Technische Universität Dresden
Fakultät Informatik

Diplomarbeit

Untersuchung der Modellierung von Variabilität in UML

Matthias Clauß
Matrikel-Nr.: 2478336

24. Juli 2001

Verantwortlicher Hochschullehrer:		Prof. Dr. rer. nat. habil. Heinrich Hußmann
Betreuer	:	Dipl.-Inform. Mike Fischer
Externer Betreuer	:	Dr.-Ing. habil. Bogdan Franczyk, Intershop Research, Jena
Institut	:	Software- und Multimediatechnik (SMT)
Lehrstuhl	:	Softwaretechnologie
Beginn am	:	01. Februar 2001
Einzureichen am	:	31. Juli 2001

Aufgabenstellung für die Diplomarbeit

„Untersuchung der Modellierung von Variabilität“

Zielstellung:
Die UML ist für die iterative/inkrementelle Entwicklung von Softwaresystemen sehr gut geeignet. Die vorhandenen Modellelemente eignen sich jedoch derzeit nur zur Modellierung eines konkreten Produktes.

In den letzten Jahren wurde das Konzept der merkmals-orientierten Domänenmodellierung (Feature-oriented Domain Analysis) in verschiedenen Ansätzen für spezifische Anwendungsgebiete weiterentwickelt (FODAcom, FeatuRSEB) und mit der Wiederverwendung von Software (RSEB) kombiniert. Teilweise wurden dabei verschiedene Darstellungsvarianten in UML entwickelt, die jedoch nur ausgewählte Problembereiche abdecken.

Ziel dieser Arbeit ist es, die Erweiterung der UML auf die Modellierung von Varianten aus der Sicht der Produktlinien-orientierten Softwareentwicklung mit den angebotenen Standard-Erweiterungsmechanismen zu untersuchen. Dazu soll betrachtet werden, wie man in UML Varianten modellieren könnte beziehungsweise welche (spezifikationskonformen) Erweiterungen dazu notwendig sind. Die Untersuchung soll auf die Standard-Erweiterungsmechanismen für UML beschränkt werden.

Es sind folgende Teilaufgaben zu lösen:

- Einarbeitung in Vorgehensweisen zur Variantenmodellierung
- Evaluierung bestehender Notationsformen zur Variantendarstellung in UML
- Erarbeitung eines Konzeptes zur Variantenmodellierung in UML
- Umsetzung an einem Beispiel aus der Intershop-Domäne
- Bewertung der Eignung der UML zur Variantenmodellierung

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Für die Wiederverwendung von Software bei der Entwicklung gibt es spätestens seit der starken Verbreitung des objektorientierten Paradigmas immer mehr Bestrebungen, diese Wiederverwendung systematisch zu betreiben. Diesem Ziel widmet sich die Domänenentwicklung, die in Softwarefamilien und Produktlinien zum Einsatz kommt.

Parallel dazu steht das Ziel, durch die Verwendung von Komponenten einen höheren Grad an Wiederverwendung zu erreichen. Mit der fortschreitenden Entwicklung der Komponentenidee ist die Tendenz erkennbar, diese mit Softwarefamilien zu kombinieren und auf diese Weise die Wiederverwendbarkeit von Komponenten auf eine systematische Grundlage zu stellen.

Für viele Anwendungsbereiche und Spezialgebiete gibt es bereits seit einiger Zeit Lösungen und Ansätze, die die jeweiligen spezifischen Anforderungen abdecken. Ein schneller Überblick fördert eine große Menge an Vorgehensweisen und Methoden zu Tage, die jede für sich ihre Berechtigung haben und in den meistens Fällen auf einen bestimmten Anwendungsbereich bezogen sind. Von diesen hat jedoch bisher keine einen Weg für eine konsistente und klar spezifizierte Notation gefunden.

Hier bietet die starke Verbreitung und allgemeine Akzeptanz der *Unified Modeling Language* – UML – eine Möglichkeit, die verstreuten Ansätze zusammenzuführen und auf eine gemeinsame Notationsbasis zu stellen. Die Integration von familien- und komponentenorientierter Softwareentwicklung würde ebenfalls davon profitieren und könnte das Konzept der Produktlinien unterstützen, was die wesentliche Reduktion von Zeitaufwand und Kosten ermöglicht.

1.2 Der Beitrag dieser Arbeit

Die Analyse der Gemeinsamkeiten und Variabilitäten stellt einen zentralen Schritt bei der Entwicklung von Softwarefamilien, Produktlinien und immer mehr auch von Komponenten dar. Darauf aufbauend läßt sich die Variabilität in einem Softwaresystem organisiert beschreiben und ermöglicht so eine konsequente Integration in den Softwareentwicklungsprozeß.

Dazu ist die durchgehende Modellierung ebendieser Variabilität notwendig. Das Ziel dieser Arbeit ist es, die verschiedenen Konzepte zur Beschreibung von Variabilität zu erfassen und daraus ein allgemein einsetzbares Konzept zur Modellierung von Variabilität mit Hilfe der UML zu entwickeln. Dieses wird mit Hilfe der Standarderweiterungen umgesetzt und soll ein erster Schritt zur Schaffung einer einheitlichen und konsistenten Notation sein, die möglicherweise in der Entwicklung eines Profils für die UML eine Standardisierung erfährt.

1.3 Gliederung der Arbeit

Zur Einführung in die Thematik werden im Kapitel 2 die zugrundeliegenden Prinzipien dargestellt und die notwendigen Begriffe zum Verständnis eingeordnet.

Im darauffolgendem Kapitel werden dann verbreitete Vorgehensmodelle zur Entwicklung von Produktlinien beziehungsweise Softwarefamilien kurz angesprochen, wobei der Fokus hier auf den für diese Arbeit relevanten Bestandteilen liegt: Modellierung und Notation. Im 4. Kapitel werden die bisher bekannten Notationskonzepte näher untersucht.

Das Kapitel 5 legt den Grundstein für das Ziel dieser Arbeit, indem die in der UML-Spezifikation vorhandenen Erweiterungsmechanismen kurz angesprochen werden.

Der zentrale Bestandteil dieser Diplomarbeit sind die folgenden drei Kapitel. In Kapitel 6 werden die gesammelten Erkenntnisse zusammengefaßt und eingeordnet. Darauf aufbauend entwickelt das 7. Kapitel ab Seite 39 eine Notation zur Modellierung von Variabilität in UML. Dabei werden die Darstellungsmöglichkeiten der einzelnen Konzepte in UML diskutiert (Abschnitte 7.2 bis 7.4) und das Ergebnis wird wie in einem UML-Profil spezifiziert. Abschnitt 7.5 diskutiert die Anwendungsmöglichkeiten der entwickelten Erweiterungen auf die Verhaltensmodellierung und in Abschnitt 7.6 werden Möglichkeiten zur Integration des Konzeptes in einem übergeordneten Kontext (zum Beispiel ein Vorgehensmodell) skizziert. Betrachtungen zur Anwendung der Erweiterungen und eine kurze Zusammenfassung schließen das Kapitel ab.

Der praktische Einsatz der Erweiterungen wird daraufhin im Kapitel 8 demonstriert, indem ein Beispiel aus dem Anwendungsbereich der Firma Intershop modelliert wird. Den Abschluß bildet die Bewertung der Eignung der UML zur Variantenmodellierung.

1.3.1 Stilistische Hervorhebungen

Für die Kennzeichnung besonderer Bereiche im Text wurde folgendes Schema verwendet:

- Zitate sind beidseitig eingerückt und in doppelte Anführungszeichen eingeschlossen.
- Englische Begriffe werden *kursiv* geschrieben. Sie befinden sich in der Regel bei der erstmaligen Erwähnung in Klammern hinter der deutschen Entsprechung. Da die Literatur zu den besprochenen Themen in der Regel englischsprachig ist, verfügen diese gewöhnlich über eine höhere Aussagekraft und der englische Begriff wird teilweise in einer eingedeutschten Form verwendet.
- Schlüsselwörter und Bezeichner sind `nichtproportional` gesetzt.
- Literaturbezüge sind in eckige Klammern eingefaßt, die Zuordnung von Kennung zu Titel findet sich im Literaturverzeichnis ab Seite 109.

Die Erklärung von unbekannten Begriffe, die nicht bei ihrem ersten Auftreten im Text erklärt werden, finden Sie im Glossar ab Seite 103.

Kapitel 2

Begriffsbestimmung

Zur Einstimmung auf viele Begriffe, die in dieser Arbeit immer wieder auftreten, soll als erstes – in diesem Kapitel – die Einordnung der nicht wenigen, teils doch sehr unterschiedlichen Begrifflichkeiten erfolgen. Daneben kann dieses Kapitel auch als kleine Erweiterung der Einleitung verstanden werden, um ein wenig auf die dieser Arbeit zugrundeliegenden Konzepte einzustimmen.

Wenn man sich die existierende Literatur, die sich mehr oder weniger mit der Behandlung von Variabilität in Software beschäftigt, taucht früher oder später ein Schlagwort auf, daß es schon seit längerem gibt: Produktlinien beziehungsweise die produktlinienorientierte Softwareentwicklung (PLSE, *product line software engineering*). Die damit verbundene Thematik liefert bereits eine Menge an Ansatzpunkten, die sich mit Variabilität beschäftigen.

Ein weiteres Stichwort mit wachsender Bedeutung stellen die Produktfamilien dar, die die Entwicklung mit Variabilität von der technischen Seite betrachten.

2.1 Produktliniendefinition

Produktlinien gibt es in anderen Branchen, zum Beispiel der Automobil- oder der Textindustrie schon seit langem und werden dort erfolgreich eingesetzt. Aus diesen Erfahrungswerten leitet sich die allgemein akzeptierte Definition von Produktlinien her:

> „A product line is a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission.“ [Withey96]

Die Definition charakterisiert Produktlinien aus der Sicht des Marktes, das heißt, die Bewertung erfolgt nach ökonomischen Kriterien. Das Ziel dabei ist, durch die kostensparende Fertigung der gemeinsamen Bauteile einen konkreten Nutzen, zum Beispiel die Senkung der Herstellungskosten beziehungsweise eine kürzere Markeinführungszeit (*time to market*) zu erreichen (siehe auch Definition nach [Weiss99, Seite 9]). Durch die Verwaltung einer Menge von gemeinsamen Merkmalen stellt eine Produktlinie ein Konzept zur strategischen – also langfristig geplanten – Wiederverwendung dar. Aus dieser Menge werden nach Aufstellung der Produktlinie einzelne Produkte bestimmt, indem die für ein spezifisches Produkt erforderlichen Merkmale ausgewählt werden.

Ein Beispiel für eine Produktlinie stellt eine Familie von MP3-Abspielprogrammen dar. Ausgehend von der Idee einer Abspielsoftware für MPEG-Layer-3-Dateien lassen sich verschiedene Produkte bestimmen: eine minimalistische Form für die Taskleiste von Windows-Systemen, ein ganz normaler MP3-Player, eine Variante mit zusätzlichen Fähigkeiten wie das automatische Konvertieren von Musik-CDs oder mit zusätzlichen Visualisierungen der Musik. Zu dieser Produktlinie gehört ebenfalls ein MP3-Player für das Autoradio, der in einem Spezialchip (DSP) realisiert ist und seine Daten von einem

vorhandenen CD-Laufwerk bezieht. Ein Ziel dieser Produktlinie soll es sein, den einmal entwickelten Abspielalgorithmus so kostensparend wie möglich in verschiedenen Absatzmärkten zu etablieren.

Für die Entwicklung einer Produktlinie ist es vor der Erstellung der einzelnen Produkte erforderlich, die gemeinsamen Bestandteile zu bestimmen — dies ist die Grundvoraussetzung für die Wiederverwendbarkeit dieser Bestandteile. Für die Strukturierung der Bestandteile wird wie bei normalen Softwareprodukten eine Architektur spezifiziert. Für die Strukturierung der gemeinsamen Bausteine einer Produktlinie ist analog eine Architektur üblich, die als Produktlinienarchitektur (PLA) bezeichnet wird. Gegenüber normalen Softwarearchitekturen fungiert eine PLA gleichzeitig als Referenzarchitektur für die einzelnen Produkte und legt teilweise die Architektur der einzelnen Produkte bereits fest. Dies ist aus zwei Gesichtspunkten erforderlich — einerseits setzen die vorhandenen gemeinsamen Bausteine bereits eine bestimmte Struktur voraus (eben die PLA) und andererseits bildet die Produktlinienarchitektur die Grundlage für die Instanziierung eines Produktes beziehungsweise dessen Architektur. Eine Produktlinienarchitektur kann also als Grundlage für die Produkte dienen, jedoch auch zum Beispiel als Testplattform für die generischen Bausteine verwendet werden.

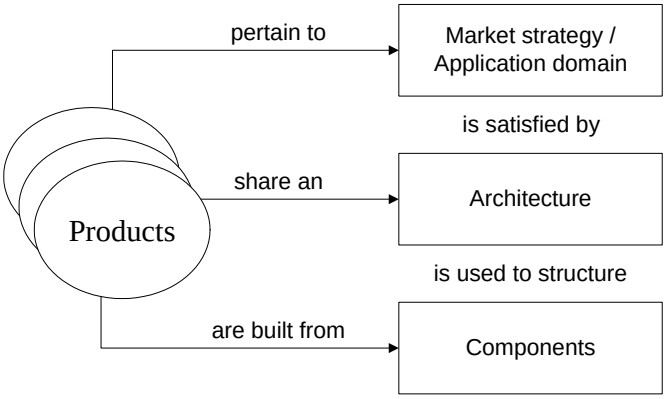


Abbildung 2.1: Software-Produktlinien (nach [Northrop])

Abbildung 2.1 verdeutlicht anschaulich die Beziehungen zwischen der Produktlinie, den damit verfolgten Zielen sowie der Architektur und ihrer Komponenten. Was die hier als Komponenten bezeichneten Bausteine der Produktlinie sind, wird im Abschnitt 2.2 näher beschrieben.

Im Gegensatz zu Produktlinien erfolgt die Unterscheidung von Produktfamilien durch technische Merkmale:

> „A product family is a group of products that can be built from a common set of assets."
> [Withey96, Czarnecki00, Northrop]

Das bedeutet, die Unterscheidung von Produktfamilien basiert auf der Betrachtung von technischen Gemeinsamkeiten, die zum Beispiel durch gemeinsam genutzte Technologien oder in allen Produkten vorhandene Bausteine ausgedrückt werden können. Eine Produktfamilie wird teilweise auch als Softwarefamilie oder Systemfamilie bezeichnet.

In der Folge kann eine Produktfamilie eine wesentlich größere Menge von Softwareprodukten umfassen und durchaus Grundlage mehrerer Produktlinien sein. Umgekehrt kann eine Produktlinie so unterschiedliche Anforderungen an ihre Mitglieder stellen, daß sie mehrere Produktfamilien einschließt ([Czarnecki00, Seite 37]).

Für das MP3-Spieler-Beispiel identifiziert sich eine (sehr große) Softwarefamilie durch die Verwendung eines gemeinsamen Algorithmus für die Dekodierung, der in Software umgesetzt ist — diese

Familie umschließt die gesamte Produktlinie. Eine andere, kleinere Softwarefamilie, die sich mit dieser teilweise überschneidet, definiert sich zum Beispiel über die zugrundeliegende Plattform — beispielsweise alle MP3-Spieler, die auf der Windows-Plattform aufsetzen, bilden eine Softwarefamilie (die nicht die gesamte Produktlinie umfaßt).

Bei der Erklärung der Produktlinien fehlt noch ein Element, das in Abbildung 2.1 als Komponente bezeichnet wurde. Dabei handelt es sich um die Bausteine, aus denen die Produkte zusammengesetzt werden oder die deren Grundlage (im Sinne einer Plattform) bilden. Bei der Betrachtung von Softwarefamilien findet sich ebenfalls eine Unterteilung in einzelne Bestandteile, die als Komponenten bezeichnet werden. In der Regel werden diese „Komponenten“ als *Assets* bezeichnet, teilweise auch als „*Core Assets*“. Core Assets stellen insbesondere die Bausteine dar, die allen Mitgliedern der Produktlinie/-familie gemeinsam sind und deren Kern bilden. In [Withey96, Seite 14] findet sich folgende Definition von Assets:

> „A software asset is a description of a solution or knowledge that application engineers use to build or modify products in a product line. To reduce work, the description must be able to explain, or implement through manipulation, changes necessary for different products. The description may be executable.“

Das bedeutet, als Asset werden Domänenmodelle, Prototypen, Designentscheidungen, Simulationen, Frameworks, Testdaten und Testfälle, Architekturen, Sprachen, Anforderungen und nicht zuletzt auch Programmcode, zum Beispiel in Form von Komponenten (-bibliotheken) sowie Codegeneratoren und Spezifikationssprachen bezeichnet.

Vereinfacht ausgedrückt, stellt ein Asset ein eigenständiges Arbeitsergebnis aus einer beliebigen Phase des Entwicklungsprozesses dar, das in irgendeiner Form dokumentiert ist. Im Normalfall wird es sich dabei um verschiedene Anforderungen, Architekturen beziehungsweise Muster, Modelle (zum Beispiel in UML) und Softwarekomponenten (im Sinne von implementierter Funktionalität mit definierter Schnittstelle, als gekapselte Einheit) handeln.

Obwohl der Komponentenbegriff auch in einem weiteren Sinne verstanden werden kann, sollen damit in der Folge nur Softwarekomponenten bezeichnet werden. Assets (=Bausteine) bilden dann die Obermenge von Komponenten und schließen auch alle anderen Entwicklungsergebnisse und -teilergebnisse ein.

Auf das Beispiel angewendet, finden wir Assets zum Beispiel im Programmablaufplan (PAP) des Dekodier-Algorithmus, in der Spezifikation des Dateiformats für Abspiellisten und der Anleitung, wie die MP3-Software zu installieren ist (als Teil des Handbuches). Andere Assets, die als Komponenten in Softwareform vorliegen, sind zum Beispiel der in einer Hochsprache kodierte Algorithmus, die COM-Komponente mit den Abspieltasten für die Benutzerschnittstelle und die Schnittstelle zwischen diesen beiden.

2.2 Domänenbegriff

Um Produktlinien oder Produktfamilien zu entwickeln, ist in beiden Fällen die Entwicklung der gemeinsamen Bestandteile erforderlich, die im Normalfall die Anforderungen eines bestimmten Anwendungsbereiches erfüllen sollen. Der Prozeß der Entwicklung dieser Bestandteile wird nach [Weiss99] als Domänenentwicklung (*domain engineering*) bezeichnet, der entsprechende Anwendungsbereich als Domäne (*domain*).

Die Definition nach Czarnecki und Eisenecker ([Czarnecki00, Seite 754]) sieht dagegen eine Domäne als einen Bereich von Fachwissen, der

1. für spezifische Anforderungen ausgewählt wird,
2. Konzepte und Begriffe beinhaltet, die durch Anwender des Fachbereiches verstanden wird und

3. das Wissen einschließt, wie Softwaresysteme (oder Teile davon) für diesen Fachbereich zu bauen sind.

Die deutsche Entsprechung ist in diesem Sinne „Fachgebiet“ oder „Fachbereich“ und wird in dieser Arbeit alternierend verwendet.

In vielen Fällen wird durch ein Softwareprodukt oder eine Produktlinie mehr als ein Fachbereich durch die Funktionalität abgedeckt. Das führt zum sogenanntem Multi-Domänen-Konzept (*multi domain*), in dem sich mehrere Bereiche überschneiden und der Schnittbereich die Kernfunktionalität beschreibt ([Knauber00, Folie 5]).

Die Definition des Domänenbegriffes ist nicht einheitlich und wird mit unterschiedlicher Granularität ausgelegt (Domäne als Fachgebiet versus Domäne als Gesamtbereich einer Anwendung). In der Folge soll in dieser Arbeit eine Domäne als Gesamtheit einer Anwendung verstanden werden, eine differenzierte Aufteilung in Fachgebiete kann mittels des Begriffes Sub-Domänen (*sub domains*) erreicht werden. Diese Interpretation ist für die Verwendung im Produktlinienkontext besser geeignet, da der Großteil der Literatur den gesamten Bereich der Entwicklung der gemeinsamen Bestandteile in Bezug zu einer Domäne betrachtet (*domain analysis*, *domain engineering*).

Die Infrastruktur, die in einer Produktlinie oder Softwarefamilie durch die Bausteine gebildet wird, stellt eine Plattform für die Instanziierung der Mitglieder dar. Für die Instanziierung gibt es zwei grundsätzliche Vorgehensweisen: Es werden die für ein spezifisches Produkt erforderlichen Merkmale ausgewählt und das Produkt mehr oder weniger daraus generiert (generative Ansätze, zum Beispiel in [Czarnecki00]). Oder die benötigten Komponenten werden zum Beispiel aus einem Komponentenpool (oder kommerziellen Angeboten, COTS) ausgewählt und zum Beispiel mittels zusätzlichem Skriptcode (*glue code*, Verbindungskleber) integriert.

Nach Griss [Griss98] ist das Domänenmodell eine abstrakte Beschreibung einer Anwendungsfamilie und stellt ein Rahmenwerk zur Beschreibung der wesentlichen Eigenschaften respektive Merkmale bereit. Die Domänenarchitektur ist dagegen präziser und beschreibt die Struktur von typischen oder allen Anwendungen der Domäne. Dazu definiert sie auch Subsysteme und deren Verbindungen sowie wichtige Techniken und Dienste.

In Verbindung mit Domänenmodellen wird oftmals auch ein sogenanntes „*Domain dictionary*“ – ein spezielles Domänen-Lexikon – eingeführt, mit dem eine einheitliche, domänenbezogene Begriffswelt für die Benutzer der Domänenarchitektur aufgebaut wird.

Zusammenfassend läßt sich sagen, daß man Produktlinien als marktorientierten Ansatz zur systematischen Wiederverwendung betrachten kann. Deren softwaretechnische Realisierung wird in der Regel mittels einer oder mehrerer Softwarefamilien erfolgen, die sich im wesentlichen auf ein Domänenmodell stützen. In der Praxis treten auch Fälle auf, in denen Produktlinien hierarchisch angeordnet sind [Bosch99], und jeweils zur Strukturierung von Teilbereichen einer Produktfamilie verwendet werden. Bei der Unternehmensorganisation können dann einzelne Produktlinien bestimmten Geschäftsbereichen (*business units*) zugeordnet werden und decken jeweils ein eigenes Spezialgebiet ab.

In der weiteren Arbeit wird nicht mehr explizit zwischen Produktlinien-, Softwarefamilien- beziehungsweise Domänenentwicklung unterschieden, da die Bedeutung sehr ähnlich und für die Untersuchung von Variabilität nicht weiter relevant ist.

2.3 Merkmalsmodellierung

In der Definition von Produktlinien wurde bereits der Begriff des Merkmales (*feature*) erwähnt. Merkmale stellen ein Mittel zur kurzen und prägnanten Beschreibung wesentlicher — aber nicht aller — Eigenschaften eines Systems dar und werden zum Teil auch als Abstraktion von Anforderungen verstanden.

In der Literatur lassen sich zwei wesentliche Definitionen von Merkmalen unterschieden: nach [Kang90] stellt ein Merkmal eine „für den Endnutzer sichtbare Charakteristik eines Systems“ dar. Die

allgemeinere Darstellung definiert ein Merkmal als „eine erkennbare Charakteristik eines Konzeptes, das für einen Interessensbeteiligten von Bedeutung ist“ [Czarnecki00, Simons96]. Beiden gemeinsam ist, das ein Merkmal eine wesentliche Eigenschaft eines Systems darstellt. Als solches dient es als kurze und prägnante Beschreibung, die als Basis für die Verwaltung von wiederverwendbaren und konfigurierbaren Anforderungen verwendet wird.

Der Vorteil von Merkmalen gegenüber den Anforderungen ist, daß sie zum einen mittels kurzer prägnanter Schlagwörter beschrieben werden und, wesentlich bedeutender, in ihrem Ausdrucksbereich weniger eingeschränkt sind. Während Anforderungen in der Regel unabhängig von Implementierungsdetails beschrieben werden sollen, können Merkmale als Gruppierung von implementierungsspezifischen Anforderungen (zum Beispiel qualitative Eigenschaften in Abhängigkeit von verwendeten Technologien) verwendet werden [Svahnbrg] und damit sowohl funktionale als auch nicht-funktionale Anforderungen ausdrücken. Daneben können Merkmale auch Aspekte reflektieren, die zu erreichende Ziele beschreiben und die das Grundelement der aspektorientierten Programmierung [Kiczales97] sind.

Die Abstraktion mittels Merkmalen führt zu einer Beschreibungsebene, die als Merkmalsmodellierung in die produktlinien- und familienorientierte Softwareentwicklung Einzug gehalten hat. In FODA [Kang90] wurde erstmals systematisch ein Merkmalsmodell entwickelt, das grafisch mittels Merkmalsdiagrammen repräsentiert wird. Diese Merkmalsdiagramme stellen eine Baumstruktur dar, in der jeder Knoten ein Merkmal repräsentiert. Die Wurzel eines Merkmalsdiagramms wird als Konzept bezeichnet. Konzepte werden in [Czarnecki00, Seite 84] als Generalisierung von objektorientierten Klassen eingeordnet und beschreiben eine Klasse von Phänomenen, die im Gegensatz zu OO-Klassen keine vordefinierte Semantik besitzen — OO-Klassen sind demzufolge Spezialisierungen von Konzepten, da ihre Instanzen Identität, Verhalten und Zustand besitzen.

Mit der Merkmalsmodellierung werden die Gemeinsamkeiten und Unterschiede (*commonalities and variabilities*) dargestellt, die in dem modellierten System bestehen. Die Unterschiede bilden dabei die Variabilität und differenzieren die einzelnen Mitglieder voneinander. Zur Bestimmung einzelner Mitglieder werden aus dem Modell die benötigten Merkmale ausgewählt, eine solche konkrete Auswahl wird auch als Merkmalsselektion oder Konfiguration bezeichnet.

Die Merkmalsmodellierung des MP3-Beispiels verfeinert das Konzept der MP3-Wiedergabe in die verschiedenen Merkmale der Domäne. Allen Produkten gemeinsam ist zum Beispiel der verwendete Wiedergabealgorithmus und das Format für die Musikquelle. Die technische Umsetzung des Formats der Listen dagegen ist abhängig von der Darstellung in der zugrundeliegenden Plattform und wird durch die verschiedenen alternativen Merkmale Binärdatei, Textdatei und XML-Textdatei ausgedrückt. Als optionale Merkmale der Domäne können zum Beispiel Möglichkeiten zum digitalen Management von Abspielrechten (DRM) oder die Unterstützung zusätzlicher Visualisierungen sein. Dieses hypothetische Merkmalsmodell ist in Abbildung 2.2 grafisch dargestellt.

2.4 Entwicklung von Produktlinien

Um das Verständnis für familien- und produktlinienorientierte Softwareentwicklung zu unterstützen, soll hier kurz das prinzipiell übliche Vorgehen bei der Entwicklung entsprechender Softwaresysteme skizziert werden. Die konkreten Vorgehensweisen sind sehr unterschiedlich, einige werden im Kapitel 3 kurz vorgestellt.

Die typische Vorgehensweise stellt sich wie folgt dar:

1. Auswahl des Anwendungsgebietes (Domäne) beziehungsweise des zu modellierenden Marktes/Marktbereiches (*scoping, scoping of economics*) — dabei wird der Umfang abgegrenzt, die anzubietenden Dienste identifiziert und die Zielgruppe der Nutzer bestimmt.

2. Domänenanalyse — Analyse des Anwendungsgebietes und Merkmalsmodellierung

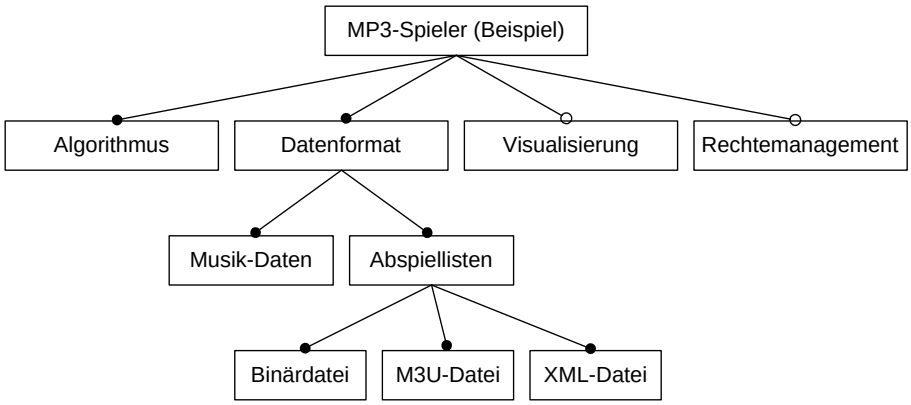


Abbildung 2.2: Beispielhaftes Merkmalsmodell in der Notation nach [Czarnecki00]

3. Architekturentwicklung und Integration/Entwicklung der Assets
4. Ableitung von Produkten: Bestimmung der spezifischen Funktionalität und Assemblierung oder Generierung des Produktes unter Verwendung der Assets (auch bezeichnet als Instanziierung der Produktlinie)
5. Feedback zur PL-Entwicklung: Einarbeitung neu entwickelter Bausteine und Weiterentwicklung der Assets

Die Domänenanalyse, das Architektur-Design und die Asset-Entwicklung werden auch unter dem Begriff Produktlinienentwicklung (*product line engineering*) zusammengefaßt. Auf der Domänenentwicklung aufbauend werden dann einzelne Produkte (*application engineering*) entwickelt beziehungsweise generiert, wobei die dabei gewonnenen Erfahrungen im Verlauf einer iterativen Entwicklung in die Weiterentwicklung der Produktlinie einfließen.

Ein möglicher Entwicklungsprozeß wird in Abbildung 2.3 dargestellt, bei dem die konventionelle Softwareentwicklung um die Domänenentwicklung erweitert wird. Im Normalfall sollten vor der Entwicklung einer Produktlinie bereits Erfahrungen mit dem Anwendungsgebiet existieren, zum Beispiel durch bereits (traditionell) entwickelte Softwaresysteme. Die daraus verallgemeinerten Bausteine bilden dann die Domänenarchitektur und werden in einem zentralen Pool (Reuse-Repository) gesammelt, aus dem sich die Anwendungsentwickler bedienen.

Unterschiedlich ist auch hier die Unterscheidung von Domänenanalyse und -entwicklung: die Domänenentwicklung wird entweder als Oberbegriff für die gesamte Entwicklung (einschließlich Analyse) verwendet oder als Teilprozeß, der auf den Ergebnissen der Analyse aufbaut.

Charakteristisch ist in jedem Falle die Trennung von Anwendungs- und Domänenentwicklung: die erstere baut auf der Domänenentwicklung auf und liefert ihr zusätzliche Anforderungen und Rückmeldungen (*feedback*) zur Weiterentwicklung. Abhängig davon, wie konsequent dieses Konzept umgesetzt wird, sollte die Anwendungsentwicklung alle ihre benötigten Bausteine aus der Domäne beziehen beziehungsweise neu entwickelte in die Domäne integrieren. Demzufolge werden dann auch die Rollen des Produktlinien-Entwicklers (*domain analyst/engineer*) und des Anwendungsentwicklers (*application engineer*) voneinander differenziert. Die Hauptverantwortung liegt bei dem Domänenanalyst, dieser muß den ganzen Anwendungsbereich kennen und so gut wie möglich modellieren. Der Anwendungsentwickler dagegen bekommt durch das Architekturmodell bereits einiges an Hilfe zur Auswahl der benötigten Merkmale und kann sich darauf konzentrieren, die Bedürfnisse des Kunden so optimal wie möglich zu erfüllen (maßgeschneiderte Software). Organisatorisch wird die Domänenentwicklung

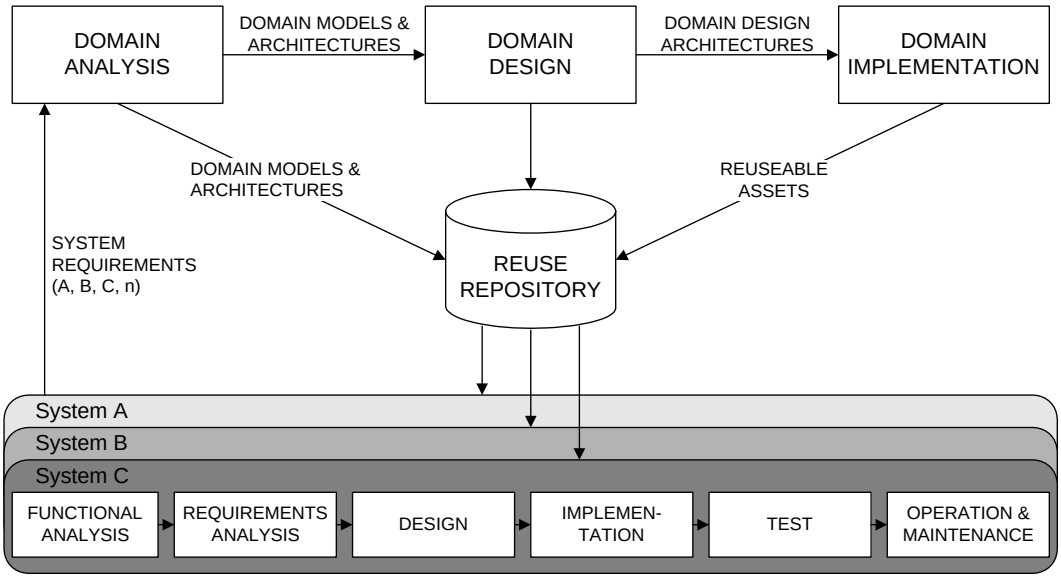


Abbildung 2.3: Entwicklungsprozeß mit Wiederverwendung nach [Chhut]

(Entwicklung und Pflege der Assets) dann oftmals in eine separate Entwicklungsgruppe ausgelagert, die für die Pflege der gesamten Domäne und ihrer Assets zuständig ist.

2.5 Variabilität in Produktlinien/Softwarefamilien

Die bisher beschriebenen Konzepte dienen primär dazu, die gemeinsam verwendbaren Teile eines Softwaresystems zu identifizieren und zu entwickeln. Ein zentraler Prozeßschritt ist, die Gemeinsamkeiten und Unterschiede (*commonalities and variabilities*) in der Analysephase (unter anderem mit Hilfe der Merkmalsanalyse) zu extrahieren und möglichst explizit zu modellieren. Bei der weiteren Entwicklung des Softwaresystems (beziehungsweise der -systeme) treten in der Regel weitere Variabilitäten auf, die durch unterschiedliche Ausprägungen eines Elementes gekennzeichnet werden. Die Analyse der Gemeinsamkeiten (als Komplement der Variabilitäten) ist die erste Grundvoraussetzung für die systematische Wiederverwendung von Bausteinen — die Variabilität dagegen unterscheidet die verschiedenen Produkte der modellierten Domäne voneinander.

Dazu ist es notwendig, in Analyse, Design, Implementierung und Test der Bausteine über Möglichkeiten zu verfügen, mit denen variable Elemente als Ausdruck von Variabilität modelliert werden können. Nach entsprechender Vorarbeit wird im Kapitel 7 eine Notation zur Darstellung von Variabilitäten in UML entwickelt, die zugleich dem UML-Standard genügt. Deren Aufgabe besteht dabei darin, dem Softwareentwickler Ausdrucksmittel zur Verfügung zu stellen, mit denen sich Variabilitäten in möglichst allen Modellierungsphasen beschreiben lassen; und die ebenso für Produktlinien und insbesondere für die Modellierung von Softwarefamilien geeignet sind.

Wenn in diesem Zusammenhang von Variabilität gesprochen wird, ist immer „Variabilität zur Entwicklungszeit“ gemeint. Diese ist deutlich zu unterscheiden von Laufzeit-Variabilität, die das dynamische Verhalten eines Programms oder Softwaresystems bestimmt. In dieser Arbeit ist deswegen implizit immer Variabilität zur Entwicklungszeit (teilweise auch als Selektionsvariabilität bezeichnet) gemeint, solange keine explizite Unterscheidung erfolgt.

Eine Definition und die Beschreibungsmöglichkeiten von Variabilität sind in Kapitel 6 zu finden. Eine gute Argumentation für die Softwareentwicklung mit systematischer Wiederverwendung im Allgemeinem findet sich in [Atkinson01, Kapitel 14]. In [Czarnecki00] erfolgt eine konzentrierte Betrachtung der Eigenschaften der Merkmalsmodellierung und eine Zusammenfassung der bekannten Vorgehensmodelle.

Kapitel 3

Methoden und Vorgehensmodelle

In diesem Kapitel werden einige der zur Zeit verfügbaren Entwicklungsmethoden kurz skizziert, die sich in der einen oder anderen Weise mit Wiederverwendung und Softwarefamilien beziehungsweise Domänenanalyse beschäftigen. Insbesondere die zuerst aufgeführten Methoden beschäftigen sich explizit mit der Wiederverwendung unter Produktlinien- beziehungsweise Softwarefamilien-Aspekten und entwickeln entsprechend geeignete Vorgehensmodelle.

Die Unterteilung in merkmals- und komponentenorientierte Methoden ist nicht als Klassifizierung zu verstehen, da teilweise einige Überschneidungen existieren. In diesen Fällen wurde die Unterscheidung nach dem stärker vertretenen Konzept getroffen.

3.1 Merkmalsorientierte Methoden

FODA — Feature-Oriented Domain Analysis

Die merkmalsorientierte Domänenanalyse [Kang90] wurde 1990 am Software Engineering Institute (SEI) der Carnegie Mellon University entwickelt. Diese Vorgehensweise liefert ein methodisches Vorgehen für die Domänenanalyse, indem Gemeinsamkeiten von verwandten Softwaresystemen analysiert und durch die generische Beschreibung von Anforderungen beschrieben werden. Dazu wird erstmals die Notation der Merkmalsdiagramme (*feature diagrams*) verwendet, die eine Abstraktionsebene für Requirements schaffen und die Abhängigkeiten zwischen diesen in grafischer Form veranschaulichen. Zur Beschreibung der Domäne verwendet FODA drei Modelltypen: Informations-, Merkmals- und operationale Modelle. Das FODA-Vorgehensmodell ist jedoch laut [Coriat00] weder architektur- noch komponentenorientiert und liefert nur eine unzureichende Prozeßbeschreibung.

MBSE — Model-Based Software Engineering

MBSE wurde 1997 vorgestellt [CMU] und integriert FODA, als Vorgehensmodell für die Domänenanalyse. Diese wird zusätzlich um Methoden für die Anwendungsentwicklung ergänzt. MBSE ist inzwischen in der *Software Product-Line Practice* (PLP) Initiative [Northrop] aufgegangen, die ein fortwährend auf der Basis von SEI-Workshops weiterentwickeltes Framework (ein “lebendes Dokument“) für die Produktlinien-orientierte Softwareentwicklung bereitstellt.

FODAcom — FODA for telecommunications

FODAcom ist eine Erweiterung von FODA für die italienische Telekommunikationsbranche [Vici00]. Sie erweitert FODA auf Anforderungsebene um die Beschreibung mit UML-Anwendungsfällen (use

cases), nutzt UML jedoch nicht für den verbleibenden Entwicklungsprozeß. Dabei wurden auch verschiedene Schlüsselkonzepte von RSEB integriert: die explizite Verwendung von Use-Case-Erweiterungen (*extension points*) und die Architekturmodellierung in verschiedenen Schichten (*layered architecture*). Use-Case- und Merkmalsmodelle werden gleichberechtigt verwendet, um die Anforderungen zu beschreiben. Die Wiederverwendung wird durch sogenannte *reuse checkpoints* unterstützt, an der zusammengefaßte Domänenmodelle generiert werden.

FeatuRSEB — Featured RSEB

Auf der Grundlage von Erfahrungen mit dem Einsatz von FODA in der Telekommunikationsbranche (FODAcom) entwickelten Griss, Favaro und d'Alessandro [Griss98] die RSEB-Methode (siehe Seite 13) weiter zu FeatuRSEB. Dabei kombinierten sie die Wiederverwendung von Komponenten mit der Merkmalsmodellierung aus FODA ("*featured RSEB*").

Die Funktion des Use-Case-Diagrammes in RSEB wird durch das Merkmalsdiagramm ergänzt. Im Gegensatz zu RSEB wird die Funktion der Use-Case-Modellierung aufgeteilt: Use-Cases modellieren das „Was tun die Systeme?" einer Domäne, durch das Merkmalsmodell wird das „Welche Funktionalität gibt es?" dargestellt.

Die explizite Variantendarstellung, die bei RSEB oftmals in die Use-Case-Modelle einfließt (*variation points*, [Jacobson97]), wird durch das Merkmalsdiagramm übernommen. Dieses nimmt im „4+1"-Modell [Kruchten95] die zentrale Position ein, die den primären Zusammenhang zwischen den einzelnen Sichten herstellt (*feature-model centric*). Die Modellierung aus der Benutzersicht (durch Use-Case-Modell) wird durch die Sicht des Wiederverwenders (*Reuser*) ersetzt.

PuLSE — Product-Line Integrated Software Engineering

PuLSE wird am Institut für Experimentelle Softwareentwicklung (IESE) der Fraunhofer Gesellschaft entwickelt [Bayer99a]. Es setzt sich aus verschiedenen Teilphasen zusammen, deren Gesamtheit eine umfassende Methodik für die Produktlinienentwicklung und ein umfangreiches Prozeßmodell zur Verfügung stellt. Diese Phasen sind:

- PuLSE-Eco — *Scoping and Economics*,
- PuLSE-CDA — Domänenanalyse,
- PuLSE-DSSA — Architekturentwicklung als zentraler Bestandteil einer Produktlinien-Infrastruktur,
- PuLSE-I — Anwendungsentwicklung: Prozeß zur Ableitung von Mitgliedern und der Weiterentwicklung (*maintenance*) der Infrastruktur,
- PuLSE-EM — Konfigurationsmanagement und
- RE-PLACE — Re-Engineering für Produktlinien.

Dabei werden abstrakte technische Komponenten beschrieben, die in den einzelnen Phasen entstehen. Das gesamte Prozeßmodell stellt ein Rahmenwerk für den konkreten Einsatz dar, bei dem der erste Schritt darin besteht, das PuLSE-Modell an die Erfordernisse der jeweiligen Umgebung anzupassen.

Generative Programming

Die generative Programmierung nach Czarnecki und Eisenecker [Czarnecki00] ist ein merkmalsorientierter Ansatz zur Umsetzung von Produktfamilien. Merkmalsdiagramme bilden einen integrierten

Bestandteil zur Verwaltung der Variabilität, die durch generative Verfahren umgesetzt wird. Damit soll die Wiederverwendung auf Basis der Modellierung von Softwaresystemfamilien erreicht werden. Dazu werden aufbauend auf bestimmten Anforderungen hoch-angepaßte und -optimierte Bausteine aus elementaren, wiederverwendbaren Implementierungskomponenten zusammengesetzt, indem das sogenannte Konfigurationswissen für die Abbildung zwischen Problem- und Konfigurationsbereich genutzt wird. Verschiedene Teile der Konfiguration können dabei zu verschiedenen Zeitpunkten (*binding times*) verwendet werden.

In der generativen Programmierung wird die Merkmalsmodellierung nach FODA durch die Einführung von Oder-Merkmalen (*or-features*) erweitert und nach verschiedenen Kriterien kategorisiert. Von diesen Oder-Merkmalen muß bei der Selektion mindestens eines ausgewählt werden, wenn das übergeordnete Merkmal ausgewählt wurde.

FORM — Feature-Oriented Reuse Method

Kyo C. Kang, der wesentlich an FODA beteiligt war, erweiterte dieses in FORM [Kang98] um Konzepte für die Design- und Implementierungsphase. Analog zu FODA startet der Entwicklungsprozeß jedoch ebenfalls mit der sorgfältigen Analyse der Domäne als Voraussetzung für spätere Entwicklungsschritte.

Die Analyse beginnt mit der Identifikation von Merkmalen zum Beispiel mit Hilfe der Domänensprache und vorhandenen Benutzerhandbüchern. Die gewonnenen Merkmale werden klassifiziert in *Capabilities* (Fähigkeiten), *Operating environment* (Ausführungsumgebung), *Domain technology* (Technologie) und *Implementation technique* (Implementierungstechniken).

Die Zusammenhänge zwischen den Merkmalen (die Merkmalsorganisation) werden in einem Merkmalsdiagramm grafisch dargestellt: dieses besteht aus einer UND/ODER-Hierarchie (Baumdarstellung) und den logischen strukturellen Beziehungen, die in Kompositionen, Generalisierungen und Implementierung unterschieden werden. Die Beziehungsart wird durch unterschiedliche grafische Gestaltung der Linientypen dargestellt. Die Merkmale selbst werden in notwendig, optional und alternativ klassifiziert. Die verwendeten Beispieldiagramme enthalten relativ wenig Hierarchieebenen und keine explizite Unterscheidung eines Konzeptknotens.

Neben dem Merkmalsdiagramm enthält das Merkmalsmodell *Composition rules*, die Beziehungen zwischen Merkmalen modellieren, und Herkunft und Entscheidungen (*issues and decisions*), die auswahlrelevante Modellaspekte dokumentieren.

Daneben wird anhand des Merkmalsmodells ein Kriterium für den Grad der Gemeinsamkeiten aufgestellt: viele UND-Knoten im oberen und ODER-Knoten im unterem Bereich des Baumes indizieren einen hohen Grad von Wiederverwendungsmöglichkeiten. Außerdem wird ein typisches Problem angesprochen: für größere Domänen werden die Merkmalsdiagramme komplex und unhandlich, so daß das Merkmalsdiagramm zur Darstellung der Beziehungen um Textbeschreibungen der Merkmale ergänzt wurde.

3.2 Komponentenbasierende Methoden

RSEB — Reuse-driven Software Engineering Business

RSEB von Jacobson, Griss und Jonsson [Jacobson97] ist eines der Standardwerke, die sich mit der systematischen Wiederverwendung von Software beschäftigen. Basierend auf der Modellierung mit UML werden Vorgehensweisen beschrieben, um die komponentenbasierte Wiederverwendung in den Softwareentwicklungsprozeß zu integrieren. RSEB ist eines der ersten Werke, die sich ansatzweise mit Variabilität auf Komponentenebene beschäftigen: durch die Einführung und Definition von Variationspunkten für Klassen, Use Cases und Typen werden unterschiedliche Ausprägungen von Elementen

modelliert. Diese Modellierung beschränkt sich jedoch auf die genannten Elemente und auf Unterschiede, die sich durch Generalisierungstechniken beschreiben lassen. [Coriat00] charakterisiert die Wiederverwendung in RSEB als opportunistisch, aber nicht systematisch.

KobrA — Component-Based Product Line Development

KobrA [Atkinson00] ist ein Vorgehensmodell für die komponentenbasierte Entwicklung von Anwendungs-Frameworks. Es enthält Methoden für die Komponentenentwicklung, die so konkret und beschreibend wie möglich gefaßt sind. Dazu gehört die strikte Trennung von Produkten und Prozessen sowie die eindeutige Zuordnung der Zwischenergebnisse zu Prozeßschritten. Alle Ergebnisse des Entwicklungsprozesses dienen der Beschreibung von individuellen Komponenten — mit dem Vorteil, daß die Komponenten und ihre beschreibenden Dokumente unabhängig voneinander wiederverwendet werden können.

Aus der Produktlinienperspektive stellt KobrA eine gebrauchsfertige und objektorientierte Instanz der PuLSE-Methode dar und übernimmt von dieser das Vorgehensmodell. Die Aktivitäten in PulSE werden durch entsprechende Entwicklungsaktivitäten in KobrA abgebildet und untergliedern sich in die Entwicklung des Komponentensystems (als Framework bezeichnet) und die darauf aufbauende Anwendungsentwicklung. Die Unterscheidung der einzelnen Framework- beziehungsweise Anwendungsentwicklungs-Aktivitäten erfolgt durch das Maß der Generalität/Spezifik des jeweiligen Arbeitsschrittes und nicht durch dessen Detaillierungsgrad.

Sowohl Framework- als auch Anwendungsentwicklung basiert auf der Entwicklung einzelner Komponenten. Bei der Transformation in eine ausführbare Anwendung wird aus den instanziierten UML-Modellen durch eine Reihe von Arbeitsschritten (Verfeinerungen und Übersetzungen), die grundsätzlich orthogonal zu der Framework-Entwicklung sind, ausführbarer Code erzeugt.

SPLIT — Software Product Line Integrated Technology

SPLIT [Coriat00] ist eine experimentelle komponentenbasierte und architekturzentrierte Vorgehensweise zur Umsetzung von Produktlinien, die von Anforderungen gesteuert wird. Das SPLIT-Vorgehensmodell orientiert sich an STARS [McCabe93] – dem Vorgehensmodell des Verteidungsministeriums der Vereinigten Staaten – und gliedert sich in vier Teile: Anforderungsentwicklung, Produktlinienarchitektur, Prozeßmodell und Komponentenentwicklung. Die Entwicklung baut auf der Unterteilung in Assets auf und ist prozeßgesteuert, für die Modellierung wird die UML-Notation verwendet. Die Domänenentwicklung gliedert sich in 3 Teile: Anforderungen, Architektur und Softwarekomponenten. Dabei wird die Bedeutung der Nachvollziehbarkeit (*traceability*) betont, mit der die korrekte Anwendungsentwicklung gesichert wird und die die verschiedenen Entwicklungsebenen miteinander verbindet.

In SPLIT werden Variationspunkte zur Beschreibung und Lokalisierung von Variabilität verwendet und durch ein mehrstufiges Entscheidungsmodell ergänzt. Dieses Entscheidungsmodell dient zur Unterstützung bei der Auswahl der Varianten für konkrete Produkte. Die Entwicklung der Anforderungen erfolgt systematisch mittels Fähigkeiten (*capabilities*) und Zwänge (*forces*); letztere bilden die Verfeinerung von nicht-funktionalen Anforderungen. Die Gruppierung der Anforderungen erfolgt mit Hilfe von Aspekten. Das Architekturmodell wird mit Hilfe von drei Sichten (*business*, *technology*, *subsystem*) beschrieben, die jeweils einzelne Perspektiven von Interessensbeteiligten beschreiben. Das Ganze mündet in ein konzeptionelles Framework für die Architekturbeschreibung in UML.

FAST — Family-Oriented Abstraction, Specification and Translation

FAST ist das von Lucent Technologies eingesetzte Vorgehensmodell zur Entwicklung von Softwarefamilien [Weiss99]. Es ist hauptsächlich prozeßorientiert und systematisiert die Extraktion und Entwicklung von Anforderungen und deren Dokumentation an die Mitglieder einer Softwarefamilie. Es unter-

stützt die schnelle Erstellung von Software (rapid software production) und die automatische Generierung der Familienmitglieder. Diese teilen sich Anforderungen, Design und Programmcode. Zur Auswertung und Entscheidungsunterstützung enthält FAST Methoden zur Kostenanalyse und -Bewertung. Die Unterstützung der Modellierung beziehungsweise Notation von Softwarebausteinen ist jedoch wenig ausgeprägt.

FUSION — Object-Oriented Development

Fusion [Coleman94] wurde als Vorgehensmodell bei Hewlett Packard zur Unterstützung der Softwareentwicklung entwickelt. Es gliedert sich in die Phasen Analyse, Design und Implementierung und spricht Probleme der Wiederverwendung explizit an. Eine eigene Notation für alle Modelle wird durch Verwaltungswerkzeuge unterstützt. Fusion läßt sich anpassen und für kleinere Projekte in einer vereinfachten Version nutzen. Auffallend ist die Entwicklung eigener Notationen für Klassen, Relationen (ähnlich wie E/R-Modelle) und Aktivitätsmodelle (wie Kollaborationsdiagramme in UML), die sich wesentlich an die strukturierte Modellierung anlehnen.

Catalysis — Objects, Components and Frameworks in UML

Catalysis [Souza99] ist ein objektorientiertes, umfassendes Vorgehensmodell, das sich auf die komponentenorientierte Entwicklung konzentriert. Die eingesetzte Modellierungsnotation orientiert sich jedoch mehr an OOSE als an UML [Stevens99], dabei werden teilweise auch die Bedeutungen von UML-Elementen verändert (zum Beispiel Use Cases werden zu *actions*, mit leicht veränderter Bedeutung). Im Mittelpunkt steht eine Architektur, die mit expliziten Verbindungen (*connectors*) beziehungsweise steckbaren Komponenten (*pluggable components*) arbeitet.

Das Vorgehensmodell von Catalysis ist gut strukturiert (Phasen, Abschnitte, Schritte, Aufgaben) und unterstützt die evolutionäre Implementierung und iterative Auslieferung (*iterative deployment*). Daneben werden eine Menge von Hilfsmitteln zur Gestaltung des Projektablaufs angeboten, zum Beispiel Word-Vorlagen und Rollen der Beteiligten für Projektstartbericht und UML-Diagramme zum Erfassen der Geschäftsprozesse.

Kapitel 4

Bestehende Notationen für Variabilität

Aus den diskutierten Vorgehensmodellen und weiteren Literaturquellen sollen in diesem Kapitel die Notationsformen beziehungsweise -vorschläge extrahiert werden, die zur Darstellung von Variabilität und relevanten Konzepten verwendet wurden.

4.1 Merkmalsmodelle nach FODA

Nach FODA besteht ein Merkmalsmodell aus 4 Bestandteilen: dem Merkmalsdiagramm (*feature diagram*) zur hierarchischen Zerlegung von Merkmalen, den Merkmalsdefinitionen einschließlich Beschreibung und Bindezeit (*binding time*), den Auswahlregeln (composition rules), um erlaubte und ungültige Kombinationen von Merkmalen zu bestimmen und den Gründen (*rationale*) für oder gegen die Auswahl eines Merkmals. Letztere werden oftmals umgangssprachlich beschrieben, da eine formale Spezifikation in vielen Fällen (zum Beispiel der geschätzte Ressourcenverbrauch bei Merkmals-Einbindung) nicht möglich ist beziehungsweise nicht vollständig wäre.

Das Merkmalsdiagramm ist als Baumdiagramm aufgebaut, dessen Wurzel als Konzept (*concept*) bezeichnet wird. Jeder dem Konzept untergeordnete Knoten stellt ein Merkmal dar. Es werden drei Typen von Merkmalen unterschieden: notwendige (*mandatory*), optionale und alternative Merkmale. Prinzipiell kann ein Merkmal nur ausgewählt werden, wenn dessen übergeordneter Knoten ebenfalls ausgewählt ist — die in jeder Auswahl enthaltene Merkmalsmenge stellt die Gemeinsamkeiten des Modells (beziehungsweise der Domäne) dar. Zusätzliche, nicht durch die Baumstruktur ausgedrückte Abhängigkeiten können durch zusätzliche Verbindungen beschrieben werden: für die Abhängigkeit voneinander (*requires*, teilweise auch als *mutually-inclusive* bezeichnet) und für den gegenseitigen Ausschluß (*mutually-exclusive*, *mutex*). Zusätzlich sollte jedes Merkmal mit Gründen und Argumentationen für die Auswahl versehen werden, die als *rationale* zusammengefaßt werden und zum Beispiel Ressourcenverbrauch oder sinnvolle Selektionen beschreiben.

Merkmale werden in FODA in verschiedene Kategorien unterteilt, die in Abschnitt 6.1.2.4 dargestellt sind. Zusätzlich findet eine Gruppierung nach der Bindezeit statt, die in Übersetzung (*compile-time*), Aktivierung (*activation-time*, *load-time*) und Laufzeit (*runtime*) unterschieden wird.

4.2 Notationen mit Erweiterung der UML

4.2.1 Variationspunkte in RSEB

In „Software Reuse“ von Jacobson, Griss und Jonsson [Jacobson97] werden zur Modellierung Variationspunkte (*variation points*) verwendet. Für Notationszwecke wird eine Form der UML verwendet, die sehr viel mit Stereotypen und den Möglichkeiten des *Model Managements* arbeitet.

Variationspunkte treten in RSEB in erster Linie in Anwendungsfällen (*use cases*) und Typen (Komponenten, Schnittstellen, Klassen) der Analysephase auf. Gekennzeichnet werden sie durch einen gefüllten Punkt in der grafischen Darstellung und der textuellen Bezeichnung des Variationspunktes in geschweiften Klammern (vergleichbar mit der Notation von Constraints in UML) neben dem betreffenden Modellelement. Zusätzlich wird das Objekt noch mit dem Stereotyp «`variation point`» versehen.

Zur Herstellung der Verbindung zwischen Anwendungsfällen (*use cases*) und Objektmodell werden Abhängigkeiten (*dependencies* in UML) verwendet, die mit «trace» stereotypisiert werden. Dabei kann ein Variationspunkt aus dem Use-Case-Modell in verschiedenen Variationspunkten im Objektmodell umgesetzt sein. Die Darstellung der Abhängigkeiten erfolgt in der Regel in der Detaildarstellung von Packages, in der sich Use Cases und Objekte gleichzeitig darstellen lassen [Jacobson97, Seite 139].

Die Auswahl und Darstellung der Mechanismen zur Umsetzung der so ausgedrückten Variabilität (Vererbung, Erweiterung, Parametrisierung, Schablonen, Generierung) bleibt dem Modellentwickler überlassen.

4.2.2 FeatuRSEB-Notation

Bei der Modellierung der Gemeinsamkeiten und Unterschiede in FeatuRSEB [Griss98] wird zwischen obligatorischen (*mandatory*), optionalen und alternativen (*variant*) Merkmalen unterschieden. Diese bilden wie in FODA über eine „besteht aus“-Beziehung (*consists-of-relationship*) eine hierarchische Baumstruktur, die mit dem Konzept als Wurzel abgeschlossen wird. Die alternativen Merkmale (*vp-features*, Variationspunkte) werden zusätzlich nach dem Zeitpunkt ihrer Festlegung unterschieden: während der Anwendungserstellung (*reuse time*, `XOR`-Verknüpfung aus Entwicklersicht) oder zur Laufzeit (*use time*, `OR`-Verknüpfung aus Entwicklersicht). Zur Modellierung von Abhängigkeiten zwischen optionalen und variablen Merkmalen, die nicht durch die Baumstruktur dargestellt werden, können zusätzliche Einschränkungen (constraints) verwendet werden: Abhängigkeiten (*requires*) und wechselseitiger Ausschluß (*mutual exlusion*).

In einer erweiterten Darstellung kann das Merkmalsmodell als UML-Klassendiagramm dargestellt werden, in dem jeder Merkmalsknoten als Klasse mit dem Stereotyp «feature» modelliert wird. Die Beziehungen zwischen den Klassen werden durch benannte *Dependency*-Beziehungen dargestellt, die zusätzlichen Merkmalseigenschaften wie Bindezeitpunkt und Typ des Merkmals werden als Attribute der Klasse abgebildet:

Attribut	Beschreibung / Wertebereich
`description`	textuelle Beschreibung des Merkmals
`source`	Wo wurde es identifiziert (Fallbeispiele etc.)?
`nature`	funktional, architectural, implementation
`existence`	Merkmalstyp: mandatory, optional oder variant
`alternative`	existieren Alternativen?
`category`	operational, context, ...
`bindingTime`	Zeitpunkt der Festlegung: reuse oder run-time
`issuesAndDecisions`	Auswahlkriterien und Entscheidungshilfen
`notes`	ergänzende Kommentare

4.2.3 KobrA-Notation

In einigen weiteren Werken [Bayer99a, Svahnbrg] findet sich die nicht standardisierte Abwandlung der UML-Notation, optionale Modellelemente mit unterbrochenen anstelle von durchgezogenen Linien darzustellen. Die Beziehung zu den entsprechenden Merkmalen wird in der Regel über Kommentare hergestellt. Wie in Abschnitt 4.3.2 gezeigt, wird diese Darstellung auch für nicht-objektorientierte Ausdrucksmittel verwendet.

In [Muthig00a] wird eine Erweiterung der UML skizziert, in der variable Kardinalitäten einer Assoziation mit einer gefüllten Raute gekennzeichnet sind. In Aktivitätsdiagrammen werden Verzweigungen, die mit optionalen Elementen korrelieren, als schwarz gefüllte (beziehungsweise invertierte: weise Schrift auf schwarzem Hintergrund) Rauten dargestellt. Im Kollaborationsdiagramm werden optionale Elemente mit den schon erwähnten gestrichelten Elementen gekennzeichnet. Unterstützt wird diese Darstellung durch ein Auswahlmodell für die optionalen Elemente in Tabellenform. Realisieren lassen sich diese Darstellungsformen nur durch die spezifische Werkzeugunterstützung, indem individuelle Linientypen und Füllfarben für die Modellelemente gesetzt werden.

4.3 UML-konforme Erweiterungen

4.3.1 Merkmalsmodellierung nach Hein et al

In [Hein00] entwickelt eine Autorengruppe der Firma Bosch die bisher dargestellten Notationen weiter und paßt sie an die Anforderungen ihrer Domäne – der Fahrzeug-Peripherie-Überwachung (*car periphery supervision*, CPS) – in der Industrie an.

Zur Darstellung von Variabilität auf Requirement-Ebene werden dazu Parameter verwendet, so daß die betroffene Anforderung den Charakter eines Musters (*template*) erhält.

Erweitertes Metamodell

Als Notationssprache zur Merkmalsmodellierung wurde aus pragmatischen Gründen – Verbreitung und Werkzeugunterstützung – die UML verwendet. Die Autoren bewerten die bisher bekannten Darstellungsmöglichkeiten – mit und ohne UML – zur Merkmalsmodellierung als ungenügend. Insbesondere der Mangel an Ausdrucksmöglichkeiten für Merkmale, die an verschiedenen Stellen und mit verschiedenen Merkmalen im Merkmalsmodell auftreten, führte zur Entwicklung eines erweiterten Metamodells zur Merkmalsmodellierung. Es baut auf der in FODA eingeführten und in FeatuRSEB auf UML angepaßten Notation für Merkmale mittels UML-Klassen und Abhängigkeitsrelationen auf und erweitert diese um eine zweite Strukturdimension, die Rollen. Damit wird es einem Merkmal möglich, in mehreren Rollen – analog zu verschiedenen Stellen im Merkmalsbaum – aufzutreten. Die Rollen besitzen zusätzliche Meta-Attribute für Bindezeitpunkt und Zusammensetzungstyp (*decomposition type*), von denen letzterer die Zusammensetzung der untergeordneten Merkmale beschreibt. Die Modellierung erfolgt mittels Stereotypen, diese bilden dabei ein eigenes Metamodell (Vererbungsstruktur und Assoziationen), das in Abbildung 4.1 dargestellt ist. Gegenüber dem Original wurden die Relationen zwischen den Stereotypen etwas vereinfacht — jede gerichtete Assoziation hat dort zusätzlich ein alternatives, ungerichtetes Pendant — bei der Modellierung können wahlweise entweder gerichtete oder ungerichtete Beziehungen zwischen den Rollen benutzt werden. Für das Modell werden nur jeweils die Stereotypen verwendet, die keine Spezialisierungen haben. Auffallend ist die Unterteilung in einfache (*simple*, entspricht Merkmalsblättern im Feature-Baum), zusammengesetzte (*compound*) und alternative Merkmale, die jeweils eine optionale Entsprechung haben.

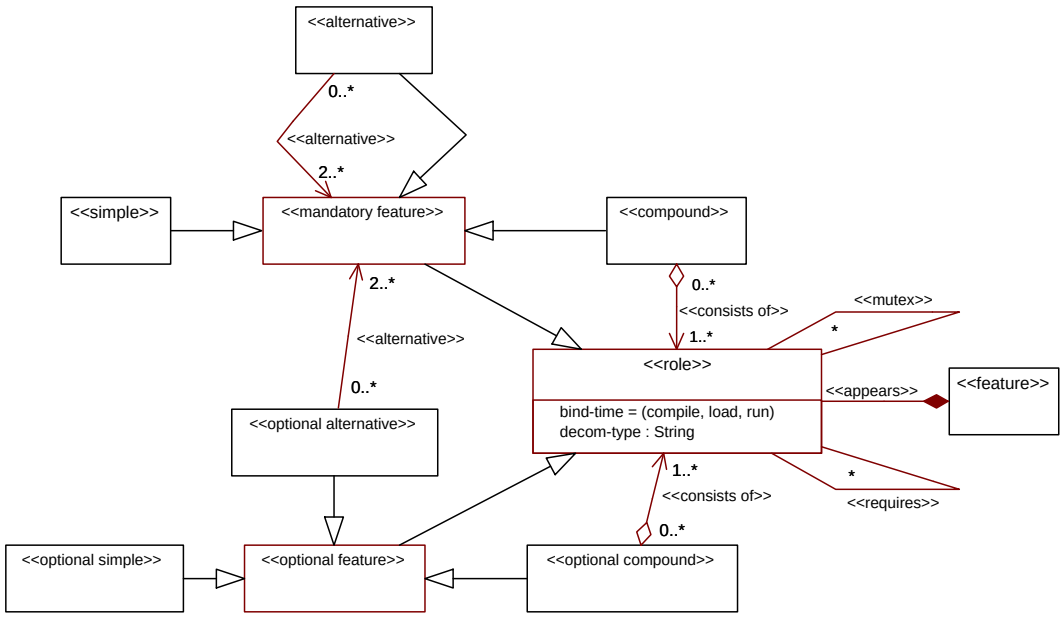


Abbildung 4.1: UML-Metamodell für Merkmalsmodellierung mit Rollen

Mapping zu UML

Zur Darstellung in UML wird eine Zuordnungsvorschrift (*mapping*) definiert. Die Darstellung jedes Merkmals erfolgt als Paket (*package*) mit dem Stereotyp «feature». Durch die impliziten Abhängigkeitsstrukturen (siehe Kapitel 5) zwischen Paketen in UML wird der Merkmalsbaum ohne zusätzliche Relationen abgebildet. Die einem Merkmal zugeordneten Rollen sind dann jeweils Elemente des Merkmals-Packages und werden jeweils aus dem Paket importiert, in dem sie definiert sind.

Im Mapping werden die Rollen in Container- (*compound*, *alternative*) und Blatt-Merkmale (*simple*) unterteilt und entsprechend durch Packages und Klassen-Symbole dargestellt. Zusätzlich werden diese durch optional und *directed* erweitert, so daß insgesamt 12 Merkmalsarten respektive Stereotypen definiert werden.

Die Spezifikation jedes Merkmales erfolgt innerhalb des Paketes in einer eigenen Klasse «feature specification», die alle Attribute enthält. Abgeleitet wird diese Klasse von einem «trace point», der als Assoziationsendpunkt für die Darstellung der requires- und mutex-Abhängigkeiten zwischen Merkmalen dient.

Diese Darstellung von Merkmalen erlaubt die implizite Gliederung analog des Merkmalsbaumes und verhindert Namenskonflikte. Ebenso können weitere Modellelemente eindeutig einem Merkmal zugeordnet werden – jedoch nur genau einem! Auf der anderen Seite sorgt die großzügige Verwendung von Klassen/Packages, von denen einige nur durch einen Stereotyp gekennzeichnet werden, nicht gerade für Deutlichkeit und erschwert den schnellen Überblick.

4.3.2 „Variante Requirements“ von Jarzabek

Stan Jarzabek entwickelt in [Jarzabek00, Cheong98] Notationensvarianten, um Variabilität in den Requirements ausdrücken zu können.

Grundlegendes Element ist die Identifikation jeder Anforderung über einen eindeutigen Bezeich-

ner, der bei aussagekräftiger Namensgebung entsprechend lang ausfällt. Über eine textuelle Notation erfolgt die formale Darstellung der Beziehungen zwischen den Anforderungen (Implikationen mit eingeschlossenen Logikausdrücken: NOT, OR, AND und XAND[1]). Zusätzlich werden die betroffenen Requirements mit dem Suffix `-ALTx` oder `-OPT` versehen, wenn sie Alternativen (`x` ist die laufende Nummer der Variante) darstellen beziehungsweise optional sind.

Die Darstellung variabler Elemente in verschiedene Diagrammarten erfolgt mittels gestrichelter Linien, die Zuordnung des Elements zu dem zugrundeliegenden Requirement über Textkommentare. Diese Notation wird in [Cheong98] für Diagrammformen der strukturierten Programmentwicklung (*Data Flow Model*, *Entity-Relationship-Model*, *State Transition Model*) angewendet und ebenso in UML-Diagrammen — ist also von der verwendeten Modellierungssprache unabhängig. Zur Unterstützung bei der Anpassung und der Auswahl von varianten Anforderungen wird der sogenannte *Configuration* oder *Customization Decision Tree* (CDT) verwendet, der im Vergleich zu einem Merkmalsmodell zusätzliche Anweisungen (*customization scripts*) darüber enthält, wie die generische Architektur an einen konkreten Einsatzfall anzupassen ist. Eine solche Anpassungsanleitung ist jeweils immer einem Blatt des Merkmalsbaumes zugeordnet.

4.3.3 Modellierung von Produktfamilien nach Gomaa

Hassan Gomaa [Gomaa00] beschreibt die Verwendung der UML zur Darstellung von Produktfamilien und der Modellierung von Anwendungsdomänen.

Im statischen Modell unterscheidet er nur zwischen Klassen, die zu allen Mitgliedern der Produktfamilie gehören, und optionalen Klassen. Zur Unterscheidung wird der Stereotyp «kernel» für stets vorhandene und «optional» für optionale Klassen benutzt. Die Varianten werden dabei mittels Generalisierungs-/Spezialisierungs-Hierarchien dargestellt. Dadurch wird auch die Modellierung von unterschiedlichen Verhalten durch verschiedene Klassen umgesetzt und entsprechend disjunkt modelliert, da jede Variante ihre eigene Zustandsmodellierung besitzt. Für Kollaborationsdiagramme werden nach Gomaa's Auffassung die optionalen Objekte durch die Anwendungsfälle bestimmt, die bereits als `kernel`, `optional` oder `variant` kategorisiert wurden, so daß keine spezifische Notation notwendig ist.

Zur Zuordnung der (optionalen) Klassen zu den entsprechenden Merkmalen (*feature-class-dependency*) wird die Package-Notation vorgeschlagen. Das betreffende Paket wird mit den Namen des Merkmals und dem Stereotyp «feature» versehen und enthält die zugeordneten optionalen Klassen. Dies impliziert eine 1:N-Beziehung zwischen Merkmal und den implementierenden Klassen. Entsprechend bilden die «kernel»-Elemente die allen Produkten (der Familie) gemeinsamen Merkmale.

Zur Modellierung auf Use-Case-Ebene werden die vorhandenen Elemente für Beziehungen («include», «extend») verwendet. Die Ableitung des Modelles für ein Zielprodukt erfolgt durch die Auswahl der benötigten optionalen Bestandteile, während «kernel»-Bestandteile immer übernommen werden.

4.3.4 Produktlinien-Modellierung in SPLIT

Die Produktlinienmodellierung in [Coriat00] fundiert auf der Modellierung von Variabilität, die durch drei Eigenschaften beschrieben wird:

- Was variabel ist und welche Ausprägungen es gibt? (Beschreibung aller Varianten),
- wo die Variabilität auftritt (ausgedrückt durch Variationspunkte) und
- Entscheidungen, die bei der Auswahl von Varianten zu berücksichtigen sind (Entscheidungsmodell, *decision model*).

[1] alle oder keines, Äquivalenz-Operation

Zur Darstellung stellt SPLIT zwei Konzepte zur Verfügung, die zur Modellierung eingesetzt werden: Variationspunkte und das Entscheidungsmodell.

4.3.4.1 Variationspunkte

Variationspunkte beschreiben das Wesen und den Ort von Variabilität. Dazu wird ein Variationspunkt (Vp) durch die folgenden Attribute charakterisiert:

Attribut	Beschreibung / Wertebereich
`name`	Name in Textform
`description`	begründete Darstellung des Vp
`stakeholder`	Für wen der Vp erzeugt wurde?
`variability mechanism`	insert, extend, parameterize
`existence`	Variantenintegration: optional, mandatory
`alternative`	Variantenbeschreibung: complete, incomplete
`decision`	OCL-Invarianten des lokalen Entscheidungsmodells
`binding time`	derivation time, configuration time, run time
`binding occurence`	Anzahl möglicher Varianten: once or multiple

An dieser Stelle fällt auf, daß alternative und optionale Varianten nicht getrennt voneinander modelliert werden, sondern sich nur durch die Attributbelegung unterscheiden — das macht das Modellierungselement „Variationspunkt“ universeller einsetzbar.

Diese Variationspunkte werden in UML als Klassen (Klassendiagramm) oder als Pakete (Package-Diagramm) dargestellt. Der Variationspunkt wird als Instanz der jeweiligen Metaklasse mit dem Stereotyp «variationPoint» dargestellt, die über eine gerichtete Assoziation (mit Stereotyp des jeweils verwendeten Mechanismus) mit dem Element verbunden wird, das die Variation enthält. Der «variationPoint»-Stereotyp wird in der iconifizierten Darstellung (siehe Kapitel 5) mit einem gefüllten Kreis gekennzeichnet, der an die Notation aus RSEB erinnert. Mit dem Variationspunkt-Element werden dann die modellierten Varianten assoziiert.

4.3.4.2 Entscheidungsmodell

Das zweite Konzept, ein mehrstufiges Entscheidungsmodell (*decision model*), unterstützt die Entwicklung von Applikationen. Dabei werden Entscheidungskriterien beziehungsweise -bedingungen für die Festlegung von Varianten modelliert, die sich über drei Modellierungsebenen erstrecken:

1. *Variation point level*: Auswahl von Varianten für die Variationspunkte
2. *Asset level*: Auswahl von Variationspunkten aus der Gesamtmenge der Vp eines einzelnen Bausteins
3. *Core level asset*: Auswahl von Gruppen von Bausteinen für ein einzelnes *Core Asset* (Anforderungen, Architektur, Komponenten)

Das globale Entscheidungsmodell ist die hierarchische Komposition dieser drei Stufen. Zur Darstellung der Entscheidungsmodelle wurden 3 Möglichkeiten identifiziert:

- in Textform in UML-Kommentaren, die mit dem jeweiligen Variationspunkt assoziiert werden
- präzise Beschreibung mit Hilfe von OCL
- mittels UML-Verbindungen, um Beziehungen und Prioritäten zwischen Variationspunkten auszudrücken — das Entscheidungsmodell ist in diesem Fall ein gerichteter azyklischer Graph von Variationspunkten.

Daneben findet sich in SPLIT noch ein Metamodell zur Beschreibung des Entwicklungsprozesses, auf das an dieser Stelle nicht weiter eingegangen wird.

Zusammenfassend finden sich hier Ausdrucksmöglichkeiten für die Modellierung von Variabilität, die sich jedoch auf die Beschreibung von Variationspunkten in Paketen und Klassen beschränkt. Damit können die wesentlichen Aspekte bereits gut verständlich beschrieben werden. Allerdings fehlt ein genereller (abstrahierender) Überblick über die Variationspunkte, um für diese eine allgemeine Übersicht zu gewinnen.

4.3.5 Softwarefamilienmodellierung nach Robak

Im Rahmen ihrer Habilitation skizziert Silva Robak [Robak01] ein Konzept zur Darstellung von Variabilität in Softwarefamilien mit Hilfe der Standarderweiterungen von UML. Aufbauend auf der aus FODA übernommenen Notation der Merkmalsmodelle (*feature models*) kombiniert sie Stereotypen und TaggedValues, um die Nachteile der bisher bekannten Notationen zu umgehen.

Dazu führt sie den neuen Stereotyp «variable» ein, mit dem alle varianten Elemente gekennzeichnet werden. Jedes dieser Elemente enthält einen TaggedValue `feature`, dessen Wert auf das zugeordnete Merkmal verweist.

Diese Notation wurde für Komponenten-, Klassen- und Aktivitätsdiagramme untersucht. Bei der Darstellung in Aktivitäts-Diagrammen wird die Semantik der Verzweigung geändert, um den Programmfluß in Abhängigkeit von der Existenz beziehungsweise Implementation der Teilzweige zu beeinflussen.

Eine weitergehende Verfeinerung dieses Ansatzes fand bisher nicht statt.

Kapitel 5

UML — Erweiterungsmechanismen

Die UML als weit verbreitete Modellierungssprache stellt Notationen zur Verfügung, um objektorientierte Softwaresysteme zu beschreiben. Bisher sind diese nur auf die Modellierung von einzelnen Produkten ausgelegt.

Derzeit aktueller Stand ist die Version 1.3 [OMG00], wobei eine überarbeitete Fassung — Version 1.4 [OMG01a] — bereits als Entwurf (*draft*) erhältlich ist. Die Ausführungen dieser Arbeit beziehen sich generell auf die UML-Version 1.4, wobei teilweise auf Unterschiede zur Version 1.3 eingegangen wird.

In der Spezifikation erfolgt die Beschreibung der Struktur der UML-Modellelemente mit Hilfe eines Metamodells, das selbst in UML notiert wird. Dabei sind auch Erweiterungsmechanismen integriert, die eine Erweiterung ohne Änderung des Metamodells ermöglichen. Die gesamte Spezifikation baut auf einer Metamodellarchitektur auf, die 4 Ebenen unterscheidet: Meta-Metaebene (M3), Metamodell (M2), Modellebene (M1) und Objektebene (M0).

In der UML-Spezifikation Version 1.4 sind drei Erweiterungsmechanismen spezifiziert, die benutzerdefinierte Erweiterungen erlauben. Das sind die Stereotypen, Schlüsselwort-Wert-Paare (*tagged values*, TaggedValues) und Einschränkungen (*constraints*). Entsprechend der abstrakten Syntaxdefinition von UML sind diese Erweiterungen im Metamodell definiert und befinden sich im Package „*Extension Mechanisms*“, das selbst dem Package „*Foundation*“ (Basismechanismen, Grundlagenpaket) untergeordnet ist. Von den Erweiterungsmechanismen werden das *Core*-Package für die Modellelemente, an denen die Erweiterungen ansetzen, und die Datentypen importiert. Jeder Erweiterungsmechanismus wird durch eine eigene Metaklasse repräsentiert, die direkt oder indirekt von der grundlegenden Metaklasse `ModelElement` abgeleitet ist.

Das Metamodell der Erweiterungsmechanismen ist in Abbildung 5.1 dargestellt. Gegenüber der Darstellung in [OMG01a] enthält dieses Diagramm zusätzlich die Attribute der im Core-Package definierten Metaklassen und die Generalisierungsbeziehung von `GeneralizableElement` zu `ModelElement`.

In diesem Zusammenhang ist auf eine unterschiedliche Verwendung des Begriffes „*Feature*“ hinzuweisen: in UML werden damit Attribute und Operationen von Klassen beziehungsweise Instanzen bezeichnet. Um Mißverständnisse in dieser Arbeit zu vermeiden, wird *Feature* beziehungsweise Merkmal nur im Zusammenhang mit Domänenmodellierung verwendet.

5.1 Stereotypen

Stereotypen bilden die Basis für systematische Erweiterungen der UML. Jedem Modellelement in UML können dazu Stereotypen zugewiesen werden, die jeweils durch einen Namen und ein optionales gra-

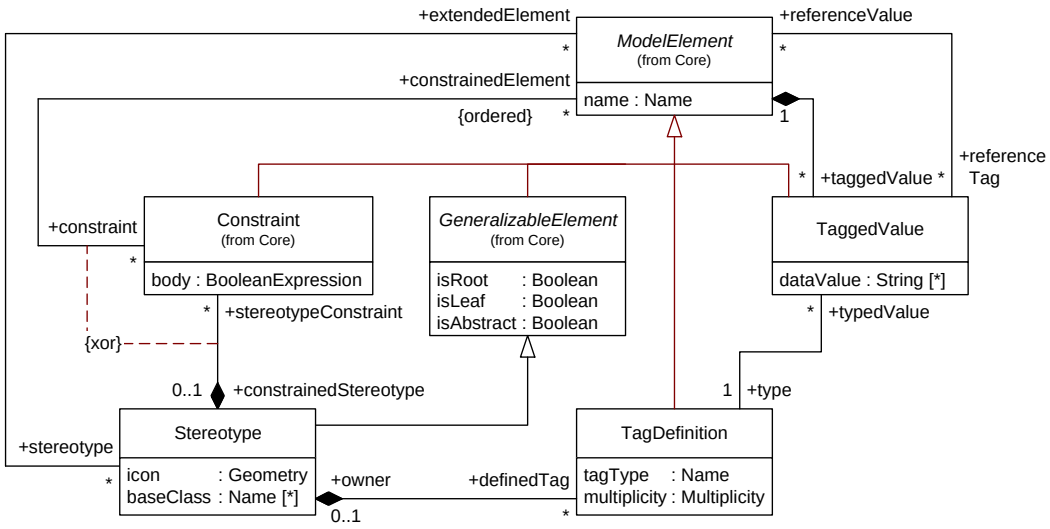


Abbildung 5.1: Metamodell der Erweiterungsmechanismen (UML 1.4)

fisches Symbol (Icon) gekennzeichnet werden. Diese stellen eine Klassifizierung des stereotypisierten Modellelementes dar, das dadurch gegenüber der Klassenhierarchie im Metamodell weiter spezialisiert wird. Praktisch wird dadurch das Metamodell erweitert, indem neue Metaklassen durch Spezialisierung mittels Stereotypen von vorhandenen Metaklassen „abgeleitet“ werden. Gegenüber Version 1.3 ist es in UML 1.4 möglich, einem Modellelement mehr als einen Stereotyp zuzuweisen, in diesem Fall entfällt die Darstellung der Stereotyp-Icons.

Das Attribut `baseClass` von `Stereotype` bezeichnet die Modellelemente, auf die dieser Stereotyp angewendet werden darf. Der Wert ist der Name einer Metaklasse der UML, die das stereotypisierbare Modellelement beschreibt. Bei entsprechender Festlegung der Basisklasse wird dadurch verhindert, daß zum Beispiel mit einem Stereotyp für Klassen eine Relation stereotypisiert werden kann. Neu in Version 1.4 ist die Möglichkeit, mehrere Basisklassen für einen Stereotyp anzugeben — damit kann der Anwendungsbereich des Stereotypen exakt festgelegt werden. Ein Stereotyp kann ebenfalls auf Subklassen der Basisklasse angewendet werden [OMG01a, Seite 2-82].

Da `Stereotype` im Metamodell eine Subklasse von `GeneralizableElement` ist, können Stereotypen in einer Vererbungshierarchie angeordnet werden. Dabei erbt ein Stereotyp von seinen Generalisierungen alle Constraints und TaggedValues und muß für die gleiche Art der Basisklasse spezifiziert sein [OMG01a, Seiten 2-83, 2-87].

Wird für einen Stereotyp ein grafisches Symbol definiert, können weitere Darstellungsformen für stereotypisierte Modellelemente unterschieden werden: in der Detaildarstellung des jeweiligen Modellelementes mit wahlweise nur textueller Darstellung, textueller und symbolischer Darstellung oder nur symbolischer Darstellung des Stereotyps und alternativ eine „minimierte“ Form, in der nur das Symbol mit dem Namen des jeweiligen Modellelementes dargestellt wird.

Wie aus dem Metamodell ersichtlich ist, können einem Stereotyp Constraints zugeordnet werden; diese gelten dann für alle Modellelemente, die mit diesem Stereotyp gekennzeichnet sind.

Einem Stereotyp können darüber hinaus auch Schlüsselwort-/Wert-Paare zugeordnet werden, die dann in jeder Instanz eines entsprechend stereotypisierten Modellelementes vorhanden sind. Diese werden bei der Spezialisierung von Stereotypen ebenso wie Constraints weitervererbt.

5.2 Constraints

Einschränkungen (*constraints*, auch Zusicherungen) erlauben den Ausdruck von Bedingungen, die durch die assoziierten Elemente erfüllt sein müssen. Dazu können frei wählbare Ausdrucksmittel benutzt werden, wie zum Beispiel in natürlicher Sprache, mathematischen Ausdrücken oder in einer herkömmlichen Programmiersprache. Die UML bevorzugt dafür die *Object Constraint Language* (OCL), mit der sich Invarianten und Vor-/Nachbedingungen (*invariants*, *preconditions* und *postconditions*) auf Objekten effizient ausdrücken lassen.

Die Evaluierung der Einschränkungen eines bestimmten Modellelementes erfolgt stets auf den Instanzen des jeweiligen Modellelementes. Für jedes Modellelement (auf M1-Ebene) können dazu beliebig viele Constraints definiert werden. Die Kontrolle auf Einhaltung der Constraints erfolgt nicht durch UML, diese müssen durch eine entsprechende Werkzeugunterstützung interpretiert werden.

Wie bei dem Stereotypen bereits angesprochen, können Constraints alternativ auch einem Stereotyp zugeordnet werden. In diesem Fall gilt die Einschränkung für alle Modellelemente, die mit dem Stereotyp gekennzeichnet sind. Diese Zuordnung wird besonders für die Spezifikation von Erweiterungen verwendet, wobei die Constraints für eine virtuelle M2-Ebene (auf der die Erweiterung beschrieben wird) spezifiziert und auf M1 berechnet werden — sie bilden damit für eine Erweiterung das Äquivalent zu den *Well-formedness Rules* in der UML-Spezifikation.

5.3 Schlüsselwort-Wert-Paare

Den dritten Erweiterungsmechanismus stellen Schlüsselwort-Wert-Paare dar (*tagged values*, TaggedValues). Dem Schlüsselwort, ein eindeutiger Name innerhalb des Elementes, wird dabei ein beliebiger Wert zugeordnet. Seit Version 1.4 ist es möglich, einem Schlüsselwort eine Liste von Werten zuzuordnen.

Mit Schlüsselwort-Wert-Paaren können gewissermaßen den Metaklassen virtuelle Attribute hinzugefügt werden, die deren Ausdrucksmöglichkeiten erweitern. Die Werte werden jedoch ebenso wie Constraints nicht durch UML ausgewertet und bedürfen daher einer entsprechenden Werkzeugunterstützung.

Durch das Metamodell werden Instanzen von `TaggedValue` eindeutig einem Modellelement zugeordnet. Gegenüber Version 1.3, in der die Zuordnung von TaggedValues zu Modellelementen oder Stereotypen durch das Metamodell nicht besonders gut beschrieben wurde, favorisiert Version 1.4 die Definition von TaggedValues über Stereotypen [OMG01a, Seite 2-88], wobei auch der Typ jedes Wertes für ein Schlüsselwort spezifiziert wird (Metaklasse `TagDefinition`). Wird eine Tag-Definition mit einem Stereotyp assoziiert, ist der zugehörige TaggedValue in allen damit stereotypisierten Modellelementen vorhanden. Diese Zuordnung wird von UML für die Spezifikation von Erweiterungen präferiert, da auf diese Weise stets eine TagDefinition für einen TaggedValue existiert. Die Werte von Schlüsselwörtern können nicht direkt mit Constraints eingeschränkt werden, dies ist indirekt über den Stereotypen beziehungsweise das Modellelement realisierbar.

5.4 Spezifikation von Erweiterungen

Die Spezifikation von Erweiterungen in sogenannten *Profiles* ist teilweise in UML beschrieben. Die Definition von Profilen wird durch die UML 1.4 besser unterstützt als in Version 1.3, insbesondere durch das überarbeitete Metamodell im Paket „*Extension Mechanisms*“. Im *Notation Guide* der UML [OMG01a, Kapitel 3] finden sich zusätzliche Richtlinien zur Darstellung und Spezifikation von Erweiterungen.

Die Notation von Erweiterungen erfolgt in UML, durch Instanziierung der Metabene (M2). Dadurch wird jedoch das Metamodell der Erweiterung beschrieben, wofür der Begriff „virtuelles Metamodell“ (VMM) sehr zutreffend ist. Dieses virtuelle Metamodell wird dann parallel zum UML-Metamodell zur Beschreibung von M1-Modellen (Anwendung von Stereotypen auf Modellelemente) instanziiert.

Stereotypen bilden in der Regel das zentrale Element eines Profils. Mit diesem werden TaggedValues verknüpft, die zusätzliche Eigenschaften des stereotypisierten Modellelementes beschreiben. Zusätzlich können Constraints mit Stereotypen assoziiert werden, derartige Constraints beziehen ihre Typen aus der Metaebene (M2) und werden auf dem Modell berechnet. Damit werden zusätzliche Einschränkungen der Erweiterung beschrieben, die über die semantischen Regeln der UML hinausgehen. Derartige Constraints sind damit das Äquivalent der *Well-formedness Rules* der UML-Spezifikation für das Profil.

Zur Gruppierung einer Erweiterung werden die Mechanismen des Model Managements verwendet, konkret müssen die Stereotypen jeder Erweiterung in einem Paket definiert sein, welches direkt oder indirekt den Stereotyp «profile» enthält [OMG01a, Seite 2-86].

Zur Abstraktion gemeinsamer Eigenschaften von Stereotypen einer solchen Erweiterung ist es möglich, die Generalisierbarkeit von Stereotypen zu nutzen, dabei können ebenfalls abstrakte Stereotypen spezifiziert werden, die nicht direkt auf Modellelemente angewendet werden können.

Durch mehrere Beispiele in der UML-Spezifikation wird die Möglichkeit verdeutlicht, für den Typ von TaggedValues eine benutzerdefinierte Aufzählung (Metaklasse `Enumeration`) zu verwenden [OMG01a, Seite 2-88]. Damit läßt sich auf effiziente und elegante Weise der Wertebereich eines Tags auf eine Auswahl einschränken, obwohl die Definition benutzerdefinierter Aufzählungen als Element einer Erweiterung nicht explizit erwähnt wurde.

Kapitel 6

Modellierung von Variabilität

> Variabilität ist das, was die Mitglieder einer Menge (zum Beispiel einer Produktfamilie) voneinander unterscheidet [Coriat00].

Ausgehend von dieser Definition stellt Variabilität den allgemeinsten Begriff dar, der veränderliche Softwareelemente beschreibt. Bei der konkreten Anwendung von Variabilitätsmodellierung – zum Beispiel Softwarefamilien oder Produktlinien – geht es darum, die Gemeinsamkeiten und Unterschiede (*commonalities and variabilities*) von Software zu erfassen. Die gemeinsamen Elemente sind dabei die Bestandteile, die sich in den verschiedenen Ausprägungen nicht verändern und dadurch mit den bestehenden Möglichkeiten der UML modelliert werden können. Demzufolge geht es in der weiteren Arbeit darum, eine Notation für die variablen Elemente von Softwaresystemen zu entwickeln.

Da Variabilität sich in dieser allgemeinen Form nur sehr schwer beschreiben läßt, gibt es bereits Ansätze, die eine modellierende Darstellung ermöglichen.

6.1 Beschreibung von Variabilität

Um Variabilität eindeutig zu beschreiben, unterschiedet man in erster Linie drei Kriterien:

1. die Beschreibung dessen, was sich verändert und welche Varianten möglich sind
2. die Lokalisierung der Variabilität, das heißt, wo die Variabilität auftritt
3. die Bedingungen, die an die Auswahl von variablen Elementen geknüpft sind

Diese und weitere Eigenschaften von Variabilität werden in den folgenden Abschnitten untersucht.

6.1.1 Variabilität und Merkmale

Um die Variationspunkte, an denen die Variabilität auftritt, zusammenfassend behandeln zu können, ist die Zuordnung zu einer abstrahierenden Ebene sinnvoll, die einen besseren Überblick und eine selektive Auswahl erlaubt.

Als Abstraktionsebene bieten sich die ersten Modelle des Softwareprozesses an, die ein sehr allgemeines Bild des zukünftigen Systems darstellen. Die Modellelemente, um die es dabei geht, sind die Anforderungen (*requirements*), Merkmale (*features*) und eventuell einzelne Use Cases. Letztere werden jedoch bereits schon zur detaillierteren Beschreibung von Requirements verwendet und sind damit noch weiter abstrahierbar. Merkmale bieten gegenüber den Anforderungen den Vorteil, auch Implementierungsdetails beschreiben zu können und werden in der Regel mit einem eindeutigen und treffenden

Bezeichner versehen, der eine platzsparende und formale Beschreibung ermöglicht. Des weiteren stellen Merkmale ein anerkanntes Modellelement dar, mit dem sich Variabilitäten modellieren lassen.

Ein Merkmal kann in unterschiedlichen Modellierungsebenen realisiert werden und sich in verschiedenen Elementen einer Systemarchitektur wiederfinden. Umgekehrt kann ein Asset (beziehungsweise ein Element im Systemsdesign) mehrere Merkmale implementieren (beziehungsweise mit anderen Assets wechselwirken, die ihrerseits an andere Merkmale gekoppelt sind) — es besteht eine N:M-Relation zwischen Merkmal und Designelement. Zur Reduktion der Komplexität wird im Idealfall eine Komponente nur an ein (oftmals funktionales) Merkmal gekoppelt und Abhängigkeiten (wenn möglich) bereits auf Merkmalsebene modelliert werden. Die Assets (respektive Designelemente) einer Systemarchitektur können wiederum Variabilitäten enthalten, die mit Merkmalen assoziiert werden.

Um die Art und Weise der durch die Merkmale ausgedrückten Variabilität zu erfassen, werden im folgendem Abschnitt die Möglichkeiten der Merkmalsmodellierung detaillierter untersucht.

6.1.2 Merkmalsmodellierung

6.1.2.1 Typen von Merkmalen

Die Merkmale werden, wie im Kapitel 4 gezeigt, in einer Baumstruktur modelliert. Die Baumstruktur reflektiert eine hierarchische Gliederung der Merkmale, wobei jeder Knoten ein Merkmal (*feature*) eines bestimmten Typs repräsentiert. Nach [Griss00a] setzt sich dabei ein Merkmal aus den Sub-Knoten zusammen (*composed-of*-Relation) und abstrahiert von diesen. Die Blätter des Merkmalsbaumes stellen meistens konkrete technische Anforderungen dar, die durch das System realisiert werden oder seine Eigenschaften charakterisieren (zum Beispiel unterstützte Protokolle).

Für die Merkmalsauswahl gilt eine prinzipielle Regel: für das Vorhandensein eines Merkmals muß immer auch das übergeordnete Merkmal vorhanden sein – wird ein Merkmal bei der Instanziierung nicht ausgewählt, entfällt der gesamte untergeordnete Teilbaum des Merkmalsdiagramms. Für die Klassifizierung eines Knotens werden die folgenden Typen unterschieden:

- **Notwendig** (*Mandatory*): Das Merkmal wird in jedem Fall eingebunden.
- **Optional**: Ein solches Merkmal ist eine unabhängige Erweiterung und kann eingebunden werden oder nicht.
- **Alternativ**: Dieses Merkmal ersetzt ein anderes, wenn es eingebunden wird. Aus einer Menge von zusammengefaßten alternativen Merkmalen kann jeweils nur eines verwendet werden. Ein solches Merkmal wird teilweise auch als Variante (*variant*) bezeichnet. Der übergeordnete Knoten, der das Auswahlkriterium repräsentiert, wird teilweise auch als Variationspunkt (*variation point*) bezeichnet.
- **Extern**: Solche Merkmale sind nicht direkt Bestandteil des modellierten Systems, sondern der Zielplattform, auf der das System eingesetzt wird. Da das modellierte System auf ihnen aufbaut und von ihnen abhängt, sind sie für die Modellierung notwendig, um Kontextinformationen darstellen zu können. Ein Beispiel für externe Merkmale sind Funktionen des Betriebssystems oder der Programmiersprache, mit denen Verbindungen per TCP aufgebaut werden können.

Die externen Merkmale sind in den in Kapitel 3 und 4 aufgeführten Werken nicht vorhanden, der Typ wurde aufgrund von Erfahrungen und Fallstudien von Bosch, Svahnberg und Gurp ([Svahnbrg]) eingeführt. Mit externen Merkmalen lassen sich Kontextbedingungen ausdrücken, die sonst nur an unpassender Stelle (zum Beispiel Implementierungsdetails in den Requirements) modelliert werden können. Insbesondere können damit plattformspezifische Fähigkeiten ausgedrückt und mit den entsprechenden Anforderungen an das Softwaresystem verknüpft werden.

In [Czarnecki00] wurde ein weiterer Merkmalstyp unterschieden, die Oder-Merkmale (*or-features*). Dabei handelt es sich um eine Form der Alternative, in der mindestens eines der untergeordneten Merkmale ausgewählt werden muß. Dabei ist die Auswahl von mehr als einem Merkmal möglich. [Griss98] differenziert ebenfalls alternative Merkmale weiter und macht dazu den Merkmalstyp abhängig vom Bindezeitpunkt — die Oder-Merkmalsverknüpfung entspricht dabei der Bindung zur Laufzeit.

6.1.2.2 Zusätzliche Abhängigkeiten zwischen Merkmalen

Um Abhängigkeiten zwischen Merkmalen modellieren zu können, die nicht durch die Baumstruktur dargestellt werden, werden zusätzliche optionale Einschränkungen (*constraints*) genutzt:

- **Abhängigkeit** (*requires*): Mit dieser Einschränkung werden Abhängigkeiten zwischen Merkmalen modelliert, die nicht durch die Baumstruktur dargestellt sind. Ergänzend zur gegenseitigen Abhängigkeit ist auch eine gerichtete Verbindung (einseitige Einschränkung) vorstellbar.
- **Wechselseitiger Ausschluß** (*Mutually Exclusive*): die damit verbundenen Merkmale schließen sich gegenseitig aus – beide können nicht gleichzeitig ausgewählt werden.
- **Konflikte** (*conflicting*): in [Bosch00] wurde eine weitere Abhängigkeit identifiziert, die eine abgeschwächte Version des wechselseitigen Ausschlusses darstellt und sich in erster Linie auf Qualitätsattribute bezieht. Diese Beziehung ist als expliziter Hinweis an den Nutzer des Merkmalsmodells (den Softwarearchitekten) zu verstehen, daß an dieser Stelle weiterer Handlungsbedarf besteht. Kann der Konflikt nicht aufgelöst werden, kann diese Konfliktbeziehung auch als wechselseitiger Ausschluß behandelt werden.

Die Konflikt-Einschränkungen stellen einen Vertreter der sogenannten *Weak constraints* — schwachen Einschränkungen — dar, die verschiedene mögliche Beziehungen zwischen Softwarebausteinen beschreiben. Deren konkretes Auftreten ist in der Regel vom konkretem Einsatzkontext, zum Beispiel dem verwendeten Betriebssystem, abhängig.

6.1.2.3 Weitere Eigenschaften

Zur Beschreibung von Merkmalen werden im Rahmen der Merkmalsmodellierung weitere Eigenschaften spezifiziert, die für die Modellierung von Variabilität relevant sind:

- Verweise auf untergeordnete und andere Modelle zur semantischen Beschreibung (analog zu Use Cases und Aktivitätsdiagrammen)
- Kategorisierung von Merkmalen, um eine Klassifizierung und eine schnelle Zuordnung der Zuständigkeit zu erreichen
- die Entscheidungsbegründung und Hinweise zur Entscheidungsfindung
- Beteiligte, die an dem jeweiligen Merkmal interessiert sind (*stakeholder*) beziehungsweise Klienten von Komponenten, die auf das Merkmal angewiesen sind
- Beispielsysteme, anhand derer das Merkmal extrahiert wurde
- Informationen zur Verfügbarkeit und Einschränkungen des Kreises, dem die Festlegung erlaubt ist. [Czarnecki00] definiert dazu das Konzept des Bereiches (*site*), der „Wann?“, „Wo?“ und „Wer?“ einer Domäne definiert. Diese Verfügbarkeitsbereiche können dann für die Verfügbarkeit (*availability site*) und das Binden (*binding site*, *binding mode*) eines Merkmales festgelegt werden. Dies erlaubt eine feine Differenzierung des Zugriffes auf und der Auswahl von Merkmalen.
- für Alternativen ist die Unterscheidung möglich, ob noch zusätzliche — zum Zeitpunkt nicht modellierte — Alternativen erlaubt beziehungsweise möglich sind.

6.1.2.4 Kategorisierung von Merkmalen

Die Unterteilung von Merkmalen erfolgt nach [Kang90, Seite 38] hauptsächlich in:

- *Operating environments*
- *Capabilities*; können unterteilt werden in:
  - *Functional features*: angebotene Dienste
  - *Operational features*: bezogen auf die Arbeit der Anwendung
  - *Presentation features*: Wie und Welche Informationen präsentiert werden?
- *Domain technology*
- *Implementation techniques*

Diese Unterteilung kann für spezifische Zwecke erweitert oder angepaßt werden. Diese Kategorisierung nach FODA wird auch in späteren Ansätzen, wie FeatuRSEB [Griss98] oder FORM [Kang98] verwendet.

6.1.3 Variationspunkte

Der Begriff Variationspunkte findet sich sowohl in der Merkmalsmodellierung — in erster Linie für alternative Merkmale — als auch in der Modellierung von konkreten Softwaresystemen. Da die Merkmalsmodellierung bereits entsprechende Beschreibungsmittel bietet, wird der Begriff Variationspunkt in der weiteren Arbeit nicht in Merkmalsmodellen verwendet, kann dafür in allen anderen Modellarten auftreten.

Prinzipiell lassen sich Variationspunkte als „m aus n“ ($0 \leq m \leq n$) betrachten und stellen damit die Entsprechung zu optionalen und alternativen Merkmalen dar. Der Unterschied besteht darin, inwieweit die Wahl von m weiter eingeschränkt wird — zum Beispiel auf $m=1$ für sich gegenseitig ausschließende Alternativen. Dabei ist jedoch zu beachten, das in Abhängigkeit von der spezifizierten Bindezeit und den verwendbaren Technologien eine abweichende Umsetzung erforderlich sein kann: wird zum Beispiel ein Merkmal zur Laufzeit festgelegt, müssen alle benötigten Alternativen als ausführbarer Code vorhanden sein.

Die in [Czarnecki00] verwendeten *Or-features* stellen eine abgeschwächte Form der Alternative dar, in der mehr als eine Variante gleichzeitig ausgewählt werden kann. Daraus abgeleitet wurden die folgenden Typen von Variationspunkten spezifiziert:

- Alternative: „1 aus n“
- Alternative aus Optionen: „0 oder 1 aus n“
- Erweiterung: „0..1 aus n“ oder „1..n aus n“
- Erweiterung mit optionalen Varianten: „0 bis n aus n“
- Erweiterung mit *Or-features*: „1 bis n aus n“

Diese Möglichkeiten belegen die unterschiedlichen Auswahlvarianten von Variationspunkten, die in der Modellierung eingesetzt werden können. Die wahrscheinlich am häufigsten verwendete Möglichkeit von Selektionsregeln für Varianten — 1 aus n — läßt sich noch unterscheiden in „genau eine“ und „eine oder keine“. Daneben sind jedoch auch mehrfache Alternativen denkbar, bei denen mehr als eine Variante ausgewählt werden kann.

6.1.4 Festlegung von Variabilität

Über die Betrachtung im Rahmen der Merkmalsmodellierung hinaus kann die Variabilität in Softwaresystemen durch weitere Kriterien qualifiziert werden. In der Regel werden damit Variationspunkte beschrieben, an denen die unterschiedlichen Ausprägungen der Systeme sichtbar werden, die einzelnen Eigenschaften beziehen sich jedoch auf Variabilität im allgemeinem und können folglich auch für andere Beschreibungsmittel genutzt werden.

6.1.4.1 Zeitpunkt der Festlegung

Eine der bedeutendsten Eigenschaften von Variabilität ist, wann die konkret zu erfüllende Eigenschaft festgelegt wird (Bindezeitpunkt, *binding time*). Abhängig von der Rolle der modellierenden Person kann dieser Zeitpunkt aus verschiedenen Sichten betrachtet werden.

Die in den verschiedenen Literaturquellen verwendeten Bindezeiten sind in der nachfolgenden Tabelle zusammengefaßt:

<table>
<tr><th>SPLIT
[Coriat00]</th><th>Catalysis
[Souza99]</th><th>nach IESE
[Anastaso]</th><th>KobrA
[Atkinson00]</th><th>FeatuRSEB
[Griss98]</th><th>FODA
[Kang90]</th></tr>
<tr><td>derivation</td><td>coding</td><td></td><td>implementation</td><td>reuse</td><td></td></tr>
<tr><td rowspan="2"></td><td>compile</td><td>compile</td><td rowspan="2">build</td><td rowspan="4"></td><td rowspan="2">compile</td></tr>
<tr><td>link</td><td>link</td></tr>
<tr><td>configuration</td><td rowspan="2"></td><td rowspan="2"></td><td>installation</td><td rowspan="2">load</td></tr>
<tr><td></td><td>start-up</td></tr>
<tr><td rowspan="3">runtime</td><td>dynamic</td><td rowspan="3">runtime</td><td rowspan="3">runtime</td><td rowspan="3">runtime</td><td rowspan="3">runtime</td></tr>
<tr><td>runtime</td></tr>
<tr><td>reflective</td></tr>
<tr><td></td><td></td><td>update[1]</td><td></td><td></td><td></td></tr>
</table>

Dabei treten sehr unterschiedliche Sichtweisen auf: zum Beispiel in Catalysis die verschiedenen Binde-Techniken, in KobrA dagegen die Orientierung an den Vorgehensphasen bei der Realisierung eines Softwaresystems. Die verschiedenen Begriffe sind bei gleicher Bedeutung in der gleichen Tabellenzeile notiert. Da sich nicht alle Zeiten 1:1 abbilden lassen, verdeutlicht die Zellenunterteilung die Abgrenzung zwischen den Bedeutungen. Nicht vorhandene Zeiten sind entsprechend leer geblieben. Es lassen sich natürlich — wie in [Cleaveland01] — noch ausführlichere Unterteilungen treffen, die zum Beispiel das Vorgehensmodell bei der Entwicklung einbeziehen.

Für die Modellierung von Variabilität sind die technologieorientierten Bindezeiten weniger geeignet. Eine sehr geeignete Unterteilung scheint die nach KobrA zu sein: Es wird jeder wichtige Entwicklungsstand berücksichtigt, feingranulare Unterteilungen zum Beispiel bei der Übersetzung oder zur Laufzeit jedoch werden zusammengefaßt. Aus diesem Grund soll für die Bindezeiten der noch zu entwickelnden Erweiterungen eine Vorauswahl getroffen werden.

Für den Abstraktionsgrad der Merkmalsmodellierung ist die Unterteilung nach SPLIT in *Derivation*, *Configuration* und *Runtime* sehr gut geeignet. Damit werden die Einsatzphasen „Entwicklung“, „Installation“ und „Anwendung“ gut abgebildet:

- *development time* beschreibt den Zeitraum von der Entwicklung eines konkreten Systems bis zur Abschluß der Übersetzung des ausführbaren Entwicklungsziels. Dazu gehört beispielsweise die Ableitung eines Produkts aus einer Produktlinie oder die Generierung von Komponenten.
- *installation time* beschreibt den Zeitraum nach der Übersetzung bis zur Herstellung der Einsatzfähigkeit, in der die benötigten Komponenten auf den jeweiligen Systemknoten (beispielsweise Client, Server, Datenbanken) installiert und konfiguriert werden.

[1] auch *post-runtime*

- *runtime* beschreibt den Zeitraum ab dem Abschluß der Installation, in dem das Programm von den Anwendern ausgeführt wird.

Werden einzelne Programmteile nachträglich (**update time**) installiert, kann dieser Vorgang je nach Ausführungszustand des Systemes der Installation oder zur Laufzeit zugeordnet werden.

Für die Entwicklung des Systems ist eine feinere Unterteilung sinnvoll, die eine Unterscheidung der einzelnen Entwicklungs- und Installationsphasen ermöglicht. Dazu soll folgender Vorschlag verwendet werden:

- *development time* beschreibt den Zeitraum der Entwicklung eines konkreten Systems, zum Beispiel die Ableitung aus einer generischen Beschreibung.
- *build time* stellt den Zeitpunkt der Übersetzung und des Linkens zu ausführbaren Programmen dar. Bei Bedarf kann dieser zusätzlich in *compile-* und *link time* unterschieden werden. Hierbei werden zum Beispiel in den Code eingebettete Anweisungen (Compilerdirektiven) zur Auflösung von Variabilität ausgeführt.
- *installation time* beschreibt den Zeitraum der Installation und Konfiguration eines Softwaresystems.
- *start-up time* beschreibt den Zeitpunkt der Initialisierung eines Softwaresystems, in dem zum Beispiel Konfigurationseinstellungen umgesetzt werden.
- *runtime* ist der Zeitraum der Ausführung eines Softwaresystems.

Beide Unterteilungen können natürlich im praktischem Einsatz den jeweiligen Bedürfnissen angepaßt werden. So sind sie als Konsens zu verstehen, der eine einheitliche Bezeichnung der verschiedenen Zeitpunkte gewährleisten soll.

6.1.4.2 Art der Festlegung

In Ergänzung zum vorhergehenden Abschnitt läßt sich die Merkmalsfestlegung nicht nur nach dem Zeitpunkt, sondern auch nach der Art der Bindung (*binding mode*; nach [Czarnecki00]) unterscheiden:

- statisch: eine solche Bindung kann nicht geändert werden (ohne das gesamte System neu zu erstellen).
- changeable: diese Bindungen werden nur bei Bedarf entfernt und neu zugeordnet. Diese Bindungsart kann als optimierte Form des dynamischen Bindens betrachtet werden.
- dynamisch: die Bindung wird nach jedem Gebrauch wieder entfernt. Dies ist sinnvoll, wenn sich die Zuordnung der Bindungen mit hoher Frequenz ändert.

Die Spezifikation des Bindemodus kann bei Bedarf verwendet werden, um eine Selektionsentscheidung genauer zu charakterisieren. Welche konkreten Kombinationen von Zeitpunkt und Modus dabei möglich sind, wird von den verwendeten Technologien und der verwendeten Kategorisierung des Zeitpunktes abhängig sein und soll hier nicht weiter untersucht werden.

6.1.4.3 Zustände im Entscheidungsprozeß

Unabhängig davon, ob Entscheidungen zur Auswahl von variablen Softwarebausteinen explizit modelliert werden oder nicht, lassen sich im Zuge der Entwicklung des Softwaresystems für jeden Variationspunkt bestimmte Zustände unterscheiden:

- Implizit — ein Variationspunkt ist, einmal eingeführt, implizit auf allen übergeordneten Modellierungsebenen vorhanden beziehungsweise wird in einem übergeordneten Modellelement aggregiert.

- Gestaltet (*designed*) — der Variationspunkt wurde explizit dargestellt (im aktuellem Modell neu eingeführt).

- Gebunden — die Variabilität wurde festgelegt, indem zum Beispiel Merkmale ausgewählt oder Variationspunkten eine konkrete Variante zugeordnet wurde.

6.1.5 Ebenen der Variabilität

Die Unterteilung der Abstraktionsebenen, auf denen Variabilität im Rahmen des Softwareentwicklungsprozesses auftritt, ist abhängig vom Standpunkt der Betrachtung. In [Svahnbrg00] findet sich die folgende Ebenenaufteilung, die sich gut für die Differenzierung der Modellierung eignet:

- Produktlinienebene (*product line level*): Beschreibung, wie sich verschiedene Produkte unterscheiden

- Produktebene (*product level*): die Variabilität beschäftigt sich mit der Architektur und der Bausteinauswahl für ein einzelnes Produkt

- Komponentenebene (*component level*): wie werden neue Implementierungen von Komponentenschnittstellen integriert und wie entwickeln sich die Schnittstellen weiter

- Designebene (*sub-component level*): da die Komponenten durch verschiedene Softwareelemente (Beispiel: Klassen für Objekte) realisiert sind, werden auf dieser Ebene die Variabilität dieser Elemente modelliert

- Quellcodeebene (*code level*): auf dieser Ebene finden sich die größten Unterschiede zwischen einzelnen Produkten wieder

Aus der Sicht der Realisierung eines Softwaresystems nach [Anastaso] lassen sich abweichende Unterteilungen treffen, die die Implementierung der Variabilität stärker differenzieren:

- bei der Architekturbeschreibung des Systems,

- während des Systemdesigns,

- auf Quellcodeebene (bei der Implementierung),

- im kompiliertem Code,

- in der ausführbaren Anwendung

- und zur Laufzeit im ausgeführten Programm.

Eine konkrete Unterteilung wird abhängig sein von dem gewählten Entwicklungsprozeß (Vorgangsmodell) und durch die verwendeten Technologien (Programmiersprache, Plattform, ...) mitbestimmt.

6.1.6 Abhängigkeiten zwischen Bausteinen

Nicht nur bei der Merkmalsmodellierung, auch während der Analyse und dem Design der Elemente, aus denen sich eine Softwarearchitektur zusammensetzt, können Abhängigkeiten identifiziert werden. In einigen Fällen können diese über die Zuordnung zu Merkmalen aufgelöst werden, es wird jedoch ebenfalls vorkommen, Beziehungen bestehen, die auf diese Weise nicht modellierbar sind. Prinzipiell ist es anzustreben, solche Abhängigkeiten zwischen Softwarebausteinen zu minimieren.

Die Unterscheidung dieser Abhängigkeiten ist dieselbe wie für Beziehungen (*constraints*) zwischen Merkmalen.

6.1.7 Umsetzung der Variabilität

Zur Realisierung der Variabilität in den Softwarebausteinen werden in [Jacobson97], [Bosch00] und weiteren folgende Techniken genannt:

- Vererbung (*inheritance*),
- Erweiterung (*extensions*),
- Konfiguration (*configuration*),
- Parametrisierung (*templates*) und
- Generierung (*generation*).

Diese Techniken werden teilweise unterschiedlich benannt. Sie können weitgehend mit den normalen Ausdrucksmitteln der UML dargestellt werden.

6.1.8 Klassifizierung

Wie von Sharp [Sharp00a] dargestellt, läßt sich Variabilität als positiv und negativ klassifizieren. Positive Variabilität fügt Funktionalität hinzu, während negative Variabilität diese entfernt. Diese Unterscheidung ist insbesondere von den nutzbaren Vererbungsmechanismen abhängig, wie in [Ahmad00] dargestellt wird.

Davon ausgehend wird in [Anastaso] Variabilität auf Implementierungsebene wie folgt klassifiziert:

Variabilitätstyp	Bedeutung
Positiv	Funktionalität wird hinzugefügt
Negativ	Funktionen werden entfernt
Optional	Code wird eingefügt
Alternativ	Code wird ersetzt
Funktion	Funktionalität ändert sich
Plattform / Umgebung	Das Basissystem oder die Umgebung verändert sich.

In diesem Werk findet sich ebenfalls eine Betrachtung der zur Realisierung dieser Variabilitäten verfügbaren Technologien.

Diese Klassifikation von Variabilität hat jedoch auf die Modellierung in UML keinen Einfluß, sondern stellt Bewertungskriterien für die verfügbaren Techniken zur Umsetzung bereit.

6.2 Qualitätskriterien für eine Modellierungsnotation

In diesem Abschnitt sollen weitere Schwerpunkte angesprochen werden, die sich aus dem Literaturstudium und praktischen Erkenntnissen ergeben haben und bei der Entwicklung einer variablen Notation berücksichtigt werden sollten.

6.2.1 Verfolgbarkeit

Für die Nachvollziehbarkeit der Entwicklung eines Konzeptes – zum Beispiel den Weg eines Akteurs aus dem Use-Case-Modell über seine Entsprechung im Systemdesign bis zum implementierten Objekt – ist es erforderlich, zwischen diesen Elementen eine entsprechende Verbindung herzustellen. Im Zusammenhang mit Variabilität bekommt diese Fähigkeit der Verfolgbarkeit (*traceability*) eine bedeutende Rolle, da über sie die Zuordnung von Merkmalen zu System-Bestandteilen oder -eigenschaften erfolgen kann. Allgemeiner ausgedrückt kann damit der Entwicklungsweg eines bestimmten Bestandteiles verfolgt werden.

Traceability wird unterschieden nach Vorwärts- und Rückwärts-Verfolgbarkeit. Erstere dokumentiert die Entwicklung und den Gebrauch eines Konzeptes (zum Beispiel eine Systemeigenschaft oder einen Anwendungsfall), wogegen zweitere die Herkunft und die Evolution des Entwurfs aufzeigt.

Eine übliche Vorgehensweise zur Umsetzung ist die Darstellung durch Verbindungen (*links*) oder Matrizen, wobei eine explizite Verbindung zwischen den Modellierungsebenen (Architektur, Design, Implementierung) hergestellt wird. Eine entsprechende Integration in die grafische Modellierung ist zweckmäßig, um die Konsistenz der Systementwicklung zu gewährleisten.

Voraussetzung ist in jedem Fall, daß die zu verbindenden Artefakte identifiziert wurden — für Variabilität entspricht das der Identifikation des Auftretens und der möglichen Ausprägungen.

6.2.2 Skalierbarkeit

Ein weiterer Gesichtspunkt, der vor allem bei der längerfristigen Entwicklung von Produktlinien-orientierten Softwarefamilien zu berücksichtigen ist, ist die Skalierbarkeit. Damit ist speziell die Fähigkeit gemeint, bei einen evolutionären Wachsen des Gesamtsystems den Überblick zu behalten. Das geschieht zum Beispiel, wenn die Produktlinie durch neue Produkte vergrößert oder bestehende Produkte erweitert werden.

Auf Modellierungsebene kann dies zum Beispiel durch Werkzeuge unterstützt werden, die die Auftrennung komplexer Teile in kleinere Einheiten ermöglichen und entsprechende Fähigkeiten besitzen, um auch nicht-triviale Änderungen konsistent umzusetzen. Weitere Unterstützung bietet die Möglichkeit, unterschiedliche Detaillierungsstufen oder Abstraktionsebenen darstellen zu können — also zum einen abgestufte Betrachtungsmöglichkeiten für ein Modell, und zum anderen die Abstraktion von Modellen oder die Überblicksdarstellung von bestimmten Modellelementen (zum Beispiel Hervorhebung der Variationspunkte).

Eine zusätzliche Möglichkeit besteht darin, das Ausblenden von Teilen des Modells zu ermöglichen und damit speziell selektionsspezifische Eigenheiten darzustellen oder zu verstecken.

6.2.3 Separation of Concerns

Die Trennung der Funktionen (*separation of concerns*, SoC) verfolgt die klare Unterscheidung zwischen den einzelnen Funktionen, die der Software-Modellierung zugrunde liegen [Booch99]. Dazu werden diese zweckmäßigerweise möglichst explizit und deklarativ auf den Punkt gebracht und lokalisiert (zum Beispiel mit Hilfe der Modularisierung). In der Softwareentwicklung geht das einher mit der Entwicklung unabhängiger Teile oder Einheiten, die über festgelegte Schnittstellen in Verbindung treten. Die Modellierung solcher Funktionseinheiten muß entsprechend auch auf allen Modellierungsebenen unterstützt werden.

Damit wird die Verständlichkeit, Anpassungsfähigkeit und Wiederverwendung gefördert, da die Absicht und Lokalisierung der Funktion genau dargestellt und eine einfache Überprüfung der Erfüllung der Anforderungen ermöglicht wird [Czarnecki00]. In der Praxis wird diese Trennung meist durch Module oder Pakete (*packages*) [Souza99, Seite 298] erreicht, die das Gesamtsystem in überschaubare Einheiten zerlegen.

6.2.4 Probleme durch Evolution von Software

Dieser Abschnitt soll kurz einige Probleme bei der kontinuierlichen Weiterentwicklung von Softwarebausteinen aufzeigen, die sich beim praktischen Einsatz der Produktlinien-orientierten Softwareentwicklung gezeigt haben. In [Bosch99, Bosch98] findet sich die detaillierte Diskussion und Ursachenforschung, aus der an dieser Stelle nur die modellierungsrelevanten Aspekte extrahiert werden.

6.2.4.1 Konkurrierende Versionen

Es wurden vier Situationen erkannt, in denen mehrere Versionen eines Bausteins eingeführt wurden und parallel existieren:

- Bei konkurrierenden Qualitätsanforderungen: meistens werden Assets auf bestimmte Qualitätsattribute optimiert — da unterschiedliche Produkte unterschiedliche Anforderungen an gemeinsam genutzte Assets stellen, können diese nicht durch eine einzelne Version erfüllt werden.
- Wenn Variabilität durch Versionen implementiert wird — insbesondere da verschiedene Typen von Variabilität schwierig durch Konfigurations- oder Compiler-Einstellungen aufzulösen sind.
- Bei Unterschieden zwischen High-End und Low-End-Produkten — Low-End-Produkte enthalten bei einheitlichen Assets die zusätzliche (überflüssige) Funktionalität der High-End-Produkte.
- Durch unterschiedliche Anforderungen, jedoch gemeinsamer Nutzung durch die verschiedenen Geschäftsbereiche, die die Assets entwickeln.

6.2.4.2 Abhängigkeiten zwischen Assets

Da alle Assets Teil der PL-Architektur sind, treten Abhängigkeiten zwischen ihnen auf. Beim ersten Design werden diese Abhängigkeiten in der Regel überschaubar und explizit dokumentiert.

Während der Weiterentwicklung der Produktlinie treten jedoch neue Abhängigkeiten auf. Diese wären mit einer anderen Aufteilung der Architektur vermeidbar, ein solches Redesign wird jedoch nur selten durchgeführt. Typischerweise werden diese Abhängigkeiten relativ spät im Entwicklungsprozeß sichtbar, sind meistens implizit und werden nur geringfügig modelliert.

Als Ursachen für die Entwicklung zusätzlicher Abhängigkeiten wurden folgende identifiziert:

- Aufteilung von Komponenten bei wachsendem Umfang
- Funktionserweiterungen, die sich über mehrere Assets erstrecken
- Erweiterung der Funktionalität der Assets

Fazit: Diese durch die Weiterentwicklung von Software verursachten Probleme können durch eine angepaßte Organisation und das Vorgehensmodell der Entwicklung ansatzweise reduziert werden.

Die dazugehörige Unterstützung durch die Notation des Modells soll, soweit möglich, einbezogen werden. Dazu gehört insbesondere die Modellierung von zusätzlichen, durch Evolution entstandenen Abhängigkeiten. Die Versionsverwaltung und die Verwendung der Notation dagegen kann und soll nicht durch die Notation selbst reguliert werden: Dazu ist der verantwortungsvolle Umgang des Entwicklers mit den gegebenen Beschreibungsmöglichkeiten und der Einsatz spezieller Versionsmanagementsysteme geboten.

Kapitel 7

Modellierung von Variabilität in UML

In diesem Kapitel wird aufbauend auf den in Kapitel 6 analysierten Beschreibungsmöglichkeiten für Variabilität ein Notationsvorschlag für die Darstellung in UML entwickelt. Für die Notationskonzepte werden die Möglichkeiten für die Darstellung diskutiert und der Grundstein für eine standardkonforme Erweiterung der UML gelegt.

Die entwickelten Konzepte werden formal weitgehend spezifiziert und verwenden dazu die von der UML beschriebenen Notationsmöglichkeiten für Profile. Es handelt sich dabei jedoch nicht um eine vollständige Spezifikation; die Brauchbarkeit und eventuell notwendige Verfeinerungen oder Erweiterungen sollten im praktischem Einsatz verifiziert werden.

7.1 Das Konzept

Bei der Aufarbeitung der existierenden Literatur und Forschung sind unterschiedliche Abstraktionsebenen für die Behandlung von Variabilität in Softwaresystemen aufgefallen. Diese leiten sich aus völlig unterschiedlichen Ansätzen und Sichtweisen her und werden durch unterschiedliche Beschreibungsmittel dargestellt:

1. Die **Merkmalsmodellierung** als eine der höchsten Abstraktionsebenen beschäftigt sich in einem sehr frühen Stadium der Softwareentwicklung mit der Modellierung der Gemeinsamkeiten und Unterschiede zwischen den Produktinstanzen eines Bereiches, wie zum Beispiel einer Produktlinie, einer Softwarefamilie — kurzum, einer Domäne.

2. **Variationspunkte** als explizite Beschreibung eines Punktes, an dem Variabilität in Software auftritt und die durch die Auswahl von Varianten aufgelöst beziehungsweise gebunden wird. Diese Variationspunkte finden sich typischerweise auf unterschiedlichen Abstraktionsniveaus — von Paketen/Modulen über Komponenten bis zu Klassen — und sind in mehreren Entwicklungsphasen (zum Beispiel Analyse, Design, Test) vertreten.

3. Variabilität auf **Code-Level** wird dagegen durch verschiedene Techniken im Programmcode umgesetzt. Typische Vertreter dieser aus Modellierungssicht niedrigen Abstraktionsebene sind Compilerdirektiven; weiterführende Ansätze sind beispielsweise die Frames-Technik [Basset97] und JavaXML [Cleaveland01].

Für die Variabilität auf der Code-Ebene existieren bisher keine Ansätze, diese auch zu modellieren. Solche modellierenden Darstellungen sollten sich in erster Linie auch aus der bereits existierenden Beschreibung (dem Code) extrahieren lassen und sind somit eng an eine effektive Verknüpfung von Modellierungs- und Implementierungsprozeß (zum Beispiel generative Verfahren) gebunden.

Für die Modellierung von Variabilität, deren systematischer Erfassung und Ausnutzung mit dem Ziele einer Produktivitätssteigerung bei der Softwareentwicklung sind die höheren Abstraktionsebenen besser geeignet, die sich nur selten direkt auf Code (-Fragmente) abbilden lassen. Indizien dafür sind die komponentenorientierte Softwareentwicklung, bei der Komponenten als elementare Bausteine der Wiederverwendung fungieren, und die Zuordnung komplexer Softwarebausteine (zum Beispiel Frameworks für eine spezifische Domäne) zu einzelnen Merkmalen aus der Merkmalsmodellierung.

Bei dieser Sichtweise wird die Variabilität zuerst durch das Konzept der Merkmalsmodellierung beschrieben. Auf dieser abstrakten Beschreibung aufbauend erfolgt in der Regel die Entwicklung einer Architektur (für die Domäne), in der Assets modelliert und – bei Verwendung der Merkmalsmodellierung – den einzelnen Merkmalen zugeordnet werden, die sie realisieren. Diese Abstraktionsebene kann man ganz treffend mit „Modellierung im Großen“ beschreiben — die zugrundeliegenden Einzelbausteine sind selbst mehr oder weniger komplexe Systeme: zum Beispiel Subsysteme, Komponenten, Use-Cases und Testszenarien.

Für die Modellierung von Variabilität werden die anerkannten Konzepte der Merkmalsmodellierung und der Variationspunkte verwendet. Letztere werden mit einer Spezialform der Alternative, der Optionalität ergänzt. Diese erlaubt es, einzelne Elemente als optional und damit abhängig von einer Bedingung zu modellieren, ohne dazu auf die komplexere Beschreibung mittels Variationspunkten zurückgreifen zu müssen.

Diese Abstraktionen von Variabilität entsprechen nach Abschnitt 6.1.5 der Produktlinien-/Produktebene und der Komponentenebene. Auf der Ebene des Designs kann mit den zu entwickelnden Konzepten ebenfalls modelliert werden, Variabilität auf Code-Niveau wird dagegen in dieser Arbeit nicht weiter betrachtet.

7.1.1 Modellierung in UML

Die UML stellt bereits Ausdrucksmöglichkeiten für fast alle Abstraktionsebenen des Softwareentwicklungsprozesses mit definierter Semantik bereit. Diese Beschreibungsmöglichkeiten können ohne Änderungen oder Erweiterungen zur Beschreibung von nicht variierenden Softwareelementen verwendet werden. Für die Beschreibung der gemeinsamen Bestandteile (*commonalities*) von unterschiedlichen Instanzen einer Familie sind diese Beschreibungsmittel ebenfalls ausreichend, da diese genau den nicht variablen Teil von Softwaresystemen modellieren. Für die Modellierung der variablen Bestandteile von Softwaresystemen — der Variabilität (*variabilities*) — wird die UML um die folgenden zusätzlichen Beschreibungsmittel erweitert:

1. eine Notation für Merkmalsdiagramme in UML — zur Abstraktion und Verwaltung der Variabilität

2. Darstellungsformen für variable Bestandteile von Software, die sich in der statischen und dynamischen Modellierung mittels UML einsetzen lassen:

   - Variationspunkte — zur Darstellung von Alternativen und Erweiterungen
   - und Optionale Elemente — als Spezialform der Variationspunkte.

Im Folgendem werden die Darstellungsmöglichkeiten untersucht und daraus ein Notationsvorschlag erarbeitet. Darauf aufbauend werden im Abschnitt 7.5 deren Einsatzmöglichkeiten auch für die Verhaltensmodellierung beschrieben und in Abschnitt 7.6 Integrationsmöglichkeiten in ein Vorgehensmodell beziehungsweise in ein Modellierungskonzept skizziert. Den Abschluß bildet die Zusammenfassung und ein Ausblick auf die zukünftige Entwicklung der UML, speziell aus dem Blickwinkel der Modellierung von Variabilität.

7.2 Merkmalsmodellierung

Merkmale beschreiben unterscheidbare Eigenschaften eines Systems und sind unabhängig von der Implementierung. In der grafischen Merkmalsmodellierung erfolgt die Darstellung eines Merkmales als Knoten eines Baumes, der die Merkmale in einer Hierarchie strukturiert. Diese repräsentiert ein statisches Modell, dessen Einzelelemente zur Bestimmung von spezifischen Produkten selektiert werden. Die Betrachtung der Elemente der Merkmalsmodellierung erfolgte bereits in Abschnitt 6.1.2.

7.2.1 Diskussion der Darstellung

Da derzeit in UML keine Metaklasse existiert, die zur Darstellung von Merkmalen gedacht ist, muß ein geeigneter Ersatz gefunden werden, der das Wesen von Merkmalen möglichst treffend wiedergibt und mit dem die Beziehungen zwischen Merkmalen in UML darstellbar sind. Aus dem `Foundation::Core`-Package des Metamodells sind nur Subklassen von `Classifier` geeignet, konkret `Class`, `Interface`, `Node`, `Component` und `Artifact`. Artefakte und Knoten entfallen hierbei, denn sie repräsentieren physische Informationen oder Objekte. Da die Merkmale als Eigenschaften eines Softwaresystems oder einer Domäne einen feinkörnigen Modellierungsbaustein (im Sinne eines fast atomaren Elementes) darstellen, fällt die Entscheidung zwischen Klassen und Komponenten zugunsten ersterer — Komponenten repräsentieren modulare Systembestandteile, die in der Regel aus mehreren Teilen (Schnittstellen, Implementierung; respektive mehrere Klassen) bestehen [OMG01a, Seite 2-34]. Als Zusatzargument bietet sich noch die kompaktere grafische Darstellung von Klassen (mit unterdrücktem Attribut-/Operationen-Bereich) gegenüber den Komponenten an. Schnittstellen (`Interface`) stellen dagegen nur eine Beschreibung von Verhalten dar — was sich schlechter mit Merkmalen vereinbaren läßt als die allgemeinere Semantik von Klassen. Daneben ist die Interpretation der Bedeutung von Schnittstellen in der Regel geistig konkreter gefaßt, wogegen Klassen in verschiedenen Funktionen (zum Beispiel als Implementierung oder als Schnittstellenbeschreibung) auftreten können und damit in ihrer Interpretation allgemeiner gehalten sind. In diesem Sinne erscheint die Interpretation von speziell ausgezeichneten Klassen als „Merkmalsklasse“ leichter als für Schnittstellen oder Komponenten. Um die neu einzuführende Notation für Merkmalsdiagramme einfacher und übersichtlicher zu gestalten, soll davon abgesehen werden, den Konzeptknoten gesondert mit der Metaklasse `Interface` zu modellieren.

Neben den Metaklassen des `Foundation`-Paketes sind auch Elemente aus `ModelManagement` als Mittel zur Merkmalsmodellierung denkbar. Der Vorteil bei der Verwendung von Paketen für Merkmale besteht darin, daß implizit für jedes Paket ein Namensraum geschaffen wird, in den sich die zugeordneten Modellelemente einordnen lassen. Dieser Vorteil wird jedoch zum Nachteil, wenn eine Komponente mit mehreren Merkmalen korreliert ist. Dies passiert in der Regel spätestens bei der Einbeziehung von nicht-funktionalen Merkmalen in die Modellierung, die keinem Baustein eindeutig zugeordnet werden können, sondern an verschiedenen Stellen auftreten. Daneben wird diese Art der Merkmalsmodellierung — wie in [Hein00] — sehr schnell unübersichtlich und ist in der Diagramm-Darstellung umständlich zu handhaben.

Metaklassen aus `Behavioral Elements` zu verwenden erscheint unangebracht, da in diesem Fall eine prinzipielle Änderung der Semantik des jeweiligen Meta-Elementes erforderlich wäre.

Die Darstellung des Merkmalsdiagramms mit Hilfe von Klassendiagrammen ist naheliegend und erleichtert aufgrund der optischen Ähnlichkeit zu der Notation in [Czarnecki00, Svahnbrg] das Verständnis der Semantik für Domänenentwickler.

7.2.1.1 Einordnung des Merkmalsmodells

Die Modellierung von Merkmalen mit Klassen und analog die Beschreibung von Merkmalsdiagrammen mittels Klassendiagrammen hat den Vorteil, daß die gesamte Merkmalsmodellierung in einer

Diagrammart „untergebracht“ wird. Um den doch recht stark abweichenden Charakter der Merkmals-Modellelemente von anderen UML-Modellelementen abzugrenzen, ist es sinnvoll, die Merkmalsmodellierung besonders zu kennzeichnen. Dazu bieten sich in UML die Möglichkeiten des *Model Managements* an. Damit werden Modellelemente in Paketen oder dessen Spezialformen Subsystem oder Modell gruppiert.

In Analogie zu der Auszeichnung anderer Modelle in UML soll das Merkmalsmodell explizit als solches gekennzeichnet werden. Die Merkmalsdiagramme stellen eine Sicht auf das Merkmalsmodell dar und werden zum Beispiel durch mehrere Konfigurationen (für jeweils eine Merkmalsauswahl) und die Merkmalsbeschreibungen ergänzt. Darstellungsmöglichkeiten für die Konfiguration werden im Abschnitt 7.2.1.8 besprochen, die Merkmalsbeschreibung kann mit Hilfe der Merkmalsklasse (siehe Abschnitt 7.2.1.7) realisiert werden.

Für die Gruppierung des Merkmalsmodells wird ein Stereotyp für `Package` spezifiziert, in solcherart klassifizierten Paketen sollen sich nur Merkmals-Modellelemente befinden – und auch nur dort. Dieser Stereotyp kann auch für Instanzen der Metaklasse `Model` verwendet werden.

Ein Merkmalsmodell durch mehr als ein Diagramm zu beschreiben, erweist sich als sinnvoll, wenn der Umfang zunimmt. In diesem Fall ist die grafische Darstellung zweckmäßigerweise auf mehrere Diagramme zu verteilen, ein übergeordnetes Diagramm beschreibt die Einordnung in das Gesamtmodell. Die Aufteilung erfolgt einfach, indem zum Beispiel ein Blatt eines Merkmalsdiagramms in einem anderen (untergeordnetem) Diagramm weiter verfeinert wird. Dabei sollte im übergeordneten Diagramm keine weitere Verfeinerung desjenigen Merkmalsknotens stattfinden.

7.2.1.2 Klassen als Merkmalsknoten

Die Verwendung von Klassen zur Darstellung der Merkmalsknoten erfordert gegenüber der in der UML definierten Semantik und Interpretation einige Änderungen, da ein völlig anderes Konzept damit modelliert wird. Also definiert sich eine für die Merkmalsdarstellung verwendete Klasse als grafische Darstellung eines Merkmales (respektive Merkmalsknoten im Diagramm) und repräsentiert eine implementationsunabhängige, unterscheidbare Charakteristik oder Eigenschaft des modellierten Systems. Die für Klassen spezifizierte Semantik darf nicht für als Merkmalsknoten verwendete Klassen verwendet werden.

Die Bezeichnung der Merkmale erfolgt durch einen möglichst prägnanten Bezeichner (UML-Metaklasse `Name`). Dieser wird zweckmäßigerweise als Klassenname verwendet, wobei die in UML definierten Regeln für Klassennamen zu beachten sind. Die Merkmalsklassen enthalten im Gegensatz zu normalen Klassen keine Attribute, Operationen oder Methoden.

In der UML besteht die Möglichkeit, eine Klasse als abstrakt zu kennzeichnen und damit die Erzeugung von Klasseninstanzen zu verhindern. Da Merkmale nicht instanziiert werden können, sollten die Merkmalsklassen folglich als abstrakt gekennzeichnet werden. Diese Frage wird in Abschnitt 7.2.1.8 detailliert angesprochen.

7.2.1.3 Darstellung der Merkmalsknoten, Merkmalstyp

Als Grundlage der nachfolgenden Diskussion ist eine prinzipielle Einordnung der Merkmale auf einer virtuellen Metamodellebene sinnvoll. Dabei geht es um die Frage, ob man den Merkmalstyp als Meta-Attribut einordnet oder die Merkmalstypen besser als Spezialisierungen einer abstrakten Metaklasse „feature“ modelliert. Für die semantische Beschreibung sind beide Möglichkeiten ähnlich, da die grundlegende Semantik von Merkmalen allgemein (für „feature“) beschrieben wird. Um das unterschiedliche Verhalten der Merkmale bei der Auswahl jedoch deutlicher zu modellieren, erscheint die Vererbungsvariante besser geeignet. Denn bei dieser ist explizit sichtbar, daß es verschiedene Arten von Merkmalen gibt. Mit dem Typ als Meta-Attribut wird diese Unterscheidung nur durch die Wertbelegung beschrieben, die dadurch intuitiv als weniger bedeutend erkannt wird. Dies widerspricht je-

doch der expliziten Differenzierung des Merkmaltyps. In der Originaldarstellung (zum Beispiel FODA) enthält der Merkmalstyp eine der Kernaussagen des Merkmalsdiagramms — einen Teil der Kompositionsregeln, welche die Auswahl der Merkmale beeinflussen. Daraus läßt sich schlußfolgern, das die Merkmalstypen eine begrenzte, wohlunterschiedene Menge von Elementen darstellen, die jeweils eine eigene spezifische Semantik besitzen und einen Teil der Semantik gemeinsam besitzen. Diese Unterscheidung — ein Teil der Semantik ist allen Merkmalstypen gemeinsam ("das sie Merkmale sind") und der individuelle Teil legt ihr konkretes spezifisches Verhalten fest. Diese Unterscheidung läßt sich sehr gut mit einer Spezialisierungsbeziehung des Phänomens „Merkmal" zu den einzelnen Merkmalstypen beschreiben.

Für die Auszeichnung der Klassen als Merkmalsknoten und die Spezifikation des Merkmaltyps sind die sinnvollen Möglichkeiten nachfolgend aufgeführt:

1. Auszeichnung der Merkmalsklasse mit einem Stereotyp und Bestimmung des Merkmaltyps in einem TaggedValue (Beispiel: {type = mandatory | optional | alternative}).

   Pro: Der wesentliche Vorteil gegenüber den anderen Varianten ist die explizite Klassifizierung der Klasse als Merkmalsknoten, die sich dadurch am deutlichsten von anderen Klassendefinitionen unterscheidet.

   Kontra: Die Bestimmung des Merkmalstyps als TaggedValue würde auf einer virtuellen Metamodellebene den Typ als Meta-Attribut der Metaklasse „feature" einordnen. Wie vorbereitend diskutiert wurde, ist diese Einordnung jedoch weniger zutreffend als die folgende Variante. Ein weiteres Argument ergibt sich aus der grafischen Darstellung von Klassen in UML: In der kollabierten ([OMG01a, Seite 3-34]) Darstellung im Klassendiagramm wird nur der Klassenname und eventuell das Icon des Stereotyps angezeigt. In dieser Darstellung verliert das Merkmalsdiagramm einen wesentlichen Teil seiner Aussagekraft, da die durch den Typ ausgedrückten Auswahlregeln für Merkmale nicht dargestellt werden.

2. Schaffung eines abstrakten Stereotypen für die Merkmalsspezifikation und die Ableitung von spezialisierten Stereotypen, die den Merkmalstyp spezifizieren.

   Pro: Diese Variante folgt der Interpretation der semantischen Struktur der Merkmalstypen und bildet diese durch die Stereotypen ab. Daneben läßt sich diese Beschreibungsform auch in Verbindung mit UML 1.3 einsetzen, da nur ein Stereotyp pro Merkmalsklasse erforderlich ist. Durch die Beschreibung des Typs durch den Stereotypen ist es ebenfalls möglich, jedem Merkmalstyp (respektive jedem konkreten Stereotyp) ein spezifisches Icon zuzuweisen, das eine kollabierte Darstellung des Merkmals-Klassendiagramms erst sinnvoll macht.

   Kontra: Der Nachteil ist, das solche Klassen auf den ersten Blick nicht explizit in ihrer speziellen Rolle als Merkmal sichtbar sind, sondern nur der Merkmalstyp den Namen des Stereotyps bestimmt (siehe nachfolgende Diskussion im Text). Unter dem Gesichtspunkt, daß das Merkmalsmodell explizit gekennzeichnet werden sollte, ist dieser Nachteil weniger relevant und läßt sich durch eine geschickte Darstellung des Stereotyps mittels Icon weiter minimieren.

3. Zwei Stereotypen: jeweils einer für die Auszeichnung als Merkmal und für den Merkmalstyp.

   Pro: Hierbei werden die Vorteile der ersten beiden Möglichkeiten kombiniert: explizite Spezifikation einer Klasse als Merkmalsknoten und die Möglichkeit, auch die kollabierte Darstellung mit Stereotyp-Icon sinnvoll zu ermöglichen.

   Kontra: Es sind zusätzliche *Well-formedness*-Regeln zu spezifizieren, die dafür sorgen, daß die entsprechende Klasse stets die 2 Stereotypen für Merkmal und Typ besitzt. Genau betrachtet stellt diese Variante nur eine weniger formale Darstellung der vorhergehenden dar und verliert ihre Berechtigung, wenn das gesamte Merkmalsmodell deutlich von anderen UML-Modellen getrennt wird. Daneben verliert diese Möglichkeit den Vorteil, auch für UML 1.3 geeignet zu sein.

Unter Berücksichtigung der vorangegangenen Diskussion der Interpretation von Merkmalstypen auf Metamodellebene stellt die zweite Möglichkeit die Vorzugsvariante dar.

Dabei gibt es für die Bezeichnung der Sub-Stereotypen drei prinzipielle Möglichkeiten: Konkatenation von Typ und „feature“ (Beispiel: „mandatory-feature“), eine verkürzte Variante (Beispiel: „mandfeature“, „alt-feature“) oder die Beschränkung auf den Typnamen (Beispiel: „mandatory“). Erstere ist jedoch aufgrund der Länge des Stereotyp-Namens unübersichtlich (und verdeckt in der Wahrnehmung kürzere Merkmals/Klassennamen), zweitere auf den ersten Blick nicht eindeutig. Die dritte Notation erlaubt eine relativ kompakte und zugleich ausführliche Klassifizierung des Merkmals und wird formal und semantisch durch die Generalisierung der Stereotypen unterstützt.

Der darstellerische Aspekt der dritten Notation wird ebenfalls dadurch unterstützt, daß sich die Elemente des Merkmalsmodells in einem gesonderten Paket befinden sollten und in der Diagramm-Darstellung der Stereotyp des Merkmalsknotens den Merkmalstyp klassifiziert. Da das Merkmalsmodell explizit von anderen Modellen unterschieden werden soll, minimieren sich die Nachteile und der Merkmalstyp wird als Name für den jeweiligen Stereotyp gewählt.

Die Wurzel eines Merkmalsbaumes wird durch den Konzeptknoten gebildet. Dieser stellt kein Merkmal im eigentlichen Sinne dar, sondern das Konzept, dessen (identifizierte) Eigenschaften durch die Merkmalshierarchie beschrieben werden. Entsprechend wird dieser Knoten durch einen eigenen Stereotyp klassifiziert.

7.2.1.4 Grafische Darstellung des Merkmalstyps

Aufbauend auf der gerade getroffenen Entscheidung zur Spezifikation der Merkmalsklassen ist es sehr sinnvoll, die grafische Darstellung durch Definition von Icons für die Merkmalstyp-Stereotypen zu unterstützen. Zur Bestimmung der Icon-Gestaltung bietet sich die Analogie zu der existierenden Notation nach FODA an. Um die Gemeinsamkeit zwischen den Merkmalen zu betonen, sollen die Icons einen gemeinsamen Grundaufbau besitzen. Ein Vorschlag für die Darstellung findet sich bei der Beschreibung der Notation auf Seite 51.

Um die Sonderstellung der externen Merkmale zu unterstützen, kann optional der Rahmen eines externen Merkmalsknotens gestrichelt dargestellt werden. Damit wird die visuelle Erkennung von externen Merkmalen unterstützt. Die Umsetzung dieser Empfehlung ist jedoch kein Bestandteil einer UML-Erweiterung und bezieht sich auf die Werkzeugunterstützung.

7.2.1.5 Darstellung der Merkmalshierarchie

Zwischen den Merkmalsknoten besteht nach [Kang90] eine hierarchische Dekompositionsbeziehung, die einen übergeordneten Merkmalsknoten in untergeordnete Merkmale (Sub-Feature) aufteilt beziehungsweise umgekehrt Merkmale in einem übergeordneten Knoten zusammenfaßt. Für die Kompositionsbeziehung existiert in UML eine spezielle Form der Aggregationsbeziehung, die Komposition, die mit einer gefüllten Raute am zusammenfassenden Ende gekennzeichnet wird. Um eine weitere Änderung der UML-Semantik zu vermeiden, ist es natürlich angebracht, diese Kompositionsbeziehung für die Hierarchiedarstellung der Merkmalsknoten zu verwenden. In der UML können beide Assoziationsenden mit Kardinalitäten versehen werden, die angeben, wie viele Elemente jeweils an einer Instanz einer solchen Beziehung beteiligt sind.

Kardinalitäten: Für die Merkmalsmodellierung reduziert sich die Auswahl der Kardinalitäten erheblich, da ein Sub-Feature (in der Rolle eines Kompositions-Teiles) nur genau einem übergeordneten Merkmal (in der Rolle der Komposition) zugeordnet ist. In umgekehrter Richtung ist die Kardinalität eines Sub-Feature abhängig von seinem Typ — notwendige Merkmale sind immer genau einmal, optionale und alternative Merkmale überhaupt nicht oder genau einmal an der Komposition beteiligt.

Die Betrachtung dieser Kardinalitäten bezieht sich, das soll nochmals betont werden, auf die Merkmalsmodellierung und damit auf die Anzahl der ausgewählten Sub-Merkmale. Genaugenommen ist die Angabe der Kardinalitäten nicht möglich, da es keine Instanzen (im Sinne von Klasseninstanzen; siehe Abschnitt 7.2.1.8) von Merkmalen gibt. Ebenfalls aufgrund der eingeschränkten Belegungsmöglichkeiten sollten die Kardinalitäten in der grafischen Darstellung unterdrückt beziehungsweise auf `1` beziehungsweise `0..1` festgelegt werden. Dabei hat sich die Semantik der Kardinalität folglich nicht auf die Anzahl der beteiligten Instanzen, sondern auf die Multiplizität des betreffenden Merkmals bei der Auswahl zu beziehen. Zu bemerken ist abschließend, das in UML nur binäre Aggregationen/Kompositionen (Regel 2.5.3[3] [OMG01a, Seite 2-57]) möglich sind.

Beziehungstypen: In [Kang98] werden die Merkmalsbaum-Beziehungen entgegen anderen Ansätzen prinzipiell nach Komposition und Generalisierung unterschieden. Aus diesem Grund sollen die für bestimmte Merkmalstypen verwendbaren Beziehungen nicht weiter reglementiert werden, sondern die Unterscheidung und Verwendung obliegt dem Modellierer beziehungsweise der Vorgabe durch das Vorgehen der Merkmalsanalyse. Entsprechend können entweder die bereits diskutierten Kompositionen, oder Generalisierungen für die Beziehungen eines Merkmalsknotens zu allen untergeordneten Knoten modelliert werden. Für die Verfeinerung eines Merkmalsknotens ist dabei nur eine Art von Beziehungen erlaubt, wie es ebenfalls in [Kang98] zu beobachten ist. Spezialisierungsbeziehungen zwischen Merkmalen werden ebenfalls in [Svahnbrg] modelliert.

Neben der Generalisierung und der Komposition werden in [Kang98] auch *implemented_by*-Beziehungen modelliert. Diese können bei Bedarf am zweckmäßigsten mit Dependencies modelliert werden, die zur Unterscheidung von baumübergreifenden (*cross-tree*) Abhängigkeiten mit einem entsprechenden Stereotyp («implementation» oder ähnlich) annotiert werden. Da [Kang98] die einzige Quelle darstellt, in der diese verwendet werden — und derartige auch sonst nicht modelliert werden — soll die Implementierungsbeziehung nicht direkt in die Erweiterung aufgenommen werden — eine diesbezügliche Anpassung ist jedoch problemlos möglich.

XOR-/OR-Alternativen: Die Einschränkung der Beziehungen eines Merkmalsknotens zu den untergeordneten Knoten ist nur für alternative Merkmale interessant. Die Alternativen können sich gegenseitig ausschließen (wie Alternativen in FODA), oder wie *Or-features* nach [Czarnecki00] auch gleichzeitig ausgewählt werden. Um den wechselseitigen Ausschluß zwischen Alternativen zu modellieren, bietet sich ein entsprechender Constraint für die Beziehungen an. Werden dazu Kompositionen verwendet, definiert die UML-Spezifikation bereits einen `{xor}`-Constraint für Assoziationen. Für Generalisierungsbeziehungen ist dieser nicht definiert, dagegen gibt es einen vordefinierten `{disjoint}`-Constraint, der den gleichen Zweck erfüllt. Da die Semantik beider Constraints sich auf Instanzen bezieht und um eine einheitliche Bezeichnung für diesen Constraint zu schaffen, soll dieser für Merkmalsdiagramme neu definiert werden. Dieser modelliert den wechselseitigen Ausschluß der beteiligten Beziehungen, bezogen auf die Merkmalsauswahl.

Bei alternativen Merkmalen tritt ein Zuordnungsproblem auf, wenn mehrere Mengen von Alternativen einem übergeordneten Knoten zugeordnet werden (teilweise in [Czarnecki00]). Die in [Cohen92] verwendete Notation läßt diese Konstellation dagegen nicht zu. Solche Zuordnungen sind gegebenenfalls über das Einfügen von weiteren übergeordneten Merkmalsknoten (jeweils für eine Menge von Alternativen) eindeutig aufzulösen. Für diese Erweiterung wird entsprechend nur eine Menge von alternativen Unterknoten pro Merkmalsknoten zugelassen. Die Auszeichnung eines Merkmales als alternativ erfolgt in jeder Variante selbst und nicht im übergeordneten, zusammenfassenden Merkmalsknoten. Damit ist es möglich, die gesamte Alternative zum Beispiel als optional zu charakterisieren, ohne ein künstliches Zwischenmerkmal einzufügen.

Externe Merkmale: Entsprechend [Svahnbrg] werden die dort eingeführten externen Merkmale als Teil des Merkmalsbaumes modelliert. Alternativ wäre es vorstellbar, diese nicht als Teil des Merkmalsbaumes darzustellen, sondern als „isolierte Inseln“.

Im Abschnitt 7.2.3 wird eine Umsetzungsvorschrift für die Darstellung der FODA-Merkmalsdiagramme in die UML-Notation mit Hilfe dieser Erweiterung angegeben. Diese dient gleichzeitig als Beispiel für die grafische Darstellung.

In Anlehnung an die Notation nach FODA wäre anstelle der Verwendung der speziellen Kompositionsbeziehungen die Nutzung von normalen Assoziationen denkbar, die in der grafischen Darstellung der Notation der Merkmalsdiagramme näher kommen. Da in der Übereinstimmung der grafischen Darstellung mit dem Original jedoch ohnehin Kompromisse erforderlich sind (Klassen als Merkmalsknoten), soll trotzdem die Komposition verwendet werden, da sie in ihrer Semantik dem Charakter der Beziehungen zwischen den Merkmalen besser entspricht. Daneben ist die Darstellung von Merkmalsdiagrammen nicht einheitlich und wird in [Kang90], [Czarnecki00] und [Cohen92] jeweils unterschiedlich dargestellt — ein diesbezüglicher Konsens scheint noch nicht zu bestehen.

7.2.1.6 Abhängigkeiten zwischen Merkmalen

Zusätzlich zu der Kompositions-Hierarchie der Merkmalsknoten können weitere Abhängigkeiten zwischen Merkmalen bestehen, die nicht durch die Baumstruktur abgebildet werden. In Merkmalsdiagrammen werden dazu sogenannte *hard constraints* benutzt, die in *requires* und *mutex* unterschieden werden. Der wechselseitige Ausschluß (*mutex* als Kurzwort für *mutual exclusion*) bedeutet, daß von derart miteinander verbundenen Merkmalen nur eines ausgewählt werden kann. Die `requires`-Abhängigkeit drückt die gegenseitige Selektion aus, das heißt, wird ein Merkmal ausgewählt, muß das andere ebenfalls ausgewählt werden.

Für beide Merkmal-Constraints läßt sich auch jeweils eine gerichtete Variante feststellen, die jedoch nicht in der herkömmlichen Merkmalsmodellierung vorkommt. Für den `requires`-Constraint bedeutet das, daß ein Merkmal ein anderes voraussetzt, jedoch nicht umgekehrt. Der gerichtete wechselseitige Ausschluß macht dagegen keinen Sinn. Da diese Abhängigkeiten in UML mittels Beziehungen (*relationships*) modelliert werden, können die Möglichkeiten der UML (Navigierbarkeit der meisten Beziehungenstypen) in der Weise ausgenutzt werden, daß gerichtete `requires`-Constraints möglich sind — die Ausdrucksmöglichkeiten der Merkmalsmodellierung werden also an dieser Stelle etwas erweitert.

Zusätzlich gibt es noch die sogenannten *weak constraints* (schwache/weiche Abhängigkeiten, treten sozusagen nur manchmal auf) — zum Beispiel Konflikte oder temporale Abhängigkeiten zwischen Merkmalen. Diese werden teilweise nicht grafisch dargestellt, sondern zum Beispiel nur in der Merkmalsbeschreibung dokumentiert. Deswegen soll keine entsprechende Festlegung auf bestimmte Arten erfolgen, jedoch die explizite grafische Darstellung ermöglicht werden, um die Bedeutung bestimmter *weak constraints* für ein Modell zu betonen. Dazu besteht prinzipiell die Möglichkeit, das Merkmalsmodell mit zusätzlichen Beziehungen anzureichern, die mit entsprechenden Bezeichnern (dem Namen der Beziehung) unterschieden werden. Um derartige Constraints von `mutex` und `requires` zu trennen, wird ein Stereotyp definiert, der schwache Abhängigkeiten als solche kennzeichnet. Alternativ besteht die Möglichkeit, diese Abhängigkeiten in der Merkmalsbeschreibung und den entsprechenden TaggedValues festzuhalten. Diese sollten stets Bestandteil der Merkmalsbeschreibung sein, womit eine explizite Modellierung zusätzlich die Bedeutung unterstreicht.

Für die Darstellung der Abhängigkeiten eignen sich im Prinzip nur Assoziationen (Metaklasse `Association`) und *Dependencies* (`Dependency`). Erstere besitzen für die Spezifikation der beteiligten Modellelemente eine zusätzliche assoziierte Metamodellklasse `AssociationEnd`, in der Kardinalitäten, Navigierbarkeit, Sichtbarkeit und weitere Attribute beschrieben werden. Dagegen sind Dependencies stets gerichtet und drücken Abhängigkeiten zwischen Modellelementen aus. Aufgrund

der geringeren Komplexität und der passenderen Semantik sollen Dependencies für die Abbildung der *constraints* verwendet werden. Die wechselseitige Beziehung kann für `mutex` implizit angenommen werden, für `requires` sind unter Umständen zwei entgegengesetzt gerichtete Dependencies erforderlich. Laut UML [OMG01a, Seite 2-37] kann die Richtung einer *Dependency* auch unbedeutend sein und dient nur der Unterscheidung der Elemente. Ein weiterer Vorteil gegenüber Assoziationen ist, daß sich Dependencies semantisch nicht auf Instanzen beziehen, sondern direkt auf die betroffenen Modellelemente angewendet werden [OMG01a, Seite 3-92].

Bei der Interpretation des Merkmalsmodells können die Beziehungen eines Sub-Features zum übergeordnetem Merkmal als `requires`-Abhängigkeit verstanden werden. Das bedeutet, die Auswahl (Selektion) eines Merkmales impliziert die Auswahl des übergeordneten Merkmales.

7.2.1.7 Beschreibung von Merkmalen

Zur Spezifikation der Merkmale selbst — zum Beispiel Beschreibung, Gründe für die Auswahl und Quellen, aus denen das Merkmal extrahiert wurde — sollen im Folgenden nur die unbedingt notwendigen Attribute bestimmt werden, die für eine Beschreibung notwendig sind. Darüber hinaus besteht mittels Kommentaren, den Erweiterungsmechanismen der UML oder einer Erweiterung dieser Erweiterung genügend Möglichkeiten, um zusätzliche Attribute in die Modellierung einzubringen — beispielsweise mittels zusätzlicher TaggedValues.

Jedes Merkmal wird mit einem prägnanten Bezeichner versehen, der im Klassennamen des Merkmalsknoten wiedergegeben wird.

Für die Spezifikation eines Merkmales ist auf jedem Fall die Definition des Merkmals notwendig. Diese beschreibt das Merkmal ausführlich und kann in der Regel in der Beschreibung der entsprechenden Merkmalsklasse untergebracht werden. Zu dieser Beschreibung kann ergänzend die Angabe der Quellen hinzugefügt werden, aus denen das Merkmal extrahiert wurde (zum Beispiel Verweise auf existierende Systeme).

Die Auswahlregeln werden zu einem Großteil durch das Merkmalsdiagramm modelliert. Zusätzliche Auswahlregeln (zum Beispiel auch *weak constraints*) können durch entsprechende Kommentare, Relationen oder textuell beschrieben werden.

Ergänzend zu den Auswahlregeln gibt es die Angabe von Gründen für die Auswahl des Merkmales (*rationale*), die in der Regel als umgangssprachliche Beschreibung, da eine formale Spezifikation nur selten möglich ist. Dies können zum Beispiel Angaben über den zusätzlichen Ressourcenverbrauch bei Auswahl des Merkmals sein.

Optionale Attribute: Zur Dokumentation des Merkmales sind weitere Attribute (siehe Abschnitt 6.1.2.3) denkbar, von denen eine Auswahl nachfolgend aufgezählt wird:

- Zeitpunkt der Festlegung (*use-time*, *binding time*)
  `BindingTime = { development | installation | runtime }`
- Art der Festlegung (*binding mode*) (nach Abschnitt 6.1.4.2)
  `BindingMode = { static | changeable | dynamic }`
- Charakter des Merkmals: funktional, Architektureigenschaft, implementationsspezifisch
  `Nature = { functional | architectural | implementation }`

Diese Merkmalsattribute können bei Bedarf als zusätzliche TaggedValues der Merkmalsklasse hinzugefügt werden. Für diese Erweiterung wird nur der Zeitpunkt der Festlegung, als `BindingTime` bezeichnet, fest vorgesehen. Die Werte für dieses Attribut lassen sich nach vielen Kriterien unterscheiden, für Modellierungszwecke wird aufgrund der Untersuchung im Abschnitt 6.1.4.1 die Einteilung in

Auswahl bei der Entwicklung (*development*), während der Installation (*installation*) oder zur Laufzeit (*run-time*) gewählt.

Weitere Attribute, zum Beispiel für die Kategorisierung nach Abschnitt 6.1.2.4 können bei Bedarf oder in Abhängigkeit von Erfordernissen des Vorgehensmodells hinzugefügt werden. Eine ausführlichere Beschreibung von weiteren Merkmalsattributen findet sich in [Czarnecki00].

7.2.1.8 Selektion von Merkmalen

Die Selektion von Merkmalen erfolgt im Rahmen der Anwendungsentwicklung zur Bestimmung der Eigenschaften, die ein spezifisches Produkt enthalten soll. Bisher erfolgt die Darstellung der Merkmalsauswahl, die einen Teil der Produktkonfiguration darstellt, durch Matrizen oder Tabellen, in denen in der Regel in den Spalten die verschiedenen Produkte beziehungsweise Produktvarianten und in den Zeilen die verschiedenen Merkmale eingetragen sind. Die Auswahl eines Merkmals erfolgt in diesem Fall durch Markieren der entsprechenden Zelle — wobei die Auswahlregeln, die unter anderem durch das Merkmalsmodell beschrieben werden, erfüllt sein müssen.

Diese Merkmalsselektion ist das Ergebnis eines Auswahlprozesses, in dessen Verlauf die Gesamtmenge an Merkmalen, die zur Beschreibung einer Produktgruppe verwendet wurden, begrenzt wird auf die Merkmale, die ein konkretes Produkt beschreiben. Diese produktspezifische Merkmalsmenge stellt damit einen Teil der Produktkonfiguration dar. Eine derartige Merkmalsselektion auch in UML zu modellieren bringt den Vorteil, daß man nicht auf externe Beschreibungsmittel wie Tabellen angewiesen ist, die durch die UML nicht unterstützt werden. Auf der anderen Seite ist die einzige neue Information, die eine solche Darstellung enthält, die Menge der selektierten Merkmale.

Da das Merkmalsdiagramm mit Hilfe eines Klassendiagramms modelliert wird, erscheint es naheliegend, das dazugehörige Objektdiagramm für die Darstellung der Merkmalsselektion zu verwenden. Formal hat es den Vorteil, daß auf diese Weise die Struktur des Objektdiagramms an die des Merkmalsdiagramms gekoppelt ist. Dagegen spricht jedoch, daß selektierte Merkmale im Gegensatz zu Objektinstanzen keine Identität besitzen — tatsächlich ist das selektierte Merkmal identisch mit dem im Merkmalsdiagramm — es handelt sich um ein und dieselbe Instanz des Konzeptes „Merkmal“ (im Sinne der nicht vorhandenen UML-Metaklasse `Feature`). Folglich läßt sich die Klasse-Instanz-Beziehung von Objekten nicht auf Merkmale übertragen und das Objektdiagramm würde den Trugschluß unterstützen, eine solche Beziehung liegt auch bei Merkmalen vor.

Da es nicht Anliegen dieser Arbeit ist, die Frage nach dem Für und Wider von Objektdiagrammen als Merkmalselektions-Diagramme zu entscheiden, soll der Ansatz kurz angerissen werden.

Darstellung der Merkmalsselektion mit Objektdiagrammen: Laut UML-Spezifikation sind Objektdiagramme nichts anderes als Klassendiagramme, die nur Objektinstanzen enthalten [OMG01a, Seite 3-38]. Entsprechend können die für die Merkmalsmodellierung definierten Stereotypen und TaggedValues auch in Objektdiagrammen angewendet werden.

In der UML werden Objektdiagramme als „Schnappschüsse eines Detailzustandes“ zu einem bestimmten Zeitpunkt bezeichnet. Insofern ist die Analogie zu einer Produktkonfiguration berechtigt — diese zeigt eine konkrete Merkmalsauswahl. Dagegen spricht, daß ein selektiertes Merkmal nicht eine Instanz seiner Entsprechung im Merkmalsdiagramm ist. Weiterhin ist eine Merkmalsauswahl stabiler und dauert über einen längeren Zeitraum an als der Objektzustand eines Systems.

Weiterhin ergeben sich Fragen zur Behandlung der unterschiedlichen Bindezeiten von Merkmalen - vor dem Start des Systems sind zum Beispiel zur Entwicklungs- und Installationszeit gebundene Merkmale festgelegt, die Auswahl von zur Laufzeit gebundenen Merkmale muß dagegen noch getroffen werden. Insofern kann ein solches Objektdiagramm nur eine teilweise Auswahl darstellen und vermischt damit Teile von Merkmalsdiagramm (noch nicht gebundene Merkmale) und Merkmalsselektion (gebundene Merkmale). Eine solche Vermischung von Objektinstanzen und Klassen in einem Diagramm ist laut UML Notation Guide möglich.

Ebenso — wie für die als Merkmalsdiagramme verwendeten Klassendiagramme — wäre für Objektdiagramme als Merkmalsselektions-Diagramme die explizite Trennung von „normalen“ UML-Diagrammen anzuraten — dazu können die für Klassendiagramme definierten Stereotypen verwendet werden.

Mit den Objektdiagrammen würde gegenüber Matrizen oder Tabellen eine zusätzliche, grafische Notationsform gewonnen. Kritikpunkt an dieser Darstellung bleibt die scheinbare Übertragung der Klasse-/Instanz-Beziehung auf Merkmale, die einen falschen Eindruck erweckt und bei Anwendern, die mit dem Konzept der Merkmalsmodellierung nicht so vertraut sind, die falsche Gleichsetzung mit objektorientierten Ansätzen begünstigt. Eine Eigenschaft der Merkmalsmodellierung ist es ja gerade, eine Beschreibung völlig unabhängig von jeglichen Implementierungstechniken — also auch unabhängig von der Objektorientierung — zu ermöglichen und statt dessen das Softwareprodukt aus Sicht des Endanwenders zu beschreiben.

Um den Unterschied zwischen Merkmalsklassen und OO-Klassen nicht zu verwischen, sollte die Verwendung von Objektdiagrammen zur Modellierung von Merkmals-Selektionen wohl überlegt werden. Daneben sollte man die Frage stellen, welche Vorteile die Darstellung einer Merkmalsselektion als Objektdiagramm gegenüber der Verwendung von Tabellen respektive Matrizen bringt. Nachteilig kann zum Beispiel der erhöhte Aufwand sein, da eine Tabelle alle Konfigurationen, ein Objektdiagramm jedoch nur eine einzige Selektion darstellen kann.

Alternative Darstellungsvarianten: Die Darstellung der Merkmalsauswahl kann unter Einbeziehung der Werkzeugunterstützung ebenfalls durch visuelle Mittel wie eine Änderung der Linienstärke erfolgen. Die farbliche Hinterlegung von Klassen erlaubt sogar die Darstellung unterschiedlicher Konfigurationen in einem Diagramm. Solche Darstellungsmöglichkeiten werden jedoch nicht von UML unterstützt und müßten durch Erweiterungen, zum Beispiel die Beschreibung der Auswahl eines Merkmals in einem TaggedValue, implementiert werden.

7.2.2 Notation

Nachfolgend wird die aus den vorangegangenen Überlegungen entstehende Notation analog einem UML-Profil spezifiziert. Daraus resultiert die Verwendung der englischen Bezeichner für die Stereotypen, TaggedValues und Meta-Attribute.

7.2.2.1 Allgemeines

Um die Merkmalsmodellierung mit ihrer Bedeutung und Semantik von den übrigen UML-Modellen zu unterscheiden, wird ein Stereotyp für Merkmals-Pakete (die alle Elemente des Merkmalsmodells enthalten) definiert. Damit soll ebenfalls betont werden, daß die in Merkmalsdiagrammen verwendeten Klassen keine Klassen im Sinne der UML-Semantik sind.

Merkmalsdiagramme werden mit Hilfe von speziellen Klassendiagrammen modelliert, die ausschließlich der Merkmalsmodellierung dienen. Derartige Diagramme enthalten nur Klassen, die Merkmalsknoten repräsentieren, die Beziehungen zwischen diesen und eventuell Kommentare. Folglich müssen alle enthaltenen Klassen als Merkmal gekennzeichnet sein. Die verfügbaren Beziehungstypen werden nicht pauschal eingeschränkt, sollten jedoch mit einem der definierten Relations-Stereotypen (oder einen abgeleiteten Stereotypen) klassifiziert sein.

Für alternative Merkmale eines gemeinsamen übergeordneten Knotens wird die Festlegung getroffen, daß diese eine einzige Menge von Alternativen darstellen. Es ist demzufolge nicht möglich, mehr als eine Menge von alternativen Varianten für einen Knoten zu spezifizieren. Dieser übergeordnete Knoten kann jedoch weitere untergeordnete Merkmale anderen Typs enthalten, diese sind nicht Bestandteil der Alternativenmenge.

Die für die Darstellung von Merkmalsknoten verwendeten Klassen enthalten generell keine Attribute oder Operationen/Methoden und können zur besseren Unterscheidbarkeit von Klassendiagrammen stets kollabiert (iconifiziert, nur Klassenname und Stereotyp-Icon) dargestellt werden. Ein optionales Icon für jeden Merkmalsknoten dient zur grafischen Darstellung des Merkmalstyps.

Die Auswahl eines Merkmals setzt prinzipiell die Auswahl des übergeordneten Merkmales voraus. Dabei wird unterschieden nach notwendigen, optionalen und alternativen Merkmalen. Die Auswahl von alternativen Merkmale kann mit dem definierten `{xor}`-Constraint eingeschränkt werden, in diesem Fall kann nur genau eine der Alternativen ausgewählt werden.

7.2.2.2 Stereotypen

Stereotyp	isAbstract	baseClass	Parent	Beschreibung
features	false	`Package`	–	Paket, das Elemente des Merkmalsmodelles enthält
concept	false	`Class`	–	Konzeptknoten eines Merkmalsdiagrammes
feature	true	`Class`	–	Merkmalsknoten
mandatory	false	`Class`	feature	notwendiges Merkmal
optional	false	`Class`	feature	optionales Merkmal
alternative	false	`Class`	feature	alternatives Merkmal
external	false	`Class`	feature	externes Merkmal
constraint	true	`Dependency`	–	Abhängigkeit zwischen Merkmalen
weakConstraint	false	`Dependency`	constraint	schwache Abhängigkeit
requires	false	`Dependency`	constraint	Anwesenheit des Merkmales
mutex	false	`Dependency`	constraint	wechselseitiger Ausschluß

Die Tabelle spezifiziert die erforderlichen Stereotypen für Merkmalsdiagramme. „Parent“ gibt den übergeordneten Stereotypen an, „baseClass“ und „isAbstract“ bestimmen die konkrete Wertebelegung der Meta-Attribute: *baseClass* bestimmt die Metaklassen der Modellelemente, auf die ein Stereotyp angewendet werden darf; ist *isAbstract* mit `true` belegt, kann der jeweilige Stereotyp nicht in (M1-)Modellen verwendet werden (die damit bezeichnete virtuelle Metaklasse, zum Beispiel `feature`, ist abstrakt).

Nachfolgend wird die mit den Stereotypen verbundene spezifische Semantik beschrieben — ein Merkmal kann prinzipiell nur ausgewählt werden, wenn das übergeordnete Merkmal selektiert wurde:

- **«features»** kennzeichnet ein Paket, das Elemente eines Merkmalsmodells enthält.
- **«concept»** kennzeichnet den Konzeptknoten eines Merkmalsdiagramms. Dieser Knoten besitzt keine übergeordneten Merkmale.
- **«feature»** kennzeichnet einen Merkmalsknoten im allgemeinen. Dies ist ein abstrakter Stereotyp, der die gemeinsamen Eigenschaften der abgeleiteten Stereotypen beschreibt: den Merkmalscharakter einer Klasse zu kennzeichnen.
- **«mandatory»** kennzeichnet ein notwendiges Merkmal. Ein notwendiges Merkmal wird immer ausgewählt, wenn das übergeordnete Merkmal (beziehungsweise der Konzeptknoten) selektiert ist.
- **«optional»** kennzeichnet ein optionales Merkmal. Wurde der übergeordnete Knoten ausgewählt, kann ein optionales Merkmal ausgewählt werden, gegebenenfalls unter Beachtung zusätzlicher Einschränkungen (Constraints).

- **«alternative»** kennzeichnet ein alternatives Merkmal. Wurde der übergeordnete Knoten ausgewählt, muß mindestens eines der diesem Knoten untergeordneten, alternativen Merkmale ausgewählt werden. Dabei sind zusätzlich definierte Constraints und Abhängigkeiten zu berücksichtigen, zum Beispiel ist in Verbindung mit dem `{xor}`-Constraint die Auswahl nur genau eines alternativen Merkmales möglich.

- **«external»** kennzeichnet ein externes Merkmal. Ein externes Merkmal wird nicht durch das modellierte System selbst, sondern durch die zugrunde liegende Plattform (Betriebssystem, Middleware) realisiert.

- **«constraint»** bezeichnet Abhängigkeiten zwischen Merkmalen, die nicht durch die Merkmalshierarchie modelliert werden. Dies ist ein abstrakter Stereotyp, der die gemeinsame Eigenschaft der abgeleiteten Stereotypen beschreibt: die Merkmalsauswahl einzuschränken.

- **«weakConstraint»** kennzeichnet Abhängigkeiten zwischen Merkmalen, die (in Abhängigkeit von Kontextbedingungen) nicht in allen Konfigurationen auftreten.

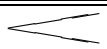
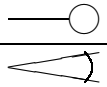
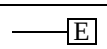
- **«requires»** bezeichnet eine *requires*-Abhängigkeit. Ist diese *requires*-Abhängigkeit gerichtet, so kann das Merkmal in der *client*-Rolle (Pfeilende) nur ausgewählt werden, wenn das Merkmal in der *supplier*-Rolle (Pfeilspitze) selektiert ist (siehe auch [OMG01a, Seite 3-92]). Bei einer ungerichteten Abhängigkeit müssen beide Merkmale oder keines von beiden ausgewählt werden.

- **«mutex»** klassifiziert den wechselseitigen Ausschluß. Von den beteiligten Merkmalen kann maximal eins ausgewählt werden.

Die Relationen zwischen Merkmalen können optional mit einer aussagekräftigen Bezeichnung versehen werden, die die Beziehung beschreibt.

Im praktischen Einsatz kann der Merkmalsbaum auch partitioniert werden, in diesem Fall stellt jedes Merkmalsdiagramm einen Teilausschnitt des Modells dar. Jedes Teildiagramm sollte dabei genau einen Knoten enthalten, der in diesem Diagramm keinen sichtbaren übergeordneten Merkmalsknoten zugeordnet ist. Diese Zuordnung muß in einem anderen (übergeordneten) Merkmalsdiagramm dargestellt werden. Im Gesamtmodell darf es keinen Merkmalsknoten geben, der nicht mit einem übergeordneten Merkmal oder dem Konzeptknoten verbunden ist. Der Konzeptknoten besitzt keine übergeordneten Merkmale.

Zur Unterstützung der grafischen Darstellung werden die folgenden Icons für die Merkmalstyp-Stereotypen vorgeschlagen:

Stereotyp	Icon-Beschreibung	Icon-Darstellung
mandatory	schematische Baumverzweigung von links nach rechts	
optional	waagerechter Strich mit hohlem Kreis	
alternative	Baum-Verzweigung mit verbindenden Kreisbogen	
external	eingerahmter Großbuchstabe 'E'	

Eine Alternatividee ist, Merkmale mit einem stilisierten 'F' und als Index die Kennung des Merkmalstyps zu symbolisieren, zum Beispiel wie folgt: F_M, F_O, F_A, F_E.

7.2.2.3 Tagged Values

Schlüsselwort	Stereotyp	Typ	Multiplizität
BindingTime	feature	`FeatureBindingtimeEnum` (Enumeration: { development, installation, runtime })	`0..1`
Rationale	feature	String	`0..1`

Der Merkmalsname ist der Name der jeweiligen Klasse. Die Entscheidungsgründe für die Auswahl des Merkmales werden in `Rationale` in textueller Form dargestellt — zum Beispiel der Ressourcenverbrauch, die mit der Auswahl des Merkmales verbunden ist. Der Bindezeitpunkt `BindingTime` legt den Zeitpunkt fest, zu dem die Existenz des Merkmales (für optionale) beziehungsweise die Auswahl einer Alternative entschieden wird (siehe Abschnitt 6.1.4.1). Eine detaillierte Merkmalsbeschreibung kann je nach Werkzeugunterstützung in der Detailbeschreibung der Klasse dokumentiert werden.

Diese TaggedValues sind an den Stereotyp «feature» gebunden und sind damit in jeder Klasse respektive Merkmal des Merkmalsdiagramms vorhanden. Wird ein Wert nicht angegeben, wird das als unspezifiziert interpretiert und die Darstellung des TaggedValue kann entfallen (analog zu [OMG01a, Seite 3-74]).

7.2.2.4 Constraints

Name	Beschreibung	baseClass
xor	wechselseitiger Ausschluß der beteiligten Beziehungen	Relationship

Der `{xor}`-Constraint wird für Beziehungen (im Metamodell: `Relationship`) in Merkmalsdiagrammen definiert und beschreibt den wechselseitigen Ausschluß der beteiligten Elemente. Die Interpretation erfolgt analog zu dem in [OMG01a, Seite 2-21] definierten `{xor}`-Constraint für Assoziationen: die Einschränkung wird angewendet auf eine Menge von gleichartigen Relationen (Instanzen einer Metaklasse) und spezifiziert, daß nur genau ein Element dieser Menge ausgewählt werden kann. Im Gegensatz zu dem Constraint für Assoziationen wird jedoch eine Einschränkung bezüglich der Anwesenheit der Beziehung getroffen. Im Kontext der Merkmalsmodellierung wird diese Einschränkung auf Alternativen angewendet und bedeutet, daß von den beteiligten Merkmalen nur genau eines ausgewählt werden kann.

Die folgenden Constraints beschreiben Einschränkungen bei der Anwendung der Stereotypen des virtuellen Metamodells (entspricht den *Well-formedness rules* der Erweiterung). Die Regeln beziehen sich jeweils auf das stereotypisierte Modellelement.

Zusätzliche Operationen:

[1] Die Operation **isStereotyped** prüft, ob das Objekt den gegebenen Stereotyp besitzt.

```
context ModelElement def:
  let isStereotyped (st_name : String) : Boolean =
    self.stereotype->exists( s | s.name=st_name )
```

features (Model Management::Package)

[1] alle enthaltenen Modellelemente eines Merkmalspaketes müssen Merkmalselemente sein: Merkmalsknoten und Konzeptknoten, Generalisierungen, Kompositionen und Dependencies sowie Constraints, untergeordnete Merkmals-Pakete und Kommentare. Zusätzlich sind Abstraktionen für dokumentarische Zwecke («trace»-Abstraktionen) erlaubt.

```
self.ownedElement->forEach( e |
  e.oclIsTypeOf(Class) and (
    e.isStereotyped('mandatory') xor
    e.isStereotyped('optional') xor
    e.isStereotyped('alternative') xor
    e.isStereotyped('external') xor
    e.isStereotyped('concept')
  ) xor (
    e.oclIsKindOf(Package) and e.isStereotyped('features')
  ) xor e.isTypeOf(Association) xor
  e.isTypeOf(Generalization) xor (
    e.oclIsTypeOf(Dependency) and e.stereotype->union(
      e.stereotype.generalization.parent
    )->exists (s | s.name='constraint')
  ) xor e.isTypeOf(Abstraction) xor e.isTypeOf(Constraint)
  xor e.isTypeOf(Comment)
)
```

[2] Es existiert genau eine Klasse, die als Konzept gekennzeichnet ist.

```
self.ownedElement->select( me |
  me.isStereotyped('concept') )->size()=1
```

concept (Core::Class)

[1] Ein Konzeptknoten wird nicht generalisiert und ist kein Teil einer Komposition.

```
self.association->select(
  ae | ae.aggregation=#none
)->isEmpty() and self.generalization->isEmpty()
```

***feature* (Core::Class)**

[1] Jede Merkmalsklasse befindet sich in einem Paket, daß als «features» klassifiziert ist.

```
self.namespace.isStereotyped('features')
```

[2] Klassen mit Merkmals-Stereotypen besitzen keine Attribute, Operationen oder Methoden und können nicht als Typen verwendet werden.

```
self.feature->isEmpty() and self.typedfeature->isEmpty()
and self.typedparameter->isEmpty()
```

[3] Merkmalsklassen sind immer abstrakt[1].

```
self.isAbstract=true
```

[4] Jeder Merkmalsknoten besitzt genau einen übergeordneten Knoten: Jede Merkmalsklasse ist entweder Teil genau einen Aggregats oder ist an genau einmal in der Rolle der Spezialisierung an einer Generalisierungsbeziehung beteiligt.

```
self.association->select(
  ae | ae.aggegration=#none
)->size()=1 xor self.generalization->size()=1
```

[5] Zwischen Merkmalsklassen sind keine normalen Aggregationsbeziehungen erlaubt.

[1] vorbehaltlich der Verwendung von Objektdiagrammen gemäß Abschnitt 7.2.1.8

```
self.association->forAll( ae :AssociationEnd |
                          ae.aggregation=#composite or
                          ae.aggregation=#none )
```

[6] Kompositionen von Merkmalsklassen besitzen die Kardinalität `1`.

```
self.association.forAll(
  ae | ae.aggregation=#composite implies
  ae.mulitplicity.range->forAll(
    r | r.lower=1 and r.upper=1
  )
)
```

[7] Ein Merkmalsknoten hat genau einen Typ und entsprechend genau einen Merkmals-Stereotyp.

```
self.isStereotyped('mandatory') xor
self.isStereotyped('optional') xor
self.isStereotyped('alternative') xor
self.isStereotyped('external')
```

mandatory (Core::Class)

[1] Die aggregierten notwendigen Merkmalsklassen von Kompositionen besitzen die Kardinalität `1`.

```
self.association.forAll( ae | ae.aggregation=#none implies
  ae.mulitplicity.range->forAll(
    r | r.lower=1 and r.upper=1
  )
)
```

optional, alternative (Core::Class)

[1] Die aggregierten optionalen und alternativen Merkmalsklassen von Kompositionen besitzen die Kardinalität `0..1`.

```
self.association.forAll( ae | ae.aggregation=#none implies
  ae.mulitplicity.range->forAll(
    r | r.lower=0 and r.upper=1
  )
)
```

7.2.2.5 UML-Modell der Erweiterungen

Abbildung 7.1 stellt die grafische Spezifikation der Erweiterung für Merkmalsmodelle in UML-Notation dar. Aus Komplexitätsgründen sind die mit den Stereotypen assoziierten Constraints nicht mit dargestellt.

7.2.2.6 Kompatibilität zu UML 1.3

Die hier vorgestellten Erweiterungen für Merkmalsmodelle lassen sich ohne Einschränkungen auch in UML Version 1.3 einsetzen. Für die Abwärtskompatibilität der Erweiterungsmechanismen gelten die in UML aufgestellten Regeln [OMG01a, Seite 2-88].

7.2.3 Zuordnung zu FODA-Elementen

Nachfolgend werden Regeln zur Abbildung von Merkmalsmodellen nach FODA oder FORM angegeben.

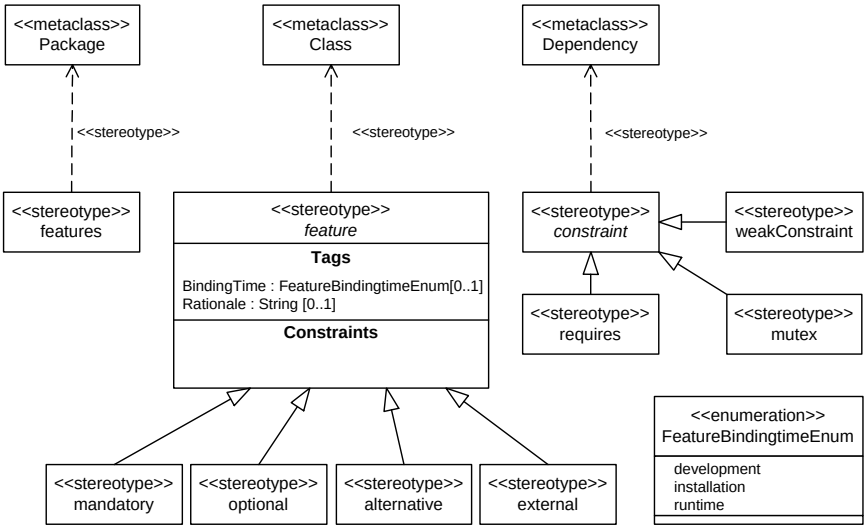


Abbildung 7.1: UML-Modell der Erweiterung für Merkmalsmodelle

7.2.3.1 Allgemeine Regeln

Jedes Merkmal wird in `Classifier`-Notation nach UML ohne Operationen und Attribute (einfaches Rechteck) dargestellt. Der Bezeichner des Merkmals (Merkmalsname) bildet generell den Klassennamen.

Soweit bekannt, wird der Bindezeitpunkt im Wert des Tags `BindingTime` beschrieben. Für Merkmale sind dabei die Werte 'development', 'installation' und 'runtime' möglich. Die Gründe für die Auswahl des Merkmales werden generell im Wert des Tags `Rationale` in Textform festgehalten.

7.2.3.2 Übertragung des Merkmaltyps

Notwendige Merkmale werden mit dem Stereotyp «mandatory» versehen. Optionale Merkmale bekommen den Stereotyp «optional». Externe Merkmale werden mit dem Stereotyp «external» gekennzeichnet.

Bei alternativen Merkmalen wird jede Variante mit dem Stereotyp «alternative» klassifiziert. Für ausschließliche Alternativen wird zusätzlich ein `{xor}`-Constraint mit den Beziehungen zum übergeordnetem Merkmal aller betroffenen Alternativen verbunden. Für Oder-Merkmale nach [Czarnecki00] entfällt dieser Constraint.

Setzt sich ein Merkmalsknoten aus mehreren disjunkten Mengen von Alternativen zusammen, muß für jede Menge von Alternativen ein zusätzlicher Merkmalsknoten eingefügt werden, um die eindeutige Zuordnung der Alternativen zu gewährleisten.

7.2.3.3 Beziehungen zwischen Merkmalen

Die in Merkmalsbäumen überwiegend vertretene Kompositionsbeziehung (*composed_of*, *consists_of*) wird in UML durch Kompositionen modelliert. Der übergeordnete Merkmalsknoten ist dabei das Aggregat (gefüllte Raute). Je nach Art der modellierten Beziehungen (zum Beispiel wie in FORM) können anstelle der Kompositionsbeziehung auch Generalisierungsbeziehungen verwendet werden. Das übergeordnete Element stellt dabei die Generalisierung (Pfeilspitze) dar.

Zusätzliche Abhängigkeiten zwischen Merkmalen (*feature constraints*) werden durch Dependency-Beziehungen modelliert, die mit der Art des Constraints stereotypisiert werden.

Die in FORM zusätzlich modellierten *implemented_by*-Beziehungen können durch Dependencies oder Abstraktionen dargestellt werden.

7.2.3.4 Grafische Darstellung

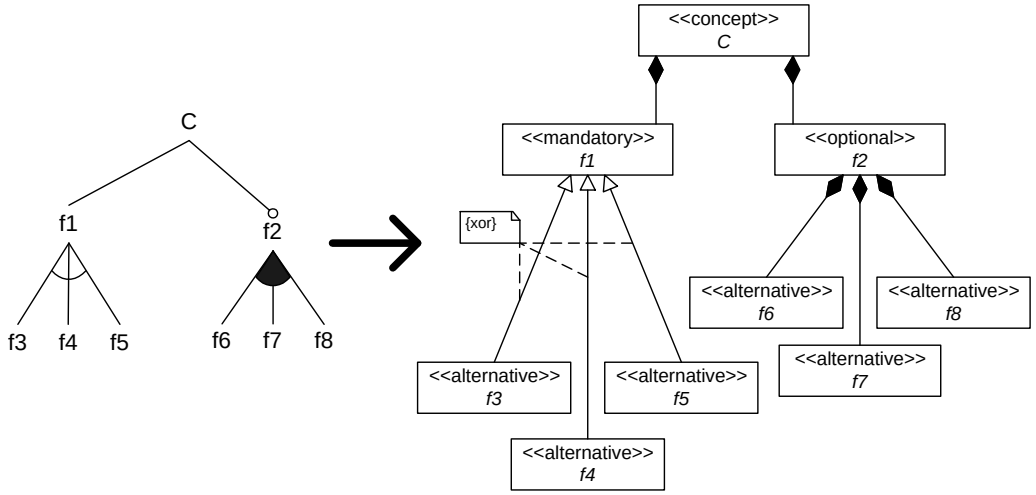


Abbildung 7.2: Darstellung von notwendigen und optionalen Merkmalen

Abbildung 7.2 verdeutlicht die Umsetzung von Merkmalen in UML. Der Knoten `C` stellt den Konzeptknoten dar. Merkmal `f1` ist ein notwendiges, `f2` ein optionales Merkmal. Beide sind Variationspunkte, `f1` für eine Menge von sich gegenseitig ausschließenden, alternativen Merkmalen und `f2` für eine Menge von *Or-features*.

Gegenüber der FODA-Notation enthält die UML-Darstellung zusätzliche Informationen über die Beziehungen der Merkmale innerhalb des Merkmalbaumes (wie in FORM [Kang98]). In Abbildung 7.2 wurden die Oder-Merkmale `f6`, `f7` und `f8` beispielhaft als Dekomposition von `f2` modelliert — `f3`, `f4` und `f5` dagegen als Spezialisierungen von `f1`. Diese Unterscheidung wird in der FODA-Notation nicht getroffen.

Dabei soll betont werden, daß der modellierte Merkmalstyp (reflektiert die Regel für die Auswahl des Merkmals) unabhängig von der Beziehung des Merkmales zum übergeordnetem ist. Die Art der Beziehung wird im Zuge der Merkmalsanalyse identifiziert und kann wie zum Beispiel in [Kang90] oder [Czarnecki00] auf die Komposition beschränkt werden — in diesem Fall entfällt die Modellierung von Spezialisierungen im Merkmalsdiagramm.

7.3 Modellierung von Variationspunkten

In der Softwareentwicklung mit Wiederverwendung — in der Regel unter Anwendung von Komponenten — finden sich seit 1997 Ansätze, um verschiedene Realisierungen von Komponenten oder allgemeiner, von Konzepten zu modellieren. In RSEB ([Jacobson97]) wurden dazu Variationspunkte modelliert, die einen Verzweigungspunkt für verschiedene Varianten darstellen. Zum Beispiel lassen sich im Falle

einer Kontoüberziehung verschiedene Reaktionen unterscheiden: die verursachende Transaktion wird abgebrochen oder eine Überziehung wird gebucht.

Das Konzept der Variationspunkte wurde auch in der softwarefamilien-orientierten Entwicklung aufgegriffen. Es bietet eine einfache Möglichkeit, Variabilität in Softwaresystemen an explizit bezeichneten Punkten (*hot spots*) zu modellieren und damit auf wesentliche Stellen zu konzentrieren — was die Behandlung und effiziente Umsetzung einfacher macht, als wenn Variabilitäten unkontrolliert über das gesamte System hinweg enthalten sind. Die explizite Modellierung von Unterschieden in Softwaresystemen ist außerdem ein notwendiges Mittel, um verschiedene Ausprägungen zum Beispiel einer Domäne zu erfassen und damit eine systematische Wiederverwendung unter Einbezug der Gemeinsamkeiten und Unterschiede zu ermöglichen.

Für das eingangs beschriebene Konzept der Modellierung von Variabilität stellen Variationspunkte ein zentrales Mittel dar, um unterschiedliche Ausprägungen von Systembestandteilen zu modellieren. Nach der Untersuchung des Wesens von Variationspunkten soll dazu eine möglichst generelle Beschreibung in UML entwickelt werden, mit der sich solche Unterschiede erfassen lassen — diese Variationspunkte lassen sich in verschiedenen Abstraktionsebenen modellieren und sind damit in unterschiedlichen Entwicklungsphasen anwendbar.

7.3.1 Untersuchung von Variationspunkten

Ganz allgemein dienen Variationspunkte zur Beschreibung von variierenden Bestandteilen eines Softwaresystems. Bei genauerer Betrachtung treten an bestimmten Stellen unterschiedliche Varianten auf, die verschiedene Ausprägungen der Variabilität eines übergeordneten Elementes beschreiben. Beispiele dafür sind die *extension points* in Anwendungsfällen oder eine Schnittstelle, von der es verschiedenartige Implementierungen gibt. Über diese relativ einfachen Fälle hinaus können die Unterschiede auch innerhalb eines Elementes, zum Beispiel in einer Klasse oder in einer Komponente auftreten. In diesen Fällen verändert sich möglicherweise nur ein Teil, zum Beispiel die Fehlerbehandlungsroutine als Ausdruck unterschiedlicher Fehlerbehandlungsstrategien.

In der Praxis existieren für diese Fälle von Variabilität zum Teil bereits Lösungsverfahren, die zum Beispiel mittels Analyse- oder Designmuster (*design patterns*) beschrieben werden. Damit werden jedoch nur die Symptome beschrieben und die Variabilität nicht explizit als solche modelliert. In diesem Fall bieten Variationspunkte eine explizite Beschreibungsmöglichkeit und eliminieren die Notwendigkeit, sich bereits frühzeitig auf konkrete Mechanismen zur Behandlung der Variabilität festzulegen — statt dessen kann die Entscheidung über die technische Umsetzung und dazu verwendete Verfahren zum passenden Zeitpunkt getroffen werden. Auf diese Weise bieten Variationspunkte „nebenbei“ eine Möglichkeit, von Mustern zur Umsetzung von Variabilität zu abstrahieren.

Durch das allgemeine Konzept, Unterschiede durch die Zerlegung in den Ort der Variabilität und die Beschreibung möglicher Varianten zu modellieren, lassen sich Variationspunkte prinzipiell in allen (modellierenden) Entwicklungsphasen und für verschiedene Modellierungselemente einsetzen. Nachfolgend werden die einzelnen „Bestandteile“ von Variationspunkten untersucht und ihre Modellierungsmöglichkeiten in UML diskutiert.

7.3.1.1 Anwendung von Variationspunkten

An erster Stelle soll geklärt werden, an welchen Stellen sich Variationspunkte sinnvoll einsetzen lassen: von ihrer Beschreibung her modellieren Variationspunkte eine Auswahl zwischen Alternativen. Diese können durch verschiedene Modellelemente gebildet werden — in Softwarefamilien wird oft der Begriff *Asset* genutzt. Diese Bausteine entsprechen im UML-Metamodell am ehesten den Spezialisierungen der Metaklasse `Classifier`. Diese ist eine Spezialisierung von generalisierbaren Elementen, ebenfalls wie Assoziationen, Stereotypen, Kollaborationen und Pakete.

Da die Anwendung von Variationspunkten auf Paketen und Kollaborationen sehr sinnvoll erscheint, soll die Spezifikation nicht nur für `Classifier`, sondern allgemeiner für `GeneralizableElement` erfolgen. Damit ist implizit ebenso gesichert, daß es mindestens ein Modellelement für die Zuordnung der Varianten zum Variationspunkt gibt (siehe 7.3.1.3). Interessanterweise können laut UML-Metamodell auf diese Weise auch Assoziationen als Variationspunkt dargestellt werden (Generalisierung von Assoziationen [OMG01a, Seite 3-89]).

Die Spezifikation für `ModelElement` würde die Anwendung auf alle UML-Modellelemente erlauben, ist jedoch extrem allgemein und erschwert eine klare Benutzung der Erweiterung — es besteht die Gefahr, daß Variationspunkte in Modellelementen modelliert werden, für die das nicht unbedingt sinnvoll ist. Dazu gehören zum Beispiel Attribute und Operationen — die bei der Entwicklung mit systematischer Wiederverwendung üblicherweise nicht variieren, um die Komplexität der Variationen kontrollierbar zu halten (vergleiche Verzicht auf Modellierung von Variabilität auf Code-Level, Abschnitt 7.1). Statt dessen soll die Beschränkung auf generalisierbare Elemente bewußt erfolgen — ohne das die übliche Anwendbarkeit (im Sinne von: wie in RSEB und den gängigen Entwicklungsmethoden) von Variationspunkten eingeschränkt wird.

Die Beschreibung für Objektinstanzen ist nicht sinnvoll, da Variationspunkte eine Alternative zur Entwicklungszeit dokumentieren — zu dieser Zeit existieren gemeinhin keine Instanzen des Modells. Demzufolge werden Variationspunkte nur auf Modellebene (M1 in UML) modelliert, und nicht in Benutzerobjekten (M0).

7.3.1.2 Festlegung der Lokalität

Ein zentrales Anliegen von Variationspunkten ist es, explizit die Position zu kennzeichnen, an der Unterschiede in dem modellierten System auftreten. In der Regel läßt sich dazu ein Bereich identifizieren, von dem verschiedene Implementierungen (beispielsweise verschiedene Sortieralgorithmen) existieren. Im Modell kann dieser Bereich aus einem oder mehreren Modellelementen bestehen. Die Varianten, die diesen Bereich durch konkrete Angaben realisieren, müssen jeweils den gesamten Bereich abdecken — also alle enthaltenen Elemente bereitstellen.

In der praktischen softwaretechnischen Umsetzung wird es oftmals notwendig sein, diesen Bereich explizit zu deklarieren (zum Beispiel durch eine zu implementierende Schnittstelle) und dadurch einen Rahmen für die Implementierung der Varianten zu schaffen. Da eine universelle Beschreibung des variablen Bereiches mangels passender Metaklassen in UML nicht möglich ist und am Ende durch die verfügbaren Technologien zur Umsetzung beschränkt wird, soll davon ausgegangen werden, daß als Beschreibungsmittel für den Umfang eines Variationspunktes existierende Modellelemente dienen — in UML werden das häufig Schnittstellen oder abstrakte Klassen sein. Danach ist es noch erforderlich, die Position des Variationspunktes und der dazugehörigen Varianten explizit festzulegen. Für diese explizite Markierung wird ein Beschreibungsmittel benötigt, daß möglichst unabhängig von dem zugrundeliegenden Modellelement ist und sich dadurch universell einsetzen läßt. Daneben sollten Variationspunkt und Varianten gut als solche erkennbar sein, weswegen Stereotypen als Erweiterungsmechanismus zu wählen sind.

Die Auszeichnung der Varianten wird etwas komplexer, da sich einem Variationspunkt theoretisch eine beliebige Anzahl von Varianten zuordnen läßt. In einem eigenen Metamodell würde der Variationspunkt als Namensraum beschrieben werden, in dem sich die dazugehörigen Varianten befinden. Unter Einsatz der UML-Erweiterungsmechanismen besteht ein akzeptabler Kompromiß darin, jede Variante in einem separaten Modellelement zu modellieren und eindeutig einem Variationspunkt zuzuordnen. Falls sich für den Namen einer Variante dabei keine natürliche Lösung findet, kann der Name des Variationspunkt-Modellelementes um einen künstlichen Teil (zum Beispiel _A, _B, ...) ergänzt werden.

Der Vorteil dieser Darstellung ist die semantische Verträglichkeit des Variationspunktes mit der UML, das heißt, es werden keine künstlichen Modellelemente mit völlig andersartiger Semantik (wie für Klassen als Merkmale) geschaffen.

Mit den Stereotypen für Variationspunkt und Varianten werden die Bestandteile eines Variationspunktes explizit als solche gekennzeichnet und im Modell beschrieben. Nachfolgend wird die für diese Lösung notwendige Verbindung des Variationspunktes mit den Varianten diskutiert.

7.3.1.3 Realisierung von Variationspunkten

Um die entsprechenden Varianten eindeutig einem Variationspunkt (Vp) zuzuordnen, ist eine entsprechende Verbindung erforderlich. Andererseits sind für die Einbindung der Varianten verschiedene Techniken möglich, die auch einen Einfluß auf die Darstellung der Modellierung haben. Deswegen lassen sich im Prinzip zwei Zustände bei der Beschreibung eines Variationspunktes feststellen: einmal in einer allgemeinen, technologieunabhängigen Beschreibung und zum anderen in der konkreten Darstellung einer Implementierungstechnik. Im zweiten Zustand sollte zweckmäßigerweise die Notationsform gewählt werden, die bereits durch UML für die jeweilige Technologie festgelegt ist. Damit verbleibt noch die Frage, welches Mittel am besten zur Beschreibung der Beziehung zwischen Variationspunkt und Variante geeignet ist, wenn die Implementierung noch nicht festgelegt wurde (zum Beispiel in Analysemodellen).

Prinzipiell geeignet sind die Relationen aus dem UML-Metamodell und alternativ TaggedValues. Letztere besitzen jedoch keine Möglichkeit zur grafischen Darstellung des Zusammenhangs zwischen Variationspunkt und Variante. Demzufolge verbleibt die Entscheidung für eine geeignete Spezialisierung von Beziehungen (`Relationship`):

- Generalisierungen beschreiben eine Vererbungs-Semantik, die nur für bestimmte Implementierungstechniken — bei Schnittstellen oder abstrakten Klassen — zutreffend ist. Für andere dagegen ist dies irreführend - so besteht für die ebenfalls mögliche Verwendung von Konfigurationseinstellungen oder der Generierung der Varianten keine Spezialisierungsbeziehung und assoziiert folglich eine falsche Semantik.

- Assoziationen sind denkbar, müssen jedoch in der Multiplizität bedeutend eingeschränkt werden, um den 1:n-Beziehung zwischen Variationspunkt und Varianten zu reflektieren. Außerdem ist eine Anpassung der semantischen Bedeutung erforderlich.

- Dependencies sind prinzipiell im Sinne einer client-supplier-Semantik gerichtet, was auf die Beziehung zwischen Variante und Variationspunkt übertragen werden kann. Daneben bezeichnen Dependencies eine Abhängigkeit, die im Falle einer Variante durchaus gegeben ist.

Dependencies entsprechen demzufolge der Semantik der Beziehung zwischen Variante und Variationspunkt am besten und sollen bei nicht spezifizierter Implementierungstechnik für die Beziehung genutzt werden. Von der semantischen Vorbelegung nach UML sind sie ebenfalls am meisten unabhängig von einer konkreten Realisierung.

Zur Darstellung der technischen Realisierung eines Variationspunktes sind die folgenden Modellierungselemente aus UML denkbar:

Variationstechnik	Verbindung Variante → Vp in UML
Vererbung (*inheritance*)	Generalisierungsbeziehung [3-88ff]
Erweiterungen (*extensions*)	Schnittstellen und Vererbung [3-53f]; Aggregation mit `0..`-Kardinalität [3-70ff]
Konfiguration	Zuordnung zu Artifakten [2-19f]; eventuell in Deployment-Diagrammen [3-177ff], über Attributbelegung (in Kommentaren)
Schablonen (*templates*)	parameterisierte Klasse [3-55f]
Generierung	parameterisierte Klassen [3-55f]; mittels Artifakten [2-19f]

In eckigen Klammern ist jeweils die Seitenzahl der dazugehörigen Beschreibung in der UML-Spezifikation [OMG01a] angegeben.

In SPLIT (vergleiche Abschnitt 4.3.4) wurde abweichend der Begriff des Variabilitätsmechanismus verwendet und in *insert*, *extend* und *parameterize* unterschieden. Dieser bezeichnet die Erweiterungstechnik, die für den Variationspunkt genutzt wird. Da in der vorstehenden Notation das Modellelement mit der Variabilität und der Variationspunkt nicht getrennt werden (da der Vp kein explizites Modellelement ist), werden diese Variabilitätsmechanismen nicht explizit modelliert, sondern durch die Multiplizität und die zur Verbindung respektive Realisierung genutzte Technik beschrieben. Bei Bedarf kann der Variabilitätsmechanismus problemlos mit einem eigenen TaggedValue oder durch andere Beschreibungsmittel explizit dargestellt werden.

Der Vorteil dieser Verschmelzung von Variationspunkt und dem ursprünglichen Modellelement mit der Variation liegt darin, daß zum einen kein zusätzliches Metaelement für den Variationspunkt selbst benötigt wird, und andererseits diese Darstellung für alle Modellierungsphasen von der Analyse bis zum Design/Implementierung verwendet werden kann.

7.3.1.4 Auswahl der Varianten

Die Bindung des Variationspunktes beschreibt den Prozeß der Auswahl der Varianten, die für diesen Variationspunkt genutzt werden.

Dabei gibt es zwei Klassen von Bedingungen für die Auswahl von Varianten: Auswahlregeln durch den Variationspunkt und zusätzliche Abhängigkeiten, die erfüllt werden müssen. Letztere können dabei zwischen den Varianten selbst und in Wechselwirkung mit anderen Systemelementen auftreten und werden im folgenden Abschnitt näher untersucht. Die Angabe der Anzahl der Varianten, die gleichzeitig ausgewählt werden können, stellt eine Zusammenfassung der einzelnen Auswahlregeln dar. Um eine universellere Modellierung zu ermöglichen, ist eine Angabe analog der Multiplizitäten oder Kardinalitäten von Assoziationen in UML sinnvoll. Gegenüber den Variationspunkten in RSEB oder SPLIT ergeben sich damit umfangreichere Ausdrucksmöglichkeiten, die die Beschreibungsmittel der UML ausnutzen. Dazu kann eine Selektionsbeschreibung als TaggedValue vom Typ `Multiplicity` angegeben werden und verwendet damit bereits definierte UML-Strukturen. Diese ist zweckmäßigerweise als Attribut des Variationspunktes anzugeben und bestimmt die Kardinalität der ausgewählten Varianten, wenn der Variationspunkt gebunden wird.

Neben dieser allgemeinen Bestimmung lassen sich für die Auswahl einer Variante auch konkrete Bedingungen angeben, die bei entsprechender Spezifikation zur Automatisierung der Selektion genutzt werden können. Diese Bedingungen sind für jede Variante einzeln zu bestimmen und müssen natürlich die Selektionsregeln des Variationspunktes berücksichtigen. Da eine derartige Festlegung nicht immer möglich sein wird (zum Beispiel in früheren Entwicklungsphasen), kann diese Bedingung nicht immer angegeben werden. Die Bedingung muß bei der Auswertung (*evaluation*) einen Wahrheitswert ergeben: bei `true` wird die Variante ausgewählt, bei `false` entsprechend nicht.

Die Formulierung sollte im Interesse einer korrekten und eindeutigen Darstellung möglichst formal erfolgen. Dazu bieten sich zum einen mathematische Ausdrücke an, zum anderen existiert mit der OCL bereits eine formale Spezifikationssprache, die auch für diesen Zweck verwendet werden kann: Bei der schriftlichen Formulierung wird im entsprechenden Attribut der Variante ein Ausdruck (formale Definition von `expression` in EBNF auf [OMG01a, Seite 6-97]) angegeben, der als Ergebnistyp `Boolean` ergeben muß. Zur sinnvollen Anwendung ist dazu die Integration der Merkmalsmodellierung erforderlich, so daß die Anwesenheit von Merkmalen als logische Variable (`true` bei Merkmalsanwesenheit, `false` sonst) in den Ausdruck untergebracht werden kann (siehe Abschnitt 7.6.2).

Um analog der Angabe von Constraints in UML einen möglichst großen Anwendungsbereich zu erschließen, soll die Verwendung der OCL nur empfohlen, jedoch nicht festgeschrieben werden.

7.3.1.5 Wechselwirkungen zwischen Varianten

Neben der direkten Zuordnung der Varianten zum Variationspunkt und den Auswahlregeln kann es auch zusätzliche Beziehungen zwischen den Varianten geben. Dabei handelt es sich in erster Linie um Abhängigkeiten oder Konfliktsituationen. Für deren modellierende Beschreibung sollten zusätzliche Möglichkeiten vorgesehen werden. Dazu eignen sich aufgrund ihrer semantischen Beschreibung Instanzen der Metaklasse `Dependency` am besten. Im Gegensatz zum Abschnitt 7.2.1.6 sollen hier nur harte Abhängigkeiten — *requires* und *mutex* — spezifiziert werden. Im Interesse einer übersichtlichen Modellierung scheint es empfehlenswert, die explizite grafische Darstellung solcher Wechselwirkungen nur wenn unbedingt notwendig einzusetzen und möglicherweise über die Zuordnung zu Merkmalen zu modellieren (siehe Abschnitt 7.6).

Daneben ergeben sich aus der Weiterentwicklung eines Softwaresystems (siehe Abschnitt 6.2.4) weitere Beziehungen. Zum Beispiel kann eine Variante durch eine effizientere Implementierung ersetzt werden, diese kann jedoch aufgrund eines höheren Ressourcenverbrauchs nicht in allen abgeleiteten Systemen eingesetzt werden. In diesem Fall ist eine explizite Modellierung von evolutorischen Beziehungen sinnvoll. Dazu soll an dieser Stelle eine entsprechende Klassifizierung solcher Beziehungen als „durch Evolution entstanden“ ermöglicht werden. Die genaue Beschreibung der Beziehung kann durch den Namen der Abhängigkeit erfolgen, oder durch Definition weiterer Stereotypen zur Klassifizierung, die von dem definierten Stereotyp abgeleitet werden können. Denkbar sind zum Beispiel die Varianten „Ersetzung“, „Erweiterung“ und „Aufteilung“.

7.3.1.6 Mehrere Variationspunkte in einem Modellelement

Am später folgenden Beispiel für einen Variationspunkt wird deutlich sichtbar, daß es auch mehr als einen Variationspunkt pro Modellelement geben kann. Dies gilt besonders dann, wenn die Variationen im Modell klar voneinander getrennt werden sollen — die Variationspunkte also unabhängig voneinander sind. Eine alternativ denkbare Beschreibung, die jede mögliche Kombination von Varianten in einer Variante modelliert, verwischt diese Trennung und stellt nur eine Kompromißlösung dar. Da es sich bei der Modellierung in UML ebenfalls um eine von der Implementation unabhängige Beschreibung handelt, kann die mögliche technische Nicht-Realisierbarkeit multipler Variationspunkte kein Argument sein. Demzufolge sollten mehrere Variationspunkte in einem Modellelement durch die Notation unterstützt werden.

Die Beschreibung von mehr als einem Variationspunkt wirft kleine Probleme bezüglich der grafischen Beschreibung auf. Gleichnamige, jedoch unterschiedlich belegte Attribute (für jeweils einen Vp) werden durch die seit UML Version 1.4 möglichen Multiplizitäten für Werte von TaggedValues möglich. In der grafischen Darstellung dagegen läßt sich mit den gegebenen Mitteln für Beziehungen keine eindeutige Zuordnung erreichen. Allerdings kann die Menge der zu einem Variationspunkt gehörenden Varianten durch ein verbindendes Element gekennzeichnet werden: mit einem Constraints oder mit Kommentaren. Da damit jedoch keine Einschränkung modelliert wird, sollten dazu in erster Linie Kommentare verwendet werden.

Die formale Zuordnung erfolgt durch entsprechende TaggedValues in Variante und Variationspunkt, mit denen die Beziehung eindeutig hergestellt wird. Dazu erhält jeder Variationspunkt einen Namen zur Identifikation, wobei der Index des Namensfeldes als Identifikation und als Index für die TaggedValues dient und auf die sich ein entsprechendes TaggedValue in der Variante bezieht. Der Name des Variationspunktes kann auch in dem Kommentar erscheinen, der die zugeordnete Variantenmenge kennzeichnet.

Wird nur ein Variationspunkt in einem Element modelliert, kann diese Zuordnung der Variante-Vp-Beziehung entfallen. Für diesen Fall läßt sich die Erweiterung auch in UML 1.3 einsetzen, indem die Multiplizität der TaggedValues ignoriert wird. Dieses Darstellungsschema — einfache Zuordnungen graphisch, komplexere Beziehungen durch formale Zuordnung — entspricht dem Schema, einen

Stereotyp grafisch durch ein Icon darzustellen, dieses Icon jedoch bei mehreren Stereotypen in einem Modellelement zu unterdrücken [OMG01a, Seite 3-34].

7.3.1.7 Attribute von Variationspunkten und Varianten

Zur Beschreibung der Eigenschaften von Variationspunkten werden bereits in der Literatur eine ganze Menge Attribute verwendet. Diese werden nachfolgend in notwendig und optional eingeordnet.

Die konkrete Verwendung der optionalen Eigenschaften ist abhängig von dem konkreten Einsatzzweck, zum Beispiel dem verwendeten Entwicklungsprozeß.

Die notwendigen Attribute sind in jedem Variationspunkt vorhanden und je nach Stand der Entwicklung konkret belegt oder unspezifiziert:

- der Bindezeitpunkt (*binding time*) bestimmt den Zeitpunkt, zu dem Auswahl der Varianten erfolgt.
- erlaubte Multiplizität der Varianten nach der Auswahl
- ein Name erlaubt die explizite Bezeichnung des Variationspunktes und ist bei mehreren Variationspunkten in einem Modellelement als Ansatzpunkt für die Zuordnung der Varianten am besten geeignet.

Daneben ergeben sich noch weitere Möglichkeiten, um Variationspunkte detaillierter zu beschreiben. Dazu gehören unter anderem:

- Der Bindemodus (*binding mode*) stellt eine weitere, implementierungstechnische Qualifizierung der *binding time* dar (siehe Seite 34).
- Der Zustand (*state*) beschreibt den Bearbeitungsstand im Entwicklungsprozeß — ob dieser Variationspunkt bereits existiert (`implicit`), in diesem Modell identifiziert wurde (`designed`) oder bereits gebunden ist (`bound`).
- Die Beschreibung liefert die Gründe für die Modellierung dieses Variationspunktes.
- die Quellenbeschreibung gibt an, auf wessen Anliegen (*stakeholder*) oder aus welchem existierendem System (*source*) die Variabilität identifiziert wurde.

Die Attribute *existence* und *binding occurence* (siehe Abschnitt 4.3.4) werden durch die Multiplizität beschrieben. Die Bedeutung und Wertbelegung von Bindezeitpunkt, Bindemodus und Entwicklungszustand wird in Abschnitt 6.1.4 erörtert.

Für die Varianten wurde bereits die folgenden notwendigen Attribute identifiziert:

- die Zuordnung zu einem Variationspunkt, wenn noch weitere Variationspunkte in dem gleichen Modellelement enthalten sind
- die Entscheidungsregel enthält einen Ausdruck, der über die Auswahl der Variante entscheidet

7.3.2 Notation

Variationspunkte dienen der expliziten Modellierung von Varianten eines Modellelementes. Sie bestehen aus einem Modellelement, in dem die Variabilität auftritt — dem Variationspunkt — und den Varianten, welche die Variabilität (unterschiedlich) umsetzen. Dabei wird jede Variante durch ein eigenes Modellelement repräsentiert. Eine Variante ist stets genau einem Variationspunkt zugeordnet.

7.3.2.1 Allgemeines

Die Notation von Variationspunkten erfolgt durch die Markierung der Variationspunkte und Varianten mit Stereotypen. Diese werden für `GeneralizableElement` spezifiziert. Als Variationspunkt wird das Modellelement gekennzeichnet, das die Variabilität enthält — also den Ansatzpunkt für die Varianten bildet. Die Varianten bilden die unterschiedlichen Ausprägungen dieser Variabilität und sind einem Variationspunkt eindeutig zugeordnet.

Die Zuordnung der Varianten zum Variationspunkt erfolgt in Abhängigkeit von der zur Realisierung des Variationspunktes genutzten Technik mit der jeweils spezifischen Notation. Wurde die zu verwendende Technik noch nicht bestimmt, werden Dependency-Abhängigkeiten zur Darstellung der Beziehung verwendet.

Die Zuordnung mehrerer Variationspunkte zu einem Modellelement ist möglich — dabei sind die beschreibenden Attribute (in TaggedValues) mehrfach vorhanden, und zwar in der Zahl der Variationspunkte. Die Zuordnungsbeziehung der Varianten zu einem der Variationspunkte kann in der grafischen Darstellung durch entsprechende Verbindung mit einem Kommentar unterstützt werden. Daneben sollten zwischen Variante und Variationspunkt sowie zwischen den Varianten eines Variationspunktes keine weiteren Beziehungen bestehen.

Die Abhängigkeiten zwischen Variationspunkten oder Varianten werden in zwei Gruppen unterteilt: *hard constraints* und evolutionäre Beziehungen. Für die Modellierung von ersteren sind der wechselseitige Ausschluß «mutex» und die Abhängigkeit «requires» vorgesehen. Beziehungen, die durch die Evolution des Systems entstehen, werden mit «evolution» gekennzeichnet und können je nach Bedarf weiter unterteilt werden, zum Beispiel in Ersetzung, Erweiterung und Dekomposition (Aufteilung in mehrere Einheiten).

7.3.2.2 Stereotypen

Stereotyp	isAbstract	baseClass	Parent	Beschreibung
variationPoint	false	`GeneralizableElement`	–	Variationspunkt
variant	false	`GeneralizableElement`	–	Variante eines Variationspunktes
constraint	true	`Dependency`	–	Beziehungen zwischen Varianten
requires	false	`Dependency`	constraint	Abhängigkeit
mutex	false	`Dependency`	constraint	wechselseitiger Ausschluß
evolution	false	`Dependency`	–	Weiterentwicklung

Die Tabelle spezifiziert die erforderlichen Stereotypen für Variationspunkte. „Parent“ gibt den übergeordneten Stereotypen an, „baseClass“ und „isAbstract“ bestimmen die konkrete Wertebelegung der Meta-Attribute: *baseClass* bestimmt die Metaklassen der Modellelemente, auf die ein Stereotyp angewendet werden darf; ist *isAbstract* mit `true` belegt, kann der jeweilige Stereotypname nicht in Modellen verwendet werden (die damit bezeichnete virtuelle Metaklasse, zum Beispiel `feature`, ist abstrakt).

Die Verbindung der Varianten mit dem Variationspunkt erfolgt über Dependency-Beziehungen — mit dem Variationspunkt in *supplier*-Rolle (Pfeilspitze) — oder alternativ mit dem technologiespezifischen Beschreibungsmittel; dabei ist jede Variante eindeutig genau einem Variationspunkt zuzuordnen.

Nachfolgend wird die mit den Stereotypen verbundene spezifische Semantik beschrieben, die für das damit ausgezeichnete Modellelement gilt:

- **«variationPoint** kennzeichnet einen Variationspunkt, also ein Modellelement mit Variabilität, die durch verschiedene Ausprägungen beschrieben wird.

- **«variant»** kennzeichnet eine Variante eines Variationspunktes, also eine konkrete Ausprägung der Variabilität eines Modellelementes.
- **«constraint»** kennzeichnet Beziehungen zwischen Varianten, die immer (bei Instanziierung der beteiligten Modellelemente) erfüllt sein müssen.
- **«requires»** kennzeichnet eine Abhängigkeit. Ist diese *requires*-Abhängigkeit gerichtet, so kann die Variante in der *client*-Rolle (Pfeilende) nur genutzt werden, wenn die Variante in der *supplier*-Rolle (Pfeilspitze) ebenfalls benutzt wird (siehe auch [OMG01a, Seite 3-92]). Bei einer ungerichteten Abhängigkeit müssen beide Varianten oder keine von beiden ausgewählt werden.
- **«mutex»** bezeichnet den wechselseitigen Ausschluß von Modellelementen. Von den beteiligten Elementen darf höchstens eines ausgewählt werden. Die Semantik dieser Beziehung entspricht dem in UML definierten `{xor}`-Constraint für Assoziationen.
- **«evolution»** bezeichnet Beziehungen zwischen Modellelementen, die durch Weiterentwicklung des Systems (Evolution) entstanden sind (zu den Rollen von Dependencies siehe [OMG01a, Seite 3-92]).

Speziell die für Dependency-Beziehungen definierten Stereotypen sind prinzipiell zwischen allen Modellelementen anwendbar, jedoch speziell für die Beschreibung von Beziehungen zwischen Varianten vorgesehen. Sinnvoll ist ebenfalls die Modellierung von Abhängigkeiten einer Variante zu anderen Modellelementen, die nicht an dem Variationspunkt beteiligt sind. Die mit den *hard constraints* spezifizierten Regeln müssen bei der Auswahl der Variante beziehungsweise Instanziierung des jeweiligen Modellelementes erfüllt sein — zum Beispiel kann die Benutzung oder Auswahl einer Variante die Existenz eines anderen Modellelementes voraussetzen (= requires-Abhängigkeit).

Zur Unterstützung der grafischen Darstellung werden die folgenden Icons für die Stereotypen vorgeschlagen:

Stereotyp	Icon-Beschreibung	Icon-Darstellung
variationPoint	gefüllter Kreis	●
variant	gefüllter, etwas kleinerer Kreis mit Index-Buchstaben	●n

7.3.2.3 TaggedValues

Schlüsselwort	Stereotyp	Typ	Multiplizität
VpName	variationPoint	`String`	`0..*`
BindingTime	variationPoint	`BindingtimeEnum` (Enumeration: { development, build, installation, start-up, runtime })	`0..*`
Multiplicity	variationPoint	`Multiplicity`	`0..*`
VariationPoint	variant	`Integer`	`1`
Condition	variant	`BooleanExpression`	`0..1`

Auf dem Tag `VariationPoint` wird zusätzlich der folgende Constraint definiert: „gültiger Index der Werteliste von «variationPoint» `.VpName`“. Die konkrete Multiplizität der Werte der Tags für «variationPoint» ist gleich der Anzahl der Variationspunkte, respektive der Anzahl disjunkter Mengen von zusammengehörenden Varianten, und für alle Variationspunkt-Tags gleich.

Die folgenden Name-Wert-Paare sind notwendige Attribute des Variationspunktes: Mit `Multiplicity` wird die Kardinalität der Varianten eines Variationspunktes bestimmt — das heißt, wie viele Varianten gleichzeitig ausgewählt werden können. `Condition` beschreibt die Auswahlbedingung für

jede einzelne Variante und hat einen Wahrheitswert (`true` ⇒ Variante wird ausgewählt, sonst nicht) als Ergebnis. Die Formulierung kann als OCL-Ausdruck, mathematische Bedingung oder in Textform erfolgen, wobei die Nutzung von OCL zu empfehlen ist. Die Wertebelegung von `BindingTime` entscheidet, wann die Variantenauswahl zu erfolgen hat und besitzt die folgenden Bedeutungen (siehe Abschnitt 6.1.4.1):

- `development`: während der Entwicklung (Ableitung, Design, Implementierung) des Systems
- `build`: beim Übersetzen (Kompilieren) des Programmcodes
- `installation`: bei der Installation auf dem Zielsystem
- `start-up`: beim Starten/Laden des Programms
- `runtime`: während der Laufzeit

Zusätzlich sind die folgenden optionalen TaggedValues für Variationspunkte möglich und können bei Bedarf angegeben werden:

Schlüsselwort	Stereotyp	Typ	Multiplizität
BindingMode	variationPoint	`BindingmodeEnum` (Enumeration: { static, changeable, dynamic })	`0..*`
State	variationPoint	`DevelopmentstateEnum` (Enumeration: { implicit, designed, bound })	`0..*`
Description	variationPoint	`String`	`0..*`
Stakeholder	variationPoint	`String`	`0..*`
Source	variationPoint	`String`	`0..*`

Diese Tags können in Abhängigkeit von den Erfordernissen verwendet werden. Die Dauerhaftigkeit der Bindung der Varianten (siehe Abschnitt 6.1.4.2) wird mit dem Bindemodus `BindingMode` beschrieben. Der Status `State` gibt an, in welchem Entwicklungszustand sich der Variationspunkt befindet (nach Abschnitt 6.1.4.3): wurde die Variation bereits in höheren Abstraktionsebenen identifiziert, ist der Variationspunkt implizit vorhanden (`implicit`), mit `designed` wurde er auf der modellierten Abstraktionsebene identifiziert und bei `bound` wurde der Variationspunkt bereits gebunden, die Variantenauswahl steht also schon fest. Die Beschreibung `Description` kann der detaillierten Information über den Variationspunkt dienen. Diese kann erforderlich sein, wenn der Variationspunkt nur ein Teil des bezeichneten Modellelementes beeinflußt und nicht in der Element-Beschreibung dokumentiert werden kann. Die Interessensbeteiligten (`Stakeholder`), die an dem Vp beteiligt sind und die Quellenbeschreibung `Source` ergänzen die Beschreibung des Variationspunktes um die Angabe, wofür und wo (zum Beispiel in existierenden Systemen) der Variationspunkt identifiziert wurde.

7.3.2.4 Constraints

Die folgenden Constraints beschreiben Einschränkungen bei der Anwendung der Stereotypen des virtuellen Metamodells (die *well-formedness rules* der Erweiterung). Die Regeln beziehen sich jeweils auf das stereotypisierte Modellelement.

Zusätzliche Operationen:

[1] Die Operation **isStereotyped** prüft, ob das Objekt den gegebenen Stereotyp besitzt.

```
context ModelElement def:
  let isStereotyped (st_name : String) : Boolean =
    self.stereotype->exists( s | s.name=st_name )
```

[2] Das Pseudo-Attribut **allStereotypes** enthält alle Stereotypen, mit denen ein Modellelement direkt oder indirekt stereotypisiert ist.

```
context ModelElement def:
  let allStereotypes : Set(Stereotype) =
    self.stereotype->union(
      self.stereotype.generalization.parent.allStereotypes
    )
```

[3] Die Operation **isConstraintRelation** bestimmt, ob die Beziehung als Constraint stereotypisiert ist.

```
context Relationship def:
  let isConstraintRelation() : Boolean =
    self.allStereotypes->exists (s | s.name='constraint')
```

[4] Das Pseudo-Attribut **allVariants** enthält die Menge aller Modellelemente, die als Varianten mit dem Objekt (welches ein Variationspunkt-Modellelement sein sollte) verbunden sind.

```
context GeneralizableElement def:
  let allVariants : Set(GeneralizableElement)=
    if self.isKindOf(Classifier) then
      self.oclAsType(Classifier).oppositeAssociationEnds
      ->select(
        ae | ae.isStereotyped('variant')
      )->union(
        self.specialization.child->select(
          ge | ge.isStereotyped('variant')
        )
      )
    else self.specialization.child->select(
      ge | ge.isStereotyped('variant')
    ) endif
    ->union(
      self.supplierDependency.client->select(
        me | me.oclIsKindOf(GeneralizableElement) and
        me.isStereotyped('variant')
      )
    )
```

[5] Das Pseudo-Attribut **vpCount** enthält die Anzahl der Variationspunkte in diesem Modellelement.

```
context GeneralizableElement def:
  let vpCount : Integer =
    self.taggedValue->select( tv | tv.name='VpName' )
    ->size()
```

variationPoint (Core::GeneralizableElement)

[1] Ein Variationspunkt kann nicht sich selbst als Variante zugeordnet sein.

```
self.allVariants->excludes( self )
```

[2] Die Werteliste aller Variationspunkt-TaggedValues enthält so viele Werte wie das Modellelement Variationspunkte hat oder ist leer.

```
self.taggedValue->select(
  tv.name='VpName' or tv.name='Multiplicity' or
  tv.name='BindingTime'
)->forAll( tv | tv.dataValue.isEmpty() or
  tv.dataValue->size()=self.vpCount
)
```

[3] Besitzt ein Modellelement mehr als einen Variationspunkt, muß das Tag `VariationPoint` in jeder Variante mit einer gültigen Indexkennzahl belegt sein.

```
self.vpCount>1 implies
self.allVariants->forAll( ge |
  ge.taggedValues->select( tv | tv.name='VariationPoint')
  ->forAll( tv | tv.dataValue->size()=1 and
    tv.dataValue->forAll( dv | dv>=0 and dv<self.vpCount )
  )
)
```

variant (Core::GeneralizableElement)

[1] Jede Variante ist über genau eine Beziehung mit einem als Variationspunkt gekennzeichneten Modellelement verbunden, andere Beziehungen sind nicht erlaubt oder im Falle von Dependencies als «constraint» gekennzeichnet.

```
let depVps : Set(ModelElement) =
  self.clientDependency->select( d |
    d.supplier.isStereotyped('variationPoint')
  )
self.clientDependency->reject( d |
  d.supplier.isStereotyped('variationPoint')
)->forAll( d | d.isConstraintRelation() ) and (
  ( self.depVps->notEmpty() implies self.depVps->size()=1
  ) xor (
    self.generalization->notEmpty() implies
    self.generalization.size()=1 and
    self.generalization->forAll( g |
      g.parent.isStereotyped('variationPoint')
    )
  ) xor (
    self.allOppositeEnds->notEmpty() implies
    self.allOppositeEnds->size()=1 and
    self.allOppositeEnds->forAll( ae |
      ae.isStereotyped('variationPoint')
    )
  )
)
```

[2] Ein Variante kann nicht sich selbst als Variationspunkt zugeordnet sein.

```
self.isStereotyped('variationPoint') implies
self.taggedvalue->forAll( tv |
  tv.name='Variationpoint' implies tv.dataValue->excludesAll(
    self.taggedvalue->select( tv | tv.name='VpName' )
  )
)
```

7.3.2.5 UML-Modell der Erweiterung

Abbildung 7.3 stellt die grafische Spezifikation der Erweiterung für Variationspunkte in UML-Notation dar. Aus Komplexitätsgründen sind die mit den Stereotypen assoziierten Constraints nicht mit dargestellt.

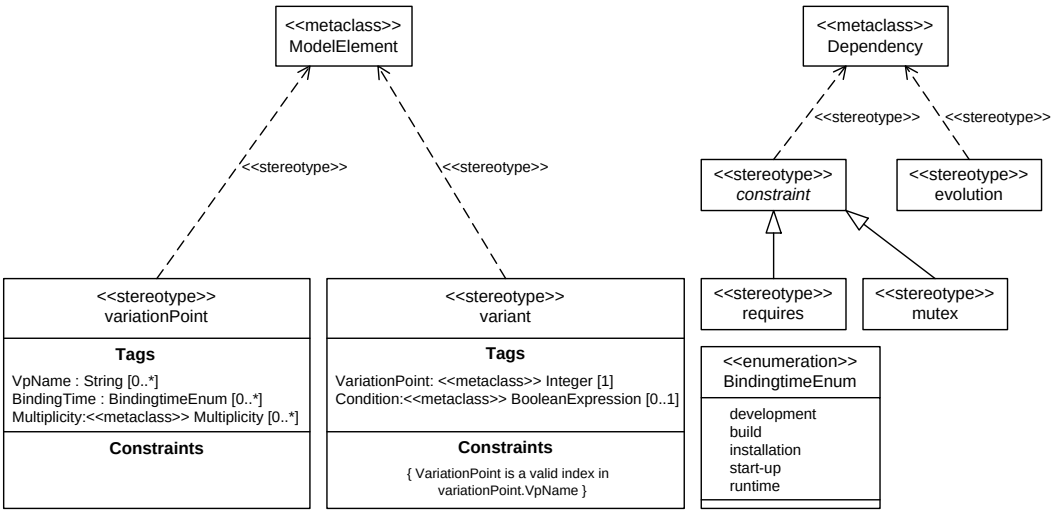


Abbildung 7.3: UML-Modell der Erweiterung für Variationspunkte

7.3.3 Beispiel für Variationspunkte

Um die Notation etwas zu veranschaulichen, soll an dieser Stelle ein kleines Beispiel dargestellt werden.

Dazu wird in Abbildung 7.4 eine Anwendung des *Strategy*-Patterns nach [Gamma95] als Variationspunkt dargestellt. Die Variabilität besteht in diesem Fall beim ersten (Index `0`) Variationspunkt in der Auswahl des Sortieralgorithmus, der beispielhaft in drei Varianten realisiert wurde. Da hierfür noch keine Implementierungstechnik festgelegt wurde, werden die Varianten mittels Dependency-Beziehungen mit dem variierenden Modellelement verbunden.

Zusätzlich läßt sich noch ein zweiter Variationspunkt (Index `1`)in der Klasse `sort` bestimmen, der mittels Parametrisierung realisiert wird und für den zu sortierenden Datentyp variiert.

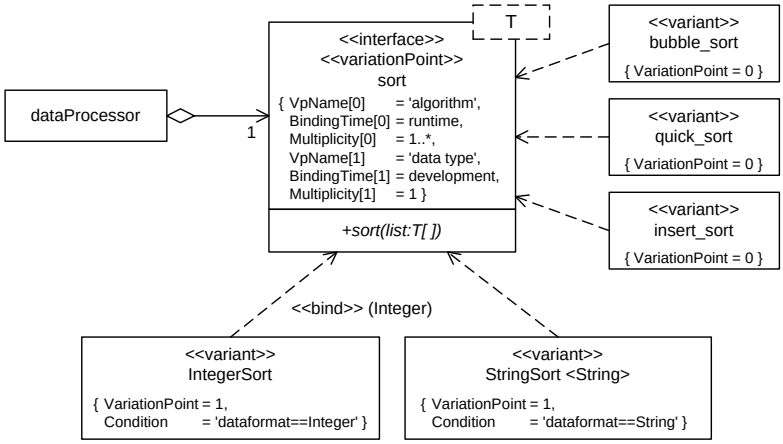


Abbildung 7.4: Beispiel für Variationspunkte

7.4 Optionale Modellelemente

Das Konzept der Variationspunkte beschreibt Variabilität mit der Darstellung von verschiedenen Alternativen. Mit dem Spezialfall einer einzigen Variante mit `0..1`-Multiplizität des Variationspunktes beschreibt man dabei optionale Elemente, die nicht immer anwesende Systembestandteile modellieren. Dafür sind jedoch mindestens zwei verschiedene Modellelemente und eine Verbindung zwischen diesen erforderlich. Diese Darstellung enthält folglich unnötigen Zusatzaufwand — für das Modellelement mit dem Variationspunkt — und ist nicht immer anwendbar.

Daneben besteht ein semantischer Unterschied: Variationspunkte modellieren in der Regel eine Auswahlmöglichkeit zwischen verschiedenen Alternativen, optionale Elemente bieten nur die Auswahl zwischen Einbindung in das System und Nichtexistenz.

Aus diesem Grund wird das Konzept der Variationspunkte durch das Konzept der optionalen Elemente ergänzt. Mit diesem lassen sich beliebige Modellelemente als optional kennzeichnen, wobei die Anwesenheit durch eine Bedingung beschrieben werden kann. Optional ist so zu verstehen, daß das entsprechende Modellelement nur unter bestimmten Auswahlbedingungen (zum Beispiel bei der Implementierung eines bestimmten Merkmales) im System vorhanden ist — also selektionsspezifisch.

Das Verhalten der optionalen Elemente ist in vielen Punkten ähnlich dem der Variationspunkte — theoretisch läßt sich das eine mit Hilfe des anderen emulieren. Aus diesem Grund greifen die nachfolgenden Betrachtungen teilweise auf Erkenntnisse aus der Diskussion der Variationspunkte zurück.

Gegenüber den Variationspunkten bieten die optionalen Elemente jedoch einen zusätzlichen Vorteil: sie lassen sich intuitiv auch in der Verhaltensmodellierung, in Sequenz- und Kollaborationsdiagrammen einsetzen, wo Variationspunkte mit einer wesentlich größeren Komplexität behaftet sind. Die Problematik von Variabilität in der Verhaltensmodellierung wird detailliert im Abschnitt 7.5 diskutiert.

7.4.1 Beschreibung der Darstellung

Prinzipiell kann allen Modellelemente der UML eine optionale Existenz zugeordnet werden. Entsprechend erfolgt die Spezifikation des kennzeichnenden Stereotypen allgemein für `ModelElement`.

Da mit der Optionalität nur eine weitere — allerdings prinzipielle — Eigenschaft des so ausgezeichneten Elementes beschrieben wird, gibt es hierbei keine aus der Optionalität resultierenden Beziehungen zu anderen Elementen. Allerdings wirkt sich die Anwesenheit des Modellelementes auf die

Anwesenheit davon abhängiger verknüpfter Elemente aus. Das betrifft insbesondere verbindende Elemente — zum Beispiel Relationen und Transitionen — und Modellelemente, deren Funktionsfähigkeit oder Existenz mit der des optionalen Elementes verbunden ist (beispielsweise Attribute und Operationen einer optionalen Klasse).

Diese Abhängigkeiten sind bei der Modellierung entsprechend mit zu modellieren. Dies kann zum Teil implizit erfolgen — zum Beispiel für verbundene Relationen — oder durch explizite Modellierung — zum Beispiel bei abhängigen Modellelementen. Die Modellierung der Abhängigkeiten kann durch die Benutzung der bereits für Merkmale und Variationspunkte (Abschnitt 7.3.1.5) verwendeten *hard constraints* explizit in grafischer Form oder textuell durch Bedingungen für die Existenz / Auswahl erfolgen.

7.4.1.1 Selektionsregeln

Zur formalen Beschreibung der Auswahl und zur Unterstützung automatisierter Verfahren sollte eine formale Spezifikation der Anwesenheitsbedingung analog der Auswahlregeln für Varianten (Abschnitt 7.3.1.4) erfolgen. Deren Beschreibung kann mittels mathematischer Ausdrücke, OCL oder in Textform erfolgen.

Wenn ein Modellelement als optional gekennzeichnet wird, sind die davon abhängigen Modellelemente — zum Beispiel beteiligte Relationen, Spezialisierungen, Sub-Zustände, ausgelöste Stimuli — ebenfalls optional. Diese implizite Optionalität ist bei der Modellierung entsprechend zu beachten. Sämtliche betroffene Elemente ebenfalls explizit zu kennzeichnen mindert die Übersichtlichkeit des entsprechenden Modells und sollte nur für ausgewählte Elemente erfolgen — als Empfehlung abhängige Bezeichner, jedoch nicht abhängige Relationen. Die Auswahlregeln der abhängigen Elemente müssen entsprechend konsistent sein und dürfen nicht zu Widersprüchen führen.

7.4.1.2 Weitere Eigenschaften

Um das Verhalten eines optionalen Elementes speziell bei der Auswahl zu modellieren, ist eine entsprechende Dokumentation durch Attribute notwendig, die die Variabilität beschreiben. Diese kann an die Beschreibung von Variationspunkten angelehnt werden und sollte als Minimum die Bindezeit enthalten, die in diesem Fall beschreibt, wann die Anwesenheit des Elementes entschieden wird. Die Diskussion der sinnvollen Werte erfolgte bereits in Abschnitt 6.1.4.1.

Analog zu den Variationspunkten können die in Abschnitt 7.3.1.7 genannten zusätzlichen Attribute ebenfalls für optionale Elemente verwendet werden.

7.4.2 Notation

Optionale Elemente kennzeichnen ein UML-Modellelement als nicht immer vorhanden — in der Regel im Sinne einer Auswahl, die durch ein übergeordnetes Prinzip gesteuert wird.

7.4.2.1 Allgemeines

Zur Auszeichnung optionaler Elemente in UML-Modellen werden ein Stereotyp und damit verknüpfte TaggedValues definiert, die anzeigen, daß das damit ausgezeichnete Elemente nur unter bestimmten Bedingungen vorhanden beziehungsweise nicht vorhanden ist.

Die Beschreibung der Entscheidung über die Anwesenheit kann analog zu den Ausdrucksmöglichkeiten für Constraints als textuelle Beschreibung, logischer Ausdruck oder in OCL erfolgen.

7.4.2.2 Stereotypen

Stereotyp	isAbstract	baseClass	Parent	Beschreibung
optional	false	`ModelElement`	–	optionales Element

Der Stereotyp «optional» kann auf alle UML-Modellelemente angewendet werden, um diese als optional zu kennzeichnen. Dabei ist zu beachten, daß abhängige Modellelemente ebenfalls implizit optional werden.

Zur Unterstützung der grafischen Darstellung wird das folgende Icon für den Stereotyp spezifiziert:

Stereotyp	Icon-Beschreibung	Icon-Darstellung
optional	kleiner leerer Kreis	◯

7.4.2.3 TaggedValues

Schlüsselwort	Stereotyp	Typ	Multiplizität
BindingTime	optional	`BindingtimeEnum`	`0..1`
Condition	optional	`BooleanExpression`	`0..1`

`BindingTime` beschreibt den Zeitpunkt, zu dem die Anwesenheit des Elementes entschieden wird. Die Wertbelegung hat die gleiche Bedeutung wie für Variationspunkten (Seite 65). `Condition` enthält eine Bedingung, die über die Anwesenheit des Elementes entscheidet.

Diese Name-Wert-Paare sind immer anzugeben. Zusätzlich können die optionalen TaggedValues für Variationspunkte (Seite 65) übernommen werden.

7.4.2.4 UML-Modell der Erweiterung

Abbildung 7.5 stellt die grafische Spezifikation der Erweiterung für optionale Elemente in UML-Notation dar.

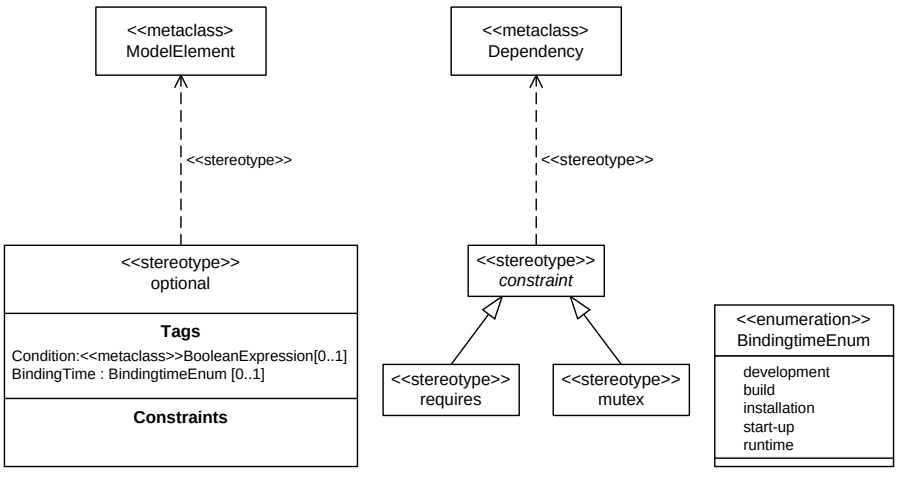


Abbildung 7.5: UML-Modell der Erweiterung für optionale Elemente

7.4.3 Beispiel

Als Beispiel soll an dieser Stelle die am Anfang angedeutete, theoretisch mögliche Abbildung eines Variationspunktes mit Hilfe von optionalen Elementen und vice versa aufgegriffen werden. Dazu ist der im Variationspunkt-Beispiel (Seite 69) modellierte Systemausschnitt in Abbildung 7.6 mit Hilfe von optionalen Modellelementen modelliert worden.

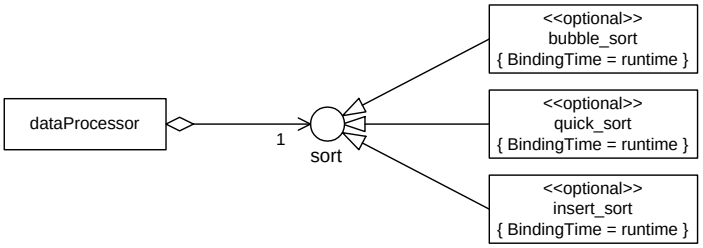


Abbildung 7.6: Beispiel der Anwendung optionaler Elemente

7.5 Verhaltensmodellierung

In diesem Abschnitt wird diskutiert, wie sich die Modellierung von Variationspunkten und optionalen Elementen auf die Modellelemente der Verhaltensmodellierung (*behavioral elements*) in UML anwenden läßt.

Die Modellierung des dynamischen Verhaltens eines Softwaresystems erfolgt in der UML mit Elementen des „*Behavioral Package*“. Dazu gehören Use-Case-Modelle, Sequenzdiagramme und Kollaborationen, State-Chart-Diagramme und die daraus abgeleiteten Aktivitätsdiagramme.

Die entwickelten Konzepte zur Systemmodellierung — Variationspunkte und optionale Modellelemente — sind prinzipiell Erweiterungen des Notationskonzeptes der UML. Allerdings beziehen sie sich in der Regel auf die statische Modellierung und beschreiben Variabilität zur Entwicklungszeit — im Gegensatz zur Verhaltensmodellierung in UML, die Variabilität zur Laufzeit beschreibt.

Nachfolgend wird diskutiert, wie diese Konzepte zur Modellierung von (Entwicklungszeit-) Variabilität in den dynamischen Beschreibungsmitteln der UML genutzt werden können.

Die Verhaltensbeschreibung eines Systems in UML baut zu einem großen Teil auf der zugrundeliegenden statischen Struktur auf, von der sie ihre Grundelemente bezieht. Dementsprechend läßt sich die Variabilität (bei der Entwicklung) nach ihrer Ursache unterteilen: treten durch die unterschiedliche Anwesenheit der beteiligten Elemente Unterschiede auf, oder gibt es zusätzliche explizite Verhaltensunterschiede bei sonst gleichen Beteiligten, deren Ursache auf die Modellierung mehrere Systeme zurückgeführt werden kann. Erstere lassen sich prinzipiell immer auf das statische Modell zurückführen und äußern sich zum Beispiel in der Weise, daß ein Anwendungszweig nur ausgeführt werden kann, wenn die beteiligten Elemente auch im System enthalten sind. Mit den expliziten Verhaltensunterschieden ist eine spezielle Form der Laufzeit-Variabilität gemeint, die erst durch die Modellierung mehrerer Systeme in einem Modell auftritt. Beispielsweise kommt es dazu, wenn die verschiedenen Systeme unterschiedliche Strategien zur Realisierung eines Konzeptes umsetzen (dazu jedoch auf den gleichen Kontextbedingungen aufsetzen).

Um eine Vermischung der Modellierung von Variabilität zur Entwicklungszeit und zur Laufzeit zu vermeiden, sollten die Erweiterungen in der Verhaltensmodellierung sehr bewußt eingesetzt werden. In erster Linie sollte damit die Beschreibung von unterschiedlichem dynamischen Verhalten erfolgen, daß aus der Modellierung mehrerer Systeme in einem Modell resultiert. Weiterhin ist zu beachten,

daß durch die Benutzung der Erweiterungen eine Änderung der Semantik der beteiligten (insbesondere assoziierten) Modellelemente erfolgt, zum Beispiel Ausführungspfade nur bei Existenz aller beteiligten Elemente ausführbar sind. Aus diesem Grund sollten, wenn möglich, derartige Bedingungen explizit modelliert werden.

7.5.1 Use-Case-Diagramme

In Use-Case-Diagrammen wird das Verhalten von Entitäten beschrieben, ohne ihre interne Struktur zu spezifizieren. Dazu werden Akteure (*actors*) und Anwendungsfälle (*use cases*) verwendet, deren Instanzen bei Gebrauch eines Dienstes miteinander interagieren [OMG01a, Seite 2-139ff]. Akteure können Beziehungen nur zu Anwendungsfällen, Subsystemen oder Klassen besitzen, diese sind stets binär. Gemeinsamkeiten zwischen verschiedenen Akteuren können mittels eines übergeordneten (abstrakten) Akteurs generalisiert werden.

Zwischen Anwendungsfällen können ebenfalls binäre Beziehungen bestehen. Gemeinsamkeiten zwischen verschiedenen Anwendungsfällen können laut [OMG01a, Seite 2-148] mit Generalisierungen und mit *include*- und *extends*-Beziehungen beschrieben werden, diese Beziehungen beziehen sich jedoch stets auf Anwendungsfälle einer Entität — zum Beispiele verschiedene Dienste (*services*) eines Subsystems, einer Klasse oder auch des Gesamtsystems.

Variabilität in Use-Case-Modellen kann bereits durch *extension points* und «extends»-Beziehungen beschrieben werden. Diese beschreiben die Erweiterung eines Anwendungsfalles, die bei Erfüllung einer Bedingung ausgeführt wird.

7.5.1.1 Selektion in Anwendungsfällen

Oftmals werden Anwendungsfälle bereits spezifiziert, bevor die realisierenden Elemente modelliert sind. In diesem Fall kann die Existenz eines bestimmten Teiles von Verhalten explizit bereits in Abhängigkeit von übergeordneten Entscheidungskriterien, zum Beispiel Merkmalen, bestimmt werden. Entsprechend ist es sinnvoll, einzelne Use-Cases als optional zu kennzeichnen.

Die Auswahl von unterschiedlichen Varianten zwischen Anwendungsfällen kann bereits ohne Erweiterungen durch «extends»-Beziehungen modelliert werden. Handelt es bei einer solchen Auswahl um Selektions-Variabilität, sollte die zusätzliche Auszeichnung als Variationspunkt erfolgen — der Erweiterungspunkt stellt einen Variationspunkt dar, die eingebundenen Anwendungsfälle beschreiben die Varianten und sind über «extends»-Beziehungen verbunden. Mit dieser expliziten Kennzeichnung ist eine Unterscheidung zu Laufzeit-Variabilitäten gegeben. Variationspunkte lassen sich analog bei der Generalisierung von Anwendungsfällen modellieren, wenn damit Selektions-Variabilität beschrieben wird (zum Beispiel verschiedene Spezialisierungen eines Anwendungsfalles für verschiedene Produkte).

Jeder Anwendungsfall beschreibt eine Sequenz von Aktionen, die einen Dienst (*service*) der Entität realisieren. Die Existenz dieses Dienstes ist natürlich abhängig von der Existenz des realisierenden Elementes, die in der Regel im statischem Modell entschieden wird (durch eine der Erweiterungen, oder durch das Systemmodell selbst).

Setzt sich ein Element aus mehreren untergeordneten Elementen zusammen, von denen einige variabel oder optional sind, ändert sich natürlich auch das Verhalten des entsprechenden Modellelementes. Je nach der gewählten Modellierung dieser Konstellation lassen sich mittels «includes» oder «extends» eingebundene, untergeordnete Use-Cases beziehungsweise die einbindende Beziehung als optional (für «include») oder als Variationspunkt kennzeichnen.

Die Anwendungsfälle für als optional oder variant gekennzeichneten Modellelemente sind an diese gebunden und damit implizit nicht immer vorhanden. Die explizite Modellierung dieser impliziten Abhängigkeit scheint für Anwendungsfälle oftmals sinnvoll: da Anwendungsfälle oftmals zur Spezifi-

kation der Anforderungen an eine Entität verwendet werden, sollten derartige Abhängigkeiten explizit dokumentiert werden.

7.5.2 Kollaborations- und Sequenzdiagramme

Eine Kollaboration modelliert das Zusammenwirken von Modellelementen zur Realisierung einer Entität. Diese — zum Beispiel ein Anwendungsfall oder eine Operation —- kann der Kollaboration explizit zugeordnet werden. Dazu enthält eine Kollaboration eine Menge von Rollen für Bezeichner und Assoziationen, welche die Beteiligten bestimmen, die für den zu erfüllenden Zweck erforderlich sind. Diese Rollen werden bei der Realisierung konkreten Bezeichnern (*classifier*) zugeordnet.

Im Kontext einer Kollaboration werden Interaktionen definiert, die mittels teilweise geordneter Nachrichten das Kommunikationsschema zwischen den Rollen beschreiben. Diese Interaktionen können dabei auf Instanzebene oder auf Rollenebene beschrieben werden.

Zur Darstellung generischer Handlungsabfolgen gibt es parametrisierte Kollaborationen, die mehrfach in verschiedenen Systemen angewendet werden können. Diese können explizit in einer parametrisierten Form mit Hilfe der Rollen modelliert werden und dokumentieren zum Beispiel Entwurfsmuster (*design patterns*).

Kollaborationen sind generalisierbar und lassen sich damit verallgemeinern oder spezialisieren. Dabei kann eine Umbenennung oder Erweiterung der Rollen erfolgen, die genauen Regeln dazu sind in [OMG01a, Seite 2-136] beschrieben.

Die Darstellung von Kollaborationen in UML erfolgt in verschiedenen Formen: für die Darstellung von Objektinteraktionen werden Kollaborations- und Sequenzdiagramme verwendet. Sequenzdiagramme stellen die zeitliche Abfolge von Stimuli zwischen Klassen oder Instanzen zur Realisierung der zugeordneten Entität (eine Operation oder ein Bezeichner, zum Beispiel ein Anwendungsfall) dar. Kollaborationsdiagramme können auf Instanz- oder Spezifikationsebene dargestellt werden. Auf der Instanzebene wird eine konkrete Menge von Interaktionsinstanzen durch die beteiligten Objekte, Verbindungen und Stimuli dargestellt. Auf Spezifikationsebene werden die beteiligten Bezeichner- und Assoziationsrollen und deren Multiplizitäten bestimmt. Dabei ist es möglich, mit zusätzlichen Modellelemente Anforderungen darzustellen, die nicht durch Rollen beschrieben werden können — zum Beispiel Generalisierungsbeziehungen zwischen den Rollen.

Daneben gibt es für Kollaborationen noch eine weitere Darstellungsart: aus externer Sicht. Dabei wird die Kollaboration als gestrichelte Ellipse gezeichnet, die mit den beteiligten Bezeichnern verbunden ist. Diese Notation stellt den Einsatz von parametrisierten Kollaborationen dar und ist ebenfalls zur Modellierung zusätzlicher Beziehungen (zum Beispiel Generalisierungen, Constraints, zusätzliche Bezeichner zur Beschreibung des Kontexts einer Kollaboration) zwischen verschiedenen Kollaborationen geeignet.

7.5.2.1 Variabilität zwischen Kollaborationen

Kollaborationen bieten durch ihre Metamodell-Beschreibung als generalisierbare Elemente die Möglichkeit, eine Vererbungs-Hierarchie aufzubauen und damit Gemeinsamkeiten zwischen Kollaborationen in einer übergeordneten Generalisierung zusammenzufassen. Umgekehrt stellt jede Spezialisierung eine spezifische Umsetzung da, die sich normalerweise von den anderen unterscheidet. Man kann diese Konstellation als Anwendung eines Variationspunktes auffassen und wenn zutreffend, unter Verwendung der Variationspunkt-Erweiterung als solche modellieren.

In diesem Fall wird eine Menge von Kollaborationen mit ihren gemeinsamen Eigenschaften (den beteiligten Rollen) als Variationspunkt beschrieben, von denen jede Kollaboration als Variante unterschiedliche Interaktionen dieser gemeinsamen Eigenschaften modelliert. Die Detaildarstellung der Interaktionen jeder Variante erfolgt individuell in eigenen Diagrammen, wobei der gemeinsame Teil der Interaktion durch den Variationspunkt beschrieben werden kann.

7.5.2.2 Variabilität in Kollaborations-Spezifikationen

In Kollaborationsdiagrammen ohne Interaktionen wird der Kontext einer Interaktion dargestellt. Dabei werden die Bezeichnerrollen (*ClassifierRoles*) als Klassen-Rechtecke, die Assoziationsrollen (*AssociationRoles*) als Linien dargestellt. In der Metamodell-Beschreibung sind Assoziationsrollen eine Erweiterung der Assoziationen und Bezeichnerrollen eine Erweiterung von Bezeichnern (*Classifier*). Dementsprechend können Classifier-Rollen als Variationspunkte oder Varianten eingeordnet werden, die durch Assoziationsrollen miteinander verbunden sind.

In diesem Fall bleibt die Variationspunkt-Semantik erhalten, gegenüber Variationspunkten in statischen Modellen können jedoch nur Assoziationsrollen zur Verbindung von Varianten und Variationspunkt verwendet werden. Bei der Interpretation kann man diese Konstellation so auffassen, daß in Abhängigkeit von einer Bedingung (die am Variationspunkt auftritt) eine oder mehrere Rollen ausgewählt werden. Dabei ist jedoch zu beachten, daß die so spezifizierte Kollaboration in der Regel durch Interaktionsdiagramme dokumentiert wird, die die entsprechende Abhängigkeit von der Auswahl von Varianten umsetzen müssen.

Darüber hinaus können beteiligte Rollen als optional gekennzeichnet werden, um die Existenz in Abhängigkeit von einer übergeordneten Bedingung zu dokumentieren. In diesem Fall ist eine Rolle zum Beispiel nur beteiligt, wenn ein bestimmtes Merkmal des Systems vorhanden sein soll. Analog zu den Regeln für optionale Elemente in statischen Modellen sind die mit einer optionalen Bezeichnerrolle verbundenen Assoziationsrollen ebenfalls implizit optional und können bei Bedarf explizit als solche gekennzeichnet werden.

7.5.2.3 Variabilität in Interaktionsdiagrammen

Die Realisierung von Kollaborationen wird durch Interaktionen oder Instanzinteraktionen beschrieben. Diese Interaktionen können als Kollaborationsdiagramm oder als Sequenzdiagramm dargestellt werden und verdeutlichen die Aktionen oder Nachrichten zwischen den Teilnehmern und den zeitlichen Ablauf.

Die zur Darstellung verwendeten Modellelemente `Instance` und `Stimulus` sind im Metamodell als Spezialisierungen von `ModelElement` definiert. Da hierbei die Interaktion zwischen Objektinstanzen modelliert wird, ist eine Notation von Variationspunkten nicht sinnvoll, da diese in den übergeordneten Klassen beschrieben werden sollten (vergleiche letzter Absatz in Abschnitt 7.3.1.1).

Wurde die Kollaborationsspezifikation dagegen unter Verwendung von Variationspunkten modelliert, sind dessen Auswirkungen — die optionale Existenz der Varianten — in der Modellierung zu berücksichtigen, beziehungsweise jede Variante wird mit einem eigenen Interaktionsdiagramm beschrieben. Gleiches gilt ebenfalls, falls beteiligte Elemente im statischen Modell als «optional» oder «variant» gekennzeichnet wurden.

Daneben kann natürlich die Kennzeichnung als optionales Modellelement auf die Elemente der Interaktionsdiagramme angewendet werden. Diese können auch als optional gekennzeichnet werden, wenn keine Variationspunkte modelliert wurden, zum Beispiel um die Abhängigkeit von einem Produktlinien-Kontext zu modellieren. Um die Konsistenz der Darstellung zu verbessern, sollten solche Abhängigkeiten ebenfalls in der Spezifikation der Kollaboration dokumentiert werden.

7.5.2.4 Beispiel für Kollaborationen mit Variabilität

Die Anwendung und grafische Darstellung der beschriebenen Notationen für Variabilität in Kollaborationen werden an dieser Stelle beispielhaft gezeigt. Dabei wurde das bereits das für die Variationspunkte verwendete Beispiel verschiedener Sortieralgorithmen als Kollaboration modelliert.

In Abbildung 7.7 findet sich die Spezifikation der entsprechenden Kollaboration für einen Sortieralgorithmus, die die beteiligten Rollen (Bezeichner und Assoziationen) und in dieser Darstellung ebenfalls die erforderlichen Methoden spezifiziert — deren Darstellung kann jedoch auch entfallen. Dabei ist die Bezeichnerrolle `observer` optional, also nicht in allen Systemen vorhanden.

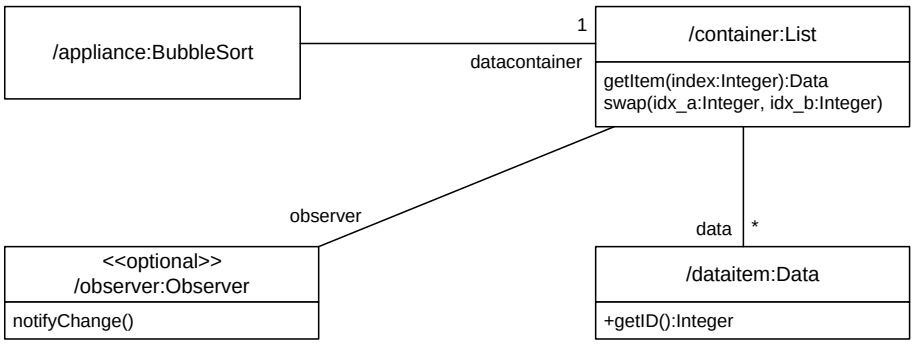


Abbildung 7.7: Spezifikation der Kollaboration mit optionaler Rolle

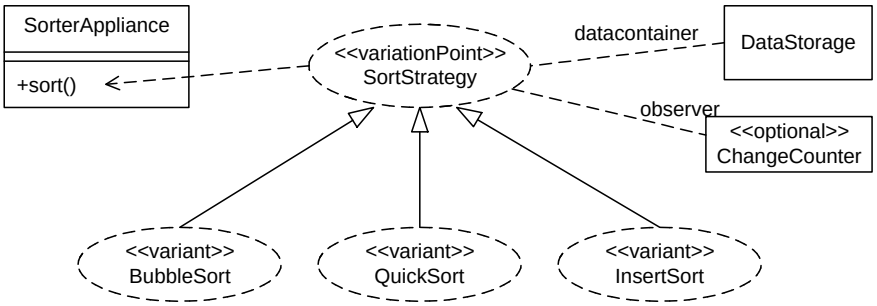


Abbildung 7.8: Einsatz der Kollaboration mit Spezialisierungen als Variationspunkt

In Abbildung 7.8 wird der Einsatz dieser Kollaboration modelliert. Dabei wird jeder Rolle ein konkreter Bezeichner (in diesem Fall eine Klasse) zugewiesen. Zusätzlich werden in diesem Diagramm drei Spezialisierungen der Kollaboration modelliert, die unterschiedliche Strategien umsetzen, jedoch auf einen gemeinsamen Kontext aufbauen. Dieser Kontext wurde als Variationspunkt identifiziert, der durch die verschiedenen Strategien als Varianten realisiert wird.

Abbildung 7.9 ist das der Variante `BubbleSort` zugeordnete Kollaborationsinstanzdiagramm, das die Objektinteraktion modelliert. Der Algorithmus ist für das Beispiel nur ansatzweise dargestellt. In der Interaktion ist eine optionale Nachricht an `Observer` modelliert, die nur bei Existenz des Observer ausgeführt wird.

7.5.3 State-Charts und Aktivitätsdiagramme

Zustandsdiagramme (*state charts*) beschreiben das Zustandsverhalten von Entitäten. Mit Zustandsdiagrammen kann das Verhalten von verschiedenen Elementen modelliert werden, zum Beispiel das Verhalten von Objektinstanzen oder auch die Interaktionen zwischen Entitäten, beispielsweise für Kollaborationen.

Zustandsdiagramme in UML bestehen aus Zuständen und Transitionen. Im Metamodell werden Zustände als Spezialisierungen von `StateVertex` modelliert und beschreiben verschiedene Arten von Zuständen. Dazu gehören auch zusammengesetzte Zustände (*composite states*), die aus untergeordneten Zuständen und deren Transitionen bestehen. Transitionen beschreiben die Übergänge zwischen Zuständen und können mit optionalen Aktionen, Bedingungen oder Ereignissen verknüpft sein.

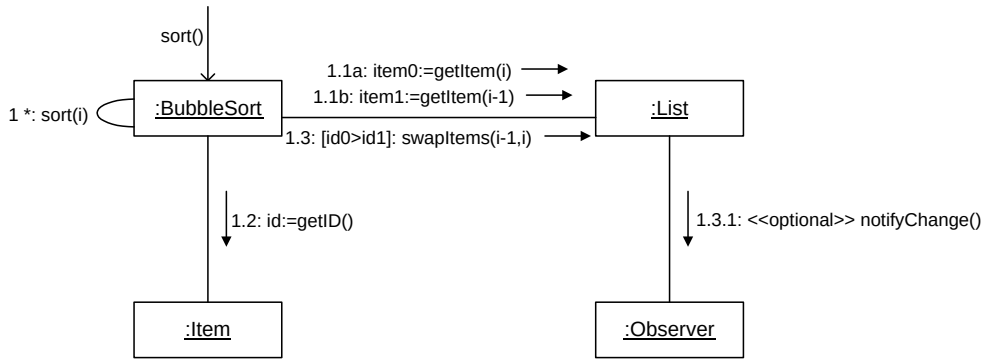


Abbildung 7.9: Interaktionsdiagramm einer Spezialisierung mit optionalem Stimuli

Aktivitätsdiagramme sind im Metamodell eine Erweiterung von Zustandsdiagrammen und modellieren den Ablauf von Prozessen zwischen einem oder mehreren Bezeichnern. In erster Linie beschreiben Aktivitätsdiagramme die Abfolge und Bedingungen von Aktionen, die eine Entität realisieren. Dazu werden die Zustände weiter verfeinert und durch detaillierte Spezifikationsangaben ergänzt. Eine zusätzlich mögliche Partitionierung erlaubt die Modellierung von Zuständigkeiten der beteiligten Elemente und wird diagrammtechnisch durch sogenannte *Swimlanes* beschrieben. Im Gegensatz zu Statecharts werden Aktivitätsdiagramme einem Bezeichner zugeordnet, der den Kontext des Modells bestimmt.

Strukturell sind sich Statecharts und Aktivitätsdiagramme sehr ähnlich — deswegen werden sie im Folgendem gemeinsam betrachtet. Mit Zustandsmodellen sind deswegen stets beide Diagrammarten gemeint.

7.5.3.1 Variationspunkte in Zustandsmodellen

In Zustandsmodellen treten durch deren programmablaufsbeschreibenden Charakter eine Menge von Konstellationen auf, die als Variationspunkte identifiziert werden könnten. Dann handelt es sich jedoch um die Auswahl von Alternativen zur Laufzeit. Variationspunkte im Sinne dieser Arbeit stellen dagegen eine Auswahl zwischen Alternativen zur Entwicklungszeit dar. Bei der Modellierung wird es aufgrund von Variationspunkten im statischen Modell sicherlich dazu kommen, daß aufgrund dessen eine Auswahl zwischen verschiedenen Ausführungszweigen (respektive zwischen verschiedenen Transitionen) getroffen werden muß. Um eine klare Trennung der Variationspunkt-Auswahl von der davon abhängigen Auswahl zur Laufzeit zu unterstützen, soll die Modellierung von Variationspunkten in Statecharts oder Aktivitätsdiagrammen nicht durch die Variationspunkt-Notation unterstützt werden (siehe Abschnitt 7.3.1.1) — dazu können bei Bedarf auch optionale Elemente verwendet werden.

7.5.3.2 Optionale Elemente in Zustandsmodellen

Wie gerade eben angesprochen hat die Modellierung von statischen Variationspunkten Auswirkungen auf den Programmablauf. Dabei ist das Vorhandensein einer Variante von der Variationspunkt-Auswahl abhängig und wirkt sich entsprechend auf ihre Beteiligung im dynamischem Modell aus. Dazu ist es möglich und sinnvoll, Modellelemente von Zustandsübergangsdiagrammen als optional zu kennzeichnen — zum Beispiel, um die Auswirkung einer Variantenauswahl auf die möglichen Ausgangs-Transitionen einer Entscheidung zu modellieren. Dies trifft sinngemäß auch für optionale Elemente im statischem Modell zu.

Daneben ist es natürlich prinzipiell möglich, zum Beispiel bestimmte Transitionen oder Zustände als optional in Abhängigkeit von einer bestimmten Bedingung zu modellieren. Davor sollte jedoch immer geprüft werden, ob diese Abhängigkeit nicht auch ohne Nutzung der Erweiterung modellierbar ist — um eine unübersichtliche Vermischung der Konzepte „Erweiterungen für Variabilität“ und „dynamische Modellierung in UML“ zu vermeiden.

7.5.4 Grafische Darstellung von Selektions-Variabilität

Abschließend soll die bereits in [Atkinson00] verwendete Möglichkeit, durch die grafische Darstellung eine Trennung zwischen „normalen“ und „selektionsspezifischen“ Modellelementen zu erreichen, erwähnt werden.

Insbesondere bei Entscheidungen in Zustandsdiagrammen kann die Möglichkeit genutzt werden, durch eine abweichende grafische Gestaltung, zum Beispiel mit einer anderen Füllfarbe oder invertierter Darstellung, rein selektionsspezifische Verzweigungen von den anderen Modellelementen zu unterscheiden. Mit diesen Verzweigungen sind solche gemeint, mit denen die Auslassung von Programmzweigen bei nicht vorhandenen Elementen realisiert wird. Derartige medienabhängige Darstellungen werden jedoch durch die UML explizit vermieden und sollten durch entsprechende UML-Elemente (zum Beispiel äquivalente TaggedValues) formal korrekt beschrieben werden.

7.6 Integration in das Gesamtmodell

In diesem Abschnitt geht es darum, die beschriebenen Erweiterungen zu einem Gesamtbild zusammenzufügen. Dieses Gesamtbild ist jedoch prinzipiell ebenfalls vom verwendeten Vorgehensmodell bei der Entwicklung abhängig. Aufgrund dessen stellen die nachfolgend beschriebenen Konzepte nur Ideenskizzen dar, die bei der konkreten Anwendung speziell angepaßt werden können (zum Beispiel mit oder in Abhängigkeit von verwendeten Werkzeugen).

Daneben stellen einige der hier angesprochenen Konzepte, zum Beispiel formale Spezifikation von Selektionen oder die Entscheidungsmodellierung interessante Aufgabenfelder für weiterführende Arbeiten dar.

7.6.1 Verfolgbarkeit (*Traceability*)

Wie in Abschnitt 6.2.1 beschrieben stellt die Verfolgbarkeit der Entwicklung eines Konzeptes über mehrere Entwicklungsstufen einen wichtigen Bestandteil der Dokumentation dar.

Derartige Konzepte werden bereits in UML durch eine Spezialisierung von Dependencies, den Abstraktionen (Metaklasse `Abstraction`), unterstützt [OMG01a, Seite 2-18f]. Dazu gehören verschiedene vordefinierte Stereotypen: «derive», «realize», «refine» und «trace». Insbesondere letzterer stellt eine rein dokumentarische Beziehung dar und erlaubt beispielweise die Verbindung von Modellelementen auf Analyse- und Design-Ebene.

Für den Einsatz der Variabilitäts-Erweiterungen ist der Einsatz dieser Beziehungen sehr anzuraten. Die Richtung der Abstraktion kann im Falle der «trace»-Beziehung per Konvention dazu genutzt werden, die Vorwärts-*traceability* zu beschreiben.

7.6.2 Selektionsbedingungen

Ein Bestandteil der Erweiterungen für Variationspunkt und optionale Elemente sind Bedingungen, die die Kriterien für die Anwesenheit beziehungsweise Auswahl von Varianten oder Modellelementen beschreiben. Diese werden als `BooleanExpression` angegeben, ergeben also nach der Berechnung einen Wahrheitswert.

Für die formale Spezifikation dieses Ausdrucks wurde OCL als Beschreibungssprache empfohlen, da sie für logische Ausdrücke spezialisiert und speziell für Modellierungszwecke angepaßt ist. Diese Verwendung entspricht dem Einsatzzweck der OCL als *Guard* ([OMG01a, Seiten 6-50]) wie beispielsweise in der Metaklasse `Guard` [OMG01a, 2-155].

Im Normalfall werden OCL-Ausdrücke auf Instanzen von Klassen — den Objekten — ausgewertet. Da die Berechnung von Selektionen unabhängig von der Existenz von Objektinstanzen ist, können Selektionsbedingungen nicht auf M0-Ebene (Benutzerobjekte) berechnet werden, sondern erfordern die Berechnung auf Modellebene (M1). Demzufolge ist die Spezifikationsebene derartiger OCL-Ausdrücke (*scope*) das Metamodell. Die Berechnung erfolgt dann analog wie für Constraints, die an Stereotypen gebunden sind [OMG01a, Seite 2-82]. Das bedeutet, die in einem Selektions-Ausdruck verwendeten OCL-Typen werden durch das Metamodell der UML gebildet, das Modell bestimmt die Werte für die Berechnung. Damit ist es möglich, in einem OCL-Ausdruck zum Beispiel die Existenz eines Modelelementes als Bedingung einzubringen.

Bei einer fortgeschrittenen Verwendung von OCL-Ausdrücken zur Spezifikation von Auswahlregeln können häufig verwendete Ausdrücke — wie schon in der Beschreibung der Erweiterungen geschehen — als zusätzliche Operationen und *Let-expressions* in OCL definiert werden, die als virtuelle Operationen beziehungsweise Eigenschaften (*properties*) der jeweiligen Metaklasse erscheinen ([OMG01a, 6-55].

Auf jedem Fall sollte dabei die Möglichkeit vorgesehen werden, Bezüge auf das für das jeweilige System zutreffende Merkmalsmodell einzubinden. Konkret bedeutet das, das die Anwesenheit von Merkmalen als Bedingung für die Anwesenheit von optionalen Elementen und die Auswahl von Varianten genutzt werden kann. Eine diesbezügliche Spezifikation der Erweiterung soll an dieser Stelle nicht erfolgen, da hierzu weitere Erfahrungen und Anforderungen aus dem Spezialgebiet des Konfigurationsmanagements einfließen sollten und eine praktische Werkzeugunterstützung, die diese umsetzen könnte, noch nicht existiert.

7.6.3 Entscheidungsmodellierung

Entscheidungen treten in Zusammenhang mit Variabilität insbesondere bei der Auswahl von Merkmalen, Varianten oder optionalen Modellelementen auf. Damit beschäftigt sich die Entscheidungsmodellierung in erster Linie mit der Formulierung von Selektions-Bedingungen, die im vorhergehenden Abschnitt beschrieben wurde.

Darüber hinaus kann auch eine explizite Modellierung des Entscheidungsprozesses erfolgen, wie es zum Beispiel in [Coriat00] propagiert wird. Dort wird das Entscheidungsmodell mit Hilfe eines Metamodells beschrieben, ohne auf konkrete Darstellungsmöglichkeiten der Entscheidungen oder Entscheidungsregeln einzugehen. In [Atkinson00] werden diese in Tabellenform durch Fragen dargestellt, wobei vorgegebene Antwortmöglichkeiten auf durchzuführende Aktionen, primär die Auswahl von Komponenten, verweisen.

Die Möglichkeit, derartige Entscheidungsmodelle ebenfalls in UML zu notieren, ist naheliegend und soll an dieser Stelle kurz als Vorschlag skizziert werden. Am besten eignen sich *State Charts* oder Aktivitätsdiagramme. Da letztere für die Prozeßmodellierung prädestiniert sind [OMG01a, Seite 2-181] und die direktere Darstellung von Entscheidungen (Rauten-Notation) erlauben, sollten diese verwendet werden.

Die Fragestellung zu jeder Entscheidung kann als Aktion oder als Entscheidung dargestellt werden. Die möglichen Antworten bilden Transitionen zu Aktionen, die die jeweils durchzuführende Aktion beschreiben oder weitere Entscheidungen beschreiben. Die Formulierung der Fragestellung und der Antwort muß unter Umständen aus Platzgründen recht knapp gehalten werden, so daß die Anreicherung um Notizen oder Verweise auf externe Dokumentation sinnvoll ist. Zusätzlich können die Fragen und Aktionen mit dokumentierenden Verweisen («trace»-Abstraktionen) auf die betroffenen Modellelemente versehen werden, zum Beispiel kann eine Frage auf einen Variationspunkt und jede möglich

Antwort auf eine der zugehörigen Varianten verweisen.

Das Entscheidungsmodell könnte auch direkt in das modellierte System eingebracht werden, zum Beispiel mit Hilfe von Kommentaren oder durch Beziehungen. Bei Kommentaren fehlt jedoch der direkte Bezug zum Modell, was schnell zu Inkonsistenzen führt und die Unterstützung durch Werkzeuge (Verifikation, Automatisierung) erschwert. Beziehungen zur Darstellung des Entscheidungsmodells würden einen azyklischen Graph von Variationspunkten/Varianten ergeben, der direkt in das Modell eingeflochten ist und dessen Übersichtlichkeit sehr stark erschwert.

7.7 Anwendung der Erweiterungen

7.7.1 Domänen- und Merkmalsanalyse

Die Erweiterung zur Merkmalsmodellierung in UML unterstützt die Diagramm-Notationen, die in der merkmalsorientierten Domänenanalyse (FODA) und deren Weiterentwicklungen verwendet werden. Dazu gehören in erster Linie die Merkmalsdiagramme, die als grafische Darstellung eines Merkmalsmodells dessen Kerninhalte grafisch anschaulich darstellen.

Die vorgeschlagene Darstellung ist für Erweiterungen offen und kann für verschiedene Ansätze und Vorgehensweisen zur Merkmalsanalyse angepaßt werden — für FORM zum Beispiel mit Unterscheidung zwischen Dekompositions-, Spezialisierungs- und Implementierungsbeziehungen zwischen Merkmalen; und für Ansätze, die die Art der Beziehungen nicht weiter differenzieren, entfällt die Unterscheidung und die Beziehungen werden stets als Dekomposition/Komposition modelliert.

Ebenso kann die Möglichkeit der Modellierung von externen Merkmalen ignoriert werden — bei gleichzeitiger Nichtbetrachtung der Beziehungstypen zwischen Merkmalen reduziert sich dann die Notation auf die von FODA bekannten Ausdrucksmöglichkeiten.

Andererseits bietet die Verwendung der UML zusätzliche Möglichkeiten, um Merkmalsmodelle besser darzustellen. So können zum Beispiel Constraints und Anmerkungen (*notes*) genutzt werden, um Abhängigkeiten zwischen Merkmalen detaillierter zu beschreiben.

7.7.2 Modellierung mit Variabilitäten

Die Erweiterungen zur Modellierung von Variabilität — Variationspunkte und Optionale Elemente — lassen sich theoretisch in allen UML-Werkzeugen einsetzen. Damit können in abstrakter Darstellung Unterschiede zwischen verschiedenen Ausprägungen des Modells beschrieben werden, die sich nicht aus Laufzeit-Variabilität, sondern aus der gemeinsamen Modellierung im Rahmen einer familienorientierten Entwicklung ergeben.

Dazu wurde das Konzept der Variationspunkte aus [Jacobson97] aufgegriffen, durch beschreibende Informationen (zum Teil aus [Coriat00]) komplettiert und syntaktisch und semantisch als UML-Erweiterung beschrieben. Der konkrete Einsatz ist unter anderem abhängig von dem zugrundeliegenden Entwicklungsansatz und kann zum Beispiel durch Generatoren realisiert werden, die aus den Informationen im generischen Modell einzelne Ausprägungen erstellen.

Optionale Elemente können in analoger Weise eingesetzt werden, um einzelne variante Elemente zu kennzeichnen. Sie entsprechen zum Beispiel varianten Elementen in [Atkinson00], werden jedoch durch zusätzliche Informationen detaillierter beschrieben.

Der konkrete Einsatz ist mindestens abhängig von der verfügbaren Werkzeugunterstützung und dem angewandten Entwicklungsprozeß. Einflußfaktoren sind zum Beispiel, in welchem Maße generative Verfahren zum Einsatz kommen und bei welchem Entwicklungsstand die Individualisierung der einzelnen Produkte beginnt.

7.8 Zusammenfassung des Konzeptes

Zusammenfassend läßt sich sagen, daß die Modellierung von Variabilität eine andere Dimension der Beschreibung von Softwaresystemen öffnet. Zentraler Bestandteil des objektorientierten Paradigmas ist die Beschreibung von Objekten mittels Klassen und deren Instanziierung. Bei der Beschreibung von Variabilität geht es dagegen um die Beschreibung von varianten Systembestandteilen einschließlich daraus entstehender Wechelwirkungen (bekannt als *feature interactions* [Clauss01]) und der Festlegung derer Anwesenheit. Diese Festlegung wird durch die Auswahl von Elementen oder das Binden von Varianten für konkrete Produkte des modellierten Bereiches beschrieben.

Die grundlegenden Bestandteile zur systematischen Entwicklung mit Variabilitäten sind eine explizite Merkmalsmodellierung sowie die Modellierung von Variabilität während der Systementwicklung.

Die Notation der Merkmalsdiagramme bringt eine zusätzliche Abstraktionsebene in die Modellierung ein, die bisher nicht in der UML enthalten ist. Neben der systematischen Entwicklung von Gemeinsamkeiten und Unterschieden birgt die Merkmalsmodellierung den zusätzlichen Vorteil, eine Notation für die nutzer- beziehungsweise kundenorientierte Kommunikation einzuführen, die die Eigenschaften eines Softwareproduktes in abstrakter Weise, anschaulich und typischerweise kurz und prägnant beschreibt.

Die Erweiterung für Variationspunkte wurde gegenüber [Jacobson97] genauer beschrieben und ist im Vergleich zu [Coriat00] näher an die im Standard definierte Semantik der UML-Modellelemente angelehnt. Optionale Elemente vervollständigen das Konzept um die Beschreibung von einfacheren Situationen von Variabilität. Damit können Variationspunkte und optionale Elemente zur Beschreibung von Variabilität in generischen Modellen verwendet werden. Beide Ansätze integrieren die verschiedenen bekannten Attribute von Variabilität und enthalten einen ersten Ansatz zur Integration der Beschreibung von Konfigurationswissen. Sie erweitern sozusagen den Sprachschatz der UML und beschreiben selektionstypische Aspekte eines Modells.

Aufgrund der standardkonformen Beschreibung und Gestaltung der Erweiterungen können diese in einem Profil für UML formal spezifiziert und standardisiert werden, diese Arbeit kann als erster Schritt in diese Richtung gelten.

Kapitel 8

Praktischer Einsatz

Dieses Kapitel beschäftigt sich mit dem praktischen Einsatz der entwickelten Notation. Dazu wird ein Beispiel aus der eCommerce-Domäne modelliert.

Aufgrund des Umfanges der zugrundeliegenden Domäne wird die mögliche Anwendung der entwickelten Erweiterungen anhand eines prototypischen Beispiels demonstriert, das auf ein Spezialgebiet der modellierten Domäne – den Bestellprozeß – fokussiert ist. Anhand dessen wird gezeigt, wie ein Merkmalsmodell aussehen kann und wie sich Merkmale in verschiedenen Modellen eines Systemdesigns wiederfinden können. Daneben werden über mehrere Modelle hinweg die im Design auftretenden Variabilitäten ebenfalls explizit modelliert.

Analog zum theoretischen Ablauf einer Domänenentwicklung wird mit der Entwicklung und Beschreibung des Merkmalsmodells begonnen. Darauf aufbauend wird in Abschnitt 8.2 ein einfaches, hypothetisches System mit Variabilitäten modelliert, das diese Merkmale beispielhaft realisiert. Abschließend erfolgt im letztem Absatz die Bewertung der Eignung der UML zur Variantenmodellierung.

Die für die Modellierung erforderliche Merkmalsanalyse ist nicht Gegenstand dieser Arbeit. Aufgrund dessen sind die dabei zusätzlich gewonnenen Erkenntnisse in Anhang A als zusätzliche Informationen dargestellt.

8.1 Merkmalsmodellierung mit Hilfe von UML

Prinzipiell lassen sich für alle in dieser Arbeit entwickelten Erweiterungen die für alle Modelle in UML spezifizierten Beschreibungsmöglichkeiten verwenden, solange dies nicht explizit ausgeschlossen wird.

Damit wird insbesondere die Gruppierung von Merkmalen mit Hilfe des *Model Management* und die Verwendung von «trace»-Abstraktionen zur Dokumentation eingebunden. Letzte können und sollten verwendet werden, um die Entwicklung eines Konzeptes (zum Beispiel eines Merkmales) zu dokumentieren. Das Model Management kann speziell auch dazu genutzt werden, um die Merkmale nicht von normalen Klassenmodellen zu unterscheiden, sondern auch, um Merkmale darüber hinaus zu gruppieren.

Gemäß den in Abschnitt 7.2.2.4 spezifizierten Constraints werden alle Merkmalsknoten als abstrakte Klassen dargestellt. Diese Regel kann im Sinne einer praxistauglicheren Anwendung auch abgeschwächt oder formal korrekt bei der Verwendung von Objektdiagrammen aufgehoben werden.

8.1.1 Bestellprozeß

Für die Demonstration der Verwendung der Erweiterungen wird ein spezieller Bereich der eCommerce-Domäne modelliert: der Bestellprozeß. Dieser enthält bereits einiges an Variabilität und stellt ein zen-

traler Bestandteil jedes Kaufgeschäftes, das mit Hilfe von elektronischen Medien abgewickelt wird, dar.

Das im folgenden modellierte Merkmalsdiagramm beschreibt die Domäne „Bestellprozeß“, die unter Geschäftsprozessen eingeordnet werden kann. Für das Beispiel wird der Kontext der Domäne — also benachbarte Domänen und Beziehungen zu diesen — nicht weiter betrachtet. Da es im Beispiel keine übergeordneten Merkmalsmodelle gibt, wird hier der Begriff des Bestellprozesses als Konzept betrachtet — es ist jedoch auch möglich, diesen als Merkmal zu modellieren, wenn zum Beispiel Geschäftsabläufe im allgemeinem betrachtet werden.

Da ein Merkmalsdiagramm dazu dient, eine Auswahl von Merkmalen zur Bestimmung der verschiedenen Produkte einer Domäne zu ermöglichen, wird es in den seltensten Fällen ein komplettes Fachgebiet beschreiben, sondern auf das verfolgte Ziel zugeschnitten sein. In diesem Beispiel ist dies eine Gruppe von Bausteinen, die die Implementierung von Bestellprozessen ermöglichen. Die einzelnen Produkte stellen demzufolge konkrete Installationen der Bausteine mit einer zugeordneten Merkmalsmenge dar, während das Ganze (quasi die Produktlinie) als generisches Bestellprozeß-System betrachtet werden kann, das sich wiederum in ein größeres, übergeordnetes System eingliedert.

In Abbildung 8.1 setzt sich das Konzept eines Bestellprozesses zusammen aus den Merkmalen `Transaction` und `Fulfillment`. Ersteres beschreibt das vertragstechnische Zustandekommen eines Auftrages (Bestellung), letzteres die Erfüllung des Auftrages. Diese Merkmale sind in jedem Bestellprozeß vorhanden und werden demzufolge als notwendig modelliert. Daneben umfaßt ein Auftrag in den meisten Fällen auch einen entsprechenden Warenkorb (`Basket`), der den georderten Inhalt beschreibt. Dieser ist jedoch nicht erforderlich, wenn es sich um die Wiederholung einer früheren Bestellung (`Recurring_Order`) handelt. Für dieses Beispiel wird zwischen diesen beiden nicht immer vorhandenen Merkmalen (Typ ‘optional’) der wechselseitige Ausschluß modelliert. Da es auch Systeme gibt, die für eine wiederholte Bestellung den (alten) Warenkorb einbinden (zum Beispiel duplizieren), wird diese Beziehung als «weakConstraint» klassifiziert: es gibt nur einen Warenkorb, wenn es keine wiederholte Bestellung ist — jedoch nicht in allen Fällen. Die Gründe dafür können in der Detailbeschreibung genauer beschrieben werden.

Daneben kann der Bestellprozeß noch mit einer Genehmigung (`Approval`) verbunden werden — in diesem Fall werden Bestellungen erst ausgeführt, nachdem eine entsprechende Bestätigung zum Beispiel durch den Abteilungsleiter eines Angestellten erfolgte.

Die weitere Betrachtung des Merkmals `Transaction` beschränkt sich im folgendem auf die finanztechnischen Aspekte. Diese beinhalten die Preisermittlung der Bestellpositionen (`Price_Model`), die Berechnung von Steuern (`Tax`), die Bezahlung (`Payment`) und optional die Berechnung der Versandkosten (`Shipping_Cost`). Letzteres ist nur erforderlich, wenn ein physischer Gütertransport stattfindet. Diese einzelnen Merkmale sind Teile von `Transaction` und werden deswegen als Dekomposition modelliert.

Die Bezahlung wird weiter verfeinert und durch verschiedene Bezahlmethoden beschrieben. Diese unterscheiden sich zum Beispiel durch den Ablauf der Bezahlung die erforderlichen Zahlungsinformationen. Da jede Methode eine spezielle Umsetzung des allgemeinen Merkmales `Payment` darstellt, wird diese Beziehung als Generalisierung/Spezialisierung modelliert. Dies läßt sich durch die Kontrollfrage „*Ist* Zahlung auf Rechnung (`Pay_By_Bill`) *eine* Bezahlung?“ überprüfen, wogegen „Ist `Pay_On_Delivery` (per Nachnahme) *Teil von* Bezahlung?“ verneint werden wird, da es dieses vollständig beschreibt. Von den konkreten Bezahlverfahren muß eines ausgewählt werden, deswegen werden diese als Alternative modelliert. Da nur ein Verfahren (pro Bestellvorgang) möglich ist, schließen sich diese gegenseitig aus (XOR-Alternative), was durch den Constraint `{xor}` verdeutlicht wird.

Das Preismodell wird ebenfalls weiter verfeinert, dazu werden verschiedene Berechnungsmöglichkeiten unterschieden. Eine Preisliste (`Pricelist`) wird stets vorhanden sein, um den Basispreis der Produkte zu bestimmen. Darauf können noch Rabatte (`Discount`) gewährt werden, deren Anwendung wird zur Laufzeit bestimmt (zum Beispiel in Abhängigkeit vom Bestellvolumen). Diese Rabatte können auch kundenbezogen (`Customer_specific`) gewährt werden, weitere Möglichkeiten wer-

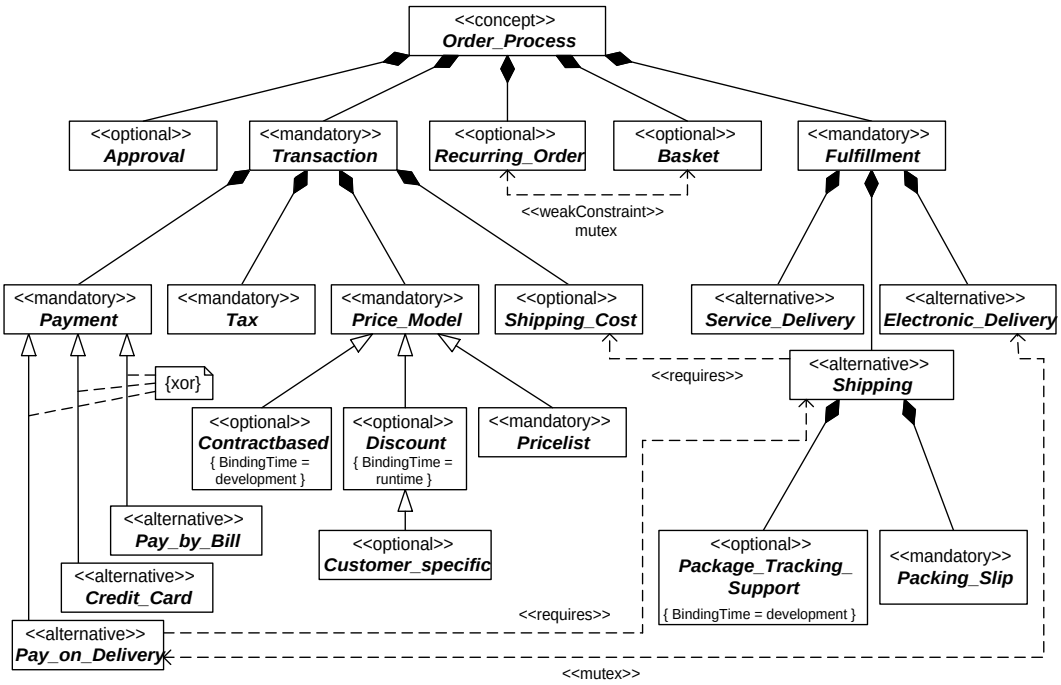


Abbildung 8.1: Merkmalsdiagramm für den Bestellprozeß

den für dieses Beispiel aus Komplexitätsgründen nicht modelliert. Neben Rabatten können aufgrund von Rahmenverträgen auch Festpreise ausgehandelt worden sein, die zum Beispiel die Anwendung von speziellen Preislisten erfordern. Ob ein konkretes System das letzte Merkmal unterstützt, wird im Beispielsystem bereits während der Entwicklung des Systems festgelegt.

Die verschiedenen Preismodelle stellen jeweils Spezialisierungen ihres übergeordneten Merkmales dar. Für diese kann auch der Name des übergeordneten Elementes in den Merkmalsnamen reingezogen werden (Beispiel: ‘customer specific discount’, ‘contract-based price model’) um die Merkmalsnamen lesbarer zu machen. Dieses wurde im Beispiel umgekehrt, um Platz zu sparen und die Semantik der Generalisierung (‘is_a’) für die Namensgebung der Merkmale auszunutzen.

Die Ausführung einer Bestellung (`Fulfillment`) wird unterschieden in den Versand (`Shipping`), elektronische Erfüllung (`Electronic_Delivery`, zum Beispiel durch Download von Software) beziehungsweise die Diensterbringung (Beispiel: Fenster putzen). Die Erfüllung einer Bestellung kann durchaus auch mehrere Arten einschließen, zum Beispiel den Versand per E-Mail und den zusätzlichen Versand eines Datenträgers und Handbuch per Post — deswegen werden diese als nicht ausschließliche Alternative (ODER-Alternative) modelliert. Jede Art der Ausführung ist Teil der Ausführung — die Ausführung kann auch die hier nicht enthaltenen Prozesse zur Umsetzung enthalten — und wird deswegen als Dekomposition von `Fulfillment` modelliert.

Der Versand erfordert die Berechnung der Versandkosten, was durch die «requires»-Beziehung zu `Shipping_Cost` modelliert wird. Für die Bezahlung bei Erhalt der Lieferung (Nachnahme) ist ebenfalls der physische Versand notwendig. Analog kann die Bezahlung per Nachnahme nicht per elektronischen Versand (Download, per E-Mail) durchgeführt werden («mutex»-Beziehung).

Das Merkmal Versand wird durch die notwendige Existenz eines Lieferscheins (`Packing_Slip`) und der optionalen Unterstützung der Verfolgung des Lieferzustandes (`Package_Tracking_Sup-`

`port`) verfeinert. Die Anwesenheit des letztgenannten Merkmales wird bereits bei der Entwicklung (`BindingTime=development`) festgelegt, und kann sich zum Beispiel darin äußern, daß das Designmodell optionale Attribute für das *Package tracking* enthält.

Im praktischem Einsatz würde dieses Merkmalsmodell umfangreicher ausfallen, da aus Gründen der Komplexität und der Verständlichkeit nicht alle Merkmale vollständig entwickelt wurden (sofern das überhaupt möglich ist, da die Evolution vor Merkmalsmodellen nicht halt macht, siehe Abschnitt 6.2.4). So wurde bei Transaktionen zum Beispiel die mögliche Integration mit ERP-Systemen nicht berücksichtigt und `Approval` und `Basket` wurden nicht weiter verfeinert.

Ebenfalls lassen sich die verschiedenen Preismodelle weiter verfeinern. An dieser Stelle hat der Modellierer abzuwägen, bis zu welchem Detailgrad noch wesentliche Informationen dazukommen. Dazu sollte auch beachtet werden, daß das Merkmalsmodell in erster Linie dazu dient, die Gemeinsamkeiten und Unterschiede zwischen verschiedenen Produkten zu modellieren und nicht, eine vollständige Charakterisierung zu erlauben. Für letzteres können zum Beispiel bereits die Möglichkeiten des *Requirements engineering* genutzt werden.

Gut erkennbar ist in der Verfeinerung von `Price_Model` die Unabhängigkeit des Merkmaltyps (Auswahlregel) von der Abstraktionsbeziehung des Merkmals. Dies ist zu betonen, da die Beziehungstypen eine Aussage über die Art der Abstraktion des Merkmales treffen und als solche keinen Einfluß auf die spätere Umsetzung in einem Systemdesign haben. Dagegen werden die Selektionsregeln (gebildet durch den Merkmalstyp und die Baumstruktur sowie zusätzliche Constraints) zusammen mit der Bindezeit in der Regel einen deutlichen Einfluß auf die Entwicklung der Systemarchitektur haben.

In dem Abbildung 8.1 zugrundeliegenden Modell wurden für die variablen Merkmale aus Komplexitätsgründen nicht alle Bindezeiten angegeben. Diese wurden bisher nicht spezifiziert und werden für dieses Beispiel erst im Rahmen späterer Modelle festgelegt. Bei der Festlegung der Zeitpunkte sind ebenfalls die Abhängigkeiten der Bindezeiten innerhalb des Merkmalsbaumes und zwischen abhängigen Merkmalen zu beachten (zum Beispiel für eine «requires»-Beziehung zwischen je einem Merkmal mit Laufzeit- und Entwicklungszeitbindung). Dieses Thema wurde bisher in der Literatur nicht berücksichtigt und sollte für die Erstellung einer vollständig spezifizierten Erweiterung weiter untersucht werden.

8.2 Modellierung von Variabilitäten in der SW-Entwicklung

Nachdem im Abschnitt 8.1.1 das Merkmalsmodell des Bestellprozesses entwickelt wurde, soll im folgenden ein darauf aufbauendes Systemdesign entwickelt werden, das Variationspunkte und optionale Elemente zur expliziten Modellierung von Variabilitäten verwendet und auf diese Weise eine Art generisches Modell beschreibt.

Entsprechend dem üblichem objektorientierten Vorgehen wird mit der Entwicklung von Anwendungsfällen der Domäne begonnen, deren Einordnung in den Prozeßablauf durch ein Aktivitätsdiagramm dargestellt wird. Die statische Struktur des entstehenden generischen Systems wird durch ein Komponentendiagramm beschrieben, das durch ein Klassendiagramm zur Dokumentation der Datenstrukturen ergänzt wird.

Zur kompakteren Beschreibung derjenigen Bedingungen, welche die Anwesenheit eines Merkmales beinhalten, wird für alle Diagramme eine virtuelle Funktion

```
feature_selected (featurename : String) : Boolean
```

definiert, die bei Anwesenheit des Merkmales respektive dessen Auswahl `true` als Ergebnis liefert, bei Nichtanwesenheit `false`.

8.2.1 Use-Case-Modell

Der erste Schritt bei der Entwicklung eines Softwaresystems ist in der Regel die Beschreibung von Requirements und Anwendungsfällen (*use cases*). Bei der Entwicklung einer Softwarefamilie bezieht sich das auf das Design einer generischen Architektur (Produktlinienarchitektur). Ohne an dieser Stelle näher auf die Integration der Variabilitäten in den Domänen-Entwicklungsprozeß einzugehen, werden nachfolgend die Anwendungsfälle für das Bestellsystem modelliert.

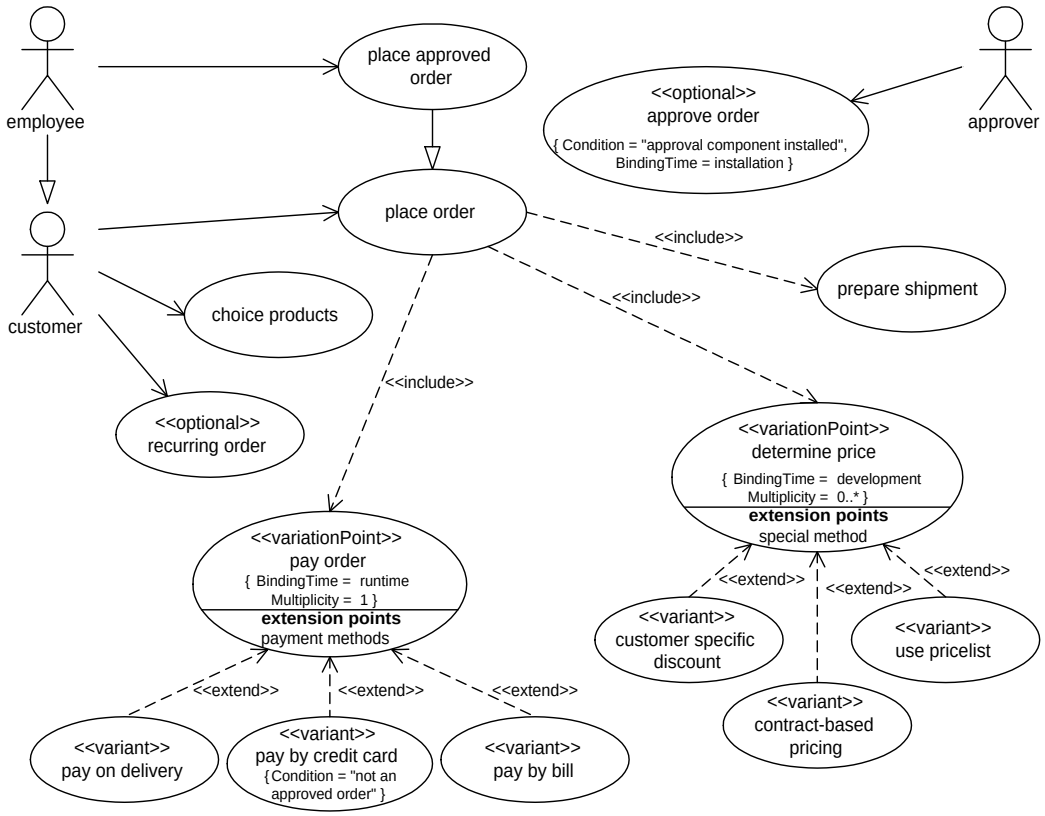


Abbildung 8.2: Use-Case-Diagramm des Bestellprozesses mit Variabilitäten

Abbildung 8.2 stellt die Anwendungsfälle des Bestellsystems dar. Wie im Abschnitt 7.5.1 dargestellt wurde, kann in Use-Case-Diagrammen mit Hilfe der extends-Beziehung die optionale Einbindung eines anderen Anwendungsfalles und damit auch Selektions-Variabilität modelliert werden. In diesem Modell wurden trotzdem Variationspunkte verwendet, um den generischen Charakter des Modells explizit zu kennzeichnen.

Die Bedingungen der Varianten wurden teilweise nicht dokumentiert, dies kann auch bewußt durch den Entwickler geschehen, wenn sie zum Beispiel nicht explizit bekannt sind oder noch zu bestimmen sind (untere Grenze der Kardinalität der TaggedValues in der Spezifikation, Seite 64). Ebenso wurden für `recurring order` bisher keine weiteren Eigenschaften der Variabilität bestimmt.

Das Modell beschreibt die Anwendungsfälle, die im Rahmen des Bestellprozesses auftreten und durch das System realisiert werden sollen. Dazu gehört die Produktauswahl (`choice products`) und das darauffolgende Absetzen der Bestellung (`place order`). Einen optionalen Anwendungsfall stellt die Wiederholung einer zurückliegenden Bestellung `recurring order` dar. Die Aufgabe einer

Bestellung wird durch einen Spezialfall verfeinert, der an einen Genehmigungsprozeß anknüpft(`place approved order`).

Die Plazierung einer Bestellung beinhaltet die Anwendungsfälle Preisbestimmung (`determine price`), Bezahlung (`pay order`) und die Vorbereitung des Versandes (`prepare shipment`). Letzteres wird in diesem Modell nur aus Sicht des Bestellers gesehen, beschreibt also insbesondere die Versandart, damit zusammenhängende Kosten und zum Beispiel die Ausgabe der Paketnummer (für das *package-tracking*-Merkmal).

Die Bezahlung der Bestellung enthält einen Erweiterungspunkt, der die Details der verschiedenen Zahlungsmethoden an erweiternde Anwendungsfälle delegiert. Die beschreiben jeweils den spezifischen Teil — zum Beispiel welche Informationen zur Zahlung erforderlich sind — wogegen `pay order` die generellen Aspekte der Bezahlung an sich — beispielweise die Darstellung der Bestellung mit Netto- und Bruttopreis und das endgültige Bestätigen der Bestellung — modelliert. Für die Bezahlung per Kreditkarte (`pay by credit card`) wurde in der Bedingung festgehalten, daß diese nicht in Verbindung mit einer authorisierten Bestellung genutzt werden kann.

Durch einen weiteren Akteur wird die Bestätigung einer Bestellung durchgeführt — sofern dieses erforderlich ist. Die Existenz dieses Anwendungsfalles wurde als optional beschrieben und ist in diesem Beispiel davon abhängig, ob die entsprechende Komponente für diese Funktionalität installiert wurde.

Das Modell kann noch verbessert werden, indem zum Beispiel die Bedingungen verfeinert werden. Dazu kann `place approved order` ebenfalls als optionales Element gekennzeichnet werden, und zwischen diesem und `approve order` eine bidirektionale «requires»-Beziehung festgestellt werden. Die UML-Spezifikation läßt insbesondere für Dependencies offen, ob diese in Use-Case-Diagrammen erlaubt sind oder nicht (Beschreibung von Use-Case-Diagrammen, [OMG01a, Seite 3-97]).

Was bereits an diesem Diagramm sichtbar wird, ist die Aufblähung der Diagrammdarstellung durch die zusätzliche Notation der TaggedValues zur Beschreibung der Variabilitäten. Daraus läßt sich bereits ableiten, das spezielle Werkzeugunterstützung zur Ausblendung dieser Details nicht von Nachteil sein kann.

8.2.2 Aktivitätsdiagramm zum Bestellablauf

Das in Abbildung 8.3 dargestellte Aktivitätsdiagramm beschreibt den Ablauf der Handlung bei einer Bestellung. Damit dokumentiert es den Zusammenhang zwischen den in Abbildung 8.2 modellierten Anwendungsfällen.

Darüber hinaus modelliert Abbildung 8.3 Variabilitäten im Ablauf des Bestellprozesses, die aufgrund der (Nicht-)Anwesenheit von Merkmalen entstehen. So ist zum Beispiel gleich zu Beginn die Entscheidung über die Wiederholung einer alten oder die Aufgabe einer neuen Bestellung nur dann erforderlich, wenn die Wiederholung von Bestellungen überhaupt durch das konkrete System unterstützt wird. Entsprechend ist die Bedingung für `select previous order` gestaltet, die analog auch für `repeat order?` gilt. Diese implizite Transitivität entsteht dadurch, daß bei Nicht-Existenz von `select previous order` für `repeat order?` nur eine gültige ausgehende Transition existiert und die Entscheidung damit obsolet wird.

Weitere selektionstypische Variabilitäten ergeben sich aus der Optionalität des Genehmigungsvorganges. Der Übergang zu diesem kann nur erfolgen, wenn die Fähigkeit zur Genehmigung von Bestellungen im System überhaupt vorhanden ist — repräsentiert durch die Bedingung, welche die Anwesenheit der Transition von der Anwesenheit des Merkmals `Approval` abhängig macht. Diese schließt jedoch wie bereits im Use-Case-Modell angedeutet eine Bezahlung per Kreditkarte aus, was durch die «mutex»-Dependency dargestellt wird.

Die genaue Interpretation dieser Beziehung ist in diesem Beispielmodell etwas unscharf, da in dieser Darstellung die Semantik eines Elementes zur Beschreibung von Selektionsvariabilität mit dem ausführen einer bestimmten Aktivität (konkret `give credit card information`) verknüpft wird. Diese Beziehung würde natürlich ebenfalls Auswirkungen auf die zuvor getätigte Entscheidung haben,

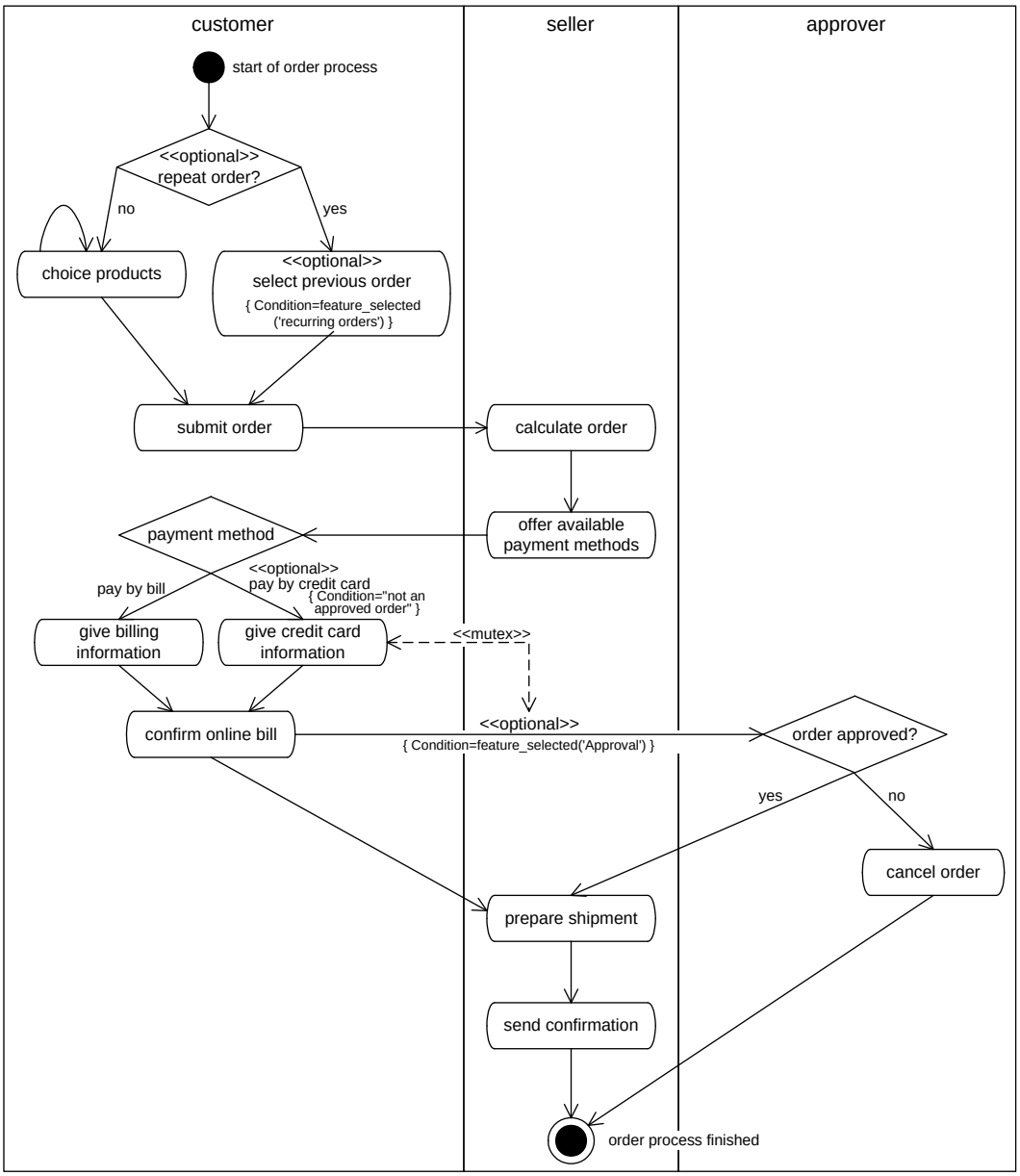


Abbildung 8.3: Prozeßmodell als Aktivitätsdiagramm mit Variabilitäten

da möglicherweise einige Transitionen nicht zur Verfügung stehen. Um das Beispiel abzurunden, wurde dazu die betroffene Transition ebenfalls als optional gekennzeichnet und mit einer entsprechenden Bedingung versehen.

Zur Vervollständigung des Diagramms wäre die Auszeichnung weiterer optionaler Elemente erforderlich. Um jedoch die Komplexität nicht unnötig zu steigern und das Beispiel überschaubar zu halten, wurden diese weggelassen. Ebenso wurde die Bedingung, die entscheidet, ob eine Genehmigung für eine betroffene Bestellung erforderlich ist, nicht modelliert.

Wie das Beispiel an den optionalen Elementen bereits andeutet, sollte die Verwendung der Erweiterungen in Verhaltensmodellen sehr besonnen angewendet werden und insbesondere in einem Profile durch eine entsprechende fortgeschrittene Spezifikation der Semantik der Erweiterungen unterstützt werden.

8.2.3 Komponenten-Modell

Abbildung 8.4 stellt ein erstes vorläufiges Komponentenmodell dar, in dem die einzelnen Bausteine des Bestellsystems bestimmt werden.

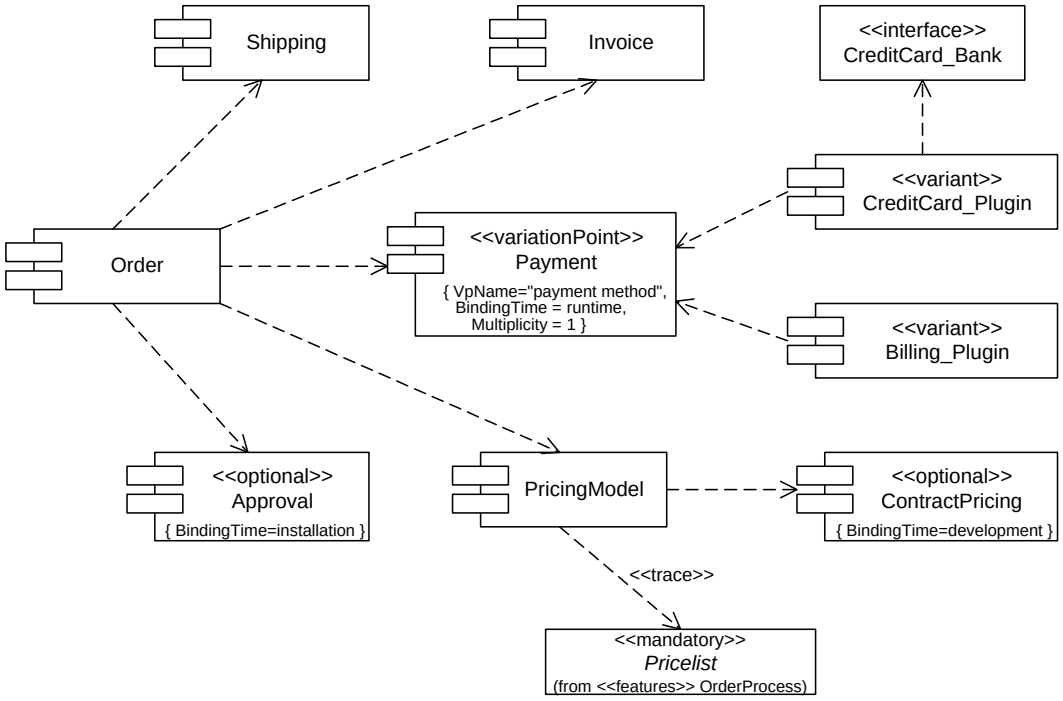


Abbildung 8.4: Komponentenmodell mit Variabilitäten

`Order` ist die zentrale Kontrollinstanz, welche die Abwicklung des Bestellprozesses steuert. In diesem Beispiel werden nur die an der Abwicklung der Bestellung beteiligten Komponenten modelliert, insbesondere ist eine weitere Verfeinerung für die Auswahl der Bestellpositionen denkbar.

Die `Shipping`-Komponente organisiert die Abwicklung der Bestellung, wie zum Beispiel das Initiieren der Logistikprozesse oder die Einstellung entsprechender Aufträge in ein ERP-System. `Invoice` behandelt dagegen die Aspekte der Rechnungsstellung wie die Aufnahme des Auftrages in die Finanzbuchhaltung und das Rechnungswesen.

In der Komponente `PricingModel` wird die komplette Preisermittlung durchgeführt. Die Preisermittlung mittels Listenpreise ist für dieses Modell in dieser Komponente integriert, was durch die «trace»-Abstraktion zu `Pricelist` exemplarisch im Diagramm dargestellt wurde. Solche dokumentatorischen Beziehungen könnten sinnvoll auch als unsichtbare Querverweise in Werkzeugen realisiert werden (siehe [OMG01a, Seite 3-8]). Die Realisierung von vertraglich festgelegten Festpreisen wurde in eine optionale Komponente `ContractPricing` ausgelagert, die nur bei Notwendigkeit derartiger Funktionalität in das System eingebunden wird. Das kann zum Beispiel aufgrund verschiedener mar-

ketingtechnischer Varianten eines Produktes, wie zum Beispiel einer Standard- und einer Enterprise-Edition, erfolgen.

Die Bezahlung einer Bestellung wird durch `Payment` abgewickelt. Diese realisiert dieses abhängig von der gewählten Bezahlmethode durch Einbeziehung von spezifischen Erweiterungen (sogenannten *Plug-Ins*). Bei Kreditkartenbezahlung (`CreditCard_Plugin`) zum Beispiel kann dieses online über eine entsprechende Schnittstelle bei dem Aussteller der Kreditkarte die Zahlungsinformationen verifizieren oder sogar gleich in Rechnung stellen. Die dazu notwendige Schnittstelle wird durch `CreditCard_Bank` repräsentiert. Das `Billing_Plugin` dagegen prüft zum Beispiel, ob es sich bei dem Kunden um eine (juristische oder natürliche) Person handelt, die aus Sicht des Verkäufers vertrauenswürdig genug für eine Bezahlung per Rechnung ist.

Diese Delegation spezifischer Aspekte an spezialisierte Untereinheiten wird durch einen Variationspunkt modelliert. Die zusätzliche und abweichende Beschreibung des Variationspunkt-Namens dokumentiert gegenüber dem Bezeichner des variierenden Elementes (der Komponente, die den Variationspunkt enthält), daß hier nur ein Teil der Funktionalität variiert. Die Verwendung von Dependencies als Verbindung zwischen Varianten und Variationspunkt dokumentiert, daß die konkret zu verwendende Technik zur Implementierung der Variabilität noch nicht entschieden wurde. Da `Payment` nur einen Variationspunkt enthält, entfällt für die Varianten das Tag `VariationPoint`. Die Attributierung des Variationspunktes spezifiziert, daß nur genau eine Variante ausgewählt werden kann, und zwar zur Laufzeit.

Im praktischen Einsatz kann es durchaus Abstufungen bei der Auswahl geben, so beschreibt die Bindezeit *runtime*, das die Entscheidung erst zur Laufzeit getroffen wird. Dabei kann jedoch zum Beispiel bei der Installation durchaus bereits eine Vorauswahl getroffen worden sein — das bedeutet, ein Bindezeitpunkt schließt alle vorangegangenen Zeitpunkte ein beziehungsweise legt den letztmöglichen Entscheidungszeitpunkt fest.

8.2.4 Klassenmodell

Das der Anwendung zugrundeliegende Datenmodell wird durch ein Klassendiagramm beschrieben, dessen Darstellung in Abbildung 8.5 zu finden ist. Die Struktur einer Bestellung als physischer Datensatz wird beschrieben durch die Klasse `Order`. Jede Bestellung besteht aus einer Menge von Bestellpositionen (`OrderItem`) und kann optional die Paketnummer (`PackageNr`) enthalten. Diese ist als optionales Attribute modelliert, daß beim Übersetzen eingebunden wird, wenn das Merkmal `Package_Tracking_Support` unterstützt wird.

Wenn Festpreise auf Vertragsbasis durch die Anwendung unterstützt werden sollen, ist ebenfalls die Klasse `Contract` einzubinden. Diese reflektiert dann ausgehandelte Verträge und werden von Auftragspositionen (`OrderItem`), die einen vertragsgebundenen Preis oder Rabatt nutzen, referenziert. Wird die vertragsgebundene Preisgestaltung nicht unterstützt, entfällt dieses Element grundsätzlich und die Preisgestaltung erfolgt immer über das zugrundeliegende Produkt (Klasse `Product`). Wurde für eine Bestellposition ein Rabatt gewährt (zum Beispiel aufgrund eines Sonderangebotes), wird die Assoziation `discount` genutzt, um die Berechnungsbasis zu referenzieren.

Zusätzlich wird für jede Bestellung der Bezahlstatus festgehalten. Da aufgrund verschiedener Bezahlmethoden unterschiedliche Daten anfallen, wurden die Bezahlinformationen in `PaymentInfo` ausgelagert. Diese stellen einen Variationspunkt dar, der mittels einer abstrakten Klasse und der Ableitung konkreter Subklassen realisiert wird.

Dieses Modell ist insbesondere in bezug auf die Attribute und Operationen noch unvollständig und konzentriert sich auf die Darstellung der Variabilitäten. Da es sich um ein relativ implementierungsnahes Modell handelt, treten nur noch Variabilitäten auf, die entweder bei der Übersetzung des Systems oder zur Laufzeit aufgelöst werden.

Unterschiede, die zur Installationszeit gebunden werden, werden beim Design eines Systems oftmals als statische Konfigurationsvariablen (zum Beispiel Initialisierungsparameter für Server-Kompo-

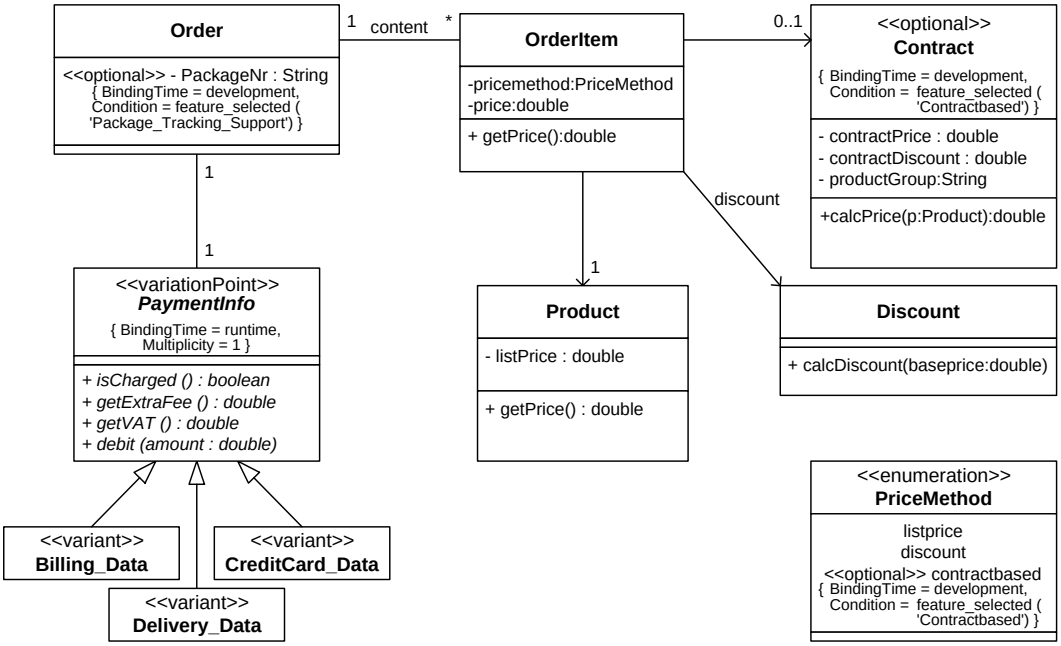


Abbildung 8.5: Klassenmodell mit Variabilitäten

nenten) oder unter Einsatz spezieller Techniken realisiert werden. So stellt zum Beispiel die für das Java-AWT verwendete Technik der Peer-Klassen eine Anwendung dar, in der die Variabilität (das zu verwendende GUI-API) bei der Installation der Ausführungsumgebung (JVM) festgelegt wird.

8.3 Bewertung der UML zur Variantenmodellierung

In diesem abschließenden Abschnitt wird die Eignung der UML zur Modellierung von Variabilität bewertet. Aus dem Kontext dieser Arbeit heraus orientiert sich diese Bewertung dabei an den bisher bestehenden Mechanismen zur Modellierung von Variabilitäten, wie die Beschreibung von Merkmalsdiagrammen und Variationspunkten. Insbesondere der letztere Begriff ist jedoch nicht genau definiert und wird mit verschiedenen Bedeutungen verwendet. Diese unterscheiden sich insbesondere im Detaillierungsgrad der damit beschriebenen Variabilität. Der gemeinsame Nenner dabei ist die Lokalisierung von Variabilität in einem Modell, welches zum Beispiel die Architektur eines Softwaresystems beschreibt. Bei den Merkmalsdiagrammen gibt es ebenfalls verschiedene Varianten, die sich im unterstütztem „Sprachschatz“ unterscheiden.

Im folgendem werden die verschiedenen Betrachtungspunkte für eine Bewertung aufgeführt und anschließend für die Erweiterungen diskutiert.

Allgemeine Kriterien: Einige allgemeine Aspekte für die Bewertung einer Modellierungsnotation wurden bereits im Abschnitt 6.2 beschrieben:

- Die **Verfolgbarkeit** der Entwicklung von Modellelementen (*Traceability*) wird bereits durch die in UML vorhandenen Mittel unterstützt, die durch die Metaklassen `Abstraction` und `Comment` beschrieben werden.

- Die **Skalierbarkeit** von Modellen wird durch die UML zum Teil nur angedeutet. Da ein Ziel der UML die Unabhängigkeit von dem Ausgabemedium und von konkreten Werkzeugen ist, betrifft dieser Aspekt weniger die UML als vielmehr die konkrete Umsetzung in einem Werkzeug.
- Die Belange der **Separation of Concerns** wird durch die Möglichkeiten des Modellmanagements unterstützt, deren Anwendung durch die spezifizierten Erweiterungen in keiner Weise eingeschränkt wird.
- Die Beschreibung der **Evolution** von Modellen wird durch die UML bisher nicht explizit unterstützt, kann jedoch mit Hilfe der vorhandenen Metaklassen erreicht werden (Kommentare, Abstraktionen, Definition benutzerdefinierter Stereotypen).

Spezielle Kriterien für Erweiterungen: Die folgenden Bewertungspunkte ergeben sich direkt aus den Anforderungen, die durch die Konzepte zur Beschreibung von Variabilität gestellt werden:

- Bestehen geeignete Modellelemente (Metamodell-Unterstützung) zur Beschreibung von Variabilitäten?
- Werden geeignete Namensräume (Namensraumproblematik) geschaffen beziehungsweise unterstützt?
- Welche Vorteile bringt die Notation gegenüber bekannten Verfahren?
- Welche Nachteile beziehungsweise Einschränkungen hat die konkrete Modellierungsnotation?

Die Erweiterungen werden im folgendem individuell nach diesen Kriterien bewertet.

8.3.1 Merkmalsmodelle mit UML

Die Merkmalsmodellierung ist ein Konzept, das bisher keinen Eingang in die UML-Spezifikation gefunden hat. Aufgrund dessen gibt es in UML (bis Version 1.4) keine spezifischen Beschreibungsmöglichkeiten für Merkmalsdiagramme und diese müssen durch eine Erweiterung eingeführt werden. Dabei treten einzelne Probleme auf, die nachfolgend diskutiert werden.

Metamodell: Im Metamodell der UML gibt es keine Elemente, die eine semantische Ähnlichkeit mit Merkmalen besitzen. Aufgrund dessen kann eine solche standard-konforme Erweiterung nur unter weitreichender Änderung der Semantik einer bestehenden Metaklasse umgesetzt werden. Ursache ist die Fokussierung der UML auf die objektorientierte Sichtweise, die sich nur eingeschränkt auf die Begriffswelt der Merkmalsmodellierung anwenden läßt. Auf der anderen Seite gibt es bisher keine weiteren Betrachtungen, inwieweit Merkmale mit dem Konzept des Objektes vereinbar sind. So läßt sich eine derartige Integration vermuten, wenn man die Instanziierung von generischen Modellen (Metamodellen[1]) zur Erzeugung eines konkreten Modells betrachtet.

Von diesem Grundsatzproblem — ob man ein Merkmal mit Hilfe eines OO-Elementes modellieren darf — abgesehen enthalten Merkmalsmodelle viele Elemente (insbesondere Beziehungen), die auch in mit UML beschriebenen Modellen vorkommen. Darüber bieten in UML notierte Modelle viel mehr Möglichkeiten, zusätzliche Informationen einzubringen und mehrere Modelle beziehungsweise Diagramme miteinander zu verknüpfen (zum Beispiel Zuordnung eines Interaktionsdiagramms zu einem Use Case), als sie bisher für Merkmalsmodelle bestehen.

Damit läßt sich die UML sehr wohl verwenden, um Merkmalsmodelle zu beschreiben. Sieht man von dem Makel ab, kein semantisch verwendbares Meta-Element für Merkmale zur Verfügung zu haben, können unter Verwendung der Metaklasse `Class` zur Darstellung von Merkmalen alle bisher bekannten Darstellungsformen von Merkmalsmodellen in UML notiert werden.

[1] "Meta" nicht im Sinne der UML, sondern „Metamodell als generisches Modell"

Namensraumproblematik: Der einzige Aspekt, der dabei Einschränkungen auferlegt, ist die Namensraumproblematik: in Merkmalsmodellen müssen nur die Merkmale eines gemeinsamen übergeordneten Merkmalsknoten eindeutig sein — jeder Merkmalsknoten stellt einen eigenen Namensraum dar. In UML-Modellen können Namensräume nur durch `Package` und Spezialisierungen von `Classifier` gebildet werden, so daß Kompromisse bezüglich der Namensgebung von Merkmalen erforderlich sind. Diese Einschränkung wird ein wenig gelindert durch die Möglichkeiten des Modellmanagements, indem Pakete zur Gruppierung von Merkmalen benutzt werden. Dies unterstützt ebenfalls die Unterteilung in funktionelle Einheiten (*Separation of Concerns*).

Zusätzliche Vorteile durch Einsatz der UML: Die Nutzung der UML zur Merkmalsmodellierung bringt für diese weitere Vorteile ein. Dazu gehören die Erweiterungsmöglichkeiten, die eigene Erweiterungen für benutzerdefinierte oder spezielle Zwecke ermöglichen. Auf diese Weise kann ebenfalls die entwickelte Erweiterung erweitert werden.

Daneben stellt die UML eine umfassende Modellierungssprache für Softwaresysteme dar und ermöglicht die konsistente Integration der Merkmalsmodellierung in die Softwareentwicklung. Die starke Verbreitung der UML reduziert daneben den Einarbeitungsaufwand.

Probleme beim Einsatz von UML: Der Nachteil, daß die UML keine Tabellen beziehungsweise Matrizen und damit keine der bisher verwendeten Notationsformen zur Darstellung von Konfigurationen unterstützt kann in erster Linie nur durch die externe Einbindung umgangen werden. Problematisch dabei ist jedoch die Konsistenz der verschiedenen Modelle, so daß dafür noch Lösungsmöglichkeiten untersucht werden sollten. Eine Möglichkeit stellt die Verwendung von Objektdiagrammen zur Darstellung einzelner Konfigurationen dar, die jedoch weiteren Diskussionsbedarf birgt und nur mit spezieller Werkzeugunterstützung sinnvoll einsetzbar sein wird.

Ein weiteres Problem stellt die Komplexität von Diagrammen dar, die mit steigender Größe schlechter darstellbar werden. Dieses Problem gilt prinzipiell für alle UML-Diagramme, könnte für Merkmalsdiagramme jedoch schneller an die Grenzen des Darstellbaren führen. Dazu bietet eine gute Werkzeugunterstützung viele Lösungsmöglichkeiten, die zum Teil an die Skalierbarkeit gekoppelt sind (Ausblendung von Details). Weitere Stichpunkte für die Realisierung einer Werkzeugunterstützung finden sich in [Bass00].

Fazit: Insgesamt läßt sich feststellen, daß die UML mit Hilfe von Erweiterungen gut zur Darstellung von Merkmalsmodellen geeignet ist und darüber hinaus Möglichkeiten bietet, Merkmalsmodelle durch bisher nicht modellierte Zusatzinformationen anzureichern und zu qualifizieren.

Der Nutzen der Merkmalsmodellierung in UML birgt viel Potential, wird im praktischem Einsatz jedoch sehr stark von einer geeigneten und für Probleme der Merkmalsmodellierung optimierten Werkzeugunterstützung abhängig sein.

8.3.2 Modellierung von Variabilitäten in UML

Die Modellierung von Variabilitäten in Softwaresystemen führt stets zu einer Art von generischem Modell, von dem verschiedene konkrete Modelle beziehungsweise Systeme abgeleitet (instanziiert) werden. Daraus ergeben sich verschiedene Konsequenzen für die Notation in UML, da die zusätzlich erforderlich Informationen zur Beschreibung der Variabilität in das Modell integriert werden müssen.

Bisher ist die UML nur für die Modellierung einzelner Softwaresysteme geeignet. Die vorhandenen Modellelemente stammen aus Konzepten für die iterative Entwicklung von Software über verschiedene Phasen (Analyse, Design, Implementierung) hinweg und reflektieren den Konsens der an der Entwicklung der UML Beteiligten.

Aus diesem Grund ist für die Software-Modellierung im Kontext der Produktfamilienentwicklung stets eine Erweiterung notwendig, die die Unterschiede zwischen den einzelnen Produkten modelliert. Diese können mit den standardmäßig vorhandenen Beschreibungsmitteln nicht dargestellt werden. Die Erweiterungsmechanismen der UML bieten diesbezüglich Möglichkeiten, derartige zusätzliche Beschreibungen zu realisieren.

Solche spezifischen Erweiterungen sind prinzipiell abhängig von der Umgebung, in der sie eingesetzt werden. Entscheidend ist zum Beispiel, welche Möglichkeiten für Variabilitäten das dem Entwicklungsprozeß zugrundeliegende Vorgehensmodell vorsieht. Ebenfalls prägend ist die Art und Weise der Softwareerzeugung, ob zum Beispiel mit generativen Verfahren oder der Beschreibung generischer Systeme gearbeitet wird.

Dahingehend kann die Eignung der UML nur für die hier eingesetzten Konzepte — Variationspunkte und Optionale Elemente — untersucht werden.

Mit diesen Konzepten läßt sich grundsätzlich Selektions-Variabilität modellieren. Dabei werden Systemelemente beschrieben, die in verschiedenen Systemen unterschiedlich ausgeprägt oder teilweise nicht vorhanden sind. Nicht ausgezeichnete Elemente werden mit den im Standard vorhandenen Möglichkeiten beschrieben und beschreiben damit die allen Produkten gemeinsamen Bestandteile. Nebenwirkungen, die durch die Variabilität in das Modell eingebracht werden (fehlende Elemente, Konflikte zwischen konkurrierenden Elementen), können damit nur unvollständig oder gar nicht abgebildet werden. Dieses Problem wird ansatzweise durch die in der Erweiterung definierten Constraint-Beziehungen aufgegriffen, erfordert jedoch die Verifikation im realem Einsatz.

Eine interessante Möglichkeit beim Einsatz von UML stellt die Entwicklung von Generatoren dar, die zum Beispiel aus den Modellen mit spezifizierter Variabilität und zusätzlichem Konfigurationswissen konkrete Modelle oder Systeme erzeugen.

8.3.2.1 Variationspunkte in UML

Variationspunkte beschreiben variierende Teile von Systemelementen, die nicht zwangsweise das gesamte Element umfassen müssen. Aufgrund dessen kann es sowohl mehrere, disjunkte Variationspunkte in einem Modellelement geben. Um die Variabilität, genauer deren Ausprägungen, darzustellen, ist es naheliegend, die Beschreibung des Variationspunktes von dem variierenden Element zu trennen und die Variabilität damit explizit zu modellieren. Diese würde dann mittels entsprechender Technologien in das variierende Systemelement eingebunden werden.

Die UML bietet für diesen Zweck keine Metaklasse mit passender oder ähnlicher Semantik. Jede Instanz einer Metaklasse von UML kann in einem System als eigene Entität implementiert werden, was für das „virtuelle Element“ Variationspunkt nicht anwendbar ist. Aus diesem Grund ist ein Kompromiß erforderlich, der den Variationspunkt in das variierende Modellelement integriert — wie es in der UML-Erweiterung für Variationspunkte umgesetzt wurde.

Eine weitere Einschränkung in UML, die mit den nicht vorhandenen Metaelement korreliert, ist der Zwang zu eindeutigen Namen zwischen allen Varianten eines Namensraumes. Normalerweise bildet ein Variationspunkt einen eigenen Namensraum, da die Varianten eindeutig genau einem Variationspunkt zugeordnet werden. Durch die Verschmelzung von Variationspunkt und varrierendem Modellelement äußert sich dieses Problem weniger stark, da jede Variante in einem eigenen UML-Modellelement beschrieben wird und damit bereits einen eindeutigen Namen besitzen muß.

Bei der Verwendung einer UML-Version kleiner als 1.4 ergibt sich ein weiteres Problem: es ist nur maximal ein Stereotyp pro Modellelement zulässig. Damit können als Variationspunkt ausgezeichnete Modellelemente nicht mehr mit anderen Stereotypen versehen werden, die vielleicht durch das Modell erforderlich wären. Dieses Problem wird mit der Umstellung auf UML Version 1.4 obsolet.

Insgesamt lassen sich unter der Berücksichtigung kleinerer Einschränkungen Variationspunkte gut mit UML modellieren. Diese bietet insbesondere die Anwendbarkeit auf verschiedene Modelle und nicht zuletzt die Integration der Modelle zu einem konsistenten Ganzem.

8.3.2.2 Optionale Elemente in UML

Die Modellierung von optionalen Elementen stellt keine weiteren Anforderungen an die UML. Sie können mit den vorhandenen Erweiterungsmechanismen problemlos umgesetzt werden.

Problematisch sind einzig die impliziten Auswirkungen auf von dem optionalen Element abhängige Modellelemente (zum Beispiel Assoziationen und Transitionen). Diese Abhängigkeiten sind oftmals komplex und lassen sich — wenn überhaupt explizit bekannt — unter Umständen nur schwierig dokumentieren. Die UML bietet dazu mit den reichhaltigen Möglichkeiten zur Modellierung, zum Beispiel Abhängigkeiten, Kommentare und Constraints mit oder ohne OCL, viele Ansatzpunkte zur Problemlösung.

8.3.3 Die Erweiterungen für Variabilität in UML

Die in dieser Arbeit entwickelten, standard-konformen Erweiterungen zur Merkmalsmodellierung und zur Beschreibung von Variationspunkten und Optionalen Elementen in UML stellen einen ersten Schritt dar, um die Modellierung von Variabilität in der *Unified Modeling Language* zu ermöglichen.

Erweiterungen und spezifische Anpassungen der entwickelten Notationen sind möglich und werden für einige Vorgehensmodelle für Produktlinien- oder Softwarefamilienentwicklung erforderlich sein. Ergänzend zur minimalen Beschreibung können auch die teilweise bereits spezifizierten (optionalen) TaggedValues verwendet werden.

Die Verwendung der Metaklasse `Class` wird sicherlich noch für einige Irritationen und kontroverse Diskussionen sorgen, stellt jedoch unter den verfügbaren Möglichkeiten die beste Realisierung von Merkmalsdiagrammen in UML dar. Die damit verbundenen Einschränkungen beim Gebrauch derartiger Modellelemente sind teilweise auch für Stereotypen der UML-Spezifikation zu finden (zum Beispiel «type», [OMG01a, Seite 2-30]) und treten bei ordnungsgemäßer Anwendung der Merkmalsmodellierung nicht in Erscheinung.

Kapitel 9

Zusammenfassung

Diese Arbeit arbeitet die vorhandenen Konzepte zur Modellierung von Variabilität im Sinne von Produktlinien beziehungsweise Softwarefamilien auf und konzentriert sich dabei auf die modellierungs- und notationsrelevanten Aspekte.

Auf dieser Grundlage wurden auf der Basis der Unified Modeling Language (UML) verschiedene Erweiterungen für die Modellierung von Variabilität entwickelt, die die standardisierten Erweiterungsmechanismen benutzen. Das Ziel bestand dabei in einer einheitlichen und UML-konformen Notation, die im Sinne der UML unabhängig von der konkreten Vorgehensweise universell anwendbar ist.

Die Merkmalsmodellierung in UML erfüllt dabei die Rolle der Merkmalsmodellierung in der Domänenanalyse und bietet für diese eine einheitliche Notation mit definierter Semantik, die die weitverbreitete UML-Notation aufgreift. Die entwickelte Merkmalsnotation unterstützt alle bisher bekannten Varianten von Merkmalsdiagrammen und kann darüber hinaus die zusätzlichen Möglichkeiten der UML nutzen.

Variabilitäten im Softwaredesign können mit Hilfe von Variationspunkten und optionalen Modellelementen beschrieben werden: diese beschreiben die durch das Merkmalsmodell abstrahierte Variabilität in einem generischen Modell in UML, daß in verschiedenen Entwicklungsphasen (Analyse, Design) zum Einsatz kommt. Dabei wird die Variabilität direkt im Modell, an der Stelle, an der sie vorkommt, modelliert und kann durch verschiedene Attribute detailliert beschrieben werden.

Der praktische Einsatz der entwickelten Notation wurde anhand eines Beispieles aus der eCommerce-Domäne der Firma Intershop demonstriert.

Die abschließende Bewertung zeigt, daß die UML für die Beschreibung von Merkmalsmodellen geeignet ist. Die Modellierung von Variabilitäten in Analyse- und Designmodellen ist dagegen von den zugrundeliegenden Entwicklungsprozessen und -technologien abhängig. Mit Variationspunkten und optionalen Elementen läßt sich Variabilität in UML explizit modellieren, dabei werden generische Modelle beschrieben.

Insgesamt bietet die Verwendung der UML viele Vorteile, die für die Modellierung von Variabilität genutzt werden sollten.

9.1 Zukünftige Aktivitäten

Interessante Fragestellungen, die sich aus dieser Arbeit ergeben oder diese fortführen könnten, sind nachfolgend in Stichpunkten aufgezählt:

- Vertiefung der Merkmalsmodellierung, zum Beispiel Untersuchung der Auswirkungen unterschiedlicher Bindezeiten, Verwendung von «weakConstraint» (↗ Abschnitt 7.2.1.6)

- Modellierung der Merkmalsauswahl in UML (zum Beispiel Verwendung von Objektdiagrammen zur Konfigurationsdarstellung ↗ Abschnitt 7.2.1.8)
- Realisierung von Variationspunkten, Wechselwirkungen zwischen Varianten von multiplen Variationspunkten (↗ Abschnitt 7.3.1.6)
- Untersuchung der Anwendung der Erweiterungen für Verhaltensmodelle (↗ Abschnitt 7.5) und in Verbindung mit Echtzeitmodellierung in UML (ROOM-UML)
- Verwendung von *design patterns* zur Umsetzung von Variabilitäten (Hinweise in [Bosch00, Seite 207f], [Sharp00b], [Czarnecki00, Seite 27], Buch *Analysis Patterns* von Fowler)
- Integration in Entwicklungsprozesse mit Produktfamilien (↗ Abschnitt 7.6.2, Kapitel 3)

Anhang A

Durchführung der Merkmalsanalyse bei Intershop

Eine Merkmalsanalyse ist nicht Bestandteil der Aufgabe zu dieser Arbeit. Da bei Intershop bisher keine Merkmalsmodellierung durchgeführt, mußten die zur Modellierung notwendigen Daten jedoch erst erarbeitet werden. Aufgrund dessen wurde eine prototypische Merkmalsanalyse durchgeführt.

Um diese Informationen von der Arbeit abzugrenzen, dient dieser Anhang der Wiedergabe der im praktischem Einsatz gewonnenen Erfahrungen. Dabei handelt es sich um unvollständig aufbereitete Daten, die der weiteren Verfeinerung und Verifikation bedürfen.

Die Analyse wurde teilweise am Beispiel einer Referenzapplikation durchgeführt. Referenzapplikationen werden bei Intershop verwendet, um exemplarisch Einsatzzwecke von enfinity zu verdeutlichen. Enfinity ist eine Plattform für die Realisierung von eCommerce-Anwendungen.

Das konkrete Vorgehen orientierte sich an den Prozeßbeschreibungen in [Kang90] und [Kang98]. Als Quellen wurden hauptsächlich die Requirements und Produktspezifikationen der Referenzapplikation und der zugrundeliegenden Plattform verwendet.

Zur Erstellung des Modells wurde das CASE-Tool Rational Rose verwendet. Damit konnten zum einem Merkmale und ihre Beziehungen flexibel bearbeitet werden und zum anderem wurde in Form der Klassenbeschreibung die Möglichkeit genutzt, eine Merkmalsbeschreibung direkt im Modell abzulegen. Ebenfalls genutzt wurde die Fähigkeit zum *Model Management*, also die Verwaltung der Merkmale und dazugehörigen Diagramme in verschiedenen Paketen.

Während der Durchführung der Merkmalsanalyse wurden das auf Seite 101 dargestellte Merkmalsdiagramm erstellt. Dieses kann nur als prototypisches Modell verstanden werden — insbesondere zeigte sich, daß eine Merkmalsanalyse ein zeitaufwendiges Vorhaben werden kann und detaillierte Kenntnisse der Domäne erfordert. Für ein vollständiges Merkmalsmodell sind neben einer Vervollständigung des Modells die textuelle Definition jedes Merkmales sowie die Dokumentation von Auswahlbegründungen erforderlich. Teilweise wurden einige Beziehungstypen noch nicht identifiziert und sind zur Darstellung der Abstraktionshierarchie als Assoziationen notiert worden.

Ein interessanter Aspekt in Abbildung A.1 ist die genauere Spezifikation der «requires»-Abhängigkeit zwischen `heterogene environment` und `platform support` mit einem Constraint.

A.1 Prüfung des Merkmalmodelles, Kontrollfragen

Neben dem Modell entstanden bei der Durchführung der Merkmalsanalyse einige Kriterien, die zur Identifikation und Verifikation des Merkmalsmodells dienen:

1. Ein Merkmal sollte in sich abgeschlossen, selbsterklärend sein
2. Merkmale sind *Abstraktionen* von Requirements!
3. Merkmale beschreiben Eigenschaften aus Sicht des Benutzers (zum Beispiel Kunde oder Anwendungsentwickler)
4. Merkmale werden ausgewählt, um ein spezifisches Produkt zu charakterisieren: die Auswahlregeln bestimmen den Typ
5. Das Merkmalsmodell sollte detailliert genug sein, um die wesentlichen Unterschiede zwischen den Produkten zu erfassen und damit die Entwicklung gemeinsamer (beziehungsweise generischer) Bestandteile zu ermöglichen.
6. Beziehungen zwischen Merkmalen können in die Merkmalsbeschreibung übernommen werden: die Komposition von Merkmalen kann als „is_part_of“ gelesen werden, die Generalisierung / Spezialisierung als „is_a“ beziehungsweise „is_kind_of“
7. Variablen sind keine Merkmale; Merkmalen wird kein Wert zugewiesen

Als Kontrollfragen (ohne Anspruch auf Vollständigkeit) wurden erarbeitet:

- Handelt es sich um eine für den Benutzer eines Produktes relevante Eigenschaft?
- Handelt es sich um eine charakteristische Eigenschaft der Domäne beziehungsweise der Produkte?
- Bringt der Merkmalsknoten neue Informationen ein?
- Erfaßt das betrachtete Merkmal eine Eigenschaft (der Domäne oder eines Produktes)?
- Ist der Knoten mit einem prägnanten Bezeichner versehen?
- Stellt der übergeordnete Knoten eine Abstraktion / Zusammenfassung des aktuellen Merkmals dar?
- Ergeben die modellierten Auswahlregeln (Merkmalstyp) eine sinnvolle Produktkonfiguration?
- Sind die Verfeinerungen des Knotens vollständig beschrieben oder fehlen noch Details (weitere Unter-Knoten)?

Die Erklärung der Bedeutung von Merkmalen für die Modellierung soll mit folgendem Zitat aus FeatuRSEB (zu finden in [Czarnecki00, Seite 79]) abgeschlossen werden:

> „... not everything that could be a feature should be a feature. Feature descriptions need to be robust and expressive. Features are used primarily to discriminate between choices, not to describe functionality in great detail; such detail is left to the use case or object models.“

A.2 *technical features*

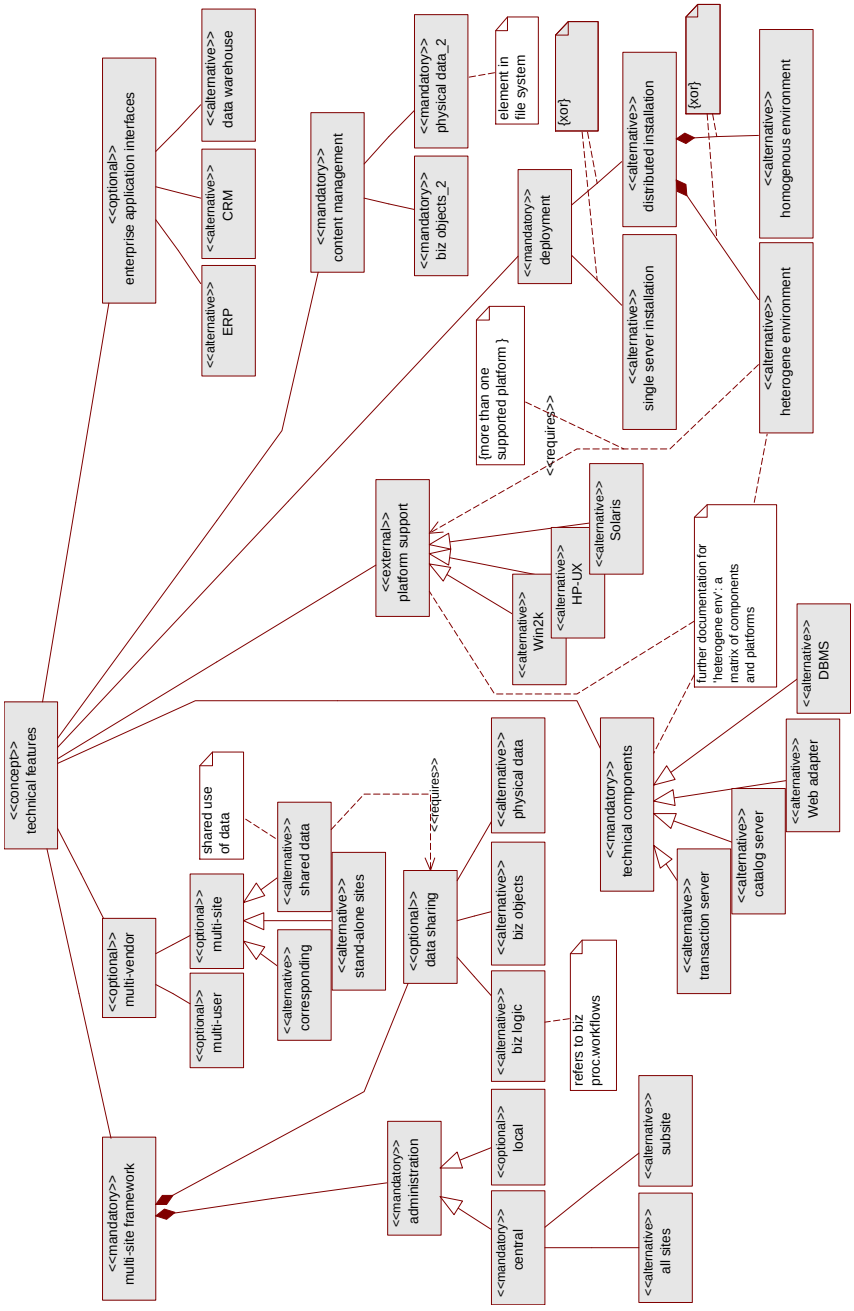


Abbildung A.1: Merkmalsdiagramm zu technischen Merkmalen

Glossar

“4+1“-Sicht – Darstellung der Zusammenhänge zwischen den Diagrammtypen der UML, in der verschiedene Aspekte durch verschiedene Diagrammarten beziehungsweise -sichten modelliert und durch einen zentralen Darstellungstyp miteinander zu einem konsistenten Modell verbunden werden [Kruchten95].

Aspekte (*aspects*) – im allgemeinem Sprachgebrauch ein relevanter Teil eines Ganzen. In AOP werden mit Aspekten Eigenschaften eines Systems beschrieben, die über mehrere Komponenten (*cross-cutting*) verteilt sind — typischerweise auf Implementierungsniveau. ↗ [Kiczales97]

Assets (Bausteine) sind Einheiten von Informationen, Code und andere Produkte des Softwareentwicklungsprozesses, die greifbare Arbeitsergebnisse darstellen und während der Produktlinienentwicklung entstehen. Beispiele für Assets sind: Requirements, Use cases, Modelle, Architekturen, Frameworks, Quellcode (-fragmente), Komponenten, Algorithmen, Testfälle, Dokumentation.

binden bezeichnet den Vorgang, bei dem zur Übersetzung oder zur Laufzeit eines Programms entschieden wird, welche Routine für einen Methodenaufruf zu adressieren ist. Dynamisches Binden kommt häufig im Zusammenhang mit Vererbung und Polymorphie zum Einsatz, wenn zum Zeitpunkt der Übersetzung (Kompilierung) noch nicht bestimmbar ist, welche Code-Adresse tatsächlich angesprungen werden muß.

binding time (Bindezeitpunkt) – damit wird der Zeitpunkt bezeichnet, zu dem die Festlegung der Variabilität erfolgt. Dabei wird zum Beispiel entschieden, ob ein optionales Element einzubinden ist und welche Variante von Alternativen zu wählen ist.

Constraints werden in UML verwendet, um zusätzliche Einschränkungen über den Modellelementen zu beschreiben.
Im Merkmalsmodell werden damit Abhängigkeiten (*requires*, *mutual exclusion*) zwischen Merkmalen beschrieben, aus denen sich auch Kompositionsregeln ableiten lassen.

COTS-Komponenten sind kommerzielle Komponenten, die unter Verwendung einer Komponententechnologie (JavaBeans, EJB, COM+) von Drittherstellern für produziert werden. Eine Diskussion der Vor- und Nachteile findet sich unter anderem in [Northrop, Folien 43-48].

DLL – eine DLL (*dynamic-link library*) wird zur Laufzeit als externes Modul eingebunden. Typisch dafür ist, das der Programmcode mit anderen Programm gemeinsam genutzt werden kann, im Gegensatz zum statischen Linken, wo die Bibliothek fest in das Programm integriert wird.

Domäne (*domain*) ist ein ausgewähltes Gebiet von (Anwendungs- oder Fach-) Wissen. Die Bestimmung erfolgt über die enthaltenen Objekte, Operationen und Beziehungen, die von den entsprechenden Spezialisten als relevant betrachtet werden. Einzelne Domänen sind zum Beispiel Sicherheit, Datenbanken oder Benutzerschnittstellen.

Domänenanalyse (*domain analysis*)– Teilphase der Domänenentwicklung (*domain engineering*), vergleichbar mit der Analysephase in herkömmlichen Prozeßmodellen.

domain engineering – Entwicklung einer Domäne, in der Regel über mehrere Phasen (zum Beispiel Analyse, Design und Implementierung). Als Ergebnis treten die Assets auf. Die Domänenentwicklung wird teilweise auch als „*Development for Reuse*“ bezeichnet. ↗ *system engineering*

eCommerce – die Gesamtheit aller betrieblichen Aktivitäten und Verfahren, die unter Verwendung elektronischer Medien zur Optimierung der Wertschöpfungskette sowie zur Pflege und Erweiterung des Netzwerkes der Geschäftsbeziehungen eines Unternehmens beitragen.

Entwurfsmuster (*design patterns*) beschreiben generalisierte Lösungsideen zu wiederkehrenden Entwurfsproblemen. Diese liegen in der Regel nicht als Code vor, sondern beschreiben lediglich den Lösungsweg.

ERP-Systeme übernehmen das operative Management von Massendaten (Stamm- und Bewegungsdaten), unterstützen die taktische Ebene durch Reporting und Aggregationstools und beinhalten Schnittstellen für dedizierte Tools. Ein typisches Beispiel ist SAP/R3.

High-End-Produkte – hochqualitative, komplexe Systeme mit relativ umfangreicher Funktionalität. ↗ Low-End-Produkte

hot spot ↗ Variationspunkt.

Icon – kleine Grafik, die als Symbol für einen Sachverhalt steht.

Klasse, OO-Klasse – repräsentiert eine generische Beschreibung einer Menge von Objekten, mit gemeinsamen Attributen, Operationen, Relationen und Semantik. Dabei werden Verhalten, Zustand und Aktionen der Instanzen in genereller Art und Weise beschrieben.

Komponente – repräsentiert eine modulare, installierbare und ersetzbare Einheit eines Systems, die eine Menge von wohldefinierten Schnittstellen anbietet und deren Implementierung kapselt (nach [OMG01a]). ↗ [Atkinson00, Hitz99]

Konzept – eine Generalisierung von objektorientierten ↗Klassen, die keine vordefinierte Semantik beinhaltet. Ein Konzept beschreibt eine Klasse von Phänomenen und ist im Gegensatz zu Klassen auch von Zeit, Umfeld und anderen Faktoren abhängig.

linken, Link-Vorgang – hierbei werden die symbolischen Namen im kompilierten Code aufgelöst und an konkrete Adressen gebunden. Auf diese Weise können Module unabhängig von ihrem späteren Einsatz in einem ablauffähigen Programm übersetzt werden.

Low-End-Produkte sind in der Regel einfach strukturierte Systeme mit einfachen Funktionen oder begrenztem Funktionsumfang. ↗ High-End-Produkte

Merkmal (*feature*) – wird verstanden als herausragende oder besondere Eigenschaft oder Charakteristik eines Softwaresystems oder von Systemen. In einer eingeschränkten Definition werden mit Merkmalen nur Eigenschaften beschrieben, die für den Endbenutzer sichtbar ist.

Methode ↗ Vorgehensmodell

Merkmalsmodell (*feature model*) – besteht aus Merkmalsdiagramm, Merkmalsbeschreibung und zusätzlichen Entscheidungselementen. Es beschreibt die für die Charakterisierung eines Problembereiches (Produktlinie, Produktfamilie, Domäne) erforderlichen Merkmale und ihre Beziehungen untereinander.

Middleware ist ein Gattungsbegriff für Software, die verschiedene Dienste bereitstellt beziehungsweise implementiert, jedoch keine Endanwendungen realisiert. Sie ist sozusagen zwischen Betriebssystem und Applikationen angesiedelt.

Modellgesteuerte Analyse (*model-driven analysis*) – nutzt mehrere komplementäre Sichten, um Informationen über den Problembereich zu übermitteln.

Produktfamilie – ist eine Menge von verwandten Systemen, die auf einer gemeinsamen Basis von Kernbausteinen aufbauen.

Produktlinie (*product line*) – eine Gruppe von Produkten mit einer gemeinsamen, explizit verwalteten Menge von Merkmalen, die die charakteristischen Anforderungen eines bestimmten Marktes erfüllen (nach [Czarnecki00]).

Produktlinienarchitektur – eine Softwarearchitektur für eine Produktlinie, welche die Architektur für alle Produktlinienmitglieder verkörpert. Sie dient als Grundlage für die Architektur eines Produktes, enthält in ihrer Spezifikation jedoch zusätzliche Angaben zu gemeinsamen und variablen Bestandteilen, Aussagen für die instanziierten Produktarchitekturen und Konsistenzregeln für die Generalität (nach [Bayer00]).

Produktlinienentwicklung (product line engineering) ist der Prozeß der Entwicklung von Assets, aus denen die individuellen Systeme (Produkte) der Produktlinie abgeleitet werden. Wird teilweise auch mit Domänenentwicklung gleichgesetzt.

Produktlinieninstanz – die Instanz einer Produktlinie ist ein einzelnes Produkt. Typischerweise besitzt es nur eine Teilmenge der Eigenschaften, die für die gesamte Produktlinie modelliert wurden.

Scoping bezeichnet den ersten Entwicklungsschritt einer Einheit, in dem die Grenzen dieser Einheit (zum Beispiel eine Domäne, Produktfamilie oder Produktlinie) festgelegt und gegenüber anderen Einheiten oder der Umgebung abgegrenzt wird.

Separation of Concerns – die Trennung der Funktionen beschreibt das Prinzip, unterschiedliche Klassen von Anforderungen beziehungsweise Absichten in unterschiedlichen Teilen eines Softwaresystems (Subsysteme, Module) zu kapseln und dadurch klar voneinander zu trennen.

Softwaresystem – ein System (oder Teilsystem), dessen Komponenten aus Software bestehen.

system engineering – Entwicklung eines konkreten Systems (Anwendung, Applikation) — bei Nutzung eines Domänenmodells aus einer Domänenbeschreibung. Wird in Kontrast zu *domain engineering* als „*Development with Reuse*“ bezeichnet.

Variante (*variant*) – Implementation eines Variationspunktes. Wird teilweise auch als Parameter bezeichnet. ↗ Variationspunkt

Variationspunkt (*variation point*, *hot spot*) – ist ein ausgezeichneter Punkt, an dem eine konkrete Ausprägung von Variabilität durch Auswahl einer Variante aufgelöst wird.

Vorgehensmodell (*approach*, *method*) wird teils auch als Entwicklungsmethode bezeichnet und beschreibt einen Entwicklungsprozeß für Software. Ziel dabei ist es, zum Beispiel durch die Definition von Arbeitsergebnissen und Tätigkeiten reproduzierbare Ergebnisse und deren Qualität sicherzustellen.

wechselseitiger Ausschluß (*mutual exclusion*, mutex) ist eine Relation zwischen zwei oder mehr Elementen, die sich gegenseitig ausschließen. Das heißt, es kann höchstens eines der beteiligten Elemente aktiv beziehungsweise enthalten sein (mathematische XOR-Relation).

Framework (Rahmenwerk) ist eine Menge von vorgefertigten Softwarebausteinen, die zur Erstellung von spezifischen Software-Lösungen verwendet, erweitert und angepaßt werden können (nach Taligent).

weak constraint – eine Einschränkung, deren Gültigkeit von bestimmten Umweltfaktoren abhängig ist und aufgrund dessen nicht immer auftritt. ↗ *constraint*

Abkürzungsverzeichnis

AOP	Aspect-Oriented Programming
API	Application Programming Interface
AWT	Abstract Windowing Toolkit
CBSE	Componenent-Based Software Engineering
CDT	Customization Decision Tree
COM	Component Object Model
COTS	Components Off-the-Shelf, auch „Commercial Off-the-Shelf“
CPS	Car Periphery Supervision
DE	Domain Engineering
DLL	Dynamic-Link Library
DRM	Digital Rights Management
EJB	Enterprise JavaBeans
ERP	Enterprise Resource Planning
FAST	Family-oriented Abstraction, Specification and Translation
FODA	Feature-Oriented Domain Analysis
FORM	Feature-Oriented Reuse Method
GUI	Graphical User Interface
JVM	Java Virtual Machine
KobrA	Komponentenbasierte Anwendungsentwicklung
MBSE	Model-Based Software Engineering
mutex	mutual exclusion
OCL	Object Constraint Language
OO	objektorientiert, Objektorientierung
PL	Product Line
PLA	Product Line Architecture
PLP	Product Line Practice
PLSE	Product Line Software Engineering
PuLSE	Product-Line integrated Software Engineerung
RSEB	Reuse-orientied Software Engineering Business
SCMBU	Single Company Multiple Business Unit
SE	Software Engineering
SEI	Software Engineering Institute (an der Carnegie Mellon Universität)
SOC	Separation Of Concerns
SOP	Subject-Oriented Programming
SPLIT	Software-Productline Integrated Technology
UML	Unified Modeling Language
VMM	Virtual MetaModel
Vp	Variationspunkt, Variation Point

Literaturverzeichnis

[Ahmad00] W. Al-Ahmad, E. Steegmans, *Inheritance in Object-Oriented Languages: Requirements and Supporting Mechanisms*, in JOOP, Jan. 2000, Vol. 12 No. 8

[Anastaso] M. Anastasopoulos, C. Gacek, *Implementing Product Line Variabilities*, Fraunhofer Institute for Experimental Software Engineering (IESE)

[Anastaso00] M. Anastasopoulos, J. Bayer, O. Flege, C. Gacek, *A Process for Product Line Architecture Creation and Evaluation — PuLSE-DSSA*, Version 2.0, Fraunhofer IESE Report No. 038.00/E, Juni 2000,
http://www.iese.fhg.de/pdf_files/iese-038_00.pdf

[Arango93] G. Arango, E. Schoen, R. Pettengill, *Design as Evolution and Reuse*, Proceedings of the 2nd International Workshop on Software Reusability (IWSR-2), Lucca: IEEE Press, 1993

[Atkinson00] C. Atkinson, J. Bayer, D. Muthig, *Component-Based Product Line Development: The KobrA Approach*, Proceedings of the First International Software Product-Line Conference (SPLC-1), August 2000

[Atkinson01] C. Atkinson et al, *Component-based Product Line Engineering with UML*, Addison-Wesley, August 2001

[Bass00] L. Bass et al, *Fourth Product Line Practice Workshop Report*, SEI Technical report CMU/SEI-2000-TR-002, Februar 2000,
http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr002.pdf

[Basset97] P. Bassett, *Framing Software Reuse: Lessons from the Real World*, Prentice Hall, 1997

[Bayer99a] J. Bayer et al, *PuLSE: A Methodology to Develop Software Product Lines*, Proceedings of the 5th Symposium on Software Resuability (SSR'99), Mai 1999,
http://www.iese.fhg.de/pulse

[Bayer99b] J. Bayer, D. Muthig, T. Widen, *Customizable Domain Analysis*, Generative and Component-based Software Engineering, 1st International Symposium, GCSE '99

[Bayer00] J. Bayer, *Lecture: Software Product Lines and Reengineering*, Fraunhofer Institut Experimentelles Software Engineering (IESE), Dezember 2000

[Boellert00] K. Böllert, I. Philippow, *Erfahrungen bei der objektorientierten Modellierung von Produktlinien mit FeatuRSEB* Proceedings of 1. Deutscher Software-Produktlinien Workshop (DSPL-1), November 2000, IESE

[Booch99] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language Guide*, Addison-Wesley, 1999

[Bosch98] J. Bosch, *Product-Line Architectures in Industry: A Case Study*, Proceedings of the 21st International Conference on Software Engineering, November 1998, http://www.ipd.hk-r.se/bosch/papers/PLA-casestudy.PDF

[Bosch99] J. Bosch, *Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study*, Proceedings of the 1st Working IFIP Conference on Software Architecture, Februar 1999, http://www.ipd.hk-r.se/bosch/papers/Asset-casestudy.PDF

[Bosch00] J. Bosch, *Design & Use of Software Architectures — Adopting and evolving a product-line approach*, Addison-Wesley, 2000

[Cheesman01] J. Cheesman, J. Daniels, *UML Components — A Simple Process for Specifying Component-based Software*, Addison-Wesley, 2001

[Cheong] Y.C. Cheong, S. Jarzabek, *Handling Variant Requirements in Generic Architectures for Software System Families*

[Cheong98] Y.C. Cheong, S. Jarzabek, *Modelling Variant User Requirements in Domain Engineering for Reuse*, Proceedings of the 8th European-Japanese Conference on Information Modeling and Knowledge Bases, Finnland, Mai 1998

[Chhut] R. Chhut, *Fundamentals of Domain Engineering*, Presentation

[Choi99] B.W. Choi, K.B. Jang, C.H. Kim, K.S. Wang, K.C. Kang, *Development of Software for the Hard Real-time Controller using Feature-Oriented Reuse Method and CASE Tools*, Proceedings of the IEEE International Symposium on Computer Aided Control System Design, August 1999

[Clauss01] M. Clauß, *A proposal for uniform abstract modeling of feature interactions in UML*, Workshop „Feature Interaction in Composed Systems“, ECOOP'01, April 2001, http://i44w3.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/program.html

[Cleaveland01] C. Cleaveland, *Program generators with XML and Java*, Prentice Hall, 2001

[CMU] *Model-based Software Engineering*, Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/mbse/

[Coad99] P. Coad, E. Lefebvre, J. de Luca, *Java Modeling in Color with UML — Enterprise Components and Process*, Prentice Hall, 1999

[Cohen92] S. Cohen, J. Stanley, S. Peterson, R. Krut, *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain*, Technical Report, CMU/SEI-91-TR-28, Juni 1992, http://www.sei.cmu.edu/pub/documents/91.reports/pdf/tr28.91.pdf

[Coleman94] D. Coleman et al, *Object Oriented Development — The FUSION Method*, Prentice-Hall, 1994

[Coplien] J. Coplien, D. Hoffman, D. Weiss, *Commonality and Variability in Software Engineering*, Bell Labs,
http://www.bell-labs.com/user/cope/Mpd/IeeeNov1998/

[Coplien99] J. Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999

[Coriat00] M. Coriat, J. Jourdan, F. Boisbourdin, *The SPLIT method*, Proceedings of the First International Software Product-Line Conference (SPLC-1), August 2000

[Czarnecki00] K. Czarnecki, U. Eisenecker, *Generative Programming — Methods, Tools and Applications*, Addison-Wesley, 2000

[Davis96a] A. Davis, M. Leffingwell, A. Dean, *Using Requirements Management to Speed Delivery of Higher Quality Applications*, Rational Software Corporation, 1996

[Davis96b] A. Davis, D. Leffingwell, *Using Requirements Management to Speed Delivery of Higher Quality Applications*, Rational Software Corporation, 1996,
http://www.rational.com/media/whitepapers/696wp.pdf

[Flege00] O. Flege, *System Family Architecture Description Using the UML*, IESE-Report 092.00/E, Dezember 2000,
http://www.iese.fhg.de/pdf_files/iese-092_00.pdf

[Gamma95] E. Gamma et al, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995,
http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

[Gomaa00] H. Gomaa, *Object-Oriented Analysis and Modeling for Families of Systems with UML*, 6th International Conference on Software Reuse (ICSR-6), Juni 2000

[Griss98] M. Griss, J. Favaro, M. d'Alessandro, *Integrating Feature Modelling with the RSEB*, International Conference on Software Reuse, Juni 1998,
http://www.hpl.hp.com/reuse/papers/icsr98.htm

[Griss00a] M. Griss, *Implementing Product-Line Features By Composing Component Aspects*, Proceedings of the First International Software Product-Line Conference (SPLC-1), August 2000,
http://www.hpl.hp.com/reuse/papers/splc1-griss.pdf

[Griss00b] M. Griss, *Implementing Product-Line Features with Component Reuse*, Sixth International Conference on Software Reuse (ICSR6), Juni 2000,
http://www.hpl.hp.com/reuse/papers/icsr2000-griss.pdf

[Hein00] A. Hein, M. Schlick, R. Vinga-Martins, *Applying Feature Models in Industrial Settings*, Proceedings of the First International Software Product-Line Conference (SPLC-1), August 2000

[Hitz99] M. Hitz, G. Kappel, *UML @ Work — Von der Analyse zur Realisierung*, dpunkt.verlag, 1999

[Jacobson92] I. Jacobson et al, *Object-Oriented Software Engineering: A Use Case driven Approach*, Addison Wesley, 1992

[Jacobson97] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse — Architecture, Process and Organization for Business Success*, Addison-Wesley Longman, 1997

[Jarzabek00] S. Jarzabek, *Product Line Approach*, NetObjectDays, Tutorial at GCSE'00, 2000

[Kang90] K. Kang et al, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report No. CMU/SEI-90-TR-2, November 1990, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

[Kang98] K. Kang et al, *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*, Annals of Software Engineering 5, 1998, http://www.postech.ac.kr/cse/se/Publication/ase-accepted-98-2-1.doc

[Kiczales97] G. Kiczales et al, *Aspect-Oriented Programming*, LNCS 2041, Springer Verlag, Juni 1997

[Knauber] P. Knauber, J.-F. Girard, *Vorlesung „Software Product Lines and Reengineering"*, Wintersemester 2000/2001, http://www.iese.fhg.de/lectures/knauber/WS2000/index.html

[Knauber00] P. Knauber, *Planung und Realisierung von Produktfamilien mit PuLSE*, Fraunhofer Institut für Experimentelles Software Engineering (IESE), November 2000

[Kruchten95] P. Kruchten, *The 4+1 View Model of Architecture*, IEEE Software, November 1995, http://www.rational.com/media/whitepapers/Pbk4p1.pdf

[McCabe93] R. McCabe, G. Campbell, *Reuse-driven Software Processes Guidebook*, STARS reports SPC-92019-CMC, Version 02.00.03, Novemer 1993, http://www.domain-specific.com/RSPgb.tar.gz

[Medvidovi99] N. Medvidovic, *Modeling software Architectures in UML*, Workshop on Architecture and UML, Denver, April 1999

[Muthig00a] D. Muthig, *Documenting and Controlling Product Lines Using the UML*, Product Line Architecture Workshop held in conjunction with the first Software Product-Line Conference (SPLC-1), Denver, 2000

[Muthig00b] D. Muthig, J. Bayer, *Helping Small and Medium-Sized Enterprises in Moving Towards Software Product Lines*, Software Product-Line Workshop held in conjunction with the 22nd International Conference on Software Engineering (ICSE), 2000

[Northrop] L. Northrop, *SEI Product Line Practice Framework, Version 2*, http://www.sei.cmu.edu/activities/plp/fmwk_v2_sym99/index.htm

[Northrop99] L. Northrop, *Keynote: What is a Product-Line?*, Workshop on Object Technology for Product-Line Architectures, Juni 1999, Projekt: Product-line Realisation and Assessment in Industrial Settings (PRAISE), ESPRIT Project 28651, http://www.esi.es/Projects/Reuse/Praise/pdf/product-lines.pdf

[OMG00] Object Management Group (OMG), *OMG Unified Modelling Language Specification*, Version 1.3, März 2000, http://cgi.omg.org/cgi-bin/doc?formal/00-03-01.pdf

[OMG01a] Object Management Group (OMG), *OMG Unified Modelling Language Specification*, Version 1.4 Draft, Februar 2001, http://cgi.omg.org/cgi-bin/doc?formal/01-02-14.pdf

[OMG01b] *OMG UML 1.4 – Appendix A: Issues Database Report*,
http://www.jeckle.de/files/UMLv1.4Rev.pdf

[Richter99] C. Richter, *Designing Flexible Object-oriented Systems with UML*, MacMillan Technical Publishing, 1999

[Robak01] S. Robak, B. Franczyk, K. Politowicz, *Extending UML for modelling variabilities for system families*, zu erscheinen in „International Jounal of Applied Mathematics und Computer Science“

[Simons96] M. Simons, D. Creps, L. Levine, D. Allemang, *Organization Domain Modeling (ODM) Guidebook*, Version 2.0, Informal Technical Report from STARS, STARS.VC-A205/001/00, Juni 1996

[Souza99] D. D'Souza, A. Wills, *Objects, Components and Frameworks with UML — The Catalysis Approach*, Addison-Wesley, 1999

[Stevens99] P. Stevens, *UML for describing product-line architectures?*, Workshop on Object Technology for Product-Line Architectures, Juni 1999, Projekt: Product-line Realisation and Assessment in Industrial Settings (PRAISE), ESPRIT Project 28651,
http://www.esi.es/Projects/Reuse/Praise/pdf/ses2-4.pdf

[Sharp00a] D. Sharp, *Containing and Facilitating Change Via Object Oriented Tailoring Techniques*, Proceedings of The First Software Product Line Conference (SPLC-1), Denver, August 2000

[Sharp00b] D. Sharp, *Component-Based Product Line Development of Avionics Software*, Proceedings of The First Software Product Line Conference (SPLC-1), Denver, August 2000

[Svahnbrg] M. Svahnberg, J. van Gurp, J. Bosch, *On the notion of Variability in Software Product Lines*,
http://www.ipd.hk-r.se/bosch/papers/SPLVariability.pdf

[Svahnbrg99a] M. Svahnberg, J. Bosch, *Evolution in Software Product Lines: Two Cases*, Journal of Software Maintenance, Vol. 11, No. 6, 1999,
http://www.ipd.hk-r.se/msv/publications/ESAPS_techreport.pdf

[Svahnbrg99b] M. Svahnberg, J. Bosch, *Characterizing Evolution in Product-Line Architectures*, Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications (SEA'99), Oktober 1999,
http://www.ipd.hk-r.se/msv/publications/299-055.pdf,
http://www.ipd.hk-r.se/bosch/papers/PLAevolution.pdf

[Svahnbrg99c] M. Svahnberg, J. Bosch, *A Case Study on Product Line Architecture Evolution*, Proceedings of the 2nd Nordic Workshop on Software Architecture (NOSA'99), Ronneby, August 1999,
http://www.ipd.hk-r.se/msv/publications/nosa99.pdf

[Svahnbrg00] M. Svahnberg, J. Bosch, *Issues Concerning Variability in Software Product Lines*, Proceedings of the 3rd International Workshop on Software Architectures für Product Families, Springer Verlag, 2000,
http://www.ipd.hk-r.se/msv/publications/iwsapf3_18.pdf

[Vici00] A.D. Vici, N. Argentieri, A. Mansour, M. d'Allessandro, J. Favaro, *FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry*, Sixth International Conference on Software Reuse (ICSR6), Juni 2000

[Warmer] J. Warmer, *The future of UML*, Klasse Objecten, http://www.klasse.nl/english/uml/uml2.pdf

[Weiss99] D. Weiss, C.T. Robert Lai, *Software Product-Line Engineering — A Family-Based Software Development Process*, Addison-Wesley, 1999

[Withey96] J. Withey, *Investment Analysis of Software Assets for Product Lines*, Technical Report, CMU/SEI-96-TR-010, November 1996, http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr010.96.pdf

[Zave97] P. Zave, M. Jackson, *Four Dark Corners of Requirements Engineering*, ACM Transactions on Software Engineering and Methodology, Vol. 6. No. 1, Januar 1997

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen wörtlich oder sinngemäß übernommenen Gedanken sind als solche kenntlich gemacht.

Dresden, 30.07.2001