nachbedingungen implementiert sind. Schwieriger ist es, das Ergebnis einer Integration übersichtlich darzustellen, weil das Werkzeug die Elemente in den Diagrammen automatisch anordnen muß.

Der Aufwand, ein Werkzeug für Hyper/UML von Grund auf neu zu entwickeln, ist sehr hoch. Da die Funktionalität des Werkzeugs zu einem wesentlichen Teil die Modellierung von Hyperebenen ausmacht und dies, wie erwähnt, weitgehend der normalen Modellierung mit der UML entspricht, bietet es sich an, anstelle einer Neuentwicklung ein bestehendes CASE-Werkzeug um Hyper/UML zu erweitern. Die Erweiterung bestünde vor allem im Hinzufügen von Dialogen zur Verwaltung von Hyperraum und Hypermodulen. Hyperebenen könnten als UML-Pakete mit dem Stereotyp *«hyperslice»* modelliert werden. Darüber hinaus müßte die Erweiterung prüfen, ob die Hyperebenen deklarativ vollständig sind, und sie müßte die Integration durchführen.

Eine derartige Erweiterung stellt bestimmte Voraussetzungen an das zu erweiternde CASE-Werkzeug. So muß es die Ergänzung seines Metamodells zulassen, damit die Erweiterung den Hyperraum und die Hypermodule eines Systems in der Syntax von Unterkapitel 6.3 und 6.5 im Repository abspeichern kann. Außerdem muß das CASE-Werkzeug eine Schnittstelle bereitstellen, über die die Erweiterung auf das Modell eines Systems, sprich auf die Elemente in den Hyperebenen zugreifen kann. Der Zugriff sollte auch schreibend möglich sein, damit die Erweiterung die Ergebnisse von Integrationen in Hypermodul-Paketen ablegen und anzeigen kann. Insbesondere die erste Voraussetzung erfüllen nur wenige Werkzeuge, z.B. MetaEdit+ [ME] oder ArgoUML [AU].

Ziel dieser Arbeit war es, Hyper/UML zunächst vollständig zu definieren und im Rahmen einer Fallstudie zu erproben. Auf Basis der Spezifikation können jetzt zukünftige Arbeiten entsprechende Werkzeuge implementieren. Dabei sollten sie auch geeignete Mittel zur Verwaltung von Hyperräumen sowie zur übersichtlichen Anzeige integrierter Hypermodule ausloten und auf Praxistauglichkeit testen.

6.11 Zusammenfassung

Die Abbildung des Hyperspace-Ansatzes auf die UML liefert einen wichtigen Beitrag für das Erreichen der Zielsetzung dieser Arbeit. Hyper/UML erlaubt die Modellierung feingranularer Produktlinien-Komponenten. Da Hyper/UML in Abstimmung auf Hyper/J entwickelt worden ist, können die Komponenten nach ihrer Modellierung ohne weitere Umsetzung in Java implementiert werden. Nimmt man beide Abbildungen zusammen, so ist es jetzt möglich, den Hyperspace-Ansatz durchgängig in allen Phasen der Produktlinien-Entwicklung zu nutzen. Die dabei entstehenden Produktlinien weisen eine handhabbare Komplexität auf und sind wartbar. Zudem können aus ihnen ausführbare Systeme beziehungsweise Hypermodule durch Integration der feingranularen Komponenten automatisiert gefertigt werden.

Das nächste Kapitel integriert den Hyperspace-Ansatz mit Hyper/UML und Hyper/J in die FeatuRSEB-Methode. In Kapitel 8 folgt die Bewertung der integrierten Methode bezüglich der Zielsetzung dieser Arbeit.

Kapitel 7

Die HyperFeatuRSEB-Methode

Das Kapitel beschreibt die im Rahmen dieser Arbeit aufgestellte Produktlinien-Entwicklungsmethode HyperFeatuRSEB (sprich: *hyper-featured RSEB*). Die Methode ist aus den in [SBR00] veröffentlichten Gedanken hervorgegangen. Sie basiert auf der FeatuRSEB-Methode (vgl. Kapitel 4) und integriert in diese den Hyperspace-Ansatz (vgl. Kapitel 5) mit seinen zwei Abbildungen Hyper/UML (vgl. Kapitel 6) und Hyper/J (vgl. Unterkapitel 5.3). Damit unterstützt die Methode besonders die Entwicklung solcher Produktlinien, aus denen schlanke Systeme in Serie gefertigt werden sollen. Illustriert wird die Methodenbeschreibung durch Ausschnitte aus der Fallstudie „Die Siedler von Catan“ (vgl. Kapitel 3). Die Studie wurde zur Erprobung der HyperFeatuRSEB-Methode durchgeführt. Eine Bewertung der Methode folgt in Kapitel 8.

7.1 Überblick

Die HyperFeatuRSEB-Methode übernimmt die charakteristischen Eigenschaften von FeatuRSEB, wandelt sie aber teilweise ab, um den Hyperspace-Ansatz zu integrieren. Dadurch wird die Komplexität entwickelter Produktlinien handhabbar gehalten, ihr Ausbau vereinfacht und die automatisierte Fertigung von Systemen anstelle der manuellen Entwicklung möglich. Die folgenden Punkte listen die Eigenschaften von HyperFeatuRSEB auf (Änderungen gegenüber FeatuRSEB sind kursiv gesetzt):

- Strukturierung von Anforderungen an eine Produktlinie in Gemeinsamkeit und Variabilität durch ein Merkmalmodell
- Beschreibung der Produktlinien-Anforderungen durch ein Use-Case-Modell, *getrennt nach den Merkmalen der Produktlinie mittels Hyper/UML*
- Objektorientierte Analyse und Entwurf von Architektur und Komponenten der Produktlinie; Dokumentation der Ergebnisse durch ein Objektmodell, *getrennt nach Produktlinien-Merkmalen mittels Hyper/UML*

- Objektorientierte Implementierung der Produktlinie, *getrennt nach Produktlinien-Merkmalen mittels Abbildungen des Hyperspace-Ansatzes auf objektorientierte Programmiersprachen wie z.B. Hyper/J*
- *Serienfertigung von Produkten aus der Produktlinie, d.h. Generierung von Use-Case-Modell, Objektmodell und Quellcode der Produkte*

HyperFeatuRSEB ist genau wie FeatuRSEB keine domänenspezifische, sondern eine allgemein anwendbare Methode. Sie eignet sich für Produktlinien und Produkte aus allen Domänen, deren Wissen mit Use-Cases beschrieben und in Objekte zerlegt werden kann. Weiterhin ist HyperFeatuRSEB unabhängig von der zur Implementierung verwendeten objektorientierten Programmiersprache. Derzeit steht Java beziehungsweise Hyper/J zur Verfügung.

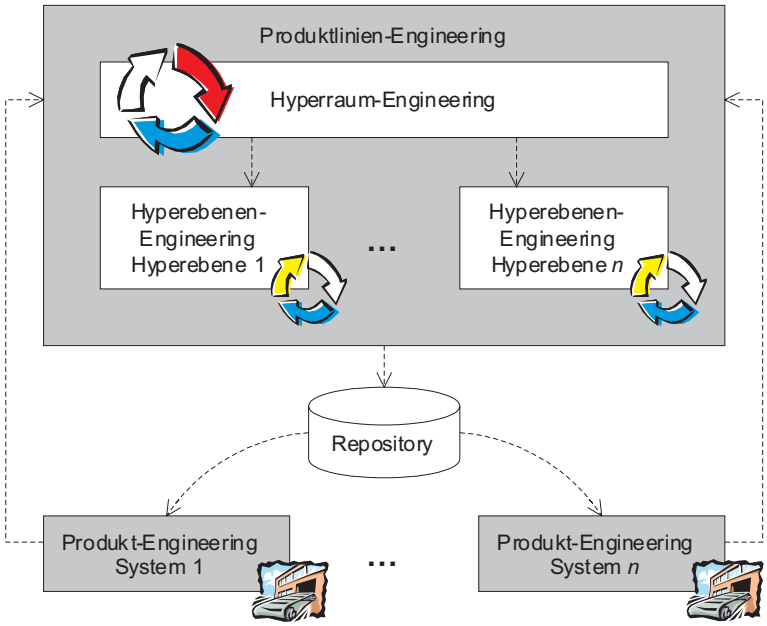

Abbildung 7.1: Vorgehensmodell in HyperFeatuRSEB

Das Vorgehensmodell in HyperFeatuRSEB zur Entwicklung von Produktlinien und zur Serienfertigung von Produkten zeigt Abbildung 7.1, in Anlehnung an das in Abbildung 2.3 auf Seite 20 vorgestellte allgemeine Vorgehensmodell. HyperFeatuRSEB unterscheidet drei Typen von Entwicklungsprozessen:

- Das Produktlinien-Engineering zerfällt in *Hyperraum-Engineering* und *Hyperebenen-Engineering*. Ersteres entwickelt und wartet das Merkmalmodell und Use-Case-Modell der Produktlinie sowie ihre Architektur. Letzteres entwickelt und wartet das nach den Merkmalen der Produktlinie in einzelne Hyperebenen aufgetrennte Objektmodell der Produktlinie.
- Das *Produkt-Engineering* generiert ausgehend von Kunden-Anforderungen automatisiert konkrete Systeme auf Basis der Ergebnisse der Prozesse

im Produktlinien-Engineering. Den dabei zugrundeliegenden Serienfertigungsgedanken symbolisiert in Abbildung 7.1 das Fließband. Fordert ein Kunde noch individuelle Anpassungen seines Systems, entwickelt sie das Produkt-Engineering in Abstimmung mit dem Produktlinien-Engineering.

Die Prozeßaufteilung ist ein Mittel, um dem Management eine bessere Steuerung der Entwicklung zu erlauben und den Einsatz der Mitarbeiter zu planen. So kommen im Hyperraum-Engineering vorzugsweise erfahrene Architekten zum Einsatz, während das Produkt-Engineering so einfach gehalten ist, daß speziell geschultes Verkaufspersonal diese Aufgabe übernehmen kann. Nur wenn ein Kunde noch individuelle Anforderungen wünscht, die von der Produktlinie nicht abgedeckt werden, arbeiten im Produkt-Engineering Entwickler mit. Um die Abstimmung zwischen den Prozessen zu vereinfachen, können einzelne Prozesse beziehungsweise deren Teams zusammengelegt werden. Besonders zu empfehlen ist eine Zusammenlegung des Hyperraum-Engineering-Prozesses mit denjenigen Prozessen im Hyperebenen-Engineering, die die Gemeinsamkeit der Produktlinie entwickeln. Mehr darüber in Abschnitt 7.2.4.

Die folgenden Unterkapitel 7.2 und 7.3 beschreiben die drei Prozeßtypen, ihre Phasen und deren Ergebnisse näher. Dabei werden Hyperraum-Engineering und Hyperebenen-Engineering in einem Unterkapitel „Produktlinien-Engineering“ zusammengefaßt. Anschließend diskutiert Unterkapitel 7.4 das Thema Werkzeugunterstützung für HyperFeatuRSEB.

7.2 Produktlinien-Engineering

Ziel des Produktlinien-Engineerings (beziehungsweise Hyperraum- und Hyperebenen-Engineerings) in HyperFeatuRSEB ist die Festlegung der gemeinsamen Architektur für alle Systeme, die aus einer Produktlinie gefertigt werden sollen, und die Bereitstellung wiederverwendbarer Komponenten, die die Architektur ausfüllen. Dabei steht vor allem die spätere Wartbarkeit der Produktlinie im Vordergrund. Ist sie nicht gewährleistet, verkürzt sich die Lebensdauer der Produktlinie und die hohen Investitionen amortisieren sich nicht (vgl. Abschnitt 2.2.1).

7.2.1 Ablauf

Das Produktlinien-Engineering gliedert sich in drei Phasen, die zu entsprechenden Ergebnissen führen (vgl. Abbildung 7.2).

Phase 1: Beschreibung von Anforderungen. Das Hyperraum-Engineering nimmt in dieser Phase gemeinsame und variable Anforderungen an die Produktlinie auf. Dazu identifiziert es aus Sicht potentieller Kunden die Merkmale der Produktlinie und hinterlegt sie mit ihren Abhängigkeiten im Merkmalmodell. Anschließend zerlegt das Hyperraum-Engineering die Anforderungen der Produktlinie simultan auf zwei verschiedene Arten: zum einen wie in FeatuRSEB in die Elemente des Use-Case-Modells, also in Akteure, Use-Cases und

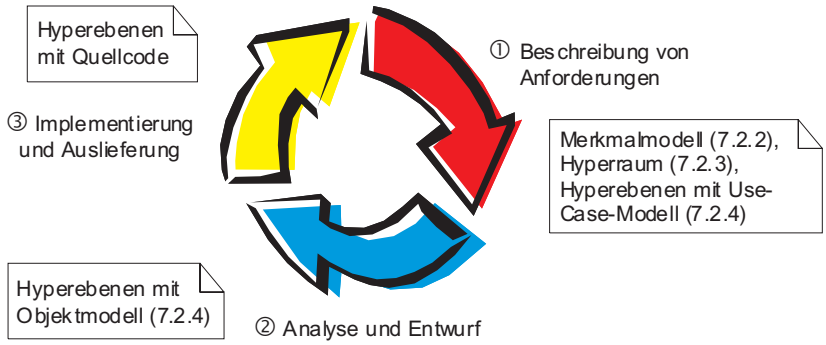

Abbildung 7.2: Ablauf Produktlinien-Engineering

Aktivitätsgraphen. Zum anderen zerlegt es das entstehende Use-Case-Modell zusätzlich noch nach den Merkmalen der Produktlinie. Es ergibt sich ein zweidimensionaler Hyperraum. Die Belange der ersten Dimension sind die Elemente des Use-Case-Modells, die Belange der zweiten Dimension die Merkmale. Den Punkten im Hyperraum sind die entsprechenden Elemente des Use-Case-Modells zugeordnet. Elemente, die zu demselben Merkmal gehören, kapselt das Hyperraum-Engineering jeweils in einer Hyperebene. Demzufolge definiert jede Hyperebene den Teil des Use-Case-Modells der Produktlinie, der für genau ein Merkmal relevant ist.

Phase 2: Analyse und Entwurf. Hier leitet das Hyperraum-Engineering zunächst die Architektur der Produktlinie aus dem Use-Case-Modell ab. Die Architektur besteht aus Paketen (in FeatuRSEB aus Komponentensystemen), die das Rückgrat des Objektmodells der Produktlinie bilden. Den Entwurf des Objektmodells übernimmt das Hyperebenen-Engineering. Pro Hyperebene gibt es einen Hyperebenen-Engineering-Prozeß. Er zerlegt das in einer Hyperebene definierte Use-Case-Modell objektorientiert in Klassen und kapselt dann die Klassen in derselben Hyperebene wie das Use-Case-Modell. Fortan enthält jede Hyperebene nur genau den Teil des Use-Case- und Objektmodells der Produktlinie, der für ein bestimmtes Merkmal relevant ist. Die Hyperebenen-Engineering-Prozesse melden ihre identifizierten Klassen an den Hyperraum-Engineering-Prozeß, damit letzterer die Zerlegungsvorgänge im Hyperebenen-Engineering koordinieren kann.

Phase 3: Implementierung und Auslieferung. Zum Schluß implementieren die Hyperebenen-Engineering-Prozesse jeweils ihr Objektmodell in der für die Produktlinie verwendeten objektorientierten Programmiersprache. Nach erfolgreichem Test liefern die Prozesse ihre Hyperebene mit Use-Case-Modell, Objektmodell und Quellcode in das Repository aus.

7.2.2 Merkmalmodell

HyperFeatuRSEB übernimmt das Merkmalmodell von FeatuRSEB in der Version aus [CE00, Kapitel 4], ohne den Prozeß seiner Erstellung oder die Merkmaldiagramm-Notation zu verändern. Damit ist das Merkmalmodell auch in HyperFeatuRSEB zentrales Modell jeder entwickelten Produktlinie. Es ermöglicht nicht nur einen schnellen Überblick zur Gemeinsamkeit und Variabilität einer Produktlinie, sondern bestimmt gleichzeitig – durch die an Merkmalen ausgerichtete Trennung von Use-Case- und Objektmodell der Produktlinie – die weiteren Aktivitäten im Produktlinien-Engineering wesentlich stärker, als das bei FeatuRSEB der Fall ist. Das nachfolgende Produkt-Engineering basiert auf der Auswahl von Merkmalen durch Kunden (vgl. Unterkapitel 7.3), so daß der Erfolg des Produkt-Engineerings ebenfalls entscheidend vom Merkmalmodell abhängt. Aus diesen Gründen ist die Identifizierung der „richtigen“ Merkmale von Produktlinien eine der wichtigsten Aufgaben in HyperFeatuRSEB. Die Modellierung von Merkmalen erläutert bereits [CE00, Abschnitt 4.9.1] in ausführlicher Form. Sie ist somit nicht Gegenstand der vorliegenden Arbeit. Abbildung 7.3 wiederholt noch einmal den Auszug aus dem Merkmalmodell der Produktlinie „Die Siedler von Catan“ (vgl. Abschnitt 4.2.2).

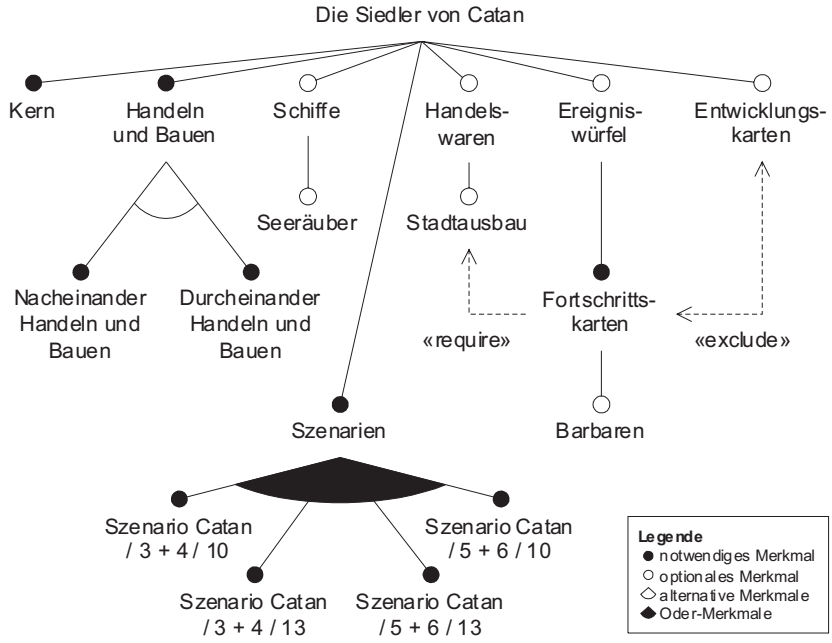

Abbildung 7.3: Beispiel Merkmalmodell

7.2.3 Hyperraum

Neu in HyperFeatuRSEB ist der Hyperraum einer Produktlinie. Der Hyperraum beschreibt das Ergebnis der simultanen objekt- und merkmalorientierten

Zerlegung der Produktlinie. Er besteht folglich aus zwei Dimensionen. Die erste Dimension repräsentiert die schon aus FeatuRSEB bekannte Zerlegung in Use-Cases, Klassen usw. Die zweite Dimension stellt die merkmalorientierte Zerlegung der Produktlinie dar. Ihre Belange sind die Merkmale aus dem Merkmalmodell. Eine Ausnahme bilden Merkmale, deren Zweck allein die Gruppierung anderer Merkmale ist. Im Beispiel trifft das auf Handeln und Bauen sowie Szenarien zu. Alle anderen Merkmale sind mit ihrem Merkmal-Belang im Hyperraum über eine *trace*-Abhängigkeitsbeziehung verknüpft.

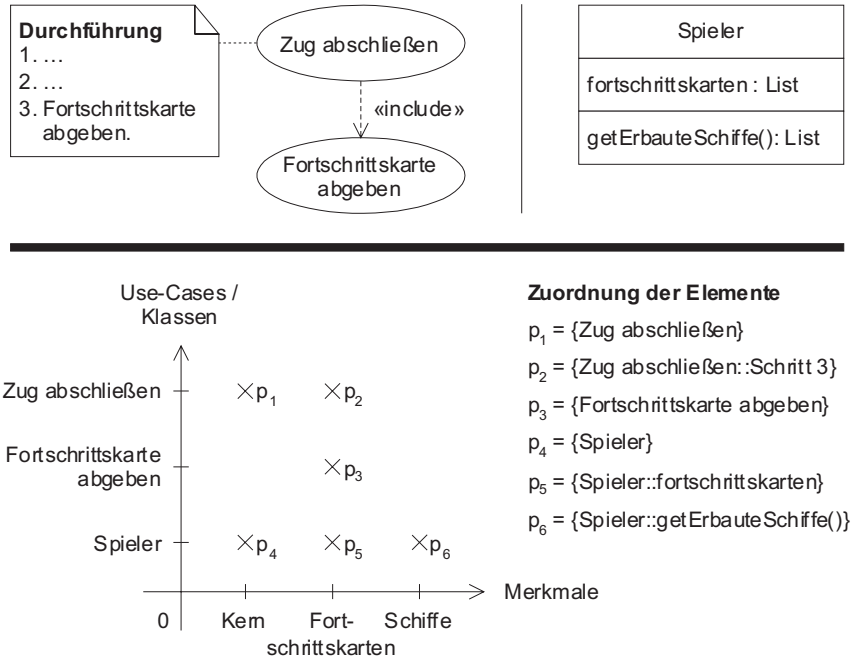

Abbildung 7.4: Beispiel merkmalorientierte Zerlegung im Hyperraum

Was bewirkt die merkmalorientierte Zerlegung der Produktlinie? Sie führt dazu, daß alle Elemente, die bei der funktionalen und objektorientierten Zerlegung in FeatuRSEB entstehen, noch zusätzlich nach Merkmal-Belangen zerlegt werden. Abbildung 7.4 gibt hierfür ein Beispiel. Im oberen Teil der Abbildung ist ein Ausschnitt der mit FeatuRSEB zerlegten Produktlinie „Die Siedler von Catan“ zu sehen: links das Use-Case-Modell, rechts das Objektmodell. Die dort vorgenommene Zerlegung in Use-Cases und Klassen spiegelt sich im unteren Teil der Abbildung in der (ersten) Dimension Use-Cases/Klassen des Hyperraums wider. Dementsprechend sind Zug abschließen, Fortschrittskarte abgeben und Spieler die Belange der ersten Dimension. In der (zweiten) Dimension Merkmale sind als Belange drei Merkmale der Produktlinie aufgetragen: Kern, Fortschrittskarten und Schiffe. HyperFeatuRSEB zerlegt jetzt das Use-Case-Modell und Objektmodell noch nach den Merkmal-Belangen, so daß sich die in der Abbildung aufgeführte Zuordnung von Elementen der Produktlinie zu Punkten im Hyperraum ergibt. Danach betrifft in der Durchführung des Use-Case Zug abschließen der dritte Schritt nur das Merkmal Fortschrittskarten, ebenso wie der Use-Case Fort-

schrittskarte abgeben. Die Klasse Spieler ist dreigeteilt. Sie gehört generell mit ihren Attributen und Operationen zum Kern, mit Ausnahme von fortschrittskarten und getErbauteSchiffe(). Diese Elemente sind dem Merkmal Fortschrittskarten beziehungsweise Schiffe zugeordnet.

Die während der merkmalorientierten Zerlegung gewonnenen, im Hyperraum abgelegten Informationen werden im nächsten Schritt dazu verwendet, das Use-Case-Modell und Objektmodell mittels Hyperebenen nach Merkmalen aufzutrennen. Das versetzt Entwickler in die Lage, die Modelle und die dahinterliegende Implementierung getrennt nach Merkmalen betrachten und bearbeiten zu können. Im Produkt-Engineering werden diese Hyperebenen über Hypermodule wieder zu Systemen zusammengeführt.

7.2.4 Hyperebenen mit Use-Case- und Objektmodell

Eine Hyperebene kapselt in HyperFeatuRSEB immer genau einen Belang der Merkmal-Dimension des Hyperraums der entwickelten Produktlinie. Damit enthält eine Hyperebene nur diejenigen Elemente des Use-Case- und Objektmodells der Produktlinie, die für ein Merkmal relevant sind. Benötigte Elemente aus anderen Hyperebenen werden nicht referenziert, sondern deklarativ vollständig definiert: Die Hyperebene definiert die Elemente nur bis zu dem Umfang, in dem sie sie nutzt beziehungsweise es die Regeln der UML und der verwendeten Programmiersprache erforderlich machen.

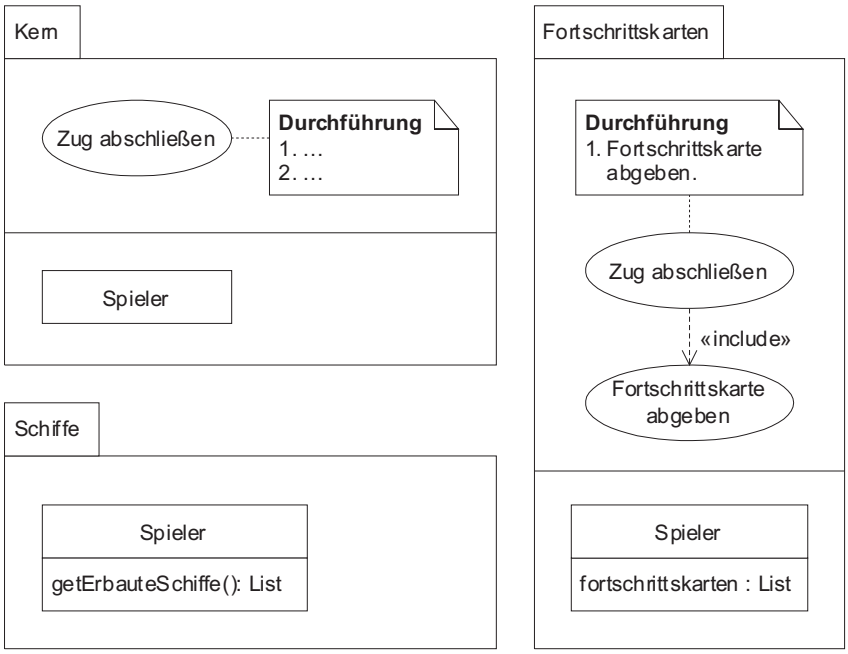

Abbildung 7.5: Beispiel Hyperebenen mit Use-Case- und Objektmodell

Abbildung 7.5 setzt das im vorangegangenen Abschnitt begonnene Beispiel fort. Entsprechend den dort identifizierten Merkmal-Belangen Kern, Fortschrittskarten und Schiffe sind hier drei gleichnamige Hyperebenen entstanden, die die den Merkmalen zugeordneten Elemente kapseln. So definiert die Hyperebene Kern die Klasse Spieler sowie den Use-Case Zug abschließen mit Schritt 1 und 2 seiner Durchführung. Den dritten Schritt, und damit die *include*-Beziehung zum Use-Case Fortschrittskarte abgeben, definiert hingegen die Hyperebene Fortschrittskarten. Analog sind die Bestandteile der Klasse Spieler auf die Hyperebenen verteilt.

Strategie zur Integration von Merkmal-Belangen

Eine Hyperebene stellt für sich genommen kein ausführbares System einer Produktlinie dar. Statt dessen muß das Produkt-Engineering die im Produktlinien-Engineering erstellten Hyperebenen gemäß den Wünschen der Kunden zu Systemen beziehungsweise Hypermodulen zusammenfügen. Diese Integration soll einem Generator übertragen werden, da HyperFeatuRSEB nur so die in der Zielsetzung der Arbeit geforderte automatisierte Serienfertigung realisieren kann. Mit anderen Worten: Der Generator muß ausgehend von den Merkmalen, die ein Kunde wählt, eine Hypermodul-Spezifikation erzeugen. Die Spezifikation beschreibt, welche Merkmale beziehungsweise Hyperebenen wie zu integrieren sind. Anschließend übergibt der Generator die Spezifikation an das Hyper/UML- und Hyper/J-Werkzeug, die daraus das Use-Case-Modell, Objektmodell und den ausführbaren Quellcode des Produkts erstellen.

Die Elemente der zu integrierenden Hyperebenen stehen jedoch meist nicht orthogonal zueinander. Manche Elemente stimmen semantisch überein und müssen durch spezielle Integrationsbeziehungen wie Verschmelzen oder Ersetzen zusammengeführt werden. Wählt ein Kunde beispielsweise die Merkmale Kern und Fortschrittskarten aus, müssen die Durchführungen des Use-Case Zug abschliessen und die Klassen Spieler verschmelzen. Woher bekommt der Generator die speziellen Integrationsbeziehungen? HyperFeatuRSEB hinterlegt sie textuell im Merkmalmodell. Jedem Merkmal, das einen Belang repräsentiert, ordnet das Produktlinien-Engineering die notwendigen Integrationsbeziehungen zu. Wenn ein Kunde ein Merkmal auswählt, übernimmt der Generator die dem Merkmal zugeordneten Integrationsbeziehungen in die Hypermodul-Spezifikation.

Das Verfahren der manuellen Aufstellung von Integrationsbeziehungen stößt in der Praxis rasch an seine Grenzen, weil die Anzahl der Beziehungen mit jedem Merkmal der Produktlinie stark ansteigt – was dem Ziel der Arbeit, die Komplexität von Produktlinien handhabbar zu halten, zuwiderläuft. Deshalb legt HyperFeatuRSEB für alle zu integrierenden Elemente die Integrationsbeziehung Verschmelzen verbindlich fest. Diese sogenannte *Integrationsstrategie* bewirkt, daß der Generator übereinstimmende, zu integrierende Elemente grundsätzlich miteinander verschmilzt, ohne daß dies im Merkmalmodell gesondert spezifiziert werden muß. Reicht die Integrationsstrategie einmal nicht aus oder würde sie semantisch zu falschen Ergebnissen führen, können Abweichungen von der Strategie weiterhin im Merkmalmodell hinterlegt werden. Beispiele hierfür folgen. Wieviele solcher Abweichungen in der Fallstudie auftraten, stellt Tabelle 7.1 zusammen.

Integrationsbeziehung	Use-Case-Modell	Objektmodell
Verschmelzen	3	–
Ordnen	3	5
Summieren	–	30
Ersetzen	8	7
Summe	14	42

Tabelle 7.1: Integrationsbeziehungen trotz Integrationsstrategie

Auffällig ist im Objektmodell die hohe Zahl an Integrationsbeziehungen Summieren. Sie ist eine Konsequenz der Integrationsstrategie. Wann immer Operationen mit Rückgabewerten verschmelzen, ist für die verschmolzene Operation ein gemeinsamer Rückgabewert zu berechnen. Da dies von der Semantik der Operationen abhängt, greift hier kein Automatismus. Statt dessen müssen die notwendigen Summieren-Beziehungen manuell aufgestellt werden. In einigen Fällen besteht jedoch die Möglichkeit, das Summieren durch Refaktorierung des Objektmodells zu vermeiden. Abbildung 7.6 zeigt im oberen Teil (a) die Hyperebene Kern mit der Klasse RohstoffeAufHandelsplätzenHandeln, deren Operation getHandelsplätze() von der Hyperebene Stadtausbau erweitert wird. Verschmelzen beide Ebenen, führt die Summenfunktion listAddAll() die zurückgegebenen Handelsplätze in einer Liste zusammen.

Das Summieren läßt sich vermeiden, wenn man die Refaktorierung „Methode extrahieren“ [Fow99, Seite 110ff] anwendet, d.h. das Hinzufügen der Listeneinträge von getHandelsplätze() in eine neue Operation addHandelsplätzeTo() auslagert. Das Ergebnis zeigt Teil (b) der Abbildung. Dort verschmelzen nur noch die Operationen addHandelsplätzeTo(), was aufgrund des nicht vorhandenen Rückgabewerts keine Integrationsbeziehung Summieren mehr erfordert. Möglich ist eine solche Refaktorierung aber nur in Fällen, bei denen das zurückzuliefernde Objekt als veränderbares Objekt an die neue Operation übergeben werden kann. In Java trifft das z.B. auf eine List oder Map zu. Die Refaktorierung ist nicht anwendbar bei Rückgaben in Form von primitiven Datentypen (int, boolean usw.) oder unveränderbaren Wert-Objekten wie z.B. String.

Die Alternative zur Entscheidung „Integrationsstrategie ist Verschmelzen“ wäre die Wahl von Ersetzen gewesen. Allerdings ist sowohl in Hyper/UML als auch in Hyper/J die Semantik der Integrationsbeziehung Ersetzen wegen der Schnittstellen-Problematik (vgl. Abschnitt 6.9.3) nicht für alle Typen von Elementen definiert, die bei der Modellierung und Implementierung von Systemen genutzt werden können. Demgegenüber ist das Verschmelzen für alle Elementtypen möglich. Mithin taugt die Integrationsbeziehung Ersetzen nicht als allgemeingültige Integrationsstrategie.

Richtlinien für den Entwurf von Hyperebenen

Die Festlegung einer Integrationsstrategie reduziert die Anzahl der Integrationsbeziehungen, die das Produktlinien-Engineering von Hand definieren und bei Änderungen warten muß. Dieser Vorteil wird um so größer, je besser Modell

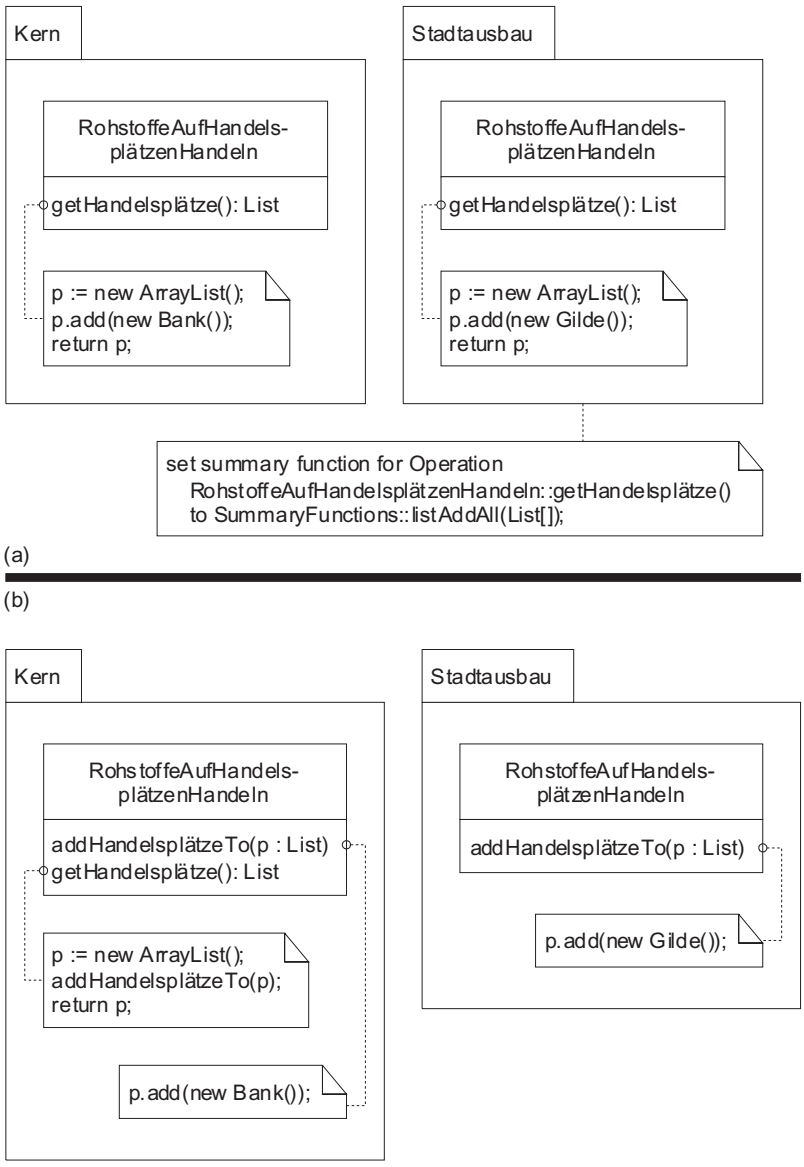

Abbildung 7.6: Vermeidung von Integrationsbeziehungen durch Refaktorierung

und Quellcode einer Produktlinie von ihrer Struktur her auf die Integrationsstrategie, also auf das Verschmelzen, ausgerichtet sind. Nach den Erfahrungen aus der Fallstudie erhält derartige Strukturen, wer bei der Modellierung und Implementierung Regeln beachtet, mit denen zusätzliche, unnötige Integrationsbeziehungen vermieden werden[1]:

- *Architektur.* Stimmen Elemente semantisch überein, müssen sie sich in allen Hyperebenen in der gleichen Hierarchie von Elementen befinden (vgl. Abbildung 6.4 auf Seite 96). Die Hyperebene als oberste Hierarchiestufe zählt hierbei nicht mit. Im Endeffekt müssen alle Hyperebenen der Produktlinien-Architektur folgen.

- *Namensvergabe.* Stimmen Elemente semantisch überein, müssen sie den gleichen, nicht leeren Namen tragen. Bei Operationen muß zusätzlich die Signatur gleich sein (vgl. die Implementierungen der Operation correspondsTo() in Abschnitt 6.6.2).

Eine Mißachtung der Regeln hat zur Folge, daß Hyper/UML und Hyper/J eigentlich übereinstimmende Elemente nicht als solche erkennen und folglich nicht verschmelzen – jedenfalls nicht ohne zusätzliche Integrationsbeziehungen. Bezogen auf das Beispiel stellt sich die Situation wie folgt dar:

- *Architektur.* Würde die Klasse Fortschrittskarten::Spieler in einem ihrer Hyperebene untergeordneten Paket namens Spiel liegen, stimmt sie nicht mehr mit Spieler aus Kern und Schiffe überein. Ein Hypermodul, das alle drei Hyperebenen integriert, enthält Spieler mit getErbauteSchiffe() und Spiel::Spieler mit fortschrittskarten.

- *Namensvergabe.* Würde die Klasse Fortschrittskarten::Spieler in Spieler2 umbenannt, stimmt sie nicht mehr mit Spieler aus Kern und Schiffe überein. Ein Hypermodul, das alle drei Hyperebenen integriert, enthält Spieler mit getErbauteSchiffe() und Spieler2 mit fortschrittskarten.

Die Einhaltung der Regeln sicherzustellen wäre vergleichsweise einfach, wenn das Produktlinien-Engineering nicht in mehrere Prozesse und Teams zerfallen würde. Eine Kontrollinstanz ist erforderlich, die die Einhaltung der Regeln für das Use-Case- und Objektmodell einer Produktlinie überwacht: das Hyperraum-Engineering. Für diese Wahl gibt es drei Gründe. Erstens hat das Hyperraum-Engineering die Produktlinien-Architektur definiert und verfügt demnach über das notwendige Wissen, um die Architektur der Modelle in den Hyperebenen zu überprüfen. Zweitens hat das Hyperraum-Engineering die Use-Case-Modelle in den Hyperebenen selbst aufgestellt und währenddessen bereits überprüft. Drittens melden die Hyperebenen-Engineering-Prozesse, wie oben erwähnt, die im Objektmodell identifizierten Klassen an das Hyperraum-Engineering. Somit ist dort eine entsprechende Abstimmung hinsichtlich Einordnung der Klassen in die Architektur und Namensvergabe möglich.

[1] Weitere Implementierungsrichtlinien für Hyper/J finden sich in [Hal01, Unterkapitel 5.6].

Darüber hinaus liefert das Merkmalmodell Hinweise, welchen Prozessen im Hyperebenen-Engineering das Hyperraum-Engineering seine verstärkte Aufmerksamkeit widmen sollte. Gemeint sind die Prozesse, die zueinander in Beziehung stehende Merkmale bearbeiten. Dort verhindert die deklarative Vollständigkeit zwar das Auftreten von direkten Referenzen zwischen den Objektmodellen der Merkmal-Belange, indirekt beziehen sich die Modelle aber häufig auf semantisch übereinstimmende Klassen. Besonders kritisch sind in diesem Zusammenhang die Abhängigkeiten der variablen Merkmale von den gemeinsamen Merkmalen. Die Hyperebenen der gemeinsamen Merkmalen definieren das grundlegende Objektmodell einer Produktlinie, auf das alle variablen Merkmale aufbauen. Somit tragen Hyperebenen-Engineering-Prozesse, die das gemeinsame Objektmodell entwerfen, maßgeblich zum Erfolg des Produktlinien-Engineerings bei. Deshalb ist es empfehlenswert, die eingangs angedachte Zusammenlegung dieser speziellen Prozesse mit dem Hyperraum-Engineering-Prozeß vorzunehmen. Das verringert nicht nur den Koordinationsaufwand, sondern senkt zugleich das Risiko, eine unverständliche, unwartbare Produktlinie zu entwickeln.

Modellierung von Variabilität in Hyperebenen

Neben dem Zwang zur Koordination eröffnet die Integrationsstrategie zusammen mit den anderen Integrationsbeziehungen eine neue Art der Modellierung und Implementierung von Variabilität in Produktlinien, die ohne Variationspunkte, *extend*-Beziehungen, Vererbung und Entwurfsmuster auskommt. Sie vermeidet die aus FeatuRSEB bekannten künstlichen Abstraktionen und reduziert dadurch die Komplexität entwickelter Produktlinien. Um einen Vergleich zwischen FeatuRSEB und HyperFeatuRSEB zu ermöglichen, wird das bisherige Beispiel in diesem Unterkapitel so erweitert, daß es den gleichen Umfang hat wie die Beispiele in Unterkapitel 4.2.

Abbildung 7.7 zeigt, wie das Use-Case-Modell von FeatuRSEB aus Abbildung 4.4 auf Seite 49 in HyperFeatuRSEB modelliert wird. Den wichtigsten Unterschied demonstriert der Use-Case Zug abschließen. Er benötigt in HyperFeatuRSEB weder Variationspunkt noch Erweiterungsstelle. Wählt ein Kunde das Merkmal Fortschrittskarten aus, verschmelzen die Durchführungen von Zug abschließen aus Kern und Fortschrittskarten, und zwar wegen der Integrationsbeziehung Ordnen in genau dieser Reihenfolge. Damit erzielt die Modellierung in HyperFeatuRSEB dieselbe Semantik wie die in FeatuRSEB, nur ohne die künstliche Abstraktion der Erweiterungsstelle.

Abbildung 7.8 zeigt, wie das Komponentensystem catan::spiel von FeatuRSEB aus Abbildung 4.7 auf Seite 53 in HyperFeatuRSEB modelliert wird. Die Klasse ZugAbschließen demonstriert den Unterschied. HyperFeatuRSEB benötigt kein Framework, bestehend aus dem Entwurfsmuster Abstrakte Fabrik und der Unterklasse ZugAbschließen2, um die Variabilität zu modellieren. Entsprechend erübrigt sich im Produkt-Engineering die Spezialisierung des Frameworks. Statt dessen werden bei Auswahl des Merkmals Fortschrittskarten die Klassen ZugAbschließen aus Kern und Fortschrittskarten und damit die Implementierungen der Operation führeAus() verschmolzen. Die Reihenfolge, in der die Implementierungen zur Ausführung kommen, steuert die Integrationsbeziehung Ordnen. Damit

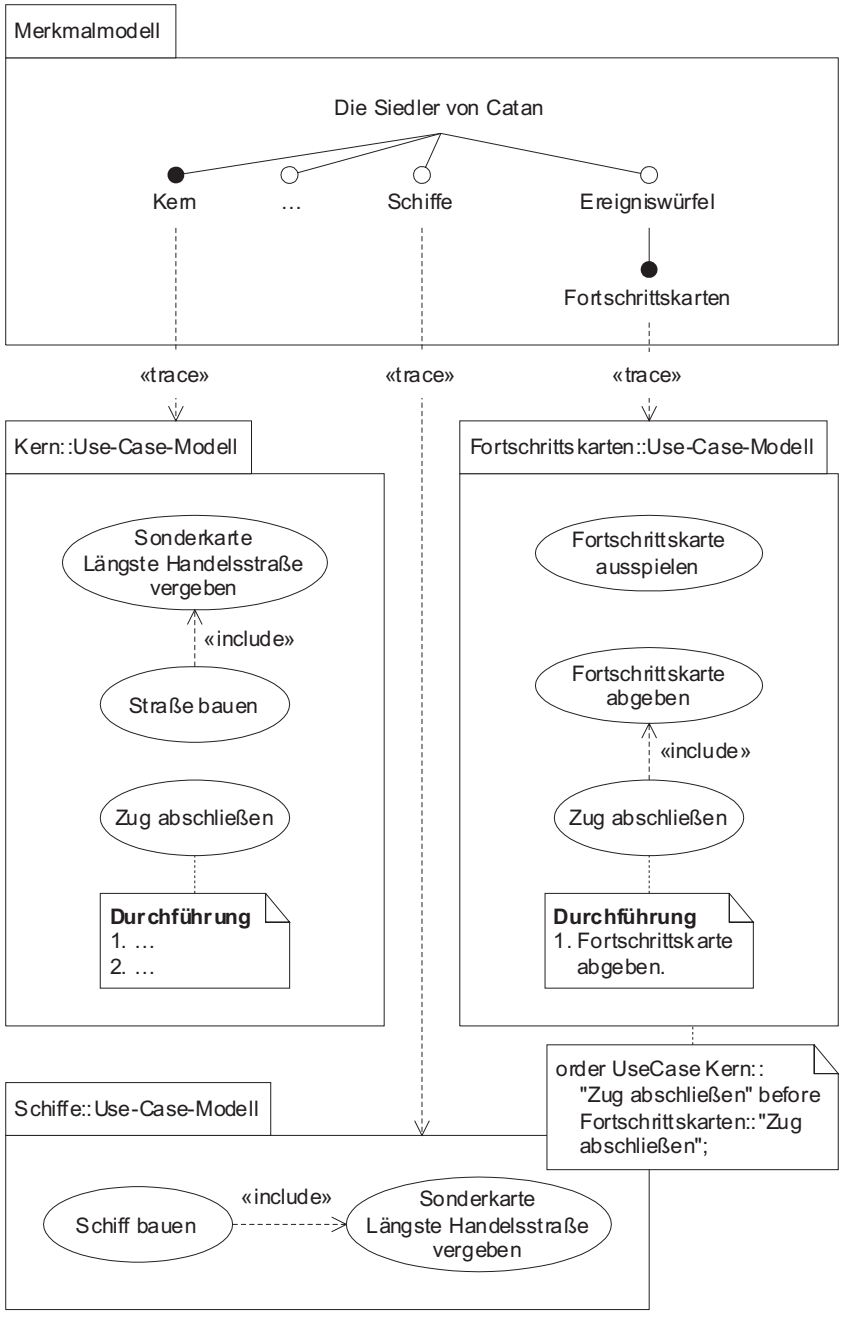

Abbildung 7.7: Modellierung von Variabilität im Use-Case-Modell

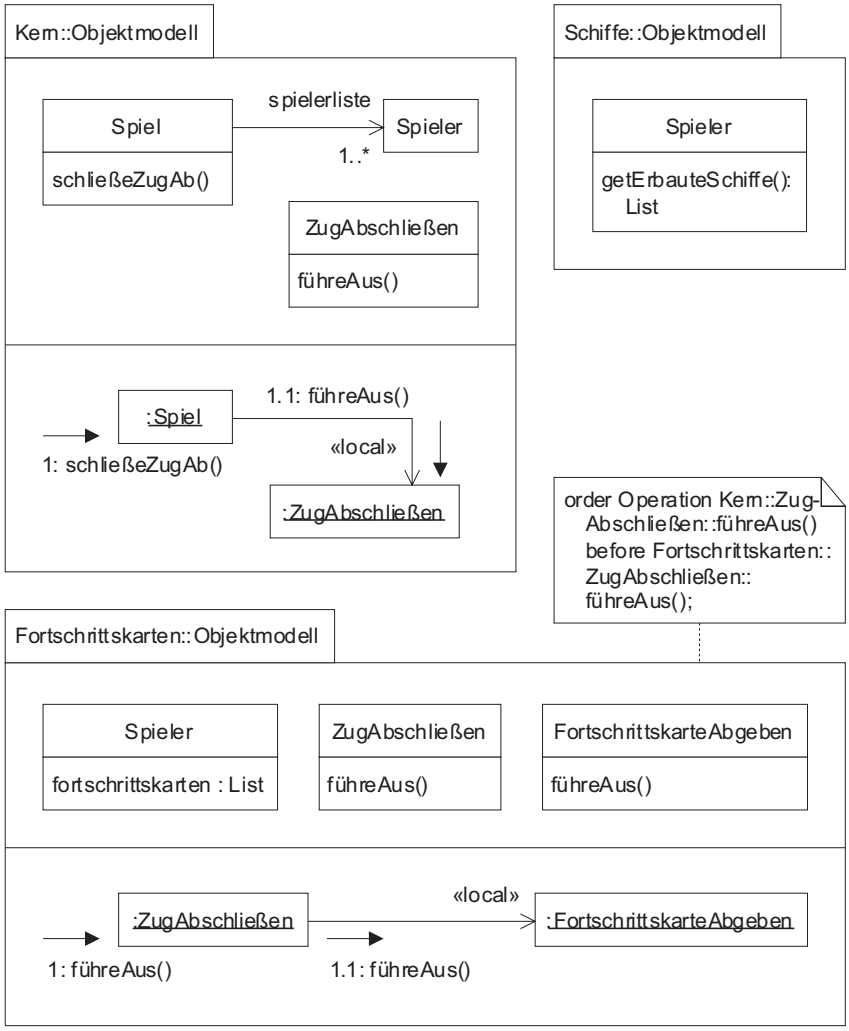

Abbildung 7.8: Modellierung von Variabilität im Objektmodell

erzielt die Modellierung in HyperFeatuRSEB dieselbe Semantik wie die in FeatuRSEB, nur ohne künstliche Abstraktionen. Außerdem hebt HyperFeatuRSEB zusätzlich die Überschneidung der drei Merkmale in der Klasse Spieler auf, wozu in FeatuRSEB weitere künstliche Abstraktionen hätten ergänzt werden müssen.

Schließlich verbucht HyperFeatuRSEB bei der Modellierung von Variabilität noch einen wesentlichen Vorteil: Während FeatuRSEB diesbezüglich nur Aussagen für Use-Case- und Klassendiagramme trifft, dehnt HyperFeatuRSEB die Modellierung von Variabilität durch Einbeziehung von Hyper/UML auf nahezu alle Diagrammarten der UML aus. Ausgenommen sind Komponenten- und Verteilungsdiagramme, weil die derzeitige Hyper/UML-Version für die dort verwendeten Elemente keine Abbildungsvorschrift definiert (vgl. Abschnitt 6.2.1). Beispiele zur Modellierung von Variabilität mit Hyper/UML enthielten bereits die zahlreichen Abbildungen in den Unterkapiteln 6.6 bis 6.9. Einige weitere Beispiele folgen in den Unterkapiteln 8.2 und 8.3. Letzteres Unterkapitel erläutert darüber hinaus Aspekte der Wartung bei den mit HyperFeatuRSEB entwickelten Produktlinien.

7.3 Produkt-Engineering

Ziel des Produkt-Engineerings in HyperFeatuRSEB ist die automatisierte Serienfertigung von Systemen für Kunden auf Basis einer Produktlinie. Durch die Automatisierung können neue Systeme in kurzer Zeit ausgeliefert werden. Pro Kunde und System gibt es einen Produkt-Engineering-Prozeß.

7.3.1 Ablauf

Das Produkt-Engineering gliedert sich in drei Phasen, die zu entsprechenden Ergebnissen führen (vgl. Abbildung 7.9).

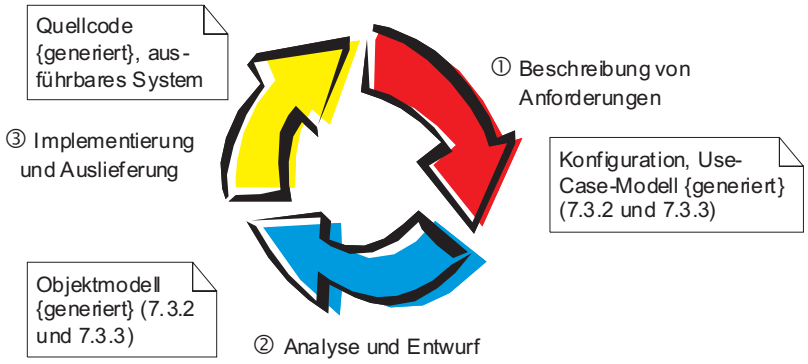

Abbildung 7.9: Ablauf Produkt-Engineering

Phase 1: Beschreibung von Anforderungen. In dieser Phase sind die Anforderungen des Kunden aufzunehmen und mit den Anforderungen, die die Produktlinie bereits erfüllt, abzugleichen. Das Produkt-Engineering in HyperFeatuRSEB faßt eine Kunden-Anforderung primär als Merkmal auf. Demnach besteht die Anforderungsbeschreibung aus einer Liste von Merkmalen (Konfiguration genannt), die der Kunde selbständig oder nach Beratung aus dem Merkmalmodell der Produktlinie zusammenstellt. Die Konfiguration wird einem Generator übergeben, der sie auf Korrektheit überprüft: Sie muß alle gemeinsamen Merkmale enthalten und darf keine Hierarchie- und Abhängigkeitsbeziehung zwischen Merkmalen verletzen. Anschließend fertigt der Generator aus der Konfiguration das vom Kunden gewünschte System, bestehend aus Use-Case-Modell, Objektmodell und Quellcode. Jetzt kann der Kunde sein System testen und prüfen, ob es seinen Anforderungen tatsächlich genügt. Nur wenn der Kunde noch individuelle Anpassungen wünscht, beschreibt das Produkt-Engineering diese durch Use-Cases. Als Diskussionsgrundlage dient das generierte Use-Case-Modell. Details zur teilautomatisierten Serienfertigung folgen in Abschnitt 7.3.3.

Phase 2: Analyse und Entwurf. Hier entwirft das Produkt-Engineering für die individuellen Anforderungen des Kunden neue Klassen, ausgehend vom generierten Objektmodell. Äußert der Kunde hingegen keine Sonderwünsche, fällt die Phase weg.

Phase 3: Implementierung und Auslieferung. Zum Schluß folgen Test und Auslieferung des Systems an den Kunden. Gegebenenfalls sind vorher individuelle Klassen zu implementieren.

7.3.2 Systeme aus der Serienfertigung

Ein in Serie gefertigtes System ist in HyperFeatuRSEB – von der technischen Seite her betrachtet – ein Hypermodul, das Merkmal-Belange aus dem Hyperraum einer Produktlinie integriert. Das Hypermodul läßt sich aufgrund der Konfiguration des Systems, d.h. der Liste der vom Kunden aus dem Merkmalmodell der Produktlinie gewählten Merkmale, in vier Schritten spezifizieren:

1. Jedes Merkmal in der Konfiguration über die *trace*-Abhängigkeitsbeziehung zum Merkmal-Belang in der Merkmal-Dimension des Hyperraums der Produktlinie verfolgen

2. Ermittelte Merkmal-Belange in die Hypermodul-Spezifikation aufnehmen. Dadurch werden alle einem Belang zugeordneten und in der entsprechenden Hyperebene gekapselten Elemente in das Hypermodul integriert.

3. Integrationsbeziehung in der Hypermodul-Spezifikation angeben, die die Integrationsstrategie verwirklicht. Da auf oberster Ebene Hyperebenen integriert werden, reicht die Angabe einer einzigen Verschmelzen-Beziehung aus. Sie verschmilzt die Hyperebenen der ermittelten Merkmal-Belange miteinander.

4. Spezielle, von der Integrationsstrategie abweichende Integrationsbeziehungen der vom Kunden gewählten Merkmale in das Hypermodul aufnehmen

Das Aufstellen der Hypermodul-Spezifikation kann durch ein Werkzeug beziehungsweise einen Generator erfolgen, sofern die Konfiguration maschinenverarbeitbar vorliegt. Genau genommen muß der Generator zwei Hypermodul-Spezifikationen erzeugen. Die erste integriert die Use-Case- und Objektmodelle der Hyperebenen und muß deshalb in der Syntax von Hyper/UML formuliert sein. Die zweite Spezifikation integriert den Quellcode der Hyperebenen. Sie folgt der Hyper/J-Syntax oder, wenn die Produktlinie nicht in Java implementiert ist, der Syntax einer Abbildung des Hyperspace-Ansatzes auf die in der Produktlinie verwendete Programmiersprache.

Entscheidet sich ein Kunde beispielsweise für die Merkmale Kern und Fortschrittskarten aus den am Ende von Unterkapitel 7.2 gezeigten Ausschnitten der Produktlinie „Die Siedler von Catan“, erstellt der Generator folgende Hypermodul-Spezifikation (Hyper/J-Syntax):

```
hypermodule Beispiel
    hyperslices:
        Merkmal.Kern,
        Merkmal.Fortschrittskarten;
    relationships:
        // Integrationsstrategie
        merge Hyperslice Kern, Fortschrittskarten;
        // Spezielle Integrationsbeziehungen (aus Fortschrittskarten)
        order UseCase Kern::"Zug abschließen" before
            Fortschrittskarten::"Zug abschließen";
        order Operation Kern::ZugAbschließen::führeAus() before
            Fortschrittskarten::ZugAbschließen::führeAus();
end hypermodule;
```

Die Spezifikation legt fest, daß die Modelle der in den Abbildungen 7.7 auf Seite 153 und 7.8 auf Seite 154 gezeigten Hyperebenen Kern und Fortschrittskarten zu einem System zu integrieren sind.

7.3.3 Systeme mit individuellen Anforderungen

Anders als bei der Serienfertigung von Autos ist bei Software-Systemen nie auszuschließen, daß Kunden noch eine individuelle Anpassung ihres Systems fordern (vgl. Unterkapitel 1.1 „Szenario 2“). Die Fertigung des Systems, also das Produkt-Engineering, läuft dann nicht mehr automatisiert, sondern nur noch teilautomatisiert ab. Dabei ist es wünschenswert, daß der manuelle Teil der Fertigung einen möglichst geringen Aufwand verursacht. FeatuRSEB bietet zwei Alternativen an, um auf individuelle Anforderungen zu reagieren:

- *Ausbau der Produktlinie.* Das Produktlinien-Engineering arbeitet die individuellen Anforderungen in die Produktlinie ein. Daraufhin fertigt das Produkt-Engineering das System auf Basis der erweiterten Produktlinie neu.
- *Änderung des generierten Systems.* Das Produkt-Engineering arbeitet die individuellen Anforderungen in die vom Generator erzeugten Modelle und den generierten Quellcode ein.

Die erste Alternative ist mit hohem Aufwand verbunden (vgl. Abschnitt 4.5.3), den Kunden nur bedingt bereit sind zu bezahlen. Einen Teil der Mehrkosten könnte deshalb der Hersteller der Produktlinie übernehmen, denn schließlich bringt ihm der Ausbau ebenfalls einen Nutzen, wenn in Zukunft weitere Kunden mit den gleichen Anforderungen an ihn herantreten. Wie wahrscheinlich dieser Fall ist, muß im Einzelfall eine Marktforschung zeigen. Scheut der Hersteller die mit einem Ausbau der Produktlinie verbundenen Risiken oder handelt es sich um vergleichsweise kleine Anpassungen, bleibt noch die zweite Alternative. Sie erfordert einen geringeren Aufwand, mündet aber in einer Einbahnstraße. Möchte der Kunde einige Zeit später ein zusätzliches Merkmal für sein System kaufen, muß das Produkt-Engineering die am früher generierten System erfolgten Anpassungen manuell auf das neu generierte System übertragen.

HyperFeatuRSEB verbessert die zweite Alternative durch Ausnutzung des Hyperspace-Ansatzes so, daß eine Neugenerierung angepaßter Systeme mit geänderter Konfiguration jederzeit und ohne nachträgliche, manuelle Eingriffe möglich ist. Wie sieht dieser dritte Weg aus? Er besteht aus folgenden Schritten, die das Produkt-Engineering in Abstimmung mit dem Produktlinien-Engineering durchführt:

1. Optionales Merkmal zum Merkmalmodell der Produktlinie hinzufügen und die Beziehungen des neuen Merkmals zu den bestehenden Merkmalen definieren. Die Konfiguration des Kunden ist um das neue Merkmal zu ergänzen.
2. Merkmal-Belang zur Merkmal-Dimension des Hyperraums der Produktlinie hinzufügen. Eine *trace*-Abhängigkeitsbeziehung muß das neue Merkmal mit dem neuen Belang verknüpfen.
3. Hyperebene für den neuen Merkmal-Belang zur Produktlinie hinzufügen
4. Individuelle Anforderungen des Kunden in der neuen Hyperebene modellieren und implementieren. Währenddessen gegebenenfalls spezielle, von der Integrationsstrategie abweichende Integrationsbeziehungen beim neuen Merkmal eintragen.
5. Erneut das vom Kunden gewünschte System fertigen, diesmal mit der oben geänderten Konfiguration. Schritt 4 und 5 solange wiederholen, bis der Kunde das System abnimmt.

Interessant ist vor allem der vierte Schritt. Die Möglichkeiten, in der die neue Hyperebene in die bestehende Produktlinie eingreifen und notwendige Anpas-

sungen vornehmen kann, leiten sich primär aus den von Hyper/UML und Hyper/J bereitgestellten Arten von Integrationsbeziehungen ab. Zuvor sind jedoch zwei Kategorien individueller Anforderungen zu unterscheiden:

- *Positive Anforderungen* fügen weitere Funktionalität zum System eines Kunden hinzu oder ändern dort Funktionalität ab.
- *Negative Anforderungen* deaktivieren oder entfernen im System die nicht vom Kunden gewünschte Funktionalität.

Dazu ein Beispiel. Angenommen, ein Kunde bestellt ein System aus der Produktlinie „Die Siedler von Catan“ mit den Merkmalen Kern und Schiffe und wünscht darüber hinaus die folgenden individuellen Anpassungen: (a) Spieler sollen für den Bau einer Stadt 1 Getreide und 1 Erz mehr bezahlen; (b) Spieler dürfen ihre Schiffe nicht versetzen. Die erste Anpassung ist nach obiger Definition eine positive Anforderung, die zweite eine negative Anforderung.

Positive individuelle Anforderungen sind in der neuen Hyperebene relativ einfach zu modellieren und zu implementieren, weil für die erforderlichen Eingriffe die Integrationsstrategie, das Verschmelzen, meist ausreicht. Gegebenenfalls müssen noch Ordnen- oder Summieren-Integrationsbeziehungen ergänzt werden. So auch für den Beispiel-Kunden. Für ihn wurde das Merkmalmodell der Produktlinie in Abbildung 7.10 um das optionale Merkmal Kunde X ergänzt. Die dazugehörige gleichnamige Hyperebene definiert die Klasse Stadt, die in der Operation baukosten() den vom Kunden geforderten Anstieg der Baukosten für eine Stadt implementiert. Bei der Fertigung des Systems verschmelzen die beiden Hyperebenen Kern und Kunde X, und damit die Deklarationen der Klasse Stadt sowie die Implementierungen von baukosten(). Durch die Integrationsbeziehung Summieren liefert die verschmolzene Implementierung eine Rohstoffmenge von 3 Getreide und 4 Erz zurück.

Negative individuelle Anforderungen müssen über die Integrationsbeziehung Ersetzen modelliert und implementiert werden, deren Mächtigkeit gegenüber dem Verschmelzen in den aktuellen Versionen von Hyper/UML und Hyper/J geringer ausfällt. So kommt es vor, daß nicht gewünschte Funktionalität zwar optisch aus der Benutzeroberfläche eines Systems verschwindet oder dort deaktiviert ist, aber der Quellcode für die Funktionalität nach wie vor im System existiert. Das Ziel „Serienfertigung schlanker Systeme“ wäre in diesem Fall nur teilweise erfüllt. Im Beispiel ersetzt die leere Implementierung der Operation Kunde X::Spiel::versetzeSchiff() bei der Fertigung die Implementierung aus Schiffe. Damit stößt ein Aufruf des entsprechenden Menüpunktes nicht mehr die Instanziierung der Klasse SchiffVersetzen und den Aufruf von führeAus() an. Die Klasse ist allerdings weiterhin im System enthalten.

Aufgrund der Modellierung und Implementierung individueller Anforderungen in einer eigenen Hyperebene gehen die Änderungen auch bei einem neuen Fertigungslauf mit eventuell geänderter Konfiguration nicht verloren. Mehr noch: Auch andere Kunden können das Merkmal Kunde X für ihr System kaufen und von den Änderungen profitieren.

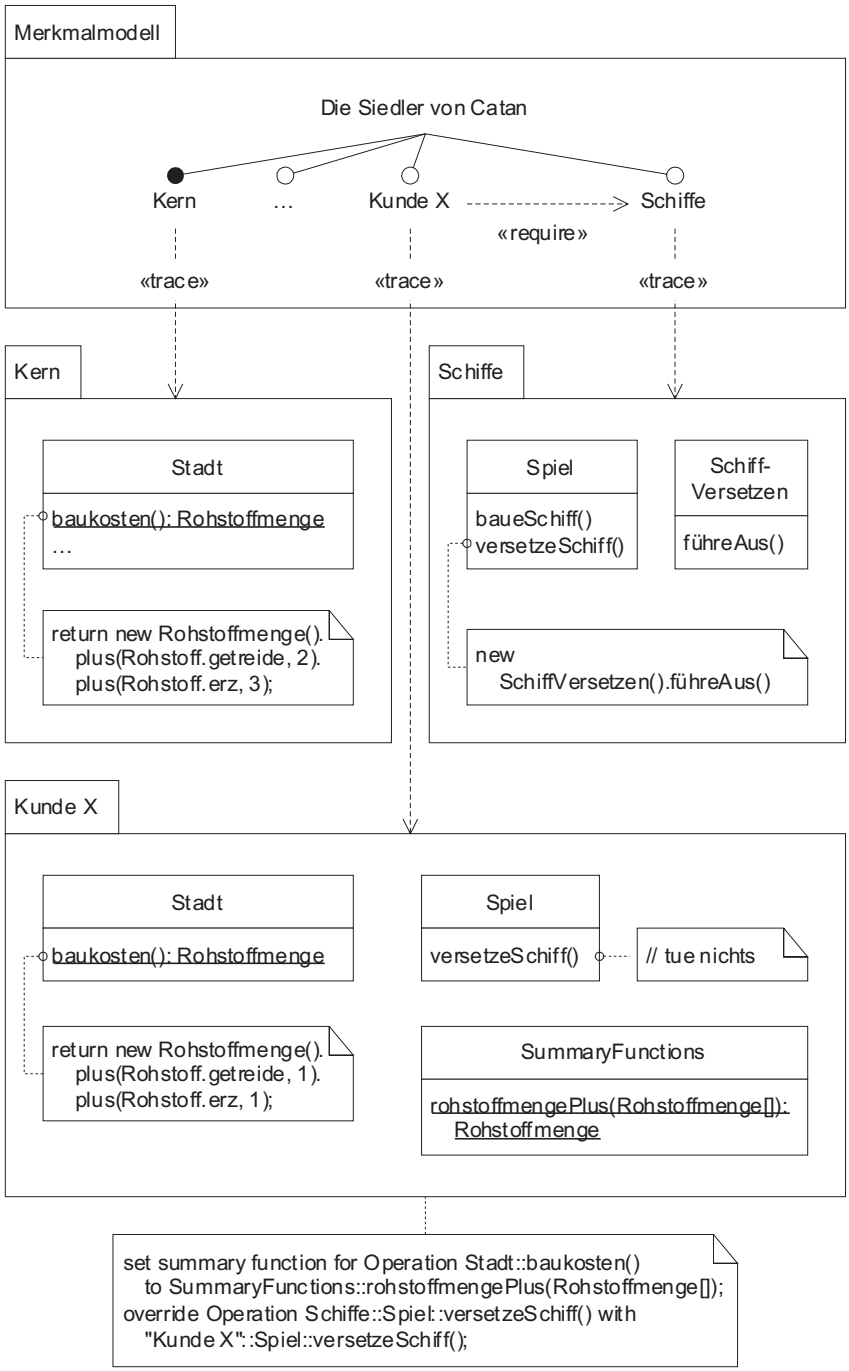

Abbildung 7.10: Beispiel System mit individuellen Anforderungen

7.4 Werkzeuge

Die Entwicklung von Produktlinien und die Serienfertigung von Produkten mit HyperFeatuRSEB müssen Werkzeuge unterstützen. Die Modellierung von Merkmalen im Produktlinien-Engineering kann mit Hilfe von AmiEddi geschehen (vgl. Unterkapitel 4.4). Die Modellierung von Hyperraum und Hyperebenen erfordert ein Werkzeug für Hyper/UML, wie es Unterkapitel 6.10 skizziert. Die Realisierung eines solchen Werkzeugs ist eine Aufgabe für zukünftige Arbeiten. Zur Implementierung von Hyperebenen mit Hyper/J und Java reichen vorhandene Entwicklungsumgebungen prinzipiell aus. Wie dies z.B. für IBM VisualAge for Java [VAJ] aussehen kann und was über das gebotene Maß hinaus nützlich wäre, beschreibt Halle in [Hal01, Abschnitt 5.4.1 und Unterkapitel 7.3].

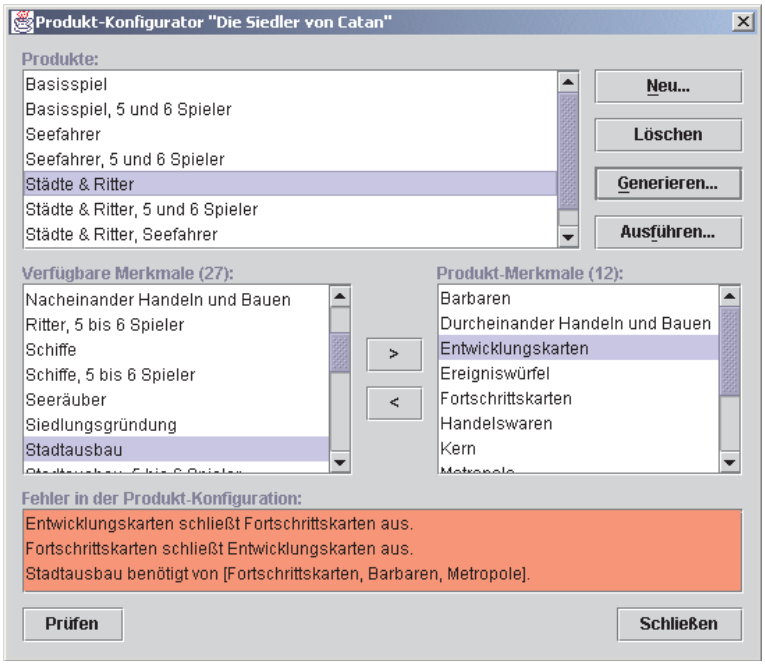

Abbildung 7.11: Produkt-Konfigurator

Das Produkt-Engineering benötigt ein Werkzeug, das die drei Schritte zur Serienfertigung von Systemen durchführt: (1) System konfigurieren, (2) Konfiguration überprüfen und (3) System zusammensetzen (vgl. Abbildung 2.6 auf Seite 27). Die ersten zwei Schritte unterstützt der Produkt-Konfigurator, der im Rahmen dieser Arbeit entwickelt wurde. Er präsentiert sich an der Benutzeroberfläche durch den in Abbildung 7.11 gezeigten Dialog. Dort legt der Kunde im oberen Bereich ein neues Produkt an und ordnet diesem im mittleren Bereich des Dialogs die gewünschten Merkmale aus dem Merkmalmodell der Produktlinie zu. Ein Klick auf Prüfen veranlaßt den Konfigurator, die Konfiguration auf Fehler hin zu analysieren und das Ergebnis dem Kunden mitzuteilen. Generieren leitet zum dritten Schritt der Serienfertigung über. Darin erstellt der

Produkt-Konfigurator die Hypermodul-Spezifikation für das zu fertigende System (vgl. Abschnitt 7.3.2) und startet das Hyper/J-Werkzeug. Das verarbeitet die Spezifikation und setzt den Quellcode aus den Hyperebenen der gewählten Merkmale zusammen (vgl. Abschnitt 5.3.9). Die Modelle in den Hyperebenen werden mangels eines Hyper/UML-Werkzeugs noch nicht integriert.

Der Produkt-Konfigurator ist für jede mit HyperFeatuRSEB entwickelte Produktlinie einsetzbar, weil er beliebige Merkmalmodelle, nicht nur das der Produktlinie „Die Siedler von Catan“, interpretiert. Eine zukünftige Erweiterung des Konfigurators ist der Entwurf einer kundenfreundlicheren Benutzeroberfläche, die die Konfiguration von Systemen ähnlich wie beim Volkswagen-Konfigurator über *check boxes* und *radio buttons* erlaubt (vgl. Abbildung 2.5 auf Seite 27). Weitere Erläuterungen zum Produkt-Konfigurator, insbesondere zu seiner Implementierung, finden sich in [Hal01, Unterkapitel 6.2 bis 6.4].

Produktlinien-Entwicklung und Serienfertigung mit HyperFeatuRSEB würden sich noch weitaus stärker vereinfachen, wenn alle der angesprochenen einzelnen Werkzeuge miteinander integriert wären. Die Grundlage hierfür ist ein gemeinsames Repository, auf das alle Werkzeuge zugreifen. Einen wichtigen Schritt dorthin bringt die nächste Version von AmiEddi. Sie wird das Merkmalmodell nicht mehr in einem proprietären binären Format, sondern in XML (Extensible Markup Language) abspeichern.

7.5 Zusammenfassung

Die HyperFeatuRSEB-Methode liefert nach Hyper/UML den zweiten und auch den abschließenden Beitrag zur Zielsetzung dieser Arbeit. Die Methode integriert den Hyperspace-Ansatz mit seinen Abbildungen in die FeatuRSEB-Methode. Dadurch wird die objektorientierte, UML-basierte Entwicklung solcher Produktlinien möglich, aus denen schlanke Systeme in Serie gefertigt werden können, ohne daß die Produktlinien wie bei FeatuRSEB zu komplex und unwartbar werden – Behauptungen, deren Richtigkeit das folgende Kapitel überprüft.

Kapitel 8

Überprüfung der Zielsetzung

Die Zielsetzung dieser Arbeit zerfällt in vier einzelne Ziele mit den Überschriften: Serienfertigung schlanker Systeme, Komplexität entwickelter Produktlinien, Wartbarkeit entwickelter Produktlinien und Skalierbarkeit der Methode (vgl. Unterkapitel 1.3). Die folgenden vier Unterkapitel behandeln jeweils eines der Ziele und überprüfen, ob das Ziel durch den Beitrag der Arbeit, bestehend aus der HyperFeatuRSEB-Methode und Hyper/UML, erreicht wird.

Keines der formulierten Ziele läßt sich auf einer rein theoretischen Basis überprüfen. Deshalb zieht dieses Kapitel erneut die Fallstudie „Die Siedler von Catan“ aus Kapitel 3 heran, um das Erreichen der Ziele durch Beispiele aus der Praxis zu untermauern. Da sich allein aufgrund des Umfangs der Studie zahlreiche Beispiele finden lassen, aus Platzgründen aber nicht alle angeführt werden können, beschränkt sich das Kapitel auf diejenigen Beispiele, anhand derer bereits die FeatuRSEB-Methode bewertet wurde (vgl. Unterkapitel 4.5). Das verdeutlicht auch die Unterschiede zwischen beiden Methoden und zeigt, wie diese Arbeit den Stand der Technik in der objektorientierten Entwicklung von Produktlinien verbessert.

8.1 Serienfertigung schlanker Systeme

Die Zielsetzung stellt folgende Behauptung auf: „Die mit HyperFeatuRSEB entwickelten Produktlinien eignen sich zur Serienfertigung von Systemen. Ein Generator setzt die Produktlinien-Komponenten automatisiert zu Systemen zusammen. Die Systeme sind schlank. Sie enthalten nur genau die Funktionalität beziehungsweise Komponenten, die die Kunden gefordert haben.“ (vgl. Unterkapitel 1.3)

Serienfertigung bedeutet automatisiertes Produkt-Engineering. Unterkapitel 7.3 beschreibt den Ablauf des Produkt-Engineerings in HyperFeatuRSEB. Im Idealfall ist die einzige dort manuell auszuführende Tätigkeit das Konfigurieren des vom Kunden gewünschten Systems, sprich das Auswählen der Merkmale, die das System aufweisen soll. Ab dann läuft alles weitere automatisch ab: Der Produkt-Konfigurator überprüft die Konfiguration, erstellt die Hypermodul-Spezifikation

für das zu fertigende System und startet das Hyper/J-Werkzeug. Das wiederum setzt das System aus den Produktlinien-Komponenten, sprich Hyperebenen zusammen (vgl. Unterkapitel 7.4). Daran anschließend kann das System sofort ausgeliefert und beim Kunden in Betrieb genommen werden. Bezogen auf die Produktlinie „Die Siedler von Catan“ dauert der automatisierte Teil der Fertigung je nach Anzahl der Merkmale bis zu 10 Minuten[1] pro System.

Wenn der Kunde hingegen individuelle Anforderungen an sein System stellt, die die Produktlinie nicht abdeckt, verzögert sich die Auslieferung des Systems entsprechend. Dennoch bietet HyperFeatuRSEB auch für solche Fälle eine Lösung an, die sich gut in den Serienfertigungsgedanken integriert. Dazu werden individuelle Anforderungen als neues Merkmal zur Produktlinie hinzugefügt und in einer eigenen Hyperebene modelliert und implementiert. Vor dem nächsten Fertigungslauf wird die Konfiguration um das neue Merkmal ergänzt, und der Produkt-Konfigurator kann unverändert benutzt werden (vgl. Abschnitt 7.3.3).

Schlanke Systeme bedeutet, daß der Generator in ein zu fertigendes System wirklich nur so viele Klassen, Attribute und Operationen aufnimmt, wie zur Erfüllung der vom Kunden geforderten Funktionalität gerade eben notwendig sind. In HyperFeatuRSEB wird die Funktionalität von Produktlinien durch Merkmale beschrieben. Modell und Implementierung eines Merkmals kapselt jeweils eine eigene Hyperebene (vgl. Unterkapitel 7.2). Der Produkt-Konfigurator steuert die Fertigung über die Hypermodul-Spezifikation derart, daß ein System nur aus den Hyperebenen zusammengesetzt wird, deren zugehöriges Merkmal in der Konfiguration des Systems aufgeführt ist. Den erzielten Effekt auf die Größe der gefertigten Systeme demonstriert Tabelle 8.1.

Produkt	Anzahl Merkmale	Anzahl Klassen	Bytecode
Produktlinie	100%	100%	100%
Basisspiel	13%	52%	44%
... + 5 und 6 Spieler	21%	53%	46%
Die Seefahrer	36%	58%	52%
... + 5 und 6 Spieler	51%	60%	55%
Städte & Ritter	31%	79%	75%
... + 5 und 6 Spieler	46%	80%	77%
Städte & Ritter + Die Seefahrer	44%	82%	80%
... + 5 und 6 Spieler	64%	84%	83%

Tabelle 8.1: Serienfertigung schlanker Systeme

Die Tabelle listet für acht aus der Produktlinie „Die Siedler von Catan“ gefertigte Produkte Angaben zur Merkmalanzahl, Klassenanzahl und Größe des Bytecodes auf. Die Angaben verstehen sich prozentual zur Produktlinie. So enthält das Produkt „Die Seefahrer“ 36 Prozent der Merkmale der Produktlinie, 58 Prozent ihrer Klassen und 52 Prozent der Bytecode-Größe der Produktlinie. Im Vergleich dazu kommt das Produkt „Städte & Ritter“ bei nur 31 Prozent der Merkmale auf eine Klassenanzahl von 79 Prozent. Das liegt daran, daß die Konfigurationen

[1] Gemessen auf einer zum Zeitpunkt des Schreibens gängigen Hardware-Plattform.

beider Produkte disjunkte Teilmengen der Produktlinien-Merkmale enthalten. Offensichtlicher ist dagegen die Reduzierung der Größe der einzelnen Produkte gegenüber der Größe der Produktlinie um bis zu 50 Prozent.

HyperFeatuRSEB ermöglicht die Fertigung schlanker Systeme. Voraussetzung ist jedoch, daß das Produktlinien-Engineering aus den Anforderungen an eine Produktlinie die „richtigen“ Merkmale ableitet und die zugeordneten Hyperebenen weitgehend unabhängig voneinander modelliert und implementiert – denn bei der Fertigung können nur vollständige Merkmale beziehungsweise Hyperebenen weggelassen werden. Benötigt ein Kunde nur einen Teil an Funktionalität aus einer Hyperebene, muß die gesamte Ebene mit in das System aufgenommen werden, und das Kriterium „Schlanke Systeme“ wird verfehlt. Eine Diskussion über eine weitere Auftrennung der Merkmale der „Siedler von Catan“ zwecks Fertigung noch schlankerer Systeme führt [Hal01, Unterkapitel 7.5].

8.2 Komplexität entwickelter Produktlinien

Die Zielsetzung stellt folgende Behauptung auf: „Die Komplexität der mit HyperFeatuRSEB entwickelten Produktlinien bleibt trotz feingranular modellierter und implementierter Komponenten handhabbar und damit für Entwickler beherrschbar.“ (vgl. Unterkapitel 1.3)

Komplexität, ihre Handhabbarkeit und Beherrschbarkeit sind Eigenschaften eines Software-Systems, die objektiv schwer zu bewerten sind, weil jeder Entwickler Komplexität subjektiv anders empfindet und beurteilt. Worauf ein erfahrener Entwickler nur einen kurzen Blick wirft, das versteht ein Einsteiger erst nach längerem Nachdenken. Dennoch sollen in diesem Unterkapitel einige objektive Argumente angeführt werden, die Anzeichen dafür liefern, daß die Komplexität der mit HyperFeatuRSEB entwickelten Produktlinien handhabbar ist, vor allem im direkten Vergleich zu Produktlinien, bei deren Entwicklung FeatuRSEB angewendet wurde.

Vergleich von FeatuRSEB und HyperFeatuRSEB

Die Wiederverwendung von Produktlinien-Komponenten in Systemen geschieht in beiden Entwicklungsmethoden über Merkmale. Um die Wiederverwendbarkeit der Komponenten zu erhöhen und die Serienfertigung schlanker Systeme zu ermöglichen, dürfen sich in den Komponenten keine Merkmale überschneiden. Es entstehen feingranulare Komponenten.

FeatuRSEB löst das Überschneidungsproblem, indem eine Produktlinie mit künstlichen Abstraktionen angereichert wird: Erweiterungsstellen und *extend*-Beziehungen im Use-Case-Modell, Entwurfsmuster und Vererbung im Objektmodell. Die Abstraktionen leisten keinen Beitrag zur eigentlichen Funktionalität der Produktlinie und führen dazu, daß die Funktionalität in viele kleine, über die Produktlinie verteilte Fragmente aufgetrennt wird (vgl. Abschnitt 4.5.2).

HyperFeatuRSEB löst das Überschneidungsproblem auf eine andere Art. Die Methode nutzt den Hyperspace-Ansatz, um Modell und Implementierung einer

Produktlinie zusätzlich merkmalorientiert zu zerlegen. Danach werden alle Teile des Modells und der Implementierung, die für ein Merkmal relevant sind, in einer eigenen Hyperebene gekapselt. Entwickler betrachten und bearbeiten die Funktionalität der Produktlinie über die Hyperebenen, d.h. sie haben in erster Linie eine merkmalorientierte Sicht auf die Produktlinie. Das Zusammenführen der getrennten Hyperebenen zu ausführbaren Systemen erfolgt über Integrationsbeziehungen (vgl. Unterkapitel 7.2).

Die oben aufgezählten künstlichen Abstraktionen treten in HyperFeatuRSEB nicht auf. Dies wird deutlich, wenn man dieselben Merkmale mit beiden Methoden modelliert und die Ergebnisse gegenüberstellt (vgl. für das Use-Case-Modell die Abbildungen 4.4 auf Seite 49 und 7.7 auf Seite 153; für das Objektmodell die Abbildungen 4.7 auf Seite 53 und 7.8 auf Seite 154). Die Vermeidung künstlicher Abstraktionen senkt die Anzahl an Elementen, die Entwickler in den Modellen und der Implementierung einer mit HyperFeatuRSEB entwickelten Produktlinie verstehen und warten müssen. Mithin sinkt die Komplexität der Produktlinie bezüglich ihrer Elementanzahl. Zudem bietet sich Entwicklern durch die Hyperebenen eine merkmal- beziehungsweise wiederverwendungsorientierte Sicht auf die Funktionalität der Produktlinie, was sich in der Fallstudie „Die Siedler von Catan“ nach Aussagen aller Beteiligten positiv auf die Überschaubarkeit der Produktlinie auswirkte und Änderungen sowie die Weiterentwicklung vereinfachte.

Den ersten erfreulichen Zwischenstand in der Bewertung der HyperFeatuRSEB-Methode scheinen jedoch drei Argumente zu schmälern, die im folgenden diskutiert werden sollen.

Argument 1: Fragmentierung der Funktionalität

HyperFeatuRSEB behebt nicht die an FeatuRSEB kritisierte Fragmentierung der Funktionalität einer Produktlinie. Statt dessen ist die Funktionalität jetzt über Hyperebenen hinweg fragmentiert. Beispielsweise verteilt sich in Abbildung 7.8 auf Seite 154 die Definition der Klasse Spieler auf alle drei Hyperebenen. Das hat einen abgewandelten Jo-Jo-Effekt [TGP89] zur Folge: Damit ein Entwickler eine Produktlinie (Klasse) insgesamt versteht, muß er mehrere Hyperebenen (Ober- und Unterklassen) aufsuchen und die Integration der Elemente (Überschreiben von Operationen) im Kopf durchgehen. Zudem steigt durch die Fragmentierung die absolute Anzahl an Elementen in der Produktlinie wieder an. Das Beispiel enthält drei Spieler-Klassen, die Variante des Modells in FeatuRSEB hingegen nur eine (vgl. Abbildung 4.7 auf Seite 53).

Gegendarstellung: Die Art und Weise, auf die Funktionalität in FeatuRSEB und HyperFeatuRSEB fragmentiert ist, unterscheidet sich in einem wesentlichen Punkt. Während FeatuRSEB Funktionalität auf künstliche Abstraktionen verteilt, die sowohl von ihrem Namen her als auch von ihrer Anordnung im Modell und Implementierung einer Produktlinie nicht unbedingt Rückschlüsse aufeinander zulassen, verteilt HyperFeatuRSEB Funktionalität auf semantisch übereinstimmende Elemente, z.B. Klassen und Operationen gleichen Namens. Dies ist eine Konsequenz der Integrationsstrategie von HyperFeatuRSEB, die übereinstimmende Elemente standardmäßig verschmilzt.

Die semantische Übereinstimmung erleichtert im Gegensatz zu künstlichen Abstraktionen die Verständlichkeit der Produktlinie. In FeatuRSEB müssen Entwickler zusätzlich zur eigentlichen Funktionalität die Konzepte hinter den künstlichen Abstraktionen verstehen. Dagegen verbirgt sich in HyperFeatuRSEB hinter einem mehrfach auftretenden Element wie z.B. der Klasse Spieler immer dasselbe, nur einmal zu verstehende Konzept. Das mildert den Jo-Jo-Effekt. Um ihn weiter abzuschwächen, ist zudem eine Werkzeugunterstützung denkbar, die Elemente partiell miteinander integriert, so daß Entwickler dies nicht manuell im Kopf leisten müssen. Unter den genannten Gesichtspunkten relativieren sich die gestiegene Elementanzahl und der damit verbundene Komplexitätsanstieg wieder. Die Fragmentierung ist mit HyperFeatuRSEB vorteilhafter gelöst als mit FeatuRSEB.

Argument 2: Deklarative Vollständigkeit von Hyperebenen

Die Elementanzahl erhöht sich nicht nur wegen der Fragmentierung der Funktionalität, sondern auch wegen der Forderung nach deklarativer Vollständigkeit der Hyperebenen. Viele Hyperebenen deklarieren Elemente nur, um sie zu nutzen, nicht aber zu erweitern. Eine Einschätzung der Dimension des Anstiegs bei der Elementanzahl erlaubt die Fallstudie. Ihr Objektmodell besteht aus 300 „echten“ Klassen. Erweiterungen und deklarative Vollständigkeit erhöhen diese Zahl auf über 690 Klassen. Im Use-Case-Modell stehen den 84 echten Use-Cases 17 erweiterte und deklarierte Use-Cases gegenüber.

Gegendarstellung: Beim Hyperspace-Ansatz kann auf die deklarative Vollständigkeit nicht verzichtet werden, weil sie die Freiheit bewahrt, Hyperebenen später beliebig (d.h. gemäß den Konfigurationsmöglichkeiten im Merkmalmodell einer Produktlinie) zu Systemen integrieren zu können. Dennoch stellt die Erfüllung der deklarativen Vollständigkeit in der Praxis ein Problem dar, allerdings weniger wegen der steigenden Anzahl an Elementen. Diese ist aufgrund der semantischen Übereinstimmung der Elemente vernachlässigbar (vgl. Gegendarstellung zu Argument 1). Statt dessen fiel in der Fallstudie der Zeitaufwand im Produktlinien-Engineering negativ auf, den die Herstellung der deklarativen Vollständigkeit verursachte. Hier ist unbedingt ein Mehr an Werkzeugunterstützung notwendig, wie sie [Hal01, Unterkapitel 7.3] skizziert.

Argument 3: Integrationsbeziehungen

Die Integrationsbeziehungen, die für das Zusammenführen der Hyperebenen zu Systemen in HyperFeatuRSEB erforderlich sind, stellen ebenfalls Elemente einer Produktlinie dar, die Entwickler verstehen und warten müssen. Integrationsbeziehungen erhöhen folglich die Komplexität, auch deshalb, weil sie den Jo-Jo-Effekt verschärfen: Entwickler müssen bei ihrer Arbeit nicht nur die allgemeine Integrationsstrategie, sondern auch die speziellen Integrationsbeziehungen beachten.

Gegendarstellung: Die Integrationsstrategie deckt bereits den größten Teil der Integration von Hyperebenen ab. Integrationsbeziehungen müssen deshalb nur für Ausnahmefälle spezifiziert und gewartet werden. Wieviele solcher Fälle in der

Praxis auftreten, zeigt Tabelle 7.1 auf Seite 149 für die Produktlinie „Die Siedler von Catan“. Die dort berechnete Summe von Ausnahmen beträgt im Objektmodell 42. Das ist verhältnismäßig wenig, verglichen mit den über 690 Klassen, aus denen die Produktlinie besteht. Die Auswirkungen von Integrationsbeziehungen auf die Komplexität von Produktlinien sind deshalb nicht so gravierend wie zunächst vermutet. Die Verschärfung des Jo-Jo-Effekts mildert die in der Gegendarstellung zu Argument 1 angesprochene Werkzeugunterstützung.

Bewertung

HyperFeatuRSEB reduziert die Komplexität von Produktlinien einerseits und macht sie dadurch besser handhabbar und beherrschbar. Andererseits führt die Reduzierung wieder zu einem Anstieg der Komplexität. Das Verhältnis von Reduzierung zu Anstieg fällt jedoch zugunsten der Reduzierung aus, so daß mit HyperFeatuRSEB entwickelte Produktlinien bezüglich Komplexität besser abschneiden als jene, die mit FeatuRSEB entwickelt wurden. Hinzu kommt noch, daß die Lösung des Überschneidungsproblems in HyperFeatuRSEB auch bei größeren Produktlinien nicht scheitert. Das veranschaulicht Abbildung 8.1. Sie zeigt, wie das Objektmodell aus Abbildung 4.10 auf Seite 62, deren feingranulare Zerlegung mit FeatuRSEB nicht mehr praktikabel war, in HyperFeatuRSEB leicht in die einzelnen Merkmale aufgetrennt werden kann.

8.3 Wartbarkeit entwickelter Produktlinien

Die Zielsetzung stellt folgende Behauptung auf: „Die Wartbarkeit der mit HyperFeatuRSEB entwickelten Produktlinien, d.h. ihr Ausbau, wird nicht durch die hohe Komplexität der Produktlinien verhindert, sondern ist in vielen Fällen sogar vergleichsweise einfach möglich.“ (vgl. Unterkapitel 1.3)

Wartbarkeit bedeutet, wie aufwendig ist es, die mit einer Methode entwickelten Produktlinien auszubauen, sprich um neue Merkmale zu erweitern. Nicht bewertet wird deshalb, inwieweit eine Methode andere Aufgaben in der Wartung von Produktlinien unterstützt, z.B. die Einarbeitung weitreichender Änderungen, die sich über die gesamte Produktlinie erstrecken. Ein typischer Auslöser solcher Änderungen sind neue volks- und betriebswirtschaftliche Rahmenbedingungen (vgl. Abschnitt 4.5.3).

Die Erweiterung einer mit HyperFeatuRSEB entwickelten Produktlinie um neue Merkmale läuft in folgenden Schritten ab:

1. Neues Merkmal zum Merkmalmodell hinzufügen und die Beziehungen des neuen Merkmals zu den bestehenden Merkmalen definieren
2. Merkmal-Belang zur Merkmal-Dimension des Hyperraums der Produktlinie hinzufügen. Eine *trace*-Abhängigkeitsbeziehung muß das neue Merkmal mit dem neuen Belang verknüpfen.
3. Hyperebene für den neuen Merkmal-Belang zur Produktlinie hinzufügen

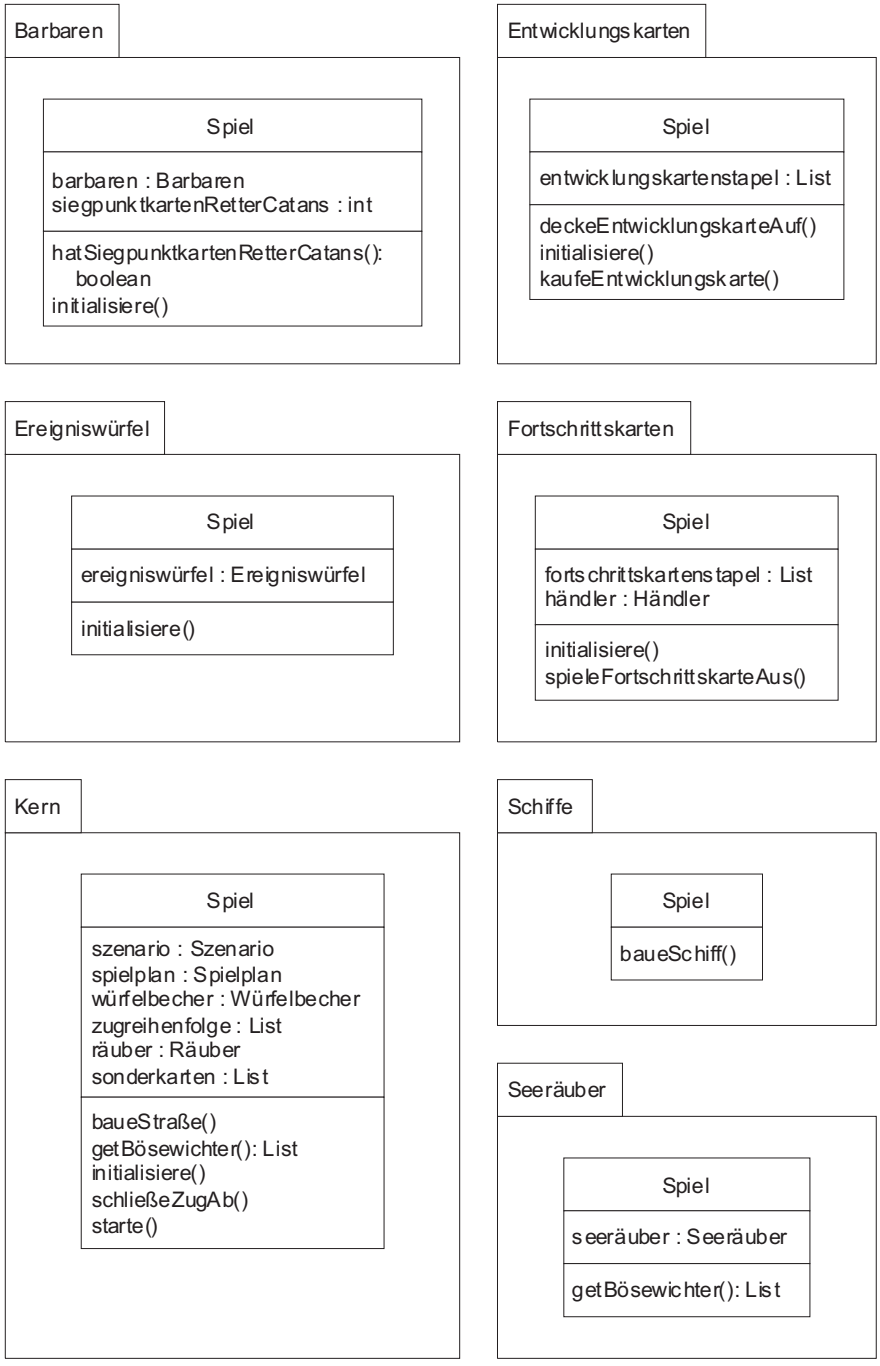

Abbildung 8.1: Komplexität entwickelter Produktlinien

4. Use-Cases für die neuen Anforderungen in der neuen Hyperebene beschreiben. Klassen für die neuen Use-Cases entwerfen und in der neuen Hyperebene modellieren sowie implementieren. Währenddessen gegebenenfalls spezielle, von der Integrationsstrategie abweichende Integrationsbeziehungen beim neuen Merkmal eintragen.

5. Wenn eine Integration der neuen Hyperebene mit den bestehenden nicht allein durch Integrationsstrategie und -beziehungen realisierbar ist, bestehende Hyperebenen refaktorieren

Der erste Schritt und Teile des vierten Schritts stammen aus FeatuRSEBs Vorgehensweise bei der Wartung. Alle anderen Tätigkeiten führt HyperFeatuRSEB neu ein. Welche Konsequenzen die angepaßte Vorgehensweise in der Praxis hat, demonstriert der Rest des Unterkapitels am selben Szenario wie Abschnitt 4.5.3.

Ausbau von Produktlinien: Merkmalmodell und Hyperraum

Die Produktlinie „Die Siedler von Catan“ soll um die Merkmale Goldfluß und Gründung erweitert werden. Das erste Merkmal ist der Ausbaustufe „Die Seefahrer“ entnommen, das zweite der Stufe „Städte & Ritter“. Zu Beginn einer Wartung ist das Merkmalmodell der Produktlinie zu erweitern. Hieran ändert HyperFeatuRSEB nichts, so daß sich die neuen Merkmale genau wie in FeatuRSEB in das Merkmalmodell einfügen. Abbildung 8.2 wiederholt das Merkmaldiagramm aus Abbildung 4.11 auf Seite 64.

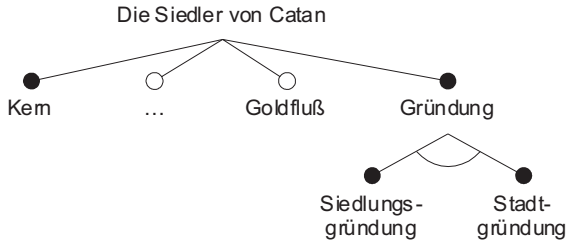

Abbildung 8.2: Wartung Merkmalmodell (Ergebnis)

Der Hyperraum der Produktlinie erhält in der Merkmal-Dimension drei neue Belange: Goldfluß, Siedlungsgründung und Stadtgründung. Das Merkmal Gründung ist kein Belang. Es gruppiert lediglich die ihm untergeordneten Merkmale, fügt aber keine Funktionalität zur Produktlinie hinzu. Merkmale und Belange werden über *trace*-Abhängigkeitsbeziehungen verknüpft.

... Hyperebenen mit Use-Case- und Objektmodell

Als nächstes werden entsprechend den neuen Merkmal-Belangen drei neue Hyperebenen in der Produktlinie angelegt. Ihr Use-Case-Modell zeigt Abbildung 8.3. Die Hyperebene Kern ist nur ausschnittsweise dargestellt, weil ihr Inhalt

fast vollständig dem Use-Case-Modell in der Ausgangssituation des Wartungsszenarios entspricht (vgl. Abbildung 4.12 auf Seite 65). Die einzige Ausnahme bildet der Aktivitätszustand Gebäude gründen, der vorher Siedlung gründen hieß. Im Zuge einer Refaktorierung wurde er umbenannt und dient jetzt als Platzhalter, der bei der Fertigung eines Systems entweder durch den Aktivitätszustand Siedlungsgründung::Siedlung gründen oder Stadtgründung::Stadt gründen ersetzt wird (vgl. Integrationsbeziehung Ersetzen in der jeweiligen Hyperebene).

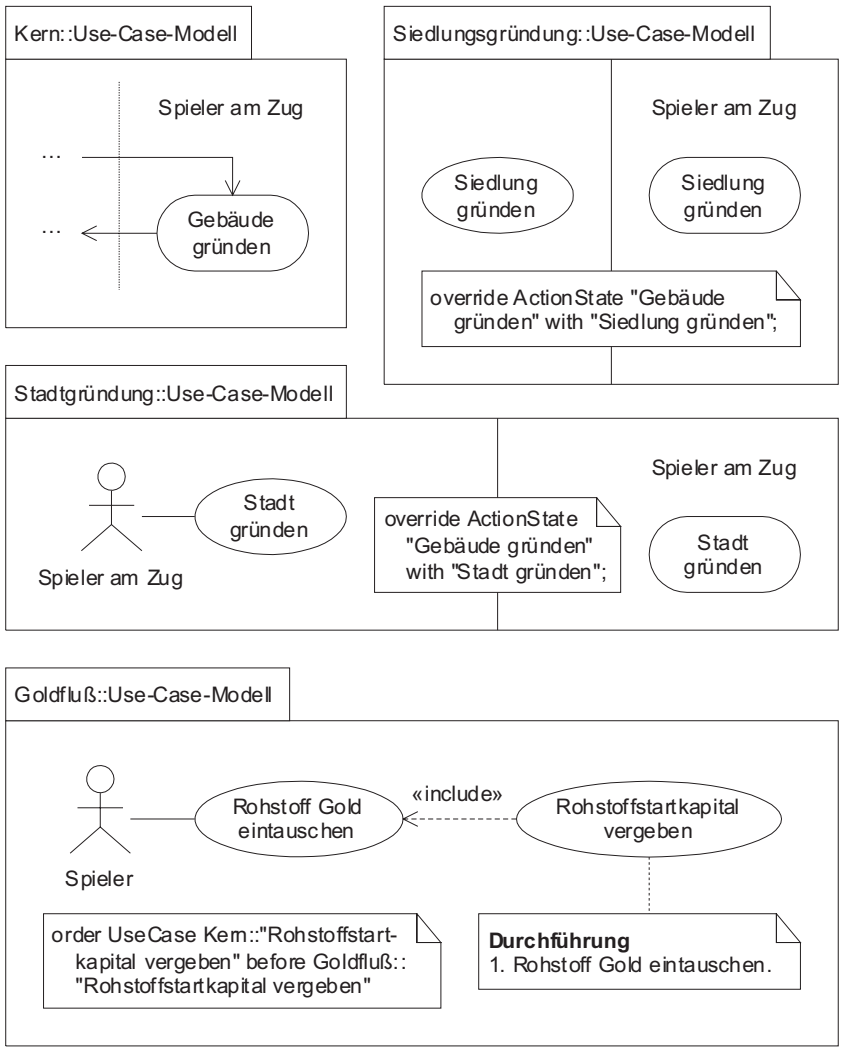

Abbildung 8.3: Wartung Hyperebenen mit Use-Case-Modell (Ergebnis)

Die Hyperebene Siedlungsgründung benötigt aus dem Kern den Use-Case Siedlung gründen, damit sie ihn dem Aktivitätszustand Siedlung gründen als Aktivität zuweisen kann. Eine direkte Referenz würde das Prinzip der deklarativen Vollständigkeit verletzen, weshalb die Hyperebene den Use-Case erneut definiert, allerdings ohne Erweiterungen an ihm vorzunehmen. Keine Erweiterung,

sondern vollständig neu ist der Use-Case Stadt gründen in der Hyperebene Stadtgründung. Der Use-Case ist dem gleichnamigen Aktivitätszustand als Aktivität zugeordnet. Schließlich erweitert die Hyperebene Goldfluß den im Kern definierten Use-Case Rohstoffstartkapital vergeben. Im Falle einer Integration hängt der Goldfluß wegen der Integrationsbeziehung Ordnen den Aufruf des neuen Use-Case Rohstoff Gold eintauschen ans Ende der Startkapitalvergabe.

Die Wartung des Use-Case-Modells erfordert mit HyperFeatuRSEB weniger tiefe Eingriffe in das bestehende Use-Case-Modell der Produktlinie als mit FeatuRSEB. Dort mußte der Aktivitätsgraph um einen Entscheidungsknoten und der Use-Case Rohstoffstartkapital vergeben um eine Erweiterungsstelle ergänzt werden (vgl. Abbildung 4.13 auf Seite 66). Noch deutlicher tritt die Reduzierung solcher Eingriffe bei einem Vergleich der Wartungsergebnisse im Objektmodell zutage.

Abbildung 8.4 zeigt die Objektmodelle der drei neuen Hyperebenen und das der bestehenden Ebene Kern. Von letzterer ist wieder nur ein Ausschnitt zu sehen, der auf die einzige notwendige Refaktorierung gegenüber der Ausgangssituation in Abbildung 4.14 auf Seite 68 hinweist: Die Operation Gründungsphase::führeRückrundenzugDurch() ruft für die Gründung eines Gebäudes die nicht implementierte Operation gründeGebäude() auf, bevor sie das Rohstoffstartkapital vergibt und eine Wegbebauung (Straße oder Schiff) gründen läßt. Die Implementierung für gründeGebäude() liefert die Hyperebene Siedlungsgründung oder Stadtgründung, je nach gewähltem Merkmal. Das, was in FeatuRSEB die Entwurfsmuster Strategie und Abstrakte Fabrik auf komplizierte Art und Weise bewerkstelligten (vgl. Abbildung 4.15 auf Seite 69), erledigt in HyperFeatuRSEB ein einfaches Verschmelzen.

Ähnliches gilt für den Rohstofftausch nach Vergabe des Startkapitals an Rohstoffen. FeatuRSEB benötigte hierfür eine Generalisierungsbeziehung zwischen den Klassen RohstoffstartkapitalVergeben und RohstoffGoldEintauschen sowie eine Abstrakte Fabrik. HyperFeatuRSEB erzielt den gleichen semantischen Effekt durch Verschmelzen von RohstoffstartkapitalVergeben::führeAus() aus Kern und Goldfluß. Die richtige Reihenfolge stellt eine Integrationsbeziehung Ordnen sicher.

Bewertung

Die HyperFeatuRSEB-Methode gewährleistet die Wartbarkeit entwickelter Produktlinien. Der Ausbau von Produktlinien erfolgt fast ausschließlich über das Hinzufügen neuer Hyperebenen. Refaktorierungen an bestehenden Hyperebenen fallen selten an und sind deutlich weniger invasiv als mit FeatuRSEB. Diese Beobachtung trifft nicht nur auf das vorgestellte Beispiel zu, sondern kann durch insgesamt über 30 Ausbauten belegt werden, die während der Entwicklung der Produktlinie „Die Siedler von Catan“ durchgeführt worden sind [Hal01, Ulr02]. Ein zusätzlicher Vorteil von HyperFeatuRSEB ist, daß bei der Wartung nur eine *trace*-Abhängigkeitsbeziehung zwischen den Modellen zu aktualisieren ist: vom Merkmal zu seinem Belang.

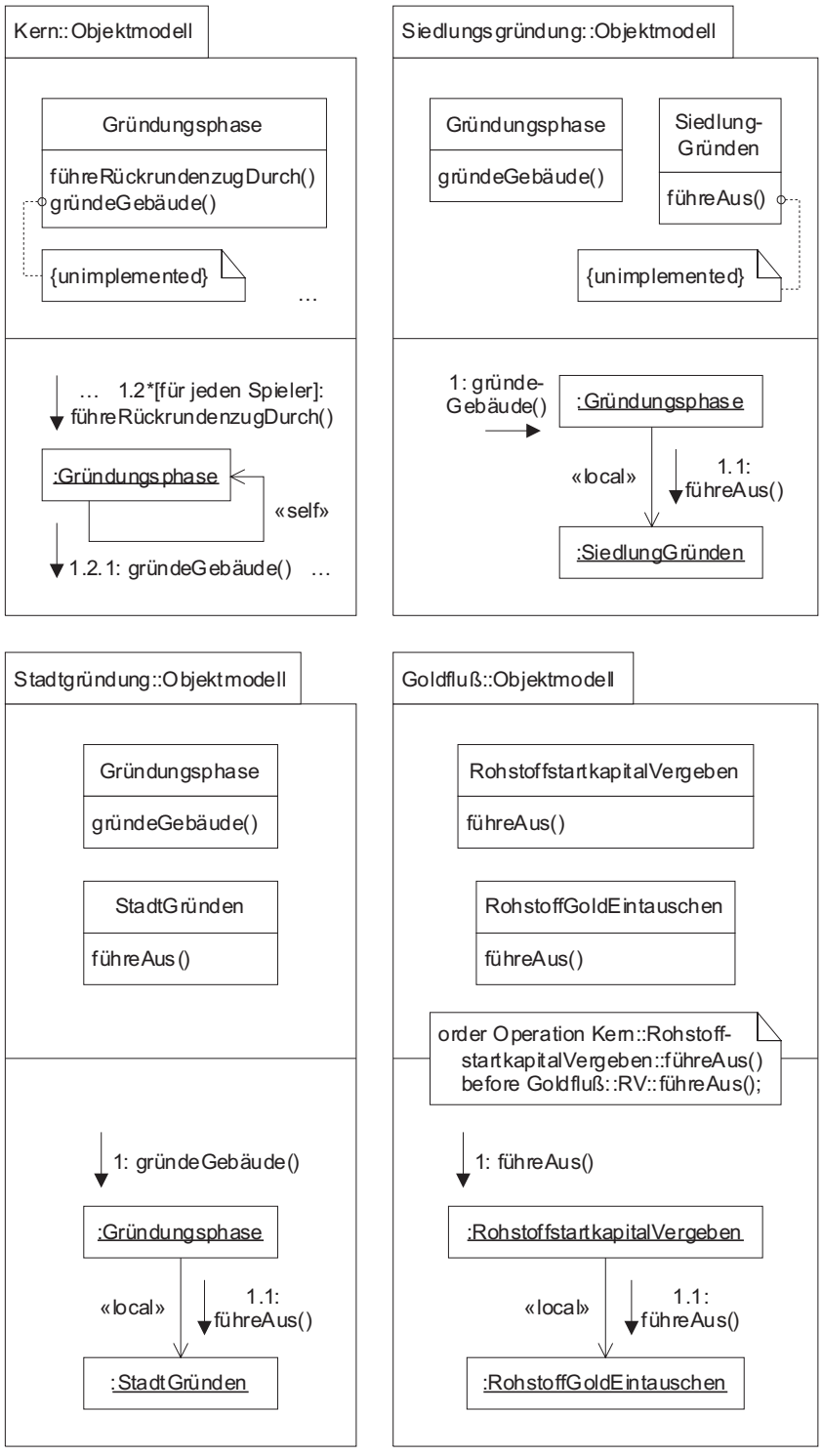

Abbildung 8.4: Wartung Hyperebenen mit Objektmodell (Ergebnis)

8.4 Skalierbarkeit der Methode

Die Zielsetzung stellt folgende Behauptung auf: „Mit HyperFeatuRSEB können Teams mit bis zu 3 Personen Produktlinien objektorientiert mindestens bis zu einem Umfang von 80 Use-Cases und 300 Klassen entwickeln. Damit skaliert die Methode besser als FeatuRSEB." (vgl. Unterkapitel 1.3)

Die Skalierbarkeit von Methoden läßt sich durch empirische Fallstudien abschätzen. Im Rahmen dieser Arbeit wurde eine derartige Studie mit HyperFeatuRSEB durchgeführt: „Die Siedler von Catan". Die während der Studie erhobenen Metriken faßt Tabelle 8.2 in der rechten Spalte zusammen. Grundlage für die Berechnung der Anzahl möglicher Produkt-Konfigurationen war das Merkmalmodell der Produktlinie, dessen vollständige Version in Abbildung 8.5 zu sehen ist. Die Berechnung geschah über einen Brute-Force-Algorithmus, da die Formeln aus [ESBC01] die für die Zahl an Konfigurationen wichtigen *require*- und *exclude*-Beziehungen in Merkmalmodellen nicht berücksichtigen. Die Klassen und Anweisungen zählte das Werkzeug JavaNCSS [JSS].

METRIK	FEATURSEB	HYPERFEATURSEB
Projektteam	1 Entwickler	3 Entwickler
Entwicklungszeit [Personen-Monate]	2	8
Merkmale	9	42
Produkt-Konfigurationen	8	38 Millionen
Use-Cases	24	84
UML-Diagramme	20	70
Klassen	122	300
Anweisungen (ohne Kommentare)	9.100	18.600

Tabelle 8.2: Skalierbarkeit von FeatuRSEB und HyperFeatuRSEB

Die Metriken für HyperFeatuRSEB beweisen die Eignung der Methode für die Entwicklung von Produktlinien mit dem in der Zielsetzung genannten Umfang. Den Metriken gegenübergestellt sind in der Tabelle die entsprechenden Werte für FeatuRSEB. Letztere ergeben sich aus den besten, in Abschnitt 4.5.4 veröffentlichten Zahlen. Der Vergleich macht deutlich, daß HyperFeatuRSEB besser skaliert als FeatuRSEB – wobei in diese Aussage auch die in den vorangegangenen Unterkapiteln vorgenommenen Bewertungen zu Komplexität und Wartbarkeit einfließen. Ansonsten wäre die Aussage vor dem Hintergrund der geringen Anzahl Fallstudien nicht zu rechtfertigen.

8.5 Zusammenfassung

Die Überprüfung der Zielsetzung zeigt, daß diese Arbeit die vereinbarten Ziele erreicht. Die HyperFeatuRSEB-Methode hebt zusammen mit Hyper/UML den Stand der Technik in der objektorientierten Entwicklung von Produktlinien auf ein Niveau, das die bisherigen Methoden, allen voran FeatuRSEB, nicht

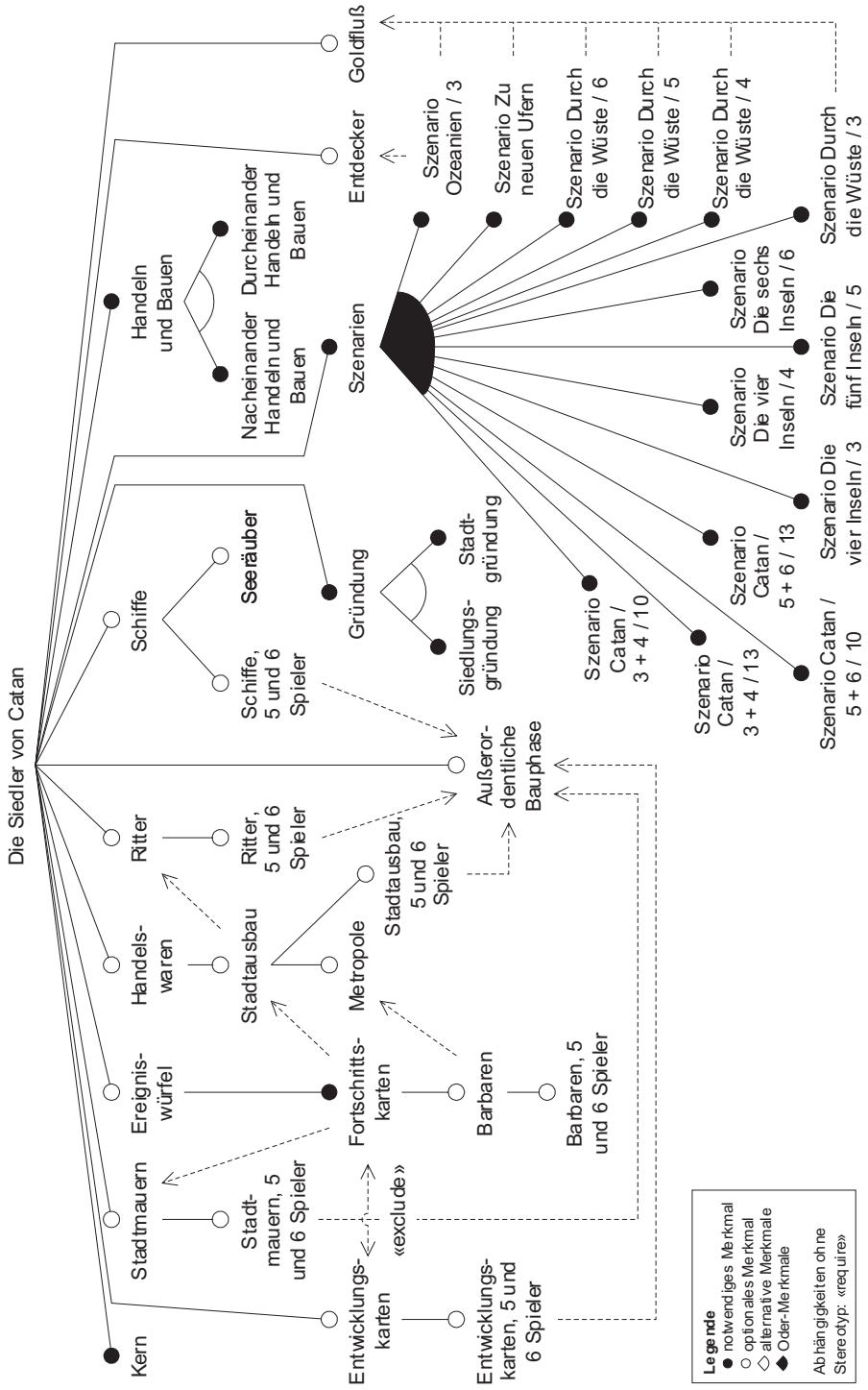

Abbildung 8.5: Merkmalmodell der Produktlinie „Die Siedler von Catan“

erreichen. Darüber hinaus kann Hyper/UML auch in der Entwicklung einzelner Systeme gewinnbringend eingesetzt werden. Denn dort ist das Trennen von Belangen beim Modellieren und die damit einhergehende Reduzierung der Komplexität ebenso von Bedeutung wie bei der Produktlinien-Entwicklung. Das verbleibende Kapitel beschließt diese Arbeit mit einem Ausblick auf zukünftig zu bearbeitende Themen.

Kapitel 9

Schlußfolgerungen

Die Schlußfolgerungen beginnen mit einem Blick zurück auf das, was die vorangegangenen Kapitel vorstellten: den Beitrag dieser Arbeit. Sie enden, indem sie einen Blick nach vorn werfen und zukünftig zu bearbeitende Themen anreißen.

9.1 Beitrag

Die Entwicklung einer Software-Produktlinie führt zu Komponenten, die in Software-Systemen wiederverwendet werden können. Die Systeme entstehen aufgrund der Wiederverwendung in kurzer Zeit. Ihre Entwicklung verursacht verhältnismäßig geringe Kosten, und das Ergebnis ist von hoher Qualität. Vorteile, die eine an einzelnen Systemen ausgerichtete Softwareentwicklung nicht in vergleichbarem Maßstab erzielt (vgl. Kapitel 1).

Noch einen Schritt weiter geht die generative Programmierung. Sie will auf der Basis von Produktlinien Systeme automatisiert generieren und überträgt dazu Konzepte der aus der Automobilindustrie bekannten Serienfertigung in die Welt des Software-Engineerings. Die Automatisierung verkürzt erneut Entwicklungszeit und -kosten für Systeme aus der Produktlinie. Darüber hinaus bricht die generative Programmierung mit dem Trend zu immer größeren, ressourcenhungrigeren Systemen, die von den Kunden nicht genutzte und deshalb auch nicht gewollte Funktionalität enthalten. Schlanke Systeme sind das Ziel.

Die bisherigen Methoden zur Entwicklung von Produktlinien, z.B. FeatuRSEB, berücksichtigen die generative Programmierung nicht. Statt dessen werden aus den Produktlinien manuell Systeme entwickelt. Zur Behebung dieses Umstands leistet die vorliegende Arbeit einen Beitrag. Sie stellt HyperFeatuRSEB vor, eine Methode zur objektorientierten Entwicklung von Produktlinien, aus denen schlanke Systeme mittels eines gleichfalls im Rahmen der Arbeit entworfenen Generators automatisiert in Serie gefertigt werden können. Die Methode basiert auf FeastuRSEB, räumt aber deren Nachteile aus. Sie ergänzt den in FeatuRSEB nicht vorhandenen Serienfertigungsgedanken und reduziert die hohe Komplexität sowie schlechte Wartbarkeit der mit FeatuRSEB entwickelten Produktlinien.

Das verbessert letzten Endes die Skalierbarkeit von HyperFeatuRSEB, so daß sich mit ihr größere Produktlinien als mit FeatuRSEB entwickeln lassen.

Die HyperFeatuRSEB-Methode beseitigt die Nachteile FeatuRSEBs durch die Nutzung des Hyperspace-Ansatzes, einer Technik aus der generativen Programmierung. Allerdings ist der Hyperspace-Ansatz nicht direkt in der Softwareentwicklung einsetzbar, da seine abstrakten Konzepte und Begriffe zuvor auf Modellierungs- und Programmiersprachen abzubilden sind. Bisher existiert eine solche Abbildung für Java, genannt Hyper/J. Um den Ansatz nicht nur in der Implementierungsphase von Produktlinien nutzen zu können, stellt diese Arbeit als zweiten Teil ihres Beitrags Hyper/UML vor: eine Abbildung des Hyperspace-Ansatzes auf die UML. Beide Abbildungen zusammen ermöglichen jetzt die durchgängige Nutzung des Hyperspace-Ansatzes in allen Phasen der Softwareentwicklung – und das unabhängig davon, ob nun Produktlinien oder einfach nur einzelne Systeme entwickelt werden sollen.

9.2 Ausblick

Die vorliegende Arbeit löst nicht alle Schwierigkeiten bei der Entwicklung von Produktlinien und der Serienfertigung von Systemen. So bleibt Raum für zukünftige Themenstellungen, die im folgenden zusammengestellt werden. Einige der Themen wurden bereits angesprochen. Detaillierte Erläuterungen sind den angegebenen Verweisen zu entnehmen.

HyperFeatuRSEB

HyperFeatuRSEBs Eignung wurde anhand einer umfangreichen Fallstudie aus der Domäne der Gesellschaftsspiele überprüft. Welche Verbesserungen sich ergeben, wenn man die Methode in anderen Domänen oder für Produktlinien von noch größerem Umfang anwendet, ist eine interessante Fragestellung für weiterführende Arbeiten. Die Antworten helfen, die Aussagen hinsichtlich Organisation der Entwicklungsprozesse und Skalierbarkeit HyperFeatuRSEBs zu präzisieren.

Ein zweites wichtiges Thema ist das Testen von Produktlinien. Je breiter die Variabilität einer Produktlinie ist, desto mehr Möglichkeiten gibt es, die Merkmale miteinander zu kombinieren. Aus den lediglich 42 Merkmalen der Produktlinie „Die Siedler von Catan“ ergeben sich über 38 Millionen verschiedene Konfigurationen. Alle daraus gefertigten Systeme manuell zu testen, würde einen unvorstellbar hohen Aufwand erfordern. Der reduziert sich beträchtlich, wenn das Testen automatisiert wird. Ist es möglich, bei der Fertigung eines Systems entsprechende Testfälle mit zu generieren, die das System am Ende der Fertigung automatisch überprüfen?

Wünschenswert ist eine bessere Werkzeugunterstützung für HyperFeatuRSEB. Allen voran ist die Entwicklung eines Hyper/UML-Werkzeugs zu nennen, gefolgt von einer Anbindung des Produkt-Konfigurators an die mit AmiEddi erstellten Merkmalmodelle und die Integration des Hyper/J-Werkzeugs in Entwicklungsumgebungen (vgl. Unterkapitel 4.4 und 7.4 sowie Abschnitt 5.3.9).

Die Implementierung einer Produktlinie muß derzeit in Java beziehungsweise Hyper/J geschehen. Zukünftige Arbeiten könnten den Hyperspace-Anssatz auf weitere objektorientierte Programmiersprachen abbilden und die Abbildungen in HyperFeatuRSEB integrieren.

Hyper/UML

Hyper/UML läßt, neben der Entwicklung eines Werkzeugs (vgl. Unterkapitel 6.10), ebenfalls einige Punkte offen: Wie ist die Abbildung um die ausgelassenen UML-Elemente Subsysteme, Komponenten und Knoten zu erweitern (vgl. Abschnitt 6.2.1)? Gibt es weitere nützliche Integrationsbeziehungen (vgl. Abschnitt 6.5.1)? Kann auf die Aufweichung des Sichtbarkeitskonzeptes der UML beim Integrieren von Belangen im Rahmen von Hypermodulen verzichtet werden (vgl. Abschnitt 6.5.3)? Eignet sich die Integrationsbeziehung Ersetzen für UML-Elemente außer Operationen und Aktivitätszustände (vgl. Abschnitt 6.9.3)?

9.3 Zusammenfassung

Diese Arbeit leistet einen wichtigen Schritt hin zur Verbesserung der Produktlinien-Entwicklung. Aber es ist nur ein erster Schritt. So untersuchen parallel laufende Arbeiten das Gebiet Requirements-Engineering für Produktlinien und gehen der Frage nach, wie gemeinsame und variable Anforderungen erhoben werden können. Ein anderer Aspekt im gleichen Zusammenhang ist das Extrahieren von Gemeinsamkeit und Variabilität aus Alt-Systemen mit Techniken des Reverse-Engineerings. Alles in allem haben Produktlinien im Software-Engineering ein weites Feld mit vielen Unbekannten eröffnet, das zu durchdringen ein lohnenswertes Ziel ist.

Anhang A

Das Profil Hyper/UML

Kapitel 6 bildete den Hyperspace-Ansatz auf die UML ab. Die dabei schrittweise entstandene Erweiterung der UML faßt dieser Anhang zusammen und gibt so einen Gesamtüberblick zu Syntax, Regeln und Semantik des entstandenen UML-Profils Hyper/UML.

A.1 Syntax

Abbildung A.1 zeigt die grafische Syntax des Profils Hyper/UML.

A.2 Regeln

Die folgenden Regeln und Operationen vervollständigen die Syntax-Definition des Profils Hyper/UML. Sie ergänzen auch einige im UML-Metamodell definierte Klassen.

AbstractConcern

Operation 1: includesUnit() liefert true zurück, wenn das übergebene Element dem Belang explizit oder implizit zugeordnet ist. Die Operation wird bei der Integration von Belangen zu Hypermodulen genutzt. Mit ihr kann festgestellt werden, ob ein Element an einer Integration teilnimmt.

```
includesUnit(u : HyperspaceUnit): Boolean;
includesUnit =
   u.concernInDimension(self.dimension) = self
```

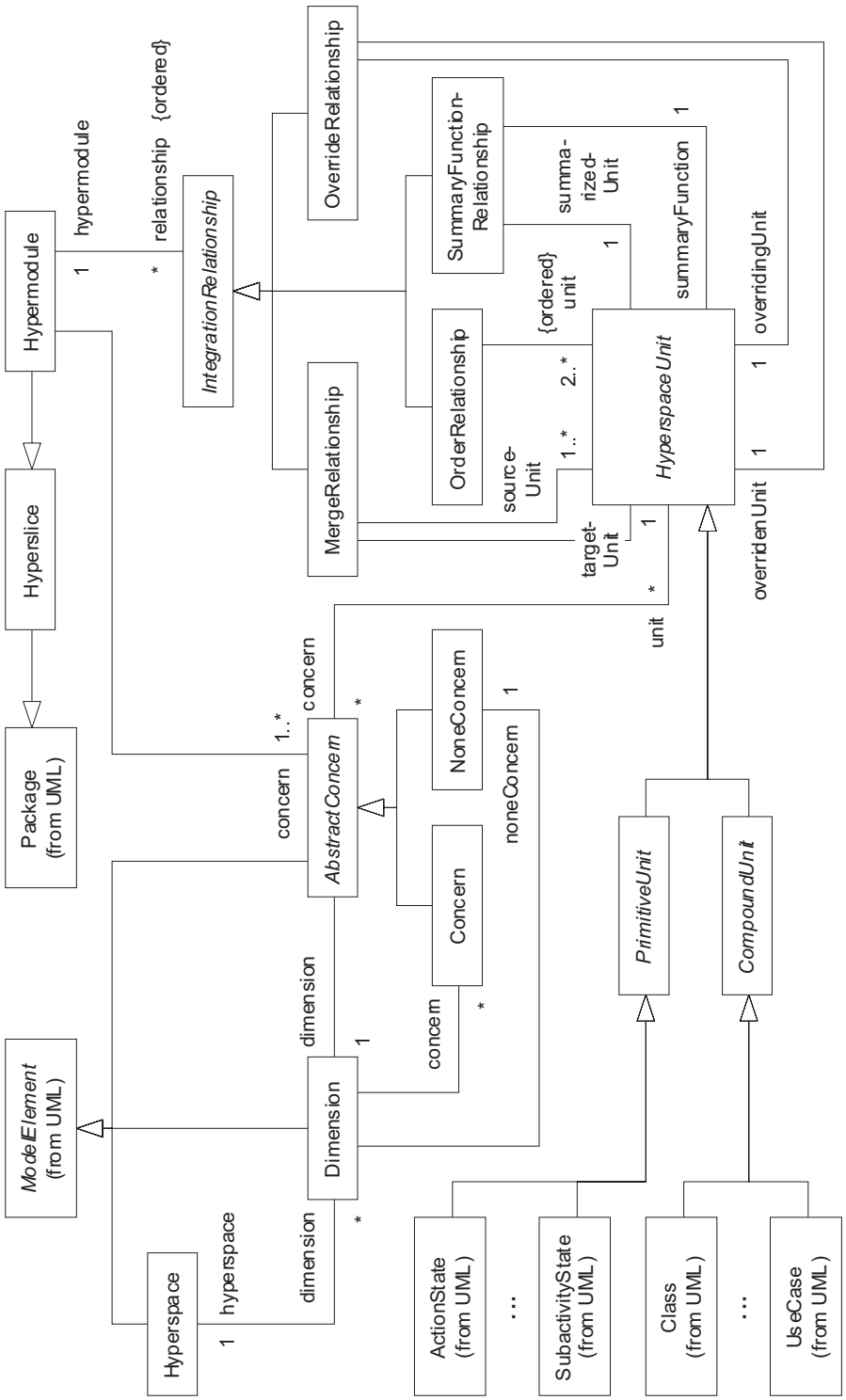

Abbildung A.1: Profil Hyper/UML – Gesamtansicht Syntax

CompoundUnit

Operation 2: units() liefert alle in dem zusammengesetzten Element enthaltenen Elemente zurück.

```
units(): Set(HyperspaceUnit);
```

Dimension

Regel 1: Die Belange einer Dimension sind verschieden benannt.

```
self.allConcerns()->isUnique( c | c.name)
```

Operation 3: allConcerns() liefert die Belange einer Dimension zurück.

```
allConcerns(): Set(AbstractConcern);
allConcerns =
   self.concern->including(noneConcern)
```

Hyperslice

Regel 2: Hyperebenen können nicht ineinander geschachtelt werden, weil sie Module darstellen, die voneinander unabhängig sein sollen, was durch die deklarative Vollständigkeit erreicht wird. Eine Schachtelung von Hyperebenen würde die Ebenen in eine hierarchische Beziehung zueinander setzen und führt zu einer Abhängigkeit der untergeordneten von den übergeordneten Ebenen.

```
self.allSurroundingNamespaces()->forAll( ns |
   not ns.isOclKindOf(Hyperslice))
```

Hyperspace

Regel 3: Pro UML-Modell, d.h. pro System existiert nur ein Hyperraum, weil der Hyperspace-Ansatz nicht mehrere Hyperräume für ein System vorsieht. Die Klasse Hyperspace ist folglich ein Singleton [GHJV95, Seite 127ff].

```
Hyperspace.allInstances->size() = 1
```

Regel 4: Die Dimensionen eines Hyperraums sind verschieden benannt.

```
self.dimension->isUnique( dim | dim.name)
```

HyperspaceUnit

Regel 5: Jedes Element ist in jeder Dimension nur einem Belang explizit zugeordnet – eine Forderung des Hyperspace-Ansatzes.

```
self.concern->isUnique( c | c.dimension)
```

Operation 4: allParents() liefert die Elementhierarchie zurück, in der sich das Element befindet. Die Liste beginnt mit dem zusammengesetzten Element, in der das Element enthalten ist und endet mit dem obersten zusammengesetzten Element. Die Liste ist leer, wenn das Element in keinem zusammengesetzten Element enthalten ist.

```
allParents(): Sequence(CompoundUnit);
allParents =
    if (self.parent()->isEmpty())
        then Sequence{}
        else self.parent()->asSequence()->union(self.parent().allParents())
    endif
```

Operation 5: concernInDimension() liefert den Belang zurück, dem das Element in der übergebenen Dimension zugeordnet ist. Diese Zuordnung muß nicht explizit geschehen, sondern kann auch implizit über die Elementhierarchie gegeben sein. Ist auch dies nicht der Fall, ist das Element automatisch dem 0-Belang zugeordnet.

```
concernInDimension(dim : Dimension): AbstractConcern;
concernInDimension =
    if (self.concern->exists( c | c.dimension = dim))
        then self.concern->any( c | c.dimension = dim)
        else if (self.parent()->notEmpty())
            then self.parent().concernInDimension(dim)
            else dim.noneConcern
        endif
    endif
```

Operation 6: parent() liefert für ein Element, das in einem zusammengesetzten Element enthalten ist, dieses zusammengesetzte Element zurück.

```
parent(): CompoundUnit;
```

MergeRelationship

Regel 6: Das Ziel-Element ist nicht zugleich Quell-Element. Das verhindert, daß ein Element mit sich selbst verschmolzen wird.

```
self.targetUnit->intersection(self.sourceUnit)->isEmpty()
```

Regel 7: Die zu verschmelzenden Elemente sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.units()->forAll( u |
   self.hypermodule.concern->exists( c |
      c.includesUnit(u)))
```

Regel 8: Die zu verschmelzenden Elemente sind vom aufgezählten Typ. Hier sind alle Elementtypen von Hyper/UML erlaubt, weil die Semantik des Verschmelzens für jeden Typ festgelegt ist.

```
self.units()->forAll( u |
   if (u.oclIsTypeOf(AssociationEnd))
      then u.participant.oclIsTypeOf(Class)
      else Set{
         Package, Model, Hyperslice,
         Interface, Class, Attribute, Operation,
         Actor, UseCase,
         StateMachine, SimpleState, CompositeState,
         Signal,
         ActivityGraph, ActionState, SubactivityState
      }->includes(u.oclType()))
   endif
```

Regel 9: Die zu verschmelzenden Elemente sind vom selben Typ. Elemente verschiedenen Typs, z.B. Use-Cases und Zustände, können semantisch nicht verschmolzen werden.

```
self.units()->forAll( u1, u2 |
   (u1.oclType() = u2.oclType()) and
   (u1.stereotype = u2.stereotype))
```

Operation 7: units() liefert die zu verschmelzenden Elemente zurück. Die obigen Regeln verwenden diese Operation.

```
units(): Set(HyperspaceUnit);
units =
   self.targetUnit->union(self.sourceUnit)
```

ModelElement (Ergänzung)

Operation 8: correspondsTo() liefert true zurück, wenn das Modellelement mit dem übergebenen Element übereinstimmt. Unterklassen, bei denen die Übereinstimmung weiteren Bedingungen unterliegt, überschreiben diese Operation.

```
correspondsTo(me : ModelElement): Boolean;
correspondsTo =
    (self <> me) and
    (self.oclType() = me.oclType()) and
    (self.stereotype = me.stereotype) and
    (self.name <> '') and (self.name = me.name)
```

NoneConcern

Regel 10: Der 0-Belang heißt in Hyper/UML analog zu Hyper/J „None“.

```
self.name = 'None'
```

Operation (Ergänzung)

Operation 9: correspondsTo() liefert true zurück, wenn die Operation mit der übergebenen Operation übereinstimmt. Zusätzlich zu den Prüfungen in ModelElement wird bei Operationen die Signatur verglichen.

```
correspondsTo(op : Operation): Boolean;
correspondsTo =
    self.oclAsType(ModelElement).correspondsTo(op) and
    self.hasSameSignature(op)
```

OrderRelationship

Regel 11: Die zu ordnenden Elemente sind nicht mehrfach in der Integrationsbeziehung angegeben, da dann die Reihenfolge der Elemente nicht eindeutig bestimmbar wäre.

```
self.unit = self.unit->asSet()
```

Regel 12: Die zu ordnenden Elemente sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.unit->forAll( u |
    self.hypermodule.concern->exists( c |
        c.includesUnit(u)))
```

Regel 13: Die zu ordnenden Elemente sind vom aufgezählten Typ. Hier sind nur Operationen, Use-Cases, Aktivitätsgraphen und -zustände erlaubt. Das Ordnen anderer Elemente hätte keine semantische Bedeutung.

```
self.unit->forAll( u |
   Set{
      Operation,
      UseCase,
      ActivityGraph, ActionState, SubactivityState
   }->includes(u.oclType()))
```

Regel 14: Die zu ordnenden Elemente sind vom selben Typ. Elemente verschiedenen Typs können semantisch nicht angeordnet werden.

```
self.unit->forAll( u1, u2 |
   (u1.oclType() = u2.oclType()) and
   (u1.stereotype = u2.stereotype))
```

OverrideRelationship

Regel 15: Das zu ersetzende Element und das Ersatz-Element sind nicht identisch, weil sonst das Ersetzen wirkungslos bliebe.

```
self.overridenUnit <> self.overridingUnit
```

Regel 16: Das zu ersetzende Element und das Ersatz-Element sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.units()->forAll( u |
   self.hypermodule.concern->exists( c |
      c.includesUnit(u)))
```

Regel 17: Das zu ersetzende Element und das Ersatz-Element sind vom aufgezählten Typ (vgl. Abschnitt 6.9.3).

```
self.units()->forAll( u |
   Set{
      Operation,
      ActionState, SubactivityState
   }->includes(u.oclType()))
```

Regel 18: Das zu ersetzende Element und das Ersatz-Element sind vom selben Typ. Elemente verschiedenen Typs, z.B. Operationen und Aktivitätszustände, können semantisch nicht ersetzt werden.

```
self.units()->forAll( u1, u2 |
   (u1.oclType() = u2.oclType()) and
   (u1.stereotype = u2.stereotype))
```

Operation 10: units() liefert das zu ersetzende Element und das Ersatz-Element zurück. Die obigen Regeln verwenden diese Operation.

```
units(): Set(HyperspaceUnit);
units =
    self.overridenUnit->union(self.overridingUnit)
```

SummaryFunctionRelationship

Regel 19: Das zu summierende Element und die Summenfunktion sind nicht identisch. Das verhindert eine Endlosschleife ausgelöst durch rekursives Summieren.

```
self.summarizedUnit <> self.summaryFunction
```

Regel 20: Das zu summierende Element und die Summenfunktion sind den im Hypermodul angegebenen Belangen zugeordnet. Andernfalls würden Elemente in die Integration einbezogen werden, die nicht an ihr beteiligt sind.

```
self.units()->forAll( u |
    self.hypermodule.concern->exists( c |
        c.includesUnit(u)))
```

Regel 21: Das zu summierende Element und die Summenfunktion sind Operationen. Das Summieren anderer Elemente hätte keine semantische Bedeutung.

```
self.units()->forAll( u | u.isOclTypeOf(Operation))
```

Operation 11: units() liefert das zu summierende Element und die Summenfunktion zurück. Die obigen Regeln verwenden diese Operation.

```
units(): Set(HyperspaceUnit);
units =
    self.summarizedUnit->union(self.summaryFunction)
```

A.3 Semantik

Tabelle A.1 listet die in Abbildung A.1 auf Seite 182 gezeigten Klassen des Profils Hyper/UML in alphabetischer Reihenfolge auf und führt an, in welchen Unterkapiteln die Semantik dieser Klassen erläutert wird.

KLASSE	UNTERKAPITEL
AbstractConcern	6.3
CompoundUnit	6.2
Concern	6.3
Dimension	6.3
Hypermodule	6.5
Hyperslice	6.4
HyperspaceUnit	6.2, 6.3
Hyperspace	6.3
IntegrationRelationship	6.5
MergeRelationship	6.6
NoneConcern	6.3
OrderRelationship	6.7
OverrideRelationship	6.9
PrimitiveUnit	6.2
SummaryFunctionRelationship	6.8

Tabelle A.1: Profil Hyper/UML – Semantik der Klassen

Literaturverzeichnis

[A+01] Colin Atkinson et al. *Component-Based Product Line Engineering with UML*. Addison-Wesley (2001).

[ABM00] Colin Atkinson, Joachim Bayer und Dirk Muthig. Component-Based Product Line Development: The KobrA Approach. In *Software Product Lines: Experience and Research Directions*, Seiten 289–309. Kluwer Academic Publishers (2000).

[AE] AmiEddi. Verfügbar unter [GP].

[AJ] AspectJ. `aspectj.org`.

[AU] ArgoUML. `argouml.tigris.org`.

[B+96] Frank Buschmann et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons (1996).

[B+99] Joachim Bayer et al. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the 5th Symposium on Software Reusability (SSR '99)*, Seiten 122–131. ACM Press (1999).

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley (1999).

[BG97] Don Batory und Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering* (23)(2), Seiten 67–82 (Februar 1997).

[Bli01] Frank Blinn. *Entwurf und Implementierung eines Generators für Merkmalmetamodelle*. Diplomarbeit, Fachhochschule Kaiserslautern (2001). Verfügbar unter [GP].

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer* (21)(5), Seiten 61–72 (Mai 1988).

[Bos00] Jan Bosch. *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley (2000).

[BP00] Kai Böllert und Ilka Philippow. Erfahrungen bei der objektorientierten Modellierung von Produktlinien mit FeatuRSEB. In *Proceedings of 1. Deutscher Software-Produktlinien Workshop (DSPL-1)*, Seiten 29–33. Fraunhofer Institut für Experimentelles Software Engineering (2000).

[CE00] Krzysztof Czarnecki und Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley (2000).

[CN99] Paul Clements und Linda Northrop. *A Framework for Software Product Line Practice, Version 2.7* (Juli 1999). Verfügbar unter `www.sei.cmu.edu/plp/`.

[Coc97] Alistair Cockburn. Using Goal-Based Use Cases. *Journal of Object-Oriented Programming* (10)(6), Seiten 56–62 (November 1997).

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall (1976).

[DW99] Wolfgang Dröschel und Manuela Wiemers (Herausgeber). *Das V-Modell 97: Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz.* Oldenbourg Verlag (1999).

[ESBC01] Ulrich W. Eisenecker, Mario Selbig, Frank Blinn und Krzysztof Czarnecki. Merkmalmodellierung für Softwaresystemfamilien. *OBJEKTspektrum*, Seiten 23–30 (September/Oktober 2001).

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley (1999).

[GAO95] David Garlan, Robert Allen und John Ockerbloom. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software* (12)(6), Seiten 17–26 (November 1995).

[GFA98] Martin L. Griss, John Favaro und Massimo d' Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse (ICSR '98)*, Seiten 76–85. IEEE Press (1998).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley (1995).

[GP] Generative Programming. `www.generative-programming.org`.

[Hal01] Martin Halle. *Fallstudie zur Entwicklung wiederverwendbarer Komponenten im Rahmen von Software-Produktlinien.* Diplomarbeit, Technische Universität Ilmenau (2001).

[HK99] Martin Hitz und Gerti Kappel. *UML@Work: Von der Analyse zur Realisierung.* dpunkt.verlag (1999).

[HS] Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces. `www.research.ibm.com/hyperspace/`.

[Iva99] Evgeni Ivanov. *Eine Methodik für die Entwicklung und Anwendung von objektorientierten Frameworks.* Dissertation, Technische Universität Ilmenau (1999).

[JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson und Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley (1992).

[JGJ97] Ivar Jacobson, Martin Griss und Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley (1997).

[JSS] JavaNCSS – A Source Measurement Suite for Java. `www.kclee.com/clemens/java/javancss/`.

[JTS] Jakarta Tool Suite. `www.cs.utexas.edu/users/schwartz/`.

[K+90] Kyo Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University (November 1990).

[K+97] Gregor Kiczales et al. Aspect-Oriented Programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP '97)*, Seiten 220–242. Springer-Verlag (1997).

[K+01] Gregor Kiczales et al. Getting Started with AspectJ. *Communications of the ACM* (44)(10), Seiten 59–65 (Oktober 2001).

[KB99] Philip Kotler und Friedhelm Bliemel. *Marketing-Management: Analyse, Planung, Umsetzung und Steuerung.* Schäffer-Poeschel, 9. Auflage (1999).

[Kle02] Jens Kleinschmidt. *Untersuchungen zu GenVoca und der Jakarta Tool Suite.* Studienarbeit, Technische Universität Ilmenau (2002).

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 2. Auflage (2000).

[Mac01] Sven Macke. *Generative Programmierung mit AspectJ.* Diplomarbeit, Fachhochschule Kaiserslautern (2001). Verfügbar unter [GP].

[MCD00] Paul Monday, James Carey und Mary Dangler. *San Francisco Component Framework: An Introduction.* Addison-Wesley (2000).

[ME] MetaEdit+. `www.metacase.com`.

[Nei80] James M. Neighbors. *Software Construction Using Components.* Dissertation, University of California at Irvine (1980).

[Oes99] Bernd Oesterreich. *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language.* Oldenbourg Verlag, 4. Auflage (1999).

[OMG01] Object Management Group. *OMG Unified Modeling Language Specification, Version 1.4* (September 2001).

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* Dissertation, University of Illinois at Urbana-Champaign (1992).

[OT98] Harold Ossher und Peri Tarr. Operation-Level Composition: A Case in (Join) Point. In *Proceedings of the Aspect-Oriented Programming Workshop at ICSE '98* (1998). Verfügbar unter `aosd.net`.

[OT01] Harold Ossher und Peri Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Software Architectures and Component Technology*, Kapitel 10. Kluwer Academic Publishers (2001).

[PBSR02] Ilka Philippow, Kai Böllert, Detlef Streitferdt und Matthias Riebisch. Methodical Aspects for the Development of Product Lines. In *Proceedings of the 2nd WSEAS International Conference on Information Science and Applications (ISA '02)*. WSEAS Press (2002). Erscheint demnächst.

[PR01] Ilka Philippow und Matthias Riebisch. Systematic Definition of Reusable Architectures. In *Proceedings of the 8th IEEE International Conference on Engineering of Computer Based Systems (ECBS 2001)*, Seiten 128–135. IEEE Computer Society (2001).

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings IEEE WESTCON*, Kapitel 3. Westcon (1970).

[S+96] Mark Simos et al. Organization Domain Modeling (ODM) Guidebook, Version 2.0. Bericht STARS-VC-A025/001/00, Lockheed Martin (Juni 1996).

[SBR00] Detlef Streitferdt, Kai Böllert und Matthias Riebisch. Feature-Modell und Architektur einer Systemfamilie. In *Tagungsband 45. Internationales Wissenschaftliches Kolloquium (IWK 2000)*, Seiten 621–626. Technische Universität Ilmenau (2000).

[Sel00] Mario Selbig. *A Feature Diagram-Editor – Analysis, Design, and Implementation of its Core Functionality.* Diplomarbeit, Fachhochschule Kaiserslautern (2000).

[SY99] Junichi Suzuki und Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '99* (1999). Verfügbar unter `aosd.net`.

[Teu] Klaus Teuber. *Die Siedler von Catan.* Franckh-Kosmos.

[TGP89] David Taenzer, Murthy Ganti und Sunil Podar. Problems in Object-Oriented Software Reuse. In *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP '89)*, Seiten 25–38. Cambridge University Press (1989).

[TO01] Peri Tarr und Harold Ossher. *Hyper/J User and Installation Manual* (2001). Verfügbar unter [HS].

[TOHS99] Peri Tarr, Harold Ossher, William Harrison und Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, Seiten 107–119. IEEE Press (1999).

[UAD] UML auf gut Deutsch. `www.system-bauhaus.de`.

[Ulr02] Dirk Ulrich. *Fallbeispiel zur Anwendung des Hyperspace-Ansatzes und Hyper/J.* Studienarbeit, Technische Universität Ilmenau (2002).

[VAJ] IBM VisualAge for Java. `www.software.ibm.com/vajava/`.

[VW] Volkswagen-Konfigurator. `www.vw-online.de`.

[WL99] David M. Weiss und Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley (1999).

[Wöh93] Günter Wöhe. *Einführung in die allgemeine Betriebswirtschaftslehre.* Vahlen, 18. Auflage (1993).

Thesen

1. Eine Software-Produktlinie umfaßt eine Gruppe von Software-Systemen, die der gleichen Domäne angehören, eine ähnliche Funktion erfüllen und auf Basis gemeinsam genutzter Komponenten entwickelt werden. Die Entwicklung der Systeme erfordert aufgrund der Wiederverwendung der Produktlinien-Komponenten weniger Zeit und verursacht geringere Kosten als bei der heute üblichen, an einzelnen Systemen ausgerichteten Entwicklung von Individualsoftware.

2. Die generative Programmierung beschäftigt sich mit der Generierung von Software-Systemen aus Produktlinien. Statt die Systeme manuell zu entwickeln, setzt ein Generator die Komponenten der Produktlinie automatisiert zu den von Kunden gewünschten Systemen zusammen. Diese sogenannte Serienfertigung von Software verkürzt erneut die Zeit, bis zu der produktlinienbasierte Systeme beim Kunden in Betrieb genommen werden können.

3. Die Serienfertigung ermöglicht schlanke Software-Systeme. Dazu setzt der Generator ein System nur aus den Produktlinien-Komponenten zusammen, die eine vom Kunden geforderte Funktionalität realisieren. Die darüber hinaus von der Produktlinie abgedeckte Funktionalität bleibt außen vor. Schlanke Systeme arbeiten vergleichsweise effizient und benötigen weniger Ressourcen als andere Systeme. Damit eignen sich Produktlinien auch als Basis z.B. für eingebettete Systeme.

4. Die Serienfertigung schlanker Systeme erfordert feingranulare Produktlinien-Komponenten, da der Generator ansonsten bei der Fertigung die unerwünschte Funktionalität nicht weglassen kann. Die Modellierung und Implementierung derart feingranularer Komponenten ist mit den verfügbaren objektorientierten Entwicklungsmethoden für Produktlinien, beispielsweise FeatuRSEB, zwar möglich, führt aber schon bei kleinen Produktlinien zu komplexen, unwartbaren Komponenten.

5. Die generative Programmierung stellt den Hyperspace-Ansatz bereit, eine Technik, mit der Komponenten feingranular entwickelt werden können. Der Ansatz unterstützt durch seine allgemein gehaltenen Konzepte und Begriffe prinzipiell alle Phasen des Softwareentwicklungsprozesses. Er ist vor seiner Anwendung lediglich auf die zu verwendenden Modellierungs- und Programmiersprachen abzubilden. Derzeit existiert eine Abbildung für Java mit dem Namen Hyper/J.

6. Die in der vorliegenden Arbeit aufgestellte Methode HyperFeatuRSEB unterstützt die objektorientierte Entwicklung von Produktlinien mit der UML (Unified Modeling Language) und beliebigen objektorientierten Programmiersprachen.

7. Die mit HyperFeatuRSEB entwickelten Produktlinien eignen sich zur Serienfertigung von Software-Systemen. Ein Generator setzt die Komponenten einer Produktlinie automatisiert zu Systemen zusammen. Die Systeme sind schlank. Sie enthalten nur genau die Funktionalität beziehungsweise Komponenten, die die Kunden gefordert haben.

8. HyperFeatuRSEB nutzt den Hyperspace-Ansatz zur Entwicklung feingranularer Komponenten. Um nicht nur die Implementierung solcher Komponenten mit Java beziehungsweise Hyper/J zu ermöglichen, sondern auch ihre Modellierung, nimmt diese Arbeit eine Abbildung des Hyperspace-Ansatzes auf die UML vor, genannt Hyper/UML. Insgesamt bleibt so die Komplexität der mit HyperFeatuRSEB entwickelten Produktlinien trotz feingranular modellierter und implementierter Komponenten handhabbar und damit für Entwickler beherrschbar.

9. Die Wartbarkeit der mit HyperFeatuRSEB entwickelten Produktlinien, d.h. ihr Ausbau, wird nicht durch die hohe Komplexität der Produktlinien verhindert, sondern ist in vielen Fällen sogar vergleichsweise einfach möglich. Das belegt eine im Rahmen der Arbeit durchgeführte empirische Fallstudie.

Köln, 29. Januar 2002

Erklärung

Ich versichere, daß ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- beziehungsweise Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, daß die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zur Folge hat.

Köln, 29. Januar 2002