Fakultät für Elektrotechnik, Informatik und Mathematik

Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien

Der Fakultät für Elektrotechnik, Informatik und
Mathematik
der Universität Paderborn
zur
Erlangung des Grades eines
„Doktor der Naturwissenschaften“
(Dr. rer. nat.)
eingereichte

D i s s e r t a t i o n

vorgelegt von

Dipl.-Wirt.-Inf. Andreas Wübbeke

aus Paderborn.

Erster Gutachter: Prof. Dr. Gregor Engels
Zweiter Gutachter: Prof. Dr. Wilhelm Schäfer

Paderborn, August 2010

Danksagung

An erster Stelle möchte ich meinem Doktorvater Prof. Dr. Gregor Engels für die wissenschaftliche Betreuung meines Themas danken. Er hat mit seinen Kommentaren und Verbesserungsvorschlägen maßgeblich zum gelingen dieser Arbeit beigetragen und mir in unzähligen Diskussion Stärken und Schwächen meiner Argumentation aufgezeigt. Ich danke Prof. Dr. Wilhelm Schäfer für die Übernahme des zweiten Gutachtens zu dieser Arbeit und für seine hilfreichen Hinweise auf den s-lab Research Days.

Mein Dank geht auch an die Geschäftsführung des s-lab, Stefan Sauer und Dr. Matthias Meyer sowie den ehemaligen Geschäftsführer Dr. Matthias Gehrke. Ihr habt in vielen Diskussionen über mein Thema und nicht zuletzt durch die tolle Organisation und ein mehr als angenehmes Arbeitsklima zum gelingen dieser Arbeit beigetragen.
Ein weiterer besonderer Dank geht an Dr. Thomas von der Maßen für die tolle Unterstützung und guten Ideen für meine Arbeit und die Bereitschaft Mitglied meiner Promotionskommision zu werden. Thomas, vielen Dank für die wunderbare Zeit in Deinem Team mit seinen tollen Kolleginnen und Kollegen.

Weiterhin danke ich den (ehemaligen) Kolleginnen und Kollegen des s-labs und des Lehrstuhls für Datenbanken und Informationsysteme. Allen voran meinen Bürokollegen (in chronologischer Reihenfolge) Hendrik Voigt, Heinrich Balzert, Baris Güldali, Yavuz Sancer und Michael Spijkerman. Ihr habt mich durch zahlreiche Diskussionen innerhalb und abseits meines Themas immer ein entscheidendes Stück weiter gebracht. Ferner gilt mein Dank allen Studien-, Bachelor- und Diplomarbeitern für ihre Mitarbeit an meinem Thema.

Mein ganz besonderer Dank aber gilt Andrea. Du hast mich vor allem in der Phase des Aufschreibens immer wieder motiviert und es geschafft das ich mich auch das ein oder andere mal vom Schreibtisch entferne. Denn gute Ideen entstehen häufig abseits dieses Ortes. Vielen lieben Dank dafür. Weiterhin möchte ich meinen Eltern dafür Danken, dass sie mich Zeit meines Lebens unterstützt und gefördert haben. Ihr habt mir rückhaltloses Vertrauen entgegen gebracht und mir immer die Freiräume gelassen die notwendig waren.

Zusammenfassung

Software-Produktlinien (SPL) stellen ein Entwicklungsparadigma dar, welches die Entwicklungzeit von Softwaresystemen bei gleichzeitig gesteigerter Qualität verkürzen soll. Um dieses Ziel zu erreichen, werden diejenigen Artefakte, die vielen oder sogar allen entwickelten Softwaresystemen gemeinsam sind, nur einmal entwickelt und wiederverwendet. Die mit Hilfe des Software-Produktlinienparadigmas entwickelten Softwaresysteme werden Produkte genannt.
Die Wiederverwendung von Artefakten wird über den gesamten Software-Produktlinienentwicklungsprozess hinweg organisiert. Da einige Artefakte nur für die Entwicklung mancher Produkte genutzt werden, stellen diese die Variabilität innerhalb der SPL dar. Das Management dieser Variabilität über alle Phasen des Entwicklungsprozesses hinweg bedeutet eine große Herausforderung für die Entwicklung von Software-Produktlinien.
Zunächst muss in jedem Artefakt die Modellierung von Variabilität ermöglicht werden. Weiterhin können zwischen variablen Artefakten Abhängigkeiten entstehen. Die Modellierbarkeit von Variabilität und ihrer Abhängigkeiten erfordert auch die Neuentwicklung oder Anpassung von Spezifikationstechniken für diese Artefakte.
Diese Arbeit fokussiert das Variabilitätsmanagement in der Anforderungs- und Testfallspezifikation für Software-Produktlinien. Dabei werden existierende Modellierungssprachen und Spezifikationstechniken um Variabilität erweitert, anstatt diese neu zu entwickeln.
Für die Erweiterung von Modellierungssprachen wird ein Sprachkonstruktionsprozess basierend auf einem Metamodell des Variabilitätsmanagements definiert. Für das Management der Abhängigkeiten zwischen variablen Artefakten wird anschließend ein featurebasiertes Variabilitätsmanagement eingeführt. Schließlich wird die Erweiterung der Spezifikationstechniken für Anforderungen, welche in dieser Arbeit durch Anwendungsfallbeschreibungen spezifiziert werden, und Testfällen beschrieben.
Der Beitrag dieser Arbeit wird durch eine prototypische Werkzeugunterstützung und eine industrielle Fallstudie evaluiert.

Abstract

Software Product Lines (SPL) are a development paradigm allowing the reduction of the software system development time while increasing their quality. To reach these goals, the artefacts being part of many or all software systems are developed only once and reused within the different software systems. The software systems developed by a software product line approach are called products.
Like a thread the reuse of artefacts is woven into the software product line development process. As some artefacts are only used for the development of some products, these artefacts constitute the variability of the SPL.
First of all, the modelling of variability has to be enabled in every artefact. Beside this, dependencies can arise between variable artefacts. The modelling of variability and dependencies also demands the redevelopment or adaption of specification techniques for these artefacts.
This thesis focuses on the variability management concerning requirements and test case specification for Software Product Lines. For this purpose, existing modelling languages and specification techniques are augmented with variability instead of redeveloping similar software development artefacts over and over again.
In order to augment the modelling languages by variability, a language construction process based on a meta model of the variability management is defined. In order to manage the dependencies between variable artefacts, a feature based variability management is introduced. Finally, the extension of specification techniques for requirements specifications based on use case descriptions and test specifications are described.
The contribution of this thesis is substantiated by a prototypical tool support and an industrial case study.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Software durchdringt in der heutigen Zeit alle Bereiche unseres Lebens. Menschen nutzen sie in den unterschiedlichsten Bereichen, wie zum Beispiel Kommunikation, Produktion und Transport. Die kosteneffiziente und qualitativ hochwertige Entwicklung von Softwaresystemen für diese unterschiedlichen Domänen stellt eine Herausforderung für Softwareingenieure dar. Die Softwaretechnik ist eine Disziplin der Informatik, die sich mit der Bereitstellung und systematischen Verwendung von Konzepten, Sprachen, Prozessen und Werkzeugen für die ingenieurmäßige Entwicklung von Softwaresystemen beschäftigt [Bal01].

Im Rahmen dieser Disziplin sind verschiedene Ansätze entstanden, die die Erstellung und den Architekturentwurf von Softwaresystemen systematisch unterstützen. Beispiele für die Erstellung sind Vorgehensmodelle, wie das allgemeine V-Modell [DW00], der Rational Unified Process (RUP) [Kru99] oder agile Methoden wie SCRUM [Pic07]. Die Architektur von Softwaresystemen kann zum Beispiel komponentenorientiert [HC01] oder serviceorientiert [Erl06] gestaltet werden. Jeder dieser Ansätze bietet bestimmte Vor- und Nachteile, sodass die Auswahl des richtigen Ansatzes für ein bestimmtes Softwareentwicklungsvorhaben wichtig ist. Entwickelt man beispielsweise ein Kundenbindungssystem, wobei zu Beginn der Entwicklung die Anforderungen an das System nicht genau bekannt sind, könnte der RUP als Entwicklungsansatz die passende Wahl sein, da sein iteratives und inkrementelles Vorgehen während der Entwicklung die Präzisierung der Anforderungen unterstützt.

Wenn nun viele Auftraggeber Kundenbindungssysteme entwickeln lassen, die überwiegend gleiche Anforderungen berücksichtigen, werden diese in den bisher vorgestellten Vorgehensmodellen für jedes einzelne Kundenbindungssystem immer wieder neu umgesetzt. Eine komponentenorientierte Architektur könnte dazu beitragen die Wiederverwendung von Teilen der Implementierung zu ermöglichen. Die Wiederverwendung von anderen Entwicklungsartefakten, wie zum Beispiel Anforderungsspezifikationen und Testfällen, wird dabei aber nicht betrachtet. Im Sinne einer effizienten Softwareentwicklung entsteht an dieser Stelle der Bedarf für ein Entwicklungsvorgehen und eine Architektur, welche die Gemeinsamkeiten berücksichtigt und deren Wiederverwendbarkeit ermöglicht. Im Gegensatz zu den bisherigen Ansätzen muss das Entwicklungsvorgehen und die Architektur nicht nur die Entwicklung eines Kundenbindungssystems unterstützen, sondern aller Kundenbindungsysteme, die über gleiche Anforderungen verfügen. Durch die Berücksichtigung aller Kundenbindungssysteme können ihre gleichen Anteile wiederverwendet werden und müssen nur einmal spezifiziert bzw. entwickelt werden.

In der fertigenden Industrie ist dieses Entwicklungsvorgehen seit langer Zeit unter dem Begriff *Produktlinie* bekannt [CN98]. Beispielsweise fertigt die Automobilindustrie viele ähnliche Fahrzeuge auf Basis einer sogenannten *Plattform*, welche die gemeinsamen Bestandteile der Fahrzeuge bereitstellt. Das Entwicklungsvorgehen ist dabei auf die organisierte Wiederverwendung der gemeinsamen Bestandteile mit Hilfe dieser Plattform als zentralem Bestandteil der Architektur ausgerichtet. Das Prinzip der Produktlinie aus der fertigenden Industrie lässt sich auch auf die Domäne der Softwaretechnik übertragen und ist unter dem Begriff *Software-Produktlinie* (SPL) oder auch *Software-Familie* bekannt [Lin02]. Im Rahmen dieser Arbeit wird ein Konzept vorgestellt, welches das zentrale Ziel der organisierten Wiederverwendung bei Software-Produktlinien unterstützt und damit die Effektivität und Effizienz des Entwicklungsparadigmas steigert. Für das weitere Verständnis der Problemstellung und des Beitrags dieser Arbeit werden zunächst einige grundlegende Begriffe des Software-Produktlinienparadigmas erklärt.

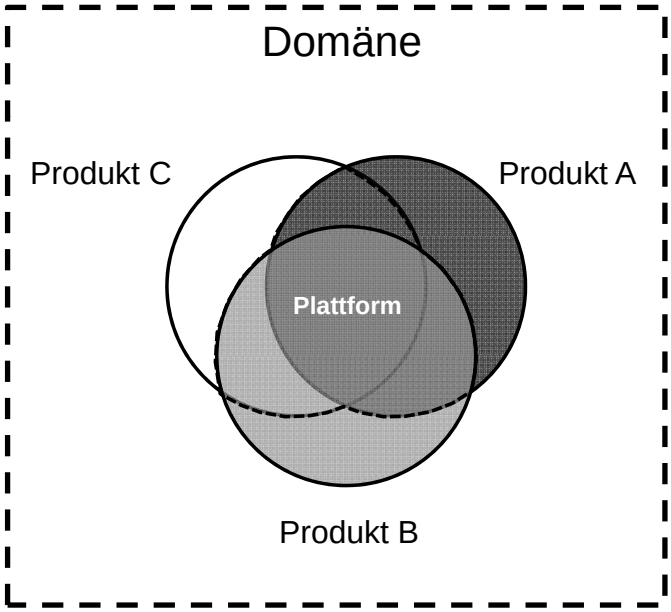


Abbildung 1.1: Software-Produktlinien-Architektur: Domäne, Plattform und Produkt [Maß07].

Software-Produktlinien benötigen einen Architekturstil, welcher auf die Wiederverwendung von Artefakten aus allen Phasen des Entwicklungsprozesses ausgerichtet ist. In Abbildung 1.1 ist der prinzipielle Aufbau einer Produktlinie verdeutlicht. Die sich überschneidenden Kreise stellen Produkte dar. Der Überschneidungsbereich beinhaltet die Funktionalitäten, die mehreren Produkten gemeinsam sind. Diese Funktionalitäten werden als Produktlinien-Plattform (kurz *Plattform*) bezeichnet. Dabei unterscheidet man *verpflichtende* (notwendige) und *variable* Plattform-Funktionalitäten. Erstere befinden sich in der Abbildung im Überschneidungsbereich aller Produkte und sind daher Teil jeden Produktes, letztere in den Überschneidungsbereichen von zwei Produkten und sind damit Teil mancher Produkte. Aus diesem Grund müssen letztere Funktionalitäten der Plattform auswählbar, d.h. variabel gestaltet werden. Die überschneidungsfreien Bereiche der Produkte stellen produktindividuelle Funktionalitäten dar.
Die Entwicklung einer Software-Produktlinie ist auf eine bestimmte Domäne ausgerichtet [PBL05]. Dies ermöglicht es, die Wiederverwendung effektiv zu gestalten, da die fachlichen Anforderungen an die Plattform eingeschränkt werden. Domänen können zum Beispiel Motorsteuergeräte- oder Mobiltelefon-Software sein.

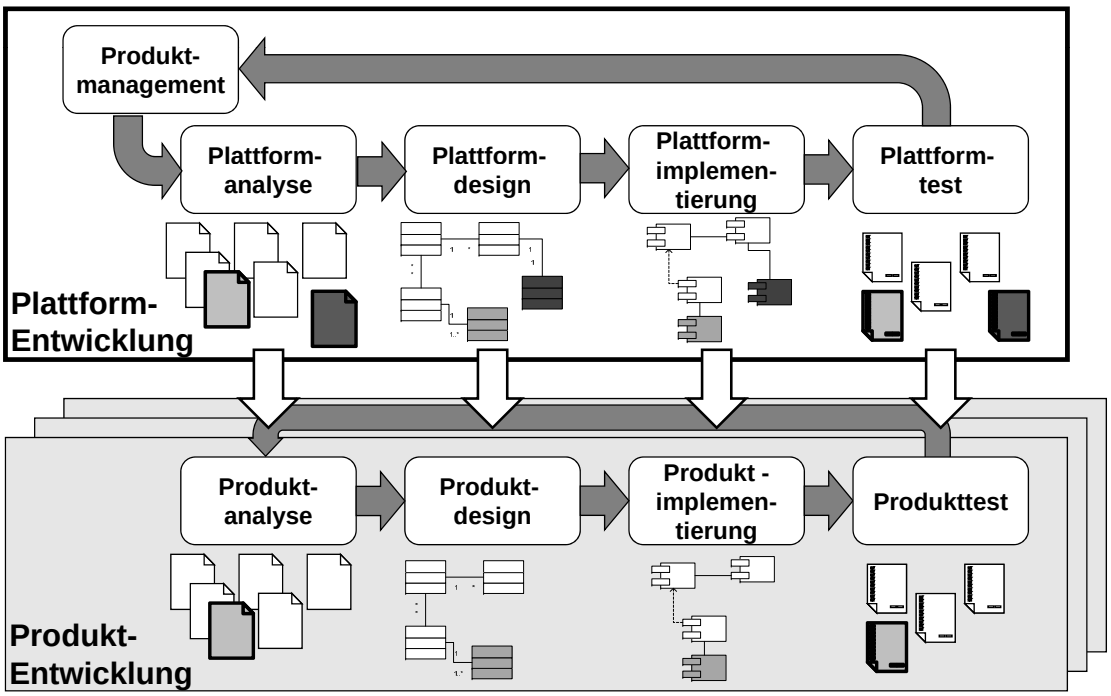


Abbildung 1.2: Software-Produktlinien Entwicklungsprozess [PBL05].

Die Produktlinien-Plattform bildet die architektonische Grundlage für die Wiederverwendung von Funktionalitäten. Um diese Wiederverwendung zu organisieren, wird ein Entwicklungsprozess benötigt, der die Plattform einbindet. Der aus der Literatur bekannte SPL-Entwicklungsprozess, wie in Abbildung 1.2 dargestellt, teilt sich in Plattform- und Produkt-Entwicklung auf [PBL05]. Erstere besteht dabei aus fünf Teilprozessen: Im Teilprozess *Produktmanagement* wird festgelegt welche Domäne adressiert wird und welche Funktionalitäten Teil der Plattform werden sollen. Die drei darauf folgenden Teilprozesse *Plattformanalyse*, *Plattformdesign* und *Plattformimplementierung* ermöglichen die Konstruktion und schrittweise Verfeinerung der Artefakte, die die Plattformfunktionalitäten beschreiben bzw. umsetzen. Dabei wird auch bestimmt, welche Funktionalitäten variabel gestaltet werden (in der Abbildung hervorgehobene Artefakte). Der letzte Teilprozess ist der *Plattformtest*. Dieser Prozess beinhaltet sowohl die Spezifikation von wiederverwendbaren Testfällen unter Berücksichtigung von Variabilität, als auch den Test der Plattform-Funktionalitäten. Aus der Plattform werden Produkte abgeleitet. Ableitung bedeutet in diesem Zusammenhang die Bindung von Variabilität und damit die Entscheidung für oder

gegen bestimmte Funktionalitäten aus der Plattform. Die Ableitung wird in Abbildung 1.2 mittels vertikaler Pfeile dargestellt.
Die aus der Plattform abgeleiteten Funktionalitäten bilden die Grundlage für die Produkt-Entwicklung. Diese besteht wiederum aus vier Teilprozessen: Die Teilprozesse *Produktanalyse*, *Produktdesign* und *Produktimplementierung* korrespondieren zu den drei konstruktiven Teilprozessen der Plattform, und der Teilprozess *Produkttest* korrespondiert zum *Plattformtest*. In diesen Teilprozessen werden die kundenindividuellen Anforderungen spezifiziert, implementiert und getestet.

Die Entwicklung von Produkten auf Basis des Software-Produktlinienparadigmas verfolgt Ziele, die in [PBL05] in folgender Weise beschrieben sind:

1. Reduzierung der Entwicklungskosten
2. Verbesserung der Qualität
3. Reduzierung der Zeit bis zur Markteinführung eines Produktes
4. Reduzierung der Wartungskosten
5. Bewältigung der Produkt-Evolution
6. Beherrschung der Komplexität
7. Verbesserung der Kostenschätzung
8. Entwicklung von kundenindividuellen Produkten

Um diese Ziele zu erreichen, entstehen bei der Konzeption und Umsetzung der Architektur sowie im Entwicklungsprozess für Software-Produktlinien verschiedene Problemstellungen, von denen einige in dieser Arbeit behandelt werden. Diese Problemstellungen werden im nun folgenden Abschnitt definiert.

1.2 Problemstellung

Um die Problemstellung dieser Arbeit zu beschreiben wird ein Beispiel in Analogie zur SPL-Darstellung aus Abbildung 1.1 skizziert. Die Produktlinie *Loyaltymanagement* erlaubt es Unternehmen *Punktesammelkarten* an ihre Kunden auszugeben,

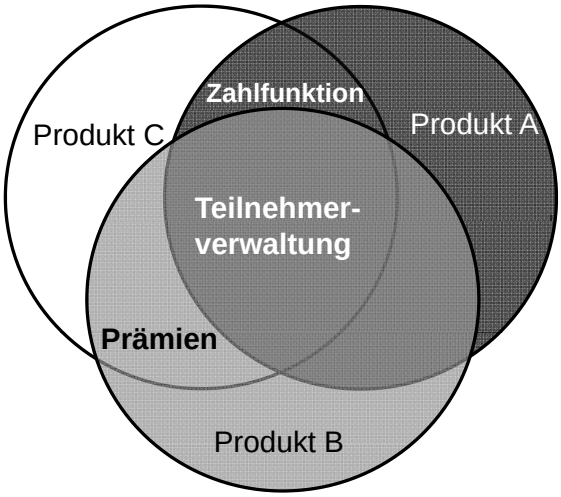


Abbildung 1.3: Beispiel Software-Produktlinie Loyaltymanagement

um einerseits dem Kunden durch Vergünstigungen Anreize zu schaffen und andererseits Informationen über das Kaufverhalten des Kunden zu gewinnen, und somit das eigene Sortiment zu optimieren. Abbildung 1.3 stellt einen Ausschnitt dieser SPL dar. Zum einen ist allen Produkten eine *Teilnehmerverwaltung* gemeinsam. In manchen Produkten werden *Prämien* für die Kunden angeboten, in anderen besitzt die Punktesammelkarte eine *Zahlfunktion*. Diese Funktionalitäten sind *variabel*, da sie nicht Teil jedes Loyaltymanagements sind.

Betrachtet man nun die Entwicklung dieser gemeinsamen und variablen Funktionalitäten im Entwicklungsprozess, entsteht die in Abbildung 1.4 dargestellte Situation. Die in dieser Arbeit adressierten Bereiche des SPL-Entwicklungsprozesses sind dabei hervorgehoben.
Diese Arbeit fokussiert auf die Teilprozesse Plattformanalyse und Plattformtest sowie die Ableitung dieser Artefakte für Produkte der Produktlinie [Wüb08, WO10]. Zunächst müssen in der Plattformanalyse die Anforderungen an die Software-Produktlinienplattform in Form von Anforderungsartefakten spezifiziert werden. Dabei ist es notwendig, diejenigen Anforderungen als variabel spezifizieren zu können, die nicht Teil jeden Produktes sind. Eine explizite Modellierung der Variabilität ermöglicht später die Auswahl von Anforderungen bei der Ableitung von Produkten. Zusammengefasst entsteht folgende Problemstellung, die in dieser Arbeit adressiert wird:

Problemstellung 1. *Definition von Modellierungssprachen für Anforderungsartefakte mit Variabilität für Software-Produktlinien.*

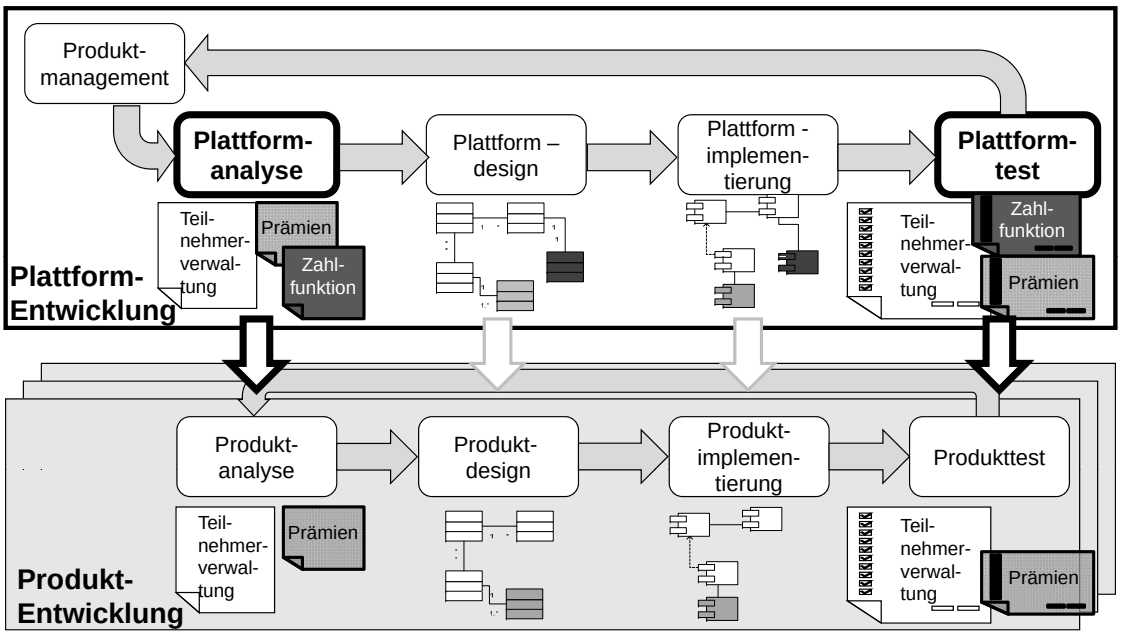


Abbildung 1.4: Übersicht ausgewählter Problemstellungen im Software-Produktlinienentwicklungsprozess

Im Rahmen des Teilprozesses Plattformtest werden Testfälle spezifiziert, die für die Überprüfung der korrekten Umsetzung der Anforderungen eingesetzt werden können. Anforderungsartefakte können also Grundlage für die Spezifikation von Testfällen sein. Da die Anforderungsartefakte der SPL Variabilität enthalten, ist die Modellierung von Variabilität auch in den Testfällen notwendig. Damit wird die mehrfache Spezifikation von Testfällen oder ihrer Bestandteile vermieden. Folgende weitere Problemstellung resultiert daraus für diese Arbeit:

Problemstellung 2. *Definition von Modellierungssprachen für Testfälle mit Variabilität für Software-Produktlinien.*

Die explizite Modellierung von Variabilität in Anforderungsartefakten ermöglicht die Auswahl von Anforderungen für ein bestimmtes Produkt. Um die korrekte Umsetzung dieser Anforderungen im Produkt überprüfen zu können, müssen nun diejenigen Testfälle ausgewählt werden, die zu den gewählten Anforderungsartefakten passen. Passende Testfälle sind solche, die genau die ausgewählten Anforderungen testen. Durch diese Forderung entsteht folgende Problemstellung:

Problemstellung 3. *Abbildung der ausgewählten Anforderungsartefakte auf dazu passende Testfälle.*

Für die Spezifikation von Anforderungsartefakten und Testfällen für Einzelsysteme existieren bereits unterschiedliche Methoden und Techniken [Poh08, Rup09, Bin99, SL05, UL07]. Da im Kontext dieser Arbeit die Spezifikation von Anforderungsartefakten und Testfällen mit Variabilität notwendig ist, wird auch folgende Problemstellung betrachtet:

Problemstellung 4. *Definition neuer oder Erweiterung existierender Anforderungs- und Testfallspezifikationstechniken für die Behandlung von Variabilität.*

Die Existenz dieser vier Problemstellungen wird durch das Software-Produktlinienparadigma bedingt. Ziel des Paradigmas ist, die in Abschnitt 1.1 beschriebenen Vorteile durch organisierte Wiederverwendung zu erreichen. Die hier definierten Problemstellungen unterstützen unterschiedliche Ziele des Software-Produktlinienparadigmas, wie in Abbildung 1.5 dargestellt ist.

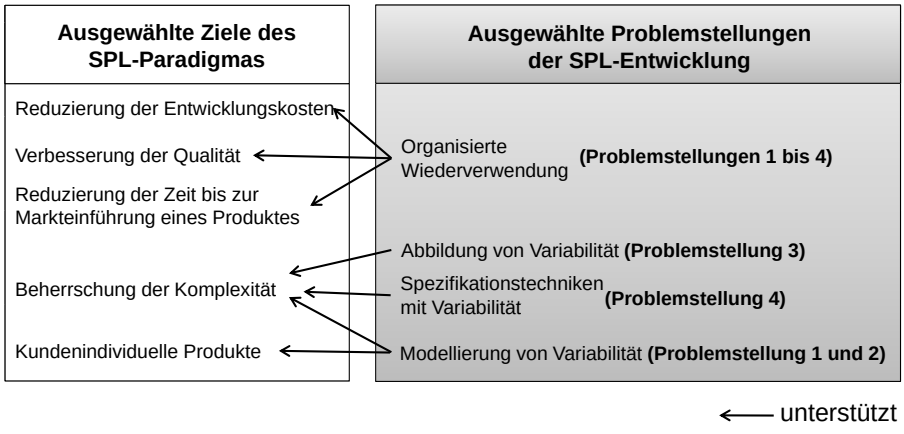


Abbildung 1.5: Ausgewählte Ziele des Software-Produktlinienparadigmas und ihre Unterstützung durch Problemstellungen dieser Arbeit

Alle vier hier vorgestellten Problemstellungen sind Teilprobleme der zentralen Problemstellung „organisierte Wiederverwendung“, welche die Ziele „Reduzierung der Entwicklungskosten“, „Verbesserung der Qualität“ und „Reduzierung der Zeit bis zur Markteinführung“ unterstützt. Die Modellierung von Variabilität sowie Spezifikationstechniken mit Variabilität unterstützen die Beherrschung der Komplexität

bei der Entwicklung von Software-Produktlinien. Dieses Ziel wird auch durch die Abbildung zwischen der Variabilität in Anforderungsartefakten und Testfällen unterstützt. Die explizite Modellierung von Variabilität unterstützt weiterhin das Ziel kundenindividuelle Produkte anbieten zu können.

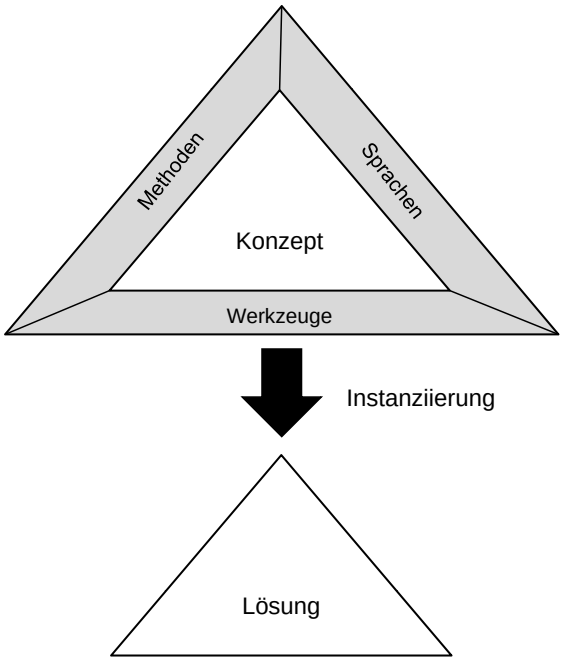


Abbildung 1.6: Zusammenhang zwischen Konzept und Lösung

Wie jedes andere Entwicklungsvorgehen auch, benötigt die Entwicklung von Software-Produktlinien Konzepte, unterstützt durch Methoden, Sprachen und Werkzeuge, mit deren Hilfe eine Lösung für die hier vorgestellten Problemstellungen instanziiert werden kann (vgl. Abbildung 1.6). Im nun folgenden Abschnitt wird der Beitrag dieser Arbeit beschrieben.

1.3 Beitrag der Arbeit

Für die Beschreibung des Beitrags dieser Arbeit wird das Paradigma aus Abbildung 1.6 instanziiert. Die Bausteine des Beitrags sind in Abbildung 1.7 um das jeweilige Konzept-Dreieck angeordnet.
Um die Modellierung und das Management von Variabilität zu ermöglichen, wird als Konzept die Metamodellierung genutzt (vgl. Abbildung 1.7 oben). Als Sprachen werden dabei die UML und OCL eingesetzt. Die Metamodellierung mit Hilfe dieser

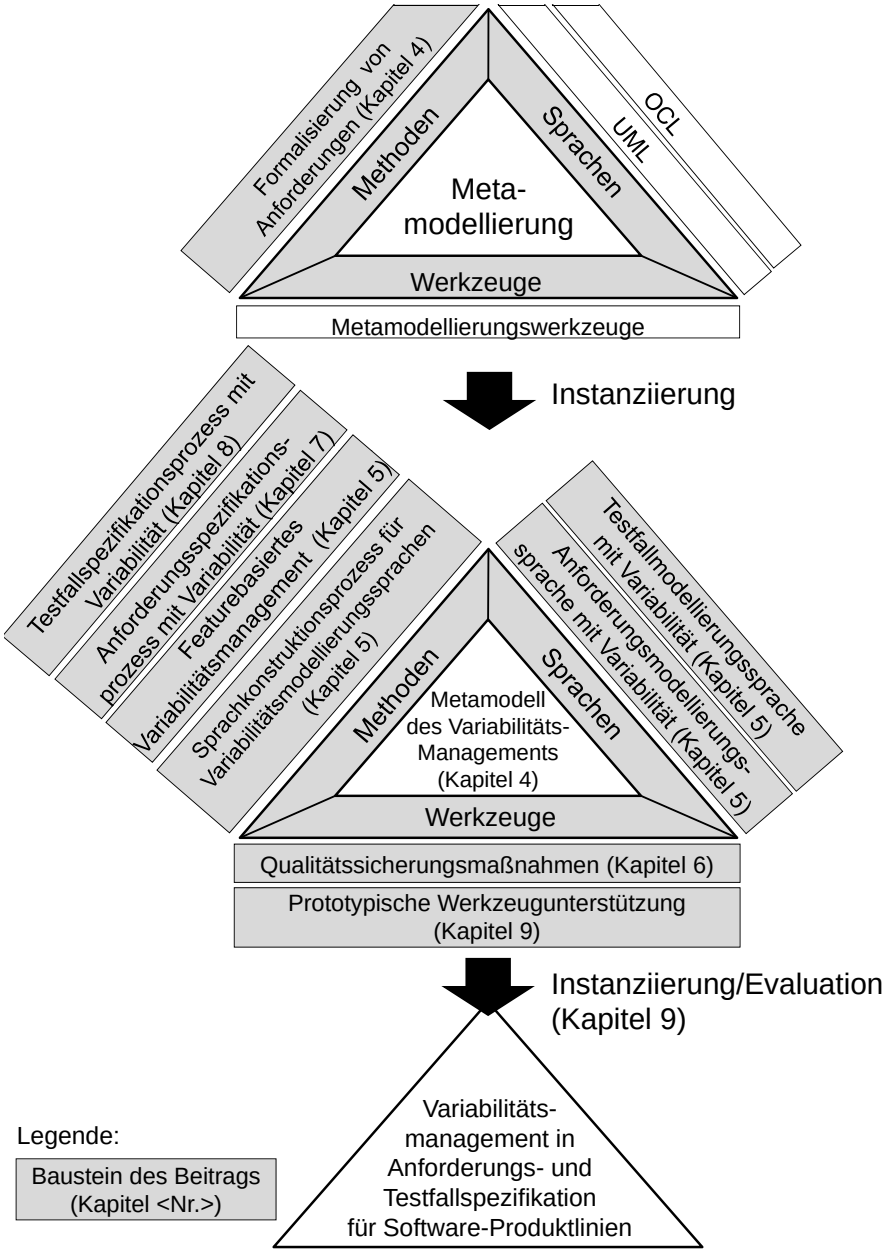


Abbildung 1.7: Beitrag der Arbeit basierend auf dem Konzept der Metamodellierung

beiden Sprachen ermöglicht die Formalisierung von Anforderungen an die Modellierung und das Management von Variabilität. Das Ergebnis dieser Formalisierung ist das Metamodell des Variabilitätsmanagement.

Das Konzept der Metamodellierung kann weiterhin durch geeignete Metamodellierungswerkzeuge unterstützt werden [Hau05, Metb, Gen, Meta]. Diese Unterstützung steht nicht im Fokus dieser Arbeit und sei deshalb nur am Rande erwähnt.

Das Metamodell des Variabilitätsmanagements stellt das Konzept für alle weiteren Bausteine dieser Arbeit dar und wird mit Hilfe der Metamodellierung instanziiert (vgl. Abbildung 1.7 Mitte).
Um Variabilität in Modellierungssprachen integrieren zu können, wird im Rahmen dieser Arbeit ein Sprachkonstruktionsprozess für sogenannte Variabilitätsmodellierungssprachen definiert. Mit Hilfe des Sprachkonstruktionsprozesses werden eine Anforderungs- und eine Testfallmodellierungssprache mit Variabilität definiert. Dieser Baustein adressiert die Problemstellungen 1 und 2 aus Abschnitt 1.2.
Um die in Modellen vorhandene Variabilität verwalten zu können wird ein featurebasiertes Variabilitätsmanagement definiert. Dieses ermöglicht unter anderem die Abbildung von Variabilität zwischen unterschiedlichen Artefakten, was in Problemstellung 3 gefordert wurde.
Mit Hilfe der Anforderungs- sowie Testfallmodellierungssprache und dem featurebasierten Variabilitätsmanagement werden ein Anforderungs- sowie ein Testfallspezifikationsprozess mit Variabilität definiert. Dabei werden existierende Spezifikationstechniken um Variabilität erweitert, wie in Problemstellung 4 gefordert.

Die Modellierung und das Management von Variabilität führt zu neuen Arten von Fehlern, die durch den Nutzer der Methoden und der Sprachen gemacht werden können. Aus diesem Grund wird ein Baustein mit Qualitätssicherungsmaßnahmen für Variabilität als Werkzeug beschrieben.
Zuletzt wird in dieser Arbeit eine prototypische Werkzeugunterstützung für Teile des Beitrags angeboten.

Die Instanziierung der beschriebenen Bausteine des Beitrags bildet die Lösung für das Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien (vgl. Abbildung 1.7 unten). Eine solche Instanziierung wird für die Evaluation des Beitrags im Rahmen einer Fallstudie durchgeführt.

1.4 Aufbau der Arbeit

Diese Arbeit ist in folgender Art und Weise strukturiert. Die Kapitel „Einleitung“ und „Zusammenfassung und Ausblick“ bilden den Rahmen dieser Arbeit. Die notwendigen Grundlagen sowie eine detaillierte Problemstellung und die Betrachtung verwandter Arbeiten werden in Kapitel 2 und 3 behandelt. Der Beitrag dieser Arbeit wird in den Kapiteln 4 bis 9 beschrieben:
Zunächst wird in Kapitel 4 das Variabilitätsmanagement diskutiert. Im darauf folgenden Kapitel 5 wird der Sprachkonstruktionsprozess für die Definition von Variabilitätsmodellierungssprachen beschrieben und das featurebasierte Variabilitätsmanagement entwickelt. In Kapitel 6 werden Maßnahmen für die Qualitätssicherung für die modellierte Variabilität beschrieben.
Die Kapitel 7 und 8 definieren Spezifikationsprozesse für Anforderungen und Testfälle mit Variabilität. Kapitel 9 stellt eine prototypische Werkzeugunterstützung sowie eine Evaluation des Ansatzes auf Basis einer Fallstudie vor. Die Arbeit schließt in Kapitel 10 mit einer Zusammenfassung und dem Ausblick auf offene und neue Problemstellungen.

Kapitel 2

Grundlagen

In diesem Kapitel werden grundlegende Konzepte, Methoden und Sprachen eingeführt, die für den Kontext des Beitrags dieser Arbeit relevant sind. Zunächst wird das Entwicklungsparadigma Software-Produktlinien beschrieben und wichtige Begrifflichkeiten in diesem Kontext definiert. Im Anschluss daran wird die Anforderungs- und Testfallspezifikation für die Entwicklung von Einzelsystemen betrachtet. Diese beiden Prozesse stellen den Ausgangspunkt für die spätere Entwicklung von Anforderungs- und Testfallspezifikationsprozess mit Variabilität für Software-Produktlinien dar.

2.1 Software-Produktlinien

Software-Produktlinien (SPL) beschreiben einen Ansatz für die organisierte Wiederverwendung bei der Entwicklung ähnlicher Produkte auf Basis einer gemeinsamen Plattform. Die Wiederverwendung betrifft hierbei nicht nur die Implementierung selbst, wie es durch bereits etablierte Techniken, wie zum Beispiel der *Objekt-Orientierung* der Fall ist, sondern alle Artefakte, die im Entwicklungsprozess entstehen. Diese Wiederverwendung ermöglicht die schnelle und kosteneffiziente Entwicklung der Produkte für eine bestimmte Domäne.
Das Entwicklungsparadigma wurde initial in der Zeit um den letzten Jahrtausendwechsel im wissenschaftlichen Umfeld bekannt. Die Idee war von Beginn an aus der Industrie motiviert. Erste Beiträge wie [Bos00] und [CN01] sind Belege dafür.

2.1.1 Das Entwicklungsparadigma

Die hier ausgeführten Grundlagen werden in ähnlicher Form auch in der oft zitierten Basisliteratur [CN01] und [PBL05] beschrieben. Der Begriff Software-Produktlinie wird daher in dieser Arbeit in folgender Art und Weise verstanden:

Definition 1. *Software-Produktlinie (software product line): A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [CN01].*

Eine Software-Produktlinie ist demnach eine Menge von software-intensiven Systemen (*Produkten*), die basierend auf einer Menge von gemeinsamen Funktionalitäten (*Features*) in einer vorgeschriebenen Art und Weise entwickelt wird. Diese gemeinsamen Funktionalitäten werden in Form einer *Plattform* organisiert und adressieren ein bestimmtes Marktsegment (*Domäne*).

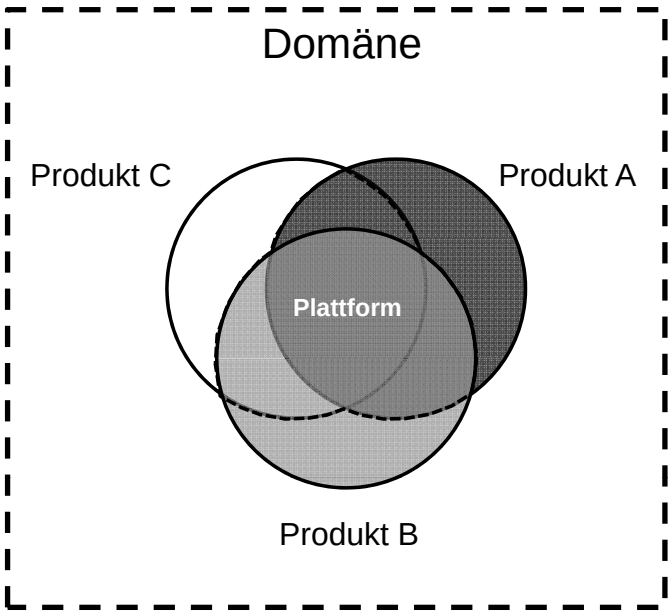


Abbildung 2.1: Software-Produktlinienarchitektur: Domäne, Plattform und Produkt [Maß07].

Der Zusammenhang der grundlegenden Begriffe Domäne, Plattform und Produkt wurde bereits in Kapitel 1 vorgestellt und ist in Abbildung 2.1 noch einmal dargestellt.

Zunächst ist für die Entwicklung einer SPL wichtig, diese auf eine bestimmte Domäne auszurichten. Mit dieser Ausrichtung wird eine Fokussierung auf ein Marktsegment oder Ziel erreicht. In dieser Arbeit wird der Begriff Domäne in folgender Weise definiert:

Definition 2. *Domäne: Eine Domäne grenzt ein bestimmtes Marktsegment oder Ziel aus Sicht der funktionalen und nicht-funktionalen Anforderungen an Produkte ein.*

Die Eingrenzung auf ein Marktsegment oder ein bestimmtes Ziel gewährleistet also die Eingrenzbarkeit auf bestimmte, für das Marktsegment oder Ziel erforderlichen, Funktionalitäten zur Erfüllung der existierenden Anforderungen. Diese können wiederum in funktionale und nicht-funktionale Anforderungen unterteilt werden [ISO, Poh08]. Eine Domäne kann zum Beispiel Software für ein Motorsteuergerät für Kraftfahrzeuge oder ein Kundenbindungssystem für den Einzelhandel sein.
In Abbildung 2.1 ist die Domäne als gestricheltes Quadrat dargestellt. Das Quadrat stellt dabei alle Anforderungen der Domäne dar. Die gestrichelte Linie deutet die Veränderlichkeit der Domäne in Bezug auf die Anforderungen über die Zeit an. Somit ist auch eine Produktlinie immerwährenden Anpassungen an die Bedürfnisse der jeweiligen Domäne unterworfen (*Software-Produktlinienevolution*). Ein Beispiel für eine sich schnell verändernde Domäne ist die Entwicklung des Marktes für Mobiltelefone in den vergangenen Jahren. Das Thema Produktlinienevolution spielt in dieser Arbeit eine untergeordnete Rolle. Es ist aber Gegenstand unterschiedlicher Forschungsansätze [Pus02, Bos00, Bos02].
In Abbildung 2.1 wird weiterhin die Plattform der Software-Produktlinie als die Überschneidung von Produkten dargestellt. Die Überschneidungsbereiche der drei beispielhaft als Kreise dargestellten Produkte skizzieren die Funktionalitäten, die in mehr als nur einem Produkt Verwendung finden. Daher sind diese Funktionalitäten Teil der Produktlinienplattform. Meyer versteht eine Plattform folgendermaßen:

Definition 3. *Software Plattform: A software platform is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced [ML97].*

Demnach gehören zu einer Software-Produktlinienplattform eine Menge von Teilsystemen und deren Schnittstellen, aus der Produkte in einer effizienten Art und Weise entwickelt und produziert werden. Der Begriff *Effizienz* ist im SPL Kontext

von Bedeutung, da eine Software-Produktlinie nur durch den effizienten Einsatz der Plattform ihre Vorteile gegenüber der Einzelprodukt-Entwicklung ausspielen kann. Ein weiterer wichtiger Begriff im Rahmen der SPL-Entwicklung ist das Produkt selbst. In Abbildung 2.1 ist das Produkt, wie bereits geschildert, als Kreis skizziert. Der Kreis begrenzt die vom Produkt innerhalb der Domäne geforderten Funktionalitäten. Der Begriff Produkt wird demnach in dieser Arbeit folgendermaßen definiert:

Definition 4. *Produkt: Ein Produkt ist ein durch einen Auftraggeber definiertes und durch die Komposition von Plattform- und produktindividuellen- Funktionalitäten erstelltes software-intensives System.*

Die für ein Produkt erforderlichen Funktionalitäten werden also durch den Auftraggeber des jeweiligen Produktes definiert. Diese Funktionalitäten werden dann zum einen aus der Plattform wiederverwendet und zum anderen durch Neuentwicklung zum Produkt hinzugefügt. Dieser Umstand wird auch in Abbildung 2.1 deutlich. Der Teil der Plattform, indem sich alle drei Produkte überschneiden, verdeutlicht den Bereich der allen Produkten der Domäne gemeinsamen Funktionalitäten. In den Bereichen wo sich nicht alle Produkte überschneiden, befinden sich Funktionalitäten in der Plattform, die nicht in allen Produkten Bestandteil sind. Diese Funktionalitäten sind *variabel* in Bezug auf Ihre Einsatzmöglichkeit in Produkten der Produktlinie. Der Begriff Variabilität wird durch [Maß07] in folgender Art und Weise verstanden:

Definition 5. *Variabilität: Produktlinienvariabilität ist die Veränderlichkeit der Produktlinie in Bezug auf die unterschiedlichen Produkte, die auf Basis der Plattform erstellt werden können. Die Produkte unterscheiden sich in Anzahl und Ausprägung ihrer Charakteristika.*

Im nun folgenden Abschnitt wird auf den Software-Produktlinienentwicklungsprozess eingegangen. Dieser Entwicklungsprozess bildet die Grundlage für die Entwicklung von Produkten auf Basis der Produktlinienplattform.

2.1.2 Entwicklungsprozess

Der Entwicklungsprozess von Software-Produktlinien unterscheidet sich signifikant von Entwicklungsprozessen für Einzelproduktentwicklung. Der Hauptunterschied besteht in der Teilung des Entwicklungsprozesses in Plattform- und Produktentwicklung. Der Entwicklungsprozess ist in Abbildung 2.2 dargestellt. Der Fokus der

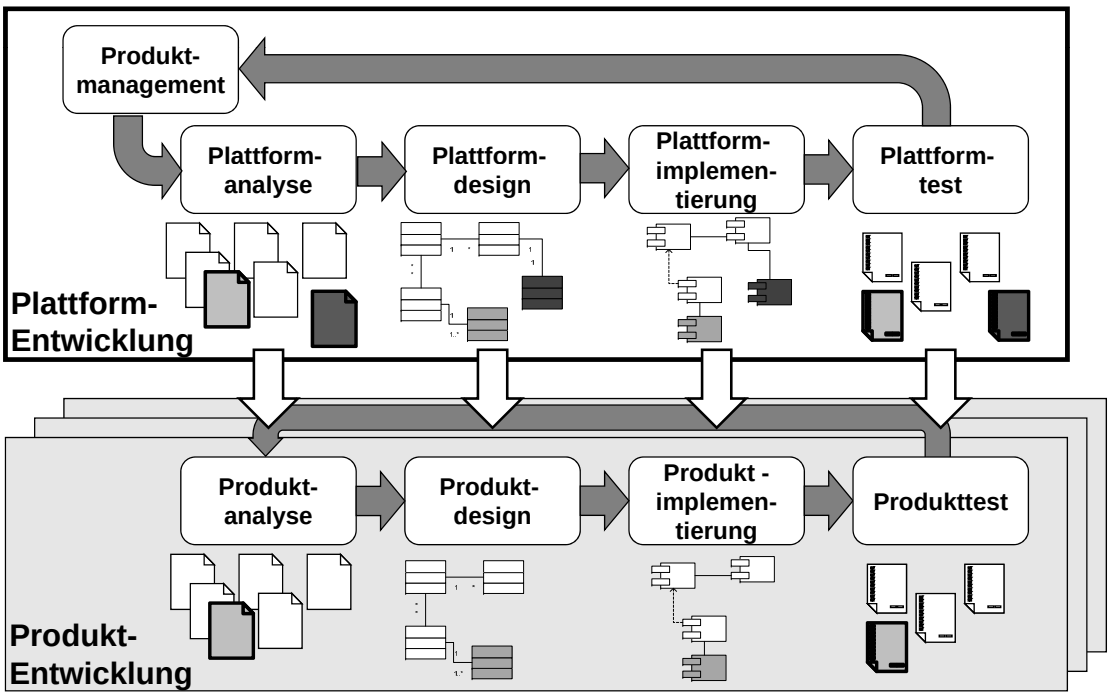


Abbildung 2.2: Software-Produktlinienentwicklungsprozess [PBL05]

Abbildung liegt auf der Präsentation der Teilung in die zwei Prozesse Plattform- und Produktentwicklung. Die Ausgestaltung der einzelnen Teilprozesse wird in [PBL05] nicht im Detail vorgenommen. Im Folgenden werden die einzelnen Teilprozesse, beginnend mit der Plattformentwicklung, näher beschrieben.

Die Plattformentwicklung besteht aus fünf Subprozessen, die als Kreislauf durchlaufen werden (graue Pfeile). Damit wird gewährleistet, dass die Ergebnisse der Phase Plattformtest zurück in die Entwicklung fließen. In der Darstellung ist der Entwicklungsprozess als lineare Abfolge von Schritten dargestellt.

Produktmanagement

Das Produktmanagement beschäftigt sich mit der Definition der von der Produktlinie adressierten Domäne. Hier wird weiterhin festgelegt, welche Funktionalitäten Teil der Produktlinienplattform sein sollen und welche nicht. Dabei muss auch definiert werden, welche Funktionalitäten in allen und welche nur in bestimmten Produkten vorhanden sein sollen. Letzteres wird aus Sicht der Plattform als Variabilität

verstanden. Für die Aufgabe der *Plattformgrößenbestimmung* ist sogenanntes Domänenwissen [PBL05] notwendig. Ein weiterer Bestandteil dieses Teilprozesses ist das Management der Produktlinienevolution. Dazu tragen zum einen die Ergebnisse der anderen Entwicklungsphasen bei, die durch den Kreislauf des Entwicklungsprozesses in die jeweils nächste Iteration einfließen. Zum anderen werden hier die sich über die Zeit verändernden Anforderungen der Domäne berücksichtigt und in den Plattformentwicklungsprozess eingebracht.

Plattformanalyse

In der Plattformanalyse werden die Anforderungen an die einzelnen Funktionalitäten der Plattform beschrieben. Dabei wird definiert, welche Anforderungen als variabel definiert werden sollen. Die Ausgabe dieses Teilprozesses sind Anforderungsartefakte, die in Abbildung 2.2 unterhalb des Teilprozesses dargestellt sind. Die grauen Artefakte symbolisieren hier Variabilität.

Plattformdesign

Im Teilprozess Plattformdesign wird die Feinspezifikation der Plattformfunktionalitäten, wie zum Beispiel die Architektur der Produktlinienplattform, auf Basis der zuvor spezifizierten Anforderungen erstellt. Hierbei dienen also die Anforderungsartefakte aus der vorhergehenden Phase als Eingabe. Der Teilprozess hat als Ausgabe die Plattformdesignspezifikation, die in Abbildung 2.2 beispielhaft durch ein Klassendiagramm dargestellt ist. Auch hier symbolisieren die grauen Anteile Variabilität. Diese wird im Teilprozess Plattformdesign auf Basis der Variabilität aus den Anforderungen spezifiziert.

Plattformimplementierung

Im Rahmen der Plattformimplementierung werden die zuvor spezifizierten Anforderungen und die Feinspezifikation in Form einer Implementierung umgesetzt. Ausgabe dieses Prozesses ist unter anderem der Quellcode, der in Abbildung 1.2 durch ein Komponentendiagramm visualisiert wird. In der Implementierung wird auch wieder die Variabilität der Plattform in geeigneter Weise berücksichtigt (grau hervorgehoben).

Plattformtest

Der letzte Teilprozess der Plattformentwicklung ist der Plattformtest. In dieser Phase werden zum einen Testfälle spezifiziert, die im späteren Produkttest in den unterschiedlichen Testphasen eingesetzt werden können. Zum anderen werden Testfälle spezifiziert und durchgeführt, um die Funktionalitäten der Plattform zu testen. Die Ergebnisse des Plattformtests und der übrigen Teilprozesse dienen wiederum als Eingabe für den Teilprozess Produktmanagement.

Produktentwicklung

Innerhalb der Produktentwicklung existieren vier Teilprozesse: Produktanalyse, Produktdesign, Produktimplementierung und Produkttest. Diese vier Teilprozesse korrespondieren mit den jeweiligen Prozessen aus der Plattformentwicklung. Ziel der Produktentwicklung ist es, auf Basis der Plattformfunktionalitäten und der darüber hinaus noch existierenden produktindividuellen Anforderungen ein Produkt zu entwickeln.
Hierzu werden zunächst für jede Phase der Produktentwicklung durch Ableitung aus der Plattform diejenigen Artefakte ausgewählt, die für die vom Produkt geforderten Funktionalitäten notwendig sind. Dies wird in Abbildung 2.2 durch die weißen Pfeile mit grauer Umrandung verdeutlicht. Eine Ableitung bedeutet also die Bindung von variablen Bestandteilen der Plattform durch Aus- oder Abwahl dieser. Eine formale Definition des Begriffs Ableitung wird in dieser Arbeit zu einem späteren Zeitpunkt gegeben.
Das Ergebnis der Ableitung sind produktspezifische Artefakte, die keine Variabilität mehr beinhalten. Im Rahmen der Produktentwicklung werden nun die produktindividuellen Funktionalitäten mit den produktspezifischen aus der Plattform zusammengeführt, um ein lauffähiges Produkt zu erstellen. Im letzten Schritt der Produktentwicklung, dem Produkttest, wird dieses Produkt mit Hilfe der um produktindividuelle Bestandteile erweiterten, produktspezifischen Testfälle getestet.

2.1.3 Rollenmodell

Für die Definition der Phasen eines Entwicklungsprozesses sind auch Rollen notwendig. Eine Rolle beschreibt die Verantwortlichkeit für eine oder mehrere Tätigkeiten,

losgelöst von einer konkreten Person. Sie kann durch mehrere Personen eingenommen werden und eine Person kann wiederum mehrere Rollen in einem Entwicklungsprozess einnehmen. Rollen werden bei der Beschreibung unterschiedlicher Entwicklungsprozesse definiert [Rup09].
Abbildung 2.3 skizziert die Rollen für den Software-Produktlinienentwicklungsprozess als UML-Anwendungsfalldiagramm [OMG09]. Die einzelnen Phasen des Entwicklungsprozesses sowie die Ableitung sind dabei als Anwendungsfall spezifiziert. Diese sind mit Akteuren verbunden, welche die Rollen darstellen. Die im Kontext dieser Arbeit relevanten Rollen und Phasen des Entwicklungsprozesses sind grau hinterlegt. Die Rollen werden nun näher erläutert.

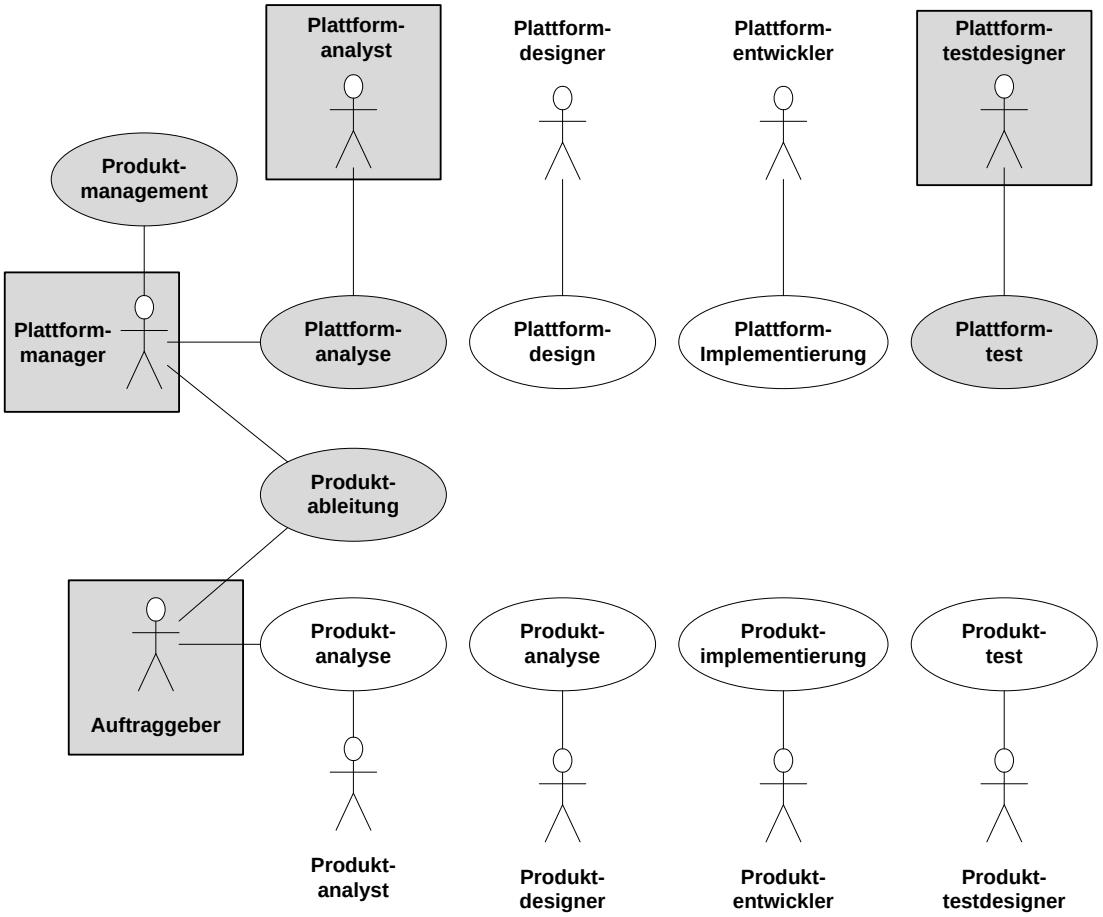


Abbildung 2.3: Rollen im Software-Produktlinienentwicklungsprozess dargestellt als UML-Anwendungsfalldiagramm

Auftraggeber

Der Auftraggeber gibt ein Produkt in Auftrag und definiert die Anforderungen an dieses. Diese Anforderungen werden bei der Entwicklung eines Produktes in zwei Mengen aufgeteilt:

- Anforderungen die durch die Plattform erfüllt werden und
- Anforderungen die im Rahmen der Produktentwicklung erfüllt werden.

Erstere Anforderungen können durch die Auswahl von Plattformartefakten erfüllt werden. Daher ist der Auftraggeber auch am Teilprozess Produktableitung beteiligt, um die für ihn passenden Plattformartefakte auszuwählen. Dabei wird er vom Plattformmanager unterstützt. Letztere Anforderungen sind für das in Auftrag gegebene Produkt individuell und werden im Rahmen des Produktentwicklungsprozesses erfüllt.

Plattformmanager

Der Plattformmanager analysiert die Domäne, die durch die SPL adressiert wird. Dies geschieht im Teilprozess Produktmanagement. Im Rahmen der Analyse wird festgelegt, welche Funktionalitäten Teil der Plattform werden sollen und ob diese variabel sein sollen (*Plattformabgrenzung*, engl. *platform scoping*). Der Plattformmanager unterstützt weiterhin den Auftraggeber bei der Auswahl der Plattformartefakte zur Erfüllung von dessen Anforderungen. Daneben unterstützt er auch den Plattformanalysten bei der Spezifikation der Anforderungsartefakte für die Produktlinienplattform.

Plattformanalyst

Der Plattformanalyst spezifiziert die Anforderungsartefakte der Produktlinienplattform, unterstützt vom Plattformmanager. Dabei nutzt er als Grundlage die Ergebnisse der Plattformabgrenzung.

Plattformtestdesigner

Der Plattformtestdesigner spezifiziert Testfälle basierend auf den Anforderungsartefakten. Er nutzt dabei auch die Artefakte aus den Teilprozessen Plattformdesign und Plattformimplementierung.

Neben den in Abbildung 2.3 dargestellten Rollen existieren weitere, wie zum Beispiel Plattformtestmanager und Plattformtester, welche für diese Arbeit nicht von Relevanz sind.

2.1.4 Variabilitätsmanagement

Variabilität, wie sie in Definition 5 definiert wurde, ermöglicht die Veränderlichkeit der Produktlinie zur Erfüllung der Anforderungen unterschiedlicher Auftraggeber. Ein solcher Auftraggeber kann variable Funktionalitäten aus der Plattform aus- oder abwählen. Der Umgang mit Variabilität in einer SPL wird als *Variabilitätsmanagement* bezeichnet und ist in dieser Arbeit in folgender Weise definiert:

Definition 6. *Variabilitätsmanagement: Das Variabilitätsmanagement umfasst die Modellierung und Verwaltung von Variabilität und ihrer Abhängigkeiten in allen Artefakten einer Produktlinienplattform. Es sorgt weiterhin für die Konsistenz der Variabilität sowohl über alle Artefakte einer Produktlinienplattform als auch über die Zeit hinweg.*

Das Variabilitätsmanagement schafft die notwendigen Voraussetzungen für die Modellierung von Variabilität in Artefakten der Produktlinienplattform. Weiterhin wird auch die Modellierung von Abhängigkeiten zwischen variablen Funktionalitäten ermöglicht. Eine Abhängigkeit liegt zum Beispiel vor, wenn die Auswahl einer variablen Funktionalität durch einen Auftraggeber die Auswahl einer anderen variablen Funktionalität bedingt. Das Variabilitätsmanagement ermöglicht weiterhin die Verwaltung der modellierten Variabilität und der Abhängigkeiten.
Bei der Modellierung von Abhängigkeiten kann es zu Inkonsistenzen zwischen variablen Funktionalitäten kommen. Zum Beispiel kann die gemeinsame Auswahl zweier variabler Funktionalitäten ausgeschlossen werden und gleichzeitig die Auswahl der einen die Auswahl der anderen bedingen. Das Variabilitätsmanagement sorgt daher für die Konsistenz der modellierten Variabilität und der Abhängigkeiten.

Variabilitätstypen

Bei der Modellierung von Variabilität werden zwei unterschiedliche Typen unterschieden: der additive und der modifizierende [Maß07]. Ersterer Typ besagt, dass eine Funktionalität, z. B. ein Element in einem Modell, ausgewählt oder weggelassen werden kann. Die Modifikation hingegen erlaubt die Ausprägung einer Funktionalitäten mit Hilfe der Auswahl eines Wertes aus einer Wertemenge. Beispielsweise kann die Anzahl der Kundenkarten, die eine Person in einem Kundenbindungsprogramm besitzen darf, über eine Modifikation bestimmt werden.
Bei der additiven Variabilität können zwei Typen von Abhängigkeiten zwischen Funktionalitäten definiert werden:

- Implikation
- Ausschluss

Erstere bedeutet, dass die Auswahl einer additiven Funktionalität die Auswahl einer anderen additiven Funktionalität herbeiführt. Im letzteren Fall schließen sich die beiden Funktionalitäten aus.
Für die Modifikation existiert als Abhängigkeit die *Beeinflussung*. Hierbei führt die Auswahl eines Wertes einer Modifikation zur Auswahl eines bestimmten Wertes einer anderen Modifikation.

Neben den bisher betrachteten Typen von Variabilität unterscheidet [PBL05] weitere:

- **Zeitliche und räumliche Variabilität:** Die Zeitliche Variabilität bezeichnet die Veränderung eines Artefakts über die Zeit. Im Unterschied dazu existiert bei der räumlichen Variabilität das Artefakt zur gleichen Zeit in unterschiedlicher Ausprägung an unterschiedlichen Orten. In dieser Arbeit wird Variabilität als räumliche Variabilität verstanden, da bei der Software-Produktlinienentwicklung unterschiedliche Produkte zur gleichen Zeit angeboten werden.
- **Externe und interne Variabilität:** Externe Variabilität ist solche, die für den Auftraggeber relevant ist. Die interne Variabilität ist für den Kunden nicht relevant und auch nicht sichtbar. In [PBL05] wird die Veränderlichkeit der technischen Realisierung als interne Variabilität bezeichnet.

Bindungszeitpunkte von Variabilität

Im Rahmen der Ableitung eines Produktes werden additive Funktionalitäten und Werte an Modifikationen aus- bzw. abgewählt. Dies wird auch als Auflösung oder *Bindung* von Variabilität bezeichnet. Die Bindung der Variabilität kann in unterschiedlichen Phasen des Entwicklungsprozesses passieren. [SGB01, LKK04] unterscheiden folgende Bindungszeitpunkte (vgl. auch [Maß07]):

- Produktspezifikation: Additionen und Modifikationen werden bei der Produktspezifikation an Variationspunkte gebunden.
- Architekturspezifikation: Additionen und Modifikationen werden beim Entwurf an Variationspunkte gebunden.
- Kompilierung: Additionen und Modifikationen werden zum Zeitpunkt der Kompilierung des Quellcodes an zugehörige Variationspunkte gebunden.
- Installation: Additionen und Modifikationen werden bei der Installation eines Produktes gebunden.
- Start: Additionen und Modifikationen werden beim Start eines Produktes an zugehörige Variationspunkte gebunden.
- Laufzeit: Additionen und Modifikationen werden zur Laufzeit des Produktes gebunden.

Im Rahmen dieser Arbeit ist der Bindungszeitpunkt Produktspezifikation relevant, da die Anforderungsartefakte und ihre dazugehörigen Testfälle zu diesem Zeitpunkt festgelegt werden müssen.

2.2 Anforderungsspezifikation für Einzelsystem-Entwicklung

Die Anforderungsspezifikation (Requirements Engineering) ist traditionell die erste Tätigkeit während der Entwicklung von Softwaresystemen [Bal01, Poh08]. Sie hat Auswirkungen auf alle weiteren Phasen des Entwicklungsprozesses. Die so spezifizierten Anforderungen sind in den meisten Fällen Änderungen während der weiteren

Entwicklung unterworfen. Daher ist die Anforderungsspezifikation in vielen Softwarentwicklungsvorgehen eine kontinuierliche Tätigkeit [Rup09].
Die (formale) Spezifikation von Anforderungen führt zu Modellen, die bestimmte Eigenschaften des späteren Softwaresystems definieren. Diese Modelle stellen ein Abbild der Wirklichkeit dar [BS04]. Ein Modell besitzt nach [Sta74] drei Eigenschaften:

- Abbildung: Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
- Verkürzung: Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffer bzw. Modellnutzer relevant erscheinen.
- Pragmatismus: Pragmatismus bedeutet soviel wie Orientierung am Nützlichen. Ein Modell ist einem Original nicht von sich aus zugeordnet. Die Zuordnung wird durch die Fragen „Für wen?“, „Warum?“ und „Wozu?“ relativiert. Ein Modell wird vom Modellschaffer bzw. Modellnutzer innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Zweck für ein Original eingesetzt. Das Modell wird somit interpretiert.

Die Spezifikation von Anforderungen an Softwaresystemen erfordert zunächst die Klärung des Begriffs Anforderung. Die IEEE [IEE90] definiert eine *Anforderung* in folgender Art und Weise:

Definition 7.

a condition or capability needed by a user to solve a problem or achieve an objetive

a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document

a documented representation of a condition or capability as in definition 1 or 2

Werden Anforderungen nach einem definierten Vorgehen und mit Hilfe einer Modellierungssprache erfasst, wird dies als Anforderungsspezifikation bezeichnet. Die dabei entstehenden Modelle werden in dieser Arbeit als *Anforderungsartefakte* bezeichnet. Es werden funktionale und nicht-funktionale Anforderungen unterschieden: Eine funktionale Anforderung beschreibt eine Funktion, die ein System oder

eine Komponente bereitstellen muss [IEE90]). Nicht-funktionale Anforderungen sind Eigenschaften, die das System besitzen muss [RR99].
Die IEEE-Norm definiert, dass eine Anforderungsspezifikation adäquat, eindeutig, vollständig, widerspruchsfrei, priorisiert, prüfbar, änderbar und nachverfolgbar sein muss [IEE98]:

- Adäquatheit: Die dokumentierten Anforderungen müssen mit den tatsächlichen Anforderungen des Kunden an das Softwareprodukt übereinstimmen und diese angemessen und korrekt wiedergeben.
- Vollständigkeit: Es müssen alle Anforderungen des Kunden an das Softwareprodukt enthalten sein.
- Widerspruchsfreiheit: Die dokumentierten Anforderungen dürfen sich nicht gegenseitig widersprechen, da das Softwareprodukt ansonsten nicht realisierbar ist.
- Verständlichkeit: Die dokumentierten Anforderungen müssen sowohl für den Kunden als auch für den Anbieter verständlich sein.
- Präzision: Die dokumentierten Anforderungen müssen eindeutig von allen Beteiligten interpretierbar sein, um Fehler bei der Realisierung und Validierung zu vermeiden.
- Prüfbarkeit: Die dokumentierten Anforderungen müssen so beschrieben sein, dass sie im realisierten Softwareprodukt nachgewiesen werden können.

Für die Spezifikation von Anforderungen existieren unterschiedliche Arten von Modellen. In dieser Arbeit werden für die Modellierung von Anforderungen Anwendungsfallmodelle eingesetzt, die im nun folgenden Abschnitt beschrieben werden.

2.2.1 Anwendungsfallmodellierung

Anforderungen haben häufig einen informellen Charakter, da der Auftraggeber zu Beginn der Entwicklung selten exakt spezifizieren kann, welche funktionalen und nicht-funktionalen Eigenschaften das System später haben soll. Die Spezifikation von Anforderungsartefakten ist somit der erste Schritt für eine Formalisierung von

Anforderungen. Dabei ist es von Interesse, dass der Auftraggeber mit der Modellierungstechnik für Anforderungsartefakte vertraut ist oder diese schnell erlernen kann. Somit ist gewährleistet, dass er die Spezifikation von Anforderungsartefakten zum Beispiel durch Inspektionen (*Reviews*) unterstützen kann. An dieser Stelle bieten sich zum Beispiel textuelle Anwendungsfallbeschreibungen, im folgenden auch Anwendungsfälle genannt, an. Sie werden in [Coc00] beschrieben und in der Industrie häufig für die Spezifikation von Anforderungen genutzt [MW08], [BFGL06], [FS05] und [FP05].
Die Vorteile sind dabei

- die strukturierte Spezifikation von Texten,
- die gute Verständlichkeit und
- die hohe Ausdrucksmächtigkeit.

Im Gegensatz zu reinen Prosa-Texten bietet eine textuelle Anwendungsfallbeschreibung eine Strukturierung in Vorbedingung, Auslöser, Ablaufschritte und Nachbedingung an. Dabei bietet sie ein hohes Maß an Verständlichkeit, auch für ungeschulte Benutzer, da Anforderungen in natürlicher Sprache erfasst werden. Wenn zum Beispiel die Vorbedingung einer Anwendungsfallbeschreibung mit Hilfe der *Object Contraint Language* (OCL) [OMG03] definiert wird, ist die Verständlichkeit für einen ungeschulten Auftraggeber mit hoher Wahrscheinlichkeit gering. Eine natürlich-sprachliche Vorbedingung hingegen kann eher verstanden werden. Die natürliche Sprache bietet auch den Vorteil einer großen Ausdrucksmächtigkeit, im Vergleich zu kontrollierter Sprache. Aus diesem Grund eignen sich textuelle Anwendungsfälle für die Anforderungsdefinition in den frühen Entwicklungsphasen und auch für die Kommunikation mit dem Auftraggeber.
Neben den genannten Vorteilen existieren aber auch Nachteile. Diese entstehen durch

- die potentielle Missverständlichkeit und Ungenauigkeit von spezifizierten Anforderungen,
- den manuellen Aufwand in der Folge und
- eine potentiell geringe Einheitlichkeit.

Die Verwendung natürlicher Sprache kann dazu führen, dass Anforderungen missverständlich spezifiziert werden. Wörter wie zum Beispiel „kann“ oder „sollte“ lassen Interpretationen zu. Dadurch kann es bei der späteren Detaillierung und der darauf folgenden technischen Umsetzung der Anforderungen zu Fehlinterpretationen kommen. Die Verwendung der natürlichen Sprache führt weiterhin zu manuellem Aufwand, da eine maschinelle Verarbeitung in diesem Kontext nicht oder nur eingeschränkt möglich ist. Neben dieser Tatsache bietet die Freiheit bei der Spezifikation von textuellen Anwendungsfallbeschreibungen auch das Risiko einer geringen Einheitlichkeit. Sofern verschiedene Personen an der Modellierung beteiligt sind, kann es zu sehr unterschiedlich spezifizierten Anwendungsfallbeschreibungen kommen. Dabei entsteht das Risiko, dass die gewünschten Informationen bzw. deren Detailtiefe nicht enthalten sind bzw. erreicht wurden.

Existierende Anwendungsfallmetamodelle

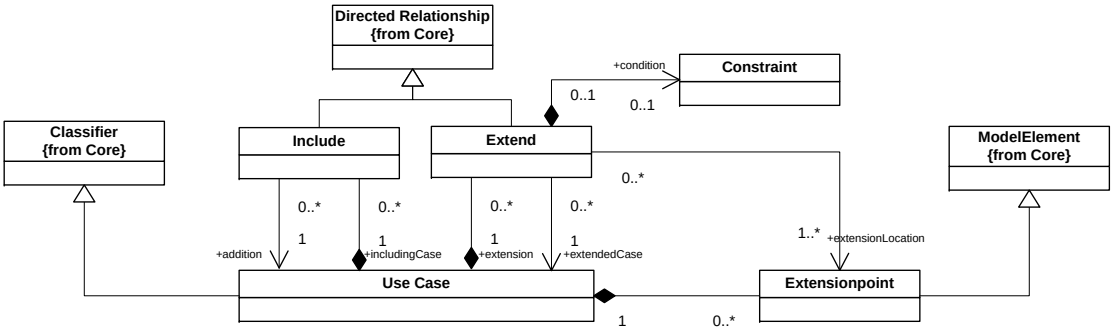


Abbildung 2.4: UML Anwendungsfallmetamodell

In der Literatur werden textuelle Anwendungsfallmodelle in unterschiedlicher Form definiert. Die bekannteste Definition wird in [Coc00] gegeben. In [NUOT01] wird ein Metamodell für Anwendungsfallbeschreibungen auf Basis des Metamodells des UML-Anwendungsfalls definiert. Die Anwendungsfallbeschreibung beinhaltet auch alternative Abläufe, Ausnahmen, sowie Vor- und Nachbedingungen. Das UML-Anwendungsfallmetamodell ist in Abbildung 2.4 dargestellt. Für die Beschreibung des Ablaufs eines Anwendungsfalls können später UML-Aktivitätsdiagramme eingesetzt werden.

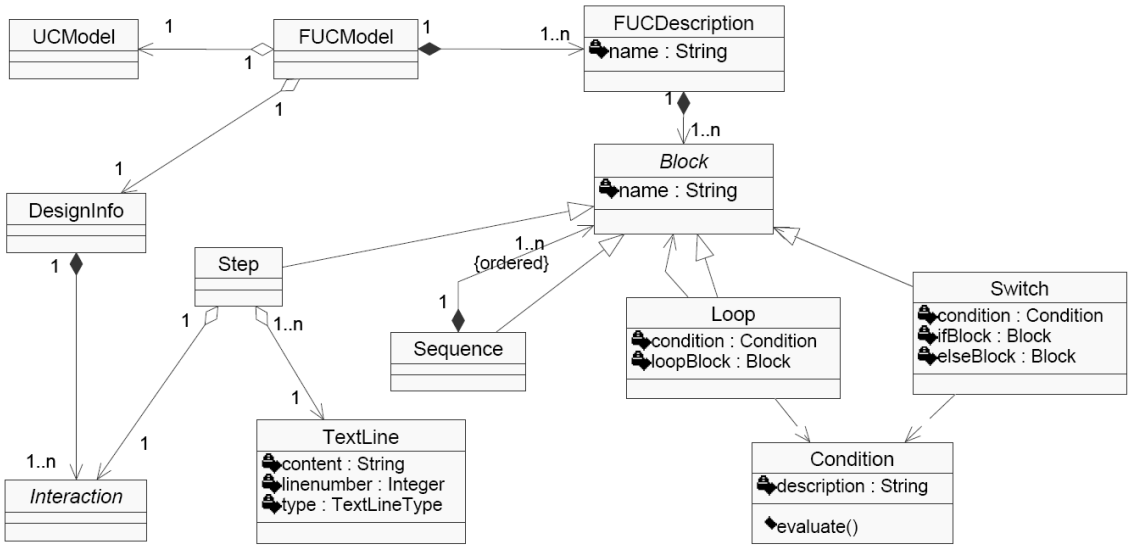


Abbildung 2.5: Metamodell Anwendungsfallbeschreibung nach [FP05]

In [FP05] wird das Metamodell für Anwendungsfallbeschreibungen wie in Abbildung 2.5 definiert. Dabei wird der Fokus auf die Modellierung von Abläufen, bestehend aus Schleifen, Entscheidungen und Sequenzen gelegt. Die Modellierung von Vor- und Nachbedingungen, sowie Ausnahmen steht nicht im Fokus.

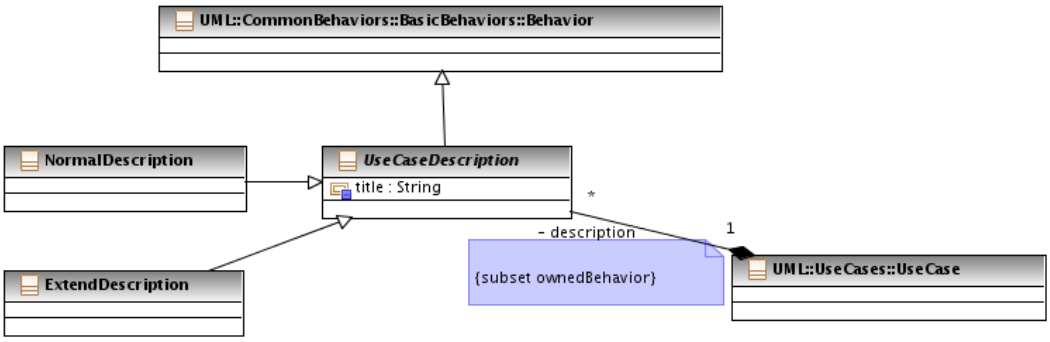


Abbildung 2.6: Metamodell Anwendungsfallbeschreibungnach [Som08]

Der Ansatz aus [Som08] basiert auf dem UML-Anwendungsfallmetamodell. Der Ansatz orientiert sich in seiner Definition stärker an [Coc00]. In Abbildung 2.6 ist die

Anwendungsfallbeschreibung (UseCaseDescription) als Teil des UML-Anwendungsfalls definiert. Zu jedem Anwendungsfall können beliebig viele Anwendungsfallbeschreibungen existieren.

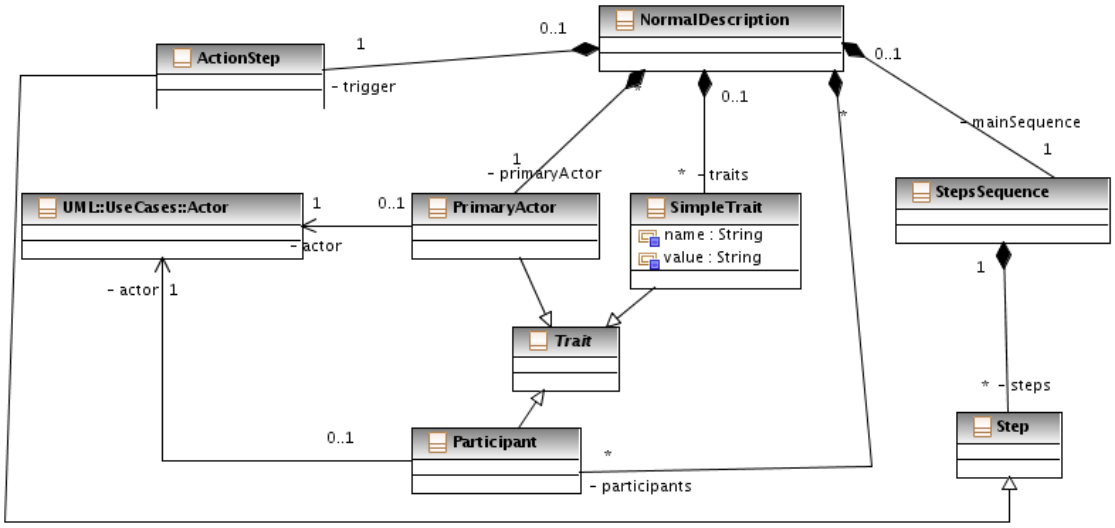


Abbildung 2.7: Metamodell Anwendungsfallbeschreibung: Normaler Ablauf nach [Som08]

Abbildung 2.7 visualisiert die Definition des normalen Ablaufs (NormalDescription). Dabei werden wiederum Ausnahmen und alternative Abläufe nicht berücksichtigt.

Die vorgestellten Ansätze besitzen viele Gemeinsamkeiten und einige Unterschiede. Um nun für diese Arbeit ein eindeutiges Verständnis von Anwendungsfallbeschreibungen zu erreichen, werden die betrachteten Ansätze in einem Anwendungsfallmetamodell vereint.

Konsolidierung der Anwendungsfallmetamodelle

Das konsolidierte Anwendungsfallmetamodell ist in Abbildung 2.8 dargestellt. Zunächst sind *Anwendungsfallbeschreibungen* Teil eines *Use Cases*. Durch diese Definition wird die Verbindung zwischen UML-Use Case und Anwendungsfallbeschreibung hergestellt, wie in [NUOT01, Som08] beschrieben. Jede Anwendungsfallbeschreibung besitzt mindestens ein *Ziel* [Coc00] und mindestens einen *Akteur* [NUOT01, FP05,

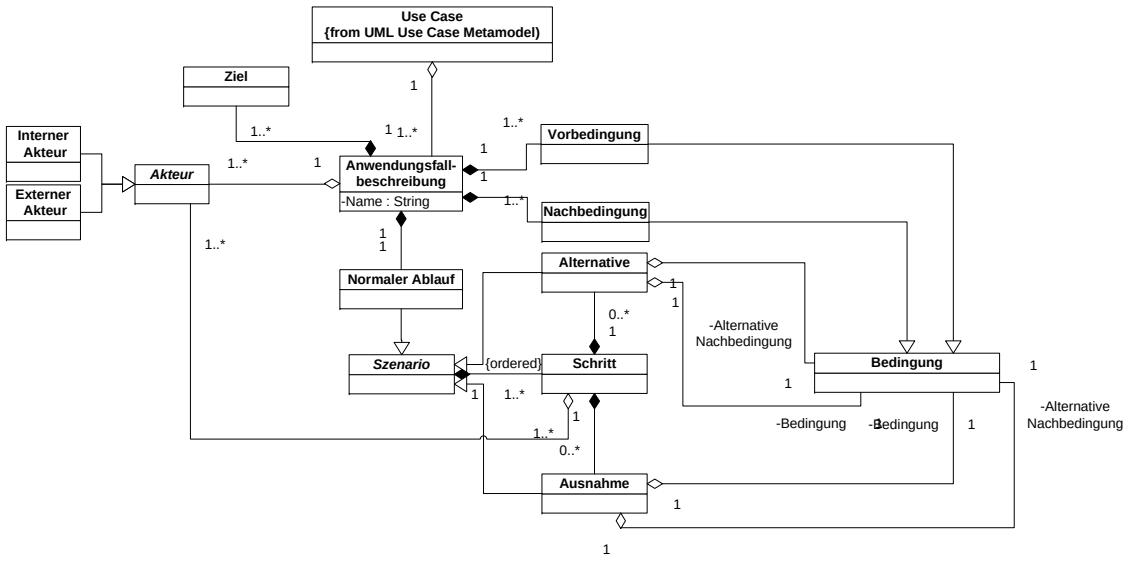


Abbildung 2.8: Anwendungsfallmetamodell

Som08]. Ein Akteur kann ein interner, also beispielsweise ein (Teil-) System, oder ein externer, beispielweise eine Person, sein [Coc00].

Weiterhin besitzt die Anwendungsfallbeschreibung mindestens eine *Vor-* und *Nachbedingung* [Coc00, NUOT01]. An dieser Stelle wären unterschiedliche Definitionen möglich. In [NUOT01] werden Vor- und Nachbedindungen an jeden Schritt einer Anwendungsfallbeschreibung assoziiert, bei [Coc00] nur an den Anwendungsfall selbst. In dieser Arbeit wurde die Entscheidung für letztere Variante getroffen, da immer auf den Zustand vor und nach der Durchführung einer Anwendungsfallbeschreibung fokussiert wird.

Eine Anwendungsfallbeschreibung besitzt nun genau einen *normalen Ablauf* [Coc00, Som08]. Dieses Szenario besteht aus einer geordneten Menge von *Schritten*. Jeder Schritt hat mindestens einen Akteur, der an seiner Durchführung beteiligt ist. Ein Schritt kann wiederum beliebig viele *alternative* oder *Ausnahme-Szenarien* enthalten, welche wiederum aus Schritten besteht [Coc00, NUOT01]. Alternativ- und Ausnahmeszenarien besitzen jeweils eine Eintrittsbedingung (*Bedingung*) und eine *alternative Nachbedingung*.

2.3 Softwaretest für Einzelsystem-Entwicklung

Die Qualität von Software ist ein entscheidender Faktor für den Erfolg von Produkten und Unternehmen. Viele Beispiele haben in der Vergangenheit gezeigt, dass eine unausgereifte oder fehlerhafte Software zu großen Verzögerungen bei der Fertigstellung oder sogar zum Scheitern eines Projektes führen kann. Eines der vielen bekannten Beispiele ist das Gepäckabfertigungssystem des Flughafens in Denver [Don01].
Zur Sicherstellung der geforderten Softwarequalität und der Beseitigung von Fehlern spielen Softwaretests eine wichtige Rolle. Ziel dieser Tests ist es, Fehler in Softwaresystemen aufzudecken und durch die Behebung der aufgedeckten Fehler die Qualität der Software zu verbessern.
Im folgenden Abschnitt werden die Grundlagen des Testens von Software, basierend auf [Obe09a], vorgestellt und die für diese Arbeit wichtigen Begriffe erläutert.

2.3.1 Grundlagen des Softwaretests

Um die Qualität von Software sicherzustellen, gibt es unterschiedliche Qualitätssicherungsmaßnahmen, wie in Abbildung 2.9 dargestellt. Zum einen gibt es konstruktive Maßnahmen, die sich mit der Verbesserung der Softwarequalität im Softwareentwicklungsprozess, etwa durch die Verwendung von Entwurfsmustern (Pattern), beschäftigen. Zum anderen gibt es prozessbasierte oder organisatorische Maßnahmen, die durch die Orientierung an Standard-Entwicklungsprozessen und Qualitätsstandards wie ISO9000 [ISO], CMM [Dym02] oder SPICE [HDHMM06] versuchen die Qualität einer Software, ebenfalls bei der Entstehung, zu verbessern. Ein weiterer Ansatz der Software-Qualitätssicherung ist der sogenannte analytische Ansatz. Bei diesem werden Analysen auf Entwürfen, Modellen und zu testenden Systemen (System Under Test, SUT) ausgeführt und es wird versucht aufgrund der Analyseergebnisse die Qualität einer Software zu verbessern. Die analytischen Qualitätssicherungs-Maßnahmen unterteilen sich in die statische und die dynamische Analyse [Lig02].
Während bei der statischen Analyse der Code analysiert, Modelle überprüft und Entwürfe untersucht werden, wird bei der dynamischen Analyse die Ausführung der zu testenden Software mit systematisch festgelegten Eingabedaten getestet.

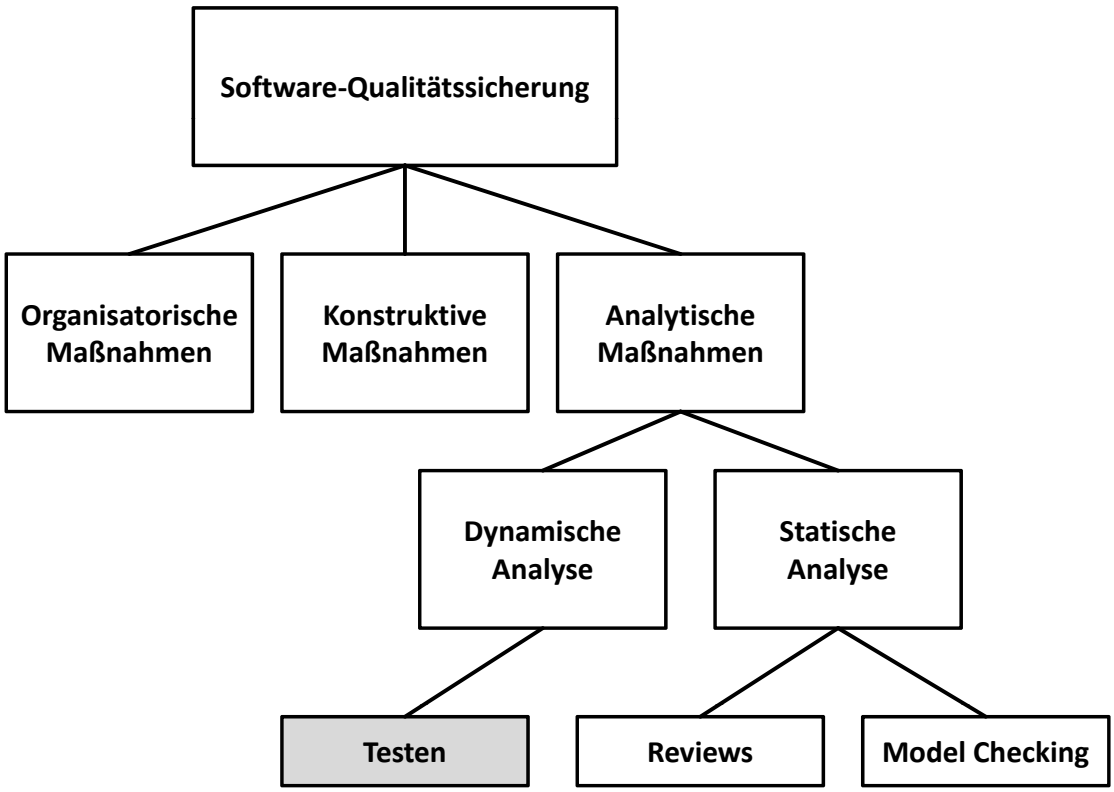


Abbildung 2.9: Software-Qualitätssicherungsmaßnahmen

Durch die Behebung der beim Testen des Systems gefundenen Fehler kann die Qualität der Software gesteigert werden. In dieser Arbeit steht das Testen im Fokus.

Fehlerbegriff

Durch Testen ist es möglich, Fehler in einem System zu finden. Der Begriff Fehler wird in der Literatur in Fault, Error und System Failure unterteilt [SL05]. Ein Fault beschreibt einen Defekt der im Code versteckt ist, aber vielleicht nie gefunden wird. Ein Error beschreibt einen Fehlerzustand eines Systems, der entdeckt wurde, aber nicht zum Abstürzen des Systems führt. Ein System Failure dagegen beschreibt einen Defekt, der immer zu einem Absturz des Systems führt. Ist die Unterscheidung in Fault und Failure nicht nötig, so kann allgemein von Defekt bzw. Fehler gesprochen werden.

Testbegriff

Der Begriff des Testens ist in [IEE90] wie folgt definiert:

Definition 8. *A test is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.*

Testen ist demnach also eine Tätigkeit, um Fehler in einem System oder einer Komponente gezielt und systematisch aufzudecken. Es ist jedoch nicht möglich, durch Testen alleine die Abwesenheit von Fehlern nachzuweisen. Selbst wenn alle ausgeführten Testfälle keinen einzigen Fehler mehr aufdecken, kann trotzdem nicht mit völliger Sicherheit ausgeschlossen werden, dass das Programm Fehler enthält. Es könnte weitere Testfälle geben, die weitere Fehlerzustände der Software entdecken könnten. Dijsktra formulierte diesen Sachverhalt in [Dij70] wie folgt:

> Program testing can be used to show the presence of bugs, but never to show their absence.

Um die Fehlerfreiheit eines Systems zu beweisen, müssten andere Vorgehensweisen, wie zum Beispiel die formale Code-Verifikation oder Model Checking [Bin99], angewandt werden.
Wird durch einen Test ein Defekt in einem System gefunden, so muss zunächst die Stelle in der Software lokalisiert werden, die den Defekt verursacht. Das Lokalisieren und Beheben des Defektes ist Aufgabe des Softwareentwicklers und wird auch als Debugging (Fehlerbereinigung, Fehlerkorrektur) bezeichnet.

Testziele

In der Literatur gibt es verschiedene Beschreibungen der Zielsetzungen von Softwaretests. Nach [SL05] verfolgt das Testen von Software folgende Ziele:

- Ausführung des Programms mit dem Ziel, Fehlerwirkungen nachzuweisen
- Ausführung des Programms mit dem Ziel, die Qualität zu bestimmen
- Ausführung des Programms mit dem Ziel, Vertrauen in das Programm zu erhöhen

- Analysieren des Programms oder der Artefakte, um Fehlerwirkungen vorzubeugen

Oft wird der gesamte Prozess, ein Programm auf systematische Weise auszuführen, um die korrekte Umsetzung der Anforderungen nachzuweisen, das Vertrauen zu erhöhen und Fehlerwirkungen aufzudecken, als Test bezeichnet.

Testbeschreibung

In der Literatur gibt es sehr viele verschiedene Bezeichnungen und Beschreibungsarten für die verschiedenen Arten von Softwaretests. In [SL05] wurden folgende Kategorien definiert, denen zufolge Tests bezeichnet werden können:

- Testziel: Das Testziel bezeichnet den Zweck einer Testart
- Testmethode: Der Test wird nach der Methode benannt, die zur Spezifikation und Durchführung der Tests eingesetzt wird (z. B. geschäftsprozess-basierter Test)
- Testobjekt: Der Test wird nach Art des Testobjektes, das getestet werden soll, benannt (z. B. GUI-Test, Datenbank-Test)
- Teststufe: Der Test wird nach der Stufe des zugrunde liegenden Vorgehensmodells benannt (z. B. Komponententest, Systemtest)
- Testperson: Der Test wird nach dem Personenkreis bezeichnet, welcher die Tests durchführt (z. B. Entwicklertests, Anwendertest)
- Testumfang: Tests werden aufgrund ihres Umfangs unterschieden (z. B. partieller Regressionstest, Volltest)

Je nachdem unter welchem Blickwinkel ein Test beschrieben wird, rückt eine dieser Kategorien in den Vordergrund. Um einen Test hinreichend zu beschreiben, sollten alle diese Kategorien für den Test definiert werden.

Testaufwand

Um die Fehlerfreiheit eines Programms nachzuweisen, müsste das Programm in allen möglichen Situationen, mit allen möglichen Eingaben und unter Berücksichtigung

aller möglichen Randbedingungen getestet werden [SL05]. Ein solch vollständiger Test ist schon für kleine und mittlere Programme praktisch nicht durchführbar. Durch die Vielzahl an kombinatorischen Möglichkeiten zum Beispiel bei der Eingabe von Daten ergibt sich eine zu hohe Anzahl an Tests, die durchzuführen wären. Somit kann in der Praxis immer nur ein Teil aller denkbarer Testfälle berücksichtigt werden.
Da ein vollständiger Test nicht möglich ist, muss der Testaufwand immer in einem vernünftigen Verhältnis zum erzielbaren Ergebnis stehen. Testen ist ökonomisch sinnvoll, solange die Kosten für das Finden und Beseitigen eines Fehlers niedriger sind als die Kosten, die mit dem Auftreten eines Fehlers bei der Nutzung verbunden sind [PKS00]. Der Testaufwand muss daher immer in Abhängigkeit vom erwarteten Risiko bei fehlerhaftem Verhalten des Programms gewählt werden.

Testüberdeckungsstrategie

Da in den meisten Fällen ein vollständiger Test eines Testobjektes nicht möglich oder ökonomisch sinnvoll ist, werden oftmals Kriterien definiert, die festlegen, wie ein Testobjekt mit Testfällen „überdeckt“ werden soll. Zunächst unterscheidet man dabei Black- und Whitebox-Techniken. Erstere ignorieren beim Test den inneren Aufbau (Quellcode) des Testobjektes und betrachten nur seine Schnittstellen mit den möglichen Ein- und Ausgaben. Beim Whitebox-Testen (auch Glasbox-Testen genannt) wird der innere Aufbau des Testobjektes berücksichtigt. Hier sind zum Beispiel quellcodebasierte Testüberdeckungsstrategien einzuordnen. Die Kombination aus beiden Techniken wird als Greybox bezeichnet. In dieser Arbeit wird auf Testüberdeckungsstrategien aus dem Bereich Blackbox fokussiert.
Die Testüberdeckungsstrategie selbst ist eine Möglichkeit, um ein Kriterium für das Ende eines Tests zu definieren. Dabei gibt die Testüberdeckung an, zu welchem Grad ein Testobjekt durch die Testfälle überdeckt werden muss. Um ein solches Testüberdeckungskriterium bestimmen zu können, ist zum Beispiel die Kenntnis des inneren Ablaufs (*Ablaufgraph*) des Testobjektes einsetzbar. Mögliche Überdeckungskriterien wären hier Kanten-, Knoten-, Zweig- oder Pfadüberdeckung [Bin99]. Eine andere Testüberdeckungsstrategie ist die *Äquivalenzklassenanalyse*. Diese teilt Ein- oder Ausgabeparameter in Wertebereiche ein, die aus Sicht des Testens das gleiche Verhalten beim Testobjekt auslösen [SL05]. Bei den in dieser Arbeit zur Anforderungsdefinition verwendeten Anwendungsfallbeschreibungen kann ein Überdeckungskriterium basie-

rend auf der Ablaufbeschreibung der Anwendungsfälle definiert werden. Zusätzlich können mit Hilfe einer Äquivalenzklassenanalyse die Ein- und Ausgabeparameter der einzelnen Schritte der Anwendungsfallbeschreibung untersucht werden.

Der fundamentale Testprozess

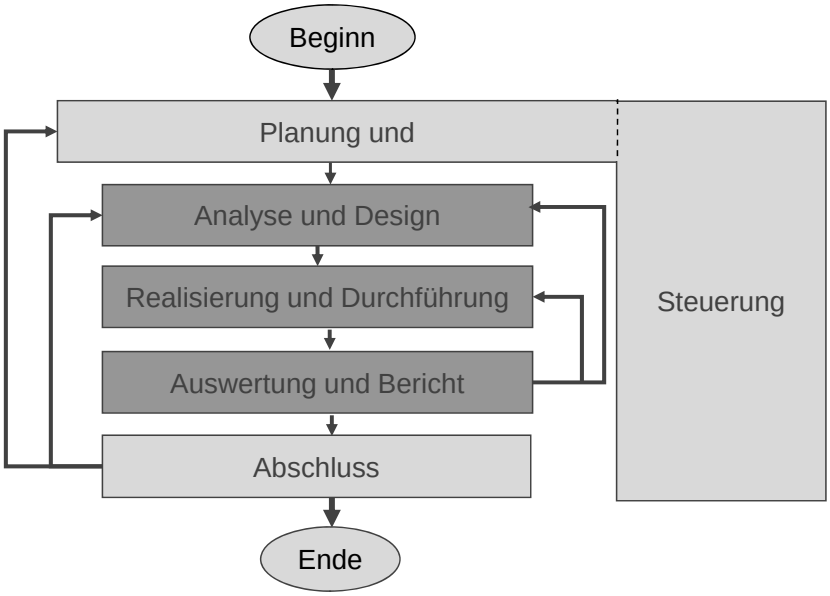


Abbildung 2.10: Der fundamentale Testprozess [SL05]

Um ein Softwareprojekt strukturiert testen zu können, ist es sinnvoll die Testaktivitäten in Arbeitsabschnitte zu unterteilen und in einen Testprozess zu integrieren. In [SL05] wird mit dem *fundamentalen Testprozess* ein solcher Prozess definiert (vgl. Abbildung 2.10). Die einzelnen Arbeitsschritte des Testprozesses sind: *Planung und Steuerung*, *Analyse und Design*, *Realisierung und Durchführung*, *Auswertung und Bericht* sowie *Abschluss* der Testaktivitäten.
Aufgabe der ersten Phase des Prozesses, der Testplanung und Steuerung, ist es eine Teststrategie, die Testziele und den Testansatz festzulegen. Außerdem muss in dieser Phase ein detaillierter Testplan inklusive benötigter Ressourcen, angestrebter Testüberdeckung und grobem Zeitplan definiert werden. In der Steuerungsphase, die sich über den ganzen Testprozess erstreckt, werden die Ergebnisse gemessen, analysiert und aufgezeichnet und es wird gegebenenfalls aufgrund der Analysen in den Testprozess steuernd eingegriffen.

In der zweiten Phase, der Testanalyse und Designphase, werden detaillierte Testkriterien und Bedingungen aus dem gemeinsamen Testplan spezifiziert. Als Basis dafür dienen alle Artefakte, die Anforderungen für das Testobjekt enthalten. Bei dieser Spezifikation wird zunächst geprüft, ob die Testbasis die nötige Reife besitzt, d. h., ob alle Anforderungen dokumentiert sind. Ist dies der Fall, wird eine Testumgebung entwickelt und die logischen Testfälle werden spezifiziert. Zur Spezifizierung der Testfälle gehören auch die Definition der erwarteten Ergebnisse und die Festlegung der Randbedingungen.
In der Testrealisierungs- und Durchführungsphase werden Testdaten und Testszenarien definiert, die Testumgebung vorbereitet und die Tests ausgeführt. Das bedeutet, dass den logischen Testfällen Eingabedaten hinzugefügt und sie somit in konkrete Testfälle überführt werden. Während der Ausführung der Tests werden die Testergebnisse mit den erwarteten Ergebnissen verglichen. Sollte bei diesem Vergleich eine Abweichung zu den erwarteten Ergebnissen gefunden werden, deutet dies auf einen Fehler hin. Der Fehler kann dabei in der Software selber, aber auch in der Spezifikation des Testfalles oder in den Anforderungsspezifikationen des Systems liegen. Der Fehler wird daraufhin analysiert und behoben und der entsprechende Testfall wiederholt.
In der Testauswertungs- und Berichtsphase werden die Testergebnisse mit den definierten Zielen verglichen. Anhand der in der Planungsphase festgelegten Test-Endekriterien und den definierten Zielen wird entschieden, ob weitere Tests durchgeführt werden oder ob der Testprozess beendet werden kann. Bei Beendigung des Testprozesses wird ein umfassender Abschlussbericht erstellt, der nochmals die Testdurchführungsphase resümiert und alle Ergebnisse darstellt, die zum Beenden der Testaktivitäten geführt haben.
In der letzten Phase des Testprozesses werden die Testaktivitäten abgeschlossen. Alles Wissen rund um die Software und die Tests wird gesammelt und dokumentiert, um die gemachten Erfahrungen für die Wartungsphase der Software und für zukünftige Projekte zu erhalten. Noch übrige Fehler werden als Änderungsanforderung festgehalten.
Diese Arbeit fokussiert die Testdesign- und die Realisierungs-Phase des *Fundamentalen Testprozesses*.

2.3.2 Testen im Entwicklungsprozess

Das Testen einer Software ist eng mit dem verwendeten Vorgehensmodell der Softwareentwicklung verbunden. Obwohl das Testen einen eigenen Prozess darstellt, können die einzelnen Phasen eines Vorgehensmodells mit bestimmten Teststufen verbunden werden. Aus Sicht des Testens spielt das allgemeine V-Modell eine besondere Rolle [Boe79]. Das Modell zeigt die Testaktivitäten als gleichwertig zur Entwicklung und Programmierung. Die folgenden Prinzipien lassen sich auch auf das Software-Produktlinienparadigma übertragen.

Das allgemeine V-Modell

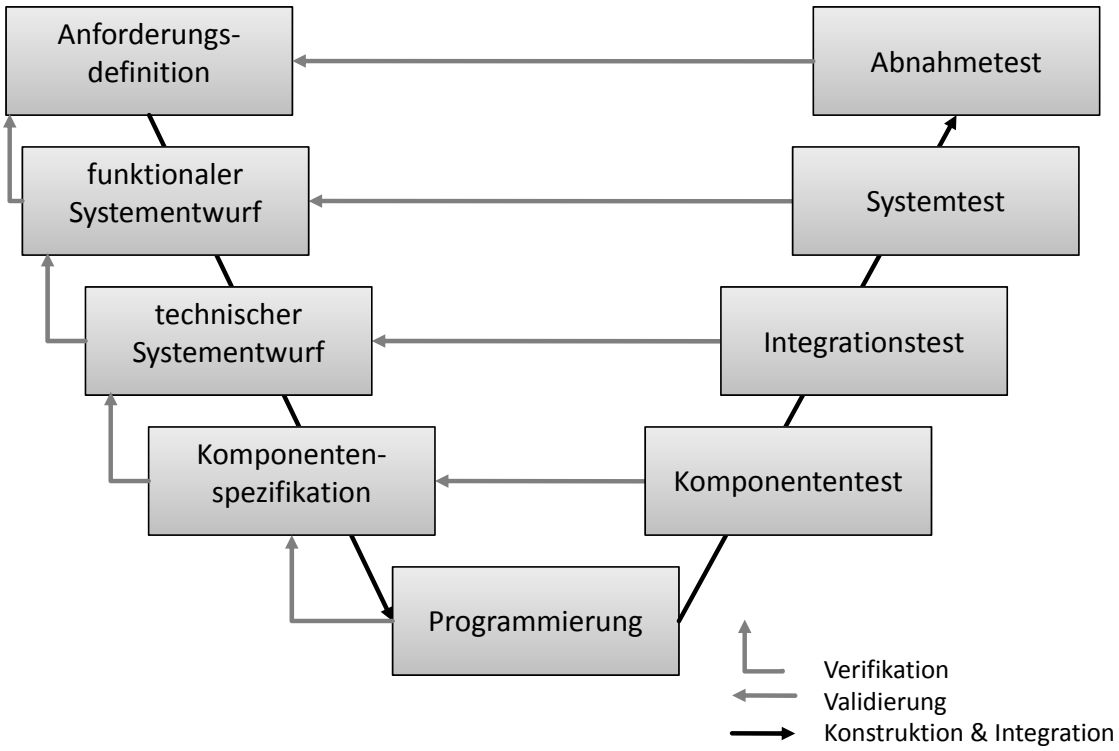


Abbildung 2.11: Das allgemeine V-Modell [Boe79]

Das allgemeine V-Modell, dargestellt in Abbildung 2.11 beschreibt die Entwicklungstätigkeiten und Testarbeiten als gleichberechtigte, zueinander korrespondierende Tätigkeiten. Der Name „V-Modell“ entsteht durch die v-förmige Anordnung der einzelnen Tätigkeiten. Der linke Ast des Modells steht für den immer detaillierter

werdenden Entwicklungsprozess. Er umfasst die Tätigkeiten Anforderungsdefinition, funktionaler Systementwurf, technischer Systementwurf, Komponentenspezifikation und Programmierung. Der rechte Ast des Modells steht für Integrations- und Testarbeiten. Er umfasst die Tätigkeiten: Komponententest, Integrationstest, Systemtest und Abnahmetest.
Im Verlauf der Testaktivitäten werden elementare Programmbausteine sukzessive zu größeren Teilsystemen zusammengesetzt und jeweils auf die richtige Funktion geprüft.
Fehler werden immer auf der Ebene gefunden, auf der sie entstanden sind. Daher ordnet der rechte Ast jeder Spezifikations- oder Entwicklungsphase des Entwicklungsprozesses eine korrespondierende Teststufe zu.
Die erste Stufe, der Komponententest, prüft, ob jede einzelne Komponente für sich die Vorgaben ihrer Spezifikation erfüllt. Der Integrationstest überprüft, ob Gruppen von Komponenten den spezifizierten Vorgaben entsprechend zusammenspielen und interagieren. Beim anschließenden Systemtest wird überprüft, ob das zusammengesetzte System als Ganzes die definierten Anforderungen erfüllt. Der Abnahmetest ist die letzte Teststufe. Er überprüft, ob das System aus Kundensicht die vertraglich festgehaltenen Anforderungen erfüllt. In jeder Teststufe wird also überprüft, ob die Entwicklungsergebnisse der jeweiligen Stufe die Anforderungen erfüllen, die für die jeweilige Abstraktionsebene spezifiziert wurden.
Das Prüfen der Entwicklungsergebnisse gegenüber den definierten Anforderungen wird Validierung genannt. Beim Validieren bewertet der Tester, ob ein (Teil-) Produkt eine festgelegte (spezifizierte) Aufgabe tatsächlich löst und deshalb für seinen Einsatzzweck tauglich bzw. nützlich ist [SL05]. Es wird also überprüft, ob das Produkt die durch die Anforderungen spezifizierte Aufgabe erfüllen kann.
Neben der Validierung fordert das V-Modell auch noch die sogenannte verifizierende Prüfung. Die Verifikation ist im Gegensatz zur Validierung auf eine einzelne Entwicklungsphase bezogen und soll die Korrektheit und Vollständigkeit eines Phasenergebnisses relativ zu seiner Spezifikation nachweisen [SL05]. Es wird also geprüft, ob die Spezifikation korrekt umgesetzt wurde, unabhängig davon, ob damit auch die beabsichtigte Aufgabe erfüllt wird. In der Praxis beinhaltet jeder Test beide Aspekte, wobei der Validierungsanteil mit steigender Teststufe zunimmt.
Die Unterscheidung in verschiedene Teststufen im V-Modell ist mehr als eine zeitliche Unterteilung von Testaktivitäten. Es werden vielmehr sehr unterschiedliche

Stufen definiert, die unterschiedliche Ziele verfolgen und an denen unterschiedliche Testmethoden, Testwerkzeuge und Personen beteiligt sind.
Diese Arbeit fokussiert den Systemtest von Software Produktlinien. Daher wird im Folgenden diese Teststufe näher betrachtet und eine Definition für Testfälle für den Systemtest getätigt.

Systemtest

In der Teststufe Systemtest wird das integrierte System getestet, um zu überprüfen, ob die spezifizierten Anforderungen vom System erfüllt werden. Im Gegensatz zu den beiden niedrigeren Teststufen, in denen das System gegen technische Spezifikationen aus Sicht der Softwareentwickler getestet wird, betrachtet der Systemtest das System aus der Sicht des Auftraggebers und des späteren Anwenders. Viele Funktionen und Systemeigenschaften resultieren aus dem Ineinandergreifen aller Systemkomponenten und sind somit erst auf dieser Ebene des Gesamtsystems beobachtbar und überprüfbar.
Ziel des Systemtests ist es zu überprüfen, ob und wie gut das fertige System die definierten funktionalen und nicht-funktionalen Anforderungen erfüllt. Außerdem wird versucht unvollständig, fehlerhaft oder widersprüchlich umgesetzte Anforderungen im System aufzudecken und undokumentierte oder vergessene Anforderungen zu identifizieren.
Die Überlegungen, die in dieser Arbeit angestellt werden, beziehen sich ausschließlich auf funktionale Anforderungen, die auf Systemtestebene getestet werden.

2.3.3 Testfallspezifikation für den Systemtest

Im Rahmen der Testfallspezifikation für den Systemtest werden die drei Aktivitäten „Analyse der Testbasis“, „Spezifikation von logischen Testfällen“ und „Spezifikation von konkreten Testfällen“ unterschieden. Die einzelnen Aktivitäten bauen aufeinander auf und werden in der angegebenen Reihenfolge durchlaufen.

Analyse der Testbasis

Die Testbasis bildet die Grundlage für die Testfallspezifikation. Sie kann zum Beispiel aus Anwendungsfallbeschreibungen, Architekturbeschreibungen, Design- und Schnittstellenbeschreibungen bestehen. Bei der Zusammenstellung einer Testbasis

wird analysiert, ob die zu verwendenden Artefakte ausreichend detailliert sind oder ob sie gegebenenfalls noch nachgebessert werden müssen. Die Testbasis bildet die Grundlage für den Entwurf der logischen sowie konkreten Testfälle.

Definition 9. *Eine Testbasis besteht aus Artefakten des Entwicklungsprozesses, die hinsichtlich ihrer korrekten Umsetzung im zu testenden Softwaresystem überprüft werden sollen.*

Spezifikation von logischen Testfällen

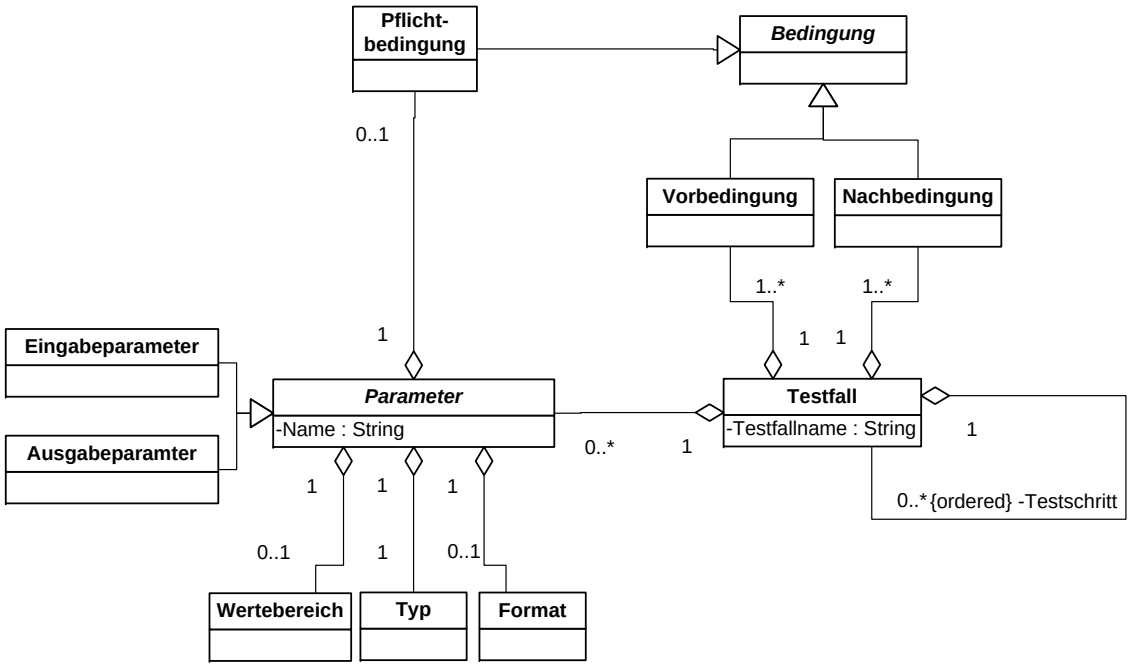


Abbildung 2.12: Metamodell logischer Testfall

Der in dieser Arbeit genutzte Aufbau eines logischen Testfalls ist in Abbildung 2.12 dargestellt. Der logische Testfall besteht aus einer geordneten Menge von *Testschritten*, die wiederum logische Testfälle darstellen [Bin99, SL05]. Jeder Testfall besitzt mindestens eine *Vor-* und *Nachbedingung*, die den Zustand des Systems unter Test vor und nach der Durchführung des Testfalls beschreibt.
Weiterhin kann jeder Testfall beliebig viele *Ein-* und *Ausgabeparameter* besitzen. Diese werden mindestens durch einen Namen und einen *Typ* definiert. Der Typ kann zum Beispiel numerisch oder alphanumerisch sein. Falls der Typ des Parameters zum

Beispiel alphanumerisch ist, kann über *Format* der Aufbau näher beschrieben werden, zum Beispiel über reguläre Ausdrücke [HU94]. Weitere Einschränkungen können für einen Parameter über den *Wertebereich* definiert werden. Hier kann zum Beispiel eine Menge mit bestimmten gültigen Werten hinterlegt werden. Zuletzt kann jeder Parameter über eine *Pflichtbedingung* verfügen, die beschreibt, unter welcher Bedingung der Parameter ausgefüllt werden muss. Eine Pflichtbedingung kann zum Beispiel definieren, dass ein Parameter gefüllt werden muss, wenn ein bestimmter anderer Parameter gefüllt ist.
Logische Testfälle werden mit Hilfe der Testbasis und einer Testüberdeckungsstreategie entworfen. Sie werden ohne konkrete Werte für Ein- und Ausgabeparameter erstellt.

Spezifikation von konkreten Testfällen

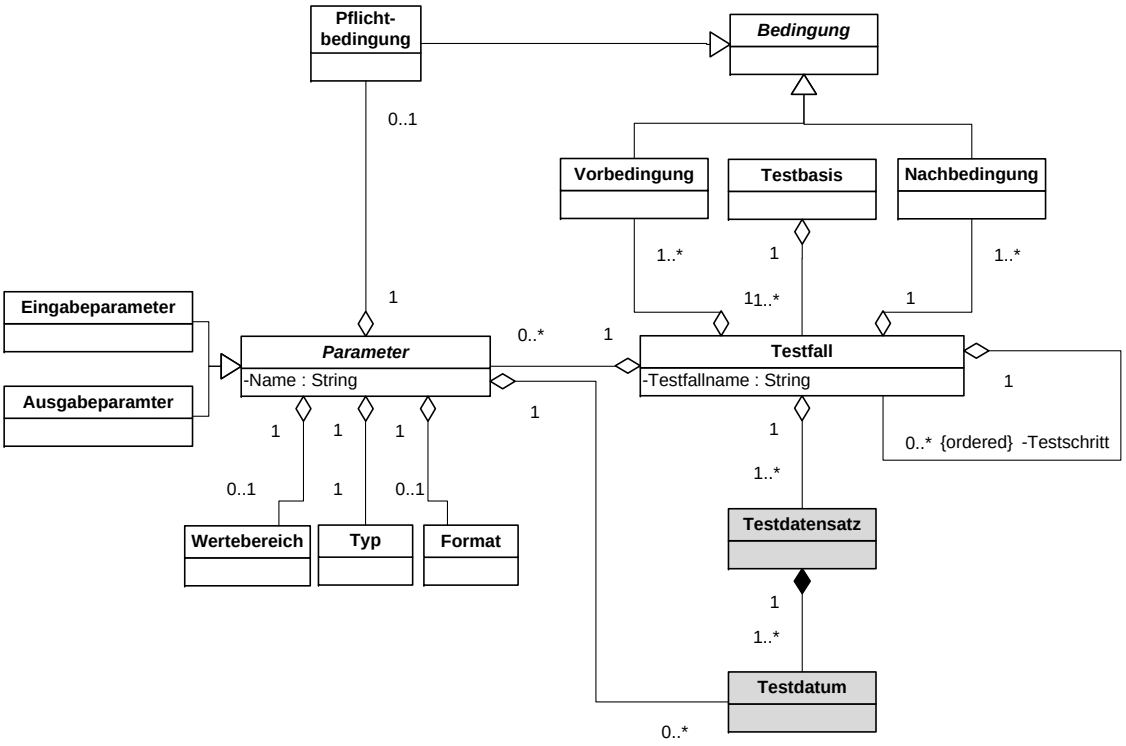


Abbildung 2.13: Metamodell konkreter Testfall

Ein konkreter Testfall wird basierend auf einem logischen Testfall entworfen. Er unterscheidet sich im Aufbau nicht von einem logischen Testfall, ergänzt diesen aber

um Testdatensätze (vgl. Abbildung 2.13). Ein konkreter Testfall besitzt mindestens einen Testdatensatz, welcher wiederum mindestens ein Testdatum besitzt. Das Testdatum beschreibt eine konkrete Ausprägung für einen Ein- oder Ausgabeparameter. Bei der Definition von Testdatensätzen können die vorhandenen Ein- und Ausgabeparameter des logischen Testfalls auf unterschiedliche Art und Weise überdeckt werden. Dies kann zum Beispiel mit Hilfe der Grenzwertanalyse geschehen [SL05]. Diese Analyse untersucht speziell das Verhalten des Testobjektes bei Ein- bzw. Ausgaben an den Grenzen des Definitionsbereichs des jeweiligen Parameters.

2.4 Zusammenfassung

In diesem Kapitel wurden die für diese Arbeit wichtigen Grundlagen mit ihren Begriffen diskutiert. Nach einer Darlegung der Grundlagen zum Thema Software-Produktlinie (vgl. 2.1) erfolgte die Beschreibung des Paradigmas selbst und des dabei genutzten Entwicklungsprozesses mit seinen Rollen. Im Anschluss daran wurde auf das für diese Arbeit zentrale Thema Variabilitätsmanagement eingegangen.
Im Anschluss daran folgte die Erörterung des Themas Anforderungsspezifikation für die Einzelsystem-Entwicklung und dabei speziell die Spezifikation von Anwendungsfällen und Anwendungsfallbeschreibungen (Abschnitt 2.2). Dabei wurde auf Basis der existierenden Literatur ein Metamodell für Anwendungsfallbeschreibungen definiert. Mithilfe dieser Sprache sollen in dieser Arbeit Anforderungen spezifiziert werden. Der sich daran anschließende Abschnitt 2.3 beschrieb Grundlagen der Software-Qualitätssicherung und dabei speziell für das dynamische Testen von Software. Dabei fokussierte die Beschreibung auf die Teststufe Systemtest. Weiterhin wurde ein Metamodell des Testfalls definiert, welches eine Sprache für die Definition von Testfällen innerhalb dieser Arbeit darstellt.
Die somit gelegten Grundlagen sollen im nun folgenden Abschnitt die ausführliche Definition der dieser Arbeit zugrundeliegenden Problemstellung unterstützen und darauf aufbauend den Vergleich mit existierenden Ansätzen erlauben.

Kapitel 3

Problemdefinition und verwandte Arbeiten

In diesem Kapitel werden die bereits in Kapitel 1 der Arbeit adressierten Problemstellungen detailliert. Dazu werden die im letzten Kapitel erläuterten Begriffe genutzt. Aus den Problemstellungen werden weiterhin Anforderungen an ihre Lösung abgeleitet. Diese Anforderungen werden im Anschluss dazu genutzt, existierende Ansätze zur Lösung der Problemstellungen zu evaluieren.

3.1 Problemdefinition

3.1.1 Adressierte Problemstellungen

Das Software-Produktlinienparadigma soll die organisierte Wiederverwendung von gemeinsamen Artefakten für viele Produkte ermöglichen. In diesem Kontext entstehen viele Herausforderungen, die durch spezifische Ansätze adressiert werden müssen.
Diese Arbeit fokussiert dabei auf die organisierte Wiederverwendung von Anforderungsartefakten und Testfällen. Die beiden Typen von Artefakten hängen eng zusammen, da Anforderungsartefakte die Basis für die Spezifikation von Testfällen darstellen.

Die Problemstellungen, die im Kontext der Wiederverwendung von Anforderungsartefakten und Testfällen entstehen, sind in Abbildung 3.1 dargestellt. In der Ab-

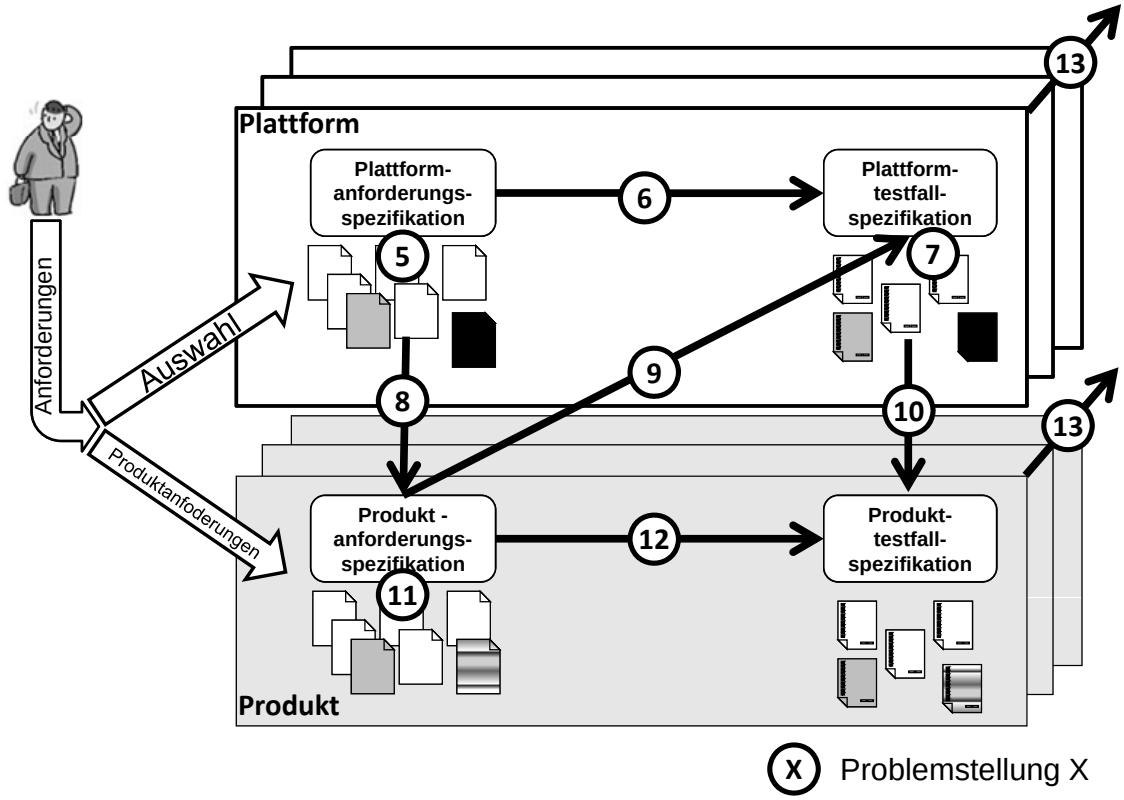


Abbildung 3.1: Problemstellungen des Variabilitätsmanagements in Anforderungs- und Testfallspezifikation für Software-Produktlinien

bildung sind dazu die SPL-Plattform und daraus abgeleitete Produkte dargestellt. Zunächst werden nun diejenigen Problemstellungen näher beschrieben, die in dieser Arbeit behandelt werden. Diese wurden auch in Kapitel 1 bereits grob erläutert. Im Abschnitt 3.1.2 werden dann die übrigen Problemstellungen beschrieben. Die Nummer der jeweiligen Problemstellung entspricht der Nummer der Problemstellung aus der Abbildung.

Die Plattform einer SPL enthält alle Anforderungsartefakte, die allen oder vielen Produkten der SPL gemeinsam sind. Diejenigen, die nicht allen gemeinsam sind, müssen als variable spezifiziert werden, damit ein Auftraggeber diese später für sein Produkt auswählen kann. Für die Modellierung von Variabilität in Anforderungsartefakten müssen daher zwei Teilproblemstellungen gelöst werden (vgl. Problemstellung 5):

Problemstellung 5. *Definition von Modellierungssprachen für Anforderungsartefakte mit Variabilität für Software-Produktlinien und Definition neuer oder Erweiterung existierender Anforderungsspezifikationstechniken für die Behandlung von Variabilität (vgl. Problemstellungen 1 und 4 aus Kapitel 1).*

Zum einen muss die jeweils genutzte Modellierungssprache die Modellierung von Variabilität unterstützten, d. h. die Definition einer solchen Sprache mit Variabilität ist notwendig. Zum anderen müssen die Spezifikationstechniken für die Spezifikation von Anforderungen diese Modellierungssprache in geeigneter Weise einsetzen. Dazu müssen entweder existierende Spezifikationstechniken erweitert oder neue definiert werden.

Die Anforderungsartefakte mit Variabilität bilden nun die Testbasis für die Spezifikation von Testfällen (vgl. Abschnitt 2.3). In diesem Kontext entstehen die Problemstellungen 6 und 7:

Problemstellung 6. *Definition neuer oder Erweiterung existierender Testfallspezifikationstechniken für die Behandlung von Variabilität. (vgl. Problemstellung 4 aus Kapitel 1).*

Problemstellung 7. *Definition von Modellierungssprachen für Testfälle mit Variabilität für Software-Produktlinien (vgl. Problemstellung 2 aus Kapitel 1).*

Zum einen müssen die Spezifikationstechniken für Testfälle, zum Beispiel die Äquivalenzklassen- und Grenzwertanalyse (vgl. Abschnitt 2.3) die in den Anforderungsartefakten vorhandene Variabilität unterstützen. Um nun die Testfälle wiederverwendbar und mit möglichst geringer Redundanz spezifizieren zu können, ist die Modellierung von Variabilität in Testfällen notwendig. Ohne diese Möglichkeit könnte man Testfälle zu jeder möglichen Kombination von Varianten in den Anforderungsartefakten schreiben. Dabei würde ein sehr hohes Maß an Redundanz in Bezug auf die Testfallspezifikation für die gemeinsamen Anforderungsartefakte entstehen. Somit ist die Definition einer Modellierungssprache für Testfälle mit Variabilität notwendig.

Problemstellung 8. *Auswahl der für den Auftraggeber passenden Anforderungsartefakte aus der Plattform.*

Die variablen Anforderungsartefakte erlauben es Auftraggebern die für ihr Produkt passenden Anforderungen auszuwählen. Die dabei entstehenden Abhängigkeiten zwischen variablen Anforderungsartefakten (vgl. Abschnitt 2.1.4) erschweren die Übersicht und damit eine konsistente Auswahl. Die Architektur der SPL und speziell das Vorgehen für die Auswahl von Anforderungen sollten Fehler verhindern (vgl. Problemstellung 8).

Wenn nun ein Auftraggeber die für sein Produkt passenden Anforderungen ausgewählt hat, sollten auch die zu diesen Anforderungen passenden Testfälle auf einfache Art und Weise ausgewählt werden können. In diesem Zusammenhang entstehen die Problemstellungen 9 und 10:

Problemstellung 9. *Abbildung der ausgewählten Anforderungsartefakte auf dazu passende Testfälle (vgl. Problemstellung 3 aus Kapitel 1).*

Problemstellung 10. *Ableitung der zu den ausgewählten Anforderungen passenden Testfälle.*

Zunächst muss eine Abbildung zwischen variablen Anforderungsartefakten und variablen Testfällen ermöglicht werden. Die Abbildung wird während des Testfallspezifikationsprozesses spezifiziert. Sie ermöglicht anschließend die automatische Ableitung der zu den gewählten Anforderungen passenden Testfälle.

Aufgrund des Umfangs des Themengebietes „Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien“ können nicht alle darin existierenden Problemstellungen in dieser Arbeit behandelt werden. Neben den zu behandelnden, die bisher beschriebenen wurden, existieren weitere, die im folgenden kurz erläutert werden.

3.1.2 Problemabgrenzung

Die weiteren Problemstellungen lassen sich gut von den bisher beschriebenen abgrenzen, da sie fast ausschließlich die Produktspezifikation für Software-Produktlinien betreffen. Als weitere, sowohl Plattform- als auch Produktspezifikation betreffende Problemstellung, existiert die Behandlung der Produktlinienevolution.

In Abbildung 3.1 wird auf der linken Seite die Eingabe der Anforderungen des Auftraggebers in den Entwicklungsprozess dargestellt. Neben der Auswahl von Anforderungsartefakten aus der Plattform, können auch weitere Anforderungen existieren, die nicht durch die Plattform abgedeckt werden. Dieser Anforderungen werden *Produktanforderungen* genannt. Im Rahmen der Produktanforderungsspezifikation muss nun das Einfügen dieser Anforderungen durch Erweiterung oder Anpassung der aus der Plattform abgeleiteten Anforderungsartefakte geschehen (vgl. Problemstellung 11).

Problemstellung 11. *Erweiterung und Anpassung der abgeleiteten Anforderungsartefakte um produktspezifische Anforderungen des Auftraggebers.*

Analog zu Problemstellung 11 müssen auch Anpassungen und Erweiterungen in den aus der Plattform abgeleiteten Testfällen durchgeführt werden, um die Produktanforderungen testen zu können (vgl. Problemstellung 12).

Problemstellung 12. *Erweiterung und Anpassung der abgeleiteten Testfälle für den Test der produktspezifischen Anforderungen des Auftraggebers.*

Die Produktlinienevolution [Bos02, McG03] beschäftigt sich mit der Veränderung von Software-Produktlinien mit allen ihren Produkten über die Zeit. Im Vergleich zur Einzelproduktentwicklung ist diese Aufgabe mit großen Herausforderungen verbunden, da die Plattform und die Produkte beidermaßen von Veränderungen betroffen sein können und zudem voneinander abhängen. Wird beispielsweise ein Fehler in der Plattform gefunden, muss entschieden werden, ob die Korrektur auch in alle bereits existierenden Produkte aufgenommen werden kann. Zum Beispiel könnte die Umsetzung einer Produktanforderungen dies erschweren, wenn sie den vom Fehler betroffenen Bereich des Produktes verändert hat (vgl. Problemstellung 13).

Problemstellung 13. *Erweiterung und Anpassung der Plattform- und Produktartefakte über die Zeit (Produktlinienevolution).*

3.1.3 Anforderungen an eine Lösung

Die beschriebenen Problemstellungen können dazu verwendet werden um Anforderungen an ihre Lösung zu definieren. Mit Hilfe dieser Anforderungen können dann zum einen existierende Ansätze evaluiert werden und zum anderen können sie die Definition einer Lösung unterstützen.

Problem-stellung	**Anforde-rung**	**Name**	**Metrik**
5	A1	Modellierung von Variabilität in Anforderungen	⊕, ⓪, ⊖
5	A2	Behandlung von Abhängigkeiten in Anforderungen	⊕, ⓪, ⊖
5	A3	Erweiterung existierender Anforderungsartefakte um Variabilität	ja, nein
6	A4	Äquivalenzklassenanalyse mit Variabilität	ja, nein
6	A5	Grenzwertanalyse mit Variabilität	ja, nein
6	A6	Ablaufanalyse mit Variabilität	ja, nein
7	A7	Modellierung Testschritte mit Variabilität	ja, nein
7	A8	Modellierung Vor- und Nachbedingung mit Variabilität	⊕, ⓪, ⊖
7	A9	Modellierung Ein- und Ausgabeparameter mit Variabilität	ja, nein
7	A10	Modellierung Parametereigenschaften mit Variabilität	⊕, ⓪, ⊖
7	A11	Modellierung Testdaten mit Variabilität	⊕, ⓪, ⊖
8	A12	Unterstützung der Auswahl von Anforderungen aus der Plattform	⊕, ⓪, ⊖
9	A13	Abbildung von variablen Anforderungen auf variable Testfälle	⊕, ⓪, ⊖
10	A14	Automatisierte Ableitung von Testfällen auf Basis ausgewählter Anforderungen	⊕, ⓪, ⊖

Tabelle 3.1: Definierte Anforderungen auf Basis der Problemstellungen

In Tabelle 3.1 sind die aus den Problemstellungen resultierenden Anforderungen dargestellt. Jede Anforderung ist dabei einer Problemstellung zugeordnet, besitzt eine eindeutige Nummer, eine Beschreibung sowie eine Metrik, zur Messung ihrer Erfüllung.
In Anforderung A1 wird die Modellierung von Variabilität in Anforderungsartefakten beschrieben. In dieser Arbeit werden Anforderungen in Form von Anwendungsfallbeschreibungen spezifiziert (vgl. Abschnitt 2.2). Daher wird für die Überprüfung der Anforderung das Metamodell der Anforderungsbeschreibung (vgl. Abbildung 2.8) genutzt. Wird die Modellierung von Variabilität in allen Elementen des Anwendungsfallmetamodells durch einen Ansatz berücksichtigt, wird mit einem ⊕ bewertet. Entsprechend resultiert die teilweise Berücksichtigung in einem ⓪ und keine Berücksichtigung in einem ⊖.
Anforderung A2 fordert die explizite Modellierung von Abhängigkeiten in Bezug auf die Variabilität. Die vollständige Unterstützung aller drei Typen von Abhängigkeiten (Implikation, Beeinflussung und Ausschluss, definiert in Abschnitt 2.1.4) resultiert in einem ⊕. Eine teilweise Unterstützung hat ein ⓪, keine Unterstützung ein ⊖ zur Folge.
Anforderung A3 fordert, dass die Anforderungsspezifikation neben der Modellierung neuer Anforderungsartefakte mit Variabilität auch die Erweiterung existierender Anforderungsartefakte um Variabilität unterstützt. Dies ist zum Beispiel dann wichtig, wenn existierende Produktentwicklungen in einem Unternehmen zu einer Produktlinie zusammengeführt werden sollen. Dabei ist neben der Modellierung selbst auch die Identifikation von Variabilität in den Produkten notwendig. Unterstützt ein Ansatz diese Anforderung wird er mit „ja“ bewertet.
Die Anforderungen A4 bis A6 behandeln klassische Testfallspezifikationstechniken, wie sie in Abschnitt 2.3 eingeführt worden sind. Unterstützt ein Ansatz die Modellierung von Variabilität in der jeweiligen Technik, wird er mit „ja“ bewertet.
Mit den Anforderungen A7 bis A11 wird die Modellierung von Variabilität in Testfällen behandelt. Die Überprüfung dieser Anforderungen wird mit Hilfe des Testfallmetamodells aus Abbildung 2.13 durchgeführt.
Mit A12 wird die Unterstützung der Auswahl von Anforderungen aus der Plattform gefordert. Dabei ist es von Interesse, dass die Auswahl für den Auftraggeber auf einfache Art und Weise ermöglicht wird und die Auswahl konsistent ist.

Die letzten beiden Anforderungen A13 und A14 fordern die Abbildung von variablen Anforderungen auf variable Testfälle während der Testfallspezifikation, sodass eine automatisierte und konsistente Ableitung von Testfällen möglich ist.

3.2 Verwandte Arbeiten

Es existieren verschiedene Ansätze die das Themengebiet „Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien“ adressieren. Die nun vorgestellten Ansätze und ihre Evaluation basieren auf [Wüb08, OWES10]. Es wird dabei auf Ansätze für den Bereich der Anwendungsfallmodellierung und die Teststufe Systemtest fokussiert (vgl. auch Kapitel 2).

3.2.1 Existierende Ansätze

Eine grundlegende Arbeit im Bereich des Testens von Software-Produktlinien ist [McG01, McG02]. Der Ansatz behandelt die Modellierung von Variabilität sowohl in der Anforderungsdefinition als auch in der Systemarchitektur. Er nutzt Anwendungsfälle und dazugehörige Anwendungsfallbeschreibungen, fokussiert dabei aber nur auf die Beschreibung von Abläufen und nicht auf Vor- und Nachbedingungen, wie sie im Anwendungsfallmetamodell aus Abschnitt 2.3 gefordert sind. Die Modellierung von Abhängigkeiten zwischen variablen Bestandteilen eines Anwendungsfalls wird nicht adressiert. Der Ansatz bleibt in Bezug auf Überdeckungskriterien und die Modellierung von Variabilität in Testfällen eher oberflächlich.

Der CADeT Ansatz (Customizable Activity Diagrams, Decision Tables, and Test Specifcations) ist ein modellbasierter Testansatz, der vor allem auf den kombinatorischen Test von Produktlinienplattformen abzielt [Oli08]. Die Modellierung von Variabilität und von Abhängigkeiten in den Anforderungsartefakten wird nicht näher bzw. nicht beschrieben. Der Fokus liegt auf UML-Aktivitätsdiagrammen mit Variabilität, die als Testmodell manuell aus den Anforderungsartefakten hergeleitet werden. Die Testfälle werden auf Basis von Ablaufanalysen über die Aktivitätsdiagramme spezifiziert und berücksichtigen dabei Variabilität. In den Testfällen selbst wird Variabilität in Bezug auf Testschritte und Vor- sowie Nachbedingungen modelliert. Der Ansatz verwendet ein Featuremodell [Maß07] für die Auswahl von Anforderungen und Testfällen aus der Produktlinienplattform.

Der PLUC (Product Line Use Case) und PLUTO (Product Line Use Case Test Optimisation) Ansatz verwendet Anwendungsfallbeschreibungen im Sinne des in Abschnitt 2.2.1 definierten Anwendungsfallmetamodells. Dabei werden Abhängigkeiten zwischen der Variabilität teilweise berücksichtigt. Im Ansatz wird die Kategorie-Partitions-Methode für Konfiguration von zu testenden Produkten genutzt. Der Ansatz modelliert Testfälle nicht explizit, sondern definiert nur, welche Kombinationen an Varianten in den Anforderungen getestet werden sollen. Weiterhin bietet der Ansatz die Möglichkeit auf Basis von Anwendungsfallbeschreibungen eine Auswahl von Anforderungen für Produkte zu tätigen. Die dazu passenden Testfälle werden dann auf Basis dieser Auswahl definiert.

ScenTED (Scenario based TEst case Derivation) ist ein Ansatz basierend auf UML-Aktivitätsdiagrammen als Testmodell [KPRR04]. Die Modellierung der Anforderungen aus denen die Testmodelle abgeleitet werden wird dabei nicht betrachtet. Der Ansatz bietet eine Ablaufanalyse an, die Variabilität berücksichtigt und erstellt damit Sequenzdiagramme als Testfälle. Die Modellierung von Abhängigkeiten und die Abbildung von Anforderungen auf Testfälle ist nicht Teil des Ansatzes.

Der Ansatz aus [NTJ06] fokussiert wiederum auf die Modellierung von Variabilität in Anwendungsfallbeschreibungen und hierbei vor allem auf die formale Modellierung von Vor- und Nachbedingungen mit Variabilität. Er formuliert einen Ansatz für die Ablaufanalyse von Anwendungsfallbeschreibungen für die Definition von Testfällen und modelliert Variabilität in Vor- und Nachbedingungen derselben. Die Modellierung von Abhängigkeiten zwischen Anwendungsfallbeschreibungen und Testfällen, sowie die Ableitung derselben steht nicht im Fokus dieser Arbeit.

[GLRW04] beschreibt die Spezifikation von Testfällen mit Hilfe eines Entscheidungsbaums. Variabilität wird in Testfällen, Testschritten sowie Vor- und Nachbedingungen definiert. Die Auswahl von Testfällen für ein Produkt geschieht mit Hilfe des Entscheidungsbaums.

Der Ansatz aus [HVF05] basiert auf hierarchisch angeordneten UML-Aktivitätsdiagrammen, die um Stereotypen für die Modellierung von Variabilität erweitert worden sind. Die Definition von Testfällen wird mit Hilfe einer Ablaufanalyse über die Aktivitätsdiagramme durchgeführt.

[KLKL07] beschreibt ein holistisches Rahmenwerk für die Produktlinienentwicklung. Dabei werden Anforderungen in Form von Anwendungsfallbeschreibungen spezifiziert. Als Testmodell kommen Sequenzdiagramme, erweitert um Variabilität, zum Einsatz. Mit Hilfe dieses Testmodells werden Testfälle spezifiziert. Der Ansatz unterstützt die Auswahl von Anforderungen und Testfällen aus der Produktlinienplattform mit Hilfe des orthogonalen Variabilitätsmodell [PBL05].

Andere Ansätze

Neben den für die diese Arbeit relevanten Ansätzen existieren weitere, die im Folgenden kurz zusammengefasst werden sollen. In [KNK05] wird ein Ansatz für die Modellüberprüfung (Model Checking) von UML-Zustandsautomaten mit Variabilität präsentiert. [Mis06] definiert einen Ansatz für die Beschreibung von Vorbedingungen in Testfällen auf Basis einer formalen Algebra. Allgemeine Teststrategien für den Modul-, Integrations- und Systemtest werden in [MH03] beschrieben. [RMP07, RMP06] definieren einen Ansatz für den Integrations- und Performancetest auf Basis des ScenTED Ansatzes. In [Sch07, OSW08] werden Ansätze für den kombinatorischen Test von Produktlinienplattformen präsentiert. Der Ansatz aus [UGKB07] benutzt eine Spezifikation gegeben als logische Ausdrücke erster Ordnung. Der Ansatz adressiert damit die Generierung von Eingabedaten für Testfälle. In [WSS08] wird ein modellbasierter Testansatz auf Basis von UML-Zustandsautomaten präsentiert.

3.2.2 Evaluationsergebnisse

Die Ergebnisse der Evaluation sind in Tabelle 3.2 zusammengefasst. Sie zeigt, dass keiner der Ansätze alle gestellten Anforderungen an „Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für SPL“ erfüllt. Die Anforderungen A3, A4, A5, A9, A10 und A11 wurden bisher durch keinen Ansatz adäquat adressiert. Es zeigt sich, dass die Defizite vor allem in den Bereichen Modellierung von Variabilität

Nr.	**McGregor**	**CADeT**	**PLUC/ PLUTO**	**ScenTED**	**Nebuts Ansatz**	**Gepperts Ansatz**	**Hartmanns Ansatz**	**Kangs Ansatz**
A1	◎	◎	⊕	⊖	◎	⊖	◎	◎
A2	⊖	◎	◎	⊖	⊖	⊖	⊖	◎
A3	nein	nein	nein	nein	nein	nein	nein	nein
A4	nein	nein	nein	nein	nein	nein	nein	nein
A5	nein	nein	nein	nein	nein	nein	nein	nein
A6	nein	ja	nein	ja	ja	ja	ja	ja
A7	ja	ja	nein	ja	nein	ja	nein	ja
A8	⊖	◎	⊖	⊖	⊕	⊖	⊖	⊖
A9	nein	nein	nein	nein	nein	nein	nein	nein
A10	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
A11	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
A12	◎	⊕	◎	⊖	⊖	⊕	◎	⊕
A13	⊖	◎	◎	⊖	⊖	⊖	⊖	⊕
A14	⊖	⊕	⊕	⊖	⊖	⊕	⊖	⊕

Tabelle 3.2: Evaluation existierender Ansätze mit Hilfe der definierten Anforderungen

in Testfällen liegen. Weiterhin existiert bisher kein Ansatz der die Erweiterung von existierenden Anforderungsartefakten um Variabilität betrachtet. Die Äquivalenz- und Grenzwertanalyse im Rahmen der Testfallspezifikation mit Variabilität stand bisher auch nicht im Fokus.

3.3 Zusammenfassung

Zentraler Gegenstand der Diskussion in diesem Kapitel waren zunächst die für das Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien relevanten Problemstellungen (vgl. Abschnitt 3.1.1). Aufgrund der Größe des adressierten Themengebietes wurden einige Problemstellungen für diese Arbeit ausgewählt und die übrigen abgegrenzt.
Im Anschluss daran erfolgte die Definition von Anforderungen zur Lösung der ausgewählten Problemstellungen (vgl. Abschnitt 3.1.3). Diese Anforderungen bildeten weiterhin die Grundlage für die Evaluation existierender Ansätze. Diese wurden in Abschnitt 3.2 zunächst beschrieben und anschließend mit Hilfe der Anforderungen verglichen.
Die Bewertung zeigte, dass keiner der existierenden Ansätze die Anforderungen in seiner Gesamtheit erfüllen konnte. Daher wird in den nun folgenden Kapiteln dieser Arbeit ein eigener Ansatz für das Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für Software-Produktlinien eingeführt. Begonnen wird dabei mit der formalen Definition von Variabilitätsmanagement als zentralem Paradigma für den eigenen Ansatz. In diesem Zusammenhang werden auch existierende Ansätze speziell für das Variabilitätsmanagement betrachtet.

Kapitel 4

Variabilitätsmanagement

Variabilität und ihr Management bildet das zentrale Paradigma für die geplante Wiederverwendung von Modellen einer Produktlinienplattform [CN01, PBL05, Sof]. Daher werden in diesem Kapitel die Anforderungen an Variabilitätsmodellierung und ihr Management erläutert und formal definiert. Diese Anforderungen bilden das Fundament für den im Anschluss beschrieben Beitrag dieser Arbeit.

4.1 Methodisches Vorgehen

Um Variabilitätsmanagement für Software-Produktlinien zu ermöglichen, müssen die Anforderungen daran verstanden und definiert werden. In dieser Arbeit wird für die formale Definition dieser Anforderungen schrittweise ein *Metamodell des Variabilitätsmanagememts* erstellt. Für die Definition des Metamodells wird das UML-Metamodell als Sprache genutzt [OMG09]. Das UML-Metamodell ist hinsichtlich seiner Ausdrucksmächtigkeit eingeschränkt: Wenn zum Beispiel eine Klasse eine bestimmte Bedingung erfüllen muss, damit eine Assoziation zu einer anderen Klasse existieren darf, ist diese Bedingung mit den Mitteln des UML-Metamodells nicht definierbar. Für die Definition solcher *Bedingungen* an Klassen des Metamodells wird daher die *Object Constraint Language* (OCL, [OMG03]) und ihr Metamodell genutzt [RG99].
Die Formalisierung der Anforderungen ist in Abbildung 4.1 dargestellt. Horizontal sind in der Abbildung die Modellebenen dargestellt. Jede Modellebene definiert die Sprache für Modelle der darunter liegenden Ebene [Béz01].
Die Anforderungen an ein Variabilitätsmanagement sind eine informelle Eingabe

in das Vorgehen und werden im Rahmen dieser Arbeit zu einem Metamodell und OCL-Bedingungen formalisiert. Sie basieren zum einen auf der existierenden Literatur [Bos00, CN01, PBL05] und zum anderen auf Erfahrungen aus Projekten mit dem Industriepartner arvato services [MW08, MW09, AEMW09]. Mithilfe dieses Metamodells wird dann die Erfüllung der Anforderungen durch existierende Ansätze für Variabilitätsmanagement evaluiert.

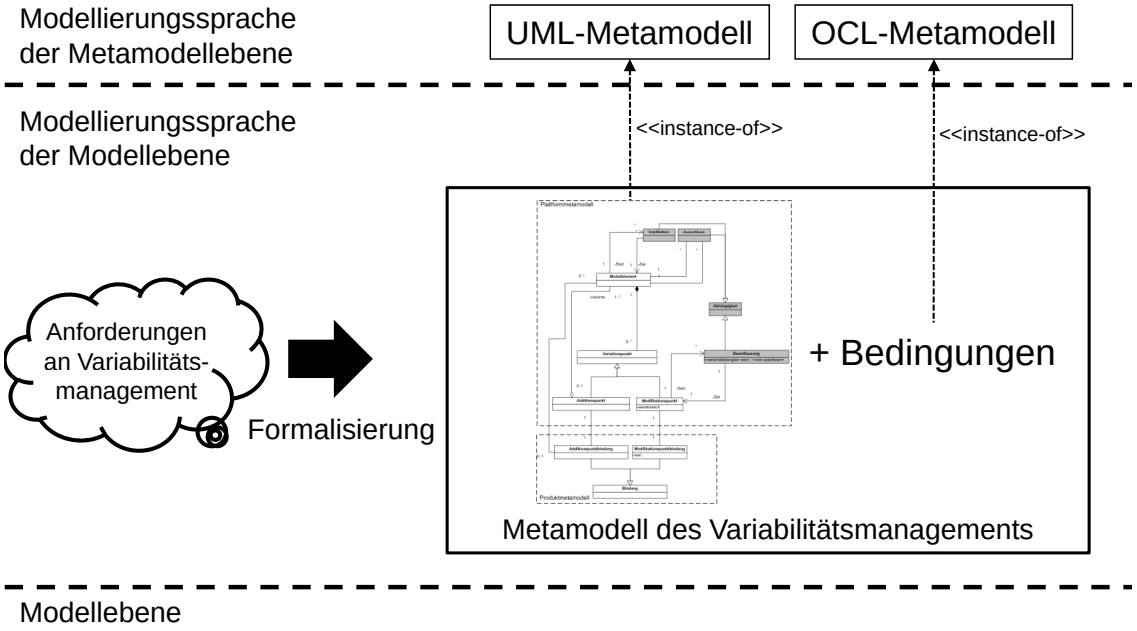


Abbildung 4.1: Formalisierung der Anforderungen zum Metamodell des Variabilitätsmanagements

4.2 Variabilitätsmanagement in Software-Produktlinien

Das Variabilitätsmanagement befasst sich mit der Verwaltung von Variabilität in Modellen über den gesamten Software-Produktlinienentwicklungsprozess (vgl. Abschnitt 2.1.1) hinweg.

Dabei soll das Variabilitätsmanagement zunächst die Modellierung von Variabilität in Modellen ermöglichen. Diese Anforderung wird im folgenden Abschnitt näher beschrieben.

4.2.1 Modellierung von Variabilität

Variabilität beschreibt die Veränderlichkeit einer SPL mit Hilfe von Modellelementen (vgl. Abschnitt 2.1.4). Ein Variationspunkt definiert dabei die Art und den Ort der Veränderlichkeit in einem Modellelement.

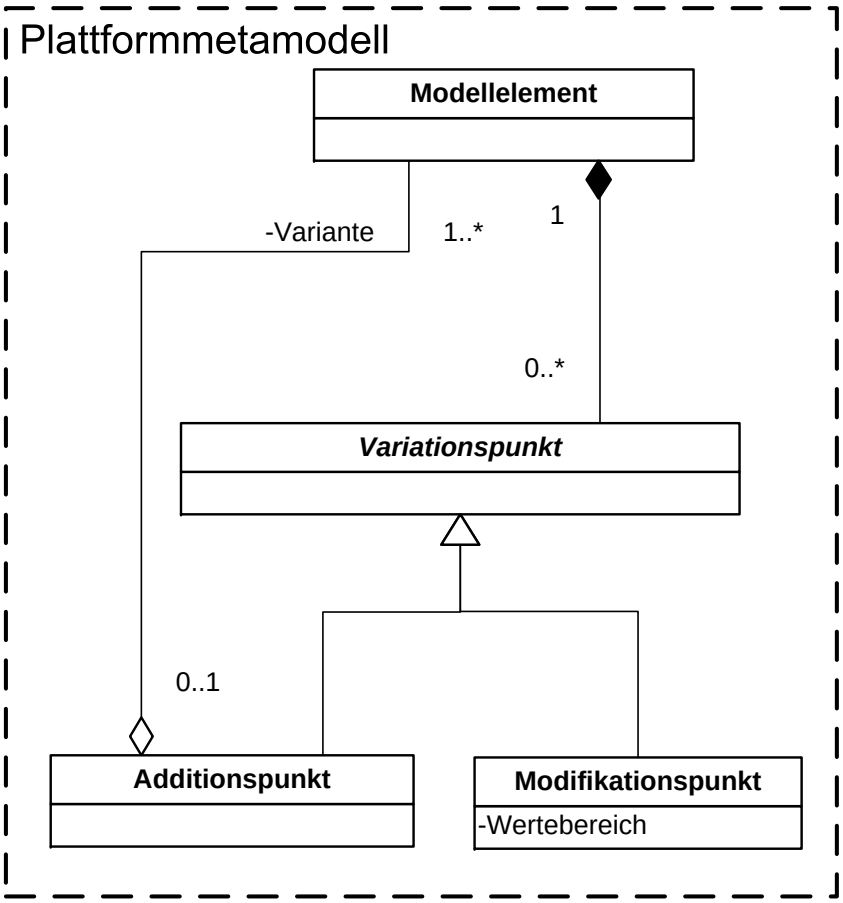


Abbildung 4.2: Metamodell der Variabilität, basierend auf [Maß07]

Abbildung 4.2 beschreibt diesen Zusammenhang in Form eines Metamodells der Variabilität basierend auf [Maß07]. Das Metamodell bildet das Paket *Plattformmetamodell*. Ein *Modellelement* kann demnach beliebig viele *Variationspunkte* besitzen.

Dabei werden *Additions-* und *Modifikationspunkte* unterschieden. Um den Gegenstand der Veränderlichkeit zu beschreiben, wird an einen Additionspunkt wiederum mindestens ein Modellelement assoziiert. Somit ermöglicht ein Additionspunkt das Hinzufügen oder Weglassen eines Modellelements. Ein Modifikationspunkt besitzt ein Attribut *Wertebereich*, welches die Art der Modifikation spezifiziert. Der Modifikationspunkt beschreibt demnach die Veränderlichkeit auf Basis der Auswahl eines bestimmten Wertes aus einem zuvor definierten Wertebereichs. Der Modifikationspunkt selbst ist Bestandteil eines Modellelements. Ein an einen Additionspunkt assoziiertes Modellelement wird als *Variante* bezeichnet. Jede Variante kann wiederum Variationspunkte besitzen. Dadurch entsteht eine hierarchische Dekomposition von Modellelementen, Variationspunkten und Varianten bzw. Modifikationspunkten.
Eine Anforderung an das Variabilitätsmanagement ist nun die Modellierung von Variabilität in Modellelementen. Um dies zu realisieren werden *Variabilitätsmodellierungssprachen* benötigt, die es erlauben, (variable) Modellelemente und Additions- sowie Modifikationspunkte zu modellieren.

4.2.2 Bindung von Variabilität

Die Modellierung von Variabilität in Modellelementen ist die Voraussetzung für die Auswahl von Modellelementen für Produkte einer SPL. Die Aus- und Abwahl von Modellelementen an einem Additionspunkt sowie die Auswahl von Werten an Modifikationspunkten wird als Auflösen der Variabilität oder *Bindung* bezeichnet [Maß07]. Die Bindung ist im Metamodell in Abbildung 4.3 integriert. Sie bildet ein eigenes Paket *Produktmetamodell*, welches durch das Paket Plattformmetamodell importiert wird (vgl. Abbildung 4.4).

Zwei Typen von Bindungen werden dabei unterschieden: Eine Additionspunktbindung berücksichtigt einen Additionspunkt und eine Menge von Modellelementen. Die Bindung trifft eine *Auswahl* von Modellelementen aus der Menge der an den Additionspunkt assoziierten Varianten. Diese Einschränkung wird mit Hilfe der OCL an die Additionspunktbindung definiert:

Bedingung 1. ***Context*** *Additionspunktbindung* ***inv:***
self.Additionspunkt.Variante→includesAll(self.Modellelement)

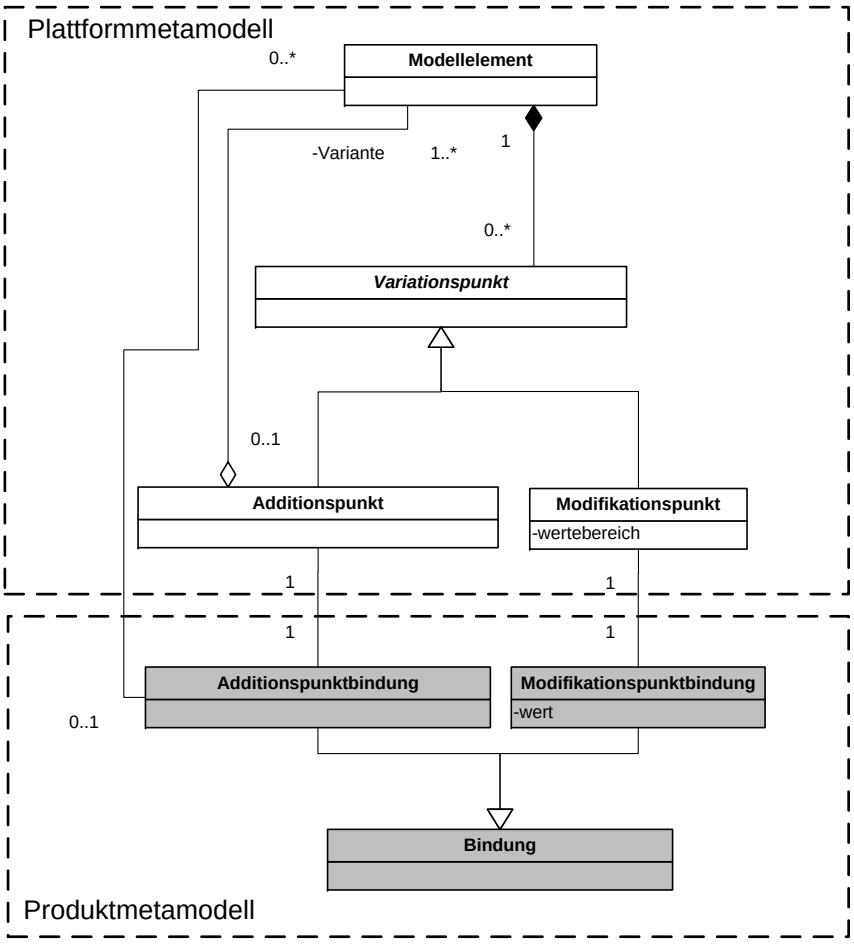


Abbildung 4.3: Metamodell des Variabilitätsmanagements mit Bindung, modifiziert von [Maß07]

Die Modifikationspunktbindung berücksichtigt einen Modifikationspunkt und einen *Wert* aus dem Wertebereich des Modifikationspunktes. Diese Einschränkung wird analog zur Einschränkung an Additionspunkten mit Hilfe der OCL an der Modifikationspunktbindung formuliert:

Bedingung 2. ***Context*** *Modifikationspunktbindung* ***inv:***
self.Modifikationspunkt.Wertebereich→includesAll(self.Wert)

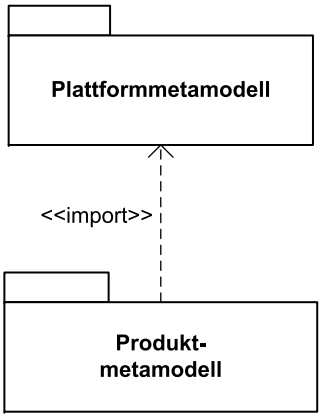


Abbildung 4.4: Paketabhängigkeiten zwischen Plattform- und Produktmetamodell

Die Bindung aller Variationspunkte und der daran vorhandenen Varianten und Werte wird als *Ableitung* eines Produktes aus der Produktlinienplattform bezeichnet (vgl. Abschnitt 2.1.4). Die hieraus resultierende Anforderung an ein Variabilitätsmanagement ist die Unterstützung der Bindung von Variabilität und damit der Ableitung von Produkten aus der Produktlinienplattform.

4.2.3 Abhängigkeit

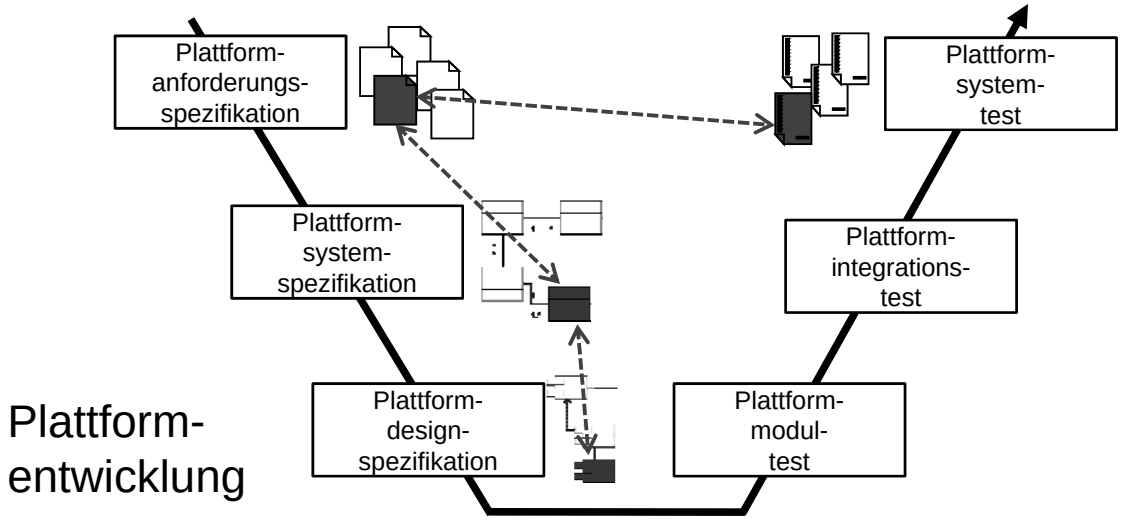


Abbildung 4.5: Abhängigkeiten der Variabilität im V-Modell

Bei der Modellierung von Modellelementen in einer SPL kann es zu Abhängigkeiten zwischen diesen kommen. Diese Abhängigkeiten können sowohl fachlich als auch technisch motiviert sein. Erstere entstehen, wenn zum Beispiel zwei Modellelemente zwei Funktionalitäten beschreiben, die niemals gemeinsam in einem Produkt der SPL existieren dürfen. Letztere können zum Beispiel eine Abhängigkeit zur Laufzeitumgebung des Produktes bedeutet. Je nach Umgebung müssen andere Modellelemente gewählt werden, damit das Produkt auf der existierenden Hardware lauffähig ist.
Abbildung 4.5 illustriert diesen Sachverhalt beispielhaft an Modellen in der Plattformentwicklung. Im Beispiel der Abbildung existieren Abhängigkeiten zwischen Modellelementen der Plattformanforderungsspezifikation und anderen Modellelementen im Entwicklungsprozess (grau hervorgehoben). Ein Modellelement aus der Plattformanforderungsspezifikation wird zum Beispiel durch ein Modellelement in der Plattformsystemspezifikation verfeinert und später in der Plattformtestfallspezifikation berücksichtigt. Diese Abhängigkeit hat im Kontext dieser Arbeit folgende Bedeutung: Falls ein Modellelement bei der Ableitung eines Produktes gebunden wird, wirkt sich die Bindung dieses Modellelements auch auf die Bindung der abhängigen Modellelemente aus. Die Abhängigkeit zwischen zwei Modellelementen wird im Metamodell in Abbildung 4.6 definiert.

Beliebig viele Modellelemente bzw. Modifikationspunkte können voneinander abhängig sein. Es werden drei Typen von Abhängigkeiten unterschieden: Die unidirektionalen Abhängigkeiten *Implikation* und *Beeinflussung* und die bidirektionale Abhängigkeit *Ausschluss*. Die drei Typen haben unterschiedliche Auswirkungen auf die abhängigen Modellelemente bzw. Modifikationspunkte:
Eine Implikation zwischen zwei Modellelementen bedeutet, dass die Bindung eines Modellelements pc_1 an eine Additionspunktbindung die Bindung eines abhängigen Modellelements pc_2 herbeiführt, aber nicht umgekehrt (unidirektional). Das abhängige Modellelement wird an die Additionspunktbindung des Additionspunktes gebunden, an dem das abhängige Modellelement assoziiert ist.
Die Implikation wird als Bedingung mit Hilfe der OCL in der Metaklasse Additionspunktbindung formuliert:

Bedingung 3. ***Context*** *Additionspunktbindung* ***inv:***
*self.Modellelement→**forAll**(m:Modellelement|m.Implikation*
*→**forAll**(i:Implikation|i.Ziel.Additionspunktbindung→notEmpty()))*

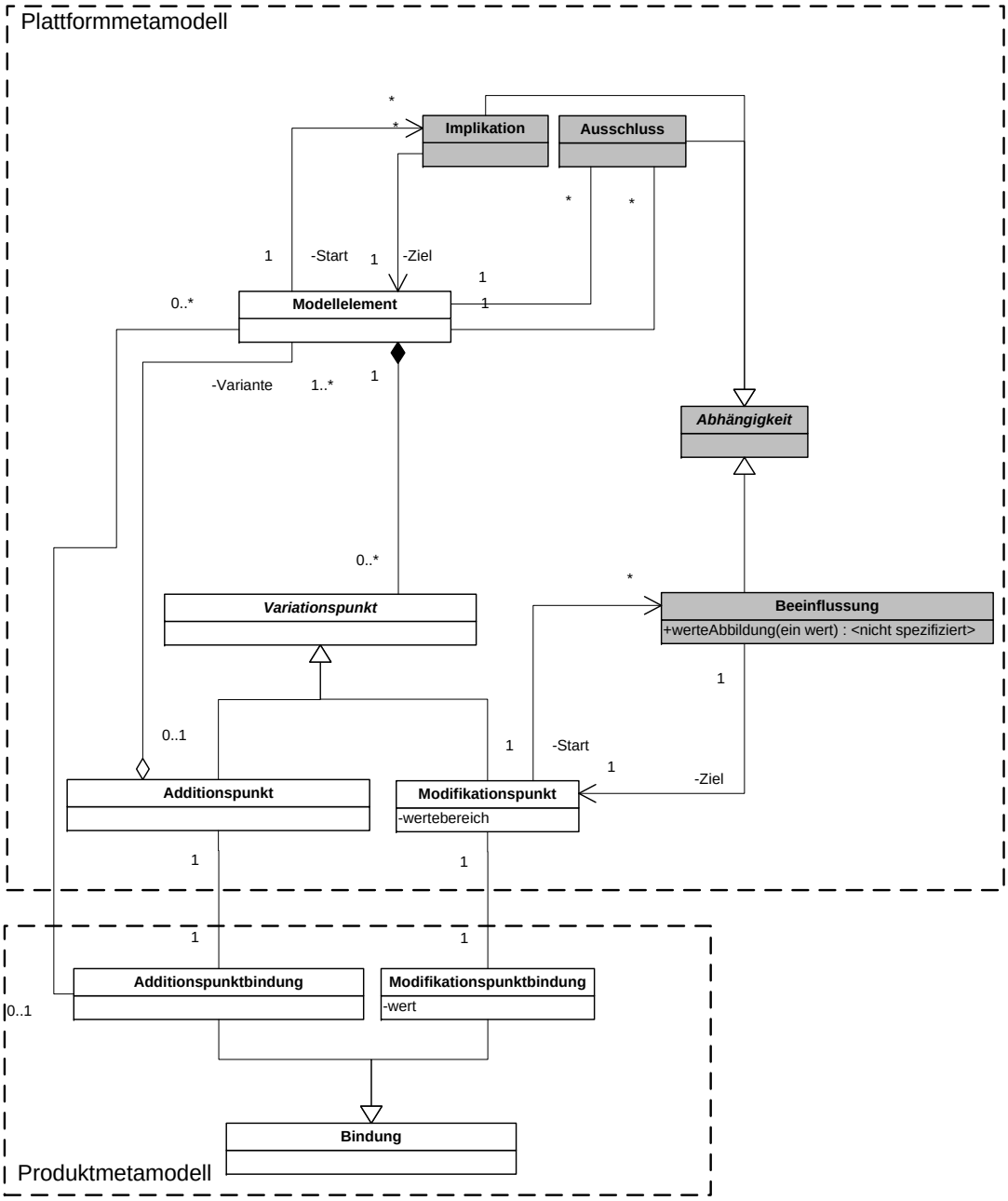


Abbildung 4.6: Metamodell des Variabilitätsmanagements mit Bindung und Abhängigkeiten, modifiziert von [Maß07]

Die Beeinflussung bedeutet, dass die Bindung eines Wertes an eine Modifikationspunktbindung eines Modifikationspunktes mp_1 die Bindung eines Wertes an eine Modifikationspunktbindung des abhängigen Modifikationspunktes mp_2 in definierter

Art und Weise ausprägt. Die Art und Weise der Ausprägung ist in der Operation *werteAbbildung* der Metaklasse definiert. Die Operation erhält als Eingabe den Wert aus der Modifikationspunktbindung von *Start* und liefert als Rückgabewert den Wert für die Modifikationspunktbindung von *Ziel*.

Bedingung 4. ***Context*** *Modifikationspunktbindung* ***inv:***
*self.Modifikationspunkt.Beeinflussung→**forAll**(b:Beeinflussung|*
b.Ziel.Modifikationspunktbindung.Wert = b::werteFunktion(self.Wert)))

Der Ausschluss beschreibt, dass die Bindung von Modellelement pc_1 an eine Additionspunktbindung die Bindung von Modellelement pc_2 an eine Additionspunktbindung verhindert und umgekehrt (bidirektional):

Bedingung 5. ***Context*** *Additionspunktbindung* ***inv:***
*self.Modellelement→**forAll**(me:Modellelement|me.Ausschluss|*
*→**forAll**(a:Ausschluss|*
a.Modellelement.Additionspunktbindung→isEmpty()))

Das Variabilitätsmanagement sollte die drei Typen von Abhängigkeiten unterstützen.

4.2.4 Konsistenz

Unter Konsistenz wird die fehlerfreie Ableitung aller möglichen Produkte aus der Plattform verstanden. Fehlerfrei bedeutet, dass es möglich ist, alle modellierten Abhängigkeiten während der Ableitung eines Produktes einzuhalten. Die Menge der möglichen Produkte enthält alle Produkte, die aus der Kombination aller Modellelemente und variablen Modellelemente abgeleitet werden können. Konsistenz wird somit in dieser Arbeit folgendermaßen definiert:

Definition 10. *Ein Variabilitätsmanagement ist konsistent, falls die Bindung jedes Variationspunktes ohne Verletzung von Abhängigkeiten möglich ist.*

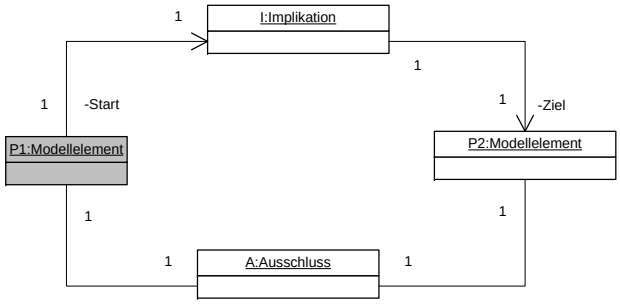


Abbildung 4.7: Beispiel für eine Verletzung mit Implikation und Ausschluss

Es werden dabei zwei mögliche Typen von Verletzungen betrachtet, die durch Nutzer des Variabilitätsmanagements modelliert werden können und dadurch die Bindung von Variationspunkten unmöglich machen.

In Abbildung 4.7 ist eine Verletzung durch die gleichzeitige Modellierung einer Implikation und eines Ausschlusses dargestellt. In diesem Beispiel kann $P1$ niemals Teil eines Produktes sein.
Um diesen Typ der Verletzung auszuschließen, wird eine Bedingung an das Modellelement definiert:

Bedingung 6. ***Context*** *Modellelement* ***inv:***
*self.Implikation→**forAll**(i:Implikation|i.Ziel*
*→**forAll**(z:Ziel|z.Ausschluss*
*→**forAll**(a:Ausschluss|*
a.Modellelement →excludes(self))))

Die zweite Art der Verletzung liegt vor, falls sich zwei hierarchisch dekomponierte Modellelemente ausschließen (vgl. Abbildung 4.8). In diesem Beispiel kann $P2$ niemals Teil eines Produktes werden.
Diese Art der Verletzung wird über folgende Bedingung ausgeschlossen:

Bedingung 7. ***Context*** *Modellelement* ***inv:***
*self.Ausschluss→**forAll**(a:Ausschluss|a.Modellelement*
*→**forAll**(me:Modellelement|me.Additionspunkt*
*→**forAll**(a:Additionspunkt|a.Variante→excludes(self))))*

Das Variabilitätsmanagement sollte die Erhaltung der Konsistenz für eine SPL unterstützen und Inkonsistenzen aufzeigen.

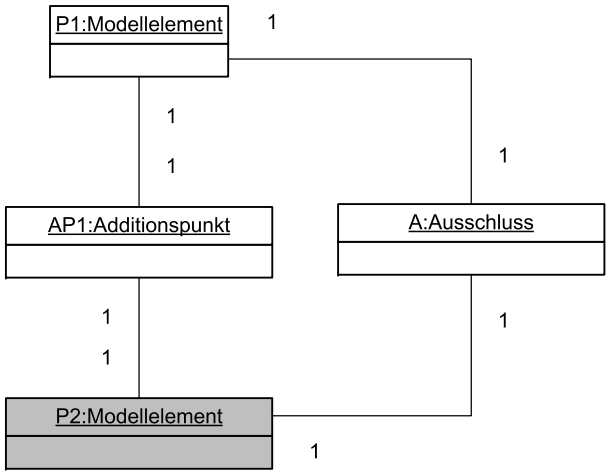


Abbildung 4.8: Beispiel für eine Verletzung mit Additionspunkt und Ausschluss

4.2.5 Weitere Anforderungen an das Variabilitätsmanagement

Neben den bisher beschriebenen Anforderungen an Variabilitätsmanagement existieren in der Literatur weitere Anforderungen, die in dieser Arbeit nicht im Fokus stehen. Nach [BBM05] sollte ein Ansatz für Variabilitätsmanagement folgende weitere Anforderungen erfüllen:
Zum einen existiert die Anforderung hinsichtlich der Verfolgbarkeit [GAd98], welche die Modellierung von Relationen zwischen Variationspunkten und Varianten einer SPL zum Gegenstand hat. Verfolgbarkeit wird in dieser Arbeit durch die explizite Modellierung von Assoziationen und Abhängigkeiten zwischen Modellelementen und Additions- sowie Modifikationspunkten realisiert.
Eine weitere Anforderung ist die Visualisierung von Variabilität in einer SPL. Damit ist die explizite grafische Darstellung der hierarchischen Dekomposition der Variabilität (vgl. Abschnitt 4.2.1) und der Abhängigkeiten zwischen Varianten gemeint. Eine Visualisierung kann den Benutzer des Variabilitätsmanagements bei der Spezifikation von Variabilität in Modellen und der Modellierung der Abhängigkeiten unterstützen und somit Fehler vermeiden lassen.
Darüber hinaus beschreiben [Bos02] und [McG03] die Anforderung der Unterstützung der Produktlinienevolution. Unter Evolution wird in diesem Kontext die Veränderung der Produktlinienplattform und ihrer Produkte über die Zeit verstanden.

Dazu gehören zum Beispiel die Erweiterung der Plattform um neue Anforderungen oder die Extraktion von Funktionalitäten aus Produkten und deren Überführung in die Produktlinienplattform. Diese Aufgabe kann durch die in dieser Arbeit vorgestellten Anforderungen wie explizite Modellierung von Variabilität und Abhängigkeiten zwischen Varianten unterstützt, aber nicht hinreichend gelöst werden.

4.3 Evaluation existierender Ansätze

Die durch das Metamodell des Variabilitätsmanagements formalisierten Anforderungen werden nun für die Evaluation existierenden Ansätze genutzt. Dazu werden die Anforderungen zunächst konsolidiert und Metriken für deren spätere Bewertung erstellt.

4.3.1 Anforderungen

Die im letzten Abschnitt vorgestellten Anforderungen für das Variabilitätsmanagement werden in Tabelle 4.1 dargestellt. Jede Anforderung erhält dabei eine eindeutige Nummer, einen Namen und eine Kurzbeschreibung. In der letzten Spalte der Tabelle ist dann die jeweilige Metrik für die Bewertung der Anforderung angegeben.

Ein ⊖ in einer Metrik bedeutet, dass diese Anforderung nicht erfüllt wird. Ein ⓪ steht für die nicht vollständige und ein ⊕ für eine vollständige Erfüllung der jeweiligen Anforderung. Eine nicht vollständige Erfüllung bedeutet, dass Teile der Anforderung erfüllt werden, aber auch Teile unerfüllt bleiben, bzw. nicht hinreichend festgestellt werden kann, ob ein Ansatz die Anforderung vollständig erfüllt. Eine vollständige Erfüllung betrifft jeden Teil einer Anforderung. Die Metrik für die Anforderungen A1 und A2 lassen nur die Erfüllung (ja) oder Nichterfüllung (nein) der jeweiligen Anforderung zu. Hierbei ist eine teilweise Erfüllung nicht möglich, da die Anforderungen atomar sind. Die Anforderung A4 bietet eine Auswahlmenge der unterstützten Abhängigkeitstypen an. Mit Hilfe der konsolidierten Anforderungen und der dazu spezifizierten Metriken werden nun im nächsten Abschnitt existierende Ansätze für das Variabilitätsmanagement evaluiert.

Nr.	Name	Kurzbeschreibung	Metrik
A1	Additionspunktmodellierung	Dedizierte Modellierungssprache für Additionspunkte	ja, nein
A2	Modifikationspunktmodellierung	Dedizierte Modellierungssprache für Modifikationspunkte	ja, nein
A3	Hierarchische Dekomposition	Hierarchische Dekompositon von Variationspunkten und Varianten	$\oplus$, ⓪, ⊖
A4	Abhängigkeitstypen	Berücksichtigung von Implikation, Ausschluss und Beeinflussung	$Auswahl = \{Implikation(I), Ausschluss(A), Beeinflussung(B)\}$
A5	Bindung	Bindung von Variationspunkten und Ableitung von Produkten	$\oplus$, ⓪, ⊖
A6	Konsistenz	Erhaltung und Überprüfung der Konsistenz	$\oplus$, ⓪, ⊖

Tabelle 4.1: Anforderungen an ein Variabilitätsmanagement

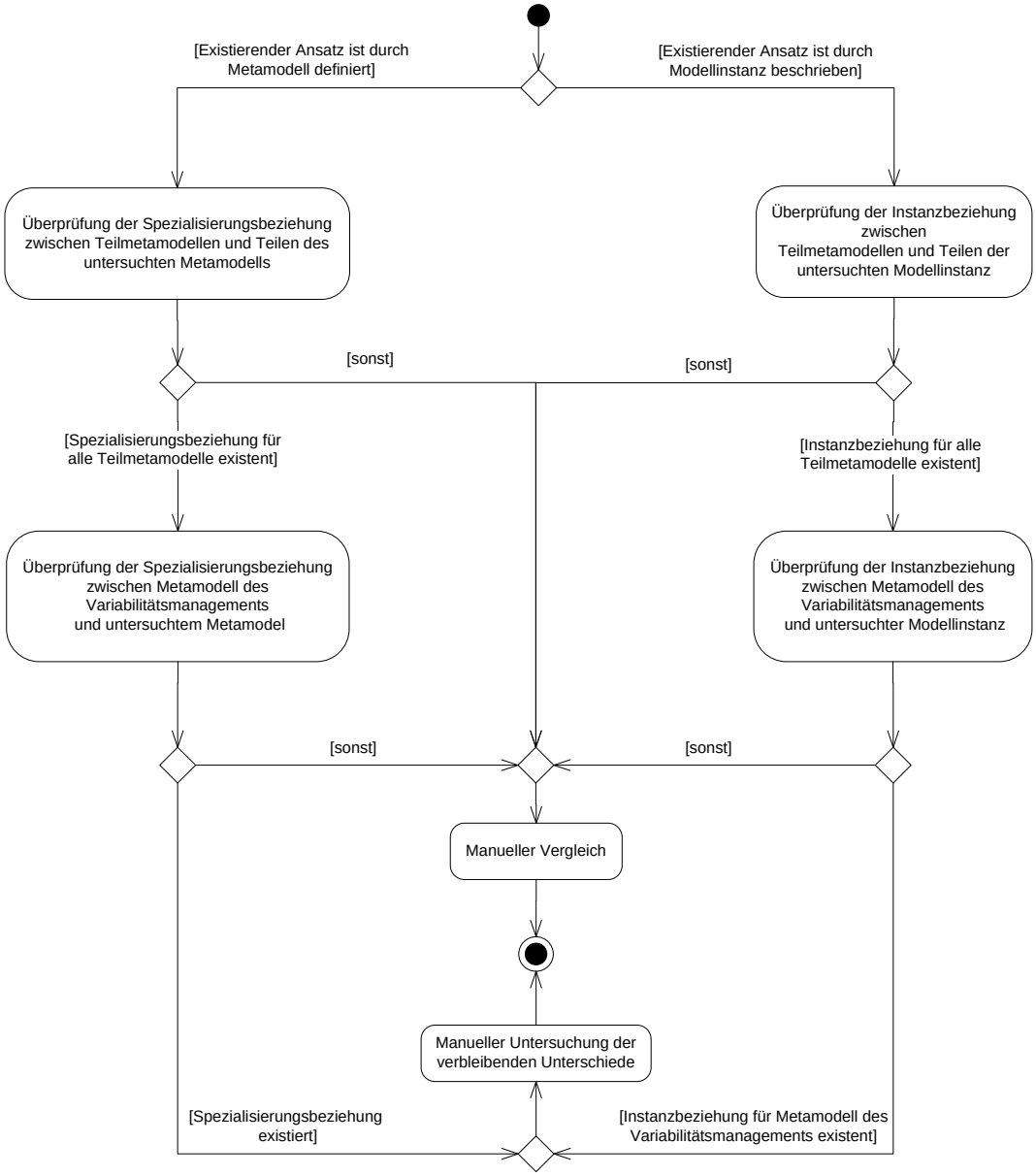


Abbildung 4.9: Vorgehen für die Überprüfung von Metamodellen auf Spezialisierungsbeziehungen

4.3.2 Evaluationsvorgehen

Die Festlegung der Werte für die einzelnen Metriken während der Evaluation folgt einem definierten Vorgehen. Dabei werden vier Szenarien unterschieden, die in Abbildung 4.9 als UML-Aktivitätsdiagramm dargestellt sind. Im Folgenden werden die vier Szenarien genauer beschrieben.

Szenario 1: Existierender Ansatz ist durch ein Metamodell definiert und erfüllt die Anforderungen teilweise

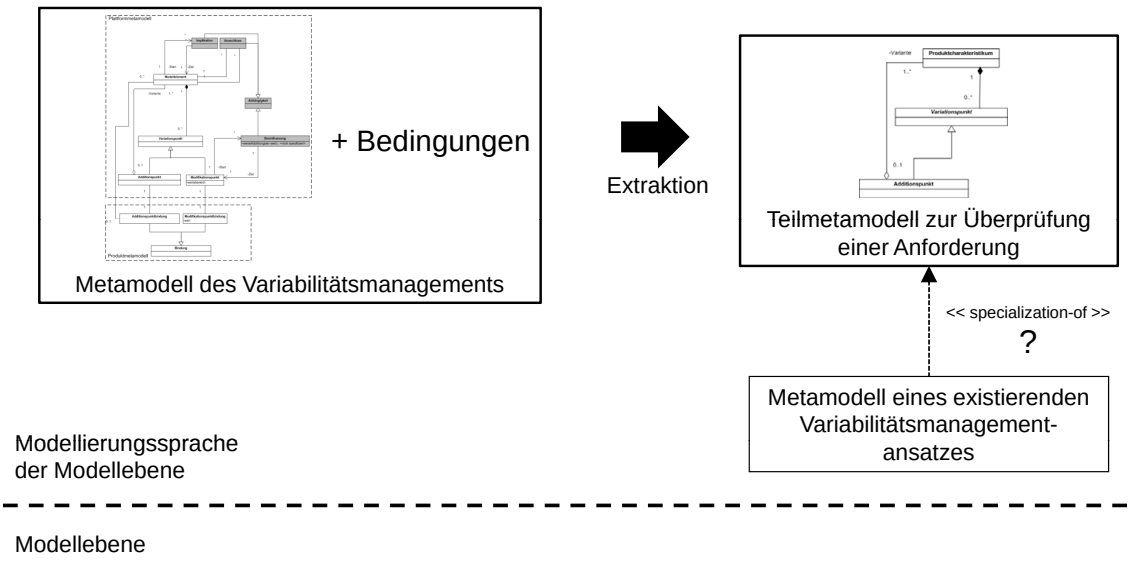


Abbildung 4.10: Evaluation existierender Ansätze auf Basis eines Metamodells hinsichtlich der Erfüllung von Teilen der Anforderungen

Existierende Ansätze werden auf Basis ihres Metamodells evaluiert, falls ein solches existiert. Dies ist sinnvoll, da das Metamodell die Sprache für alle Modellinstanzen definiert und sich daher auf der selben Modellebene befinden, wie das Metamodell des Variabilitätsmanagements.
Zunächst wird jede in Tabelle 4.1 definierte Anforderung als ein Teilmetamodell aus dem Metamodell des Variabilitätsmanagements extrahiert (vgl. Abbildung 4.10). Dies soll an dieser Stelle beispielhaft für die Anforderung der hierarchischen Dekomposition geschehen:

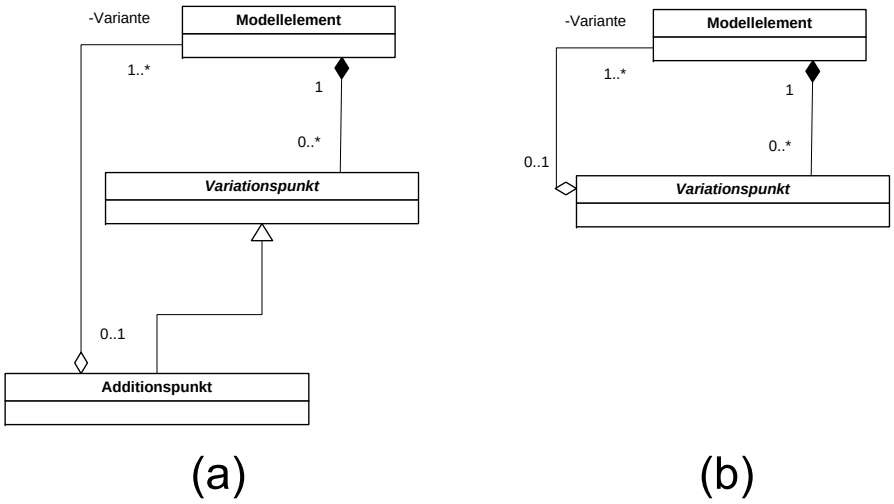


Abbildung 4.11: Teilmetamodell für die hierarchische Dekomposition

Das Teilmetamodell, dargestellt in Abbildung 4.11.a, ist ein Ausschnitt des Metamodells des Variabilitätsmanagement aus Abbildung 4.6. Das Teilmetamodell definiert die hierarchische Dekomposition von Varianten und Variationspunkten. In Abbildung 4.11.b ist das Teilmetamodell um die Vererbung des Additionspunktes abstrahiert worden. Dies ist sinnvoll, falls ein untersuchter Ansatz nicht die explizite Modellierung von Additions- und Modifikationspunkten unterscheidet. Beide Versionen des Teilmetamodells werden nun für die Evaluation eingesetzt.
Mit Hilfe dieses Teilmetamodells wird überprüft, ob ein Teil des Metamodells des existierenden Ansatzes eine Spezialisierung des Teilmetamodells darstellt. In [EE95] wird eine solche Spezialisierungsbeziehung für Zustandsdiagramme definiert. Im Kontext dieser Arbeit wird eine Spezialisierung daher in folgender Weise definiert:

Definition 11. *Spezialisierung (specialization-of): Ein Metamodell ist eine Spezialisierung eines anderen Metamodells, falls beide strukturell identisch sind (Isomorphismus) und ein ontologischer Abgleich zwischen den Metaklassen beider Modelle existiert.*

Der ontologische Abgleich (engl. *ontological alignment*) [ES04, KS05] existiert genau dann zwischen Metamodellklassen, wenn deren Bedeutung identisch ist. In dieser Arbeit wird die Überprüfung der Spezialisierung zwischen Metamodellen manuell durchgeführt.

Alle Teilmetamodelle die nicht durch den existierenden Ansatz spezialisiert werden, werden im Anschluss noch einmal manuell überprüft. Sind alle Anforderungen erfüllt, wird mit Szenario 2 fortgefahren und das gesamte Metamodell auf eine Spezialisierungsbeziehung zum Metamodell des Variabilitätsmanagements untersucht. Sind nicht alle Anforderungen erfüllt, werden die noch verbleibenden Unterschiede im existierenden Metamodell untersucht. Ziel ist es dabei Eigenschaften zu exponieren, die im Metamodell des Variabilitätsmanagements unberücksichtigt geblieben sind.

Szenario 2: Existierender Ansatz ist durch ein Metamodell definiert und erfüllt die Anforderungen vollständig

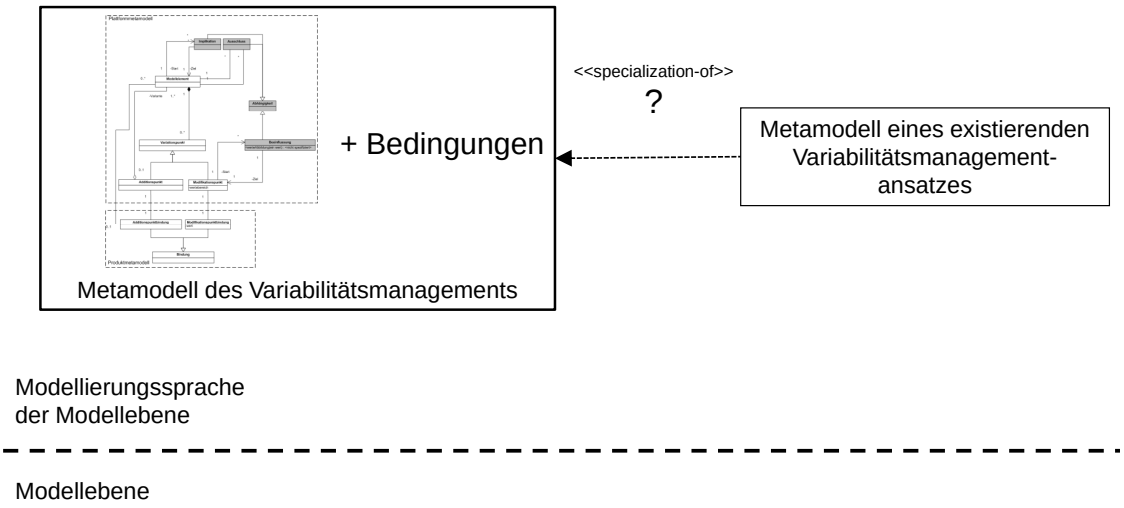


Abbildung 4.12: Evaluation existierender Ansätze auf Basis eines Metamodells hinsichtlich der Erfüllung aller Anforderungen

Sind alle Anforderungen, definiert durch Teilmetamodelle, erfüllt, wird die Überprüfung auf das gesamte Metamodell des Variabilitätsmanagements erweitert (vgl. Abbildung 4.12). Es wird überprüft, ob das untersuchte Metamodell eine Spezialisierung des Metamodells des Variabilitätsmanagements ist. Ist dies nicht der Fall, werden die Unterschiede manuell überprüft. Ist das untersuchte Metamodell eine Spezialisierung, wird überprüft, ob der untersuchte Ansatz Eigenschaften besitzt, die im Metamodell des Variabilitätsmanagements unberücksichtigt geblieben sind.

3. Existierender Ansatz wird auf Basis einer Modellinstanz untersucht und erfüllt die Anforderungen teilweise

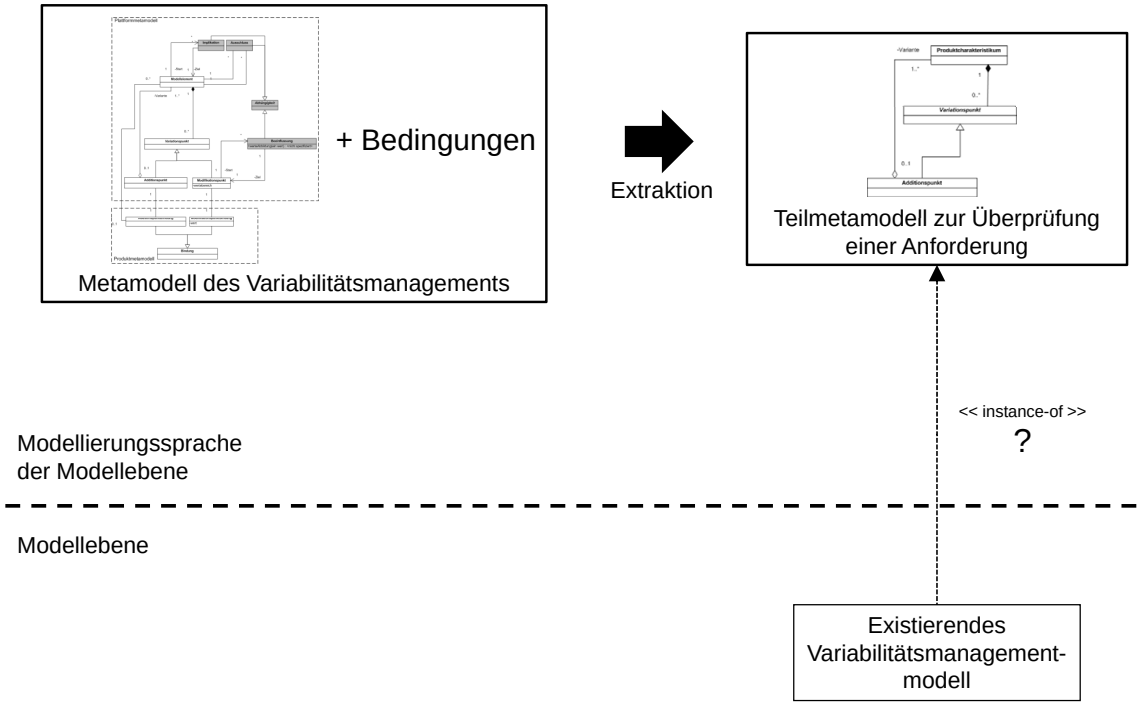


Abbildung 4.13: Evaluation existierender Ansätze auf Basis von Modellinstanzen hinsichtlich der Erfüllung von Teilen der Anforderungen

Falls kein Metamodell für einen existierenden Ansatz verfügbar ist, kann die Überprüfung auch anhand von Modellinstanzen geschehen. Hierbei kann überprüft werden, ob Teile von Modellinstanzen eines existierenden Ansatzes Instanzen von Teilmetamodellen sind (vgl. Abbildung 4.13). Alle Teilmetamodelle die nicht durch den existierenden Ansatz instanziiert werden, werden im Anschluss noch einmal manuell überprüft. Sind alle Anforderungen erfüllt, wird mit Szenario 4 fortgefahren und die Modellinstanzen werden gesamtheitlich auf eine Instanzbeziehung zum Metamodell des Variabilitätsmanagements untersucht. Sind nicht alle Anforderungen erfüllt, werden die noch verbleibenden Unterschiede in den Modellinstanzen untersucht. Ziel ist es, dabei Eigenschaften zu exponieren, die im Metamodell des Variabilitätsmanagements unberücksichtigt geblieben sind.

Szenario 4: Existierender Ansatz wird auf Basis einer Modellinstanz untersucht und erfüllt die Anforderungen vollständig

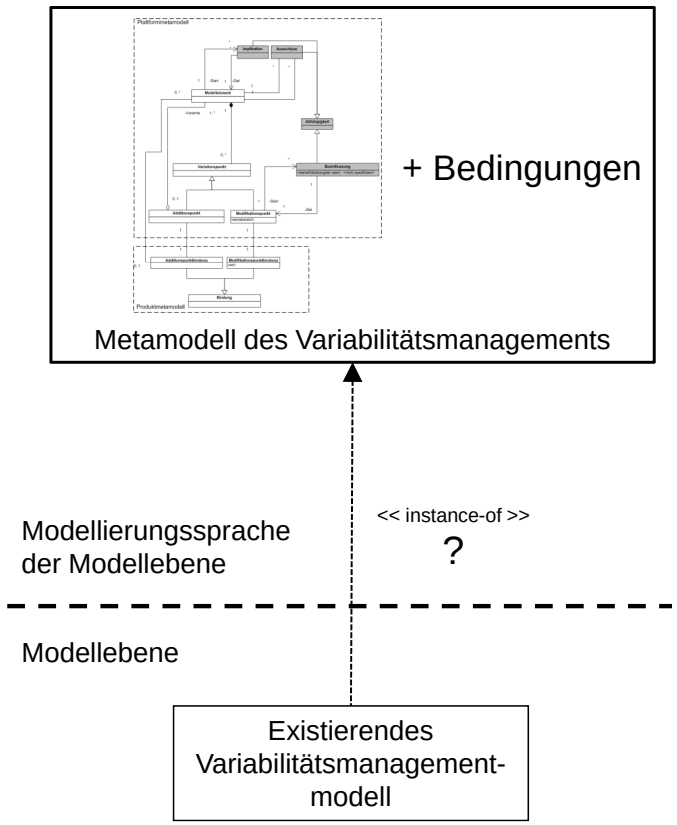


Abbildung 4.14: Evaluation existierender Ansätze auf Basis von Modellinstanzen hinsichtlich der Erfüllung aller Anforderungen

Sind alle Anforderungen, die durch Teilmetamodelle definiert werden, erfüllt, wird die Überprüfung auf das gesamte Metamodell des Variabilitätsmanagement erweitert (vgl. Abbildung 4.14). Dabei wird überprüft, ob die untersuchten Modellinstanzen Instanzen des Metamodells des Variabilitätsmanagements darstellen. Ist dies nicht der Fall, werden die Unterschiede manuell überprüft. Anschließend wird überprüft, ob der untersuchte Ansatz Eigenschaften besitzt, die im Metamodell des Variabilitätsmanagements unberücksichtigt geblieben sind. Sind die Modellinstanzen Instanzen des Metamodells des Variabilitätsmanagements, wird nur der letzte Schritt durchgeführt.

4.3.3 Evaluationsergebnisse

Im Rahmen der Evaluation von Ansätzen zum Variabilitätsmanagement werden Ansätze aus verschiedenen Bereichen der Software-Produktlinienentwicklung berücksichtigt. Dabei existieren drei Kategorien von Ansätzen:

- Ansätze aus der Entwicklungsphase Plattformanforderungsspezifikation
- Ansätze aus der Entwicklungsphase Plattformtestfallspezifikation
- Ansätze für ein Variabilitätsmanagement, die den gesamten Software-Produktlinienentwicklungsprozess adressieren

Für die Plattformanforderungsspezifikation werden die Ansätze von John [JM02], Bertolino [BG03b] und das Featuremodell (FM) [Maß07] evaluiert. In die zweite Kategorie fallen CADeT [OG05, GO08, Oli08], PLUTO [BG03b, BG03a, BFGL04, BFGL06], der Ansatz von Nebut [NPLJ02, NPLJ03, NFLJ03, NTJ06] und ScenTED [KPRR03, RRKP03, KPRR04, RKPR05b, RKPR05a, RMP06, RMP07]. Die gesamtheitlichen Ansätze werden durch das orthogonale Variabilitätsmodell (OVM) [PBL05] und die Ansätze von Schmid [SJ03] und Schnieders [SP07] vertreten.
Es existieren weitere Ansätze wie zum Beispiel [McG01], [Bac01], [KT03], [GLRW04], [KNK05], [EBB05], [Mis06], [KLKL07] und [JKB08]. Diese Ansätze stellen Teilmengen oder Grundlagen der für die Evaluation ausgewählten Ansätze dar und werden deshalb nicht explizit in die Bewertung aufgenommen.

Für die Evaluation werden weiterhin folgende Annahmen getroffen:

- Grundlage ist die zur Verfügung stehende Literatur. Dabei wird eine korrekte Interpretation der dargestellten Ansätze unterstellt.
- Für die Evaluation auf der Basis von Modellinstanzen wird die korrekte Interpretation dieser und die korrekte Interpretation der Sprachmächtigkeit unterstellt. Ist die Mächtigkeit der Sprache in Bezug auf eine Anforderung unklar, wird dies in der Evaluation vermerkt.

Die Untersuchung der existierenden Ansätze hat teilweise im Rahmen von studentischen Arbeiten [Emk09, Obe09b, Mba09] und wissenschaftlichen Beiträgen [Wüb08, OWES10] stattgefunden. In Tabelle 4.2 sind die Ergebnisse der Evaluation zusammengefasst. Im Folgenden wird für jeden betrachteten Ansatz eine kurze

Nr.	John	Bertolino	FM	CADeT	PLUTO	Nebut	ScenTED	OVM	Schmid	Schnieders
A1	ja	ja	ja	ja	ja	ja	ja	ja	ja	ja
A2	nein	ja	ja	nein	ja	nein	nein	nein	ja	nein
A3	⊕	⓪	⊕	⊖	⓪	⓪	⓪	⊖	⊖	⊖
A4	-	B	I, A, B	I, A	I	-	-	I, A	I, A, B	I, A
A5	⊖	⊖	⓪	⓪	⓪	⓪	⓪	⓪	⓪	⓪
A6	⊖	⊖	⓪	⊖	⊖	⊖	⊖	⊖	⊖	⊖

Tabelle 4.2: Bewertung existierender Ansätze zum Variabilitätsmanagement hinsichtlich der Erfüllung der gestellten Anforderungen

Zusammenfassung der Evaluation gegeben. Die jeweils adressierten Anforderungen stehen in Klammern. Weitere Details finden sich auch in den referenzierten Arbeiten.

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator
<variant> by controlling the distance to cars in front </variant>
Actors: driver, gas regulator
Trigger: actor driver, <variant> actor distance regulator </variant>
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

1.) <keep velocity> is selected by actor driver
2.) Does a distance regulator exist?
get vactual, vtarget (<Calculate Velocity>)
<variant OPT> get dactual, dtarget (<Calculate Distance>) </variant>
3.) Does a distance regulator exist?
<variant ALT 1: no; only cruise control>
- compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget : gas regulator decrease velocity
- restart <keep velocity>
else restart <keep velocity>
</variant>
<variant ALT 2: yes, cruise control + distance regulator>
- - compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget and atarget £ aactual: gas regulator decrease velocity
- restart <keep velocity>
If vactual < vtarget and atarget > aactual: gas regulator increase velocity
- restart <keep velocity>
else restart<keep velocity>
</variant>

Abbildung 4.15: Beispiel Use Case Beschreibung [JM02]

John Der Ansatz von John wird durch Szenario 3 des Vorgehens evaluiert, da er kein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.
Der Ansatz basiert auf einer XML-ähnlichen Variabilitätsmodellierungssprache, um Use Cases und ihre Beschreibungen mit Variabilität zu modellieren (vgl. Abbildung 4.15). Das Beispiel in Abbildung 4.15 ist eine Instanz des Teilmetamodells für die Überprüfung der hierarchischen Dekomposition. Variationspunkte werden als Fragen dargestellt mit davon abhängigen Varianten, modelliert durch XML-ähnliche Tags (A3). Die Modellierung von Implikationen, Ausschlüssen oder Beeinflussungen sind im Ansatz und auch im Beispiel aus Abbildung 4.15 nicht modelliert (A4).

Bertolino Der *Product Line Use Case* Ansatz wird durch Szenario 3 des Vorgehens evaluiert, da er kein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt sind.
Im Ansatz liegt der Fokus auf der Modellierung von Anforderungen in Form von Anwendungsfallbeschreibungen. In Abbildung 4.16 ist eine solche Anwendungsfallbeschreibung dargestellt. Der Ansatz bietet dabei eine Variabilitätsmodellierungssprache mit Additionspunkten (*Alternative*, *Optional*) sowie Modifikationspunkten (*Parametric Tag*) und Varianten (A1 und A2). Das Beispiel in Abbildung 4.15 ist eine Instanz des Teilmetamodells für die Überprüfung der hierarchischen Dekomposition (A3), da in Varianten, die an Variationspunkten assoziiert sind, wiederum Variationspunkte existieren. Durch die Modellinstanz wird aber nicht deutlich, ob beliebig viele Dekompositionen möglich sind. Somit ist nicht entscheidbar, ob eine hierarchische Dekomposition vollständig möglich ist. Der Ansatz unterstützt die Modellierung von Beeinflussungen für manche Typen von Variabilität durch die Modellierung einer Bedingung auf Basis des *Parametric Tag* (A4).

Featuremodell (FM) Das Featuremodell wird durch Szenario 1 des Vorgehens evaluiert, da es ein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.
Das Metamodell des Featuremodells ist in Abbildung 4.17 dargestellt. Es unterstützt die Modellierung von Additionspunkten und Modifikationspunkten (A1 und A2). Erstere ist durch die Domänenbeziehung realisiert. Das Teilmetamodell der hierarchischen Dekomposition wird durch einen Teil des untersuchten Metamodells spezialisiert. Die Metaklassen Feature und Domänenbeziehung, sowie ihre Assoziationen

PL USE CASE GamePlay

Goal: Play a game on a **[GP0]** Mobile Phone and record score
Scope: The **[GP0]** Mobile Phone
Level: Summary
Precondition: The **[GP0]** Mobile Phone is on
Trigger: Function GAMES has been selected from the main menu
Primary actor: The Mobile Phone user
Secondary actors: The {**[GP0]** Mobile Phone} (the system)
The Mobile Phone Company
Main success scenario
1. The system displays the list of the {**[GP1]** available} games
2. The user selects a game
3. The user selects the difficulty level
4. The user starts the game and plays it until completion
5. The user records the score achieved {and **[GP2]** sends the score to Club XXX via WAP}

Extensions
1a. No game is available:
1a1. return to main menu
3a. The user starts the game and plays it until an incoming call arrives. **See CallAnswer**.

Variations
GP0: Alternative:
0. Model 0
1. Model 1
2. Model 2
GP1: Parametric
if **GP0**=0 then display msg “No game available”

else if **GP0**=1 then Snake II or Space Impact

else if **GP0**=2 then Snake II or Space Impact or Bumper.

GP2: Optional
when **GP0**=2

Abbildung 4.16: Product Line Use Case [BFGL06]

entsprechen strukturell dem Teilmodell und können ontologisch mit den Klassen Modellelement und Variationspunkt abgeglichen werden (A3). Ein Modifikationspunkt wird über ein Attribut eines Features modelliert.

CADeT Der *CADeT* Ansatz wird durch Szenario 1 des Vorgehens evaluiert, da er ein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt. Abbildung 4.18 zeigt den CADeT Ansatz in der Übersicht. Der Ansatz nutzt bei der Modellierung der Variabilität nur Additionspunkte mit Varianten der Typen *man-*

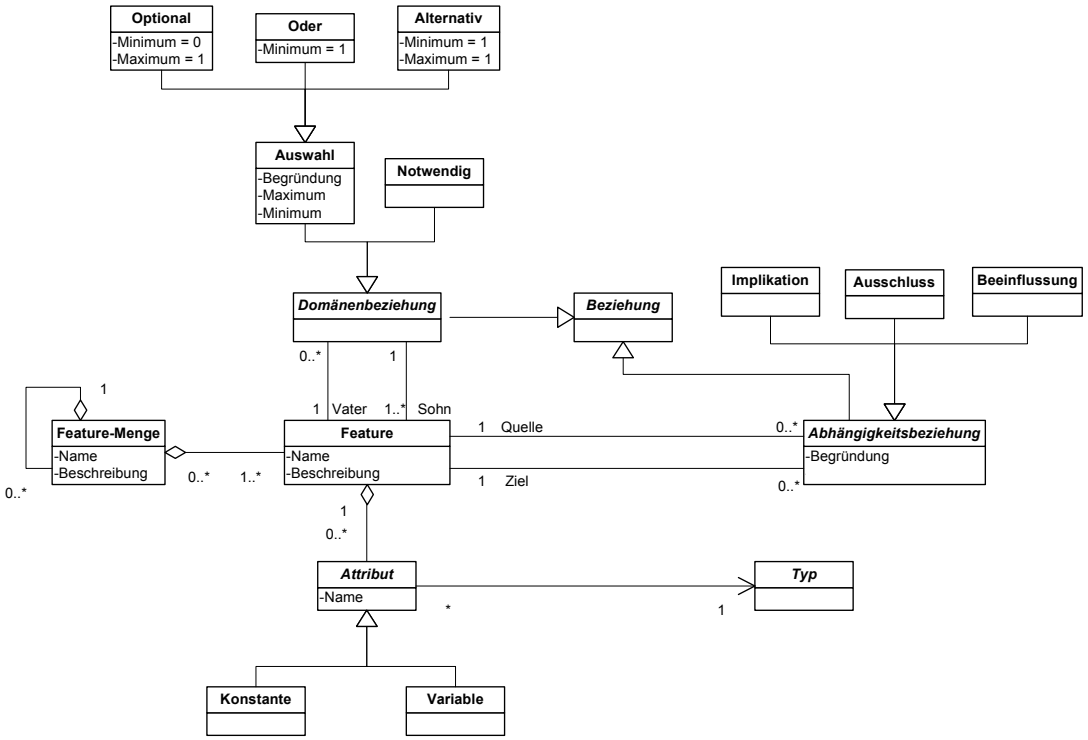


Abbildung 4.17: Metamodell des Featuremodells [Maß07]

datory und *optional* (A1). Das Teilmetamodell der hierarchischen Dekomposition wird im Metamodell des Ansatzes nicht spezialisiert: Die Klasse *Variationspunkt* ist sowohl im Teilmetamodell als auch im Metamodell vorhanden. Im Metamodell hat die Klasse eine Assoziation zur Klasse *«CADeT model» Test specification*, die der Assoziation aus dem Teilmetamodell entspricht. Die zweite Assoziation aus dem Teilmetamodell der hierarchischen Dekomposition kann aber weder direkt, noch über andere Klassen des Metamodells gefunden werden. Der Ansatz bietet daher keine hierarchische Dekomposition von Variationspunkten und Varianten (A3). CADeT benutzt ein Featuremodell und Entscheidungstabellen für die Modellierung von Abhängigkeiten und unterscheidet dabei Implikationen und Ausschlüsse, aber keine Beeinflussung (A4).

PLUTO Der PLUTO Ansatz basiert auf dem Product Line Use Case Ansatz und leitet Testfälle auf dieser Basis ab. Die Evaluationsergebnisse entsprechen daher bis auf die Bindung denen des Product Line Use Case Ansatz. Die Bindung ist im

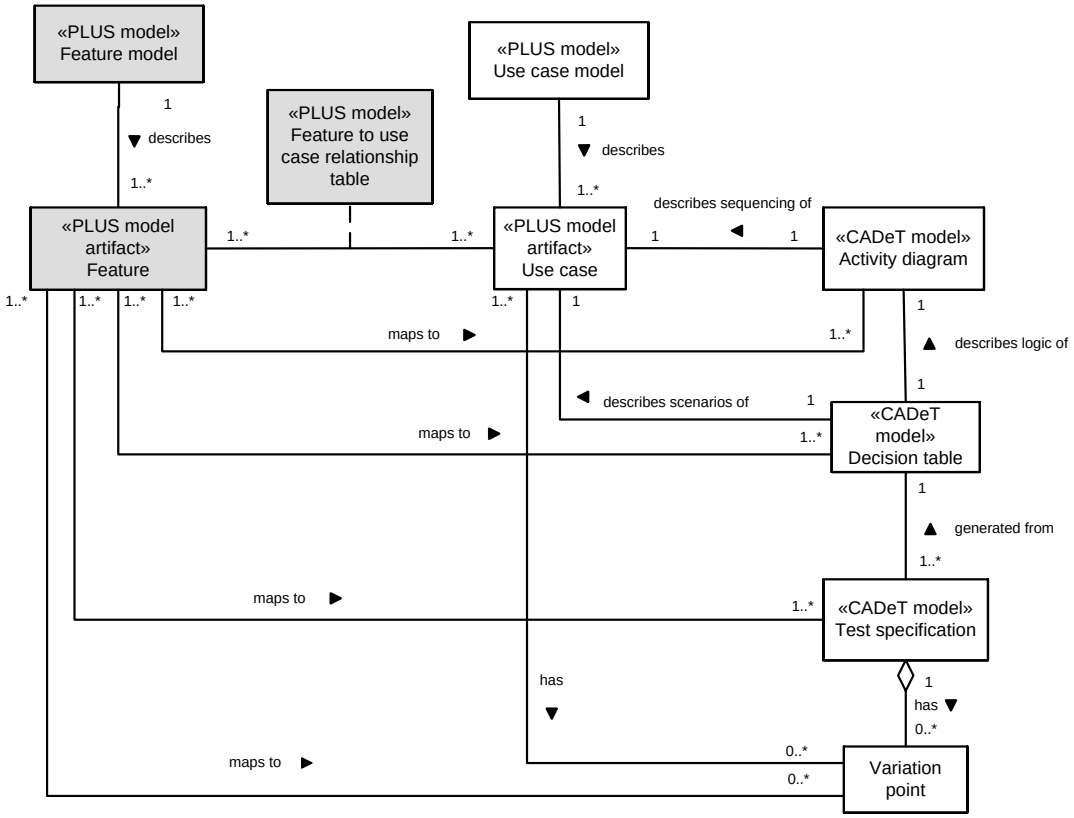


Abbildung 4.18: Metamodell CADeT Ansatz [Oli08]

PLUTO Ansatz für die Generierung von Testfällen beschrieben (A5).

Nebut Der Ansatz von Nebut et al. wird durch Szenario 3 des Vorgehens evaluiert, da er kein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.
Die gegebenen Modellinstanzen sind eine Instanz des Teilmetamodells der hierarchischen Dekomposition. Die Beispiele lassen aber keinen Schluss darauf zu, ob eine hierarchische Dekomposition von Variabilität mit dieser Sprache möglich ist (A3). Dies liegt darin begründet, dass nicht klar wird, ob beliebig viele Dekompositionen möglich sind.

ScenTED Der ScenTED-Ansatz wird durch Szenario 3 des Vorgehens evaluiert, da er kein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.

Der Ansatz fokussiert die Modellierung von Variabilität in Aktivitätendiagrammen. Das Teilmetamodell der hierarchischen Dekomposition wird in den Beispielen des Ansatzes instanziiert. Es ist aber nicht ersichtlich, ob eine hierarchische Dekomposition mit dieser Sprache möglich ist (A3). Dies liegt wiederum darin begründet, dass nicht ersichtlich ist, ob beliebig viele Dekomposition möglich sind.

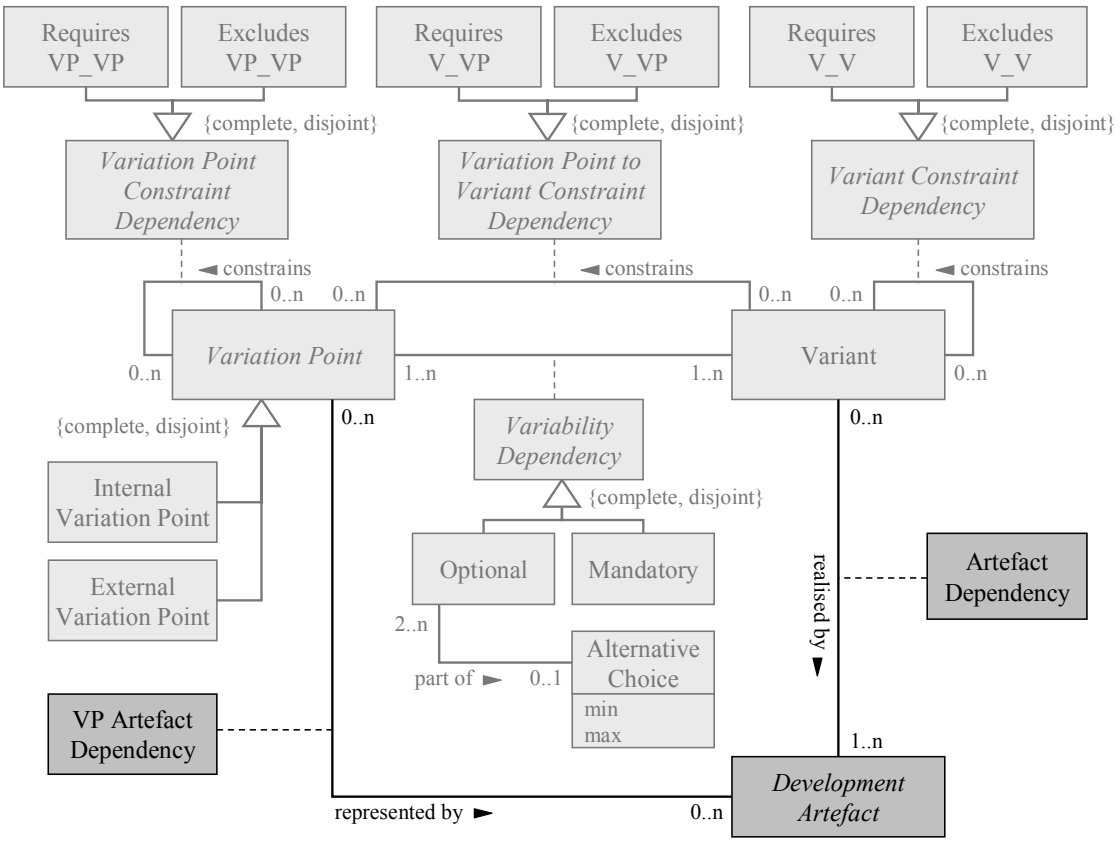


Abbildung 4.19: Metamodell des orthogonalen Variabilitätsmodells [PBL05]

Orthogonales Variabilitätsmodell (OVM) Das OVM wird durch Szenario 1 des Vorgehens evaluiert, da es ein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.

Das Metamodell des OVM ist in Abbildung 4.19 dargestellt. Das Teilmetamodell der hierarchischen Dekomposition wird im Metamodell des OVM nicht spezialisiert. Die beiden Klassen *Variante* und *Variationspunkt* stellen einen ontologischen Abgleich zu den gleichnamigen Metaklassen im Teilmetamodell der hierarchischen Dekom-

position dar, aber die Kardinalitäten der beiden vorhandenen Assoziationen führen nicht zu einer hierarchischen Dekomposition von Variationspunkten und Varianten. Statt dessen entsteht ein ungerichteter Graph. Die hierarchische Dekomposition von Variationspunkten und Varianten wird somit nicht unterstützt (A3).

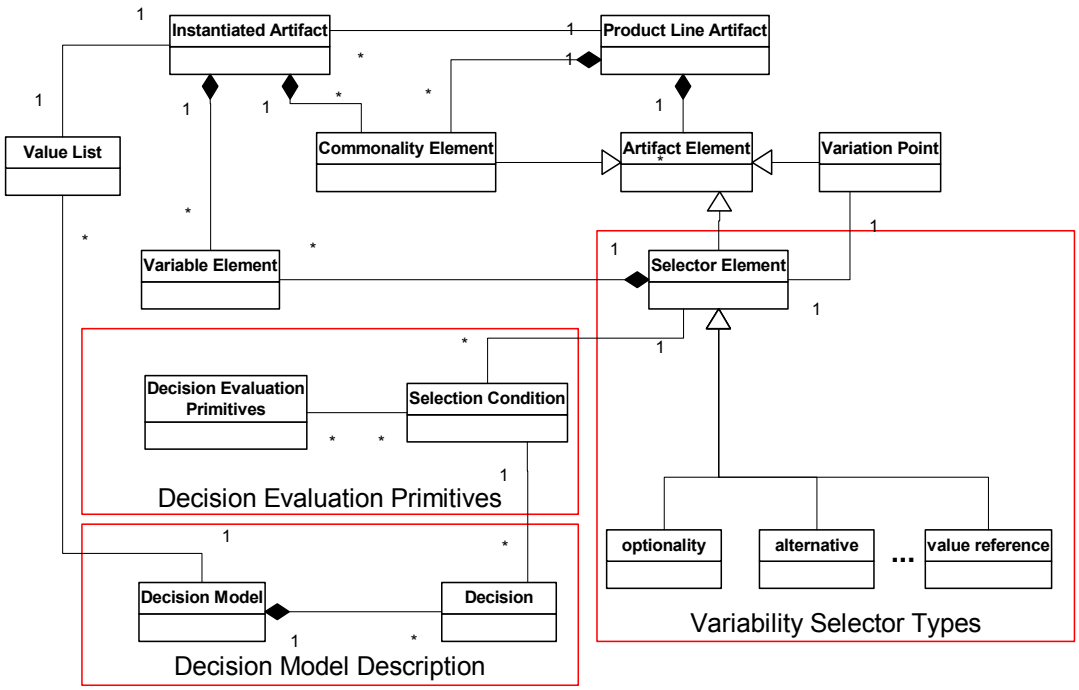


Abbildung 4.20: Metamodell des Entscheidungsmodells von Schmid und John [SJ03]

Schmid Der Ansatz von Schmid et al. wird durch Szenario 1 des Vorgehens evaluiert, da er ein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.

Der Ansatz (Abbildung 4.20) versucht ein Variabilitätsmanagement für den gesamten Software-Produktlinienentwicklungsprozess zu realisieren. Das Teilmetamodell der hierarchischen Dekomposition wird im Metamodell des Ansatzes nicht spezialisiert. Die Klasse *Variationspunkt* stellt einen ontologischen Abgleich zur Klasse Additionspunkt im Teilmetamodell dar. Weiterhin stellt die Klasse *Variable Element* einen ontologischen Abgleich zur Klasse *Variante* dar. Die Klasse ist auch mit den passenden Kardinalitäten über die Klasse *Selector Element* mit der Klasse Variationspunkt assoziiert. Die zweite Assoziation des Metamodells der hierarchischen

Dekomposition zwischen den beiden Klassen ist aber weder direkt noch über andere Klassen definiert. Die hierarchische Dekomposition ist damit nicht Teil des Ansatzes (A3).

Schnieders Der Ansatz von Schnieders et al. wird durch Szenario 3 des Vorgehens evaluiert, da er kein Metamodell definiert und nur Teile der definierten Anforderungen erfüllt.
Im Ansatz von Schnieders wird Variabilität mit Hilfe von Additionspunkten definiert (A1). Modifikationspunkte sind aber nicht vorgesehen (A2). Aus den gegebenen Modellinstanzen wird deutlich, dass eine hierarchische Dekomposition von Variabilität mit dieser Sprache nicht möglich ist (A3). Durch den Einsatz eines Featuremodells ist es möglich Implikationen und Ausschlüsse zu modellieren (A4).

Verbleibende Unterschiede Neben der Überprüfung der Erfüllung der durch das Metamodell des Variabilitätsmanagement gestellten Anforderungen sind darüber hinaus Eigenschaften interessant, die durch das Metamodell des Variabilitätsmanagements nicht adressiert werden.
In diesem Kontext wurden in den Ansätzen Featuremodell, orthogonales Variabilitätsmodell und dem Ansatz von Schmid deutlich, dass verschiedene Typen von Additionspunkten unterschieden wurden (z. B. *Optional*, *Alternative*, *Oder*, ...). Diese Typen schränken die Bindung der am Additionspunkt assoziierten Varianten in definierter Art und Weise ein. Zum Beispiel definiert der Typ *Alternative*, dass genau eine der Varianten gebunden werden darf.
Typen von Variationspunkten können durch eine Spezialisierungsbeziehung zur Metaklasse *Additionspunkt* auch im Metamodell des Variabilitätsmanagements definiert werden. Sie stellen dabei eine Einschränkung des Paradigmas dar, aber keine Erweiterung.

4.3.4 Kritik

Die Evaluation existierender Ansätze auf Basis von Metamodellen oder Modellinstanzen gegen ein Metamodell, welches die zu evaluierenden Anforderungen definiert bietet Vor- und Nachteile, die in diesem Abschnitt diskutiert werden:

Vorteile

- Der Vergleich ist (teil-) automatisierbar.
- Das Vergleichsergebnis ist mess- und nachvollziehbar, da auf Basis von formalen Modellen verglichen wird.
- Eigenschaften existierender Ansätze, die nicht im Metamodell der Anforderungen berücksichtigt wurden, können durch den Vergleich herausgestellt werden.

Durch die formale Definition von Anforderungen auf Basis eines Metamodells ist der Vergleich existierenden Ansätze mit diesem Metamodell automatisiert durchführbar. Dadurch kann die Existenz eines Teilmetamodells in dem Metamodell eines existierenden Ansatzes überprüft werden. Dazu wird überprüft, ob die Metaklassen des Teilmetamodells mit ihren Assoziationen, Kardinalitäten und OCL-Bedingungen im Metamodell des existierenden Ansatzes vorhanden sind. Ist dies der Fall kann die Überprüfung auf das gesamte Metamodell erweitert werden.
Durch dieses Vorgehen ist das Vergleichsergebnis mess- und nachvollziehbar, da deutlich wird, ob der existierende Ansatz die gleichen Eigenschaften ((Meta-) Klassen, Assoziationen, OCL-Bedingungen) bietet und wo Unterschiede liegen. Die Analyse der gefundenen Unterschiede muss manuell durchgeführt werden. Neben diesen Vorteilen existieren folgende Nachteile des gewählten Vorgehens:

Nachteile

- Unterschiede müssen manuell überprüft werden.
- Der Vergleich zwischen Modellinstanzen und dem Metamodell der Anforderungen ist eine manuelle Analyse.
- Die Erfüllung einer Anforderung ist teilweise nicht entscheidbar, wenn auf Basis von Modellinstanzen evaluiert wird. Eine Modellinstanz stellt immer nur ein Wort für die durch ein Metamodell definierte Sprache dar. Daher kann auf dieser Basis abhängig vom Beispiel nicht oder nur eingeschränkt auf die Modellierungsmöglichkeiten der Sprache geschlossen werden.
- Der ontologische Abgleich zwischen den Metaklassen bietet Fehlerpotential hinsichtlich der inkorrekten Definition von Abgleichen.

Das Vorgehen ist in der Lage Unterschiede zwischen verglichenen Metamodellen aufzuzeigen. Diese Unterschiede bedürfen einer manuellen Analyse. Diese Analyse stellt eine Interpretation durch eine Person dar und folgt somit keinen formalisierten Kriterien. Eine manuelle Analyse ist auch beim Vergleich von Modellinstanzen mit den als Metamodell formalisierten Anforderungen notwendig, da die Modellinstanzen normalerweise nur in konkreter Syntax gegeben sind. Hierbei kann es zu Fehlern bei der Interpretation kommen. Ein weiteres Problem bei der Evaluation auf Basis von Modellinstanzen ist der Beispielcharakter einer solchen Instanz. Eine Modellinstanz lässt nur einen eingeschränkten Schluss auf die Modellierungsmöglichkeiten der Sprache selbst zu. Daher kann die Erfüllung von Anforderungen unter Umständen nicht hinreichend evaluiert werden, falls die existierenden Beispiele keinen eindeutigen Schluss auf die Sprache zulassen.
Zuletzt ist auch der ontologische Abgleich eine manuelle Tätigkeit. Beispielsweise kann fälschlicherweise ein solcher Abgleich zwischen zwei Metaklassen definiert werden, deren Bedeutung aber nicht gleich ist.
Zusammenfassend betrachtet, bietet das Vergleichsvorgehen die Möglichkeit Teile des Vergleichs auf Basis von formalen Modellen durchzuführen. Dieser muss aber immer durch manuelle Tätigkeiten unterstützt werden.

4.4 Zusammenfassung

Um ein Variabilitätsmanagement für Software-Produktlinien zu realisieren, definierten die Ausführungen in diesem Kapitel zunächst die Anforderungen daran. Dazu wurden in Abschnitt 4.2 die Anforderungen an Variabilitätsmanagement schrittweise in Form eines Metamodells formalisiert. Dieses Vorgehen hat im Vergleich zur informellen Definition von Anforderungen den Vorteile, das ein teilautomatisierter Vergleich zwischen existierenden Ansätzen und dem Metamodell der Anforderungen möglich ist. Das dafür notwendige Vergleichsvorgehen ist in Abschnitt 4.3.2 vorgestellt worden. Es eignet sich für den Vergleich von Metamodellen und Modellinstanzen mit Metamodellen. Somit kann die Erfüllung von Anforderungen anhand von Metamodellen messbar gestaltet werden, da die Existenz oder Nicht-Existenz von (Meta-) Klassen, Assoziationen und OCL-Bedingungen zeigt, ob eine Anforderung erfüllt wird oder nicht.

Auf Basis des Vergleichsvorgehens wurde in Abschnitt 4.3 eine Evaluation existierender Ansätze für das Variabilitätsmanagement durchgeführt. Die Evaluationsergebnisse zeigen, dass einige Ansätze Beiträge zur Erfüllung von Anforderungen an Variabilitätsmanagement anbieten, aber keiner der Ansätze alle gestellten Anforderungen in geeigneter Weise erfüllt. Die Anforderungen hinsichtlich der Bindung (A5) und Konsistenz (A6) sind bisher durch keinen Ansatz hinreichend adressiert.

Das Metamodell des Variabilitätsmanagement bietet neben der Formalisierung der Anforderungen weiterhin den Vorteil, dass es sich als Basis für die Entwicklung eines Ansatzes zum Variabilitätsmanagement in allen Modellen des Software-Produktlinienentwicklungsprozesses eignet. Jeder Ansatz, der auf dieser Basis entwickelt wird, erfüllt damit automatisch die formalisierten Anforderungen. Das Metamodell des Variabilitätsmanagements definiert das Vorkommen und den Umgang mit Variabilität allgemeingültig. Die Nutzung dieses Metamodells für die Modellierung von Variabilität in den Modellen des Software-Produktlinienentwicklungsprozesses wird im nächsten Kapitel beschrieben.

Kapitel 5

Featurebasiertes Variabilitätsmanagement

Das im letzten Kapitel vorgestellte Metamodell des Variabilitätsmanagements definiert die Anforderungen an die Modellierung von Variabilität und den dabei existierenden Abhängigkeiten, sowie der Ableitung von Produkten aus einer Produktlinienplattform. Um die Spezifikation von Variabilität in Modellen des Software-Produktlinienentwicklungsprozesses zu ermöglichen, müssen die definierten Anforderungen durch alle Modellierungssprachen der Produktlinienplattform erfüllt werden.
Um dies zu erreichen, wird in diesem Kapitel ein Sprachkonstruktionsprozess für die Erweiterung von *Modellierungssprachen* zu *Variabilitätsmodellierungssprachen* vorgestellt und dieser exemplarisch auf die Modellierungssprachen für Anwendungsfallbeschreibungen und Testfälle angewendet.
Darüber hinaus wird mit dem featurebasiertem Variabilitätsmanagement ein Ansatz für das Variabilitätsmanagement in einem zentralen Modell konstruiert. Dieser Ansatz basiert auf dem Featuremodell, welches im letzten Abschnitt evaluiert worden ist.

5.1 Modellierung von Variabilität in Modellen

Für die Modellierung von Variabilität in Modellen des Software-Produktlinienentwicklungsprozesses werden Variabilitätsmodellierungssprachen benötigt. Eine Variabilitätsmodellierungssprache kann entweder neu oder auf Basis einer bereits existierenden Modellierungssprache definiert werden. Da in der Softwaretechnik bereits

unterschiedliche Modellierungssprachen für die Spezifikation vom Modellen existieren, wie zum Beispiel die UML [OMG09], wird in dieser Arbeit ein Vorgehen für die Erweiterung solcher Modellierungssprachen zu Variabilitätsmodellierungssprachen beschrieben. Ziel dabei ist es, die Veränderungen an existierenden Modellierungssprachen so gering wie möglich zu halten. Somit ist die Einarbeitung für Nutzer, die bereits mit einer Modellierungssprache vertraut sind, einfach.

5.1.1 Sprachkonstruktionsprozess für Variabilitätsmodellierungssprachen

Im Rahmen des Software-Produktlinienentwicklungsprozesses werden unterschiedliche Modellierungssprachen für die Spezifikation von Modellen, wie Anwendungsfällen, Quellcode und Testfällen genutzt. Die Erweiterung von Modellierungssprachen um Variabilität kann durch unterschiedliche Vorgehen unterstützt werden. Die OMG zum Beispiel definiert in der Version 2.2 der UML [OMG09, FFVM04] *Profile* für die domänenspezifische Spezialisierung der UML. Die Ausgangsbasis der Erweiterung ist aber immer die UML selbst. Ein UML-Profil für Variabilität wird zum Beispiel in [KL07] auf Basis der orthogonalen Variabilitätsmodell aus [PBL05] definiert. Da das SPL Paradigma den gesamten Entwicklungsprozess beeinflusst (vgl. Abschnitt 2.1.2), wird auch die Variabilitätsmodellierung in Modellen notwendig, die nicht auf der UML oder eines ihrer Profile basiert. Daher wird in dieser Arbeit ein Vorgehen definiert, welches zum Ziel hat, nicht nur die Sprache UML um Variabilität zu erweitern.

Die Erweiterung einer Modellierungssprache um Variabilität basiert daher auf der Spezialisierung des Metamodells des Variabilitätsmanagements, welches in Kapitel 4 eingeführt wurde. Abbildung 5.1 stellt die Erweiterung im Kontext der Modellebenen dar. Durch die Spezialisierungsbeziehung bietet eine Variabilitätsmodellierungssprache grundsätzlich die Möglichkeit der Modellierung von Variabilität. Durch die Formulierung von OCL-Bedingungen an die Variabilitätsmodellierungssprache wird dann festgelegt, welche Modellelemente der Variabilitätsmodellierungssprache in welcher Form variiert werden können. Durch Instanziierung des Metamodells einer Variabilitätsmodellierungssprache können dann Modelle mit Variabilität spezifiziert werden.

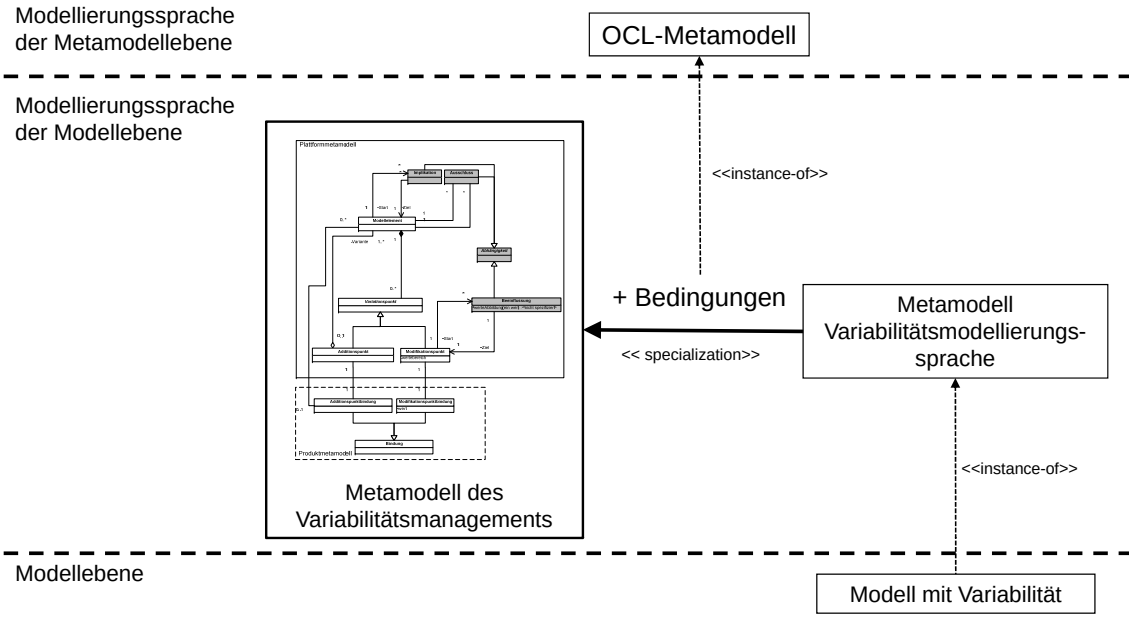


Abbildung 5.1: Definition von Variabilitätsmodellierungssprachen durch Spezialisierung des Metamodells des Variabilitätsmanagements und Einschränkung mit OCL-Bedingungen

Für die Erweiterung einer Modellierungssprache um Variabilität zu einer Variabilitätsmodellierungssprache wird nun ein Vorgehen definiert, welches in Abbildung 5.2 als UML-Aktivitätendiagramm dargestellt ist. Das Vorgehen arbeitet auf Metamodellen solcher Modellierungssprachen, die die Modellierung von Variabilität nicht unterstützen. Im Folgenden werden die einzelnen Schritte des Vorgehens näher erläutert:

1. Auswahl von Metaklassen im Metamodell der Modellierungssprache

Im ersten Schritt werden diejenigen Metaklassen des Metamodells der zu erweiternden Modellierungssprache ausgewählt, an denen Variabilität spezifizierbar sein soll. Diese Auswahl ist eine manuelle Tätigkeit.
Abbildung 5.3 zeigt dazu ein Beispiel mit einem Ausschnitt eines Metamodells einer Modellierungssprache in UML-Klassendiagrammsyntax. Ausgewählt für die Erweiterung um Variabilität werden die Metaklassen Y und Z. Die Metaklassen X und T werden nicht ausgewählt, da sie Spezialisierungen der *Metaklasse* Z darstellen und daher die Erweiterung von dieser erben.

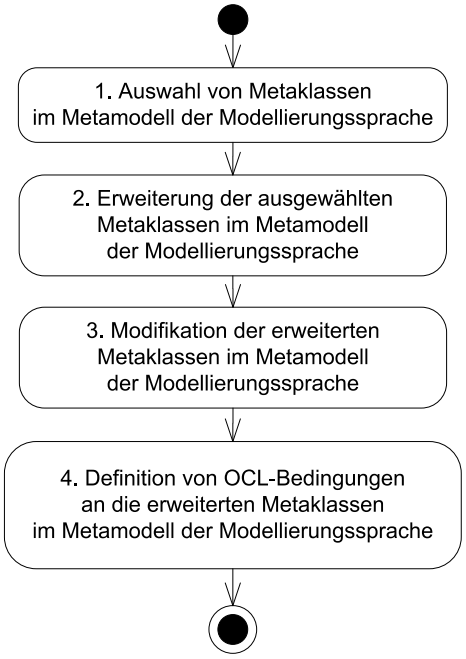


Abbildung 5.2: Vorgehen für die Erweiterung einer Modellierungssprache zu einer Variabilitätsmodellierungssprache

2. Erweiterung der ausgewählten Metaklassen im Metamodell der Modellierungssprache

Um die Assoziation von Variationspunkten zu den ausgewählten Metaklassen zu ermöglichen, wird eine Spezialisierungsbeziehung zwischen diesen Metaklassen und dem Metamodell des Variabilitätsmanagements definiert. Das Metamodell des Variabilitätsmanagements definiert für seine Metaklasse *Modellelement* das Vorkommen von Variabilität. Daher *spezialisieren* die Metaklassen aus dem Metamodell einer Modellierungssprache die Metaklasse *Modellelement* des Metamodells des Variabilitätsmanagements.
Abbildung 5.4 stellt diese Spezialisierungsbeziehung für die *Metaklassen* Y und Z dar.

3. Modifikation der erweiterten Metaklassen im Metamodell der Modellierungssprache

Die im letzten Schritt definierten Spezialisierungsbeziehungen können bei der Spezifikation von Modellen mit der Modellierungssprache zu einer eingeschränkten Nutz-

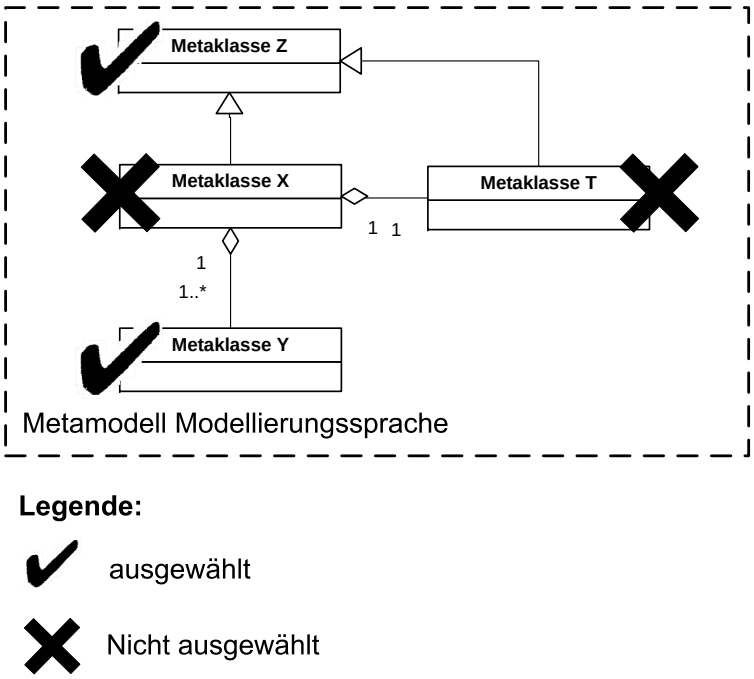


Abbildung 5.3: Schritt 1: Beispiel für die Auswahl von Metaklassen aus dem Metamodell einer Modellierungssprache

barkeit führen. Um dies zu verdeutlichen wird das Beispiel aus Abbildung 5.4 wieder aufgegriffen. Abbildung 5.5 stellt dann das Problem dar:

Der Nutzer der Modellierungssprache will nur einen Additionspunkt mit zwei Klassen von Typ *Metaklasse Y* an die Klasse vom Typ *Metaklasse X* assoziieren (vgl. Abbildung 5.5.(a)). Es soll keine weitere Klasse vom Typ *Metaklasse Y* direkt an die Klasse vom Typ *Metaklasse X* assoziiert werden. Der Nutzer möchte damit die beiden Klassen als Alternativen modellieren. Das in (a) dargestellte Modell ist aber keine gültige Instanz des Metamodells aus Abbildung 5.4). Erst durch das Assoziieren einer Klasse vom Typ Metaklasse Y direkt an der Klasse vom Typ Metaklasse X, wie in (b) dargestellt, ist die Instanzbeziehung hergestellt.

Um diesen Fall modellieren zu können, werden die Kardinalitäten der Assoziation zwischen den *Metaklassen X* und *Y* von 1 und 1..∗ auf 0..1 und 0..∗ *gelockert*. Somit muss eine Klasse des Typs *Metaklasse Y* nicht mehr zwingend an einer Klasse des Typ *Metaklasse X* assoziiert sein und eine Klasse des Typs *Metaklasse Y* selbst muss

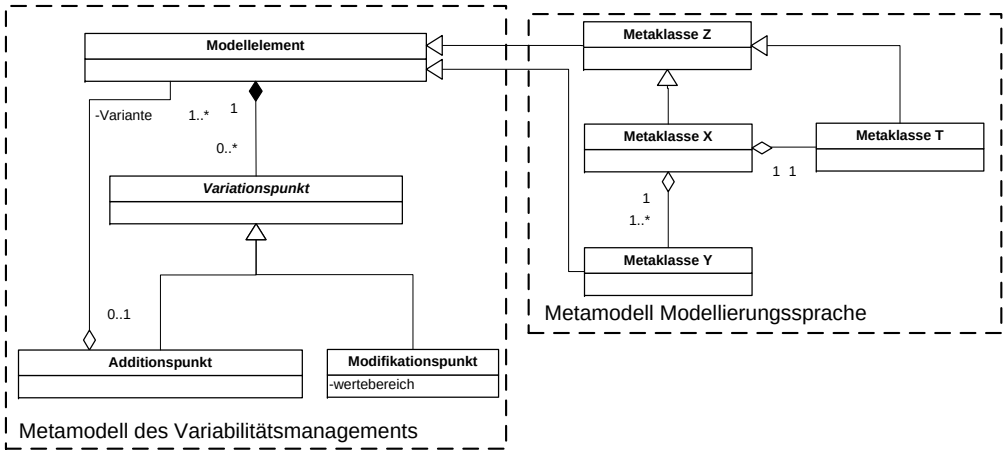


Abbildung 5.4: Schritt 2: Beispiel für die Spezialisierungsbeziehung zwischen Metaklassen des Metamodells einer Modellierungssprache und der Metaklasse *Modellelement* des Metamodells des Variabilitätsmanagements

nicht zwingend an eine Klasse des Typs *Metaklasse X* assoziiert sein. in Abbildung 5.6 ist die Lockerung hervorgehoben. Auch die Kardinalitäten zwischen *Metaklasse X* und *T* werden von 1 und 1 auf 0..1 und 0..1 gelockert. Allgemein formuliert tritt die eingeschränkte Nutzbarkeit bei den Kardinalitäten 1 bzw. 1..∗ auf. Gelockert werden diese Kardinalitäten durch 0..1 bzw. 0..∗.

4. Definition von OCL-Bedingungen an die erweiterten Metaklassen im Metamodell der Modellierungssprache

Die Lockerung der Kardinalitäten führt neben dem erwünschten Effekt aber auch dazu, dass Modellinstanzen gebildet werden können, die aus Sicht des Ausgangsmetamodells und der Modellierung von Variabilität nicht erwünscht sind. Die drei möglichen Fälle sind beispielhaft in Abbildung 5.7 dargestellt und stellen Instanzen des Metamodells aus Abbildung 5.6 dar.

Im Modell aus (a) existiert keine Klasse vom Typ *Metaklasse Y* im Modell. Im Ausgangsmetamodell (vgl. Abbildung 5.4) wird aber mindestens die Assoziation einer Klasse vom Typ *Metaklasse Y* gefordert. Ziel der Erweiterung um Variabilität ist es nun, dass entweder Klassen des Typs *Metaklasse Y* oder auch Additionspunkte mit Klassen des Typs *Metaklasse Y* an *Metaklasse X* assoziierbar sind. In jedem Fall muss mindestens eine von beiden Assoziationen existieren. Dies wird als OCL-Bedingung an die *Metaklasse X* formuliert:

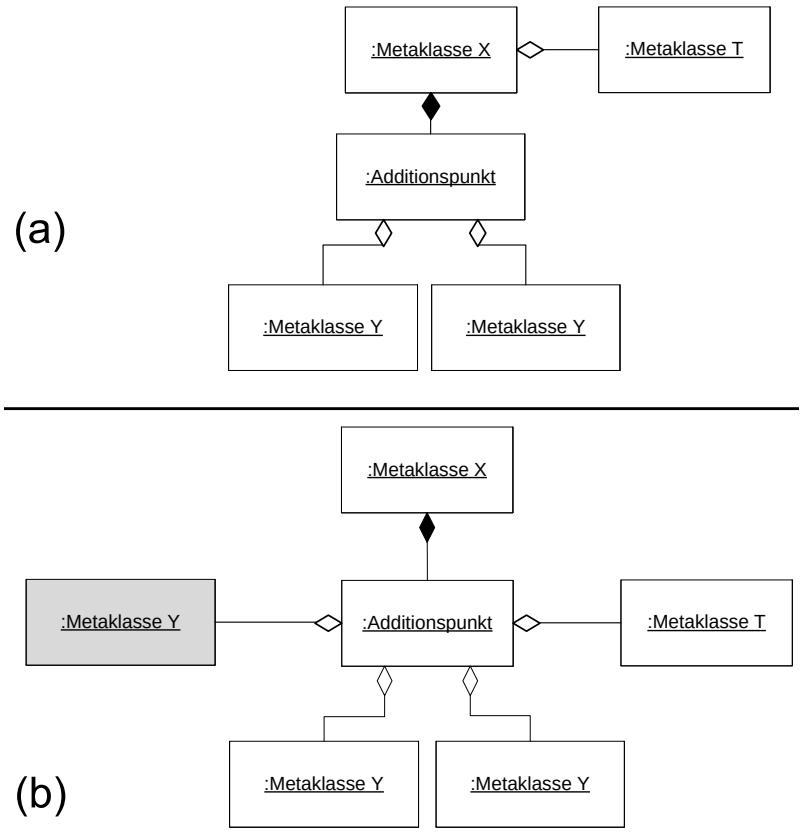


Abbildung 5.5: Beispiel einer nicht modellierbaren Instanz

Bedingung 8. ***Context*** *Metaklasse X* ***inv:***
*self→**exists**((ap:Additionspunkt|ap.Metaklasse Y) or Metaklasse Y)*

Im Modell aus (b) ist die Klasse $Y1$ vom Typ *Metaklasse Y* an Klasse X des Modells assoziiert. Es existiert eine weitere Klasse $Y2$, die keine Assoziation zu einer Klasse vom Typ *Metaklasse X* besitzt. Im Ausgangsmetamodell ist definiert, dass eine Klasse vom Typ *Metaklasse Y* immer an genau eine Klasse des Typs *Metaklasse X* assoziiert sein muss. Daher wird folgende Einschränkung formuliert, damit Klassen des Typs *Metaklasse Y* immer entweder eine direkte Assoziation zu einer Klasse des Typs *Metaklasse X* besitzen oder über einen Additionspunkt daran assoziiert sind:

Bedingung 9. ***Context*** *Metaklasse Y* ***inv:***
*self→**exists**((ap:Additionspunkt|ap.Metaklasse X) xor Metaklasse X)*

Im Modell aus (c) ist sowohl die Klasse $T1$ des Typs *Metaklasse T* direkt als auch die Klasse $T2$ über einen Additionspunkt an eine Klasse X assoziiert. Somit wäre

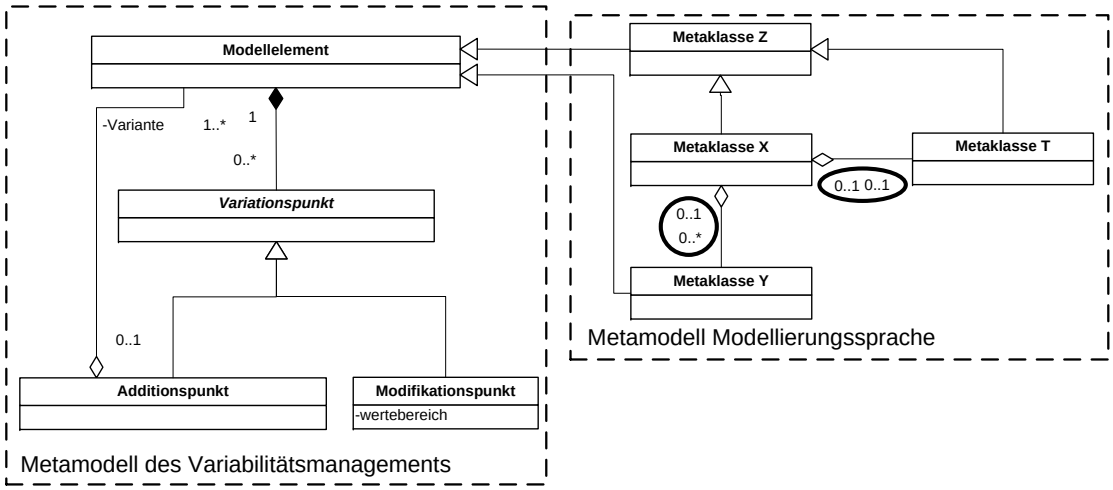


Abbildung 5.6: Schritt 3: Beispiel für die Modifikation der erweiterten Metaklassen des Metamodells einer Modellierungssprache

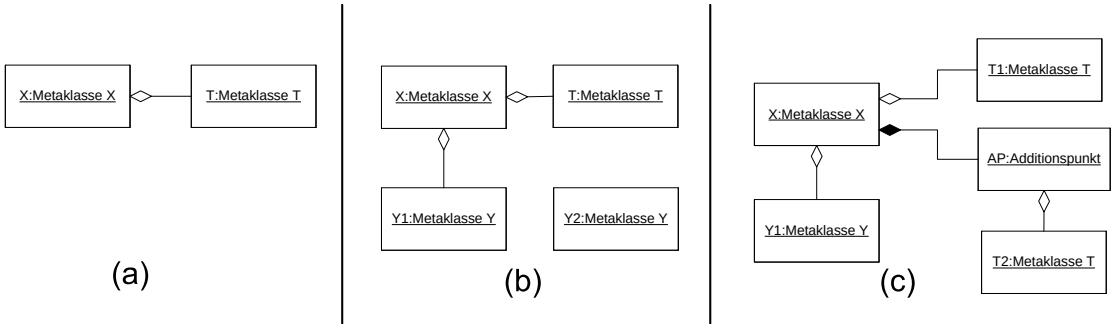


Abbildung 5.7: Beispiele für Probleme bei der Lockerung von Kardinalitäten

es bei einer Bindung der Klasse $T2$ möglich zwei Klassen des Typs *Metaklasse T* an Klasse X zu assoziieren. Im Ausgangsmetamodell ist diese Assoziation auf genau eine Klasse des Typs *Metamodell X* und *Metamodell T* eingeschränkt. Auch hier wird in der gleichen Art wie in Bedingung 9 eingeschränkt, dass entweder genau eine Klasse direkt oder aber über einen Additionspunkt assoziiert sein kann.

Bedingung 10. ***Context*** *Metaklasse X* ***inv:***
*self→**exists**((ap:Additionspunkt|ap.Metaklasse T) xor Metaklasse T)*

Bedingung 11. ***Context*** *Metaklasse T* ***inv:***
*self→**exists**((ap:Additionspunkt|ap.Metaklasse X) xor Metaklasse X)*

Die im vorgestellten Beispiel genutzten OCL-Bedingungen lassen sich für die Lockerung von Kardinalitäten verallgemeinern, wie in Tabelle 5.1 dargestellt wird.

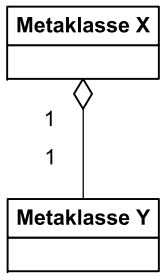
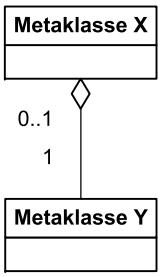
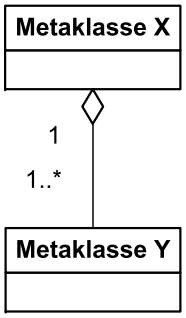
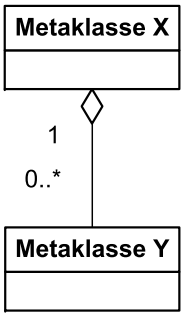
Lockerung von	**auf**	**Muster-OCL-Bedingung**
		**Context** Metaklasse Y **inv**: self→**exists**((ap:Additionspunkt|ap.Metaklasse X) **xor** Metaklasse X)
		**Context** Metaklasse X **inv**: self→**exists**((ap:Additionspunkt|ap.Metaklasse Y) **or** Metaklasse Y)

Tabelle 5.1: Muster-OCL-Bedingungen für die Lockerung von Kardinalitäten

Für jede Metaklasse, die um eine Spezialisierungsbeziehung erweitert wurde und alle Metaklassen, die solche Metaklassen spezialisieren, muss nun durch Bedingungen die Art und Weise der modellierbaren Variabilität definiert werden.

Abbildung 5.8 stellt die dazu notwendigen Entscheidungen in Form eines *Entscheidungsbaums* dar [SM09]. Die Blätter des Entscheidungsbaum sind *Maßnahmen* und geben die Art der zu definierenden Bedingungen an:

- (A) Falls Additionspunkte an der betrachteten Metaklasse modellierbar sein sollen, muss durch eine OCL-Bedingung eingeschränkt werden, welche Metaklassen des Metamodells der Modellierungssprache als Variante an den Additionspunkten assoziiert sein dürfen. Das Muster für eine solche OCL-Bedingung hat folgende Gestalt:

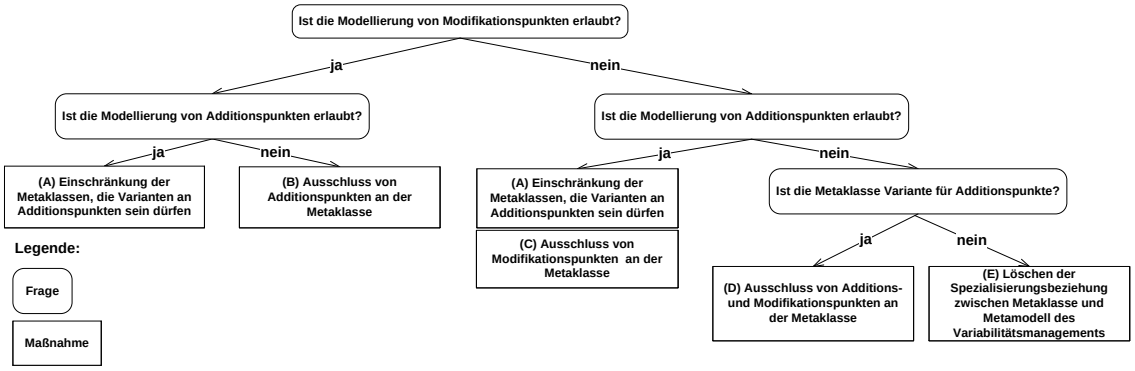


Abbildung 5.8: Entscheidungsbaum für die Definition von Bedingungen an die erweiterten Metaklassen des Metamodells der Modellierungssprache

Bedingung 12. ***Context*** *Bezeichner der Metaklasse* ***inv:***
*self.Additionspunkt→**forAll**(ap:Additionspunkt|ap.Variante*
*→**forAll**(v:Variante|*
v.elementType.conformsTo(„Bezeichner einer Metaklasse“) or
v.elementType.conformsTo(„Bezeichner einer Metaklasse“) or
...)

Alle Metaklassen, die als Varianten assoziierbar sein sollen, werden als Disjunktion in die Überprüfung des Elementtyps eingefügt. Eine notwendige Bedingung für die Auswahl einer Metaklasse als Variante ist, dass sie im Metamodell der Modellierungssprache eine direkte Assoziation zur betrachteten Metaklasse besitzen.

- (B) Falls keine Additionspunkte modellierbar sein sollen, werden diese durch eine OCL-Bedingung ausgeschlossen. Das Muster für eine solche OCL-Bedingung hat folgende Gestalt:

Bedingung 13. ***Context*** *Bezeichner der Metaklasse* ***inv:***
self.Additionspunkt→isEmpty()

- (C) Falls keine Modifikationspunkte modellierbar sein sollen, werden diese durch eine OCL-Bedingung ausgeschlossen. Das Muster für eine solche OCL-Bedingung hat folgende Gestalt:

Bedingung 14. ***Context*** *Bezeichner der Metaklasse* ***inv:***
self.Modifikationspunkt→isEmpty()

- (D) Falls weder Additionspunkte noch Modifikationspunkte modellierbar sein sollen, die Metaklasse aber Variante an Additionspunkten anderer Metaklassen sein kann, wird durch eine OCL-Bedingung die Modellierung von Additions- und Modifikationspunkten an der Metaklasse ausgeschlossen.

Bedingung 15. ***Context*** *Bezeichner der Metaklasse* ***inv:***
self.Additionspunkt→isEmpty() and self.Modifikationspunkt→isEmpty()

- (E) Falls weder Additionspunkte noch Modifikationspunkte modellierbar sein sollen und die Metaklasse keine Variante an Additionspunkten anderer Metaklassen sein kann, wird die Spezialisierungsbeziehung zum Metamodell des Variabilitätsmanagements gelöscht, da die Modellierung von Variabilität an dieser Metaklasse nicht möglich sein soll.

Die Entscheidung, ob an eine Metaklasse Modifikationspunkte oder Additionspunkte assoziierbar sein können, wird manuell durchgeführt. Gleiches gilt für die Bestimmung der Metaklassen die als Varianten an einem Additionspunkt assoziierbar sein sollen.

Für ein konkretes Beispiel wird nun das Metamodell aus Abbildung 5.6 wieder aufgegriffen. Im Beispiel aus der Abbildung wird für *Metaklasse X* entschieden, dass Additionspunkte assoziierbar sein sollen. Die Varianten der Additionspunkte werden auf *Metaklasse Y* eingeschränkt, da diese Metaklasse einerseits eine direkte Assoziation zur *Metaklasse X* besitzt, also damit diese notwendige Voraussetzung erfüllt. Andererseits ist die Modellierung dieser Metaklasse als Variante auch von den Nutzern der Sprache gewünscht:

Bedingung 16. ***Context*** *Metaklasse X* ***inv:***
*self.Additionspunkt→**forAll**(ap:Additionspunkt|ap.Variante*
*→**forAll**(v:Variante| v.elementType.conformsTo(„Metaklasse Y“)))*

An *Metaklasse Y* selbst sollen keine Additionspunkte assoziierbar sein, da sie selbst keine weiteren Metaklassen aggregiert. Daher wird mit Hilfe einer OCL-Bedingung die Modellierung von Additionspunkten an dieser Metaklasse verboten.

Bedingung 17. ***Context*** *Metaklasse Y* ***inv:***
self.Additionspunkt→isEmpty()

Das vorgestellte Vorgehen wird im Folgenden dazu genutzt existierende Modellierungssprachen für Anwendungsfallbeschreibungen und Testfälle zu Variabilitätsmodellierungssprachen zu erweitern.

5.1.2 Anforderungsmodellierungssprache mit Variabilität

Anwendungsfallbeschreibungen, wie in Kapitel 2 definiert, ermöglichen die Spezifikation von Anforderungen in semiformaler Form. Wenn Anwendungsfallbeschreibungen im Rahmen einer Software-Produktlinienentwicklung genutzt werden, muss die Modellierung von Variabilität in ihnen möglich sein, um deren Veränderlichkeit beschreiben und Wiederverwendung ermöglichen zu können.
In der Literatur existieren verschiedene Ansätze zur Beschreibung von Variabilität in UML-Anwendungsfällen. In [HP04] und [Maß07] wird die Erweiterung von UML-Anwendungsfalldiagrammen beschrieben. Dabei ist es möglich, UML-Anwendungsfälle als variabel zu deklarieren und diese durch variable *extends*- oder *include*-Beziehungen mit anderen Anwendungsfällen zu kombinieren. [Maß07] erweitert zu diesem Zweck auch das Metamodell des UML-Anwendungsfalls um Variabilität. In [FGLN04] wird die Variabilitätsmodellierung innerhalb einer Anwendungsfallbeschreibung beschrieben. Dabei sind Modellelemente von Anwendungsfällen, wie zum Beispiel Schritte oder Vor- und Nachbedingungen, als variabel spezifizierbar.
Beide Arten von Ansätzen für die Modellierung von Variabilität in Anwendungsfällen offenbaren einige Schwächen:

- Die Modellierung von Variabilität nur mit UML-Anwendungsfällen als Varianten ermöglicht keine Modellierung von Variabilität zum Beispiel in Vorbedingungen von Anwendungsfallbeschreibungen [Maß07, HP04].
- Der Ansatz aus [FGLN04] ist nicht in das Metamodell des UML-Anwendungsfalls oder einer Anwendungsfallbeschreibung integriert worden. Dadurch ist die Sprache nicht wohldefiniert.
- Für Anwendungsfallbeschreibungen wurde nicht explizit definiert, welches Modellelement in welcher Form variieren kann.

- Abhängigkeiten zwischen Varianten und Modifikationspunkten in und zwischen Anwendungsfallbeschreibungen sowie anderen Modellen werden in [FGLN04] nicht berücksichtigt.

Aus diesen Gründen wird in dieser Arbeit mit Hilfe des im letzten Abschnitt vorgestellten Vorgehens eine Erweiterung des Anwendungsfallmetamodells aus Abschnitt 2.2.1 durchgeführt. Im Folgenden werden die einzelnen Schritte des Vorgehens aus Abbildung 5.1 durchgeführt:

1. Auswahl von Metaklassen im Anwendungsfallmetamodell

Das Anwendungsfallmetamodell ist in Abbildung 5.9 dargestellt. Die markierten Metaklassen werden für die Erweiterung um Variabilität ausgewählt. Durch diese Auswahl ist es möglich an jeder Metaklasse Variabilität zu modellieren, da die übrigen Metaklassen Spezialisierungen der ausgewählten Metaklassen darstellen.

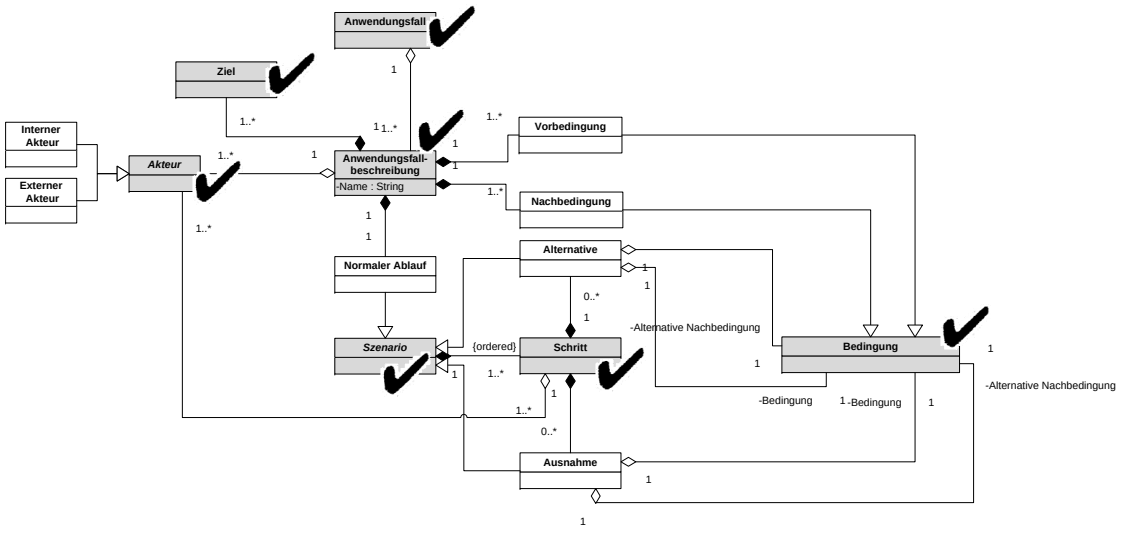


Abbildung 5.9: Auswahl der zu erweiternden Metaklassen des Anwendungsfallmetamodells

2. Erweiterung der ausgewählten Metaklassen

Die im ersten Schritt ausgewählten Metaklassen erhalten im zweiten Schritt eine Spezialisierungsbeziehung zur Metaklasse *Modellelement* aus dem Metamodell des Variabilitätsmanagements (vgl. Abbildung 5.10).

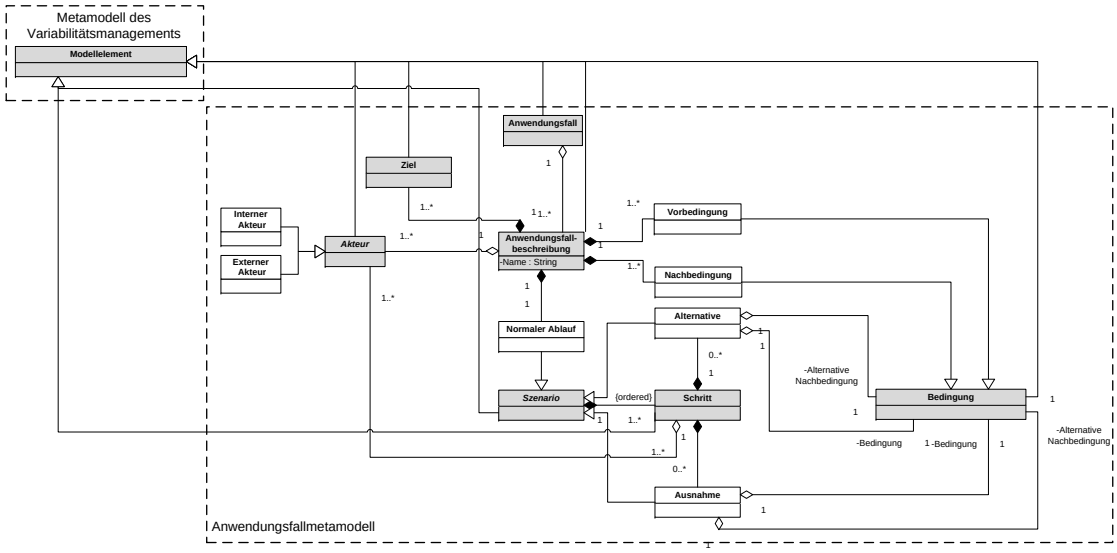


Abbildung 5.10: Erweiterung des Metamodells des Anwendungsfalls um Variabilität

3. Modifikation der erweiterten Metaklassen

Die Kardinalitäten 1 und 1..∗ an Assoziationen von Metaklassen die von einer Spezialisierungsbeziehung zum Metamodell des Variabilitätsmanagements betroffen sind, werden auf die Kardinalitäten 0..1 und 0..∗ gelockert. Dies ist in Abbildung 5.11 dargestellt. Dadurch ist es zum Beispiel möglich in einem Anwendungsfallmodell eine Anwendungsfallbeschreibung zu modellieren, die einen Additionspunkt mit zwei Zielen als Varianten besitzt. Bei der Ableitung des Anwendungsfallmodells muss dann mindestens eines der beiden Ziele gebunden werden. Ohne die Lockerung der Kardinalitäten müsste an einer Anwendungsfallbeschreibung immer mindestens ein Ziel assoziiert sein.

4. Definition von OCL-Bedingungen an die erweiterten Metaklassen

Bei der Lockerung der Kardinalitäten werden OCL-Bedingungen nach den Mustern aus Tabelle 5.1 an die Metaklassen definiert. Im Anschluss daran wird geklärt, welche Metaklasse in welcher Art variiert werden kann:

Jede Metaklasse des Anwendungsfallmetamodells soll durch Modifikationspunkte erweiterbar sein. In Bezug auf die Modellierung von Additionspunkten an den Metaklassen ergeben sich die in Tabelle 5.2 dargestellten Ergebnisse des Entschei-

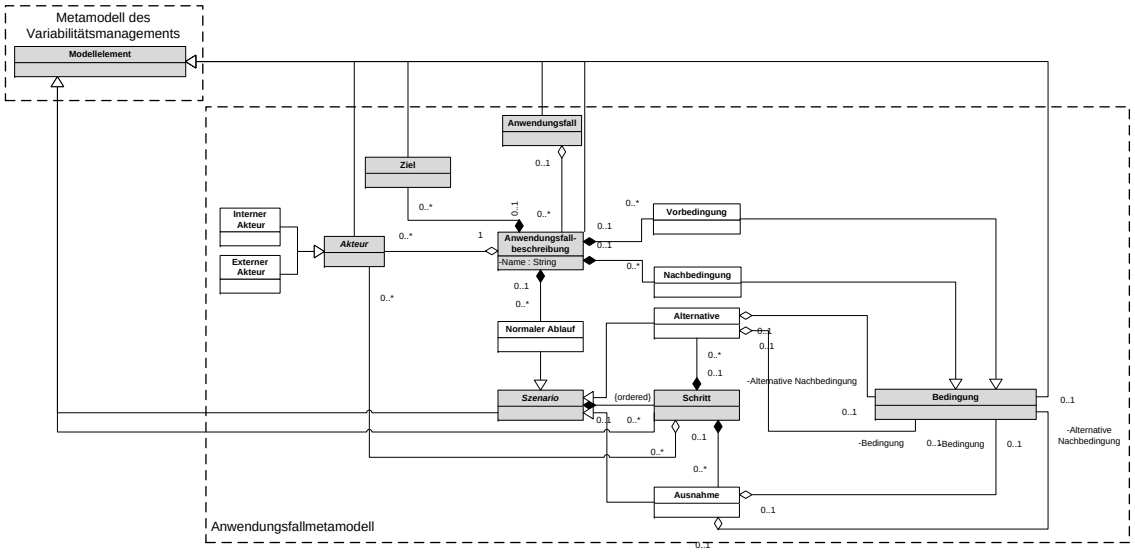


Abbildung 5.11: Lockerung der Kardinalitäten im Anwendungsfallmetamodell

dungsbaums aus Abbildung 5.8:

Metaklasse	**Modifikations-punkte**	**Additions-punkte**	**Variante**	**Maßnahmen**
Akteur	ja	nein	-	(B)
Anwendungsfall	ja	ja	-	(A)
Anwendungsfall-beschreibung	ja	ja	-	(A)
Bedingung	ja	nein	-	(B)
Schritt	ja	ja	-	(A)
Szenario	ja	ja	-	(A)
Ziel	ja	nein	-	(B)

Tabelle 5.2: Antworten der Entscheidungsbaumfragen für die Metaklassen des Anwendungsfallmetamodells

Für die Metaklassen an denen Additionspunkte assoziierbar sein sollen, müssen nun die Metaklassen ausgewählt werden, die als Varianten an Additionspunkte assoziiert werden können. Die Auswahl für die betroffenen Metaklassen ist in Tabelle 5.3 dargestellt.

Metaklasse	**Als Varianten assoziierbare Metaklassen**
Akteur	-
Anwendungsfall	Anwendungsfallbeschreibung
Anwendungsfall-beschreibung	Akteur, Normaler Ablauf, Vorbedingung, Nachbedingung, Ziel
Bedingung	-
Schritt	Akteur, Ausnahme, Alternative
Szenario	Schritt
Ziel	-

Tabelle 5.3: Anwendungsfallmetamodell: Auswahl der Metaklassen die als Varianten an Additionspunkte assoziierbar sein sollen

Als Maßnahmen werden, wie in Abschnitt 5.1.1 beschrieben, OCL-Bedingungen an die jeweiligen Metaklassen definiert. Die Metaklasse Anwendungsfallbeschreibung erlaubt beispielsweise die Modellierung von Additionspunkten mit den in Tabelle 5.3 definierten Metaklassen als Varianten (Maßnahme (A), Bedingung 18):

Bedingung 18. ***Context*** *Anwendungsfallbeschreibung* ***inv:***
*self.Additionspunkt→**forAll**(ap:Additionspunkt|ap.Variante*
*→**forAll**(v:Variante| v.elementType.conformsTo(„Akteur“) or*
v.elementType.conformsTo(„Ziel“) or
v.elementType.conformsTo(„Vorbedingung“) or
v.elementType.conformsTo(„Nachbedingung“) or
v.elementType.conformsTo(„Normaler Ablauf“)))

An die Metaklasse Akteur sollen keine Additionspunkte modellierbar sein (Maßnahme (B), Bedingung 19):

Bedingung 19. ***Context*** *Akteur* ***inv:***
self.Additionspunkt→isEmpty()

Die übrigen OCL-Bedingungen werden analog zu diesen zwei Beispielen mit Hilfe des Sprachkonstruktionsprozesses aus Abschnitt 5.1.1 spezifiziert.

5.1.3 Testfallmodellierungssprache mit Variabilität

Ein Testfall basiert auf dem in dieser Arbeit definierten Testfallmetamodell aus Abschnitt 2.3. Dieses Metamodell wurde in Abschnitt 3.2.2 dazu eingesetzt existierende Variabilitätsmodellierungssprachen für Testfälle zu evaluieren.
Da keine der existierenden Sprachen alle Anforderungen an die Modellierung von Variabilität in Testfällen erfüllt, wird in dieser Arbeit nun mit Hilfe des Sprachkonstruktionsprozesses aus Abschnitt 5.1.1 das Testfallmetamodell um Variabilität erweitert. Im folgenden werden die einzelnen Schritte des Vorgehens aus Abbildung 5.1 durchgeführt.

1. Auswahl von Metaklassen im Testfallmetamodell

Die Erweiterung des Testfallmetamodells um Variabilität geschieht analog der Erweiterung des Anwendungsfallmetamodells. Dazu werden wiederum die Metaklassen des Testfallmetamodells ausgewählt, an denen Variabilität modellierbar sein soll, wie in Abbildung 5.12 dargestellt ist.

2. Erweiterung der ausgewählten Metaklassen

Die im letzten Schritt ausgewählten Metaklassen erhalten eine Spezialisierungsbeziehung zur Metaklasse *Modellelement* des Metamodells des Variabilitätsmanagements (vgl. Abbildung 5.13).

3. Modifikation der erweiterten Metaklassen

Die Kardinalitäten der Assoziationen an erweiterten Metaklassen werden gelockert, falls sie 1 oder 1..$*$ entsprechen. Dies erfolgt analog zum Vorgehen bei Anwendungsfallbeschreibungen (vgl. Abschnitt 5.1.2). In Abbildung 5.14 ist die Lockerung der Kardinalitäten der erweiterten Metaklassen dargestellt.

4. Definition von OCL-Bedingungen an die erweiterten Metaklassen

Bei der Lockerung der Kardinalitäten werden OCL-Bedingungen nach den Mustern aus Tabelle 5.1 an die Metaklassen definiert. Im Anschluss daran wird geklärt, welche Metaklasse in welcher Art variiert werden kann.

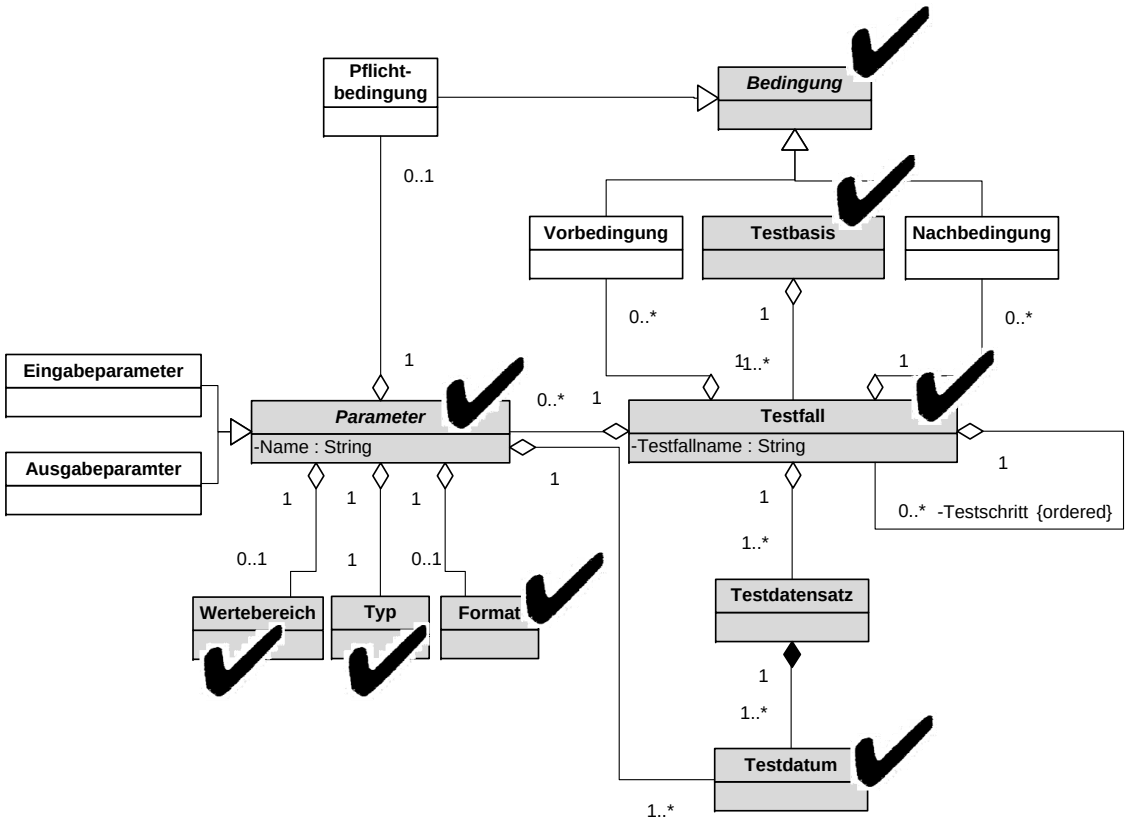


Abbildung 5.12: Auswahl der Metaklassen des Testfallmetamodells für die Erweiterung um Variabilität

Mit Hilfe des Entscheidungsbaums aus Abbildung 5.8 werden die Maßnahmen bestimmt, die zur Einschränkung der erweiterten Metaklassen in Form von OCL-Bedingungen formuliert werden müssen. Tabelle 5.4 stellt das Ergebnis der Entscheidungen für alle erweiterten Metaklassen dar.

Für die Metaklassen an denen Additionspunkte assoziierbar sein sollen, müssen nun die Metaklassen ausgewählt werden, die als Varianten an Additionspunkte assoziiert werden können. Die Auswahl für die betroffenen Metaklassen ist in Tabelle 5.5 dargestellt.

Die Spezifikation der OCL-Bedingungen geschieht nun analog zur Anforderungsspezifikation mit Variabilität (vgl. Abschnitt 5.1.2).

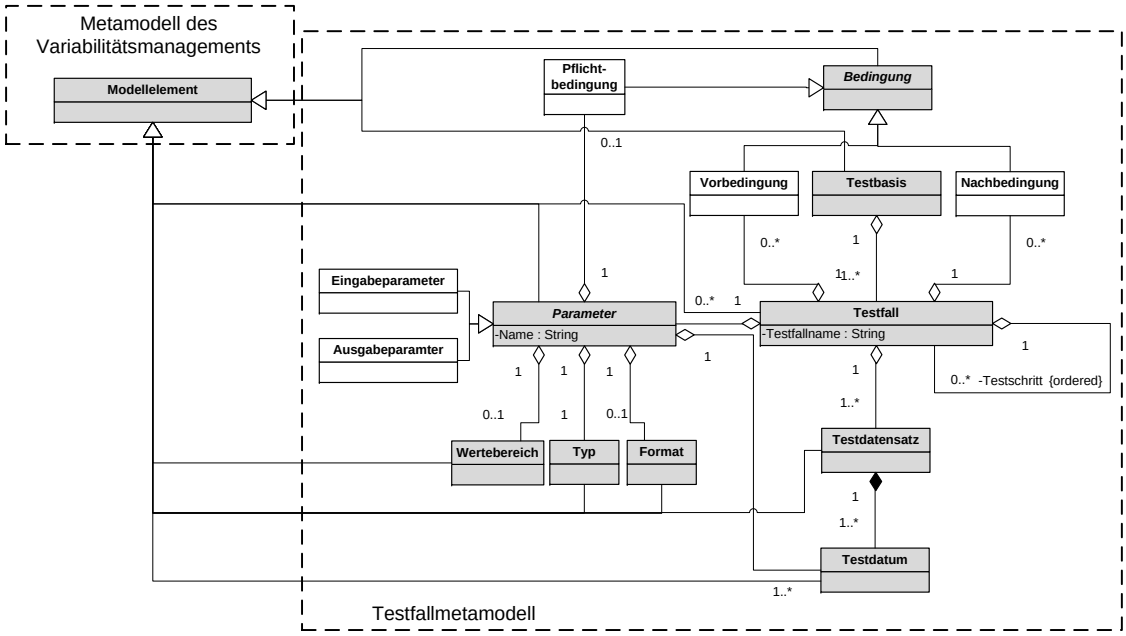


Abbildung 5.13: Erweiterung des Metamodells des Testfalls um Variabilität

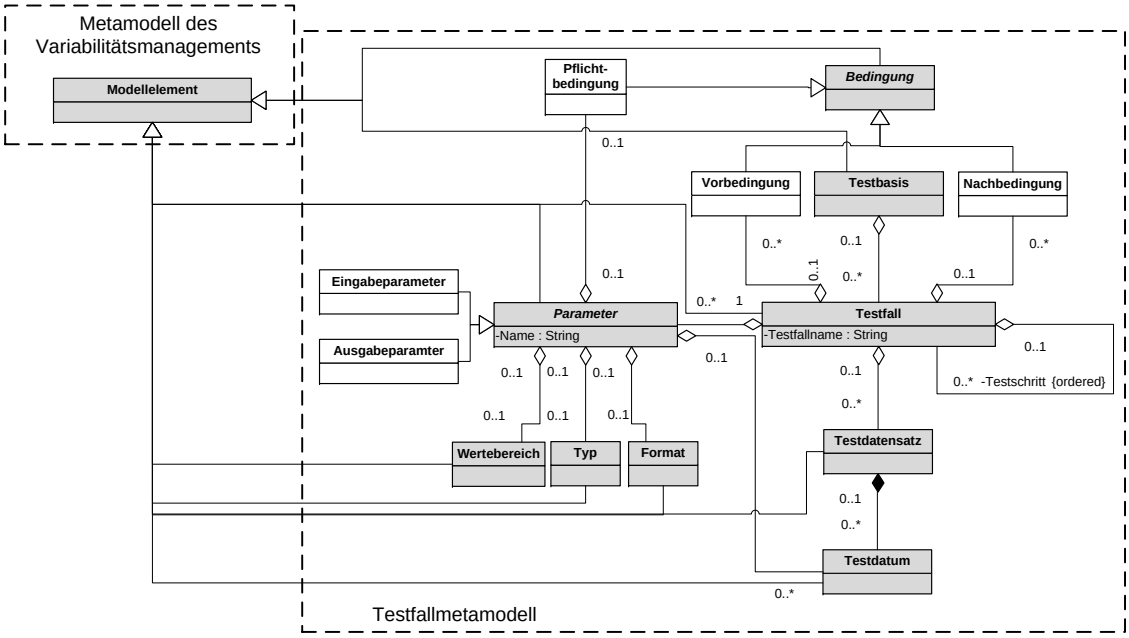


Abbildung 5.14: Lockerung der Kardinalitäten der Assoziationen an erweiterten Metaklassen

Metaklasse	Modifikations-punkte	Additions-punkte	Variante	Maßnahmen
Bedingung	ja	nein	-	(B)
Format	ja	nein	-	(B)
Parameter	ja	ja	-	(A)
Testbasis	ja	ja	-	(A)
Testdatensatz	ja	ja	-	(A)
Testdatum	ja	nein	-	(B)
Testfall	ja	ja	-	(A)
Typ	ja	nein	-	(B)

Tabelle 5.4: Antworten der Entscheidungsbaumfragen für die Metaklassen des Testfallmetamodells

Metaklasse	Als Varianten assoziierbare Metaklassen
Bedingung	-
Format	-
Parameter	Testdatum, Format, Typ, Wertebereich
Testbasis	Testfall
Testdatensatz	Testdatum
Testdatum	-
Testfall	Testschritt, Parameter Vorbedingung, Nachbedingung, Testdatensatz
Typ	-

Tabelle 5.5: Testfallmetamodell: Auswahl der Metaklassen die als Varianten an Additionspunkte assoziierbar sein sollen

5.1.4 Definition einer konkreten Syntax für Anforderungsmodellierungs- und Testfallmodellierungssprache mit Variabilität

Für die Nutzung der im letzten Abschnitt getätigten Erweiterung von Anwendungsfall- und Testfallmetamodell in der Praxis wird in diesem Abschnitt eine konkrete Syntax definiert. Die Modellierung von Variationspunkten und Varianten wird mithilfe

von *Tags* vorgenommen, die den Variationspunkt bzw. die Variante umschließen [BPSM+08]:

Ein Variationspunkt wird durch ein Tag

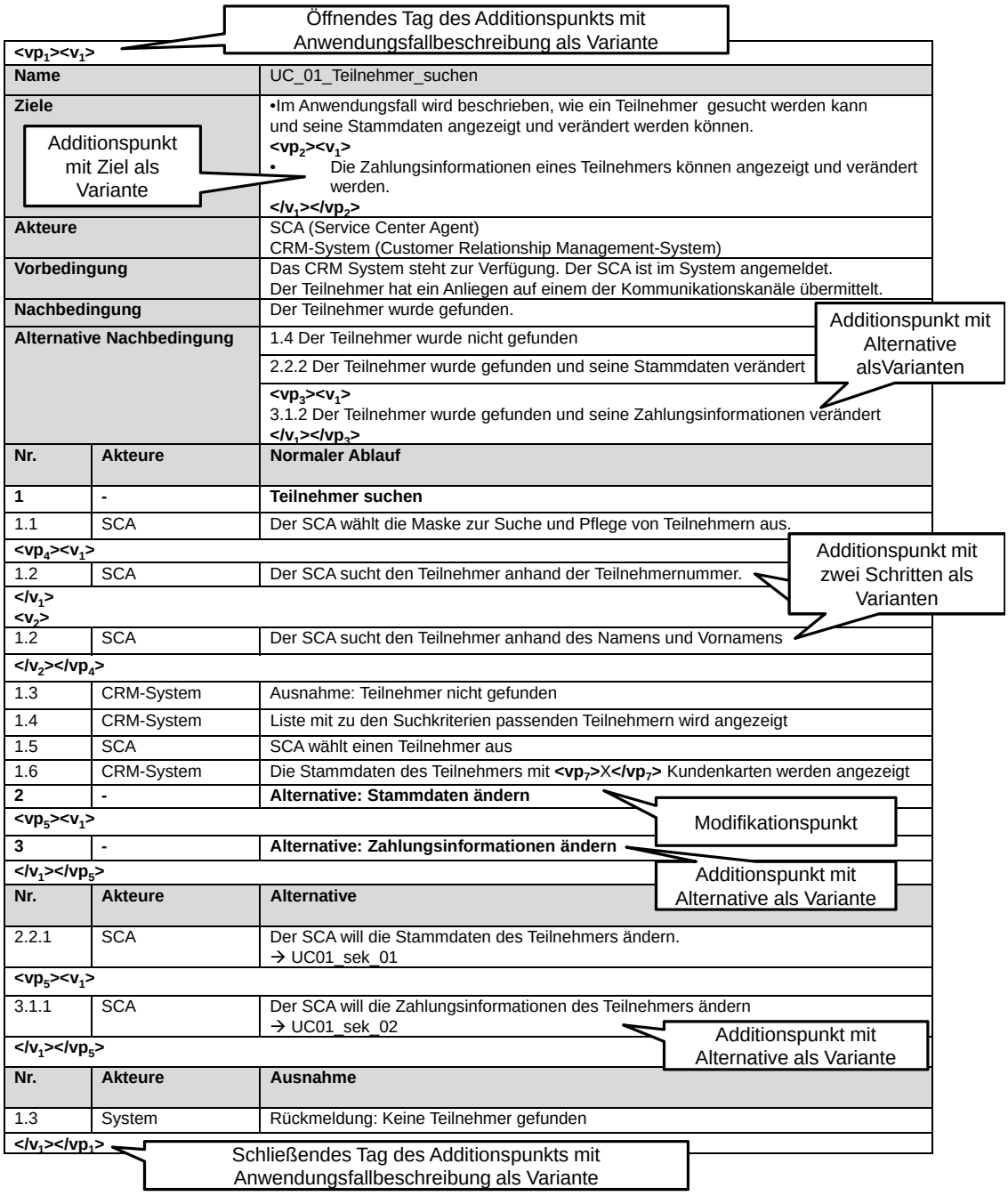
$<vp_x>$ Inhalt des Variationspunktes $</vp_x>$

mit x als eindeutigem Bezeichner eingeschlossen. Die Varianten eines Additionspunktes werden durch Variantentags eingeschlossen:

$<v_x>$ Variante $</v_x>$.

Jede Variante erhält einen eindeutigen Bezeichner x in Bezug auf den Additionspunkt von dem sie abhängt.

5.1.5 Beispiel: Anwendungsfallbeschreibungen mit Variabilität



<vp$_1$><v$_1$>		
Name		UC_01_Teilnehmer_suchen
Ziele		•Im Anwendungsfall wird beschrieben, wie ein Teilnehmer gesucht werden kann und seine Stammdaten angezeigt und verändert werden können. **<vp$_2$><v$_1$>** • Die Zahlungsinformationen eines Teilnehmers können angezeigt und verändert werden. **</v$_1$></vp$_2$>**
Akteure		SCA (Service Center Agent) CRM-System (Customer Relationship Management-System)
Vorbedingung		Das CRM System steht zur Verfügung. Der SCA ist im System angemeldet. Der Teilnehmer hat ein Anliegen auf einem der Kommunikationskanäle übermittelt.
Nachbedingung		Der Teilnehmer wurde gefunden.
Alternative Nachbedingung		1.4 Der Teilnehmer wurde nicht gefunden
		2.2.2 Der Teilnehmer wurde gefunden und seine Stammdaten verändert
		<vp$_3$><v$_1$> 3.1.2 Der Teilnehmer wurde gefunden und seine Zahlungsinformationen verändert **</v$_1$></vp$_3$>**
Nr.	**Akteure**	**Normaler Ablauf**
1	**-**	**Teilnehmer suchen**
1.1	SCA	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
<vp$_4$><v$_1$>		
1.2	SCA	Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.
</v$_1$><v$_2$>		
1.2	SCA	Der SCA sucht den Teilnehmer anhand des Namens und Vornamens
</v$_2$></vp$_4$>		
1.3	CRM-System	Ausnahme: Teilnehmer nicht gefunden
1.4	CRM-System	Liste mit zu den Suchkriterien passenden Teilnehmern wird angezeigt
1.5	SCA	SCA wählt einen Teilnehmer aus
1.6	CRM-System	Die Stammdaten des Teilnehmers mit **<vp$_7$>**X**</vp$_7$>** Kundenkarten werden angezeigt
2	**-**	**Alternative: Stammdaten ändern**
<vp$_5$><v$_1$>		
3	**-**	**Alternative: Zahlungsinformationen ändern**
</v$_1$></vp$_5$>		
Nr.	**Akteure**	**Alternative**
2.2.1	SCA	Der SCA will die Stammdaten des Teilnehmers ändern. → UC01_sek_01
<vp$_5$><v$_1$>		
3.1.1	SCA	Der SCA will die Zahlungsinformationen des Teilnehmers ändern → UC01_sek_02
</v$_1$></vp$_5$>		
Nr.	**Akteure**	**Ausnahme**
1.3	System	Rückmeldung: Keine Teilnehmer gefunden
</v$_1$></vp$_1$>		

Abbildung 5.15: Beispiel: Modellierung von Variabilität in Anwendungsfällen

Mit Hilfe der konkreten Syntax für die Modellierung von Variabilität sind in der Anwendungsfallbeschreibung aus Abbildung 5.15 beispielhaft einige Modellelemente als variabel spezifiziert worden. Zunächst ist die Anwendungsfallbeschreibung

selbst eine Variante am Variationspunkt vp_1. Die Anwendungsfallbeschreibung besitzt weiterhin einen Additionspunkt vp_2, an den ein Ziel als Variante assoziiert ist. Im „Normalen Ablauf“ der Anwendungsfallbeschreibung ist mit vp_4 ein weiterer Additionspunkt vorhanden, welcher zwei Varianten vom Typ Schritt besitzt. In Schritt 1.6 des normalen Ablaufs ist mit vp_7 ein Modifikationspunkt spezifiziert. Der Schritt *Zahlungsinformationen ändern* besitzt weiterhin einen Additionspunkt (vp_5), an den eine Alternative als Variante assoziiert ist. Der Additionspunkt existiert ein weiteres Mal in der Anwendungsfallbeschreibung, um die Schritte der Alternative zu umschließen. Dies ist notwendig, da dass Vorkommen einer Alternative bzw. Ausnahme im normalen Ablauf einer Anwendungsfallbeschreibung von der eigentlichen Spezifikation ihrer Schritte räumlich getrennt ist.

5.1.6 Beispiel: Testfall mit Variabilität

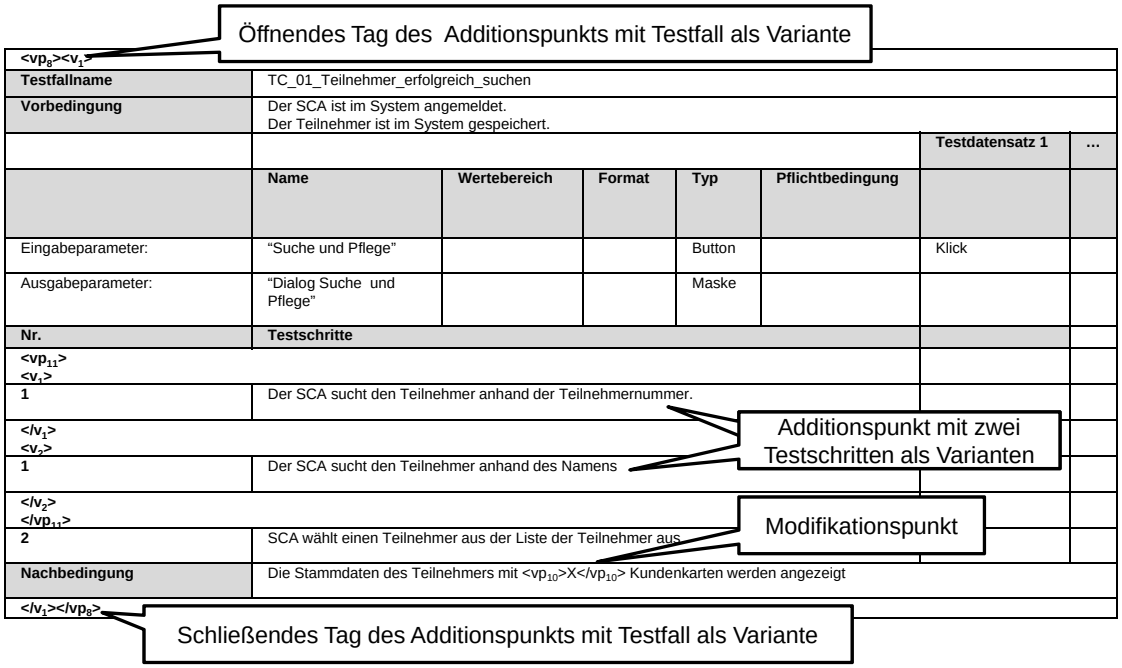


Abbildung 5.16: Beispiel: Modellierung von Variabilität in Testfällen

Für die Beschreibung von Variabilität in Testfällen wird die gleiche konkrete Syntax wie für Anwendungsfallbeschreibungen verwendet. Abbildung 5.16 zeigt den Ausschnitt eines Testfalls. Der Testfall ist selbst Variante am Additionspunkt vp_8.

Weiterhin besitzt der Testfall einen Additionspunkt $vp_1 1$ an dem zwei Testschritte als Varianten assoziiert sind. In der Nachbedingung des Testfalls existiert ein Modifikationspunkt vp_{10}.

5.2 Konstruktion des featurebasierten Variabilitätsmanagements

Der im vergangenen Abschnitt vorgestellte Sprachkonstruktionsprozess ermöglicht die Erweiterung von Modellierungssprachen für die Modellierung von Variabilität. Neben der Modellierung der Variabilität selbst, ist auch die Modellierung von deren Abhängigkeiten untereinander eine Problemstellung (vgl. Abschnitt 3.1.1).
Um diese Problemstellung zu adressieren existiert die Idee, Variabilität und ihre Abhängigkeiten in einem zentralen Modell zu managen [PBL05, SJ03]. Im Folgenden wird daher das Featuremodell als zentrales Modell für das Variabilitätsmanagement eingeführt.

5.2.1 Featuremodell als zentrales Modell für das Variabilitätsmanagement

Das Featuremodell basiert auf der Feature Oriented Domain Analysis (FODA), die in [KCH+90] erstmals vorgestellt wurde. Sie kann im Rahmen der SPL-Entwicklung in der frühen Phase der Plattformanforderungsspezifikation eingesetzt werden. Der Begriff Feature ist in der Literatur leider nicht eindeutig definiert. Im Allgemeinen werden aber unter dem Begriff Feature markante und wichtige Charakteristiken einer Domäne oder eines Software-Systems verstanden [Maß07].
Das Ergebnis der Evaluation aus Kapitel 4 zeigt, dass das *Featuremodell* (FM), die an das Variabilitätsmanagement gestellten Anforderungen am besten erfüllt. Aus diesem Grund wird das Featuremodell für das Management von Variabilität ausgewählt.

Mit Hilfe des Featuremodells können notwendige und variable Features einer Produktlinienplattform, ihre hierarchische Dekomposition und die zwischen Features existierenden Abhängigkeiten modelliert werden (vgl. Abschnitt 4.3.3). Im folgenden Abschnitt wird zunächst eine SPL als laufendes Beispiel in Form eines Featuremodells beschrieben.

5.2.2 SPL Loyaltymanagement als laufendes Beispiel

Um die in den noch folgenden Kapiteln beschriebenen Bausteine des Beitrags zur Lösung der in Abschnitt 3.1.1 definierten Problemstellungen zu illustrieren wird nun ein Beispiel gegeben [Obe09a]. Das Beispiel kommt aus dem Bereich der Kundenbindungsprogramme. Solche Programme dienen zur Steigerung der Kundenbindung und Kundentreue. Häufig sind solche Programme mit einer Kundenkarte (Klub-Karte, Bonuskarte) verknüpft, die jeder Teilnehmer des Programms erhält. Durch Vorlegen der Kundenkarten beim Bezahlen in teilnehmenden Geschäften können die Teilnehmer Rabatte, Sammelpunkte für Sachprämien oder sonstige Vergünstigungen erhalten. Somit wird ein Anreiz für die Teilnehmer geschaffen, die Kundenkarte bei jedem Kauf vorzulegen. Die Anbieter einer Kundenkarte erhalten von jedem neuen Teilnehmer personenbezogenen Daten, wie zum Beispiel den Namen, die Adresse, das Alter oder den Beruf. Beim Vorlegen der Kundenkarte an einer Kasse werden Informationen über den Kaufvorgang des Teilnehmers an den Anbieter der Karte übertragen. Diese Daten können zur Analyse des Kaufverhaltens, zur gezielten Werbung oder anderen Marketingmaßnahmen im Rahmen eines Customer Relationship Managements (CRM) genutzt werden. Als laufendes Beispiel wird die Software-Produktlinie „Loyaltymanagement“ definiert, die ein Kundenbindungsprogramm darstellt.

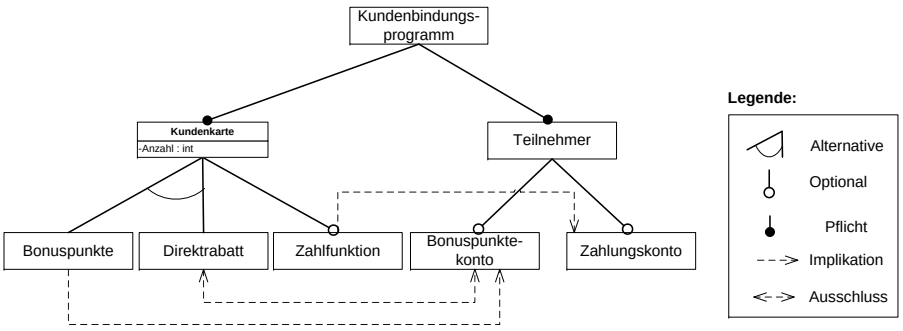


Abbildung 5.17: Beispiel Featuremodell Kundenbindungsprogramm

In Abbildung 5.17 ist das Featuremodells der Beispiel-Produktlinie *Kundenbindungsprogramm* in konkreter Syntax dargestellt, welche in [Maß07] definiert wird. Das Wurzel-Feature *Kundenbindungsprogramm* teilt sich in die zwei Sub-Features *Kundenkarte* und *Teilnehmer* auf, welche beide verpflichtend zu wählen sind. Das Feature Kundenkarte besitzt einen Modifikationspunkt *Anzahl*, welcher die Anzahl der Kundenkarten im Produkt festlegt. Das Feature Teilnehmer besitzt die beiden optionalen Sub-Features *Bonuspunktekonto* und *Zahlungskonto*. Das Feature Kundenkarte besitzt drei Sub-Features: *Bonuspunkte* und *Direktrabatt* bilden eine alternative Auswahl. Das Feature *Zahlfunktion* ist optional. Zwischen den einzelnen Features existieren noch verschiedene Abhängigkeiten.

5.2.3 Abbildungsmodell für die Verbindung von Featuremodell und den übrigen Modellen des Software-Produktlinienentwicklungsprozesses

Durch die Nutzung des Featuremodells als zentralem Modell für das Variabilitätsmanagement wird die Modellierung von Abhängigkeiten auf dieses Modell reduziert. Um nun die Informationen über die Abhängigkeiten zwischen Modellelementen der übrigen Modelle des Software-Produktlinienentwicklungsprozess nicht zu verlieren, muss eine Abbildung zwischen Features und mit Variabilität behafteten Modellelementen hergestellt werden.
Features bieten eine sehr abstrakte Sichtweise auf die Funktionalitäten einer Produktlinienplattform und der darin vorhandenen Variabilität. Die übrigen Modelle mit ihren (mit Variabilität behafteten) Modellelementen konkretisieren diese Funktionalitäten.

Die Konkretisierungsbeziehung zwischen Features und Modellelementen ist beispielhaft in Abbildung 5.18 dargestellt. Diese Beziehung wird in der Literatur auch als Featureinteraktionsproblem bezeichnet:

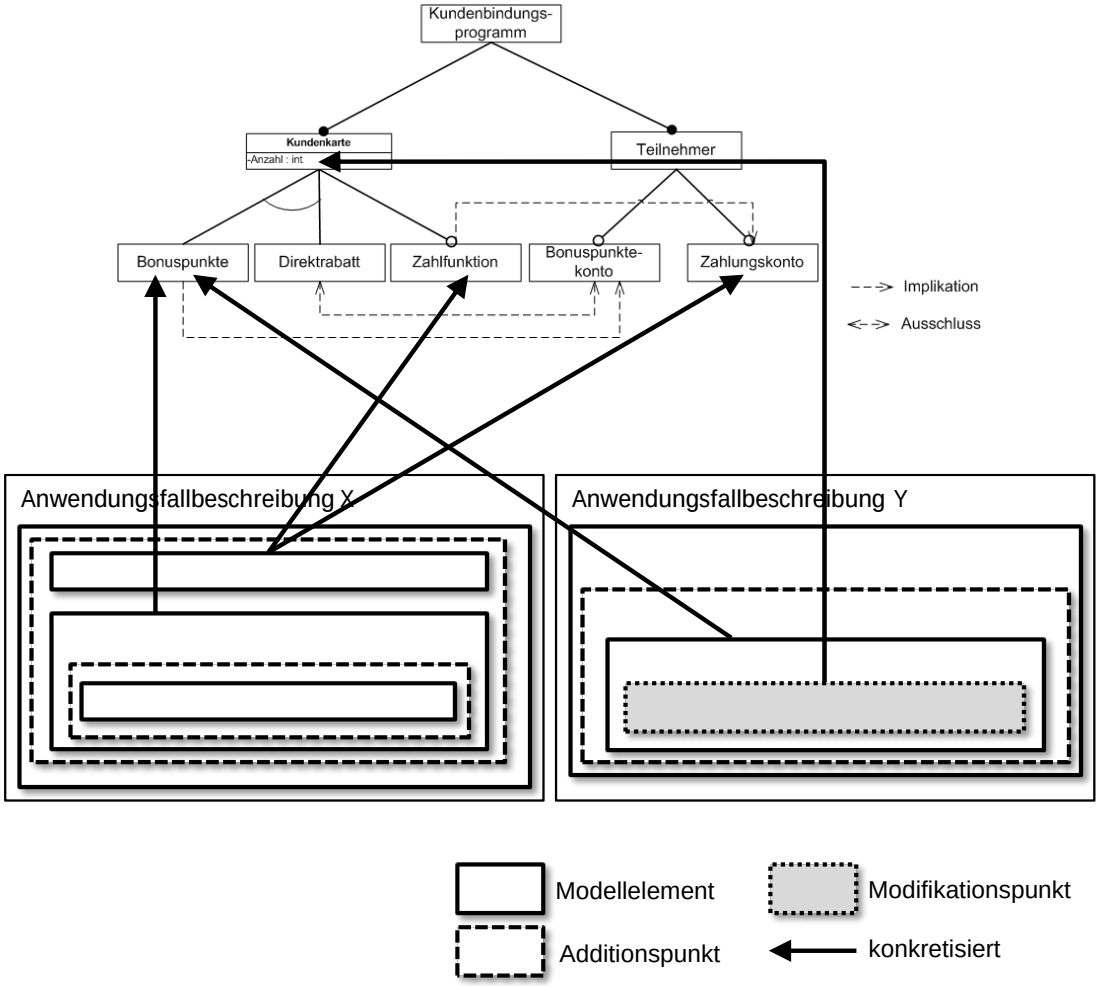


Abbildung 5.18: Beispiel für den Zusammenhang von Features und Modellelementen aus anderen Modellen des Entwicklungsprozesses

> Das Problem besteht darin, dass individuelle Features typischerweise nicht direkt zu einer individuellen Komponente oder einem Cluster von Komponenten (oder auch Modellen, Anmerkung des Autors) verfolgt werden können - das bedeutet, wenn ein Produkt durch die Auswahl einer Gruppe von Features definiert wird, dann ist eine sorgfältig abgestimmte und komplizierte Mixtur an Teilen von verschiedenen Komponenten (Modellen, Anmerkung des Autors) involviert [GAd98].

Ein Feature kann also durch unterschiedliche Modellelemente in verschiedenen Modellen konkretisiert werden (vgl. Feature Bonuspunkte in Abbildung 5.18). Ein

Modellelement kann weiterhin mehrere Features konkretisieren (vgl. Zahlfunktion und Zahlungskonto). Ein Modifikationspunkt in einem Feature kann wiederum durch Modifikationspunkte in Modellelementen konkretisiert werden.

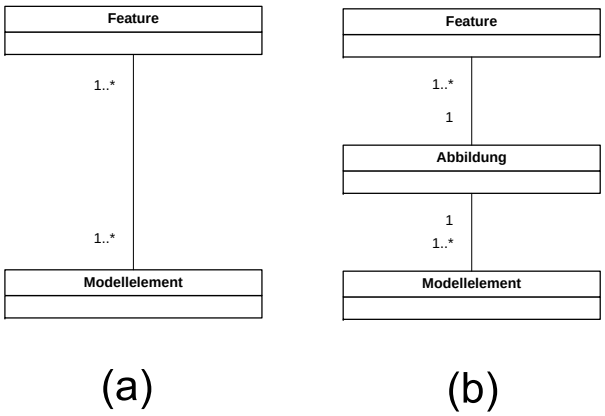


Abbildung 5.19: Zusammenhang von Features und Modellelementen (a) und die Definition der Metaklasse Abbildung (b)

Das Beispiel macht deutlich, dass zwischen Features und Modellelementen eine n-zu-m Beziehung besteht (vgl. Abbildung 5.19.a). Diese Beziehung kann über eine Abbildung aufgelöst werden, wie in Abbildung 5.19.b dargestellt ist.
Durch den Einsatz des Featuremodells als zentralem Modell des Variabilitätsmanagements ist die Ableitung von Produkten aus der Produktlinienplattform über die Bindung der Variationspunkte im Featuremodell möglich. Alle anderen Bindungen werden durch die Abbildung von Featuren auf Modellelemente automatisch vorgenommen.

Um die Abbildung zwischen Features und Modellelementen zu realisieren, wird zwischen dem Metamodell des Featuremodells und dem Metamodell des Variabilitätsmanagements ein *Metamodell des Abbildungsmodells* definiert, welches die Abbildung zwischen Features und Modellelementen beschreibt. Dieser Zusammenhang ist in Abbildung 5.20 dargestellt. Zwei Typen von Abbildungen werden dabei unterschieden:

- Abbildungsimplikation: Abbildung zwischen Features und Modellelementen die Varianten sind.

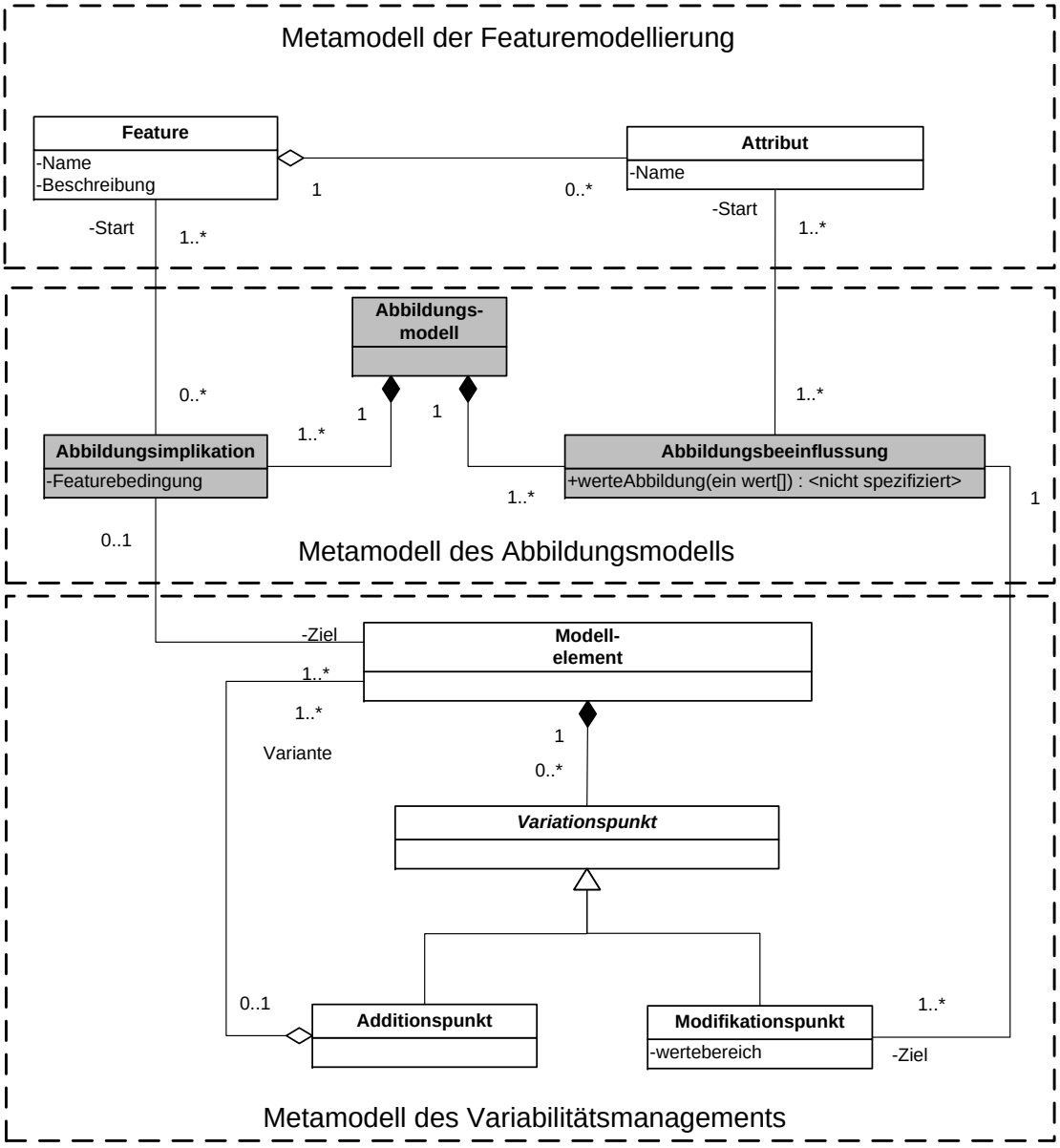


Abbildung 5.20: Metamodell des featurebasierten Variabilitätsmanagements mit Metamodell des Featuremodells, Metamodell des Abbildungsmodells und Metamodell des Variabilitätsmanagements

- Abbildungsbeeinflussung: Abbildung zwischen Featureattributen und Modifikationspunkten die an Modellelementen assoziiert sind.

Beide Typen von Abbildungen werden im Folgenden erläutert.

Abbildungsimplikation

Eine Abbildungsimplikation ist eine Abbildung zwischen mindestens einem Feature und mindestens einem Modellelement. Features werden als Featurebedingung in der Abbildungsimplikation definiert. Die Featurebedingung spezifiziert die Bindung, d. h. die Aus- oder Abwahl eines Features und die Verknüpfungen zwischen Features mit Hilfe von Konjunktion und Disjunktion. Die Sprache für Featurebedingungen ist als *kontextfreie Grammatik* G_{fc} definiert [RP06]:

$$G_{fc} = (\{C\}, \{\neg, \wedge, \vee, f\}, \{C \rightarrow f, C \rightarrow \neg f, C \rightarrow C \wedge C, C \rightarrow C \vee C\})$$

Das Nicht-Terminal C steht hierbei stellvertretend für *Condition* (Bedingung) und das Terminal f für ein Feature aus dem Featuremodell. Die Featurebedingung gibt an, unter welcher Bedingung die von dieser Abbildungsimplikation abhängigen Modellelemente gebunden werden. Eine Featurebedingung wird mithilfe der eindeutigen Featurebezeichnungen aus dem Featuremodell und den aussagenlogischen Operatoren $\wedge$ für die Konjunktion, $\vee$ für die Disjunktion und $\neg$ für die Negation gebildet. Die Konjunktion hat Vorrang vor der Disjunktion. Ein Feature kann Teil beliebig vieler Featurebedingungen sein und eine Featurebedingung kann beliebig viele Features enthalten (vgl. Abbildung 5.20). Gültige Featurebedingungen sind zum Beispiel: $feature_1$, $\neg feature_1$ oder $feature_1 \wedge \neg feature_2 \vee feature_3$.

Jedes Modellelement hängt genau dann von einer Featurebedingung ab, wenn es als Variante an einem Additionspunkt assoziiert ist:

Bedingung 20. ***Context*** *Additionspunkt* ***inv:***
*self.Variante→**forAll**(v:Variante|v.Abbildungsimplikation*
→notEmpty()))

Abbildungsbeeinflussung

Die *Abbildungsbeeinflussung* bildet mindestens ein Attribut eines Features auf mindestens einen Modifikationspunkt in einem Modellelement ab. Die Werte, die an Attribute der Features gebunden werden, bestimmen den Wert, der in Modifikationspunkten von Modellelementen gebunden wird. Die Ausprägung des Wertes wird

über die Operation *werteAbbildung(ein wert[]):<nicht-spezifiziert>* bestimmt. Die Operation erhält als Eingabe die Menge der Werte der Attribute der Features und bestimmt den Wert der an den assoziierten Modifikationspunkten der Modellelemente gebunden wird.

Konkrete Syntax für das Abbildungsmodell

Für das Abbildungsmodell kann eine konkrete Syntax in unterschiedlicher Form definiert werden. In Abbildung 5.21 ist das Abbildungsmodell beispielhaft in Form von zwei Tabellen für Abbildungsimplikation und Abbildungsbeeinflussung mit jeweils drei Spalten dargestellt. Die Abbildungsimplikation besitzt die Spalten *Bindungskonfiguration*, *Featurebedingung* und *Modellelemente*. Die Abbildungsbeeinflussung hat die Spalten *Bindungskonfiguration*, *Werteabbildung* und *Modifikationspunkte*. Die Spalten haben folgende Bedeutungen:

Bindungskonfiguration

Die *Bindungskonfiguration* stellt einen eindeutigen Bezeichner für eine Abbildungsimplikation oder -beeinflussung im Abbildungsmodell dar. Jedes Modellelement, das an einem Additionspunkt assoziiert ist, besitzt genau eine Bindungskonfiguration, die die Assoziation zur Abbildungsimplikation realisiert. Jeder Modifikationspunkt in einem Modellelement hat eine Bindungskonfiguration, die angibt an welcher Abbildungsbeeinflussung er assoziiert ist. Die in Abschnitt 5.1.4 eingeführte konkrete Syntax für die Variabilitätsmodellierung in Anwendungsfallbeschreibungen und Testfällen wird dahingehend erweitert:

Ein Modifikationspunkt wird durch ein Tag

$<vp_x|BK_y>$ Inhalt des Variationspunktes $</vp_x>$

mit x als eindeutigem Bezeichner und BK_y als Bindungskonfiguration eingeschlossen. Die Varianten eines Additionspunktes werden durch Variantentags eingeschlossen, die eine Bindungskonfiguration BK_y besitzen:

$<v_x|BK_y>$ Variante $</v_x>$.

Featurebedingung

Die Featurebedingung wird auf Basis der in diesem Abschnitt definierten Grammatik G_{fc} spezifiziert und beschreibt die Assoziation zwischen Abbildungsimplikation und Features des Featuremodells.

Modellelemente

Die Modellelemente einer Abbildungsimplikation realisieren die Assoziation zwischen Abbildungsimplikation und den assoziierten Modellelementen. Ein Modellelement wird durch den eindeutigen Bezeichner des Variationspunktes ($vpID$) an dem es assoziiert ist, und dem Bezeichner des Modellelements selbst (vID), welcher wiederum für einen Variationspunkt eindeutig ist, referenziert: $vpID(vID)$.

Werteabbildung

Die Werteabbildung realisiert die Operation *werteAbbildung(ein wert[]):<nicht-spezifiziert>* aus der Metaklasse Abbildungsbeeinflussung. Die Abbildung definiert das Bildungsgesetz für jeden Eingabewert aus den Modifikationspunkten der Features. Die Ausgabe wird an die assoziierten Modifikationspunkte der Modellelemente gebunden.

Modifikationspunkte

Modifikationspunkte der Abbildungsbeeinflussung spezifizieren die Referenzen zu den assoziierten Modifikationspunkten. Die Referenz besteht aus der eindeutigen Bezeichnung des Modifikationspunktes.

5.2.4 Feature- und Abbildungsmodell im Beispiel

Ein Beispiel für Featuremodell, Abbildungsmodell und Modelle des Software-Produktlinienentwicklungsprozess illustriert Abbildung 5.21. Die Abbildungsimplikation mit der Bindungskonfiguration BK_1 besitzt eine Featurebedingung, bestehend aus der Auswahl des Features *Direktrabatt* und der gleichzeitigen Abwahl des Features *Zahlfunktion*. In der Spalte Modellelemente ist ein Modellelement eingetragen. Dieses ist in $Modell_1$ am Variationspunkt vp_1 assoziiert und trägt selbst den Bezeichner v_1. In $Modell_1$ existiert also ein Modellelement mit diesem Additionspunkt vp_1. Darin

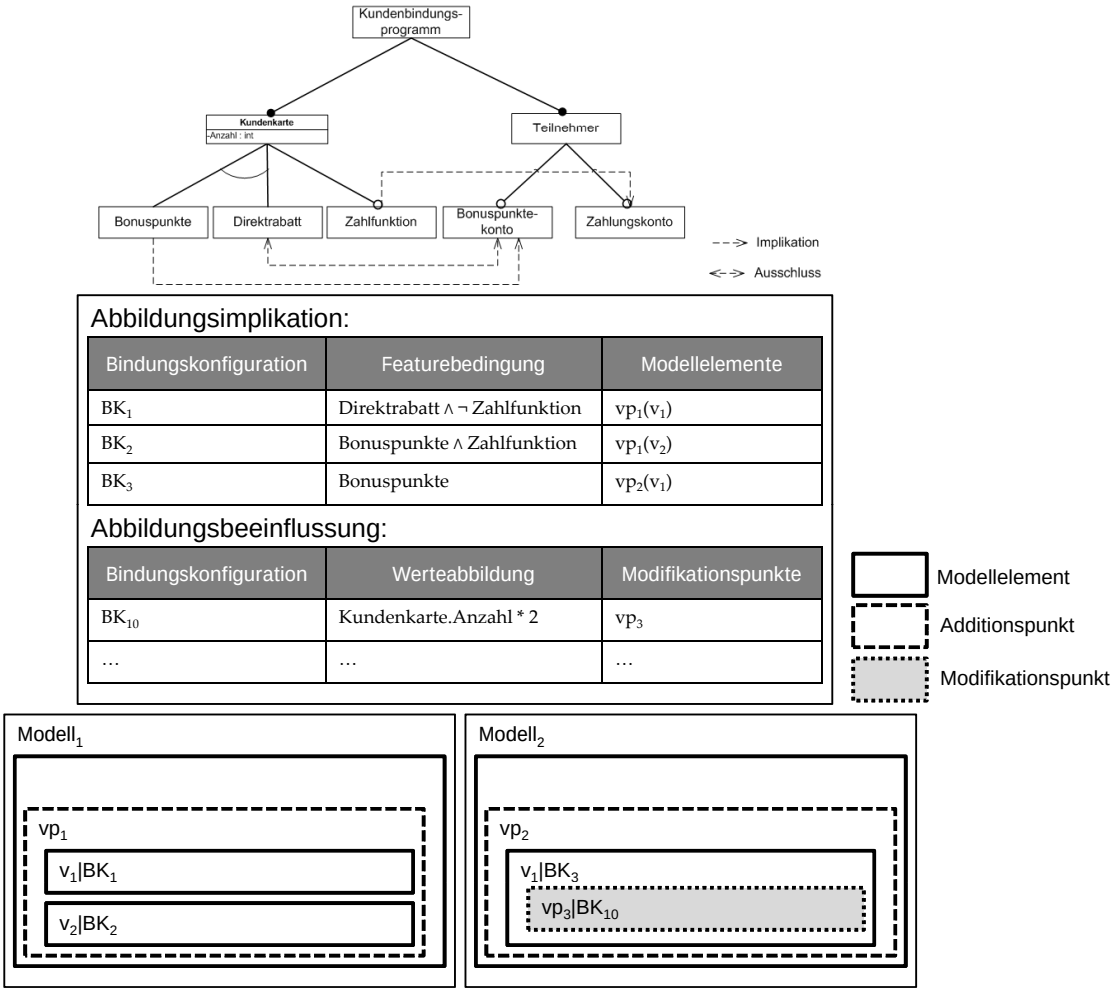


Abbildung 5.21: Beispiel: Spezifikation von Abbildungen zwischen Features und Modellelementen mit Hilfe des Abbildungsmodells

sind zwei Modellelemente als Varianten vorhanden. Die in der Abbildungsimplikation BK_1 angegebene Variante $vp_1(v_1)$ ist eine der beiden Varianten und besitzt daher die Bindungskonfiguration BK_1. Die andere Variante im Additionspunkt hat die Bindungskonfiguration BK_2. Diese Bindungskonfiguration ist wiederum eine Abbildungsimplikation mit der Featurebedingung *Bonuspunkte* $\wedge$ *Zahlfunktion*. Unter Varianten dieser Bindungskonfiguration ist das Modellelement mit $vp_1(v_2)$ angegeben. In $Modell_2$ ist ein Modellelement vorhanden, welches einen Additionspunkt mit dem Bezeichner vp_2 besitzt. Darin ist eine Variante vorhanden, welche an Bindungskonfiguration BK_3 assoziiert ist. Diese Bindungskonfiguration hat als Feature-

bedingung *Bonuspunkte*. Innerhalb des Modellelements $vp_2(v_1)$ existiert ein weiterer Variationspunkt. Hierbei handelt es sich um einen Modifikationspunkt, der an die Abbildungsbeeinflussung BK_{10} assoziiert ist. Die Abbildungsbeeinflussung besitzt eine Werteabbildung mit der Abbildungsvorschrift $Kundenkarte.Anzahl * 2$. In der Abbildungsvorschrift wird der Modifikationspunkt *Anzahl* des Features *Kundenkarte* als Eingabe genutzt.

5.3 Zusammenfassung

Gegenstand dieses Kapitels waren zwei zentrale Beiträge dieser Arbeit:
Zunächst wurde ein Sprachkonstruktionsprozess für die Erweiterung von Modellierungssprachen um Variabilität definiert (vgl. Abschnitt 5.1). Mit diesem Vorgehen ist es möglich in allen Typen von Modellen, die im Rahmen des Software-Produktlinienentwicklungsprozesses genutzt werden, Variabilität zu modellieren. Da das Vorgehen das Metamodell des Variabilitätsmanagement instrumentalisiert, erfüllen die erweiterten Modellierungssprachen alle in Kapitel 4 definierten Anforderungen an die Modellierung von Variabilität. Somit ist die Beschreibung der Veränderlichkeit und schließlich die Wiederverwendung vom Modellen für unterschiedliche Produkte einer SPL möglich. Als Anwendungsbeispiel für das Vorgehen dienten die Modellierungssprachen von Anforderungsbeschreibungen und Testfällen.
Der zweite Beitrag dieses Kapitels betrifft das Variabilitätsmanagement (vgl. Abschnitt 5.2). Es existiert in diesem Kontext die Anforderung der Modellierung der Variabilität und ihrer Abhängigkeiten in einem zentralen Modell, da die Modellierung der Abhängigkeiten innerhalb und zwischen den Modellen des Software-Produktlinienentwicklungsprozesses schnell unüberschaubar wird. Um dieses Problem zu adressieren, wurde das Featuremodell als zentrales Modell des Variabilitätsmanagement ausgewählt und eine Abbildung zwischen diesem und den Modellelementen der Modelle des Software-Produktlinienentwicklungsprozesses definiert.

Durch dieses *featurebasierte Variabilitätsmanagement* ist es nun möglich Abhängigkeiten zwischen Varianten und Modifikationspunkten zentral im Featuremodell zu spezifizieren und dieses Modell für die Ableitung von Produkten aus der Plattform zu nutzen. Die Variabilität in den übrigen Modellen der Plattform kann über das Abbildungsmodell automatisch gebunden werden.
Durch den hier gewählten Ansatz kann sein Nutzer die Features und die Variabilität der Plattform in einem zentralen Modell managen.

Kapitel 6

Qualitätssicherung

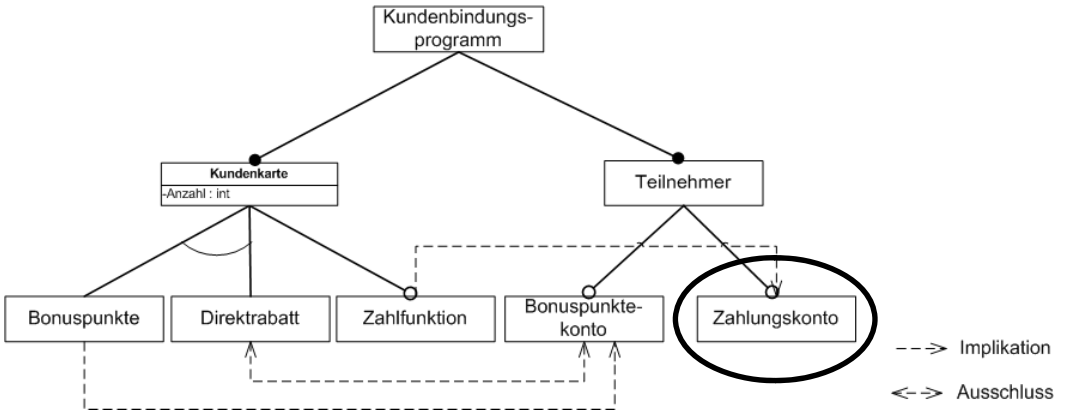
Im letzten Kapitel wurde zum einen die Konstruktion von *Variabilitätsmodellierungssprachen* und zum anderen ein Ansatz für das zentrale Management von Variabilität mithilfe des *featurebasierten Variabilitätsmanagement* beschrieben. Durch die Erweiterung von Modellierungssprachen um Variabilität zu Variabilitätsmodellierungssprachen ist es für den Nutzer möglich, die Veränderbarkeit von Modellen des Software-Produktlinienentwicklungsprozesses zu beschreiben.
Diese Möglichkeit kann bei Ihrer Nutzung zu neuen Arten von Fehlern führen, welche während der Modellierung von Variabilität in Modellen passieren und so zum Beispiel die Ableitung von fehlerfreien Produkten verhindern können.
In diesem Kapitel werden daher *Maßnahmen* definiert, die zum einen die Modellierung von Variabilität in den Modellen des Software-Produktlinienentwicklungsprozesses und zum anderen die Abbildung zwischen diesen Modellen und dem Featuremodell auf Fehler überprüfen.

In diesem Kapitel wird für jede Maßnahme zunächst ein *Fehlermodell* beschrieben [KV03], welches die Art der möglichen Fehler definiert. Im Anschluss daran wird das Vorgehen zur Identifikation der Fehler beschrieben.
Zunächst werden die Maßnahmen für Qualitätssicherung der Abbildung zwischen Featuremodell und Modellen des Software-Produktlinienentwicklungsprozesses beschrieben. Im Anschluss daran ist die Identifikation von Fehlern in der Modellierung von Variabilität in den Modellen des Software-Produktlinienentwicklungsprozesses Gegenstand der Betrachtung.

6.1 Identifikation von Fehlern im featurebasierten Variabilitätsmanagement

In diesem Abschnitt werden mögliche Fehler bei der Spezifikation des featurebasierten Variabilitätsmanagements in Form von Fehlermodellen definiert und anschließend deren algorithmische Identifikation und Behebung beschrieben.

6.1.1 Variable Features ohne Abbildung zu Modellelementen



Abbildungsimplikation:

Bindungskonfiguration	Featurebedingung	Modellelemente
BK_1	Direktrabatt ∧ ¬ Zahlfunktion	$vp_1(v_1)$
BK_2	Bonuspunkte ∧ Zahlfunktion	$vp_1(v_2)$
BK_3	Bonuspunkte	$vp_2(v_1)$

Abbildungsbeeinflussung:

Bindungskonfiguration	Werteabbildung	Modifikationspunkte
BK_{10}	Kundenkarte.Anzahl * 2	vp_3
…	…	…

Abbildung 6.1: Beispiel: Feature *Zahlungskonto* ohne Abbildung auf Modellelemente

Abbildung 6.1 zeigt als Beispiel ein Feature- und ein Abbildungsmodell. Dabei

wird das optionale Feature *Zahlungskonto* in keiner Featurebedingung genutzt, also auf kein Modellelement abgebildet. Das bedeutet, dass die Aus- oder Abwahl eines solchen Features, während der Ableitung eines Produktes aus der Plattform, keine Auswirkung auf die funktionalen und nicht-funktionalen Eigenschaften des Produktes hat.

Fehlermodell

Bei variablen Features, die nicht Teil einer Abbildung sind, können zwei unterschiedliche Fehler vorliegen, die in Tabelle 6.1 beschrieben werden.

Fehler	**Begründung**	**Auswirkung**
In der Plattform fehlen Modellelemente, die das Feature konkretisieren.	Das Feature steht stellvertretend für funktionale oder nicht-funktionale Eigenschaften, die von einigen Nutzern von Produkten dieser Produktlinie gewünscht sind.	Der Nutzer wählt das Feature während der Ableitung eines Produktes aus, die gewünschte Eigenschaft ist im Produkt aber nicht enthalten. Die Erwartung des Nutzers wird nicht erfüllt.
Das Feature ist überflüssig.	Das Feature wird nicht durch Modellelemente konkretisiert, da die mit dem Feature spezifizierten funktionalen oder nicht-funktionalen Eigenschaften nicht benötigt werden.	Das Feature wird niemals ausgewählt, stellt aber eine Inkonsistenz zwischen Featuremodell und den übrigen Modellen des Software-Produktlinienentwicklungsprozess dar.

Tabelle 6.1: Fehlermodell für variable Features ohne Abbildung

Im Folgenden wird nun ein Algorithmus zur Identifikation und Behebung der definierten Fehler beschrieben.

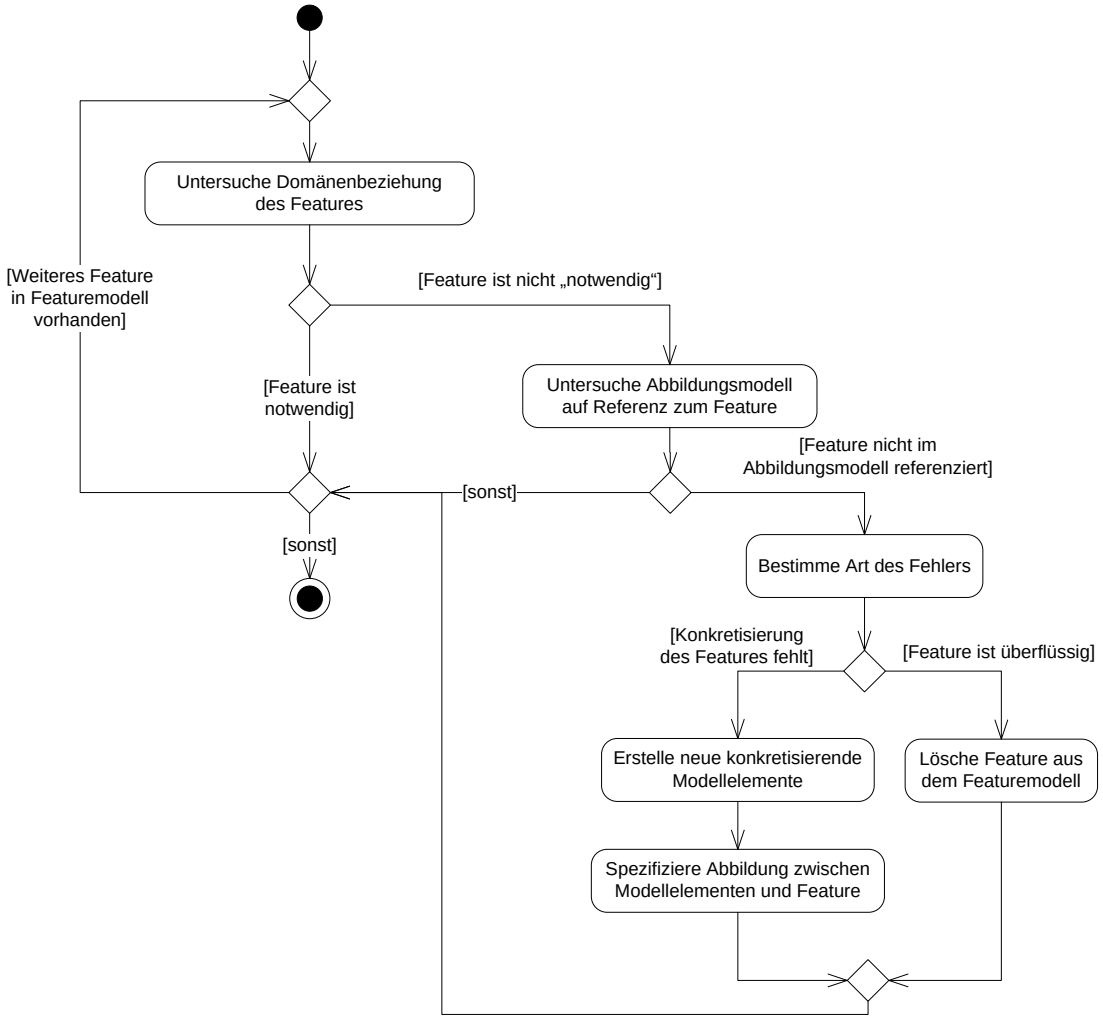


Abbildung 6.2: Algorithmus: Identifikation und Behebung der Fehler bei variablen Features ohne Abbildung zu Modellelementen

Identifikation und Behebung von Fehlern

Die Identifikation und Behebung der beiden im Fehlermodell definierten Fehler ist in Abbildung 6.2 als Algorithmus in Form eines UML-Aktivitätsdiagramm dargestellt: Der Algorithmus durchläuft alle Features des Featuremodells. Für jedes Feature wird festgestellt, ob seine Domänenbeziehung, d. h. die Beziehung zum Vaterfeature vom Typ *notwendig* ist. Ist dies der Fall, muss das Feature nicht weiter untersucht werden, da es nicht variabel ist und damit keine Abbildung auf variable Modellelemente besitzen kann. Besitzt das Feature aber eine variable Domänenbeziehung zu

seinem Vaterfeature wird im zweiten Schritt überprüft, ob es im Abbildungsmodell referenziert ist. Das Feature ist genau dann referenziert, wenn es Teil mindestens einer Featurebedingung ist.
Falls es nicht Teil einer Featurebedingung ist, liegt ein Fehler vor. Die Bestimmung der Art des Fehlers muss manuell durchgeführt werden. Falls das Feature überflüssig ist, wird es gemeinsam mit allen daran referenzierten Abhängigkeiten aus dem Featuremodell gelöscht. Besitzt das zu löschende Feature Kindfeatures, erhalten diese das Vaterfeature des zu löschenden Features als neues Vaterfeature.
Falls das Feature nicht überflüssig ist, müssen neue Modellelemente in den übrigen Modellen der Plattform spezifiziert werden, die das Feature konkretisieren und anschließend eine Abbildung zwischen diesen neuen Modellelementen und dem Feature über das Abbildungsmodell spezifiziert werden.

6.1.2 Kontradiktion in einer Featurebedingung

Die Spezifikation von Abbildungsimplikationen zwischen Features und Modellelementen der übrigen Modelle der Plattform ist eine manuelle Tätigkeit bei der Fehler passieren können.

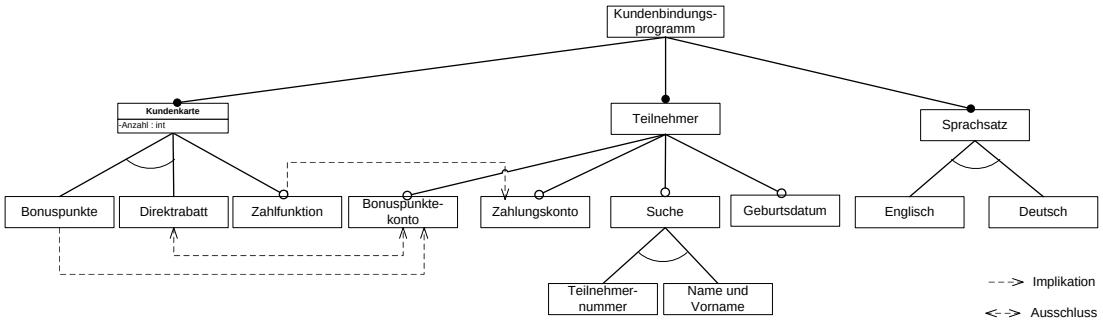


Abbildung 6.3: Beispielfeaturemodell für die Kontradiktion

In Abbildung 6.3 ist ein Beispielfeaturemodell dargestellt. Auf Basis dieses Modells wird nun folgende Featurebedingung für eine Abbildungsimplikation definiert:

Featurebedingung: $Direktrabatt \wedge Bonuspunktekonto$

Die Featurebedingung ist ein gültiges Wort der Grammatik G_{fc}, welche die Sprache der Featurebedingungen definiert (vgl. Abschnitt 5.2.3). Die Featurebedingung

ist erfüllt, wenn die beiden Features *Direktrabatt* und *Bonuspunktekonto* gemeinsam gewählt werden. Zwischen beiden Features ist im Featuremodell aus Abbildung 6.3 ein Ausschluss modelliert, sodass beide Features niemals zusammen gewählt werden können. Die Featurebedingung ist somit eine *Kontradiktion*, also mit anderen Worten die Konjunktion zweier widersprüchlicher Features [Hor06]. Jedes Modellelement, welches an der Abbildungsimplikation mit dieser Featurebedingung assoziiert ist, wird folglich niemals für ein Produkt der SPL ausgewählt werden.
Im Folgenden wird zunächst das Fehlermodell für diese Art von Fehler und im Anschluss daran die Identifikation der Fehler beschrieben.

Fehlermodell

Fehler	**Begründung**	**Auswirkung**
Die Featurebedingung ist eine Kontradiktion.	Die mittels Konjunktion verbundenen Features besitzen Domänen- oder Abhängigkeitsbeziehungen, die zu einem Widerspruch in der Featurebedingung führen.	Die Modellelemente der Abbildungsimplikation können niemals für Produkte ausgewählt werden.

Tabelle 6.2: Fehlermodell für eine Kontradiktion in einer Featurebedingung

Das Fehlermodell für die Kontradiktion bei Featurebedingungen ist in Tabelle 6.2 definiert. Für den im Fehlermodell definierten Fehler wird nun ein Algorithmus für dessen Identifikation vorgestellt.

Identifikation und Behebung von Fehlern

Die bei Featurebedingungen möglichen verschiedenen Aussagetypen sind in Tabelle 6.3 dargestellt. Eine Kontradiktion innerhalb einer Featurebedingung kann nur zwischen Features entstehen, die durch Konjunktion verbunden sind (Aussagetypen 1 bis 4). Eine Ausnahme bildet dabei die Konjunktion von ausschließlich negierten Features, da diese Aussage logisch äquivalent zur Negation der Disjunktion dieser Features ist (Aussagentyp 4). Für Aussagentyp 3 kann ohne Betrachtung des Featuremodells eine Kontradiktion nachgewiesen werden, sie ist in jedem Fall sicher. Für

Nr.	Aussagentyp	Analyse
1.	$A \wedge B$	Kontradiktion ist möglich, falls zum Beispiel Feature A und Feature B im Featuremodell eine Ausschlussbeziehung besitzen.
2.	$\neg A \wedge B$	Kontradiktion ist möglich, falls zum Beispiel Feature B Feature A impliziert.
3.	$A \wedge \neg A$	Kontradiktion ist sicher, da das Feature nicht gleichzeitig gewählt und abgewählt sein kann.
4.	$\neg A \wedge \neg B$	Kontradiktion ist unmöglich, da logisch äquivalent zu $\neg(A \vee B)$.
5.	$A \vee B$	Kontradiktion ist unmöglich, da keine Konjunktion vorhanden.
6.	$\neg A \vee B$	Kontradiktion ist unmöglich, da keine Konjunktion vorhanden.
7.	$\neg A \vee \neg B$	Kontradiktion ist unmöglich, da keine Konjunktion vorhanden.

Tabelle 6.3: Analyse der Aussagetypen

eine Kontradiktion in den Aussagetypen 1 und 2 muss das Featuremodell bestimmte Eigenschaften erfüllen, die im Folgenden betrachtet werden:

Identifikation der Kontradiktion des Aussagetyps $A \wedge B$:

Es können zwei Fälle identifiziert werden, bei der eine Featurebedingung, die den Aussagetyp $A \wedge B$ enthält, eine Kontradiktion darstellt. Diese werden in Abbildung 6.4 visualisiert:
In jedem der beiden Fälle sind die beiden Features A und B über zwei unterschiedliche Domänenpfade mit einem gemeinsamen *Vaterfeature* verbunden. Auf dem Domänenpfad zwischen dem Vaterfeature und dem jeweiligen Feature A und B können beliebig viele weitere Features spezifiziert sein.
Im ersten Fall (a) ist mindestens ein direkter oder indirekter Ausschluss zwischen zwei Featuren der beiden Domänenpfade spezifiziert. Indirekt bedeutet, dass beliebig viele weitere Ausschlüsse zwischen Features existieren können, die einen Ausschlusspfad zwischen zwei Features der Domänenpfade von Feature A und B definieren.

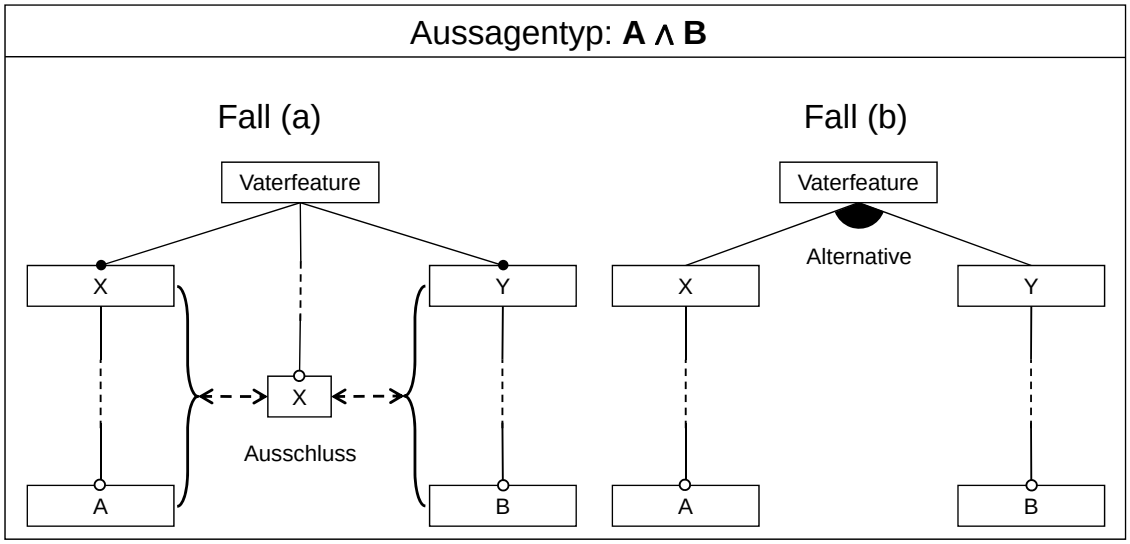


Abbildung 6.4: Kontradiktion des Aussagetyps $A \wedge B$ aufgrund von Ausschluss oder Alternative

Im Beispiel aus Abbildung 6.4.a ist dies der Ausschlusspfad $B \longleftrightarrow X \longleftrightarrow A$. Dies macht eine gemeinsame Auswahl der beiden Features A und B im Rahmen einer Ableitung unmöglich. Somit stellt die Bedingung $A \wedge B$ eine Kontradiktion dar.
Im Fall (b) sind die beiden Domänenpfade unterhalb des Vaterfeatures als „Alternative“ spezifiziert. Auch in diesem Fall ist die Featurebedingung eine Kontradiktion, da immer nur einer der beiden Domänenpfade während der Ableitung eines Produktes gewählt werden kann.

Abbildung 6.5 zeigt den Algorithmus zur Überprüfung von Featurebedingungen auf Kontradiktion als UML-Aktivitätsdiagramm.

Für jede Featurebedingung werden die Teilbedingungen überprüft, deren Features ausschließlich durch Konjunktionen verbunden sind. Eine Teilbedingung einer Featurebedingung ist jede durch eine Disjunktion separierte Folge von durch Konjunktionen verbundene Features. Die Features einer solchen Teilbedingung werden dann paarweise überprüft, um eine Kontradiktion der Featurebedingung ausschließen zu können.
Zunächst werden die beiden Domänenpfade der betrachteten Features bestimmt. Im Anschluss daran wird überprüft, ob die beiden Pfade am gemeinsamen Vaterfeature die Domänenbeziehung Alternative besitzen. Ist dies der Fall, liegt eine Kontradik-

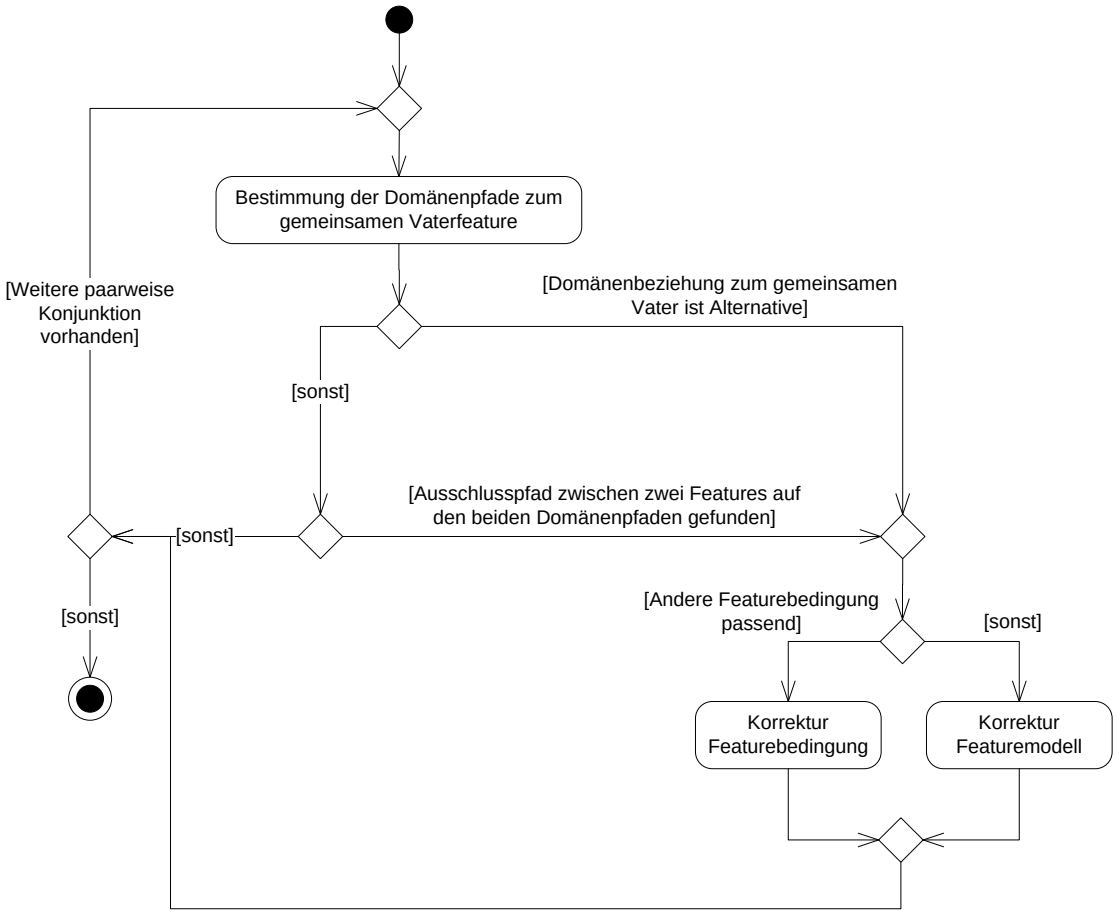


Abbildung 6.5: Algorithmus: Identifikation und Behebung der Kontradiktion in Featurebedingungen des Aussagetyps $A \wedge B$

tion vor. Liegt keine Alternative vor, wird überprüft, ob es einen Ausschlusspfad zwischen einem Feature auf dem Domänenpfad von Feature B und A gibt. Ist dies der Fall liegt eine Kontradiktion vor. Ist dies nicht der Fall, wird die nächste paarweise Konjunktion von Features aus der Featurebedingung überprüft.

Liegt eine Kontradiktion vor, muss manuell überprüft werden, wie dieser Fehler gelöst werden soll. Die Featurebedingung der Abbildungsimplikation kann verändert werden, falls eine passende Featurebedingung definiert werden kann. Ist dies nicht möglich, muss das Featuremodell verändert werden.

Identifikation der Kontradiktion des Aussagetyps $\neg A \wedge B$:

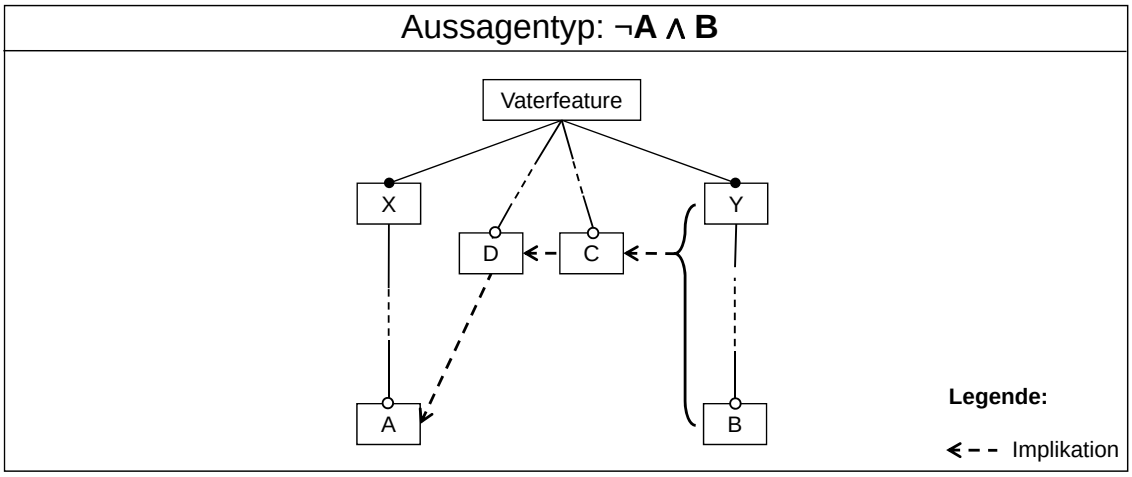


Abbildung 6.6: Kontradiktion des Aussagetyps $\neg A \wedge B$ aufgrund einer Implikation

Für den Aussagentyp $\neg A \wedge B$ existiert ein Fall, der zu einer Kontradiktion der Aussage führen kann, wie in Abbildung 6.6 dargestellt ist. Auch in diesem Fall sind die beiden Features A und B über zwei unterschiedliche Domänenpfade mit einem gemeinsamen *Vaterfeature* verbunden. Auf dem Domänenpfad zwischen dem Vaterfeature und dem jeweiligen Feature A und B können beliebig viele weitere Features spezifiziert sein.
Ein Feature auf dem Domänenpfad des Features B bis zum gemeinsamen Vaterfeature Feature A, impliziert direkt oder indirekt das Feature A. Indirekt bedeutet, dass beliebig viele weitere Implikationen zwischen Features existieren können, die einen Implikationspfad zwischen Feature A und B definieren. Im Beispiel aus Abbildung 6.6 ist dies der Implikationspfad $B \rightarrow C \rightarrow D \rightarrow A$.

Existiert ein solcher sogenannter Implikationspfad, kann die Featurebedingung nicht erfüllt werden, da die Auswahl des Features B die Auswahl des Features A impliziert. Die Featurebedingung sieht aber die Auswahl von Feature B gemeinsam mit der Abwahl von Feature A vor.
Abbildung 6.7 zeigt den Algorithmus zur Überprüfung von Featurebedingungen auf Kontradiktion als UML-Aktivitätsdiagramm.

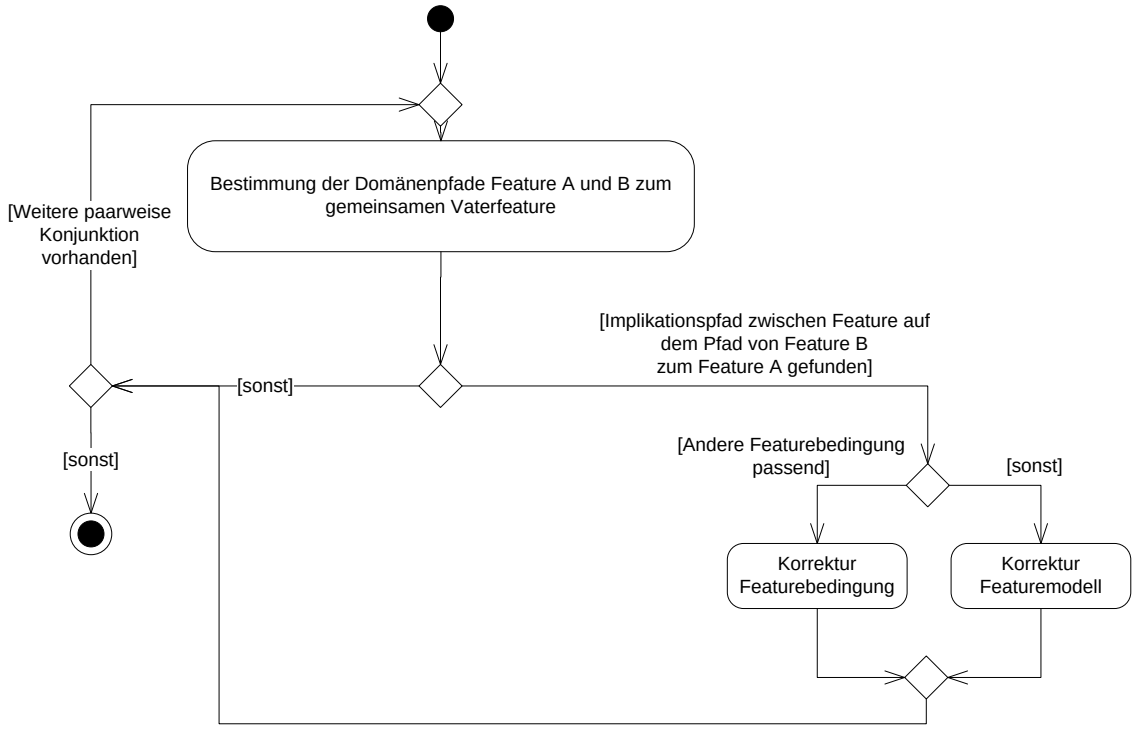


Abbildung 6.7: Algorithmus: Identifikation und Behebung der Kontradiktion in Featurebedingungen des Aussagetyps $\neg A \wedge B$

Für jede Featurebedingung werden die Teilbedingungen überprüft, deren Features ausschließlich durch Konjunktionen verbunden sind. Die Features einer solchen Teilbedingung werden dann paarweise überprüft, um eine Kontradiktion der Featurebedingung ausschließen zu können.
Zunächst werden die beiden Domänenpfade der betrachteten Features $\neg A$ und B bestimmt. Im Anschluss daran wird überprüft, ob es einen Implikationspfad zwischen einen Feature auf dem Domänenpfad von B und dem Feature A gibt. Ist dies der Fall liegt eine Kontradiktion vor. Ist dies nicht der Fall, wird die nächste paarweise

Konjunktion von Features aus der Featurebedingung überprüft. Liegt eine Kontradiktion vor, muss manuell überprüft werden, wie dieser Fehler gelöst werden soll. Zum einen kann die Featurebedingung der Abbildungsimplikation verändert werden, falls eine passende Featurebedingung definiert werden kann. Ist dies nicht möglich, muss das Featuremodell verändert werden.

6.2 Identifikation von Fehlern in Modellen des SPL-Entwicklungsprozesses

Im Rahmen der Erweiterung von Modellierungssprachen um Variabilität wurden Kardinalitäten gelockert (vgl. Abschnitt 5.1.1). Ein Beispiel für die Lockerung ist in Abbildung 6.8 anhand des Testfallmetamodells dargestellt. In (a) ist das Metamodell vor der Erweiterung und in (b) nach der Erweiterung um Variabilität dargestellt. Im Rahmen der Erweiterung wurden die Kardinalitäten an den Assoziationen zwischen den Metaklassen *Parameter* und *Typ*, *Testfall* und *Parameter*, sowie *Testfall* und *Testdatensatz* von 1 auf 0..1 sowie 1..∗ auf 0..∗ gelockert.

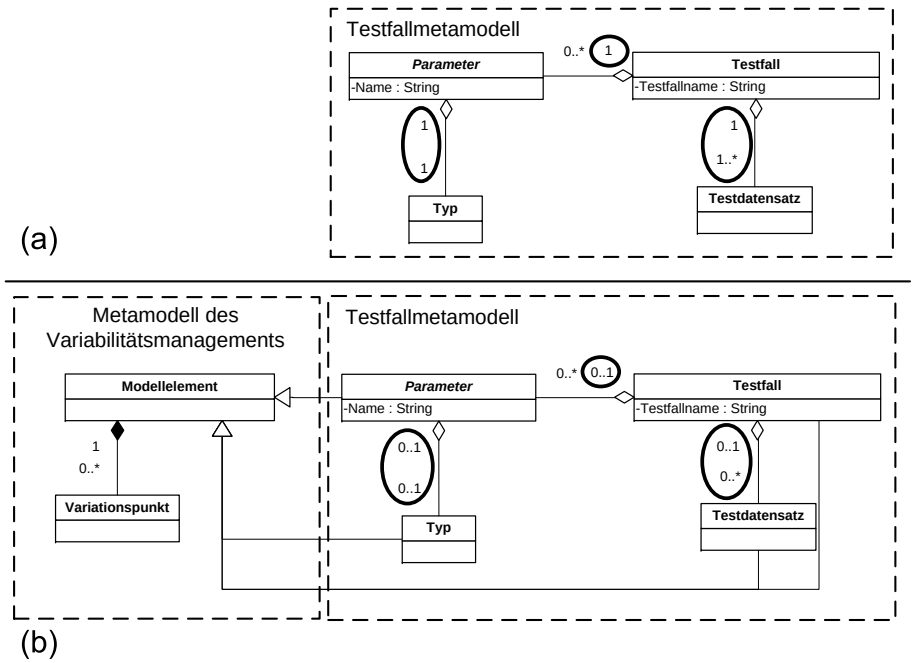


Abbildung 6.8: Beispiel für Lockerung von Kardinalitäten im Testfallmetamodell für die Modellierung von Variabilität

In Abbildung 6.9 ist ein Beispielmodell als Instanz Testfallmetamodells aus Abbildung 6.8.b gegeben. Es existieren dabei zwei Additionspunkte an einem *Parameter*. Der eine besitzt zwei Varianten vom Typ *Typ*, der andere eine. Falls nun im Rahmen einer Produktableitung zum Beispiel zwei der Varianten gebunden werden, sind an den Parameter zwei Typen assoziiert. Das Ausgangsmetamodell in Abbildung 6.8.a schließt eine solche Modellinstanz aber aus, was auch für alle abgeleiteten Produkte gelten muss, da in ihnen keine Variabilität mehr vorhanden sein soll.

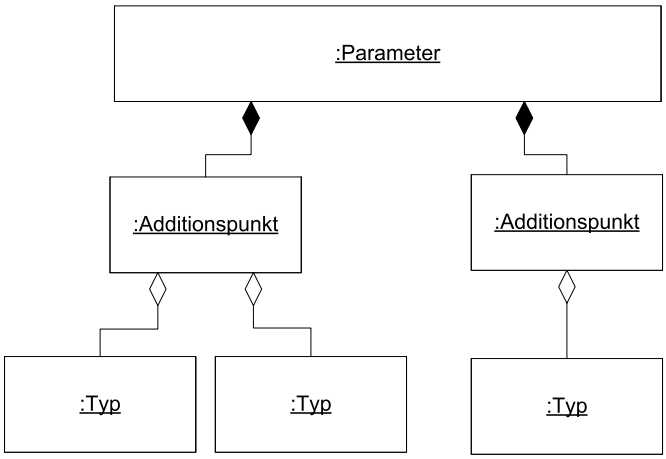


Abbildung 6.9: Beispielmodell die Kardinalität 0..1 und der Bindung von mehr als einer Klasse

Allgemein formuliert, muss bei Varianten und einer Kardinalität von 0..1 verhindert werden, dass mehr als eine Variante während der Ableitung gebunden werden kann. Dies ist genau dann der Fall, wenn die Featurebedingungen der Varianten sich paarweise ausschließen.

In Abbildung 6.10 ist ein zweites Beispielmodell als Instanz des Testfallmetamodells für die Bindung keiner Klasse an die Kardinalitäten 0..1 und 0..$*$ aus Abbildung 6.8.b gegeben. In (a) ist ein Testfall mit einen Additionspunkt und zwei Testdatensätzen als Varianten spezifiziert. In (b) ist das Beispielmodell nach der Ableitung eines Produktes dargestellt. Dabei wurde keine der beiden Varianten an die neu hinzugefügte Additionspunktbindung assoziiert (vgl. Bindung von Variabilität in Abschnitt 4.2.1). Das Ausgangsmetamodell in Abbildung 6.8.a verlangt aber die Assoziation mindestens eines Testdatensatzes an jeden Testfall. Dies muss auch für jeden Testfall gelten, welcher für ein Produkt abgeleitet worden ist. Daher muss

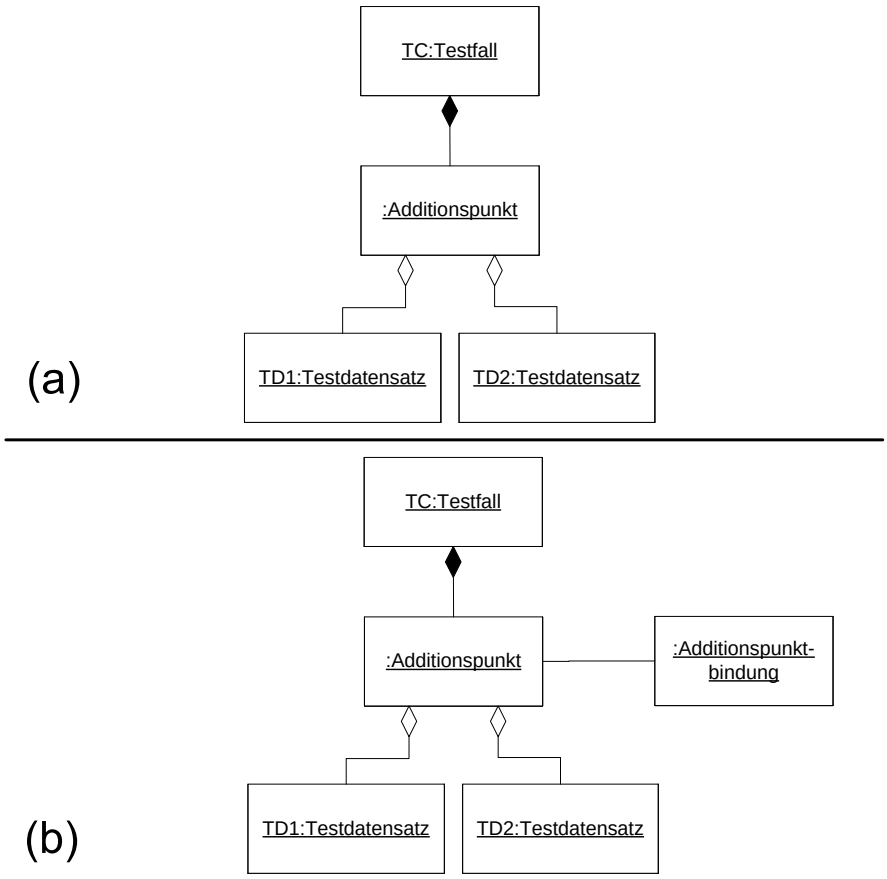


Abbildung 6.10: Beispielmodell für die Kardinalität 0..1 und 0..∗ bei der Bindung keiner Klasse

sichergestellt werden, dass bei der Ableitung des Testfalls mindestens eine Variante an die Additionspunktbindung gebunden wird oder mindestens ein Testdatensatz direkt am Testdatensatz assoziiert ist.

6.2.1 Fehlermodell

Bei der Spezifikation von Modellen können aus Sicht der Variabilität zwei Fehler passieren, die in Tabelle 6.4 beschrieben werden.
Im Folgenden werden nun zwei Algorithmen zur Identifikation und Behebung der definierten Fehler beschrieben.

Fehler	**Begründung**	**Auswirkung**
Varianten, die über Additionspunkte an eine Klasse assoziiert sind, von denen nach der Ableitung nur genau eine gebunden werden darf, schließen sich nicht gegenseitig aus.	Die Featurebedingungen der Varianten schließen sich paarweise nicht gegenseitig aus.	Das abgeleitete Modell enthält mehr als eine Variante, sollte aber genau eine enthalten.
An einer Klasse fehlt nach der Ableitung eines Produktes eine Klasse, die laut Ausgangsmetamodell in jedem Fall existieren und assoziiert sein muss (*Pflichtklasse*).	Es existiert keine direkt assoziierte Pflichtklasse an der betrachteten Klasse und die als Varianten an Additionspunkten assoziierten Pflichtklassen erlauben die Bindung keiner Variante.	Das abgeleitete Modell enthält keine Pflichtklasse assoziiert an der betrachteten Klasse, sollte aber mindestens eine besitzen.

Tabelle 6.4: Fehlermodell für Modellierung von Variabilität in Modellen

6.2.2 Identifikation und Behebung von Fehlern

In Abbildung 6.11 ist ein Algorithmus für die Identifikation und Behebung des ersten in Tabelle 6.4 beschriebenen Fehlers als UML-Aktivitätsdiagramm dargestellt. Der Algorithmus erhält als Eingabe zwei Featurebedingungen und das Featuremodell, auf dem diese beruhen. Ziel ist es nun zu zeigen, dass sich die beiden Featurebedingungen paarweise ausschließen. Ein Ausschluss existiert genau dann, wenn beide Featurebedingungen nicht zusammen eintreten können, also eine Kontradiktion bilden. Aus diesem Grund wird das Vorgehen für die Identifikation einer Kontradiktion in Featurebedingungen aus dem letzten Abschnitt für diese Überprüfung verwendet. Eine Kontradiktion kann nur in Verbindung mit der Konjunktion von Featuren austreten (vgl. Tabelle 6.2).

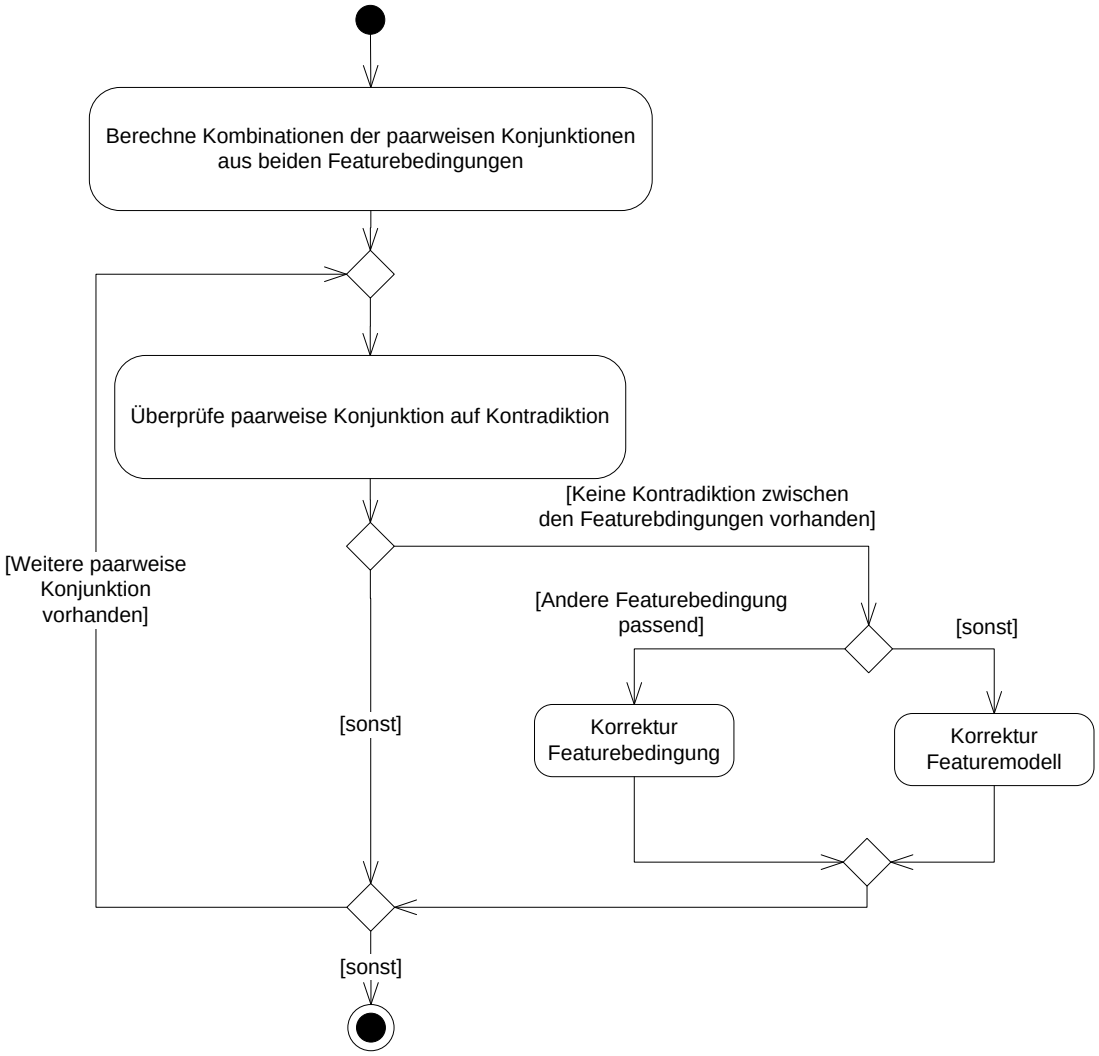


Abbildung 6.11: Algorithmus: Überprüfung des paarweisen Ausschlusses von Featurebedingungen von Varianten

Daher werden zunächst alle möglichen Kombinationen aus Teilbedingungen berechnet und durch Konjunktionen verbunden. Jede Kombination wird nun auf eine Kontradiktion hin untersucht. Falls keine Kontradiktion existiert, muss entschieden werden, ob entweder die Featurebedingung verändert werden kann oder das Featuremodell angepasst wird.

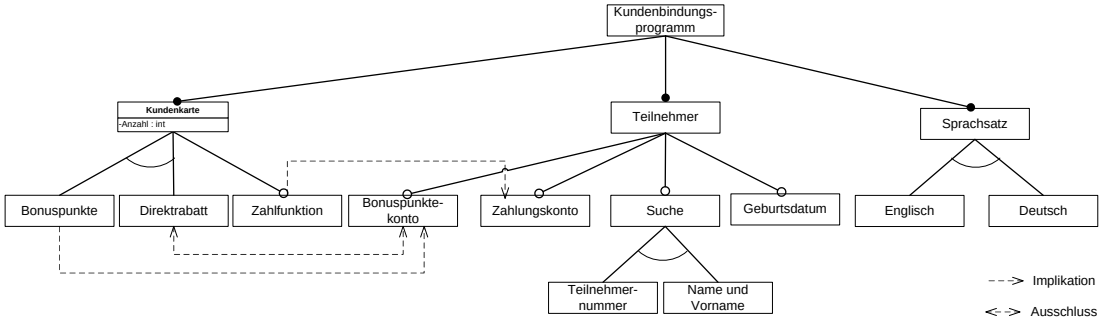


Abbildung 6.12: Beispiel Featuremodell

Beispiel für die Überprüfung von zwei Featurebedingungen auf Kontradiktion:

Abbildung 6.12 zeigt ein Beispielfeaturemodell. Für zwei Varianten einen Additionspunktes werden folgende zwei Featurebedingungen definiert:

> **Featurebedingung Variante 1:** Name und Vorname $\wedge$ Geburtsdatum
> **Featurebedingung Variante 2:** Teilnehmernummer $\vee$ Zahlfunktion

Für diese beiden Featurebedingungen soll nun überprüft werden, ob sie Alternativen darstellen. Zunächst werden dazu die möglichen Kombinationen paarweiser Konjunktionen berechnet. Dabei werden alle Teilfeaturebedingungen, die durch Disjunktionen verbunden sind, paarweise durch Konjunktionen verbunden und anschließend überprüft. In unserem Beispiel ergeben sich folgende zu überprüfende Kombinationen:

> **Kombination 1:**
> Name und Vorname $\wedge$ Geburtsdatum $\wedge$ Teilnehmernummer
> **Kombination 2:**
> Name und Vorname $\wedge$ Geburtsdatum $\wedge$ Zahlfunktion

In Kombination 1 existiert eine Kontradiktion, da die beiden Features *Name und Vorname* und *Teilnehmernummer* als eine Alternative am gemeinsamen Vaterfeature definiert sind (vgl. Abbildung 6.12). In Kombination 2 ist keine Kontradiktion vorhanden. Daher kann hier zum Beispiel durch Anpassung der Featurebedingung eine Kontradiktion herbeigeführt werden:

Featurebedingung Variante 2^***:**
Teilnehmernummer $\vee$ Zahlfunktion $\wedge\neg$Geburtsdatum

Mit der Erweiterung der Bedingung durch die Negation des Features Geburtsdatum wird Kombination 2 verändert und stellt eine Kontradiktion dar, da das Feature *Geburtsdatum* nicht gleichzeitig gewählt und abgewählt werden kann:

Kombination 2^***:**
Name und Vorname $\wedge$ Geburtsdatum $\wedge$ Zahlfunktion $\wedge\neg$Geburtsdatum

In Abbildung 6.13 ist die algorithmische Überprüfung des zweiten Fehlers aus Tabelle 6.4 als UML-Aktivitätsdiagramm dargestellt.

Eingabe für den Algorithmus ist ein abgeleitetes Modell, sowie das Ausgangsmodell und das um Variabilität erweiterte Metamodell. Für jede Kardinalität 1 oder 1..$*$ einer Assoziation im Ausgangsmodell, die zu 0..1 bzw. 0..$*$ gelockert wurde, muss nach der Ableitung überprüft werden, ob an der jeweils betrachteten Klasse mindestens (Kardinalität 1..$*$) oder genau (Kardinalität 1) eine Klasse assoziiert ist (*Pflichtklasse*).
Im Algorithmus aus Abbildung 6.13 wird zunächst überprüft, ob eine Pflichtklasse direkt an die betrachtete Klasse assoziiert ist. Ist dies der Fall wird mit der Überprüfung der nächsten Klasse fortgefahren. Ansonsten werden die an der betrachteten Klasse assoziierten Additionspunkte betrachtet, an denen die Pflichtklasse als Variante assoziiert ist.
Es wird überprüft, ob eine Additionspunktbindung am Additionspunkt existiert, an dem eine Pflichtklasse assoziiert ist. Ist dies der Fall, wird mit der nächsten Klasse des Modells fortgefahren.
Ansonsten existiert weder eine direkt an der betrachteten Klasse assoziierte Pflichtklasse noch eine über eine Additionspunktbindung assoziierte. In diesem Fall muss entweder die Featureauswahl für die Ableitung verändert oder das Featuremodell bzw. die Featurebedingung von Varianten manuell verändert werden.

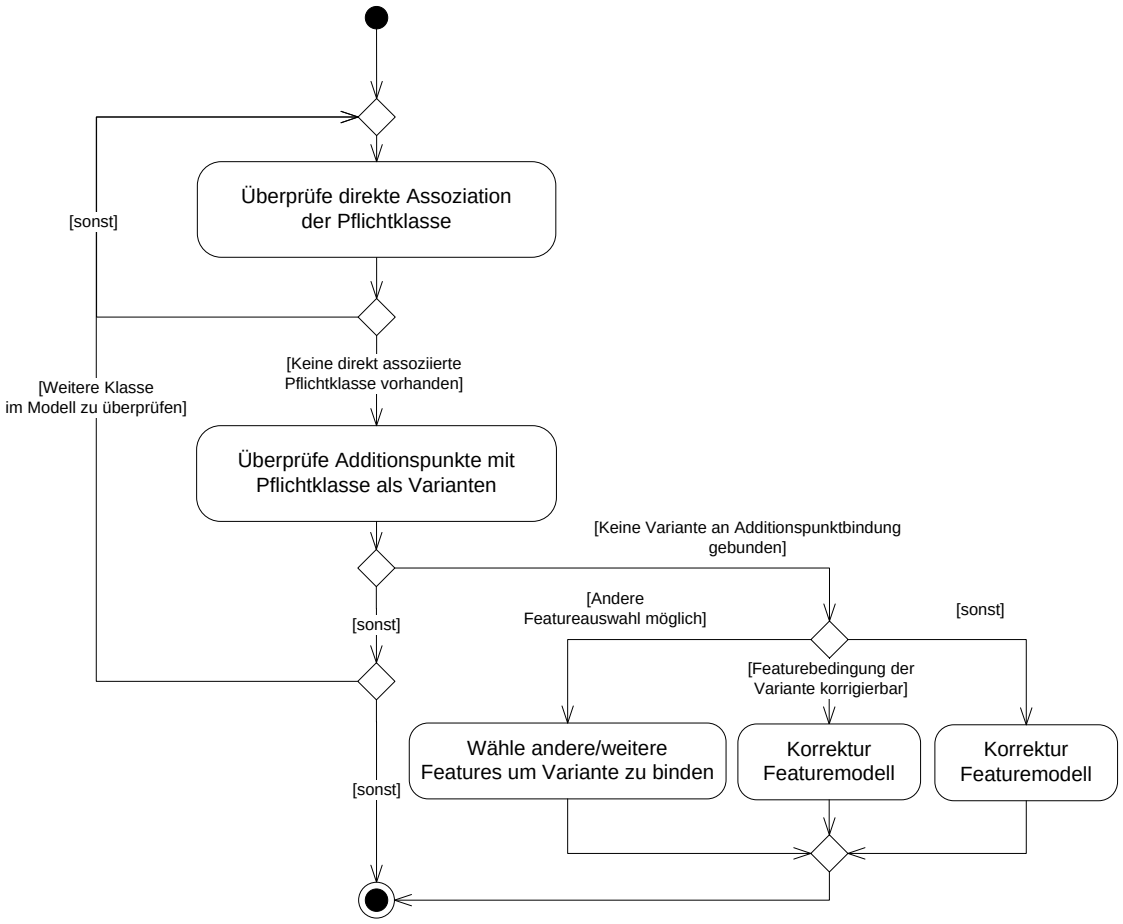


Abbildung 6.13: Algorithmus: Überprüfung der Bindung mindestens einer Variante an eine Additionspunktbindung

6.3 Zusammenfassung

Die Möglichkeit der Modellierung von Variabilität in Modellen der Produktlinienplattform und ihrem featurebasierten Management führt zu neuen Arten von Fehlern, die durch den Nutzer gemacht werden können. In diesem Kapitel wurden daher zum einen die möglichen Fehler in Form von Fehlermodellen definiert und zum anderen Maßnahmen zu deren Identifikation beschrieben.
Zunächst erfolgte dafür in Abschnitt 6.1 die Untersuchung des featurebasierten Variabilitätsmanagements und die Definition eines Fehlermodells für die Abbildung zwischen Features und Modellelementen. Mit Hilfe dieser Fehlermodelle wurden Algorithmen definiert, um diese Fehler zu identifizieren.

In gleicher Art und Weise verfuhr Abschnitt 6.2 mit der Spezifikation von Variabilität in Modellen.
Die spezifizierten Fehlermodelle und Algorithmen zur Identifikation der darin definierten Fehler ist ein Beitrag für die Sicherung der Qualität der Modelle und des Variabilitätsmanagements für die Produktlinienplattform. Mit ihrer Hilfe ist eine frühzeitige Erkennung von Fehlern in Bezug auf die in der Produktlinienplattform spezifizierte Variabilität vor dem eigentlichen *Test* eines abgeleiteten Produktes möglich.

Kapitel 7

Plattformanforderungsspezifikations-prozess mit Variabilität

Die in Kapitel 5 definierte Anforderungsmodellierungssprache mit Variabilität sowie das featurebasierte Variabilitätsmanagement werden in diesem Kapitel dafür eingesetzt einen Spezifikationsprozess für Anforderungen mit Variabilität zu definieren. Zunächst wird dazu ein Prozessmetamodell für die Definition von Spezifikationsprozessen vorgestellt. Im Anschluss daran wird der Spezifikationsprozess selbst beschrieben und durch Beispiele veranschaulicht.

7.1 Prozessmetamodell für die Prozessdefinition

Prozesse beschreiben einen schrittweisen Ablauf, um Eingabeprodukte in Ausgabeprodukte zu transformieren [Int05]. Ein Prozess wird in dieser Arbeit durch das Prozessmetamodell aus Abbildung 7.1 definiert, welches in Anlehnung an das Software Process Engineering Metamodel (SPEM) [OMG08] definiert ist. Ein *Prozess* besteht aus mindestens einem *Prozessschritt*. Jeder Prozessschritt besitzt mindestens einen *Akteur*. Akteure führen die Transformation der Eingabe- in Ausgabeprodukte durch. Jeder Prozessschritt hat daher mindestens ein *Eingabe-* und *Ausgabemodell*. Diese stellen in dieser Arbeit die in [Int05] beschrieben Ein- und Ausgabeprodukte dar.
Vorschriften für die Transformation der Eingabe- in Ausgabemodelle sind durch Algorithmen beschrieben. Als Sprache für die Definition von Algorithmen dient das UML-Aktivitätsdiagramm. Dieses bietet den Vorteil einer übersichtlichen und stan-

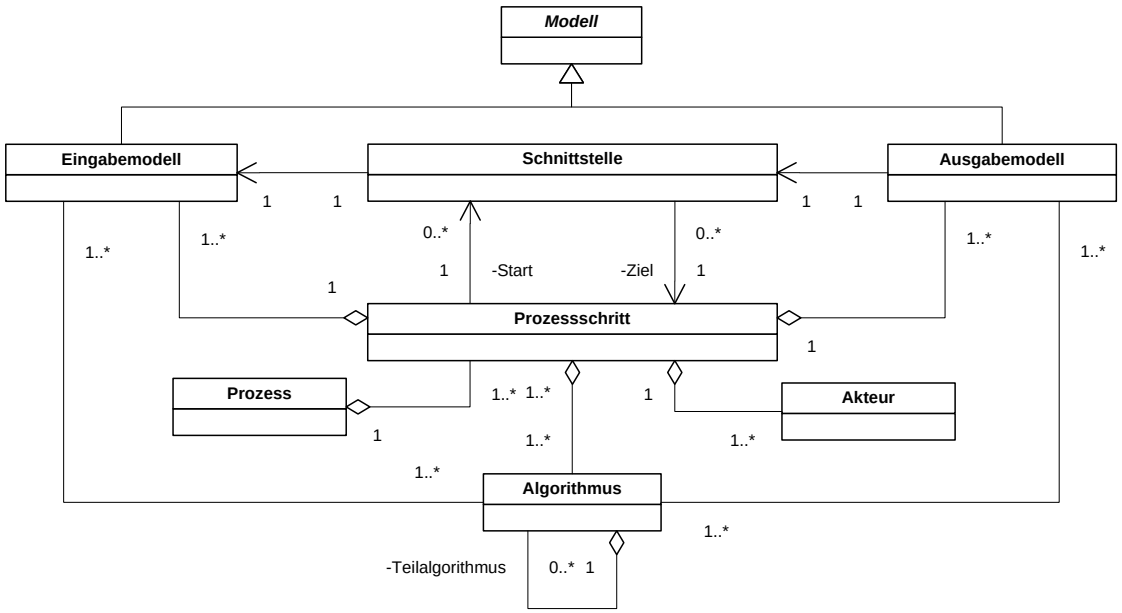


Abbildung 7.1: Prozessmetamodell als Sprache für die Definition von Prozessen

dardisierten Darstellung. Jeder Prozessschritt besitzt mindestens einen *Algorithmus*, der wiederum an mindestens ein Eingabe- und ein Ausgabemodell assoziiert ist. Modelle können Ein- bzw. Ausgabe mehrerer Algorithmen sein. Verschiedene Algorithmen können zum Beispiel unterschiedliche Modellelemente der Modelle transformieren. Ein Algorithmus kann wiederum beliebig viele *Teilalgorithmen* beinhalten. Zwischen Prozessschritten können beliebig viele *Schnittstellen* existieren. Eine Schnittstelle überstellt ein Ausgabemodell eines Prozessschrittes als Eingabemodell für einen anderen Prozessschritt.

In Abbildung 7.2 ist ein Prozessmodell als Instanz des Prozessmetamodells gegeben. Für das Modell wurde eine konkrete Syntax gewählt. Alle Prozessschritte gemeinsam stellen den Prozess dar. Ein Teilalgorithmus wird durch einen gestrichelten Pfeil an einen Algorithmus assoziiert. Ein- und Ausgabemodelle werden durch Kanten an Algorithmen assoziiert.

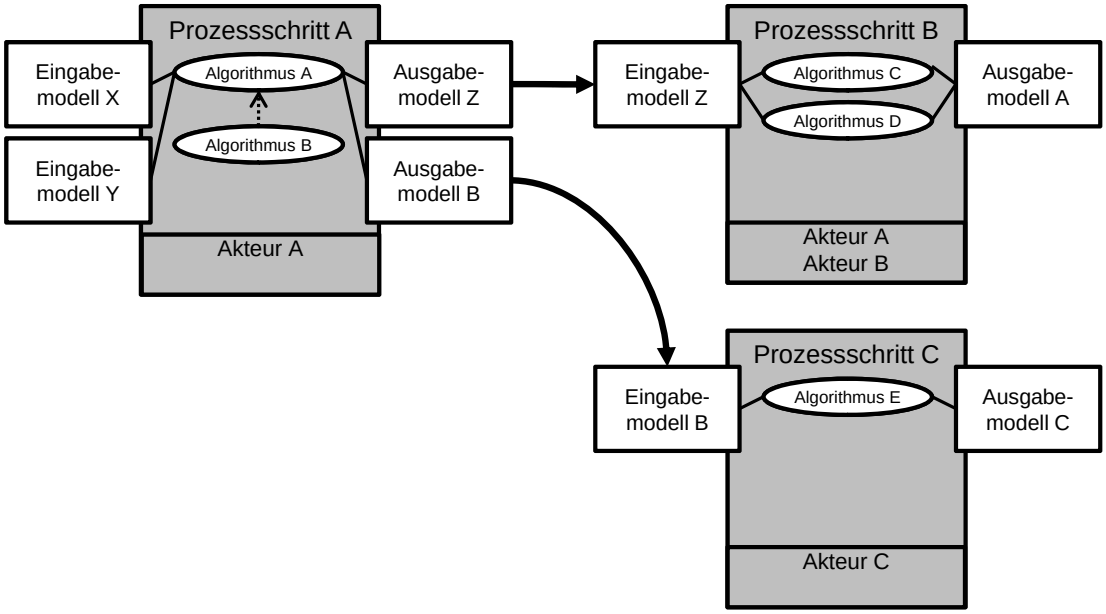


Abbildung 7.2: Beispielprozess in konkreter Syntax

7.2 Forderungen an den Anforderungsspezifikationsprozess

In Abschnitt 2.1 dieser Arbeit wird die Anforderungsspezifikation im Kontext der Softwareentwicklung beschrieben. Im Kontext der Entwicklung von SPL entstehen nach [PBL05] drei Forderungen, die ein Anforderungsspezifikationsprozess leisten soll:

1. Analyse der Gemeinsamkeiten: Welche Anforderungen sind allen Produkten einer SPL gemeinsam?

2. Analyse der Variabilität: Welche Anforderungen differieren in verschiedenen Produkten der SPL und wie genau sieht diese Differenz aus?

3. Modellierung der Variabilität: Definition von Modifikations- sowie Additionspunkten mit ihrer Varianten und Abhängigkeiten in Anforderungen.

Die Analyse und Spezifikation von Variabilität stellt demnach die besondere Herausforderung im Spezifikationsprozess für Anforderungen bei SPL dar. Die dritte

Aufgabe lässt sich durch das in Kapitel 4 vorgestellte Variabilitätsmanagement unterstützen. Dies bedeutet für den Spezifikationsprozess von Anforderungen,

- dass gemeinsame und variable Features im Featuremodell spezifiziert werden,
- dass Modifikations- und Additionspunkte und ihre Varianten in Anforderungen mithilfe der vorgestellten Anforderungsmodellierungssprache mit Variabilität spezifiziert werden,
- und dass die Variabilität der Modellelemente von Anforderungen mithilfe des Abbildungsmodells auf Features im Featuremodell abgebildet werden.

Für die Spezifikation von Variabilität in Anforderungen können zwei unterschiedliche Szenarien unterschieden werden, die im nun folgenden Abschnitt beschrieben werden.

7.3 Überblick über den Anforderungsspezifikationsprozess

Die Spezifikation der Anforderungen für eine Produktlinienplattform geschieht in dieser Arbeit beispielhaft auf Basis des in Kapitel 2.2.1 eingeführten und in Kapitel 5 um Variabilität erweiterten Anwendungsfallmetamodells. Abbildung 7.3 stellt den Anforderungsspezifikationsprozess mit Hilfe der in Abschnitt 7.1 definierten Sprache dar. Zwei unterschiedliche Szenarien werden für die Durchführung des Prozesses unterschieden:

Identifikation von Variabilität

Im ersten Szenario, dargestellt in Abbildung 7.3 oben, existieren bereits Anforderungsmodelle, in denen Variabilität implizit modelliert worden ist. Zunächst wird die vorhandene Variabilität *identifiziert*. Dazu werden *Schlüsselbegriffe* für implizite Variabilität eingesetzt. Im Anschluss daran wird die identifizierte Variabilität im Anforderungsmodell explizit *modelliert*. Für das Management der explizit modellierten Variabilität wird ein Featuremodell erstellt und dessen Features über das Abbildungmodell mit der Variabilität im Anforderungsmodell assoziiert.

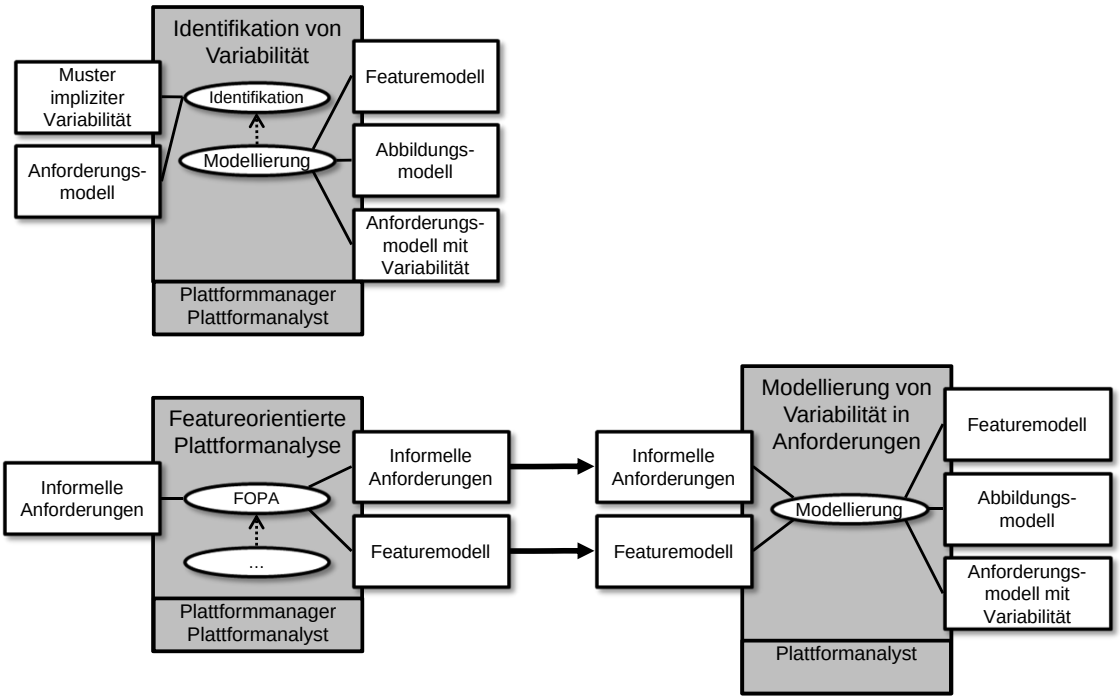


Abbildung 7.3: Überblick über den Anforderungsspezifikationsprozess mit Variabilität

Die Identifikation und Modellierung wird durch die beiden Rollen Plattformmanager und Plattformanalyst durchgeführt. Ersterer ist für das Variabilitätsmanagement mit Hilfe des Featuremodells und letzterer für die Anforderungsmodelle verantwortlich (vgl. Abschnitt 2.1).

Featureorientierte Plattformanalyse

In diesem Szenario, dargestellt in Abbildung 7.3 unten, startet die Definition des Anforderungsmodells auf Basis von informellen Anforderungen, da noch keine Anforderungsmodelle existieren. Zunächst wird im ersten Prozessschritt auf Basis dieser informellen Anforderungen ein Featuremodell der Produktlinienplattform modelliert. An diesem Schritt sind der Plattformmanager und der Plattformanalyst beteiligt. Die informellen Anforderungen und das Featuremodell stellen dann die Eingabe für den zweiten Prozessschritt dar. In diesem wird das Anforderungsmodell mit expliziter Variabilität spezifiziert.

Die Variabilität im Anforderungsmodell wird über das Abbildungsmodell an das Featuremodell assoziiert. Diese drei Modelle sind Ausgabe des Prozessschritts, welcher durch die Rolle Plattformanalyst durchgeführt wird.

7.4 Identifikation von Variabilität in existierenden Anforderungsmodellen

Die Idee in diesem Szenario besteht darin, die in Anwendungsfällen *implizit* vorhandene Variabilität explizit zu modellieren. Implizit bedeutet, dass die Variabilität nicht durch eine dedizierte Sprache modelliert wurde, sondern durch die Sprachmittel der Anwendungsfallbeschreibung. Die Idee der impliziten Variabilität wurde auch in [PBKS04] erkannt. Ausgangspunkt der Spezifikation sind dabei Anwendungsfallbeschreibungen, die bereits für verschiedene Produkte eingesetzt wurden, aber dabei nie mit expliziter Variabilität modelliert wurden.

Für die Identifikation von Variabilität in existierenden Anforderungsmodellen werden Schlüsselbegriffe eingesetzt. Diese Begriffe sind Worte, die auf potentielle Variabilität hinweisen. Schlüsselbegriffe werden manuell erstellt und können je nach Modell unterschiedlich sein. In Tabelle 7.1 sind beispielhaft Schlüsselbegriffe beschrieben, welche für Anwendungsfallbeschreibungen eingesetzt werden können. Diese Begriffe sind Teil einer Untersuchung von Anwendungsfallbeschreibungen des Industriepartners arvato services. Die Liste der Schlüsselwörter aus Tabelle 7.1 wurde im Rahmen einer studentischen Arbeit definiert [Emk09]. Neben diesen Begriffen können andere Schlüsselbegriffe, wie zum Beispiel *unter Umständen (u. U.)* oder *Falls* ein Hinweis auf Variabilität sein. Deshalb muss die Entscheidung welcher Begriff bei der Suche nach Variabilität eingesetzt werden soll, im Vorfeld manuell getroffen werden.

Die Schlüsselbegriffe werden nun im Algorithmus zur *Identifikation impliziter Variabilität* eingesetzt. Dieser ist als UML-Aktivitätsdiagramm in Abbildung 7.4 beschrieben.

Im ersten Schritt des Algorithmus wird im gegeben Anforderungsmodell nach einer Schlüsselbegriffübereinstimmung gesucht. Ist eine Übereinstimmung gefunden, muss manuell entschieden werden, ob das gefundene Teilmodell für die Modellierung

Schlüsselbegriff	Beschreibung
ggf.	Beschreibt Variabilität, die zum Zeitpunkt der Produktableitung gebunden wird.
optional	Beschreibt Variabilität, die entweder zum Zeitpunkt der Produktableitung oder zur Laufzeit gebunden werden kann.
wenn (, dann)	Beschreibt Variabilität, die zum Zeitpunkt der Produktableitung gebunden wird.

Tabelle 7.1: Beispiel für Schlüsselbegriffe impliziter Variabilität

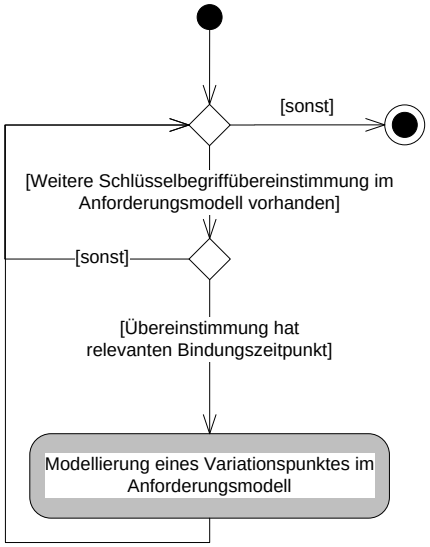


Abbildung 7.4: Algorithmus: **Identifikation** impliziter Variabilität

von Variabilität auf Sicht einer Produktlinienplattform relevant ist. Dazu muss der Bindungszeitpunkt der Variabilität untersucht werden (vgl. Abschnitt 2.1.4). Variabilität, die zur Laufzeit des Systems vorkommt, wie zum Beispiel eine Alternative in einem Anwendungsfall, wird nicht in der Produktlinienplattform modelliert. Diese Form von Variabilität ist während der Benutzung des Systems gewollt und kann zum Beispiel in einer Anwendungsfallbeschreibung mit Hilfe der Alternative zum Ausdruck gebracht werden, ohne eine Variabilitätsmodellierungssprache zu benötigen. Bei der Identifikation von Variabilität wird daher der Bindungszeitpunkt *Laufzeit* ignoriert.

Ist der Bindungszeitpunkt für die Schlüsselbegriffübereinstimmung passend, wird der Algorithmus *Modellierung eines Variationspunktes im Anforderungsmodell* aufgerufen. Der Aufruf eines Teilalgorithmus wird im UML-Aktivitätsdiagramm durch eine grau-hinterlegte Aktivität spezifiziert.
Bei der Untersuchung von Anforderungsmodellen ist weiterhin auch der Vergleich von solchen Modellen aus ähnlichen Produkten interessant, die für eine Produktlinienplattform konsolidiert werden sollen. Hierbei können die zuvor eingeführten Schlüsselbegriffe für implizite Variabilität nicht unterstützen. Vielmehr müssen ähnliche oder gleiche Teilmodelle in den verschiedenen Anforderungsmodellen identifiziert werden.
Auf Basis der Beispielschlüsselbegriffe aus Tabelle 7.1 wird nun ein Beispiel für die Identifikation impliziter Variabilität gegeben.

Beispiel für die Identifikation von impliziter Variabilität in Anwendungsfallbeschreibungen

Nr.	Akteure	Normaler Ablauf
1	**-**	**Teilnehmer suchen**
1.1	SCA	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
1.2	SCA	Der SCA sucht den Teilnehmer und zeigt die Details des Teilnehmers an. Mögliche Suchangaben: •Vorname (optional) •Name (Pflicht, wenn keine Teilnehmernummer eingegeben) •Teilnehmernummer (Pflicht, wenn kein Name eingegeben) •Geburtsdatum (optional)
1.3	SCA	Ausnahme: Teilnehmer nicht gefunden
2	**-**	**Ggf. Stammdaten ändern**
2.1	SCA	Wenn Stammdaten Teil des Teilnehmers sind, dann ändert der SCA sie in der Ansicht der Stammdaten. → Sekundärer_Anwendungsfall_UC01_sek_01
3	**-**	**Ggf. Zahlungsinformationen ändern**
3.1	SCA	Wenn Zahlungsinformationen Teil des Systems sind, dann ändert der SCA sie über einen Wechsel in die Ansicht der Zahlungsmittel. →Sekundärer_Anwendungsfall_UC01_sek_02

Abbildung 7.5: Beispiel impliziter Variabilität in der Ablaufbeschreibung eines Anwendungsfalls

In Abbildung 7.5 ist die Ablaufbeschreibung eines Anwendungsfalls für eine Teilnehmerdatenverwaltung dargestellt. Der Akteur „Service Center Agent“ (SCA) hat die Möglichkeit über eine Suchfunktion einen Teilnehmer zu suchen und anschließend seine Stammdaten oder Zahlungsinformationen zu ändern, was jeweils über einen inkludierten Anwendungsfall spezifiziert wird. In der Abbildung sind an verschiedenen Stellen Kandidaten für implizite Variabilität vorhanden (umrandet). In jedem Einzelfall muss entschieden werden, ob es sich um Variabilität in der Produktlinienplattform handelt. Für diese Entscheidung ist der Bindungszeitpunkt eines Variationspunktes von Interesse.
Der erste Kandidat im Anwendungsfall ist die Angabe „Pflicht, wenn...“ für den Suchbegriff „Name“. Hierbei kann es sich zusammen mit der „Pflicht, wenn...“ Angabe der Teilnehmernummer um zwei Arten von Variabilität handeln: Zum einen kann diese Variabilität zur Laufzeit oder aber zum anderen bereits zur Produktableitungszeit gebunden werden. Diese Entscheidung wird durch die Rollen Plattformmanager und Plattformanalyst getroffen. Es muss geklärt werden, ob es Produkte ohne das Modellelement „Name“ als Suchparameter geben soll. Ist dies der Fall, handelt es sich um Variabilität, die zur Produktableitungszeit gebunden wird.
Ein ähnlicher Fall liegt bei dem Muster „optional“ für das Suchkriterium Geburtsdatum vor. Das Geburtsdatum könnte für den Teilnehmer zur Produktableitungszeit optional sein oder aber zur Laufzeit eine optionale Eingabe sein. In diesem Fall ist auch noch eine bedingte Variabilität möglich. Wenn zur Produktableitungszeit das Geburtsdatum ausgewählt wurde, dann kann zur Laufzeit eine zweite Auswahlentscheidung getroffen werden. Die erste Entscheidung zur Produktableitungszeit ist also notwendiges Kriterium für die Auswahlentscheidung zur Laufzeit.
Das nächste Muster ist „Ggf.“. In der Abbildung markiert er den Schritt 2. Hierbei handelt es sich um Variabilität zum Zeitpunkt der Produktableitung, da sonst der Schritt für die Variabilität zur Laufzeit als ein *alternativer Ablauf* modelliert worden wäre. Dabei sei in diesem Beispiel vorausgesetzt, dass der Anwendungsfall in Bezug auf *Ausnahmen* und *Alternativen* korrekt modelliert worden ist. Das letzte Muster ist „Wenn (, dann)“ und impliziert wiederum die Optionalität der Stammdaten des Teilnehmers. Hierbei handelt es sich um Variabilität zur Produktableitungszeit.
Die Modellierung der identifizierten Variabilität in Anforderungsmodellen und die Spezifikation des Featuremodells sowie der Abbildung zwischen Anforderungsmodellen und Featuremodell wird in den nächsten beiden Abschnitten beschrieben.

7.5 Featureorientierte Plattformanalyse (FOPA)

Die featureorientierte Plattformanalyse ist der erste Schritt der Modellierung von Anforderungen für eine Produktlinienplattform. Dieser manuelle Prozessschritt wird gemeinsam durch die beiden Rollen Plattformmanager und Plattformanalyst durchgeführt (vgl. Abbildung 7.3). Ziel ist es, auf Basis von informellen Anforderungen, welche die Eingabe dieses Prozessschrittes darstellen, ein Featuremodell zu spezifizieren, welches den Aufbau und die Variabilität der Produktlinienplattform beschreibt. Es existieren verschiedene Vorgehensweisen für die Spezifikation eines Featuremodells für eine Produktlinienplattform, die an dieser Stelle genutzt werden können. Darunter sind FODA [KCH+90], FORM [KKL+98] und FeatuRSEB [LGFM98]. Einen Vergleich existierender Feature-Modellierungsansätze gibt [LMNW03].
Für die Analyse der informellen Anforderungen kann einer der genannten Ansätze eingesetzt werden. Die informellen Anforderungen und das resultierende Featuremodell stellen dann die Eingabe für den nächsten Prozessschritt dar.

7.6 Modellierung von Variabilität in Anforderungen

Die Spezifikation von Anforderungen wird durch die Rolle Plattformanalyst durchgeführt (vgl. 2.1).

Der als UML-Aktivitätsdiagramm definierte und in Abbildung 7.6 dargestellte Algorithmus beschreibt den Prozessschritt für die Modellierung von Variabilität in Anforderungsmodellen (vgl. Abbildung 7.3). Dabei wird die Modellierung von Additions- und Modifikationspunkten unterschieden. Zunächst muss entschieden werden, ob der neue Variationspunkt ein Additions- oder Modifikationspunkt sein soll. Für ersteren Fall werden dann drei Szenarien unterschieden:

1. Für eine neu spezifizierte Variante ist bereits eine passende Abbildungsimplikation vorhanden: Dies bedeutet, dass die Features der Abbildungsimplikation durch die neue Variante konkretisiert werden. In diesem Fall muss die neue Variante nur an die Abbildungsimplikation assoziiert werden.

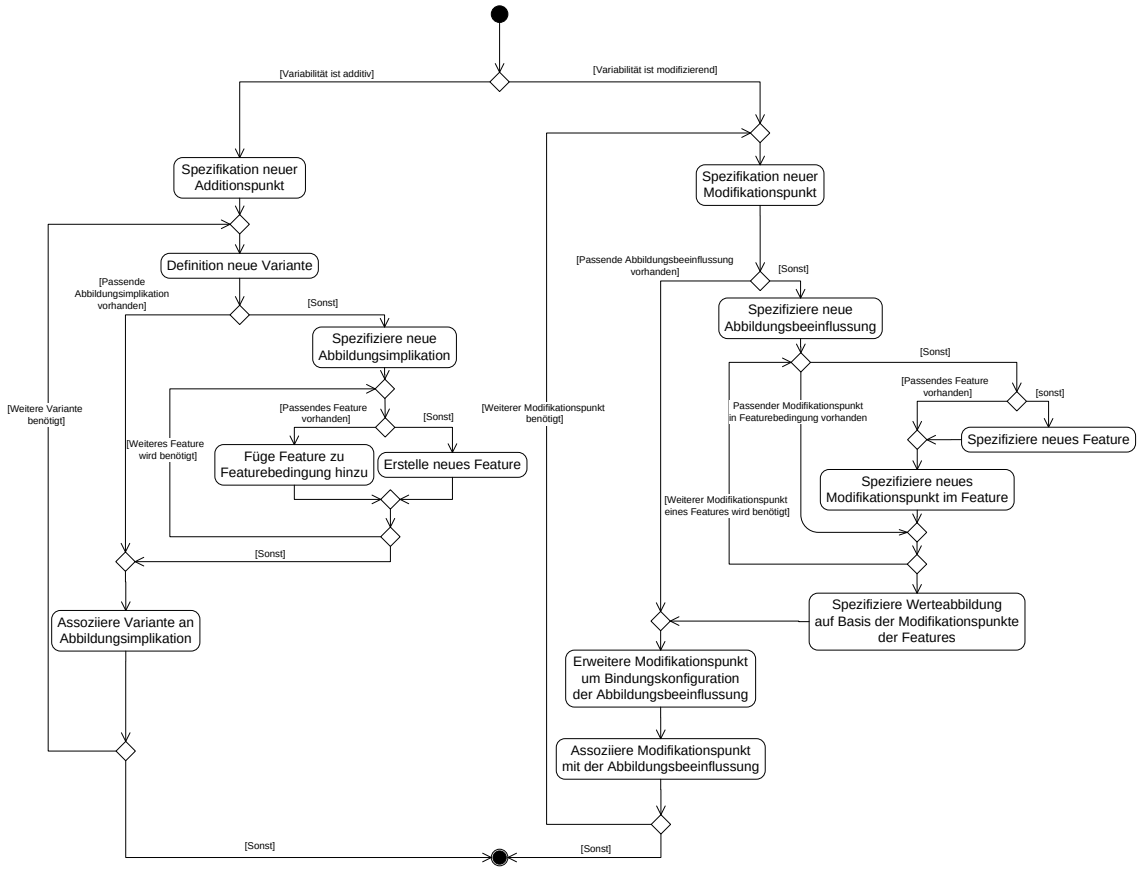


Abbildung 7.6: Algorithmus: **Modellierung** eines Variationspunktes im Anforderungsmodell

2. Für die neu spezifizierte Variante ist noch keine passende Featurebedingung vorhanden: Es wird eine neue Abbildungsimplikation mit einer neuen Featurebedingung erstellt. Hierbei werden zwei Fälle unterschieden:

   (a) Ein passendes Feature ist bereits vorhanden: Dieses Feature wird der Featurebedingung der neuen Abbildungsimplikation hinzugefügt.

   (b) Kein passendes Feature ist vorhanden: Es wird ein neues Feature im Featuremodell spezifiziert und in der Featurebedingung eingefügt.

Bei der Spezifikation von Modifikationspunkten werden vier Szenarien unterschieden:

1. Für den neuen Modifikationspunkt ist bereits eine passende Abbildungsbeeinflussung vorhanden: Der neue Modifikationspunkt wird zur Modifikationspunktmenge hinzugefügt.

2. Für den neuen Modifikationspunkt ist keine passende Abbildungsbeeinflussung vorhanden: Es wird eine neue Abbildungsbeeinflussung erstellt:

   (a) Für die Werteabbildung ist bereits ein passendes Attribut an einem Feature vorhanden: Die Werteabbildung der Abbildungsbeeinflussung wird unter Berücksichtigung dieses Attributs definiert.

   (b) Für die Werteabbildung ist kein passendes Attribut an einem Feature vorhanden:

      i. Es ist ein passendes Feature vorhanden: Ein neues Attribut wird dem Feature hinzugefügt und für die Definition der Werteabbildung der Abbildungsbeeinflussung genutzt.

      ii. Es ist kein passendes Feature vorhanden: Ein neues Feature mit einem neuen Attribut wird im Featuremodell spezifiziert. Der neue Modifikationspunkt wird für die Definition der Werteabbildung der Abbildungsbeeinflussung genutzt.

Beispiel für die Modellierung von Variabilität in Anforderungen

In Abbildung 7.7 ist der Prozessschritt für die Modellierung von Additions- und Modifikationspunkten in Anforderungen beispielhaft dargestellt. Der Prozess für Modifikationspunkte ist dabei mit gestrichelten Pfeilen illustriert. Der Plattformanalyst stellt zunächst während der Modellierung einer Anforderung fest, dass ein Modellelement eines Anforderungsmodells einen Variationspunkt enthalten soll.
Er fügt einen neuen Variationspunkt vp_1 mit eindeutigem Index hinzu (Schritt 1 in der Abbildung). In Schritt 2 werden die Varianten des Additionspunktes bestimmt. Im Beispiel der Abbildung ist dies Variante v_1. Für jede Variante muss dabei geklärt werden, ob im Abbildungsmodell bereits eine passende Abbildungsimplikation vorhanden ist. Passend bedeutet, dass die an dieser Abbildungsimplikation assoziierten Features durch das neue variable Modellelement konkretisiert werden.
Im Beispiel ist das Abbildungsmodell zunächst leer und es wird eine neue Abbildungsimplikation mit der Bindungskonfiguration BK_1 angelegt (Schritt 3 in der

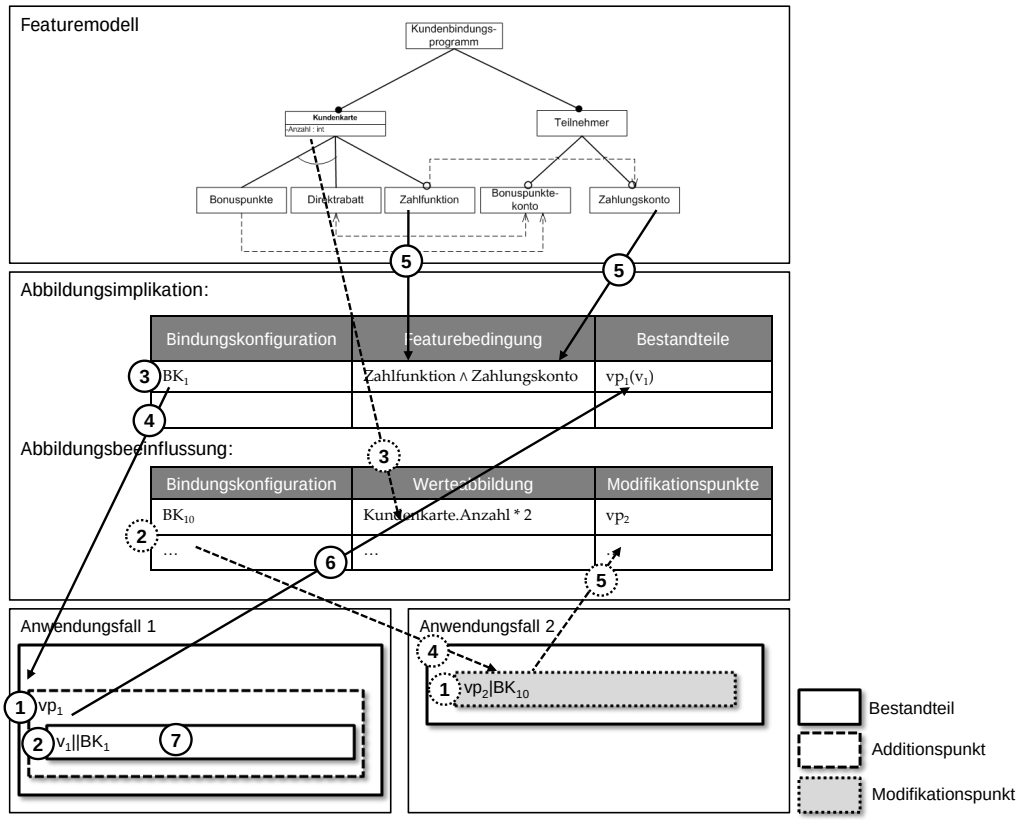


Abbildung 7.7: Beispiel für die Modellierung von Variabilität in Anforderungen

Abbildung). Diese Bindungskonfiguration wird in Schritt 4 an der Variante assoziiert ($v_1|BK_1$ in der Abbildung).

Dann wird die dazugehörige Featurebedingung mit Hilfe des Featuremodells bestimmt (Schritt 5 in der Abbildung). Dazu wird festgestellt, welche Features durch das neue variable Modellelement konkretisiert werden. Sind mehrere Features für die Featurebedingung notwendig, werden diese mittels $\wedge$ und $\vee$ zu einem logischen Ausdruck verbunden. Im Beispiel bedingen die beiden Features *Zahlfunktion* und *Zahlungskonto* die Variante v_1. Falls im Featuremodell noch keine passenden Features vorhanden sind, werden neue Features erstellt.

In Schritt 6 wird die neue Variante an der Abbildungsimplikation assoziiert. Die Variante wird dabei durch den Bezeichner des Additionspunktes und der Variante identifiziert ($vp_1(v_1)$ im Beispiel der Abbildung). Schließlich spezifiziert der Plattformanalyst den Inhalt der Variante (Schritt 7). Sind alle Varianten des Additionspunktes

an Abbildungsimplikationen assoziiert und der Inhalt jeder Variante spezifiziert, ist die Modellierung des Additionspunktes abgeschlossen.

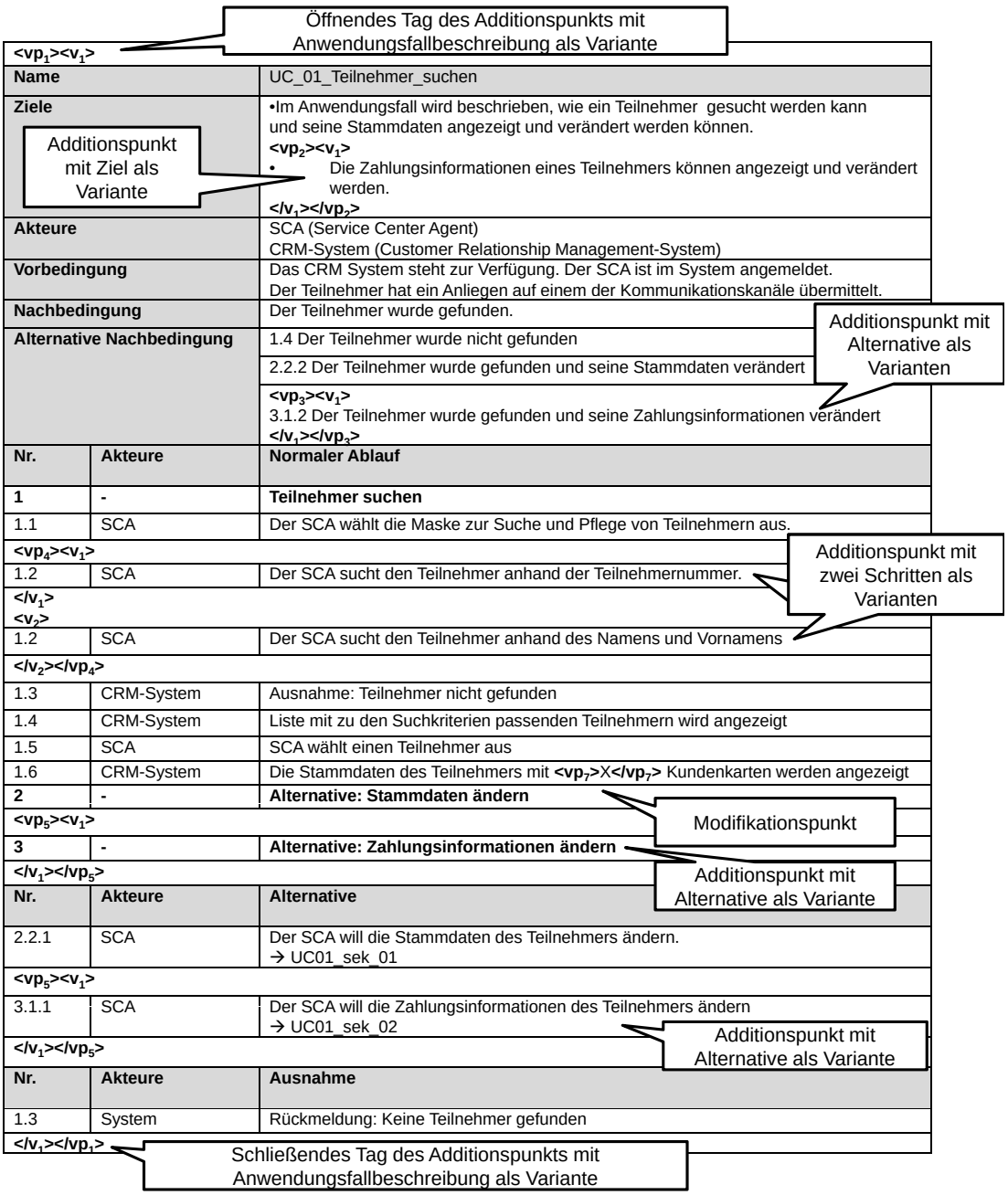
Für die Spezifikation eines Modifikationspunktes wird zunächst auch ein neuer Variationspunkt vp_2 erstellt (Schritt 1 in der Abbildung). Dann wird überprüft, ob bereits eine Abbildungsbeeinflussung mit der passenden Werteabbildung für diesen Modifikationspunkt existiert. Im Beispiel ist dies nicht der Fall und es wird eine neue Abbildungsbeeinflussung mit der Bindungskonfiguration BK_{10} erstellt (Schritt 2). In Schritt 3 wird ein passender Modifikationspunkt im Featuremodell für die Werteabbildung ausgewählt und die Abbildungsvorschrift zwischen diesem und dem neuen Modifikationspunkt spezifiziert ($Kundenkarte.Anzahl * 2$ im Beispiel). Falls kein passender Modifikationspunkt in einem Feature existiert, wird ein neuer Modifikationspunkt zu einem Feature hinzugefügt oder ein neues Feature mit einem Modifikationspunkt in das Featuremodell eingefügt. In Schritt 4 wird dann die Bindungskonfiguration der Abbildungsbeeinflussung an den Modifikationspunkt assoziiert. Zuletzt wird der Modifikationspunkt an die Abbildungsbeeinflussung assoziiert (Schritt 5).

Um neben dem eigentlichen Vorgehen auch die Veränderungen an den Modellen deutlich zu machen, ist der Algorithmus aus Abbildung 7.6 für die Modellierung von Additionspunkten und Modifikationspunkten in Anhang A.1 durch zwei Algorithmen im *Pseudocode* beschrieben [Sie03]. Zum Verständnis sind an die Anweisungen der Algorithmen die Schritte aus Abbildung 7.7 als Kommentare in geschweiften Klammern angefügt.

7.7 Beispiel: Variabilität in einer Anwendungsfallbeschreibung

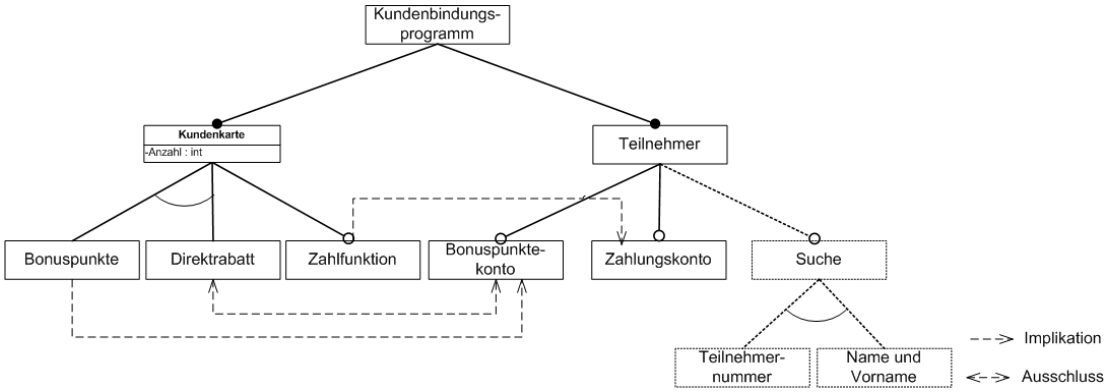
Mithilfe des im letzten Abschnitt definierten Prozesses kann Variabilität in Anwendungsfallbeschreibungen modelliert werden. Um dies zu veranschaulichen wird das Beispiel *UC_01_Teilnehmer_suchen* aus Abschnitt 5.1.4 wieder aufgegriffen (vgl. Abbildung 7.8).

Abbildung 7.9 zeigt das Feature- und Abbildungsmodell mit allen Features, Abbildungsimplikationen und -beeinflussungen für die Anwendungsfallbeschreibung aus



<vp$_1$><v$_1$>		
Name		UC_01_Teilnehmer_suchen
Ziele		•Im Anwendungsfall wird beschrieben, wie ein Teilnehmer gesucht werden kann und seine Stammdaten angezeigt und verändert werden können. **<vp$_2$><v$_1$>** • Die Zahlungsinformationen eines Teilnehmers können angezeigt und verändert werden. **</v$_1$></vp$_2$>**
Akteure		SCA (Service Center Agent) CRM-System (Customer Relationship Management-System)
Vorbedingung		Das CRM System steht zur Verfügung. Der SCA ist im System angemeldet. Der Teilnehmer hat ein Anliegen auf einem der Kommunikationskanäle übermittelt.
Nachbedingung		Der Teilnehmer wurde gefunden.
Alternative Nachbedingung		1.4 Der Teilnehmer wurde nicht gefunden 2.2.2 Der Teilnehmer wurde gefunden und seine Stammdaten verändert **<vp$_3$><v$_1$>** 3.1.2 Der Teilnehmer wurde gefunden und seine Zahlungsinformationen verändert **</v$_1$></vp$_3$>**
Nr.	**Akteure**	**Normaler Ablauf**
1	**-**	**Teilnehmer suchen**
1.1	SCA	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
<vp$_4$><v$_1$>		
1.2	SCA	Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.
</v$_1$> <v$_2$>		
1.2	SCA	Der SCA sucht den Teilnehmer anhand des Namens und Vornamens
</v$_2$></vp$_4$>		
1.3	CRM-System	Ausnahme: Teilnehmer nicht gefunden
1.4	CRM-System	Liste mit zu den Suchkriterien passenden Teilnehmern wird angezeigt
1.5	SCA	SCA wählt einen Teilnehmer aus
1.6	CRM-System	Die Stammdaten des Teilnehmers mit **<vp$_7$>**X**</vp$_7$>** Kundenkarten werden angezeigt
2	**-**	**Alternative: Stammdaten ändern**
<vp$_5$><v$_1$>		
3	**-**	**Alternative: Zahlungsinformationen ändern**
</v$_1$></vp$_5$>		
Nr.	**Akteure**	**Alternative**
2.2.1	SCA	Der SCA will die Stammdaten des Teilnehmers ändern. → UC01_sek_01
<vp$_5$><v$_1$>		
3.1.1	SCA	Der SCA will die Zahlungsinformationen des Teilnehmers ändern → UC01_sek_02
</v$_1$></vp$_5$>		
Nr.	**Akteure**	**Ausnahme**
1.3	System	Rückmeldung: Keine Teilnehmer gefunden
</v$_1$></vp$_1$>		

Abbildung 7.8: Beispiel: Anwendungsfall mit Variabilität



Abbildungsimplikation:

Bindungskonfiguration	Featurebedingung	Bestandteile
BK_1	Suche	$vp_1(v_1)$
BK_2	Zahlfunktion ∧ Zahlungskonto	$vp_2(v_1)$, $vp_3(v_1)$, $vp_5(v_1)$, $vp_6(v_1)$
BK_3	Teilnehmernummer	$vp_4(v_1)$
BK_4	Name und Vorname	$vp_4(v_2)$

Abbildungsbeeinflussung:

Bindungskonfiguration	Wertefunktion	Modifikationspunkte
BK_{10}	Kundenkarte.Anzahl	vp_7

Abbildung 7.9: Featuremodell und Abbildungsmodell für den Anwendungsfall aus Abbildung 7.8

Abbildung 7.8 dar. Das Featuremodell basiert auf dem Beispiel aus Abschnitt 5.2.2. Zu Beginn der Spezifikation wird davon ausgegangen, dass das Abbildungsmodell leer ist und das Featuremodell dem aus Abschnitt 5.2.2 gleicht.
Die Anwendungsfallbeschreibung *UC_01_Teilnehmer_suchen* soll nur unter bestimmten Voraussetzungen Teil des Produktes werden. Daher wird die Anwendungsfallbeschreibung als Variante an einen Additionspunkt assoziiert. Für diese Variante wird eine passende Featurebedingung benötigt. Da keines der Features durch die

Variante $vp_1(v_1)$ konkretisiert wird, wird ein neues, optionales Feature *Suche* eingefügt (Fall 2.b für Additionspunkte aus Abschnitt 7.6). Die Assoziation zwischen dem neuen Feature und der Variante $vp_1(v_1)$ wird über eine neue Abbildungsimplikation mit der Bindungskonfiguration BK_1 spezifiziert. Nur wenn das Feature *Suche* Teil eines abgeleiteten Produktes ist, soll auch die Anwendungsfallbeschreibung *UC_ 01_ Teilnehmer_ suchen* Teil des Produktes sein.
Die Anwendungsfallbeschreibung besitzt einen weiteren Variationspunkt mit einem Ziel als Variante. Diese Variante konkretisiert die Features Zahlfunktion und Zahlungskonto. Es existiert bisher keine Abbildungsimplikation, die eine Featurebedingung mit diesen Features besitzt. Daher wird eine neue Abbildungsimplikation mit der Bindungskonfiguration BK_2 erstellt und die Variante an dieser assoziiert. BK_2 trägt die Features Zahlfunktion und Zahlungskonto als Featurebedingung (Szenario 2.a für Additionspunkte aus Abschnitt 7.6). Der Variationspunkt vp_3 und seine Variante v_1 konkretisiert ebenfalls diese beiden Features. Daher wird die Variante an die Abbildungsimplikation mit der Bindungskonfiguration BK_2 assoziiert (Szenario 1 für Additionspunkte aus Abschnitt 7.6). Der Variationspunkt vp_4 besitzt zwei Varianten, die die Art der Suche bestimmten. Da dafür noch keine passenden Features im Featuremodell existieren, werden die beiden alternativen Features *Teilnehmernummer* und *Name und Vorname* als Sub-Features von Suche eingefügt (Szenario 2.b für Additionspunkte aus Abschnitt 7.6). Die alternativen Features werden in BK_3 und BK_4 als Featurebedingung eingesetzt und werden zu den Varianten v_1 und v_2 von vp_4 assoziiert. vp_7 ist ein Modifikationspunkt, für den eine neue Abbildungsbeeinflussung mit der Bindungskonfiguration BK_{10} erstellt wird. Die Werteabbildung ist abhängig vom Modifikationspunkt *Anzahl* des Features *Kundenkarte* (Szenario 2 für Modifikationspunkte aus Abschnitt 7.6). Die neu in das Featuremodell eingefügten Features sind in Abbildung 7.8 gestrichelt dargestellt.

7.8 Zusammenfassung

In diesem Kapitel wurde ein Spezifikationsprozess für Anforderungen mit Variabilität mit Hilfe der in Kapitel 5 beschriebenen Anforderungsmodellierungssprache mit Variabilität und dem featurebasierten Variabilitätsmanagement beschrieben.
Zunächst erfolgte dazu in Abschnitt 7.1 die Definition eines Prozessmetamodells für die Spezifikation von Prozessen. Dieses Metamodell ermöglicht eine konsistente und für den Leser nachvollziehbare Spezifikation von Prozessen in dieser Arbeit.
Das Prozessmetamodell wurde anschließend in Abschnitt 7.3 genutzt um den Anforderungsspezifikationsprozess zu spezifizieren. Zwei Szenarien waren dabei zu unterscheiden: Zum einen die Identifikation implizierter Variabilität in existierenden Anforderungsmodellen (Abschnitt 7.4) und zum anderen die Spezifikation von Variabilität in neuen Anforderungen (Abschnitt 7.6).
Das Management der Variabilität erfolgte mit Hilfe des featurebasierten Variabilitätsmanagements aus Kapitel 5. Für die Spezifikation des dabei zentralen Featuremodells wurden existierende Ansätze eingesetzt.
Der Anforderungsspezifikationsprozess mit Variabilität wurde anschließend beispielhaft auf Anwendungsfallbeschreibungen angewandt. Der in diesem Kapitel beschriebene Prozess kann darüber hinaus für andere Typen von Modellen eingesetzt werden, sodass die Spezifikation von unterschiedlichen Modellen mit Variabilität für die Plattform in einer einheitlichen Art und Weise möglich ist.
Die dabei entstehenden Modelle mit Variabilität werden im nächsten Kapitel als Eingabe für den Testfallspezifikationsprozess mit Variabilität eingesetzt.

Kapitel 8

Plattformtestfallspezifikationsprozess mit Variabilität

Wie in Kapitel 3.2 beschrieben, adressieren existierende Ansätze für die Spezifikation von Testfällen für den Systemtest von Software-Produktlinien die in Abschnitt 3.1.3 beschriebenen Anforderungen nur unzureichend. Daher wird in diesem Kapitel ein Spezifikationsprozess für Testfälle mit Variabilität beschrieben, der die gestellten Anforderungen erfüllt. Der Testfallspezifikationsprozess nutzt dabei die mit Variabilität behafteten Anforderungsmodelle, deren Spezifikationsprozess im letzten Abschnitt beschrieben wurde. Weiterhin wird auch bei der Spezifikation von Testfällen das in Kapitel 5 eingeführte *featurebasierte Variabilitätsmanagement* für das Management der Variabilität eingesetzt.

8.1 Anforderungen an den Testfallspezifikationsprozess mit Variabilität

Die Grundlage für die Spezifikation von Testfällen für die Teststufe Systemtest bilden Anforderungsmodelle, welche die fachlichen Prozesse des *Systems unter Test* (*SUT*) definieren (vgl. Abschnitt 2.3). Diese Anforderungsmodelle werden typischerweise während der Entwicklungsphase Anforderungsspezifikation definiert. Sie besitzen keine Informationen über die technische Realisierung des SUT. Die Menge dieser Modelle wird in dieser Arbeit als *Fachmodell* (*FM*) bezeichnet (vgl. auch *Platform Independent Model* [GMWE09]).

Um das SUT mit Hilfe eines Testfalls testen zu können, muss der Testfall vor der Testdurchführung durch Informationen aus der technischen Realisierung des SUT angereichert werden. Zum Beispiel sind Informationen über die Struktur von Eingabeparametern notwendig, um korrekte Testdaten für dieses spezifizieren zu können. Die Menge der Modelle, die für diese Anreicherung dienen, werden in dieser Arbeit als *Technikmodell* (*TM*) bezeichnet (vgl. auch *Platform Specific Model* [GMWE09]).

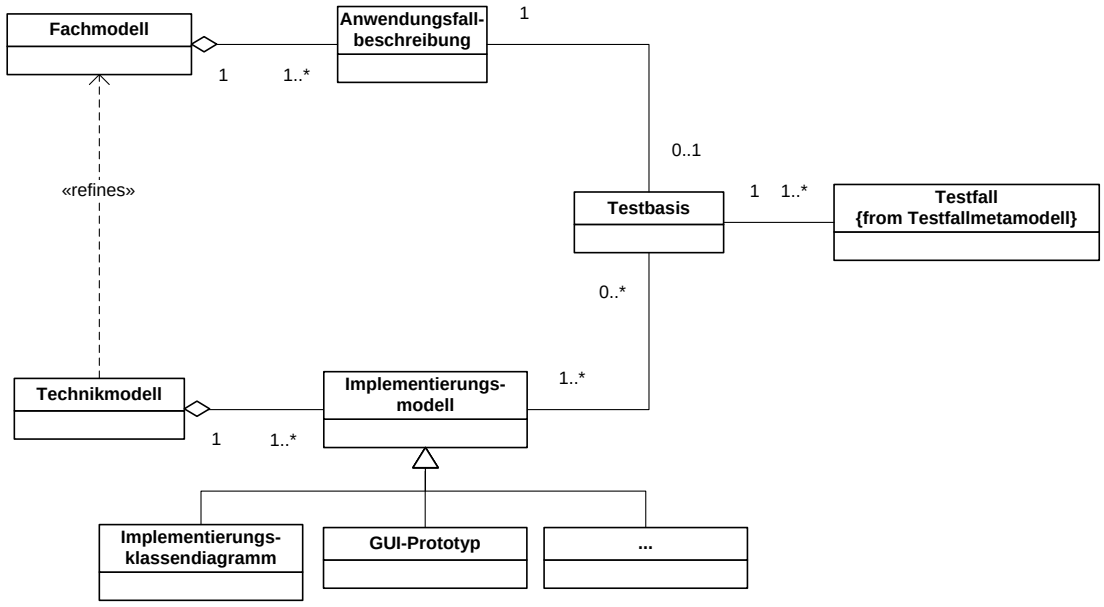


Abbildung 8.1: Testbasismetamodell

In Abbildung 8.1 ist der Zusammenhang zwischen Fach- und Technikmodell in Form eines Metamodells definiert. Im Fachmodell spezifizieren in dieser Arbeit Anwendungsfallbeschreibungen die fachlichen Prozesse des SUT. Das Technikmodell besteht aus Implementierungsmodellen, wie zum Beispiel Implementierungsklassendiagrammen, Schnittstellenbeschreibungen und GUI-Prototypen. Jeweils eine Anwendungsfallbeschreibung und mindestens ein Implementierungsmodell bilden eine Testbasis für mindestens einen Testfall aus dem Testfallmetamodell.
Die Spezifikation von Testfällen für die Plattform einer Produktlinie ist aufgrund der in den Modellen der Testbasis vorhandenen Variabilität ein Problem, welches mit klassischen, nicht-produktlinien Ansätzen nicht oder nur mit hohem Aufwand gelöst werden kann. Der hohe Aufwand entsteht, wenn zum Beispiel für jede Kombination

der in der Plattform vorhandenen Varianten und Modifikationspunkte Testfälle erstellt werden. Der Variationsgrad der Produktlinienplattform steigt mit jedem neuen Variationspunkt approximativ exponentiell an [WO10].
Die Anforderungen an einen Testfallspezifikationsprozess mit Variabilität wurden bereits in Abschnitt 3.2 definiert und für die Bewertung von existierenden Ansätzen genutzt. Die Definition basiert auf der Evaluation der existierenden Literatur aus dem Bereich des Testens von SPL, sowie der praktischen Erfahrung aus dem IndustriekonteUML-Metamodellxt. Die folgenden Punkte fassen die Anforderungen zusammen, die ein Testfallspezifikationsprozess für SPL leisten soll:

- Definition der Testbasis auf Basis von Modellen des Fach- und Technikmodells
- Wiederverwendbarkeit aller Modellelemente eines Testfalls
- Explizite Berücksichtigung der Variabilität der Modelle aus der Testbasis
- Vermeidung von Redundanz zwischen Testfällen
- Spezifikation der Abhängigkeiten zwischen Varianten und Modifikationspunkten innerhalb von Testfällen sowie zu den Modellen der Testbasis

Eine Testbasis besteht aus Modellen des Fach- als auch des Technikmodells. Beide liefern Informationen, damit alle Modellelemente eines Testfalls spezifiziert werden können. Die an eine Testbasis assoziierten Modelle besitzen Variabilität, die die Veränderlichkeit der Produktlinienplattform beschreiben. Der Testfallspezifikationsprozess sollte diese Variabilität berücksichtigen und die Veränderlichkeit auch in Testfällen mit Hilfe von Variabilität modellieren. In diesem Zusammenhang entsteht eine weitere Anforderung: Durch die Modellierung von Variabilität soll die Redundanz bei der Modellierung von Testfällen vermieden werden.

8.2 Überblick über den Testfallspezifikationsprozess mit Variabilität

Der Testfallspezifikationsprozess wird auf Basis des Prozessmetamodells aus Abschnitt 7.1 definiert. Abbildung 8.2 stellt den Prozess, bestehend aus den Prozessschritten *Analyse der Testbasis*, *Spezifikation von logischen Testfällen* und *Spezifikation von konkreten Testfällen* dar. Der Testfallspezifikationsprozess wird durch die

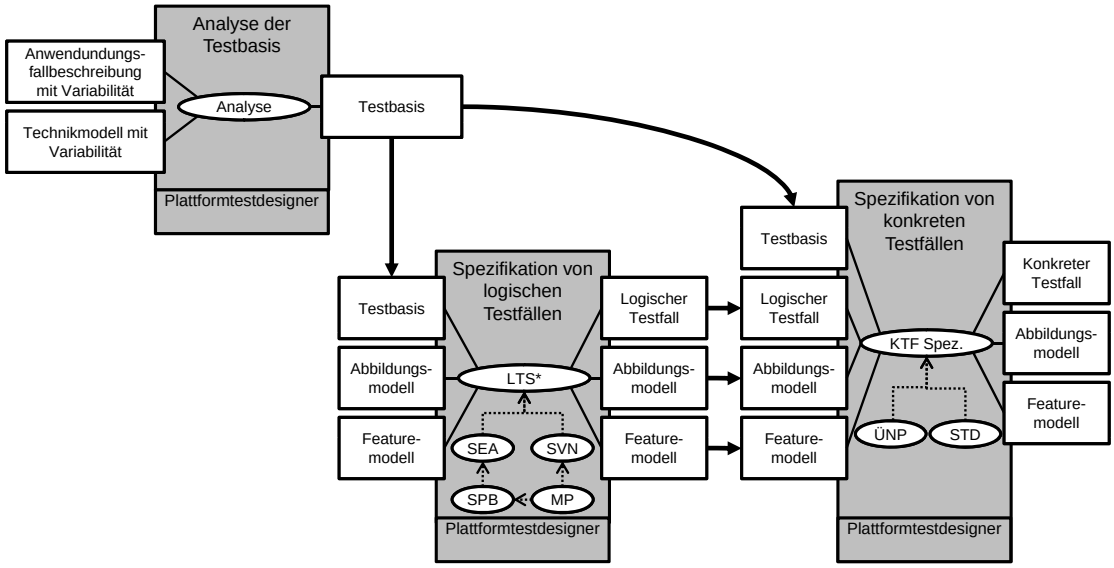


Abbildung 8.2: Überblick über den Testfallspezifikationsprozess

Rolle Plattformtestdesigner durchgeführt (vgl. Abschnitt 2.3)
Im ersten Prozessschritt werden Anwendungsfallbeschreibungen und das Technikmodell analysiert. Dabei entstehen als Ausgabemodelle Testbasen. Eine Testbasis ist zusammen mit dem Feature- und Abbildungsmodell die Eingabe für die Spezifikation von logischen Testfällen. Ausgabe dieses Prozessschrittes sind logische Testfälle und das Feature- sowie Abbildungsmodell. Die letzten beiden Modelle werden in diesem Prozessschritt um die Verwaltung der Variabilität der spezifizierten logischen Testfälle erweitert.
Die Testbasis, die logischen Testfälle, sowie das Feature- und Abbildungsmodell bilden die Eingabemodelle für den letzten Prozessschritt, die Spezifikation von konkreten Testfällen. Ausgabe dieses Prozesses sind dann konkrete Testfälle und das um die Verwaltung der Variabilität der konkreten Testfälle erweitere Feature- und Abbildungmodell.
Die Modellelemente des logischen Testfalls sind in Abbildung 8.3 im dargestellten Testfallmetamodell hervorgehoben. Der logische Testfall umfasst alle Modellelemente, ausgenommen Testdatensätze und deren Testdaten. Ein konkreter Testfall besteht dann aus einem logischen Testfall und einem Testdatensatz mit seinen Testdaten.

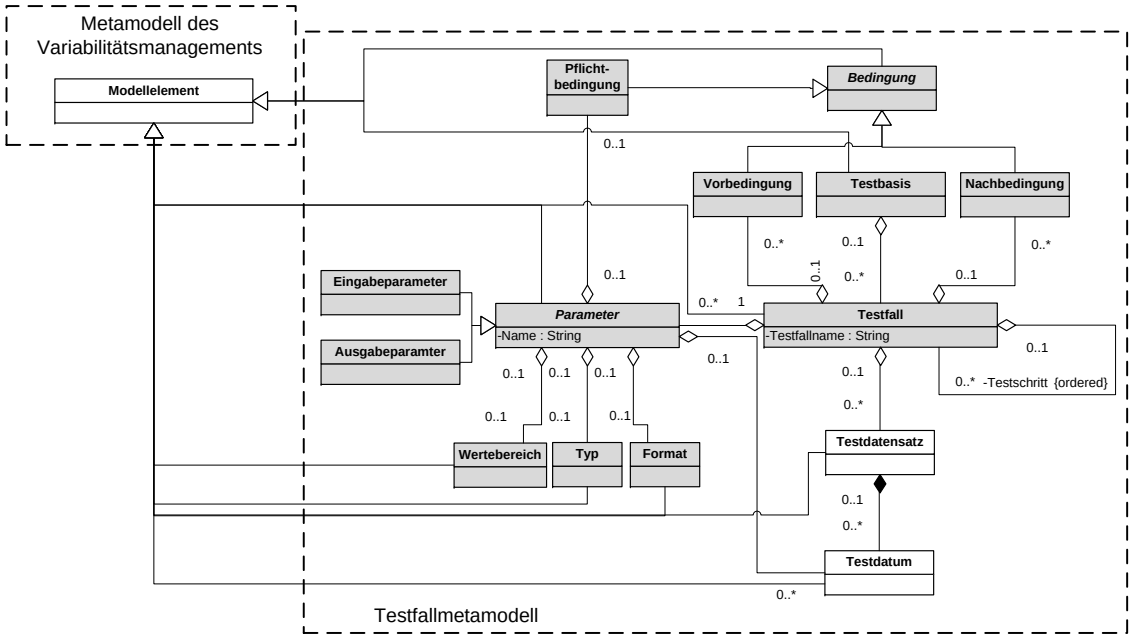


Abbildung 8.3: Testfallmetamodell mit Zuordnung seiner Modellelemente zu logischem und konkretem Testfall

8.3 Analyse der Testbasis

Zwei unterschiedliche Typen von Modellen sind notwendig, um einen Testfall zu spezifizieren. Zum einen Struktur- und zum anderen Prozessspezifikationen. Ein Testfall integriert diese beiden Typen. Beispielsweise definiert die Spezifikation einer Benutzerschnittstelle die Struktur von Eingabeparametern für einen bestimmten Anwendungsfallschritt. Eine Anwendungsfallbeschreibung definiert unterschiedliche Prozesse, die das SUT implementiert.

Die Integration von Prozess- und Strukturspezifikation wird in dieser Arbeit mit Hilfe von Verfeinerungen der Anwendungsfallbeschreibung durch Modellelemente von Implementierungsmodellen realisiert:

Definition 12. *Verfeinerung: Eine Verfeinerung definiert die Verbindung (Assoziation) mindestens eines verfeinernden Modellelements mit mindestens einem zu verfeinernden Modellelement.*

Um die Verfeinerung aus Sicht des Testens zu realisieren, wird ein *Verfeinerungsmetamodell* definiert, wie in Abbildung 8.4 dargestellt ist. Eine *Verfeinerung* kann

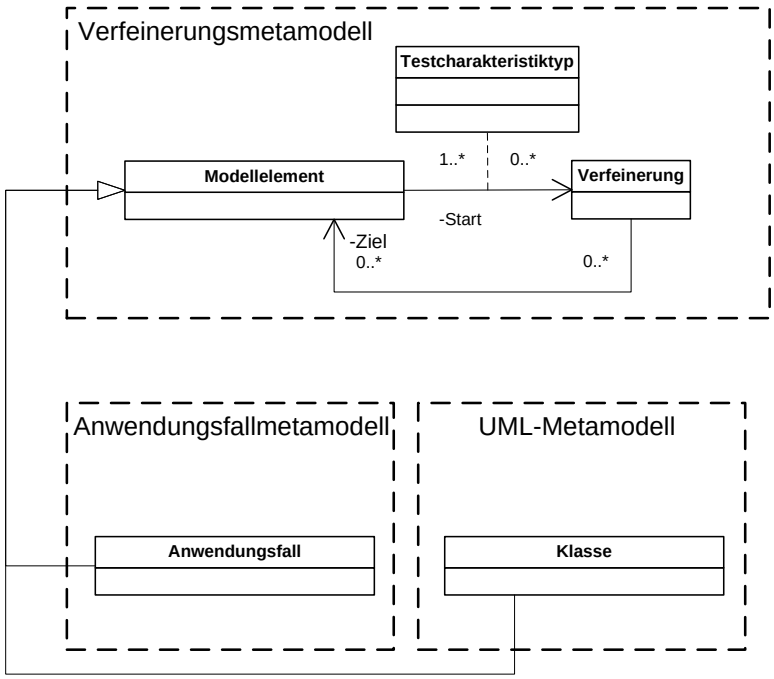


Abbildung 8.4: Das Verfeinerungsmetamodell und seine Spezialisierung durch Anwendungs- und UML-Metamodell

an jedes *Modellelement* spezifiziert werden. Solche Modellelemente, die als Start mit einer Verfeinerung assoziiert werden, erhalten einen Testcharakteristiktyp:

Definition 13. *Testcharakteristiktyp: Ein Testcharakteristiktyp definiert die Art der Verfeinerung eines Modellelements durch ein anderes Modellelement für die Spezifikation von Testfällen.*

Ein Modellelement das ein anderes verfeinert (Start-Modellelement), kann Teil mehrerer Verfeinerungen sein. Eine Verfeinerung muss mindestens ein Start-Modellelement besitzen. Ein verfeinertes Modellelement (Ziel-Modellelement) kann beliebig viele Verfeinerungen besitzen und jede Verfeinerung kann beliebig viele Ziel-Modellelemente verfeinern.
Variabilitätsmodellierungssprachen können durch Spezialisierung der Metaklasse *Modellelement* des Verfeinerungsmetamodells die Modellierung von Verfeinerungen innerhalb eines oder zwischen verschiedenen Modelltypen ermöglichen. In Abbildung 8.4 ist diese Spezialisierung beispielhaft für die Metaklassen *Anwendungsfall* und *Klasse* aus dem *Anwendungsfallmetamodell* bzw. *UML-Metamodell* dargestellt.

In dieser Arbeit werden verschiedene Verfeinerungs- und Testcharakteristiktypen unterschieden (vgl. Abbildung 8.5).

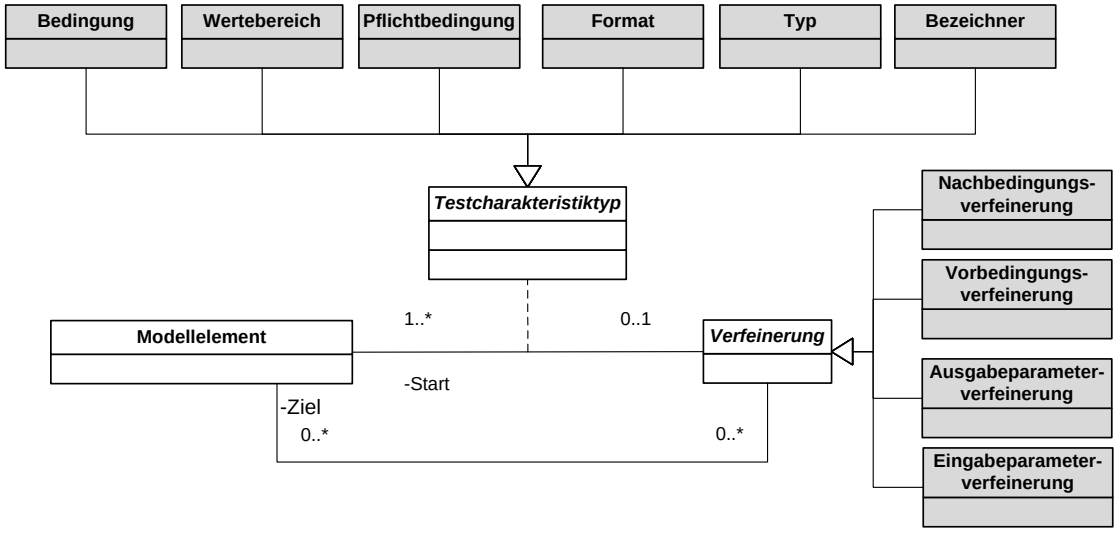


Abbildung 8.5: Typen von Verfeinerungen und Testcharakteristika

Tabelle 8.1 stellt dar, welcher Verfeinerungstyp zusammen mit welchem Testcharakteristiktyp genutzt werden kann. Die Wahl der Verfeinerungs- und Testcharakteristiktypen basiert auf den Metaklassen des Testfallmetamodells, welches in dieser Arbeit zu Grunde gelegt wird (vgl. Abschnitt 5.1). Wird im Testfallspezifikationsprozess ein anderes Testfallmetamodell zugrunde gelegt, müssen unter Umständen andere Spezialisierungen der Verfeinerungs- und Testcharakteristiktypen definiert werden.

Der Testcharakteristiktyp *Bedingung* wird für solche Modellelemente vergeben, die an eine Vor- oder Nachbedingungsverfeinerung assoziiert sind. Diese Einschränkung wird als OCL-Bedingung formuliert:

Bedingung 21. ***Context*** *Bedingung* ***inv:***
self.Verfeinerung.elementType.conformsTo(„Vorbedingungsverfeinerung“) or
self.Verfeinerung.elementType.conformsTo(„Nachbedingungsverfeinerung“)

Die übrigen Testcharakteristiktypen werden für die Verfeinerungen von Ein- und Ausgabeparametern verwendet. Sie entsprechen den an der Metaklasse *Parameter* aus dem Testfallmetamodell aggregierten Metaklassen (vgl. Abbildung 8.3). Diese

Test-charakteristik-typ /Verfeinerungs-typ	**Nach-bedingungs-verfeinerung**	**Vor-bedingungs-verfeinerung**	**Ausgabe-parameter-verfeinerung**	**Eingabe-parameter-verfeinerung**
Bedingung	X	X		
Wertebereich			X	X
Pflicht-bedingung			X	X
Format			X	X
Typ			X	X
Bezeichner			X	X

Tabelle 8.1: Verknüpfung von Verfeinerungs- und Testcharakteristiktypen

Einschränkung wird wiederum durch Bedingungen an den Metaklassen definiert, hier exemplarisch für die Metaklasse Wertebereich:

Bedingung 22. ***Context*** *Wertebereich* ***inv:***
self.Verfeinerung.elementType.conformsTo(„Eingabeparameterverfeinerung“) or
self.Verfeinerung.elementType.conformsTo(„Ausgabeparameterverfeinerung“)

Das Testfallmetamodell definiert, dass jeder Parameter mindestens einen Typ besitzen muss. Weiterhin ist für jeden Parameter auch ein Bezeichner notwendig. Diese Bedingung wird beispielhaft als OCL-Bedingung in folgender Weise formuliert:

Bedingung 23. ***Context*** *Eingabeparameterverfeinerung* ***inv:***
*self.Modellelement→**exists**(b:Modellelement|*
b.Testcharakteristiktyp.conformsTo(„Bezeichner“)

Die Definition der OCL-Bedingungen für den Testcharakteristiktyp „Typ“ für die Eingabeparameterverfeinerung sowie die für „Typ“ und „Bezeichner“ für die Ausgabeparameterverfeinerung erfolgt analog.
Ein- und Ausgabeparameterverfeinerung müssen Schritte als Ziel haben, da mit ihrer Hilfe später Testschritte spezifiziert werden.

Bedingung 24. ***Context*** *Eingabeparameterverfeinerung* ***inv:***
*self.Ziel→**forAll**(z:Ziel|z.conformsTo(„Schritt“)*

Bedingung 25. ***Context*** *Ausgabeparameterverfeinerung* ***inv:***
*self.Ziel→**forAll**(z:Ziel|z.conformsTo(„Schritt“)*

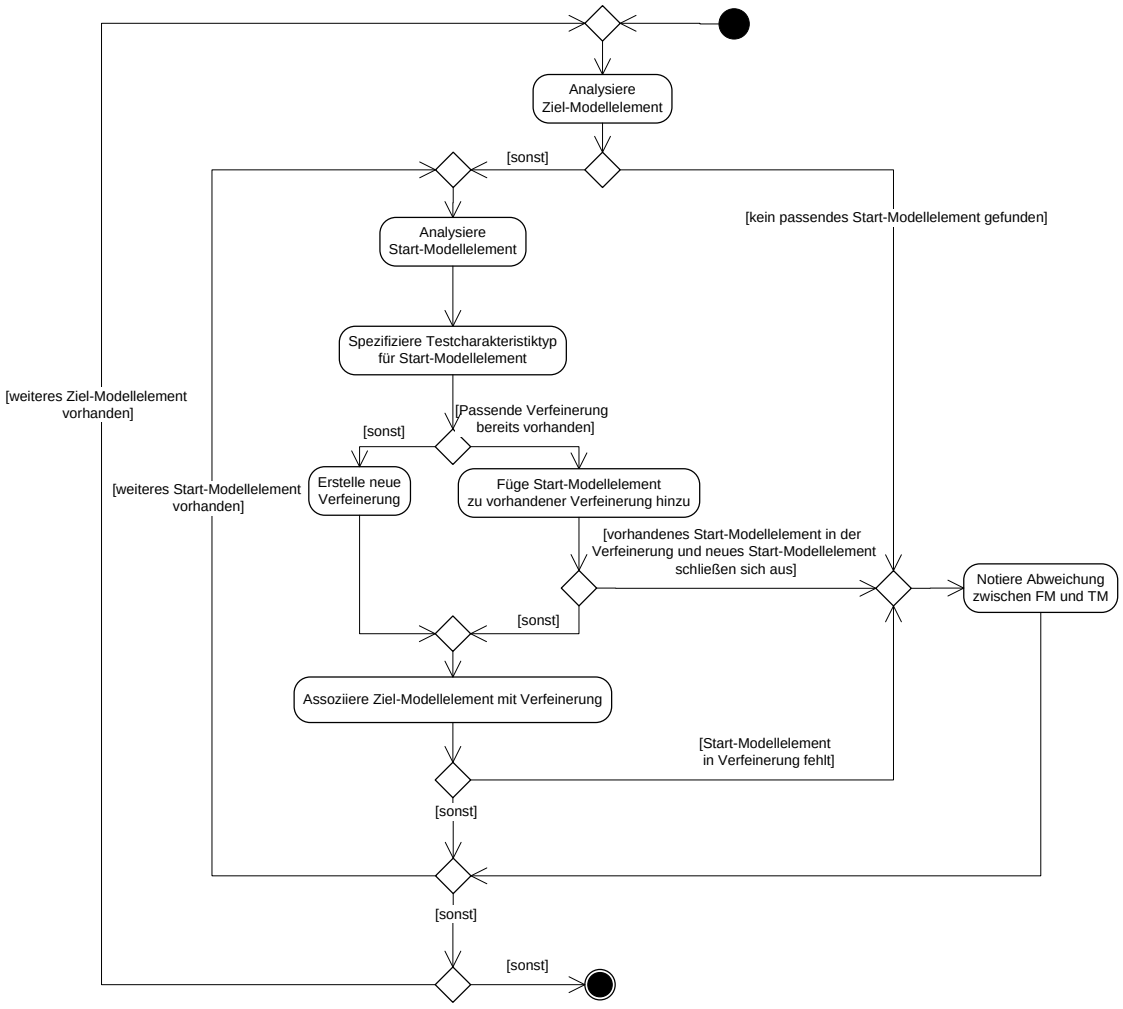


Abbildung 8.6: Algorithmus: **Analyse** der Testbasis

Abbildung 8.6 stellt den Algorithmus für die Analyse der Testbasis und damit der Verfeinerung von Anwendungsfallbeschreibungen durch Implementierungmodelle dar. Der Algorithmus iteriert über alle Elemente eines Anwendungsfalls und assoziiert Start-Modellelemente über Verfeinerungen an diese. Dabei werden vier Szenarien unterschieden:

1. Ein neues Start-Modellelement für das Ziel-Modellelement wurde gefunden und erhält einen Testcharakteristiktyp. Es existiert keine passende Verfeine-

rung: Eine neue Verfeinerung wird erstellt und das Start-Modellelement daran assoziiert. Die Verfeinerung wird dann an das Ziel-Modellelement assoziiert.

2. Ein neues Start-Modellelement für das Ziel-Modellelement wurde gefunden und erhält einen Testcharakteristiktyp. Für das Start-Modellelement existiert bereits eine passende Verfeinerung am Ziel-Modellelement. Eine passende Verfeinerung liegt dann vor, wenn sie einen Typ besitzt, der zum Testcharakteristiktyp des Start-Modellelements passt und dieser inhaltlich zu allen Start-Modellelementen der Verfeinerung passt. Das neue Start-Modellelement wird an der Verfeinerung assoziiert.

3. Es wurde kein passendes neues Start-Modellelement für das Ziel-Modellelement gefunden: Das Fehlen wird als eine Abweichung zwischen Fach- und Technikmodell notiert.

4. Ein neues Start-Modellelement und ein weiteres, bereits an der Verfeinerung assoziiertes Start-Modellelement schließen sich fachlich aus. Dies wird als eine Abweichung zwischen Fach- und Technikmodell notiert. Ein fachlicher Ausschluss liegt vor, wenn zwei Modellelemente mit dem gleichen Testcharakteristiktyp existieren, die die gleiche Funktionalität unterschiedlich definieren. Wenn zum Beispiel zwei Modellelemente den Testcharakteristiktyp „Typ“ besitzen und diese unterschiedliche Ausprägungen besitzen, liegt eine potentielle Abweichung vor. Sind nun die Modellelemente Varianten an einem Additionspunkt und stellen Alternativen dar, liegt wahrscheinlich kein Fehler vor, da in einem abgeleiteten Produkt der Typ festgelegt wäre.

8.3.1 Qualitätssicherung des Fach- und Implementierungsmodells während der Analyse der Testbasis

Während der Analyse der Testbasis kann der Plattformtestdesigner Abweichungen zwischen dem Fach- und Technikmodell feststellen. Es existieren drei Typen von Abweichungen, von denen zwei bereits im Vorgehen aus Abbildung 8.6 berücksichtigt worden sind:

- „[kein passendes Start-Modellelement gefunden]“ und „[Start-Modellelement in Verfeinerung fehlt]“: Ein Modellelement aus dem Fachmodell ist im Technikmodell teilweise oder gesamtheitlich unberücksichtigt geblieben.

- „[vorhandenes Start-Modellelement in der Verfeinerung und neues Start-Modellelement schließen sich aus]“ Im Technikmodell existieren Modellelemente die ein Modellelement in der Anwendungsfallbeschreibung annotieren und sich dabei fachlich ausschließen.

- Im Technikmodell existieren Modellelemente, die kein Modellelement des Fachmodells konkretisieren.

Die Bewertung der Ursache der Abweichung muss manuell durchgeführt werden. Dabei muss die in der Testbasis vorhandene Variabilität berücksichtigt werden.

8.3.2 Konkrete Syntax für die Verfeinerung von Modellelementen

Für die Verfeinerung von Modellelementen in der Testbasis wird eine konkrete Syntax definiert: Ein Ziel-Modellelement ist folgendermaßen spezifiziert:

Modellelement ohne Variabilität:
$<me_x|$ Liste assoziierter Verfeinerungen, durch Kommata getrennt$>$
Modellelement ist Variante:
$<v_x|BK_x|$ Liste assoziierter Verfeinerungen, durch Kommata getrennt$>$

Dabei erhält das Modellelement selbst einen Bezeichner me_x, mit x als eindeutigem Index. Ist das Modellelement bereits eine Variante, besitzt es einen eindeutigen Bezeichner (vgl. Abschnitt 5.1). An das Modellelement werden dann in Form einer Liste die Bezeichner der verfeinernden Modellelemente spezifiziert.

Ein Start-Modellelement wird im folgender Weise definiert:

$Bezeichner_x$:$Testcharakteristiktyp$

Eine Start-Modellelement wird durch einen Bezeichner definiert, bestehend aus der Bezeichnung für den Verfeinerungstyp (in dieser Arbeit sind dies Vorbedingung (VB), Nachbedingung (NB), Eingabeparameter (EP) und Ausgabeparameter (AP)) und einem eindeutigen Index x. Dazu wird ein Testcharakteristiktyp definiert. Der

Wertebereich des Testcharakteristiktyps ist in dieser Arbeit: Bedingung, Wertebereich, Pflichtbedingung, Format, Typ, Bezeichner (vgl. Abbildung 8.5).
Die Bestimmung einer Testbasis und deren Verfeinerungsbeziehungen wird nun anhand eines Beispiels veranschaulicht.

8.3.3 Beispiel für die Analyse einer Testbasis

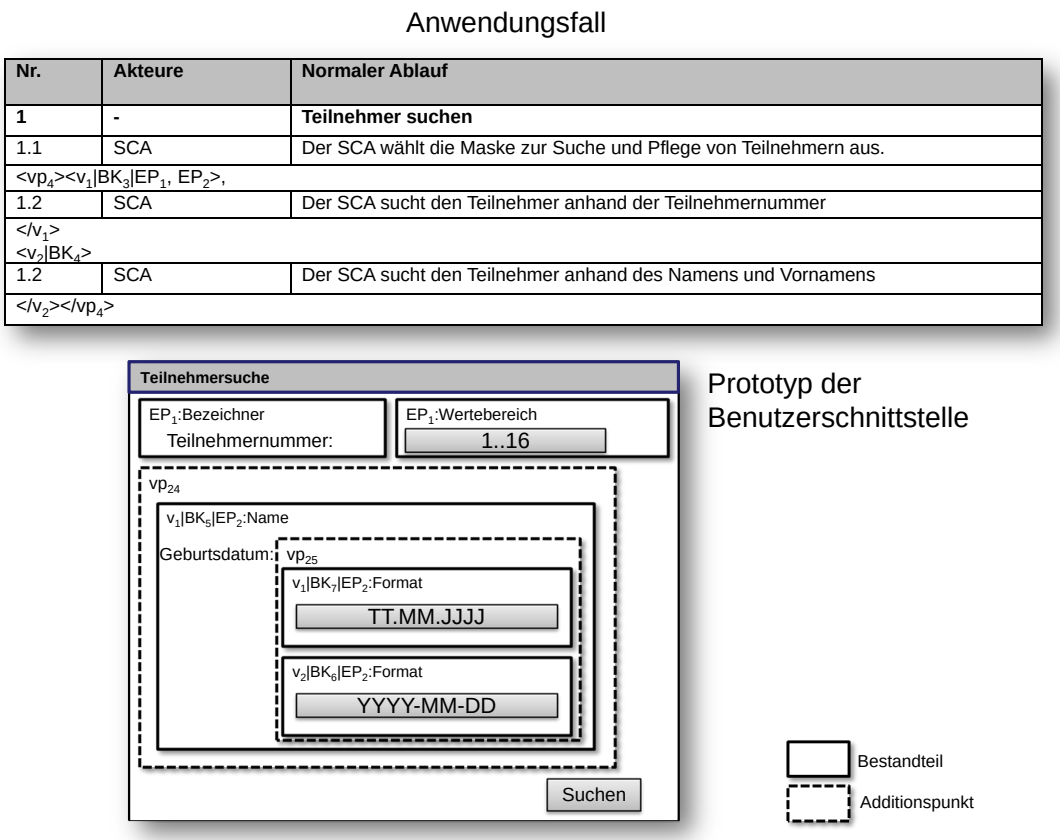


Abbildung 8.7: Verfeinerung eines Anwendungsfalls mit Modellelementen aus einem „Prototyp einer Benutzerschnittstelle“ aus dem TM

Abbildung 8.7 zeigt oben einen Ausschnitt der Anwendungsfallbeschreibung aus Abbildung 7.8 und unten den Prototypen einer Benutzerschnittstelle aus dem Technikmodell. Der Plattformtestdesigner erweitert nun die Variante v_1 von Schritt 1.2 in der Anwendungsfallbeschreibung (Ziel-Modellelement). Für den Test des Schritts können aus dem Prototypen der Benutzerschnittstelle Verfeinerungen zu Eingabe-

parametern definiert werden:
Im Schritt wird die Eingabe einer Teilnehmernummer erwartet. Im Prototypen der Benutzerschnittstelle ist der Eingabeparameter Teilnehmernummer vorhanden (Start-Modellelement). Daher wird für den Eingabeparameter eine neue Eingabeparameterverfeinerung mit dem Bezeichner EP_1 erstellt (Szenario 1 aus Abbildung 8.6). Das Start-Modellelement erhält den Testcharakteristiktyp „Bezeichner“ und wird an EP_1 assoziiert. Die Verfeinerung EP_1 selbst wird an das Ziel-Modellelement v_1 assoziiert.
Im Prototyp der Benutzerschnittstelle aus der Abbildung befindet sich mit der Definition des Wertebereichs der Teilnehmernummer ein weiteres Start-Modellelement, das die Variante v_1 des Schritts im Anwendungsfalls verfeinert. Dieses Start-Modellelement passt zum Eingabeparameter EP_1, da der Eingabeparameter Teilnehmernummer näher spezifiziert wird. Das Start-Modellelement erhält den Testcharakteristiktyp „Wertebereich“ und wird an die existierende Verfeinerung EP_1 assoziiert (Szenario 2 aus Abbildung 8.6).
Im Prototypen der Benutzerschnittstelle existiert mit *Geburtsdatum* ein weiteres Start-Modellelement, das einen Eingabeparameter für das Ziel-Modellelement spezifiziert. Für den Eingabeparameter wird eine neue Eingabeparameterannotation mit dem Bezeichner EP_2 erstellt und an die Variante v_1 des Schritts 1.2 der Anwendungsfallbeschreibung assoziiert. Alle zu dieser Verfeinerung passenden Start-Modellelemente erhalten Testcharakteristiktypen und werden an die Verfeinerung assoziiert.
Alle Start-Modellelemente, die in dieser Art und Weise an Ziel-Modellelemente einer Anwendungsfallbeschreibung über Verfeinerungen assoziiert werden, sind Teil der Testbasis dieser Anwendungsfallbeschreibung. Im Anhang B.1 ist zur Erhaltung der Konsistenz des laufenden Beispiels die Erweiterung des Feature- und Abbildungsmodells um die variablen Modellelemente des „Prototypen der Benutzerschnittstelle“ aus Abbildung 8.7 beschrieben.

Eine Besonderheit bei der Analyse der Testbasis stellen Schritte in der Anwendungsfallbeschreibung dar, die keinen externen Akteur besitzen. Diese Schritte werden durch das System unter Test durchgeführt. Da es sich beim Test auf Basis von Anwendungsfallbeschreibungen um einen Blackbox-Test der Teststufe Systemtest handelt, können diese Schritte bei der Testdurchführung nicht durch den Testtrei-

ber durchgeführt werden. Um die Korrektheit der Durchführung dieser Schritte zu testen, kann der Zustand des Systems nach Durchführung des Schrittes entweder durch Ausgabeparameter oder über die Nachbedingung eines Testschritts überprüft werden. Daher verfeinern Schritte in der Anwendungsfallbeschreibung ohne externen Akteur Schritte mit externem Akteur oder eine Nachbedingung des Anwendungsfalls.

8.4 Spezifikation von logischen Testfällen

Die Spezifikation von logischen Testfällen teilt sich in drei Abschnitte auf. Zunächst wird beschrieben, wie auf Basis der Anwendungsfallbeschreibung Testfälle und dazugehörige Schritte spezifiziert werden. Nach der Definition der Testschritte wird die Spezifikation der Ein- und Ausgabeparameter sowie der Vor- und Nachbedingungen an diesen beschrieben.

8.4.1 Spezifikation von Testschritten in Testfällen

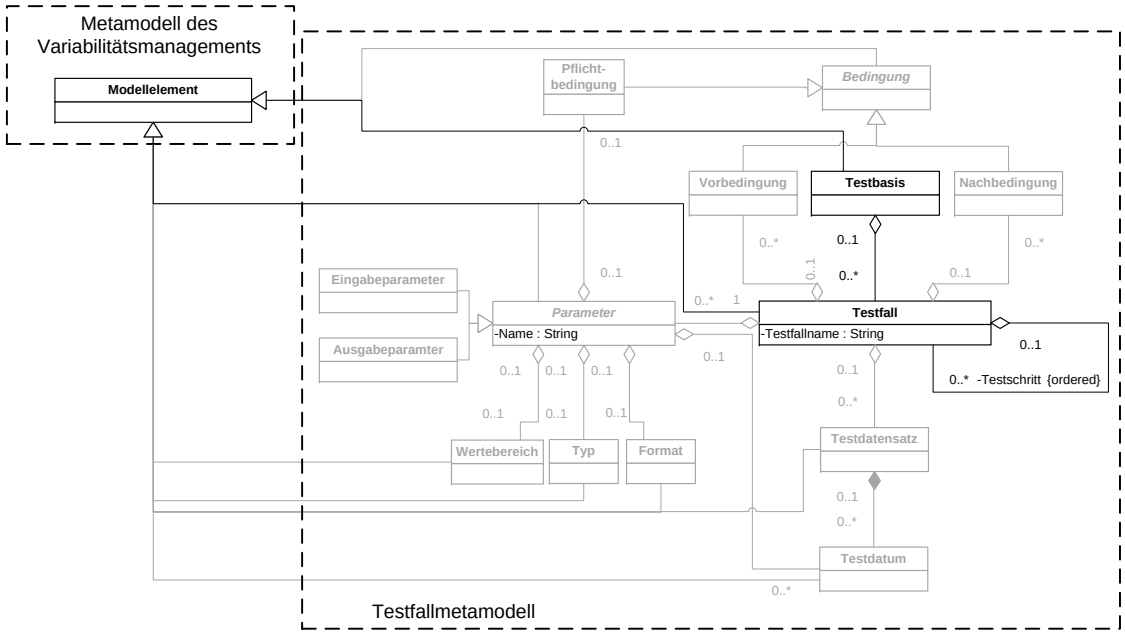


Abbildung 8.8: Testschritte im Testfallmetamodell

Die Spezifikation eines logischen Testfalls startet mit der Spezifikation des Testfalls selbst und seiner Testschritte, wie in Abbildung 8.8 hervorgehoben wird. Für die Spezifikation der Testschritte eines Testfalls werden in dieser Arbeit die in der Anwendungsfallbeschreibung der Testbasis definierten Abläufe genutzt. Zu jeweils einem Ablauf des Anwendungsfalls wird ein Testfall spezifiziert, deren Testschritte die Schritte des Ablaufs testen. Ein Ablauf stellt eine Folge von Schritten dar, startend mit dem ersten Schritt des Anwendungsfalls und endend mit einem Schritt ohne Nachfolger. Um die Abläufe in Anwendungsfällen überdecken zu können, existieren verschiedene Überdeckungskriterien, wie zum Beispiel Zweig- oder Pfadüberdeckung (vgl. Abschnitt 2.3).

Abbildung 8.9 stellt einen Algorithmus für die Spezifikation von Testfällen auf Basis eines beliebigen Überdeckungskriteriums dar. Für jeden Ablauf, der durch das Überdeckungskriterium abgedeckt wird, wird ein neuer Testfall erstellt. Im Anschluss daran werden die Vor- und Nachbedingungen des Testfalls spezifiziert, gefolgt von den einzelnen Testschritten. Die Spezifikation der Testschritte basiert auf den Schritten des betrachteten Ablaufs. Dabei existieren vier Szenarien:

1. Der Schritt im Anwendungsfall besitzt keinen externen Akteur. Es wird kein Testschritt für diesen Schritt erstellt.
   (a) Es existiert im Ablauf ein weiterer Schritt mit dem fortgefahren wird.
   (b) Es existiert kein weiterer Schritt im Ablauf oder das zuvor definierte Abbruchkriterium ist erfüllt.
2. Der Testschritt im Anwendungsfall besitzt einen externen Akteur. Es wird ein neuer Testschritt für diesen Schritt erstellt und anschließend werden die Algorithmen für die Spezifikation der Ein- und Ausgabeparameter sowie der Vor- und Nachbedingungen aufgerufen.
   (a) Es existiert im Ablauf ein weiterer Schritt mit dem fortgefahren wird.
   (b) Es existiert kein weiterer Schritt im Ablauf oder das zuvor definierte Abbruchkriterium ist erfüllt.

Ein Abbruchkriterium definiert Regeln für den Abbruch der Spezifikation eines Testfalls und wird nach der Bearbeitung jeden Schritts des betrachteten Ablaufes überprüft. Somit kann die Spezifikation abgebrochen werden, obwohl der letzte

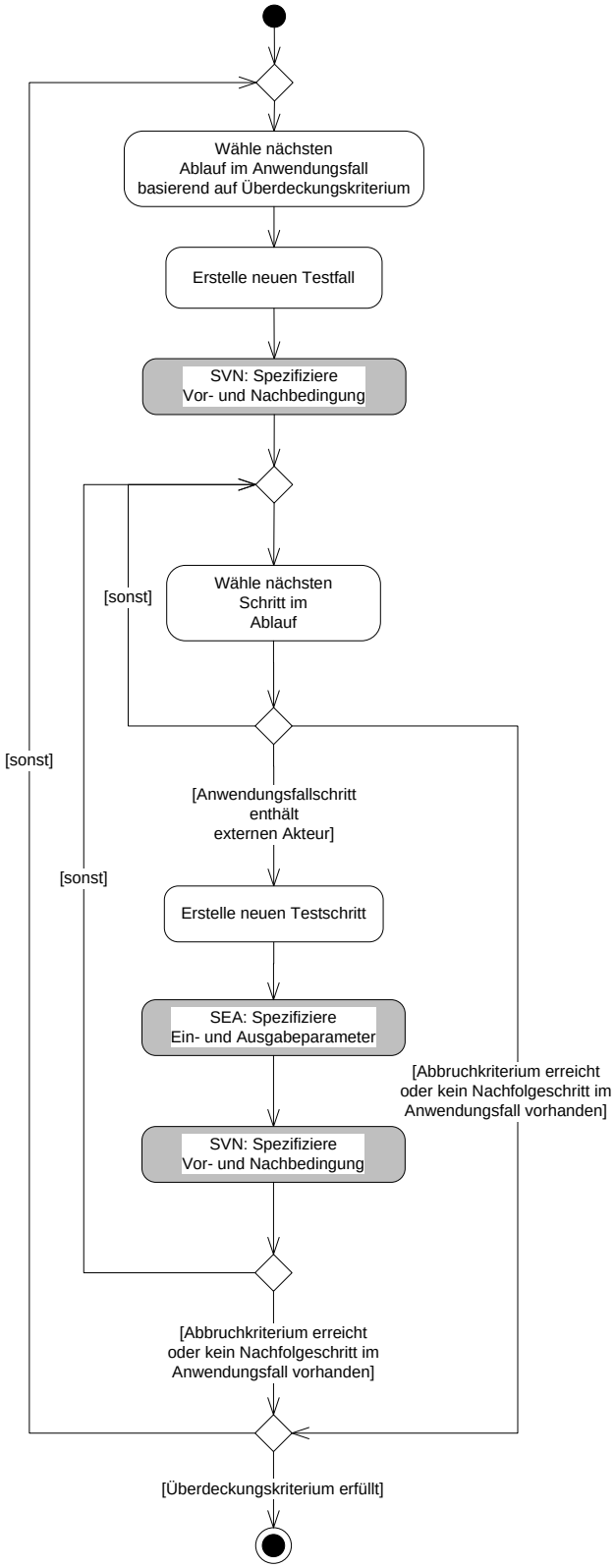


Abbildung 8.9: Algorithmus **LTS**: Logische Testfallspezifikation ohne Variabilität

Schritt des Ablaufs noch nicht erreicht worden ist.
Der in Abbildung 8.9 dargestellte Algorithmus funktioniert für Anwendungsfallbeschreibungen, die keine Variabilität besitzen. Damit er die in den Anwendungsfallbeschreibungen vorhandene Variabilität berücksichtigen kann, muss er erweitert werden.

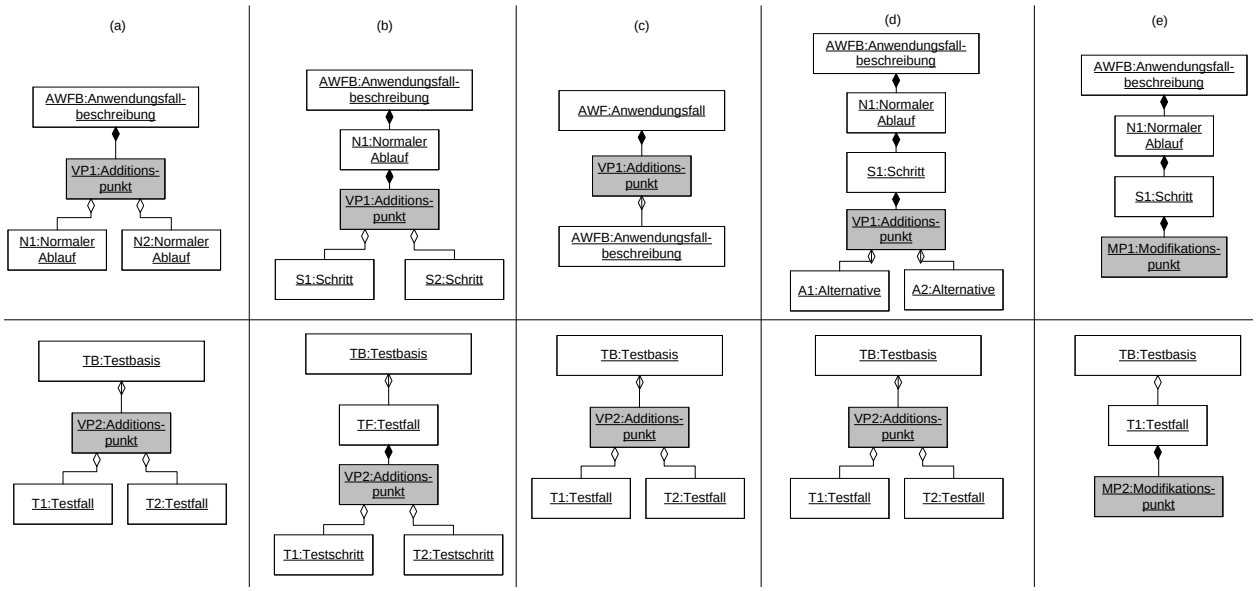


Abbildung 8.10: Variabilität in Anwendungsfällen mit Relevanz für die Spezifikation von Testfällen und ihren Testschritten

Für die Spezifikation von Testfällen mit Variabilität muss das jeweils gewählt Überdeckungskriterium auch mit Variabilität umgehen können:

- Jedes Szenario, welches eine Variante darstellt, wird durch das Überdeckungskriterium berücksichtigt.
- Additionspunkte mit Schritten als Varianten werden vom Überdeckungskriterium ignoriert, da sie bei der Spezifikation von Testfällen selbst Berücksichtigung finden werden.

Die Erweiterung von Überdeckungskriterien um Variabilität wird in der Literatur durch [RKPR05b] und [CDS06] behandelt und kann für den hier definierten Algorithmus verwendet werden.
Abbildung 8.10 stellt die fünf möglichen Fälle von Variabilität beispielhaft an Ausschnitten aus Anwendungsfallbeschreibungen in abstrakter Syntax dar. Der jeweilige Variationspunkt ist hervorgehoben:

- Fall (a): Die Anwendungsfallbeschreibung besitzt einen Additionspunkt, an dem normale Abläufe als Varianten assoziiert sind. Es wird ein neuer Additionspunkt an der Testbasis erstellt und für jede Variante des normalen Ablaufs ein Testfall als Variante des Additionspunktes erstellt.

- Fall (b): Ein Szenario (normaler Ablauf, Alternative oder Ausnahme) besitzt einen Additionspunkt an dem Schritte als Varianten assoziiert sind. Es wird ein neuer Additionspunkt am Testfall erstellt und für jede Variante des Schritts im Szenario ein Testschritt als Variante erstellt.

- Fall (c): Eine Anwendungsfallbeschreibung ist als Variante an einem Additionspunkt eines Anwendungsfalls assoziiert: Es wird ein neuer Additionspunkt an der Testbasis assoziiert und alle Testfälle, die zu dieser Anwendungsfallbeschreibung erstellt werden, daran als Varianten assoziiert. Nur wenn die Anwendungsfallbeschreibung teil eines Produktes ist, sind auch ihre Testfälle Teil des Produktes.

- Fall (d): Ein Schritt enthält einen Additionspunkt mit Alternativen (Ausnahmen) als Varianten. Der Ablauf, der diese Alternativen (Ausnahmen) beinhaltet, ist nur Teil eines Produktes, falls diese Teil eines Produktes sind. Daher wird ein neuer Additionspunkt an der Testbasis definiert und die Testfälle, die diese Alternativen (Ausnahmen) testen, hinzugefügt.

- Fall (e): Ein Schritt enthält einen Modifikationspunkt. Im Testfall (Testfallschritt) wird ein Modifikationspunkt spezifiziert.

Der Algorithmus für die Spezifikation eines logischen Testfalls ist in Abbildung 8.11 dargestellt. Er stellt eine Erweiterung des Algorithmus LTC aus Abbildung 8.9 dar.

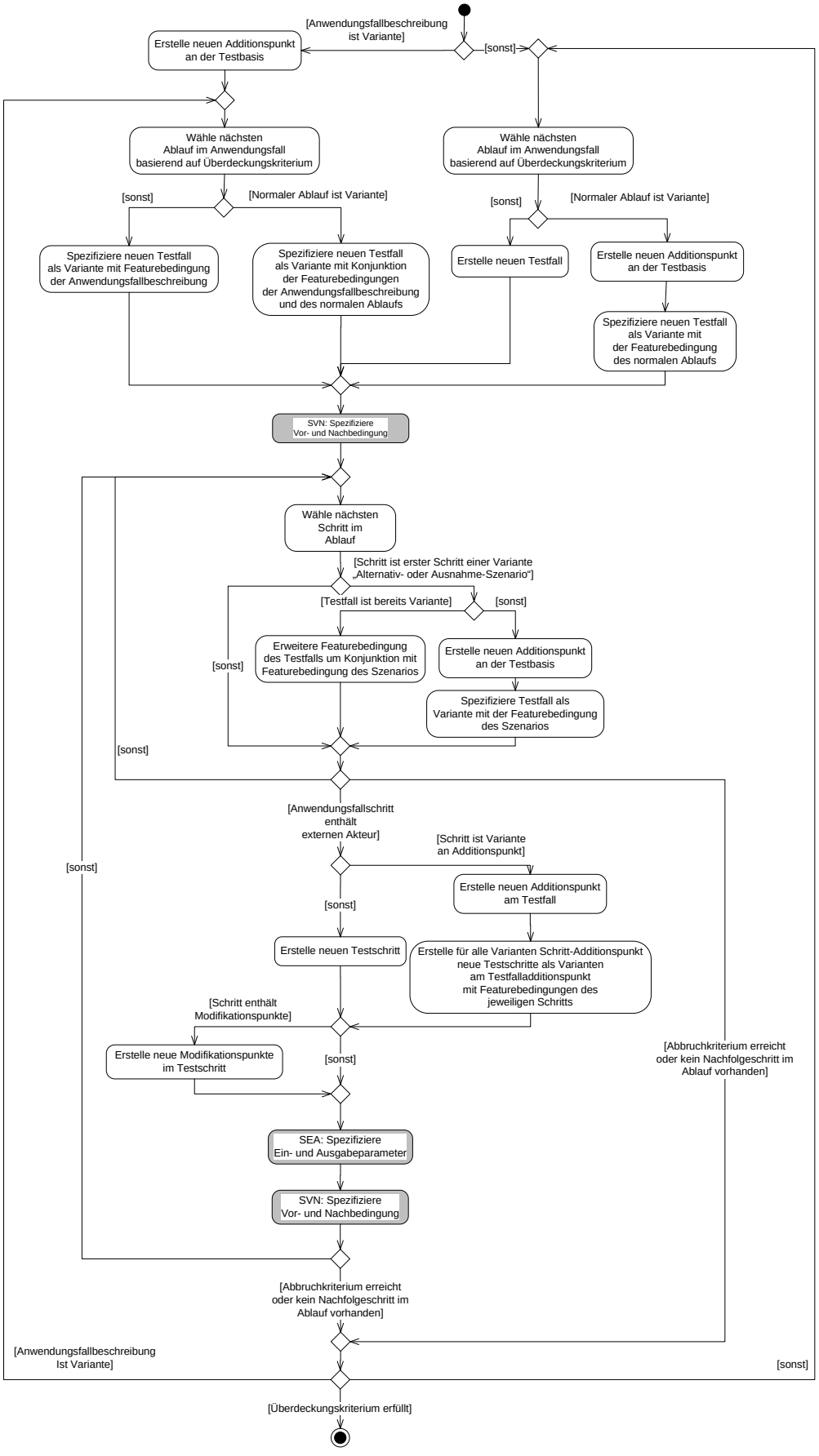


Abbildung 8.11: Algorithmus **LTS***: Spezifikation von logischen Testfällen

Für die Erweiterung wurden die in Abbildung 8.10 dargestellten Fälle von Variabilität genutzt, die für die Spezifikation von Testfällen und Testschritten wichtig sind. Im Folgenden werden die Erweiterungen in der Reihenfolge der Ausführung im Algorithmus beschrieben:

1. **Fall (c):** Falls die Anwendungsfallbeschreibung eine Variante ist, müssen auch alle ihre Testfälle Varianten sein. Dies wird zu Beginn des Algorithmus überprüft („[Anwendungsfallbeschreibung ist Variante]“). In diesem Fall wird ein neuer Additionspunkt an der Testbasis erstellt und alle Testfälle zu dieser Anwendungsfallbeschreibung als Varianten an diesen spezifiziert. Die Featurebedingung der Testfallvarianten ist die gleiche wie die der Anwendungsfallbeschreibung.

2. **Fall (a):** Jeder Ablauf in einer Anwendungsfallbeschreibung beinhaltet den ersten Schritt des normalen Ablaufs. Falls der normale Ablauf selbst eine Variante darstellt, ist der Testfall nur relevant, falls dieser normale Ablauf Teil eines Produktes ist. Dies wird im Algorithmus durch „[Normaler Ablauf ist Variante]“ überprüft. Dabei werden zwei Situationen unterschieden:

   (a) Die Anwendungsfallbeschreibung ist gleichzeitig auch eine Variante: Es wird ein neuer Additionspunkt an der Testbasis erstellt, an den der neue Testfall als Variante spezifiziert wird. Die Featurebedingung des neuen Testfalls ist die Konjunktion der Featurebedingungen von Anwendungsfallbeschreibung und normalem Ablauf, da der Testfall nur für ein Produkt ausgewählt wird, falls sowohl die Anwendungsfallbeschreibung als auch der normale Ablauf für ein Produkt gewählt werden.

   (b) Die Anwendungsfallbeschreibung ist keine Variante: Es wird ein neuer Additionspunkt an der Testbasis erstellt und der neue Testfall als Variante davon spezifiziert. Der Testfall erhält die gleiche Featurebedingung, wie der normale Ablauf.

3. **Fall (d):** Schritte aus einem Alternativ- oder Ausnahme-Szenario, welches eine Variante darstellt, sind Teil des Ablaufs („[Schritt ist erster Schritt einer Variante „Alternativ- oder Ausnahme-Szenario]“):

(a) Testfall ist bereits aufgrund von Variabilität der Anwendungsfallbeschreibung oder des normalen Ablaufs Variante: Die Featurebedingung des Testfalls wird um die Konjunktion der Featurebedingung des Szenarios erweitert. Der Testfall wird damit nur für ein Produkt gewählt, falls das Szenario gewählt wird.

(b) Testfall ist keine Variante: Es wird ein neuer Additionspunkt an der Testbasis erstellt, an den der neue Testfall als Variante spezifiziert wird. Die Featurebedingung des neuen Testfalls ist die des Szenarios, da der Testfall nur für ein Produkt ausgewählt wird, falls das Szenario gewählt wird.

4. **Fall (b):** Der Ablauf enthält einen Additionspunkt mit Schritten als Varianten („[Schritt ist Variante an Additionspunkt]“): Am Testfall wird ein neuer Additionspunkt erstellt, an den für jede Schritt-Variante eine neue Testschritt-Variante spezifiziert wird. Die Featurebedingung jeder Testschritt-Variante ist die der dazugehörigen Schritt-Variante.

5. **Fall (e):** Enthält ein Schritt Modifikationspunkte werden diese auch im Testschritt spezifiziert („[Schritt enthält Modifikationspunkte]“).

Im Anhang A.2 ist das Vorgehen für die Spezifikation von logischen Testfällen in Form von zwei Algorithmen definiert (Algorithmus 4 und 5).

Kritik

Der vorgestellte Algorithmus LTC* ist auf Anwendungsfallbeschreibungen mit Variabilität anwendbar, die Instanzen des Anwendungsfallmetamodell aus Abschnitt 5.1.2 darstellen. Für die Anwendung des Algorithmus bei anderen Arten von Modellen, die Prozesse definieren, müsste das Vorkommen von Variabilität wiederum analysiert und der Algorithmus daran angepasst werden.
Je nach Art des Modells kann der Algorithmus entweder automatisiert oder manuell ausgeführt werden. Im letzten Fall dient er dem Plattformtestdesigner als Handlungsanweisung für die manuelle Erstellung von Testfällen mit Testschritten.

Ein klassisches Problem der Testfallspezifikation ist der Umgang mit Schleifen in Ablaufbeschreibungen [UL07]. Bei einer Spezifikation von Testfällen, zum Beispiel für alle Pfade eines Anwendungsfalls, kann der Abbruch der Testfallspezifikation, abhängig von der Bedingung der Schleife, niemals erreicht werden. Dieses Problem kann zum Beispiel durch Definition eines Abbruchkriteriums mit einer bestimmten Anzahl an Schleifendurchläufen gelöst werden.

Beispiel: Spezifikation von Testschritten

<vp$_1$><v$_1$>		
Name		UC_01_Teilnehmer_suchen
…		**•…**
Nr.	**Akteure**	**Normaler Ablauf**
1	**-**	**Teilnehmer suchen**
1.1	SCA	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
<vp$_4$><v$_1$>		
1.2	SCA	Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.
</v$_1$> <v$_2$>		
1.2	SCA	Der SCA sucht den Teilnehmer anhand des Namens und Vornamens
</v$_2$></vp$_4$>		
1.3	CRM-System	Ausnahme: Teilnehmer nicht gefunden
1.4	CRM-System	Liste mit zu den Suchkriterien passenden Teilnehmern wird angezeigt
1.5	SCA	SCA wählt einen Teilnehmer aus
1.6	CRM-System	Die Stammdaten des Teilnehmers mit **<vp$_7$>**X**</vp$_7$>** Kundenkarten werden angezeigt
2	**-**	**Alternative: Stammdaten ändern**
<vp$_5$><v$_1$>		
3	**-**	**Alternative: Zahlungsinformationen ändern**
</v$_1$></vp$_5$>		
Nr.	**Akteure**	**Alternative**
2.2.1	SCA	Der SCA will die Stammdaten des Teilnehmers ändern. → UC01_sek_01
<vp$_5$><v$_1$>		
3.1.1	SCA	Der SCA will die Zahlungsinformationen des Teilnehmers ändern → UC01_sek_02
</v$_1$></vp$_5$>		
Nr.	**Akteure**	**Ausnahme**
1.3	System	Rückmeldung: Keine Teilnehmer gefunden
</v$_1$></vp$_1$>		

Abbildung 8.12: Ausschnitt aus UC_01_Teilnehmer_suchen

Im Ablauf der Anwendungsfallbeschreibung *UC_01_Teilnehmer_suchen* (vgl. Abbildung 8.12) existieren, bei einer beispielhaft angenommen Pfadüberdeckung, vier unterschiedliche Pfade, die mit Hilfe des beschriebenen Vorgehens zu vier logischen Testfällen spezifiziert werden. Als Beispiel wird der Ablauf

$$1 \rightarrow 1.1 \rightarrow vp_4 \rightarrow 1.4 \rightarrow 1.5 \rightarrow 1.6 \rightarrow 3$$

betrachtet.
Der resultierende logische Testfall ist in Abbildung 8.13 ohne Ein- und Ausgabeparameter sowie Vor- und Nachbedingungen dargestellt.

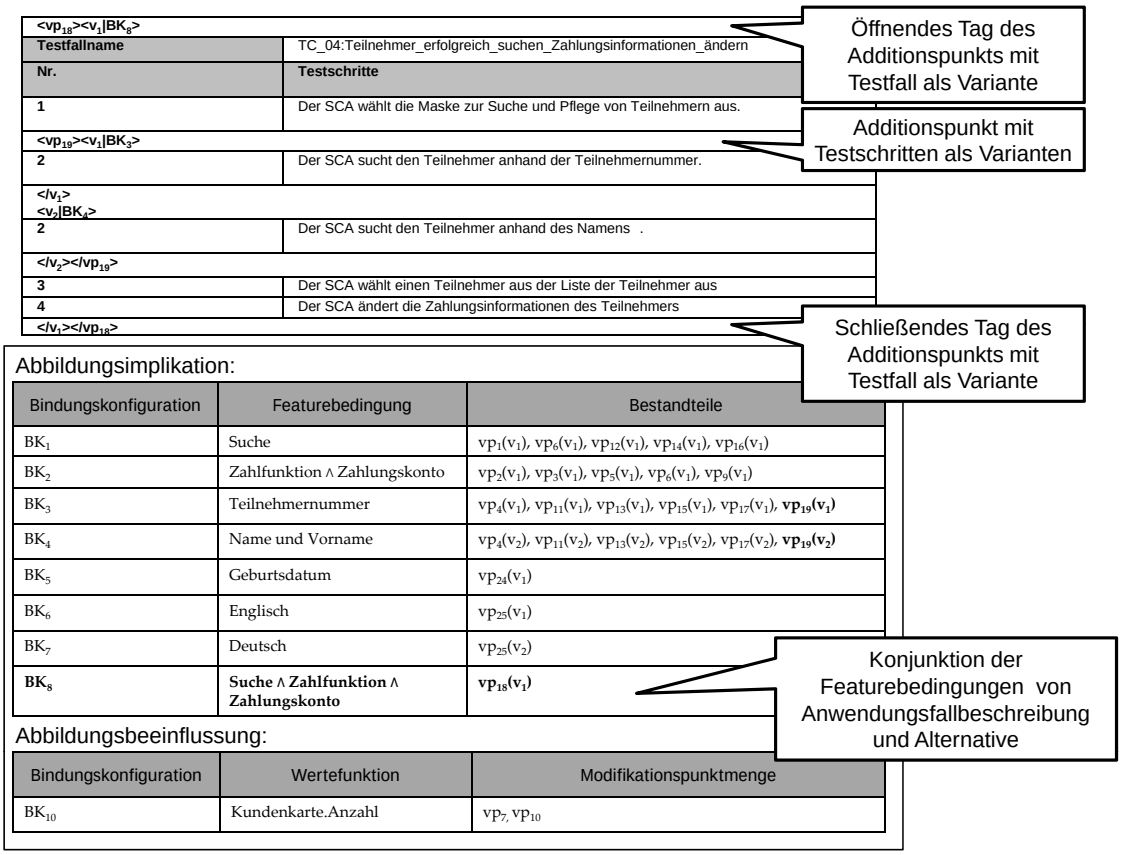


Abbildung 8.13: Beispiel Testfall und Abbildungsmodell

Zum einen testet der Testfall eine Anwendungsfallbeschreibung, die als Variante an den Additionspunkt vp_1 spezifiziert ist. Zum anderen wird eine Alternative getestet, die auch eine Variante am Additionspunkt vp_5 darstellt. Nur wenn sowohl die Anwendungsfallbeschreibung als auch die Alternative Teil eines Produktes sind,

ist es auch der Testfall. Daher wird der Testfall selbst als Variante an den Additionspunkt $vp_1 8$ spezifiziert, mit der Konjunktion der beiden Featurebedingungen *Suche* (Anwendungsfallbeschreibung) und *Zahlfunktion* $\wedge$ *Zahlungskonto* (Alternative). Da es noch keine Abbildungsimplikation mit dieser Featurebedingung gibt, wird eine neue Abbildungsimplikation mit der Bindungskonfiguration BK_8 erstellt (vgl. Abbildung 8.13 unten).
Im Testfall ist der Additionspunkt vp_{19} spezifiziert worden, der zwei Testschritte als Varianten besitzt. Diese basieren auf den Schritten, die als Varianten an den Additionspunkt vp_4 assoziiert sind.
Die neu erstellten Varianten des Testfalls sind zum Abbildungsmodell hinzugefügt worden (vgl. Abbildung 8.13 unten). Die übrigen logischen Testfälle zur Anwendungsfallbeschreibung *UC_ 01_ Teilnehmer_ suchen* sind in Anhang B.2 spezifiziert.

8.4.2 Spezifikation von Ein- und Ausgabeparametern

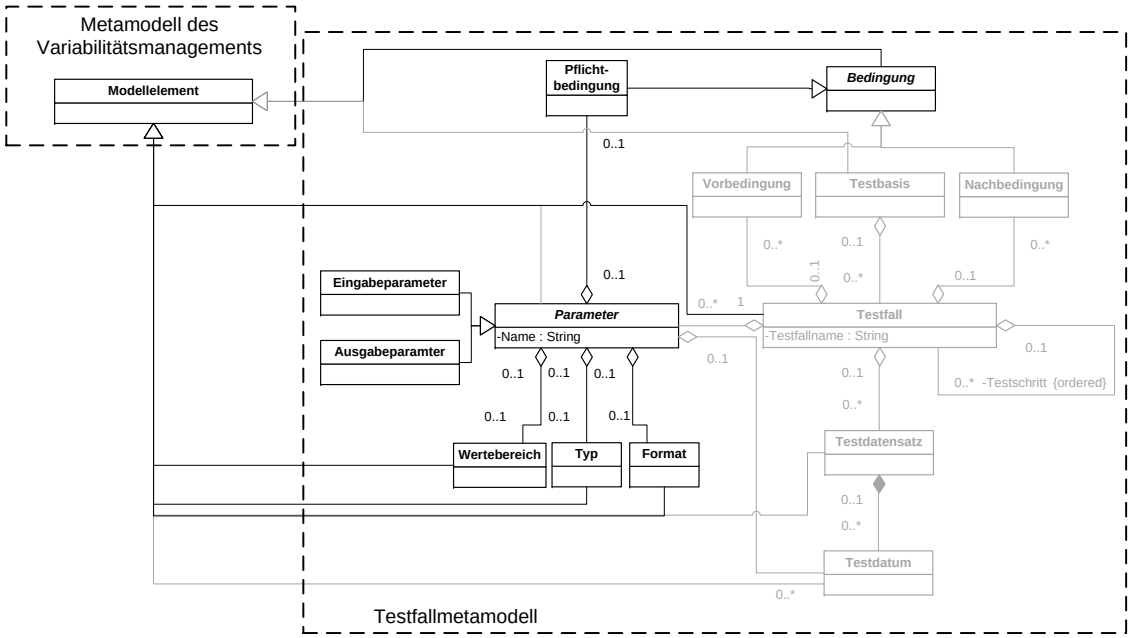


Abbildung 8.14: Ein- und Ausgabeparameter im Testfallmetamodell

Abbildung 8.14 hebt die Metaklassen des Testfallmetamodells hervor, für deren Spezifikation in diesem Abschnitt Algorithmen definiert werden. Jeder Testschritt kann *Ein-* und *Ausgabeparameter* besitzen. Jeder *Parameter* besitzt die *Bestandteile*

Name und *Typ* (zum Beispiel Datum). Zusätzlich kann der *Wertebereich* und das *Format* (zum Beispiel YYYY-MM-DD für ein Datum) angegeben werden. Darüber hinaus kann für Parameter auch eine *Pflichtbedingung* definiert werden, die spezifiziert unter welcher Bedingung der Parameter eingegeben werden muss.
Jeder Testschritt testet in dieser Arbeit einen Schritt einer Anwendungsfallbeschreibung. Um die Parameter des Testschritts zu spezifizieren, werden die Verfeinerungen an dem jeweiligen Schritt verwendet.
Der Algorithmus für die Spezifikation von Ein- und Ausgabeparametern ist in Abbildung 8.15 dargestellt. Er wird im Algorithmus LTS* bei der Spezifikation eines jeden Testschritts aufgerufen (vgl. Abbildung 8.2).

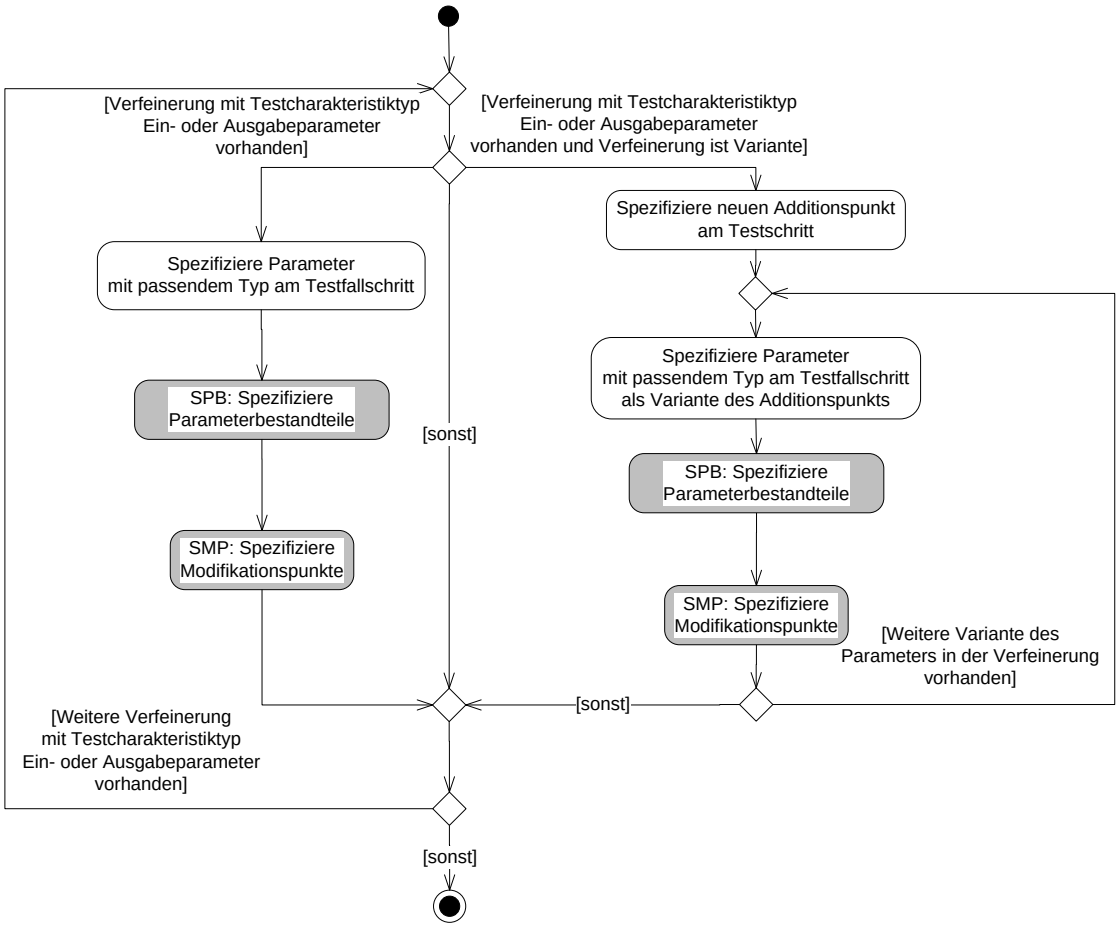


Abbildung 8.15: Algorithmus **SEA**: Spezifikation von Ein- und Ausgabeparametern an Testfallschritten

Für jeden Schritt der Anwendungsfallbeschreibung wird überprüft ob Verfeinerungen mit dem Testcharakteristiktyp Ein- oder Ausgabeparameter vorhanden sind. Eine Verfeinerung enthält Informationen genau zu einem Ein- oder Ausgabeparameter und seinen Bestandteilen.
Dabei werden nun zwei Szenarien unterschieden:

- Die Verfeinerung ist eine Variante: In diesem Fall wird am Testschritt ein neuer Additionspunkt erstellt und für jede Variante des Parameters eine neue Ein- oder Ausgabeparametervariante, abhängig vom Testcharakteristiktyp, am Additionspunkt des Testschritts spezifiziert. Falls eine Verfeinerungsvariante Modifikationspunkte enthält, werden diese auch in der Parametervariante spezifiziert.

- Die Verfeinerung ist keine Variante: Am Testschritt wird ein neuer Ein- oder Ausgabeparameter, abhängig vom Testcharakteristiktyp, spezifiziert. Falls die Verfeinerung Modifikationspunkte enthält, werden diese auch im Parameter spezifiziert.

Für jeden Parameter wird nun der *Algorithmus SPB* ausgeführt. Dabei werden die Bestandteile des Parameters spezifiziert. Diese sind *Pflichtbedingung*, *Format*, *Wertebereich* und *Typ*. Abbildung 8.16 stellt den Algorithmus dar. Die Spezifikation aller vier Typen von Bestandteilen erfolgt in der gleichen Art und Weise:
Zunächst wird überprüft ob eine Verfeinerung mit dem passenden Testcharakteristiktyp vorhanden ist. Ist diese eine Variante, wird am Parameter ein neuer Additionspunkt spezifiziert und für jede Variante der Verfeinerung eine Variante am Bestandteil spezifiziert. Falls die Verfeinerung Modifikationspunkte enthält, werden auch diese spezifiziert.
Ist die Verfeinerung keine Variante, wird der Bestandteil direkt an den Parameter spezifiziert. Falls die Verfeinerung Modifikationspunkte enthält, werden auch diese spezifiziert.
Bei den Bestandteilen *Pflichtbedingung*, *Format* und *Wertebereich* ist es möglich, dass keine Verfeinerung existiert. Für den Bestandteil *Typ* muss eine Verfeinerung existieren, da jeder Parameter einen Typ besitzen muss.
Die Spezifikation von Modifikationspunkten (*Algorithmus SMP*) ist zur besseren

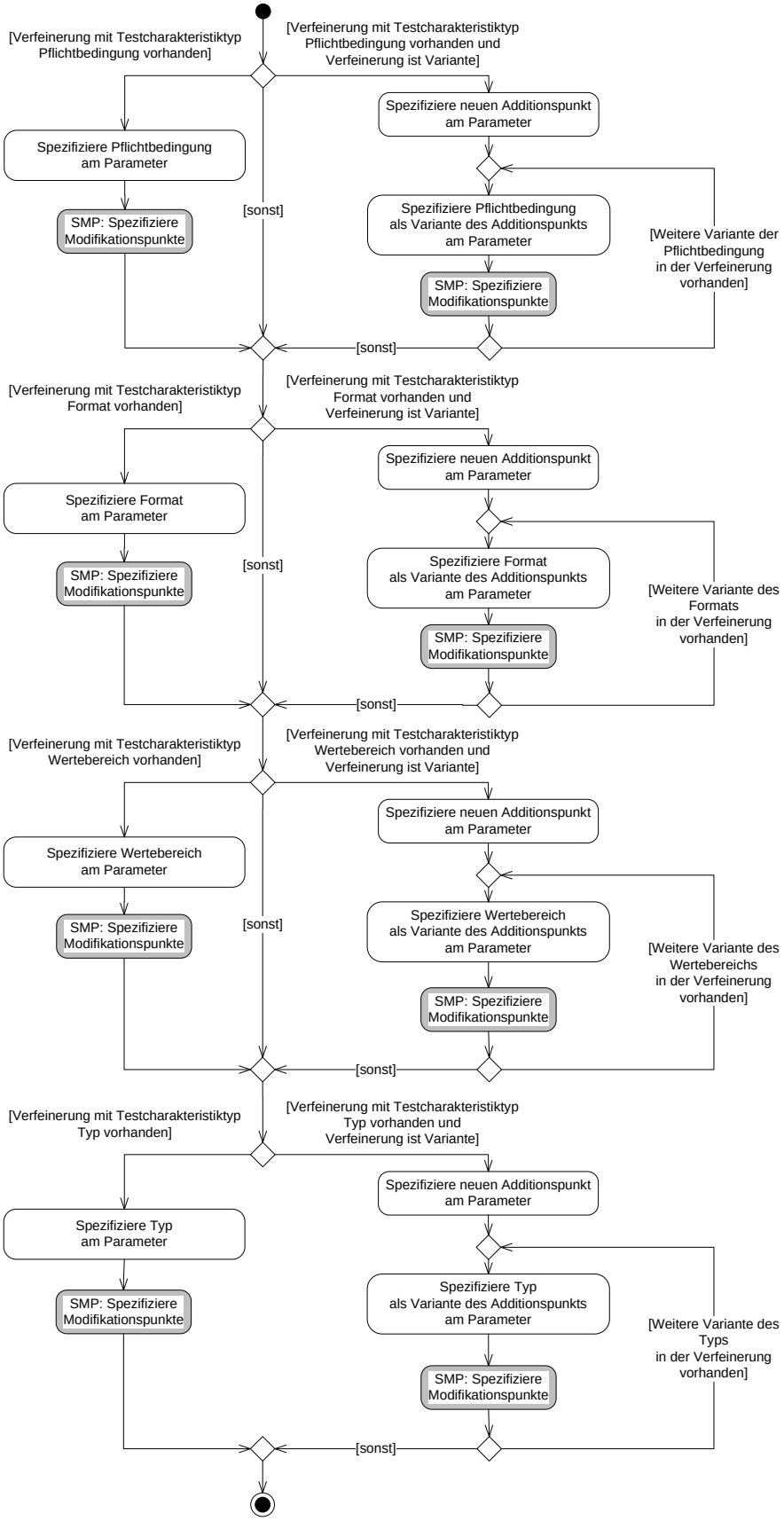


Abbildung 8.16: Algorithmus **SPB**: Spezifikation Parameterbestandteil

Übersicht aus *Algorithmus SPB* ausgegliedert worden und in Abbildung 8.17 dargestellt. Dabei wird überprüft, ob die betrachtete Verfeinerung Modifikationspunkte besitzt und auf dieser Basis werden dann Modifikationspunkte im betrachteten Bestandteil spezifiziert.

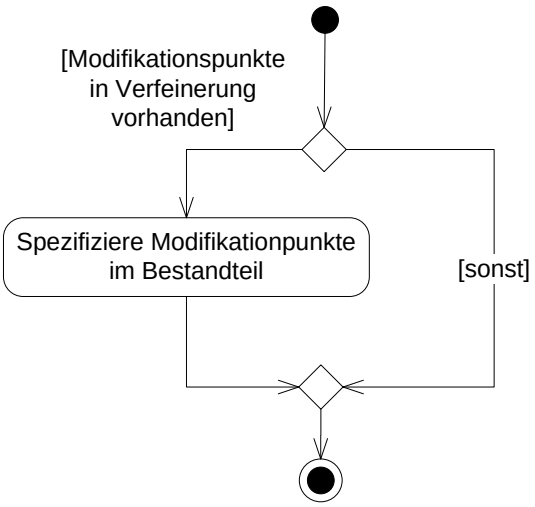


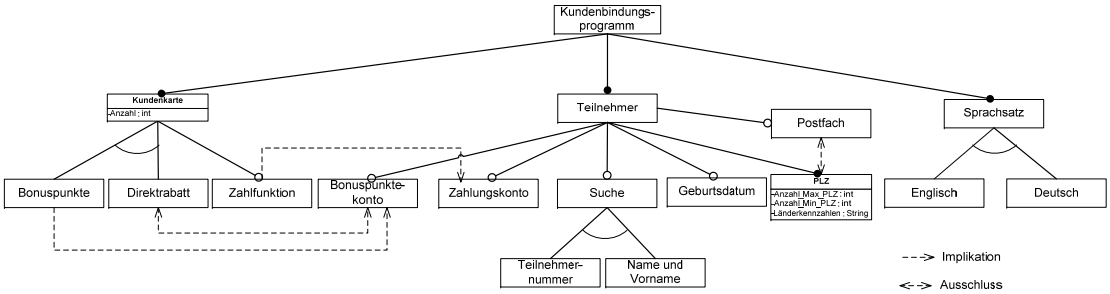
Abbildung 8.17: Algorithmus **SMP**: Spezifikation von Modifikationspunkten

Beispiel: Spezifikation von Ein- und Ausgabeparametern

<vp$_{30}$><v$_1$>							
Testfallname	TC_01_1_Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.						
Vorbedingung	Die Maske „Suche und Pflege“ wird angezeigt.						
						Testdaten-satz 1	**...**
	Name	**Wertebereich**	**Format**	**Typ**	**Pflicht-bedingung**		
Eingabeparameter:	Anzahl	1.. 9999999999		Numerisch			
Eingabeparameter:	Geburtsdatum		**<vp$_{31}$><v$_1$|BK$_7$>** TT-MM-JJJJ **</v$_1$>** **<v$_2$|BK$_6$>** YYYY-MM-DD **</v$_2$></vp$_{31}$>**	Datum			
Eingabeparameter:	PLZ	**<vp$_{32}$|BK$_{11}$>**X**</vp$_{32}$>** … **<vp$_{39}$|BK$_{13}$>**X**</vp$_{39}$>**		Numerisch	**<vp$_{33}$><v$_1$|BK$_9$>** :¬Postfach **</v$_1$>** **<v$_2$|BK$_{10}$>** True **</v$_2$></vp$_{33}$>**		
<vp$_{10}$><v$_1$|BK$_6$>							
Eingabeparameter:	Postfach	**<vp$_{34}$><v$_1$|BK$_7$>** 10000000..99999999 **</v^1>** **<v$_2$|BK$_6$>** **<vp$_{35}$|BK$_{12}$>X</vp$_{35}$>** 10000000..99999999 **</v$_2$></vp$_{34}$>**	**<vp$_{36}$><v$_1$|BK$_6$>** LL-ZZZZZZZZ **</v$_1$></vp$_{36}$>**	**<vp$_{37}$><v$_1$|BK$_7$>** numerisch **</v$_1$>** **<v$_2$|BK$_6$>** Alpha-numerisch **</v$_2$></vp$_{37}$>**	¬PLZ		
</v$_1$><vp$_{10}$>							
Ausgabeparameter:							
Nr.	**Testschritte**						
-	-						
Nachbedingung	Eine Liste mit Teilnehmern wird angezeigt.						
</v$_1$></vp$_{30}$>							

Abbildung 8.18: Beispiel für einen Testfall mit Ein- und Ausgabeparametern und Variabilität

Als Beispiel werden für den ersten Testschritt „Der SCA sucht den Teilnehmer anhand der Teilnehmernummer“ aus dem Testfall aus Abbildung 8.13 Eingabeparameter spezifiziert. Der Testfall mit seinen Eingabeparametern ist in Abbildung 8.18 dargestellt. Für die Spezifikation der Parameter wurden Verfeinerungen der Anwendungsfallbeschreibung aus dem Technikmodell genutzt. Einige der Verfeinerungen sind beispielhaft in Anhang B.3 spezifiziert. Die neu entstandenen Varianten und Modifikationspunkte wurden in das Abbildungsmodell übernommen. Es ist zusammen mit dem Featuremodell in Abbildung 8.19 dargestellt.



Abbildungsimplikation:

Bindungs-konfiguration	Featurebedingung	Bestandteile
BK_1	Suche	$vp_1(v_1)$, $vp_6(v_1)$, $vp_{12}(v_1)$, $vp_{14}(v_1)$, $vp_{16}(v_1)$, $\mathbf{vp_{30}(v_1)}$
BK_2	Zahlfunktion ∧ Zahlungskonto	$vp_2(v_1)$, $vp_3(v_1)$, $vp_5(v_1)$, $vp_6(v_1)$, $vp_9(v_1)$
BK_3	Teilnehmernummer	$vp_4(v_1)$, $vp_{11}(v_1)$, $vp_{13}(v_1)$, $vp_{15}(v_1)$, $vp_{17}(v_1)$, $vp_{19}(v_1)$
BK_4	Name und Vorname	$vp_4(v_2)$, $vp_{11}(v_2)$, $vp_{13}(v_2)$, $vp_{15}(v_2)$, $vp_{17}(v_2)$, $vp_{19}(v_2)$
BK_5	Geburtsdatum	$vp_{24}(v_1)$
BK_6	Englisch	$vp_{25}(v_1)$, $\mathbf{vp_{31}(v_2)}$, $\mathbf{vp_{34}(v_2)}$, $\mathbf{vp_{36}(v_1)}$, $\mathbf{vp_{37}(v_2)}$
BK_7	Deutsch	$vp_{25}(v_2)$, $\mathbf{vp_{31}(v_1)}$, $\mathbf{vp_{34}(v_1)}$, $\mathbf{vp_{37}(v_1)}$
BK_8	Suche ∧ Zahlfunktion ∧ Zahlungskonto	$vp_{18}(v_1)$
BK_9	**Postfach**	**$vp_{33}(v_1)$**
BK_{10}	**¬Postfach**	**$vp_{33}(v_2)$**

Abbildungsbeeinflussung:

Bindungs-konfiguration	Wertefunktion	Modifikationspunktmenge
BK_{10}	Kundenkarte.Anzahl	vp_7, vp_{10}
BK_{11}	**PLZ.Max_PLZ**	**vp_{32}**
BK_{12}	**PLZ.Länderkennzahlen= {DE, GB, F, ESP}**	**vp_{35}**
BK_{13}	**PLZ.Min_PLZ**	**vp_{39}**

Abbildung 8.19: Beispiel für das Feature- und Abbildungsmodell zum Testfall aus Abbildung 8.18

8.4.3 Spezifikation von Vor- und Nachbedingungen

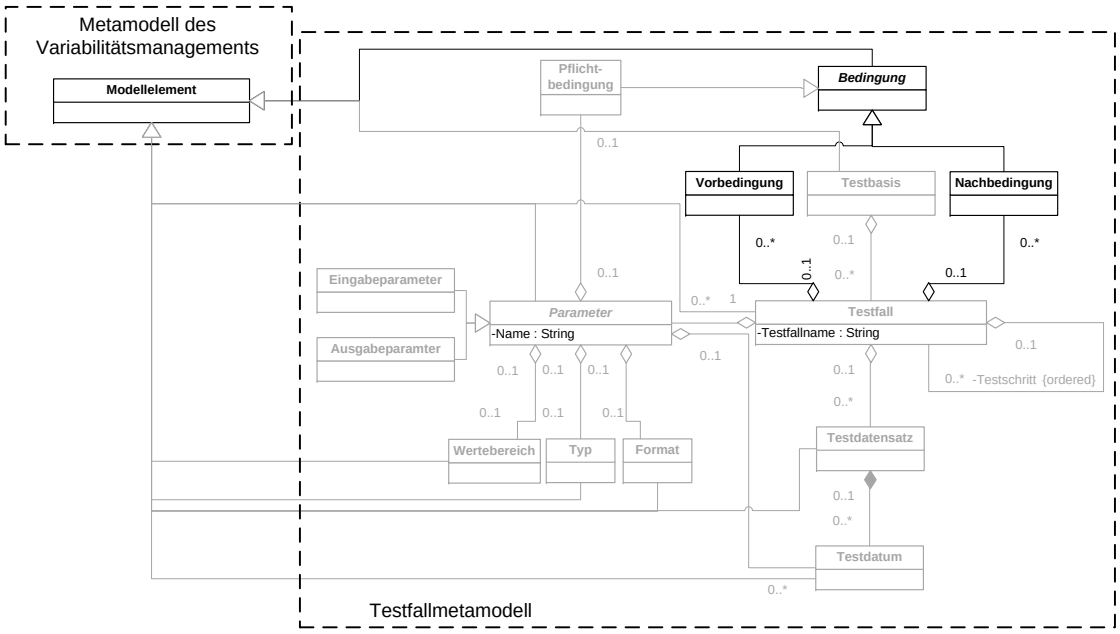


Abbildung 8.20: Vor- und Nachbedingungen im Testfallmetamodell

Neben der Spezifikation von Ein- und Ausgabeparametern an Testfallschritten, besitzen logische Testfälle auch *Vor-* und *Nachbedingungen*, wie in Abbildung 8.20 hervorgehoben ist.
Die Spezifikation dieser Bedingungen geschieht analog zu Ein- und Ausgabeparametern und ist als *Algorithmus SVN* in Abbildung 8.21 dargestellt. Der Algorithmus wird in Algorithmus LTS* an zwei Stellen aufgerufen:

- Zunächst werden für jeden neu erstellten Testfall die Vor- und Nachbedingungen spezifiziert. Für die Spezifikation der Vorbedingungen werden die Vorbedingungen der Anwendungsfallbeschreibung und ihre Verfeinerungen verwendet. Die Nachbedingungen werden in Abhängigkeit des gewählten Ablaufs der Anwendungsfallbeschreibung wiederum auf Basis der Nachbedingungen der Anwendungsfallbeschreibung und ihrer Verfeinerungen spezifiziert.

- Für jeden neu spezifizierten Testschritt werden wiederum Vor- und Nachbedingungen spezifiziert, da ein Testschritt wiederum einen Testfall darstellt. Für die Spezifikation werden die Verfeinerungen mit dem Testcharakteristiktyp Vor- bzw. Nachbedingung genutzt, die am jeweiligen Schritt des Anwendungsfalls assoziiert sind.

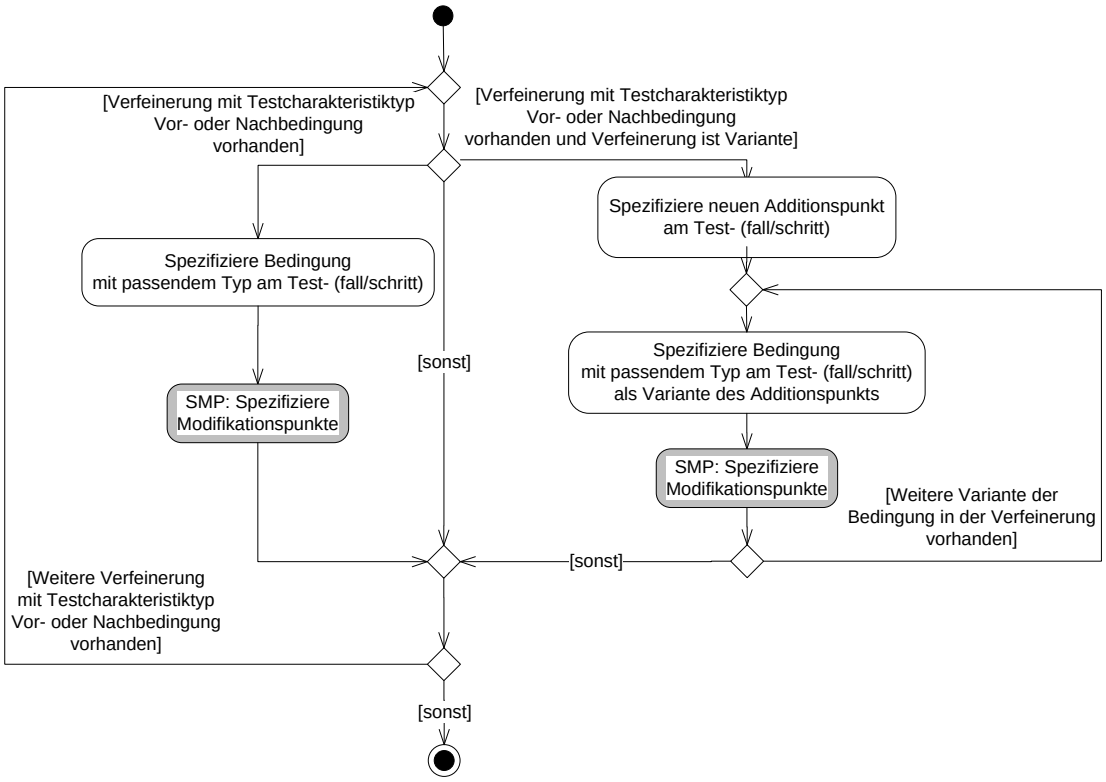


Abbildung 8.21: Algorithmus SVN: Spezifikation von Vor- und Nachbedingungen für einen logischen Testfall

Der Algorithmus selbst unterscheidet zunächst zwischen Vor- und Nachbedingungen und deren Verfeinerungen, die Varianten darstellen und solchen, die es nicht sind. Im ersten Fall wird ein neuer Additionspunkt am Testfall oder Testfallschritt erstellt und anschließend für jede Variante der Verfeinerung eine neue Bedingung als Variante an den Additionspunkt spezifiziert. Während der Spezifikation werden auch Modifikationspunkte in der Verfeinerung berücksichtigt.
Ist die Vor- oder Nachbedingung bzw. ihre Verfeinerung keine Variante, erfolgt die Spezifikation der Bedingung am Testfall oder Testschritt ohne Additionspunkt.

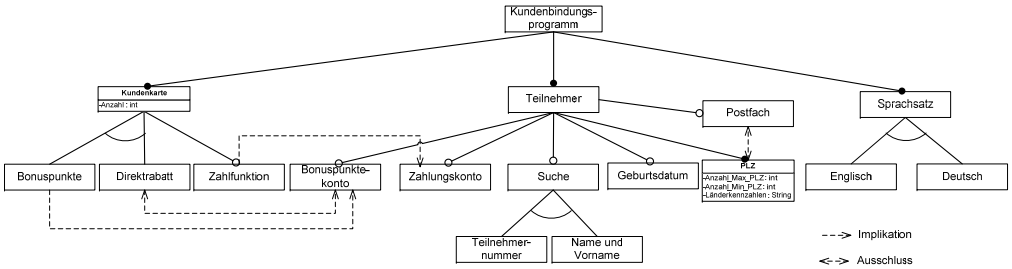
Die vorgestellten Algorithmen *LTS**, *SEA*, *SMP* und *SVN* sind in Anhang A.2 und A.3 auch als Algorithmen im Pseudocode definiert. Diese beschreiben neben dem Prinzip auch das Variabilitätsmanagement genauer.

Beispiel: Spezifikation von Vor- und Nachbedingungen

<vp$_{30}$><v$_1$>							
Testfallname	TC_01_1_Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.						
Vorbedingung	Die Maske „Suche und Pflege“ wird angezeigt.						
						Testdaten-satz 1	**...**
	Name	**Wertebereich**	**Format**	**Typ**	**Pflicht-bedingung**		
Eingabeparameter:	Anzahl	1.. 9999999999		Numerisch			
Eingabeparameter:	Geburtsdatum		**<vp$_{31}$><v$_1$|BK$_7$>** TT.MM.JJJJ **</v$_1$>** **<v$_2$|BK$_6$>** YYYY-MM-DD **</v$_2$></vp$_{31}$>**	Datum			
Eingabeparameter:	PLZ	**<vp$_{32}$|BK$_{11}$>**X**</vp$_{32}$>** … **<vp$_{39}$|BK$_{13}$>**X**</vp$_{39}$>**		Numerisch	**<vp$_{33}$><v$_1$|BK$_9$>** :¬Postfach **</v$_1$>** **<v$_2$|BK$_{10}$>** True **</v$_2$></vp$_{33}$>**		
<vp$_{10}$><v$_1$|BK$_6$>							
Eingabeparameter:	Postfach	**<vp$_{34}$><v$_1$|BK$_7$>** 10000000..99999999 **</v^1>** **<v$_2$|BK$_6$>** **<vp$_{35}$|BK$_{12}$>X</vp$_{35}$>** 10000000..99999999 **</v$_2$></vp$_{34}$>**	**<vp$_{36}$><v$_1$|BK$_6$>** LL-ZZZZZZZZ **</v$_1$></vp$_{36}$>**	**<vp$_{37}$><v$_1$|BK$_7$>** numerisch **</v$_1$>** **<v$_2$|BK$_6$>** Alpha-numerisch **</v$_2$></vp$_{37}$>**	¬PLZ		
</v$_1$><vp$_{10}$>							
Ausgabeparameter:							
Nr.	**Testschritte**						
-	-						
Nachbedingung	Eine Liste mit Teilnehmern wird angezeigt. **<vp$_{38}$><v$_1$|BK$_7$>** Das Geburtsdatum wird in der Form TT.MM.JJJJ dargestellt. **</v^1>** **<v$_2$|BK$_6$>** Das Geburtsdatum wird in der Form YYYY-MM-DD dargestellt. **</v$_2$></vp$_{34}$>**						
</v$_1$></vp$_{30}$>							

Abbildung 8.22: Beispiel für einen Testfall mit Nachbedingung und Variabilität

Als Beispiel wird der Testfall aus Abbildung 8.18 um eine variable Nachbedingung erweitert, wie in Abbildung 8.22 dargestellt ist. Für die Spezifikation der Nachbedingung wurden Verfeinerungen der Anwendungsfallbeschreibung und aus dem Technikmodell genutzt, welches in Abbildung 8.7 dargestellt ist. Die neu entstandenen Varianten wurden in das Abbildungsmodell übernommen. Es ist zusammen mit dem Featuremodell in Abbildung 8.23 dargestellt.



Abbildungsimplikation:

Bindungs-konfiguration	Featurebedingung	Bestandteile
BK_1	Suche	$vp_1(v_1)$, $vp_6(v_1)$, $vp_{12}(v_1)$, $vp_{14}(v_1)$, $vp_{16}(v_1)$
BK_2	Zahlfunktion ∧ Zahlungskonto	$vp_2(v_1)$, $vp_3(v_1)$, $vp_5(v_1)$, $vp_6(v_1)$, $vp_9(v_1)$
BK_3	Teilnehmernummer	$vp_4(v_1)$, $vp_{11}(v_1)$, $vp_{13}(v_1)$, $vp_{15}(v_1)$, $vp_{17}(v_1)$, $vp_{19}(v_1)$
BK_4	Name und Vorname	$vp_4(v_2)$, $vp_{11}(v_2)$, $vp_{13}(v_2)$, $vp_{15}(v_2)$, $vp_{17}(v_2)$, $vp_{19}(v_2)$
BK_5	Geburtsdatum	$vp_{24}(v_1)$
BK_6	Englisch	$vp_{25}(v_1)$, $vp_{31}(v_2)$, $vp_{34}(v_2)$, $vp_{36}(v_1)$, $vp_{37}(v_2)$, **$vp_{38}(v_2)$**
BK_7	Deutsch	$vp_{25}(v_2)$, $vp_{31}(v_1)$, $vp_{34}(v_1)$, $vp_{37}(v_1)$, **$vp_{38}(v_1)$**
BK_8	Suche ∧ Zahlfunktion ∧ Zahlungskonto	$vp_{18}(v_1)$
BK_9	Postfach	$vp_{33}(v_1)$
BK_{10}	¬Postfach	$vp_{33}(v_2)$

Abbildungsbeeinflussung:

Bindungs-konfiguration	Wertefunktion	Modifikationspunktmenge
BK_{10}	Kundenkarte.Anzahl	$vp_{7,}$ vp_{10}
BK_{11}	PLZ.Max_PLZ	vp_{32}
BK_{12}	PLZ.Länderkennzahlen= {DE, GB, F, ESP}	vp_{35}
BK_{13}	PLZ.Min_PLZ	vp_{39}

Abbildung 8.23: Beispiel für das Feature- und Abbildungsmodell zum Testfall aus Abbildung 8.22

8.5 Spezifikation von konkreten Testfällen

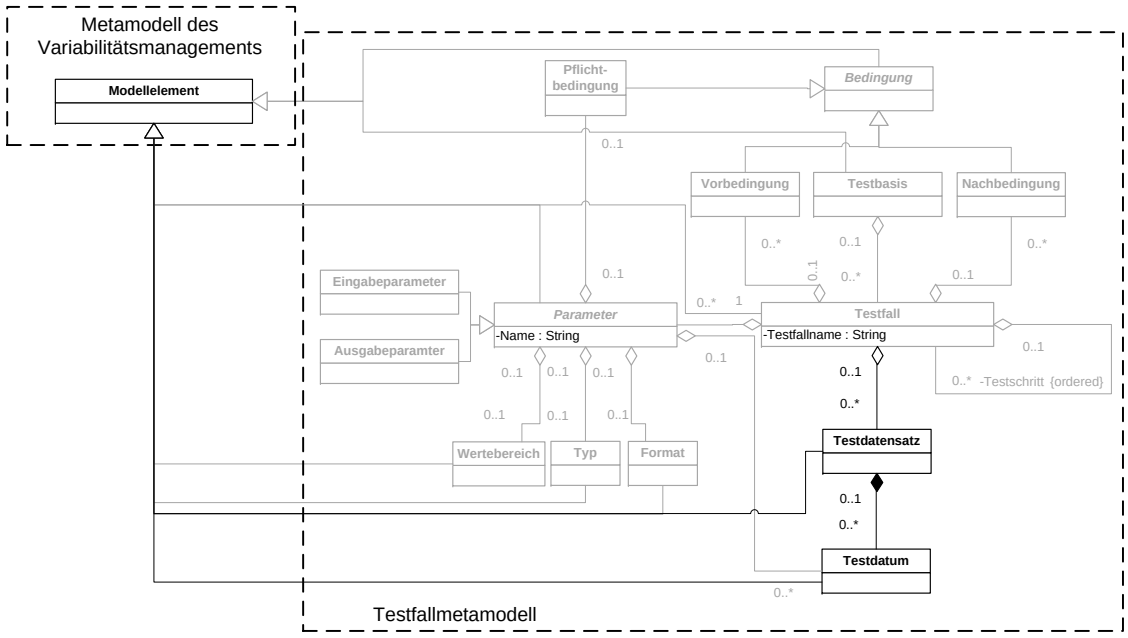


Abbildung 8.24: Metamodell des konkreten Testfalls

Ein konkreter Testfall erweitert einen logischen Testfall um einen *Testdatensatz* (vgl. Abschnitt 2.3). Der Testdatensatz besteht wiederum aus *Testdaten*, die auch zu Ein- und Ausgabeparametern assoziiert sind (vgl. Abbildung 8.24). Die Spezifikation eines Testdatums für einen Testdatensatz geschieht auf Basis der Parameter der Testschritte. Dabei werden die Testfallbestandteile Typ, Format, Wertebereich und Pflichtbedingung des Parameters berücksichtigt.
In der Literatur existieren Methoden, die auf Basis von Äquivalenzklassen und der auf Äquivalenzklassen basierenden Grenzwertanalyse, Testdatensätze und Testdaten erstellen [SL05]. In dieser Arbeit werden die Äquivalenzklassen auf Basis des Wertebereichs eines Parameters definiert, sofern dieser angegeben ist. Ist dies nicht der Fall, müssen die Äquivalenzklassen manuell bestimmt werden.
Die Bestandteile eines Parameters und der Parameter selbst können mit Variabilität behaftet sein (vgl. Abschnitt 5.1). Diese Tatsache muss bei der Spezifikation von Testdatensätzen und den Testdaten berücksichtigt werden:

Das Vorgehen für die Spezifikation von konkreten Testfällen ist in Abbildung 8.25 dargestellt. Zunächst werden auf Basis der gewählten Überdeckungsstrategie für

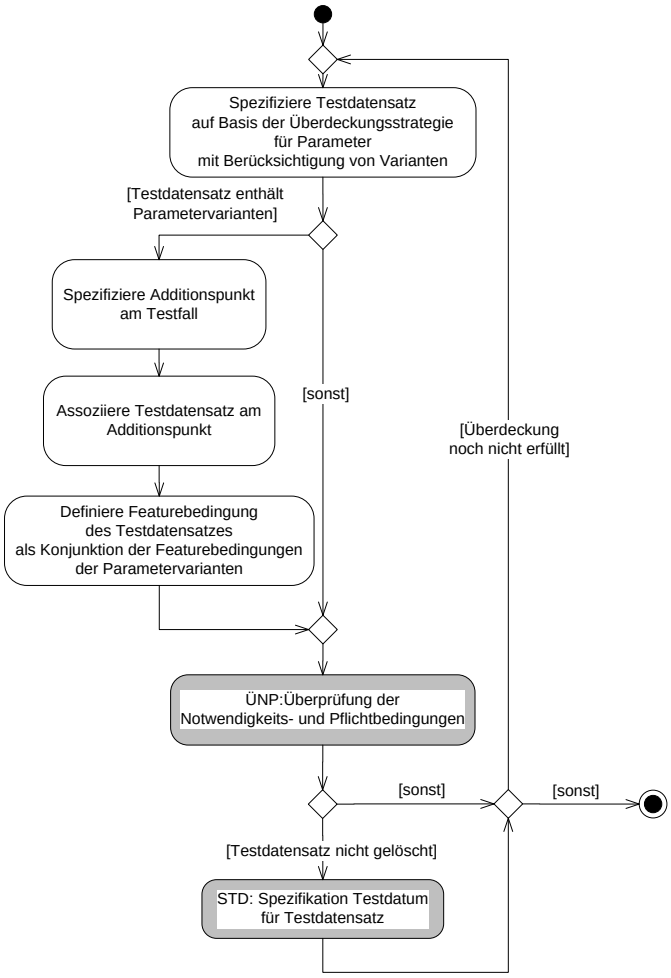


Abbildung 8.25: Algorithmus KTF: Spezifikation von konkreten Testfällen mit Variabilität

Parameter alle möglichen Testdatensätze erstellt. Die Überdeckungsstrategie kann unterschiedlich gewählt werden:

- Eingabeüberdeckung: Es wird eine Strategie für die Überdeckung der Eingabeparamter genutzt.
- Ausgabeüberdeckung: Es wird eine Strategie für die Überdeckung der Ausgabeparameter genutzt.

Innerhalb der Überdeckungsstrategie können unterschiedliche Abdeckungen der möglichen Kombinationen der Parameter gewählt werden. Dabei werden auch Parametervarianten berücksichtigt. Die größte Abdeckung kann mit einer vollständigen

Permutation aller Ein- bzw. Ausgabeparameter erreicht werden. Bei dieser Überdeckungsstrategie steigt die Anzahl der resultierenden Testdatensätze durch die kombinatorische Explosion sehr schnell an.
Nach der Spezifikation der Testdatensätze wird für jeden Testdatensatz überprüft, ob die gestellten *Notwendigkeits-* und *Pflichtbedingungen* erfüllt sind (*Algorithmus ÜNP*).
An einen notwendigen Parameter muss ein Testdatum einer bestimmten Äquivalenzklasse spezifiziert werden. Diese Notwendigkeitsbedingung ist abhängig vom Ablauf der Anwendungsfallbeschreibung, welcher durch den Testfall getestet wird und wird durch den Testdesigner definiert.
Beispielsweise testet ein Testschritt eine Ausnahme der Anwendungsfallbeschreibung und daher darf ein bestimmter Eingabeparameter kein Testdatum besitzen. Der Eingabeparameter stellt normalerweise eine Pflichteingabe dar und führt durch sein Weglassen die Ausnahme herbei.
Die Überprüfung der Pflichtbedingungen erfolgt im Anschluss an die Überprüfung der Notwendigkeitsbedingungen. Dabei wird ausgewertet, ob alle an den Parametern spezifizierten Pflichtbedingungen erfüllt sind.
Falls der Testdatensatz die existierenden Notwendigkeits- und Pflichtbedingungen nicht erfüllt, wird er korrigiert oder gelöscht. Nach der Korrektur der Notwendigkeits- und Pflichtbedingungen werden die Testdaten für den Testdatensatz erstellt.
Der *Algorithmus ÜNP* für die Überprüfung der Testdatensätze auf Verletzung der Notwendigkeits- und Pflichtbedingungen ist im folgenden Abschnitt näher beschrieben.

8.5.1 Überprüfung von Testdatensätzen

Die Überprüfung des Testdatensatzes gliedert sich in die Überprüfung der Notwendigkeits- und Pflichtbedingungen. Der Algorithmus dazu ist in Abbildung 8.26 dargestellt. Zunächst wird festgestellt, ob eine Notwendigkeitsbedingung für einen Parameter verletzt ist. Eine Verletzung liegt zum einen vor, wenn der Parameter nicht Teil des Testdatensatzes ist, obwohl er aufgrund des vom logischen Testfall überdeckten Ablaufs des Anwendungsfalls vorhanden sein muss. Zum anderen kann der Parameter auch notwendigerweise ausgeschlossen sein, ist aber in dieser Parameterkombination vorhanden.
Die Verletzung wird also durch Hinzufügen oder Löschen eines Parameters aus dem

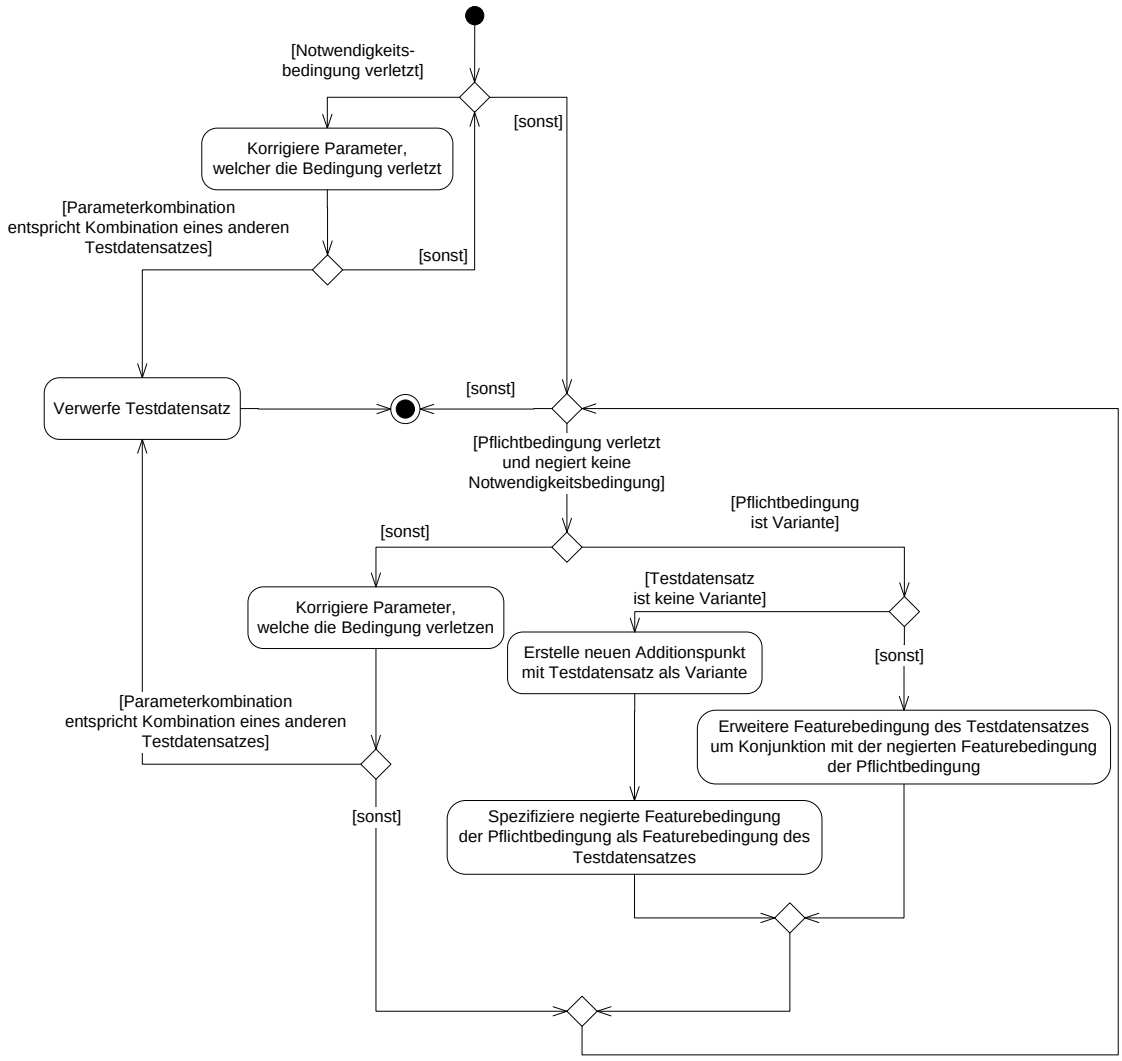


Abbildung 8.26: Algorithmus ÜNP: Überprüfung der Notwendigkeits- und Pflichtbedingungen eines Testdatensatzes

Testdatensatz korrigiert. Im Anschluss daran wird überprüft, ob die resultierende Parameterkombination der Kombination eines anderen Testdatensatzes entspricht. Ist dies der Fall, wird der Testdatensatz verworfen. Die Überprüfung auf eine Verletzung der Notwendigkeitsbedingungen wird so lange wiederholt, bis entweder keine Verletzung mehr vorliegt oder der Testdatensatz verworfen wurde.
Ist im Testdatensatz keine Notwendigkeitsbedingung verletzt, wird auf die Verletzung von Pflichtbedingungen überprüft. Eine Pflichtbedingung kann durch das Auslassen oder Wählen eines Parameters in der Kombination ausgelöst werden. Falls

eine verletzte Pflichtbedingung aber eine Negation einer Notwendigkeitsbedingung darstellt, wird diese ignoriert.
Dies ist zum Beispiel der Fall, wenn eine Pflichtbedingung vorschreibt, dass ein Eingabeparameter immer ausgefüllt werden muss, aber es in diesem Testfall notwendig ist, diesen nicht auszufüllen, um eine Ausnahme auszulösen.
Ist die Pflichtbedingung eine Variante, wird sie nur wirksam, falls sie bei der Ableitung gebunden wird. Mit anderen Worten, sofern die Bedingung nicht bei der Ableitung gebunden wird, kann der Testdatensatz Teil des Produktes sein. Somit wird der gesamte Testdatensatz als Variante an einem Additionspunkt spezifiziert. Er wird über eine Abbildungsimplikation auf das Featuremodell abgebildet. Die Featurebedingung besteht aus der Negation der Featurebedingung der verletzten Pflichtbedingung. Falls der Testdatensatz bereits eine Variante ist, wird seine Featurebedingung durch eine Konjunktion mit der negierten Featurebedingung der Pflichtbedingung verknüpft.
Ist die Pflichtbedingung keine Variante, werden die sie verletzenden Parameter durch Hinzufügen oder Löschen korrigiert. Fehlt ein für die Bedingung notwendiger Parameter, wird er hinzugefügt, ist ein Parameter vorhanden, der durch die Pflichtbedingung ausgeschlossen ist, wird dieser entfernt. Falls die durch die Korrektur entstehende Parameterkombination einer anderen existierenden Kombination entspricht, wird der Testdatensatz verworfen. Der Vorgang wird für jede verletzte Pflichtbedingung wiederholt. Sind alle Pflichtbedingungen erfüllt oder der Testdatensatz verworfen, ist die Korrektur des Testdatensatzes beendet.

8.5.2 Spezifikation eines Testdatums

Abbildung 8.27 stellt das Vorgehen für die Spezifikation eines Testdatums dar. Dafür muss zunächst die zur Notwendigkeitsbedingung passende Äquivalenzklasse auf Basis des Wertebereichs oder einer manuellen Entscheidung gewählt werden. Die Notwendigkeitsbedingung ist, wie bereits beschrieben, abhängig vom Ablauf der Anwendungsfallbeschreibung, welcher durch den Testfall abgedeckt wird.
Falls die Bestandteile Typ, Format oder Wertebereich Varianten sind, wird für jede existierende Featurebedingung aus diesen drei Bestandteilen eine Variante des Testdatums erstellt. Falls ein Bestandteil Modifikationspunkte enthält, wird für jede Variante des Testdatums ein Modifikationspunkt spezifiziert, die von der gleichen Abbildungsbeeinflussung abhängt, wie der Modifikationspunkt des Bestandteils.

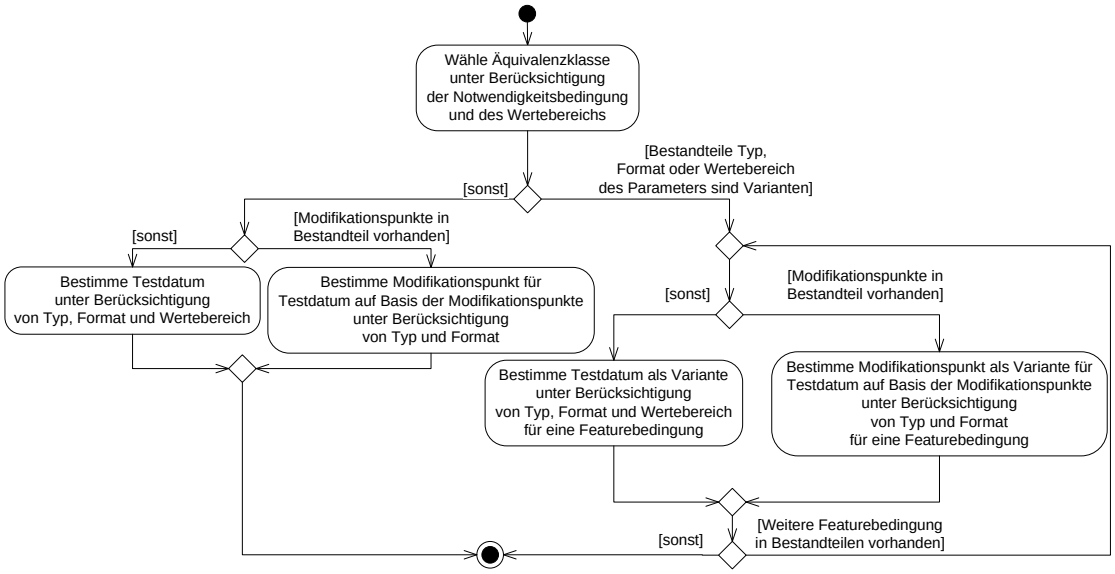


Abbildung 8.27: Spezifikation eines Testdatums für einen Testdatensatz

Sind keine Additionspunkte in den genannten Testfallbestandteilen vorhanden, wird das Testdatum direkt an den Testdatensatz und den Parameter spezifiziert. Auch hierbei werden vorkommende Modifikationspunkte bei der Spezifikation des Testdatums berücksichtigt.
Bei der Bestimmung des Testdatums selbst, müssen der *Systemzustand* vor der Testdurchführung sowie Abhängigkeiten zu anderen Eingabeparametern berücksichtigt werden. Für Ausgabeparameter muss zudem das erwartete Ergebnis bestimmt werden.

8.5.3 Beispiel für die Spezifikation von konkreten Testfällen

Als Beispiel für die Spezifikation eines konkreten Testfalls dienen die in Abbildung 8.28 dargestellten Eingabeparameter. Als Überdeckungsstrategie wird im Beispiel die Überdeckung aller möglichen Kombinationen der Parameter gewählt. Ausgabe des Schrittes sind die Testdatensätze, dargestellt in Tabelle 8.2:

Aufgrund des überdeckten Ablaufs im Anwendungsfall ist der Parameter PLZ notwendig. Er muss also Teil jedes Testdatensatzes sein. Im Rahmen der Korrektur der Testdaten werden alle Testdatensätze verworfen, die den Parameter PLZ nicht enthalten. Das Hinzufügen eines notwendigen Parameters zu einem zu korri-

Featurebedingung	$\neg Postfach \wedge Postfach$					
Anzahl	x	x	x	x		
Geburtsdatum	x	x	x		x	
PLZ	x	x				
Postfach	x					

Featurebedingung	$Postfach$	$Postfach$	$Postfach$	$Postfach$
Anzahl				x
Geburtsdatum			x	x
PLZ		x	x	
Postfach	x	x	x	x

Featurebedingung	$\neg Postfach \wedge Postfach$	$Postfach$		$Postfach$	
Anzahl	x	x			
Geburtsdatum				x	x
PLZ	x		x		x
Postfach	x	x		x	

Tabelle 8.2: Mögliche und durch Notwendigkeits- oder Pflichtbedingungen ausgeschlossene Kombinationen der Parameter

<vp30><v1>							
Testfallname	TC_01_1_Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.						
Vorbedingung	Die Maske „Suche und Pflege“ wird angezeigt.						
						Testdaten-satz 1	**...**
	Name	**Wertebereich**	**Format**	**Typ**	**Pflicht-bedingung**		
Eingabeparameter:	Anzahl	1.. 9999999999		Numerisch			
Eingabeparameter:	Geburtsdatum		**<vp31><v1|BK7>** TT.MM.JJJJ **</v1> <v2|BK6>** YYYY-MM-DD **</v2></vp31>**	Datum			
Eingabeparameter:	PLZ	**<vp32|BK11>**X**</vp32>** … **<vp39|BK13>**X**</vp39>**		Numerisch	**<vp33><v1|BK9>** :¬Postfach **</v1> <v2|BK10>** True **</v2></vp33>**		
<vp10><v1|BK6>							
Eingabeparameter:	Postfach	**<vp34><v1|BK7>** 10000000..99999999 **</v¹> <v2|BK6> <vp35|BK12>X</vp35>** 10000000..99999999 **</v2></vp34>**	**<vp36><v1|BK6>** LL-ZZZZZZZZ **</v1></vp36>**	**<vp37><v1|BK7>** numerisch **</v1> <v2|BK6>** Alpha-numerisch **</v2></vp37>**	¬PLZ		
</v1><vp10>							
Ausgabeparameter:							
Nr.	**Testschritte**						
-	-						
Nachbedingung	Eine Liste mit Teilnehmern wird angezeigt. **<vp38><v1|BK7>** Das Geburtsdatum wird in der Form TT.MM.JJJJ dargestellt. **</v¹> <v2|BK6>** Das Geburtsdatum wird in der Form YYYY-MM-DD dargestellt. **</v2></vp34>**						
</v1></vp30>							

Abbildung 8.28: Beispiel für einen Testfall mit Nachbedingung und Variabilität

gierenden Testdatensatz ist nicht notwendig, da die Parameterkombination jeweils in einem anderen Testdatensatz vorhanden ist. Dies liegt daran, dass alle Kombinationen der Parameter durch das gewählte Überdeckungskriterium erstellt wurden. Die entfernten Testdatensätze sind in Tabelle 8.2 grau hinterlegt.

Wie in der Abbildung beim Testdatensatz oben links und unten links zu erkennen ist, wurden hier durch den Algorithmus Featurebedingungen generiert, die eine Kontradiktion darstellen. In diesem Fall wurde der Parameter aufgrund der Nichterfüllung einer Pflichtbedingung verworfen. Falls dies nicht der Fall wäre, wäre diese Featurebedingung durch die in Kapitel 6 beschriebene Maßnahme zur Identifikation von Kontradiktionen gefunden worden.

Parameter	**Testdatensatz 1**	**Testdatensatz 2**
Anzahl	1	9999999999
Geburtsdatum	$<vp_{40}><v_1\|BK_7>$ 01.02.1970 $</v_1><v_2\|BK_6>$ 1970-02-01 $</v_1></vp_{40}>$	$<vp_{41}><v_1\|BK_7>$ 01.02.1970 $</v_1><v_2\|BK_6>$ 1970-02-01 $</v_1></vp_{41}>$
PLZ	$<vp_{42}\|BK_{11}>$X$</vp_{42}>$	$<vp_{43}\|BK_{13}>$X$</vp_{43}>$
Postfach		

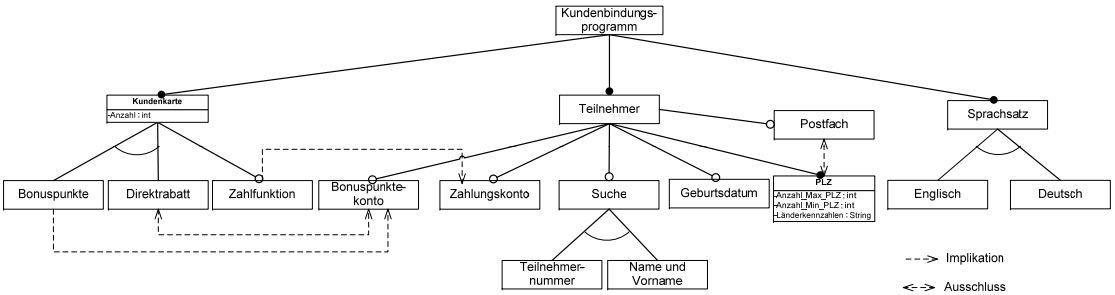
Tabelle 8.3: Konkrete Testdaten für die beiden Testdatensätze

Für die beiden verbleibenden Testdatensätze werden nun Testdaten spezifiziert (vgl. Tabelle 8.3). Dabei wird für den ersten Testdatensatz durch eine Grenzwertanalyse der beiden Parameter Anzahl und PLZ deren untere Schranke der Äquivalenzklasse gewählt. Im zweiten Testdatensatz ist es dann die obere Schranke. Für den Parameter Geburtsdatum existieren für das Format zwei Varianten. Daher werden auch für das Testdatum zwei Varianten auf dieser Basis spezifiziert. Das Testdatum PLZ erhält einen neuen Modifikationspunkt für jeden Testdatensatz. Im ersten Testdatensatz ist dies die untere Schranke des Wertebereichs und im zweiten die obere.

Abbildung 8.29 hebt die durch die Spezifikation der Testdaten notwendigen Ergänzungen im Abbildungsmodell hervor.

8.6 Zusammenfassung

Die Evaluation existierender Ansätze für die Spezifikation von wiederverwendbaren Testfällen in Kapitel 3.2.2 hat gezeigt, dass bisher nicht alle Bestandteile von Testfällen wiederverwendbar, d. h. mit Variabilität behaftet, spezifiziert werden. Insbesondere die Spezifikation von Testdaten, welche einen großen Teil des Aufwands bei der Testfallspezifikation ausmacht, wird nicht berücksichtigt. Durch die Spezifikation von Variabilität in Testfällen, kann der Spezifikationsaufwand reduziert werden, da gleiche Bestandteile von Testfällen nur einmal spezifiziert werden und Unterschiede explizit darstellbar sind.



Abbildungsimplikation:

Bindungs-konfiguration	Featurebedingung	Bestandteile
BK_1	Suche	$vp_1(v_1)$, $vp_6(v_1)$, $vp_{12}(v_1)$, $vp_{14}(v_1)$, $vp_{16}(v_1)$, $vp_{30}(v_1)$
BK_2	Zahlfunktion ∧ Zahlungskonto	$vp_2(v_1)$, $vp_3(v_1)$, $vp_5(v_1)$, $vp_6(v_1)$, $vp_9(v_1)$
BK_3	Teilnehmernummer	$vp_4(v_1)$, $vp_{11}(v_1)$, $vp_{13}(v_1)$, $vp_{15}(v_1)$, $vp_{17}(v_1)$, $vp_{19}(v_1)$
BK_4	Name und Vorname	$vp_4(v_2)$, $vp_{11}(v_2)$, $vp_{13}(v_2)$, $vp_{15}(v_2)$, $vp_{17}(v_2)$, $vp_{19}(v_2)$
BK_5	Geburtsdatum	$vp_{24}(v_1)$
BK_6	Englisch	$vp_{25}(v_1)$, $vp_{31}(v_2)$, $vp_{34}(v_2)$, $vp_{36}(v_1)$, $vp_{37}(v_2)$, $vp_{38}(v_2)$, **$vp_{40}(v_2)$, $vp_{41}(v_2)$**
BK_7	Deutsch	$vp_{25}(v_2)$, $vp_{31}(v_1)$, $vp_{34}(v_1)$, $vp_{37}(v_1)$, $vp_{38}(v_1)$, **$vp_{40}(v_1)$, $vp_{41}(v_1)$**
BK_8	Suche ∧ Zahlfunktion ∧ Zahlungskonto	$vp_{18}(v_1)$
BK_9	Postfach	$vp_{33}(v_1)$
BK_{10}	¬Postfach	$vp_{33}(v_2)$

Abbildungsbeeinflussung:

Bindungs-konfiguration	Wertefunktion	Modifikationspunktmenge
BK_{10}	Kundenkarte.Anzahl	vp_7, vp_{10}
BK_{11}	PLZ.Max_PLZ	Vp_{32}, **vp_{42}**
BK_{12}	PLZ.Länderkennzahlen= {DE, GB, F, ESP}	Vp_{35}
BK_{13}	PLZ.Min_PLZ	Vp_{39}, **vp_{43}**

Abbildung 8.29: Beispiel für das Feature- und Abbildungsmodell zum Testfall aus Abbildung 8.28

Um diesem Problem zu begegnen, wurde in diesem Kapitel ein Testfallspezifikationsprozess vorgestellt, der alle Bestandteile eines Testfalls und dabei insbesondere Testdaten wiederverwendbar gestaltet

Zunächst erfolgte dazu in Abschnitt 8.1 die Zusammenfassung der Anforderungen an einen Testfallspezifikationsprozess mit Variabilität. Anschließend wurde in Abschnitt 8.2 ein Überblick über den Testfallspezifikationsprozess gegeben, basierend auf dem Prozessmetamodell aus Kapitel 7.
Der erste Schritt des Testfallspezifikationsprozesses ist die Analyse der gegebenen Testbasis. In dieser Arbeit wurde als zentrales Element der Testbasis die Anwendungsfallbeschreibung gewählt. Um konkrete Testfälle auf Basis dieser Beschreibungen spezifizieren zu können, sind Informationen aus weiteren Modellen der Plattform notwendig. Daher wurde in Abschnitt 8.3 zunächst ein algorithmisches Vorgehen für die *Verfeinerung* von Teilen der Anwendungsfallbeschreibungen mit Teilen aus anderen Modellen der Plattform definiert. Diese Verfeinerungsbeziehungen können dann für die Spezifikation von Testfällen eingesetzt werden. Bei der Analyse der Testbasis können bereits Abweichungen zwischen unterschiedlichen Modellen der Plattform erkannt werden.
Die um Verfeinerungen ergänzte Testbasis ermöglichte im Anschluss die schrittweise Spezifikation von logischen Testfällen unter Berücksichtigung von Variabilität. Der Spezifikationsprozess wurde in Form von Algorithmen beschrieben (vgl. Abschnitt 8.4). Im Anschluss daran erfolgte die Spezifikation von konkreten Testfällen, mit ihren mit variabilitätsbehafteten Testdaten (vgl. Abschnitt 8.5).
Das Resultat der Spezifikation sind mit Variabilität behaftete Testfälle, die mit dem featurebasierten Variabilitätsmanagement verbunden worden sind. Mit Hilfe des Variabilitätsmanagements ist dann die Ableitung von zu den Anforderungen passenden Testfällen für Produkte möglich.
Durch die Nutzung der Anwendungsfallbeschreibungen als Testbasis und dem featurebasierten Variabilitätsmanagement, gliedert sich der vorgestellte Testfallspezifikationsprozess in den Ansatz dieser Arbeit ein. Er nutzt dabei die Testfallmodellierungssprache mit Variabilität, die in Kapitel 5 definiert worden ist.

Kapitel 9

Werkzeugunterstützung und Evaluation

Die in dieser Arbeit vorgestellten Beiträge zur Modellierung von Variabilität und deren Management in Modellen der Produktlinienplattform sowie der Spezifikation von Testfällen mit Variabilität, werden in diesem Kapitel auf Umsetzbarkeit in der Praxis evaluiert.
Einerseits wird eine Fallstudie zur Spezifikation von Testfällen mit Variabilität, die im Rahmen eines Kooperationsprojektes[1] zwischen dem Industriepartner arvato services[2] und dem Software Quality Lab (s-lab) der Universität Paderborn durchgeführt worden ist.
Andererseits wird für das featurebasierte Variabilitätsmanagement eine prototypische Werkzeugunterstützung vorgestellt.

9.1 Fallstudie: Testfallspezifikation mit Variabilität bei arvato services

Der in dieser Arbeit vorgestellte Ansatz wird in diesem Abschnitt durch eine Fallstudie mit dem Industriepartner arvato services evaluiert. Um ein besseres Verständnis für die dabei existierende Problemstellung zu gewinnen, wird zunächst der Kontext der Fallstudie beschrieben. Anschließend wird der Inhalt der Studie zusammengefasst und die Ergebnisse der Studie werden diskutiert.

[1] http://s-lab.upb.de/Projekte/TestkonzeptDMD3000/index.html
[2] http://www.arvato.com

9.1.1 Kontext

Der Industriepartner arvato services ist ein IT-Dienstleister, der unterschiedliche Arten von Systemen, die heterogener Natur sind, zu größeren Systemverbünden verknüpft um damit Informationen zu sammeln, zu verwalten und zu verdichten. Ein wichtiger thematischer Bereich sind dabei Kundenbindungsprogramme, die auch Gegenstand dieser Fallstudie sind.
Kundenbindungsprogramme sind Programme zur Steigerung der Kundentreue. Häufig sind solche Programme mit einer *Kundenkarte* (Klub-Karte, Bonuskarte) verknüpft, die jeder Teilnehmer des Programms erhält. Unter Vorlage der Kundenkarten beim Bezahlen in teilnehmenden Geschäften können die Teilnehmer *Rabatte*, *Sammelpunkte* für *Sachprämien* oder sonstige *Vergünstigungen* erhalten. Somit wird ein Anreiz für die Teilnehmer geschaffen, die Kundenkarte bei jedem Kauf vorzulegen. Die Anbieter einer Kundenkarte erhalten von jedem neuen Teilnehmer personenbezogenen Daten, wie zum Beispiel den Namen, die Adresse, das Alter oder den Beruf. Beim Vorlegen der Kundenkarte an einer Kasse werden Informationen über den Kaufvorgang des Teilnehmers an den Anbieter der Karte übertragen. Diese Daten können zur Analyse des Kaufverhaltens, zur gezielten Werbung oder anderen Marketingmaßnahmen im Rahmen eines *Customer Relationship Managements* genutzt werden.
Bei arvato services werden immer wieder ähnliche Kundenbindungsprogramme für unterschiedliche Kunden erstellt. Daher wird hier die Nutzung des SPL-Paradigmas angestrebt, um durch Wiederverwendung Zeit und Kosten zu sparen. Der in dieser Arbeit entwickelte Ansatz des featurebasierten Variabilitätsmanagements und der Spezifikation von Testfällen schließt dabei existierende Lücken im SPL-Ansatz von arvato services. Die Fallstudie ist im Rahmen einer Diplomarbeit entstanden [Obe09a].

9.1.2 Zusammenfassung

Um einen Überblick über die Software-Produktlinien „Loyaltymanagement“, die die Grundlage für die Ableitung von individuellen Kundenbindungsprogrammen darstellt, zu erhalten, eignet sich dessen Featuremodell [Bro09]. Dieses besitzt 199 Features und 15 Featureattribute. Von diesen Features haben 56 eine Pflichtbeziehung,

98 eine Alternativbeziehung, 42 eine Optionalbeziehung und drei eine Oderbeziehung zu ihrem jeweiligen Vaterfeature. Es existieren weiterhin zwölf Implikationen und vier Ausschlüsse zwischen Features sowie zwei Beeinflussungen zwischen Featureattributen. Aufgrund der Größe des Featuremodells muss zum einen auf seine Darstellung im Rahmen dieser Arbeit verzichtet werden und zum anderen für die Fallstudie eine Reduktion des Umfangs durchgeführt werden.
Das resultierende Featuremodell stellt einen Ausschnitt aus dem Featuremodell „Loyaltymanagement“ dar. Kriterien für die Auswahl waren

- die Berücksichtigung aller Sprachelemente des Featuremodells und
- die Gewährleistung eines möglichst großen fachlichen Zusammenhangs.

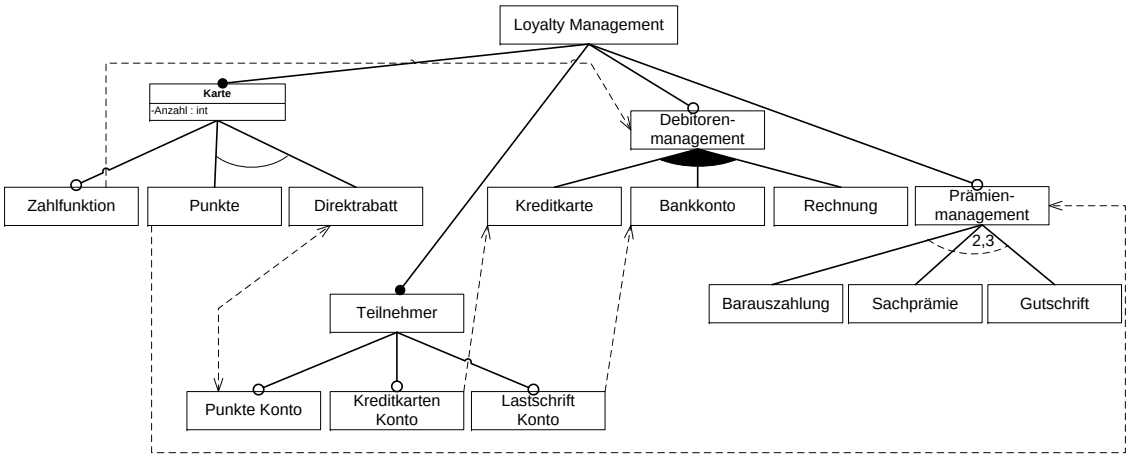


Abbildung 9.1: Reduziertes Featuremodell Loyaltymanagement

Das reduzierte Featuremodell ist in Abbildung 9.1 dargestellt.
Die Anforderungen an Produkte der SPL werden bei arvato services mit Hilfe von Anwendungsfallbeschreibungen spezifiziert. Die Erweiterung dieser Anwendungsfallbeschreibungen um Variabilität erfolgte in dieser Arbeit in Kapitel 5. Die dabei entstandene Variabilitätsmodellierungssprache für Anwendungsfallbeschreibungen wurde in dieser Fallstudie genutzt, die für die SPL Loyaltymanagement notwendigen Anforderungen zu beschreiben. Zu Beginn der Fallstudie besaßen die Anwendungsfallbeschreibungen noch keine explizite Variabilität. Es sind vielmehr die Anwendungsfallbeschreibungen von zwei existierenden Produkten untersucht worden, deren

Gemeinsamkeiten und Unterschiede anschließend mit Hilfe der Variabilitätsmodellierungssprache spezifiziert wurden.
Auch dieser Schritt basierte nur auf einer Auswahl an Anwendungsfallbeschreibungen. Abbildung 9.2 stellt diese Auswahl anhand eines Anwendungsfalldiagramms mit Anwendungsfällen und Akteuren dar.
Wichtig bei der Auswahl der Anwendungsfälle war, dass sie fachlich zu den ausgewählten Features (vgl. Abbildung 9.1) aus dem Featuremodell passten.

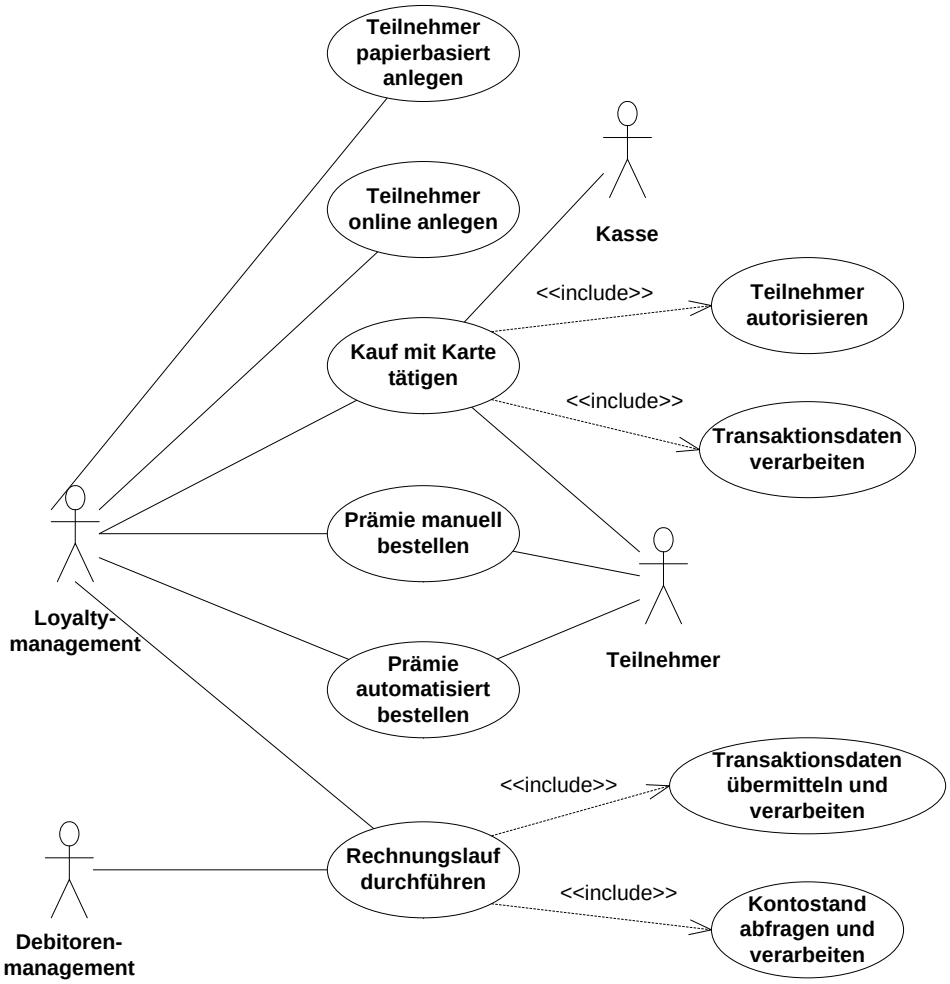


Abbildung 9.2: Reduziertes UML-Anwendungsfalldiagramm Loyaltymanagement

Auf Basis der Anwendungsfallbeschreibungen und weiterer Details des Technikmodells aus den beiden existierenden Produkten, erfolgte die Spezifikation von Testfällen mit Hilfe des in Kapitel 8 vorgestellten Testfallspezifikationsprozesses. Dabei

wurde beispielhaft eine Zweigüberdeckung der Anwendungsfallbeschreibungen genutzt und anschließend für jeden daraus resultierenden logischen Testfall ein Testdatensatz spezifiziert.

9.1.3 Ergebnisse

Die Spezifikation von Anwendungsfallbeschreibungen sowie Testfällen mit Variabilität mithilfe des in dieser Arbeit beschriebenen Ansatzes, offerierte die Möglichkeit einer vollständigen Spezifikation von Testfällen für die Wiederverwendung in unterschiedlichen Produkten. Im Rahmen der Fallstudie sind Vor- und Nachteile aufgetreten. Zunächst wird auf die Vorteile des Ansatzes eingegangen:

Vorteile:

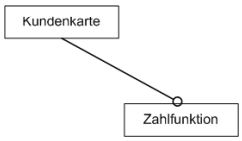
- Die Variabilitätsmodellierungssprachen für Anwendungsfallbeschreibungen und Testfälle genügten den Anforderungen von Plattformanalyst und Plattformtestdesigner hinsichtlich der Modellierung von Variabilität.

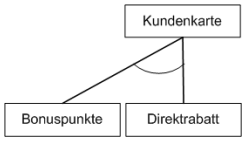
- Abbildung 9.3 zeigt alle Abbildungsimplikationen des reduzierten Beispiels aus der Fallstudie. Dabei wird deutlich, dass viele Variationspunkte in Modellen von den selben Featurebedindungen abhängen. Das Abbildungsmodell bleibt dadurch sehr übersichtlich und kompakt. Aus dieser Tatsache lässt sich schließen, dass bei steigender Anzahl an Variationspunkten in Modellen die Anzahl der Abbildungen im Abbildungsmodell eher langsam ansteigen wird. Dadurch ist zu vermuten, dass das Abbildungsmodell auch im Fall der Skalierung des Ansatzes auf Produktlinien mit vielen Features und Modellen übersichtlich bleibt.

- Die Wahl einer Variante in Bezug zu ihrem Variationspunkt hängt von ihrer Featurebedingung ab. Durch diese Tatsache ist der in dieser Arbeit gewählte Ansatz im Vergleich zu anderen existierenden Ansätzen ausdrucksmächtiger. Durch den Verzicht auf bestimmte Variationspunkttypen, wie zum Beispiel Alternative, ist es möglich, jeden Variabilitätstyp (siehe auch Kapitel2.3) und Mischungen aus verschiedenen Variabilitätstypen innerhalb eines Variationspunktes zu modellieren. Abbildung 9.4 stellt dies beispielhaft dar:
  In Zeile 1 der Tabelle ist ein Variationspunkt mit einer Variante modelliert.

Featurebedingung	Bindungs-konfiguration	zugeordnete Variationspunkte
Zahlungskonto	**BK1**	VP1, VP2, VP3, VP5, VP6, VP7, VP8, VP9, VP11, VP20, VP21, VP22, VP24, VP25, VP26, VP29, VP30, VP31, VP32, VP33, VP35, VP98
Bonuspunktekonto	**BK2**	VP2, VP4, VP10, VP11, VP23, VP24, VP26, VP28, VP35
Lastschriftkonto	**BK3**	VP1, VP7, VP8, VP9, VP12, VP13, VP14, VP15, VP16, VP17, VP18, VP19, VP20, VP21, VP22, VP25, VP31, VP32, VP33, VP36, VP37, VP38, VP39, VP40, VP41, VP42, VP43
Kreditkartenkonto	**BK4**	VP1, VP7, VP8, VP9, VP12, VP13, VP14, VP15, VP16, VP17, VP18, VP19, VP20, VP21, VP22, VP25, VP31, VP32, VP33, VP36, VP37, VP38, VP39, VP40, VP41, VP42, VP43
Rechnungskonto	**BK5**	VP1, VP7, VP8, VP9, VP12, VP13, VP14, VP15, VP16, VP17, VP18, VP20, VP21, VP22, VP25, VP31, VP32, VP33, VP36, VP37, VP38, VP39, VP40, VP41, VP42, VP43, VP48, VP51
Zahlfunktion ∧ Bonuspunkte	**BK6**	VP46
Zahlfunktion ∧ Direktrabatt	**BK7**	VP44, VP48
¬ Zahlfunktion ∧ Bonuspunkte	**BK8**	VP44, VP48, VP51
¬ Zahlfunktion ∧ Direktrabatt	**BK9**	VP44, VP48
Zahlfunktion	**BK10**	VP45, VP46, VP47, VP49, VP50, VP97
Prämienmanagement	**BK11**	VP52, VP53, VP54, VP55, VP56, VP57, VP58, VP59, VP60, VP61, VP62, VP63, VP64, VP65, VP66, VP67, VP68, VP69, VP70, VP71, VP72, VP73, VP74, VP75, VP76, VP77, VP78, VP80, VP81, VP82
Barauszahlung ∧ ¬ Gutschrift	**BK12**	VP79, VP83, VP84
¬ Barauszahlung ∧ Gutschrift	**BK13**	VP79
Barauszahlung ∧ Gutschrift	**BK14**	VP79
Debitorenmanagement	**BK15**	VP85, VP86, VP87, VP88, VP90, VP91, VP92, VP93, VP94, VP95, VP96
Kreditkarte	**BK16**	VP89
Bankkonto	**BK17**	VP89
Rechnung	**BK18**	VP89
Direktrabatt	**BK19**	VP45, VP46, VP47, VP49, VP50, VP83, VP84
Bonuspunkte	**BK20**	VP45, VP46, VP47, VP49, VP50

Abbildung 9.3: Abbildungsimplikationen der SPL *Loyaltymanagement*[Obe09b]

Features	Featurebedingungen	Variationspunkt
	BK_1: Zahlfunktion	<VP_1> <V_1|BK_1> ... </V_1> </VP_1>

	BK_2: Bonuspunkte
	BK_3: Direktrabatt
		<VP_2>
		  <V_1|BK_2>
		  ...
		  </V_1>
		  <V_2|BK_3>
		  ...
		  </V_2>
		</VP_2>

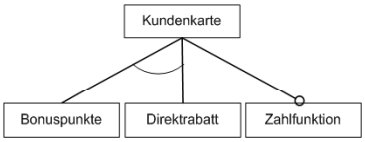
	BK_1: Zahlfunktion
	BK_2: Bonuspunkte
	BK_3: Direktrabatt
		<VP_3>
		  <V_1|BK_1>
		  ...
		  </V_1>
		  <V_2|BK_2>
		  ...
		  </V_2>
		  <V_3|BK_3>
		  ...
		  </V_3>
		</VP_3>

Abbildung 9.4: Beispiel für Ausdrucksmächtigkeit der Modellierungssprache

Diese Variante hat als Featurebedingung das optionale Feature *Zahlfunktion*, was auch die Variante optional macht. In der zweiten Zeile ist ein Variationspunkt mit zwei Varianten modelliert, deren Featurebedingungen *Bonuspunkte* bzw. *Direktrabatt* sind. Die beiden Features stellen im Featuremodell Alternativen dar, was auch die beiden Varianten zu Alternativen macht. In der dritten Zeile der Tabelle ist die Kombination der beiden bisherigen Fälle modelliert. Die erste Variante des Variationspunktes ist wiederum optional und die beiden anderen Varianten verhalten sich alternativ zueinander. Hierbei handelt es sich also um eine Mischform von Variabilitätstypen innerhalb eines Variationspunktes.

Dadurch, dass dieser Ansatz keine Einschränkung bei den Typen von Variationspunkten macht, entsteht eine größere Ausdrucksmächtigkeit im Vergleich zu Ansätzen, die eine solche Einschränkung tätigen.

- Die in Kapitel 6 eingeführten Qualitätssicherungsmaßnahmen unterstützen die Rollen Plattformanalyst und Plattformtestdesigner in geeigneter Weise und halfen bereits während der Spezifikation der Modelle Fehler in der Modellierung von Variabilität zu vermeiden.

Nachteile:

- Um eine konsistente Modellierung von Variabilität über alle Modelle der Produktlinienplattform hinweg gewährleisten zu können, muss das featurebasierte Variabilitätsmanagement in alle Werkzeuge des Entwicklungsprozesses eingebunden sein. Die Integration stellt Herausforderungen hinsichtlich der Definition von Schnittstellen und der Technologie der jeweiligen Werkzeuge, was durch den Ansatz bisher nicht adressiert wird.

- Um Fehler bei der Modellierung von Modellen mit Variabilität zu vermeiden, sollte der Ansatz vollständig werkzeugunterstützt werden. Dies ist zum aktuellen Zeitpunkt noch nicht der Fall.

- Bisher wird durch den Ansatz nur der Bindungszeitpunkt *Ableitung* explizit berücksichtigt. Andere Bindungszeitpunkte wie *Installations-* und *Laufzeit* bleiben bisher unberücksichtigt. Im Rahmen der Fallstudie wurde festgestellt, dass der Bindungszeitpunkt bei der Entwicklung einer SPL wichtig sein kann.

9.2 Prototypische Werkzeugunterstützung

Die prototypische Werkzeugunterstützung des featurebasierten Variabilitätsmanagement evaluiert die Umsetzbarkeit des in dieser Arbeit beschriebenen Ansatzes. Das Werkzeug setzt dabei das in Kapitel 5 beschriebene featurebasierte Variabilitätsmanagement beispielhaft für in Java implementierte Testskripte um. Zunächst wird die Architektur des Werkzeugs beschrieben und anschließend der Algorithmus zur Ableitung von Produkten definiert. Zuletzt wird ein Beispiel für die Ableitung eines Produktes gezeigt.

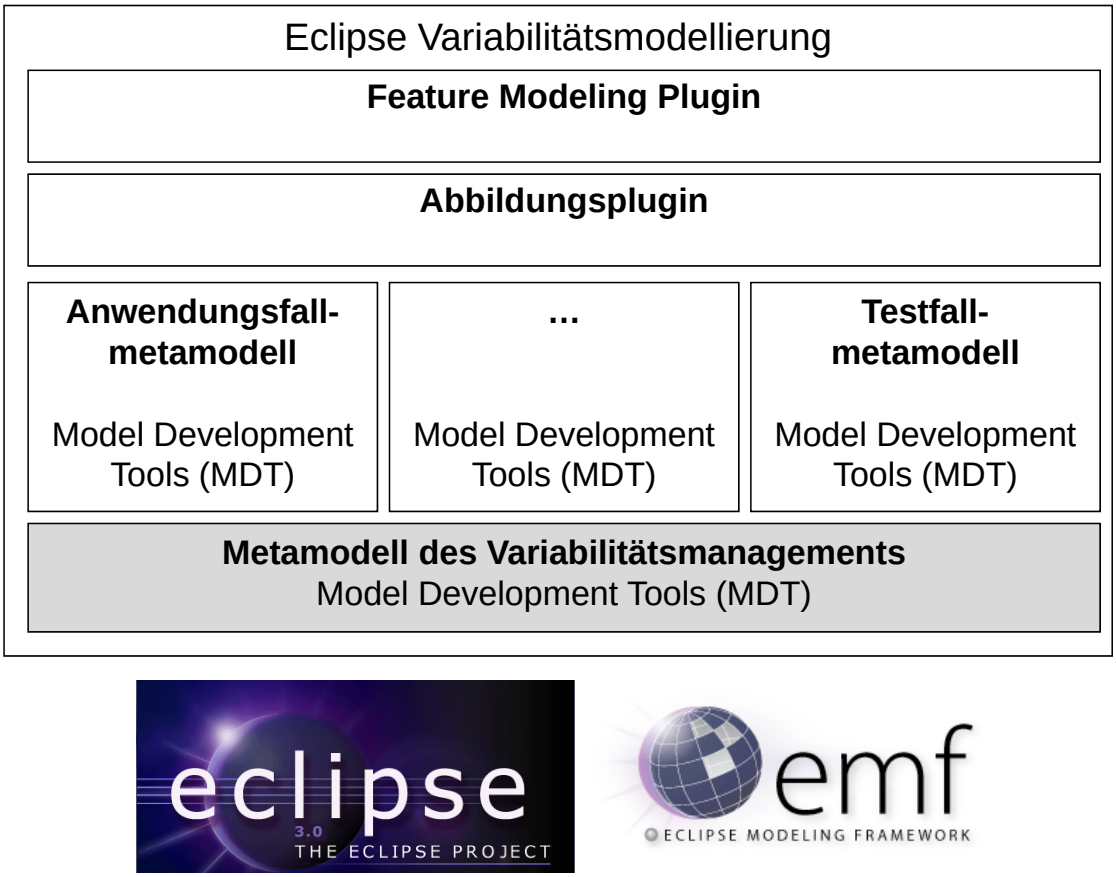


Abbildung 9.5: Überblick über die Variabilitätsmodellierung mit dem prototypischen Eclipse-Werkzeug

9.2.1 Werkzeugarchitektur

Das Werkzeug, dargestellt in Abbildung 9.5, wurde auf Basis der *Eclipse*[3]-Entwicklungsumgebung und der Erweiterung *Model Development Tool*[4] (MDT) realisiert. Die MDT implementiert das Metamodell der UML und der OCL. Mit seiner Hilfe war es möglich das in Kapitel 5 beschriebene Metamodell des Variabilitätsmanagements zu spezifizieren.
Das Metamodell des Variabilitätsmanagements konnte anschließend mit Hilfe des in Kapitel 5 beschriebenen Vorgehens in die Metamodelle von Modellierungssprachen integriert werden. Dabei wurde die in Abschnitt 5.1 definierte konkrete Syntax für

[3]http://www.eclipse.org/
[4]http://www.eclipse.org/modeling/mdt/?project=uml2

die Modellierung von Variabilität genutzt.
Für die Spezifikaton von Featuremodellen wurde das *Feature Modeling Plugin* der Universität Waterloo genutzt [Ecl]. Das Abbildungsplugin ermöglicht die Spezifikation von Abbildungsimplikationen und Abbildungsbeeinflussungen zwischen Features bzw. deren Attributen und Varianten bzw. Modifikationspunkten in Modellen.

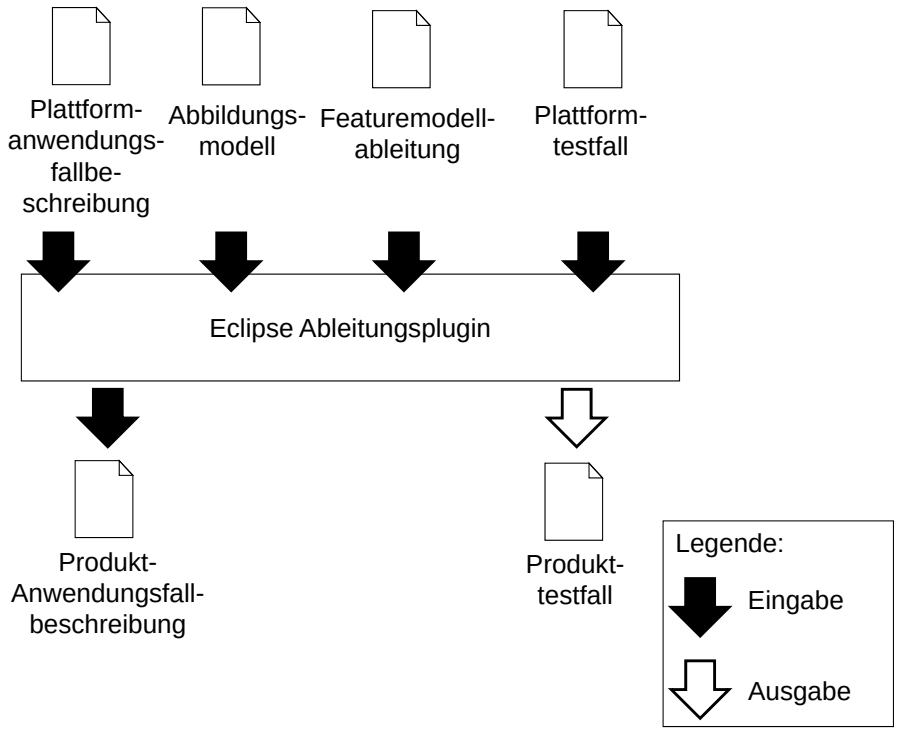


Abbildung 9.6: Architektur Ableitungsplugin für Eclipse am Beispiel für Anwendungsfallbeschreibung und Testfall

Die mithilfe der *Eclipse Variabilitätsmodellierung* erstellten Plattformmodelle mit Variabilität können mit Hilfe des Eclipse Ableitungsplugins zu Produktmodellen abgeleitet werden [Esc09]. Dazu wird die Ableitung eines Featuremodells benötigt, die mithilfe des *Feature Modeling Plugins* abgeleitet werden kann. Das Ableitungsplugin nutzt diese *Featuremodellableitung* und das Abbildungsmodell, um die Variabilität in den Plattformmodellen zu entfernen.
Der genaue Ableitungsvorgang wird im Folgenden genauer beschrieben.

Ableitungsalgorithmus

Der Algorithmus zur Ableitung von Produktmodellen aus Plattformmodellen mit Variabilität arbeitet auf der Featuremodellableitung und dem Abbildungsmodell, welches die Features auf Modellelemente der Plattformmodelle abbildet.
Zunächst werden die Additionspunkte und ihre Varianten in den Plattformmodellen betrachtet: Es wird für jede Featurebedingung im Abbildungsmodell überprüft, ob diese durch die gegebene Featuremodellableitung erfüllt wird. Ist dies der Fall, werden alle von dieser Featurebedingung implizierten Modellelemente in Plattformmodellen gebunden.

Algorithmus 1 Ableitungsalgorithmus im Pseudocode

```
1: Lese Featuremodellableitung
2: Lese Abbildungsmodell
3: for all Featurebedingungen in Abbildungsmodell do
4:    if Featurebdingung ist erfüllt then
5:       for all Modellelemente impliziert von Featurebedingung do
6:          Binde Variante
7:       end for
8:    end if
9: end for
10: for all Abbildungsbeeinflussung do
11:    Berechne Wert mit Werteabbildung und Featureattribut als Eingabe
12:    for all Beeinflusster Modifikationspunkt do
13:       Binde Wert an Modifikationspunkt
14:    end for
15: end for
```

Für die Bindung von Modifikationspunkten werden alle Abbildungsbeeinflussungen betrachtet. Mit Hilfe der Werteabbildung in jeder Abbildungsbeeinflussung wird der Wert für die beeinflussten Modifikationspunkte berechnet. Als Eingabe erhält die Werteabbildung die Werte der Featureattribute aus der Featuremodellableitung. Im Anschluss daran wird dieser Wert an jeden beeinflussten Modifikationspunkt gebunden.

Im Folgenden wird ein Beispiel für die Arbeitsweise der prototypischen Ableitung von Plattformmodellen zu Produktmodellen gegeben.

Beispiel für die Ableitung von Produkten

Das Beispiel für die Ableitung von Plattformmodellen wird auf Basis von in Java implementierten Testfällen mit Variabilität vorgestellt. Dabei wurde der logische Testfall in Java implementiert und die dazugehörigen Testdatensätze als XML-Datei mit Variabilität definiert.

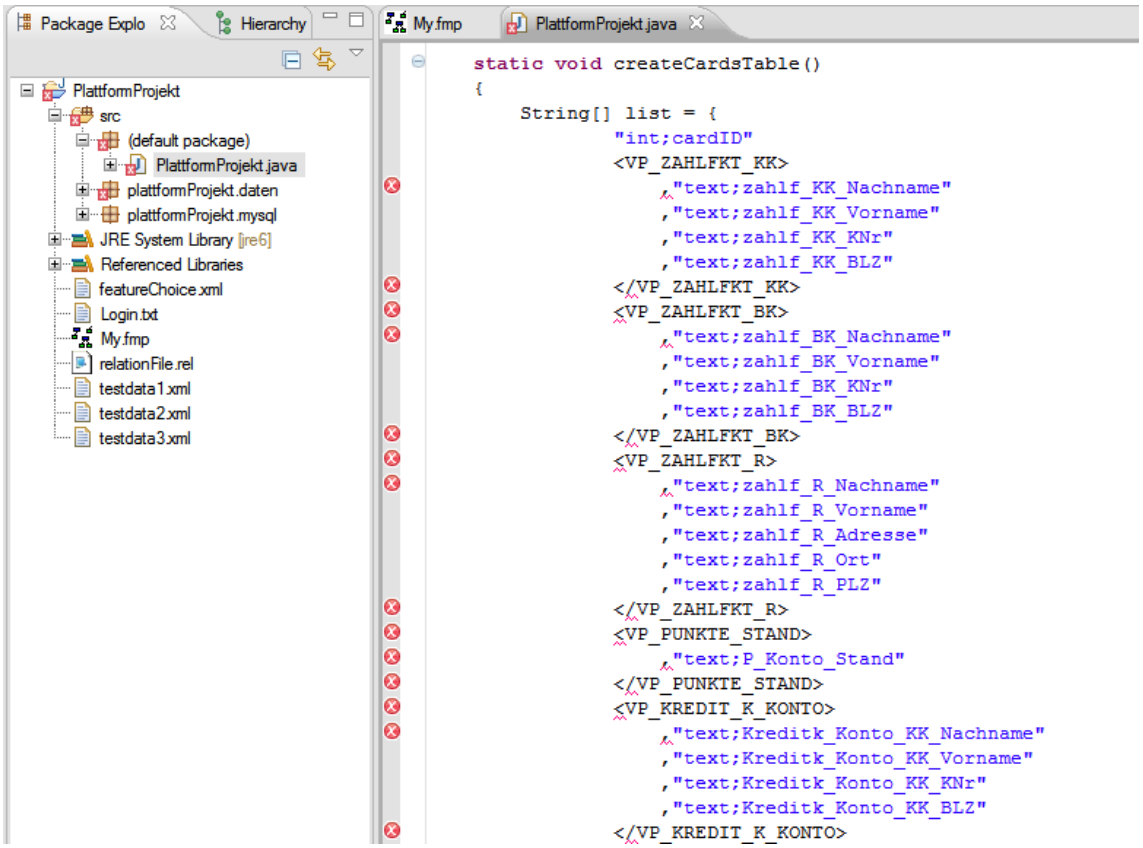


Abbildung 9.7: Schritt 1: Definition von Variabilität in Plattformtestfällen

In Abbildung 9.7 ist mit *PlattformProjekt.java* ein logischer Testfall gegeben. Auf der rechten Seite der Abbildung sind Variationspunkte innerhalb des Testskripts dargestellt. Die dazugehörigen Testdatensätze sind als *testdata1.xml* bis *testdata3.xml* spezifiziert.

Mit Hilfe des *Feature Modeling Plugins* wird beispielhaft das Featuremodell der SPL *Loyaltymanagement* aus Abbildung 9.1 modelliert (vgl. Abbildung 9.8). Das Abbildungsmodell wird in der Datei *relationFile.rel* persistiert.

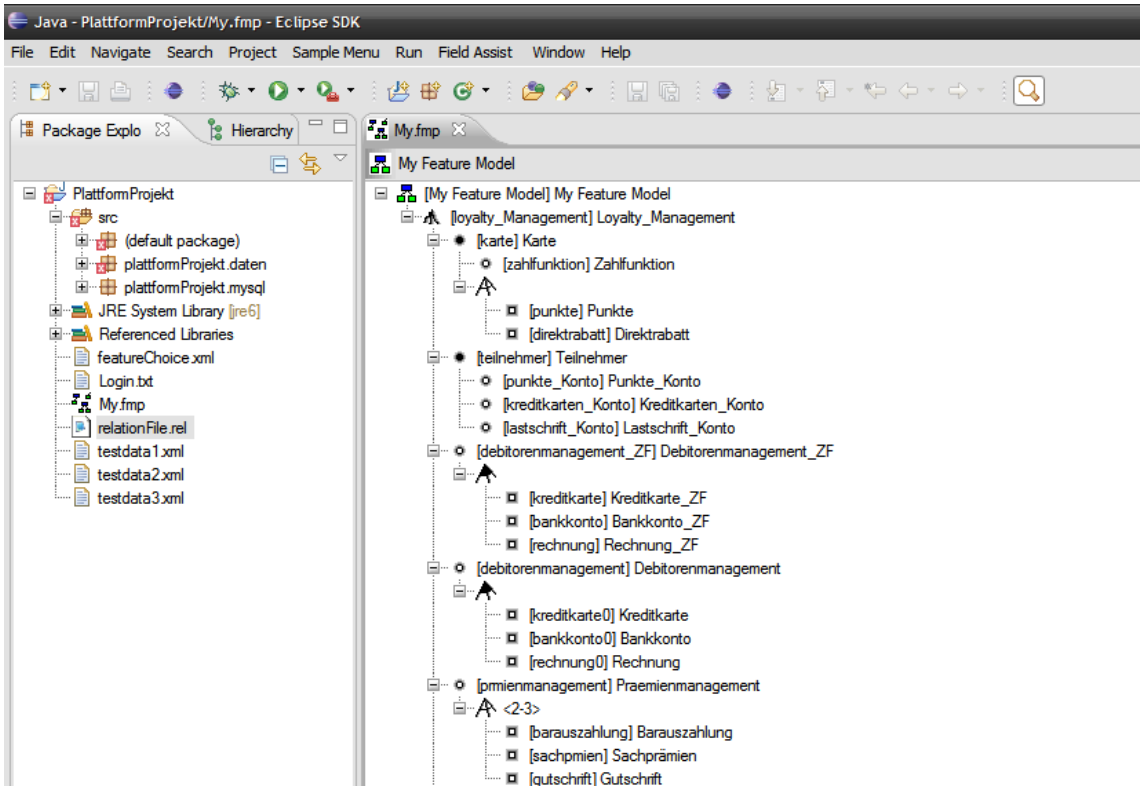


Abbildung 9.8: Featuremodell der SPL Loyaltymanagement

Mit Hilfe dieses Plugins kann durch Auswahl von Features und Attributen eine Featuremodellableitung erstellt werden, wie sie beispielhaft in Abbildung 9.9 dargestellt ist.

Die Featuremodellableitung wird nun gemeinsam mit dem Abbildungsmodell für die Ableitung des Testfalls genutzt. Das Plugin für die Ableitung ist als ein Wizard in die Eclipse-Umgebung integriert worden. Im Folgenden werden die einzelnen Schritte des Wizards beschrieben:

Zunächst muss das abzuleitende Projekt gewählt werden, wie in Abbildung 9.10 dargestellt wird.

Danach wird das zuvor bereits erwähnte Abbildungsmodell, welches in der Datei *relationFile.rel* persistiert ist, ausgewählt, um die Abbildung von Features auf Modellelemente des Testfalls herzustellen (vgl. Abbildung 9.11).

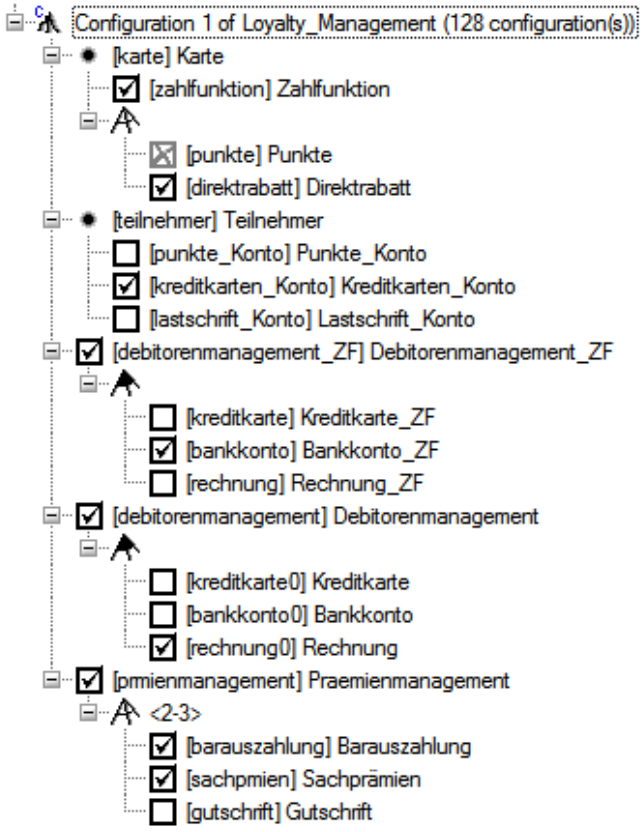


Abbildung 9.9: Featuremodellableitung des Featuremodells zur SPL Loyaltymanagement

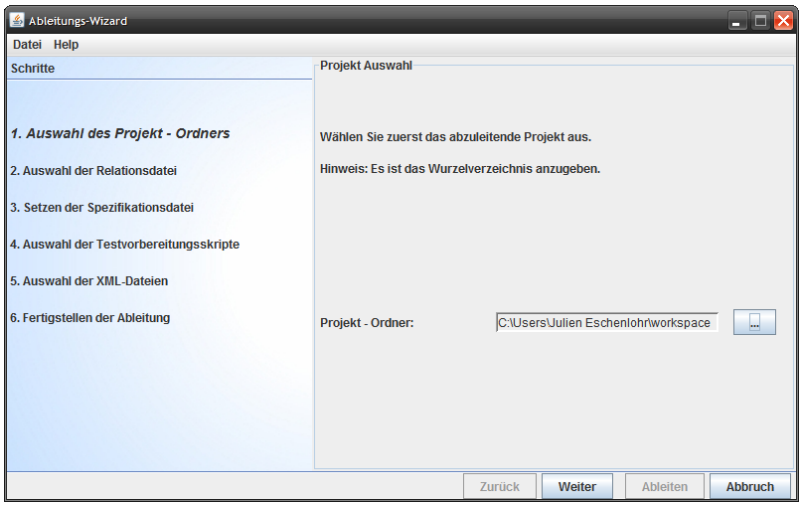


Abbildung 9.10: Ableitungswizard: Projektauswahl

Nach diesem Schritt wird die Featuremodellableitung ausgewählt. Diese wurde

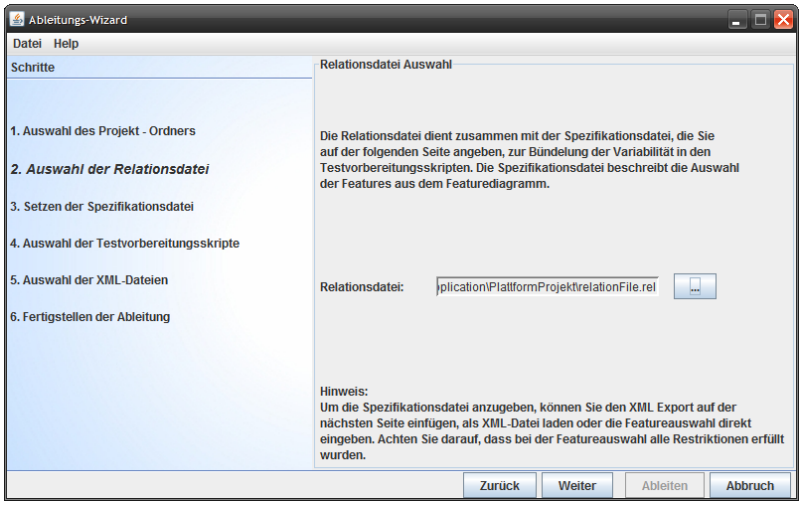


Abbildung 9.11: Ableitungswizard: Abbildungsmodellauswahl

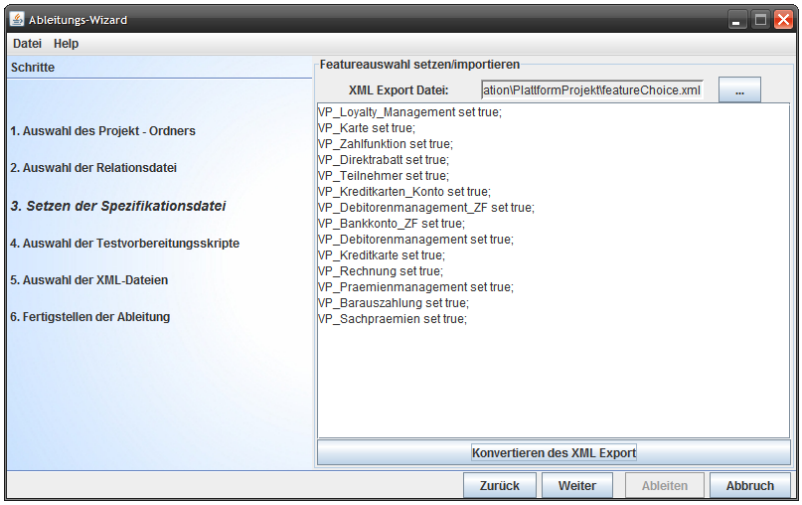


Abbildung 9.12: Ableitungswizard: Auswahl Featuremodellableitung

zuvor mit Hilfe des *Feature Modeling Plugins* erstellt und als *featureChoice.xml* persistiert (vgl. Abbildung 9.12).
Im Anschluss daran werden die Testfälle ausgewählt, die abgeleitet werden sollen. In unserem Beispiel ist dies die Datei *PlattformProjekt.java* (vgl. Abbildung 9.13).

Im letzten Schritt des Wizards werden die drei Testdatensätze zum Testfall ausgewählt, die bereits in Abbildung 9.7 dargestellt waren (vgl. Abbildung 9.14).

Abbildung 9.15 stellt den Testfall nach der Ableitung dar. Im Vergleich zum Plattformtestfall (vgl. Abbildung 9.7) fällt auf, dass die nicht ausgewählten Varianten, sowie bei den ausgewählten Varianten die Variationspunkt-Tags entfernt wurden.

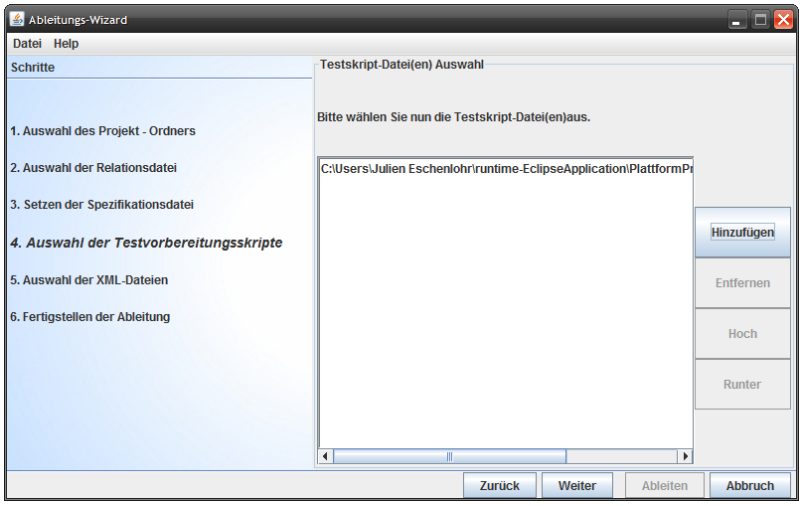


Abbildung 9.13: Ableitungswizard: Auswahl Testfall

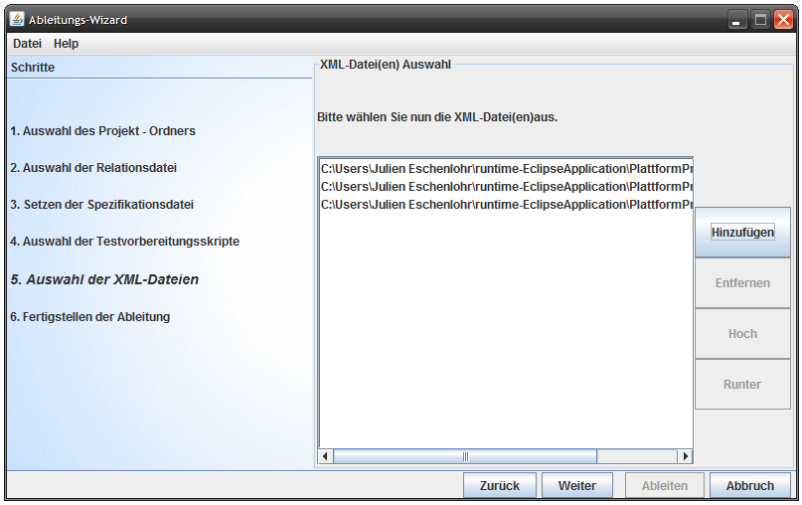


Abbildung 9.14: Ableitungswizard: Auswahl Testdatensätze

9.3 Zusammenfassung

Gegenstand dieses Kapitels war eine Fallstudie mit dem s-lab Industriepartner arvato services sowie eine prototypische Werkzeugunterstützung, um die praktische Anwendbarkeit des in dieser Arbeit vorgestellten Ansatzes zu untersuchen.
Zunächst wurde in Abschnitt 9.1 die Fallstudie vorgestellt, die die Anwendung des Ansatzes auf eine reale Produktlinie demonstriert. Im Rahmen der Anwendung erfolgte die Diskussion der Vor- und Nachteile des Ansatzes.

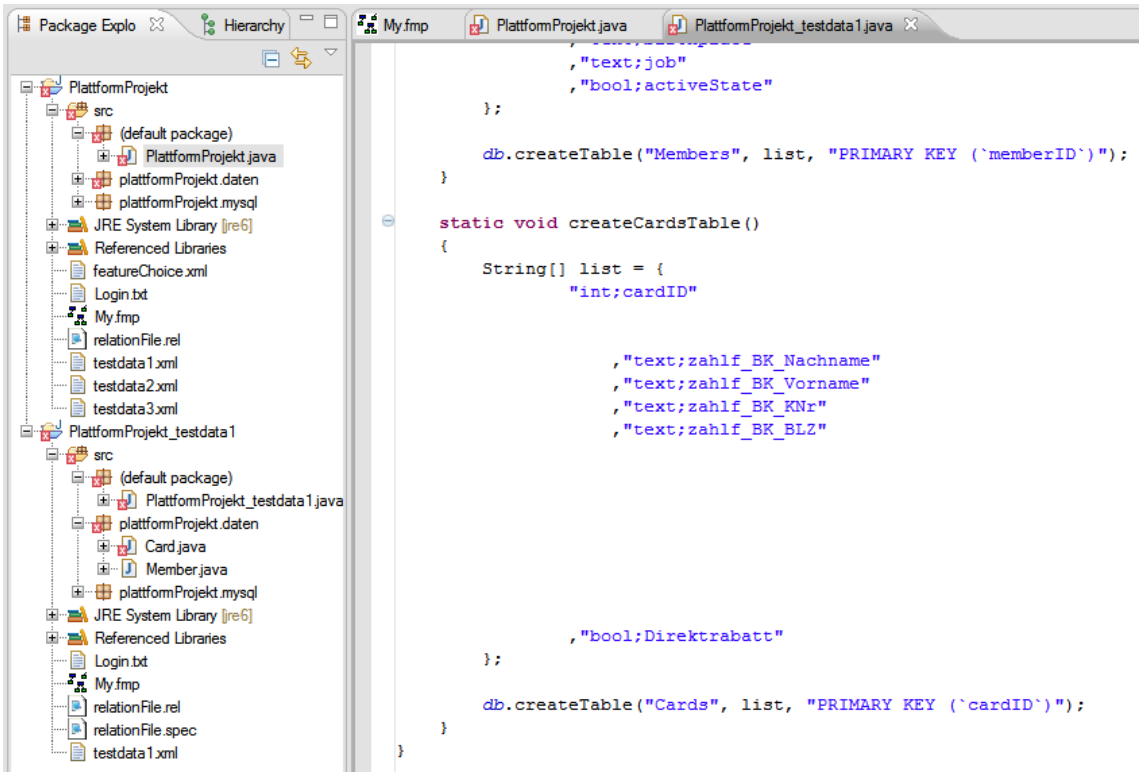


Abbildung 9.15: Beispiel abgeleiteter Produkttestfall

Anschließend wurde die Umsetzung des Werkzeugs für das featurebasierte Variabilitätsmanagement und der Abbildung von Features auf Varianten in Modellen beschrieben (vgl. Abschnitt 9.2). Dieses Werkzeug demonstriert die Funktionalität der Abbildung zwischen Features und Modellelementen, sowie der Ableitung von Produkten über das Featuremodell.

Zusammenfassend lässt sich sagen, dass der in dieser Arbeit vorgestellte Ansatz im praktischen Einsatz das Potential zum einen für die Modellierung und das Management der Variabilität in allen Modelltypen einer SPL besitzt. Zum anderen bietet der vorgestellte Testfallspezifikationsprozess die Möglichkeit alle Bestandteile von Testfällen wiederzuverwenden, was den Anpassungsaufwand während der Produktspezifikation senken konnte. Gerade die Wiederverwendung von Testdaten offerierte hierbei großes Einsparungspotential.

Kapitel 10

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Ansatz für das Variabilitätsmanagement und die Spezifikation von Anforderungen und Testfällen für Software-Produktlinien vorgestellt. In diesem Kapitel wird zunächst eine Zusammenfassung der Ergebnisse gegeben und der Ansatz anschließend auf Erfüllung der in Kapitel 3 und 4 definierten Anforderungen untersucht. Zuletzt wird ein Fazit gezogen und ein Ausblick auf weitere Forschungsthemen geworfen.

10.1 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung eines Ansatzes für die Modellierung und das Management von Variabilität für Software-Produktlinien. Mit diesem Ansatz ist die Modellierung von Variabilität in Anforderungsartefakten und Testfällen möglich. Um diese Möglichkeiten zu nutzen, wurden im Anschluss Spezifikationsprozesse für Anforderungen und Testfälle mit Variabilität definiert.

Zu Beginn der Arbeit sind die Spezifikation von Anforderungs- und Testfallartefakten für Einzelsystementwicklung untersucht und daraus Anforderungen für die Spezifikation dieser Artefakte für Software-Produktlinien definiert worden. Ein Vergleich mit existierenden Ansätzen stellte deren Schwächen im Bezug auf die Modellierung und das Management von Variabilität heraus. Daraus resultierte die Notwendigkeit zum einen für einen neuen Ansatz für das Variabilitätsmanagement in Software-Produktlinien und zum anderen für Ansätze zur Spezifikation von Anforderungen und Testfällen mit Variabilität.

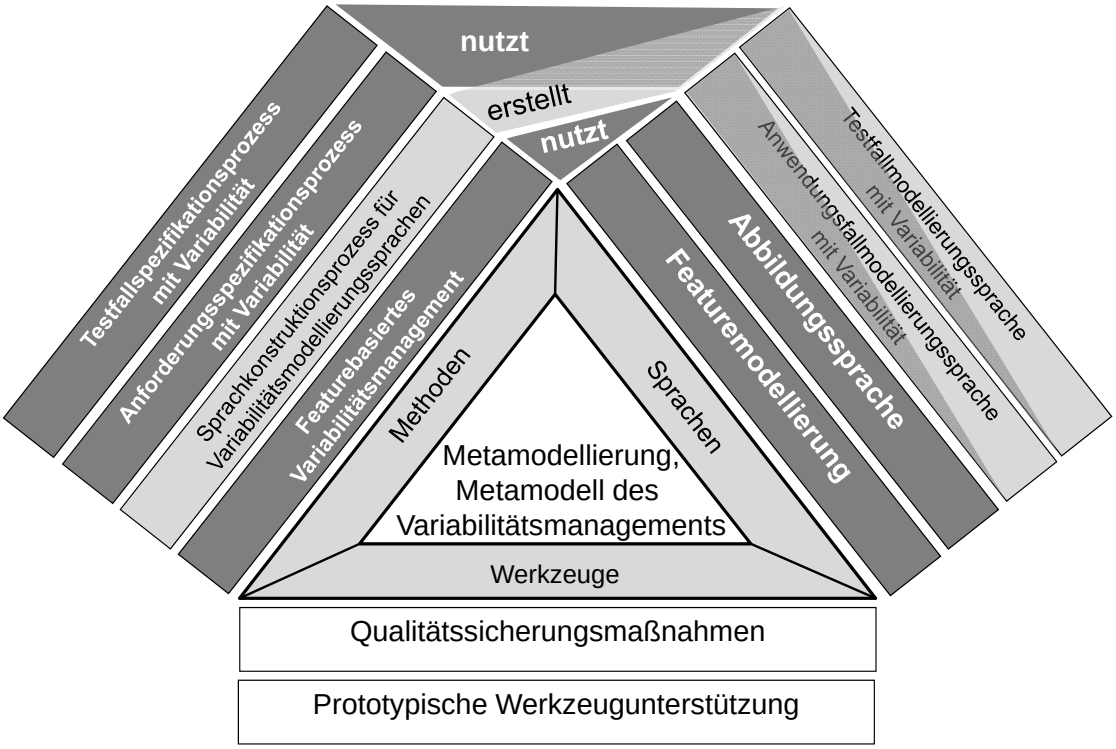


Abbildung 10.1: Überblick über den Beitrag der Arbeit, basierend auf dem Metamodell des Variabilitätsmanagements

Als zentrales Konzept wurde zunächst die Metamodellierung ausgewählt und mithilfe dieser erfolgte Formalisierung der Anforderungen an Variabilitätsmanagement. Dieses formale Modell diente anschließend als Grundlage für die Evaluation existierender Ansätze. Das Ergebnis der Formalisierung, wie in Abbildung 10.1 dargestellt, ist das Metamodell des Variabilitätsmanagements.
Dieses Metamodell stellt das zentrale Konzept für die Definition eines eigenen Ansatzes für die Modellierung und das Management von Variabilität dar.
Zunächst wurde als Methode ein Sprachkonstruktionsprozess für Variabilitätsmodellierungssprachen definiert, der es ermöglicht existierende Modellierungssprachen um Variabilität zu erweitern. Anhand dieses Prozesses erfolgte die Definition einer Anforderungs- sowie Testfallmodellierungssprache mit Variabilität.

Das Management der Variabilität in den Modellen der Software-Produktlinie wird mithilfe des featurebasierten Variabilitätsmanagements ermöglicht. Ein Featuremodell, als Instanz des Featuremodellierung, stellt hierbei das zentrale Modell für das Variabilitätsmanagement dar. Die Idee eines zentralen Modells verbessert die Übersicht über die in der SPL vorhandene Variabilität mit ihren Abhängigkeiten und unterstützt somit Tätigkeiten wie die Ableitung von Produkten sowie die Wartung und Evolution der Software-Produktlinie.
Durch das zentrale Management der Variabilität war es notwendig eine Abbildung zwischen Featuremodell und Variabilitätsmodellierungssprachen zu definieren. Dies wurde durch die Abbildungssprache realisiert.

Das featurebasierte Variabilitätsmanagement und die Anforderungs- sowie Testfallmodellierungssprache werden für den Anforderungs- und Testfallspezifikationsprozess mit Variabilität genutzt.
Der Anforderungsspezifikationsprozess mit Variabilität definiert ein Vorgehen, mit dessen Hilfe existierende Anforderungsmodelle ohne Variabilität oder auch informelle Anforderungen zu Anforderungsmodellen mit Variabilität spezifiziert werden können. Diese Modelle stellen die Eingabe für den Testfallspezifikationsprozess mit Variabilität dar. Die auf diesem Weg spezifizierten Anforderungsmodelle und Testfälle mit Variabilität können für unterschiedliche Produkte der Software-Produktlinie abgeleitet werden.

Im Zuge der Erweiterung von Modellierungssprachen zu Variabilitätsmodellierungssprachen, sowie der Definition des featurebasierten Variabilitätsmanagements wurden neue Typen von Spezifikationsfehlern in den verschiedenen Modellen identifiziert. Um diese Fehler zu finden, erfolgte die Definition von Qualitätssicherungsmaßnahmen, die durch Analysen der entsprechenden Modelle darin existierende Fehler aufdecken.

Nr.	Name	Featurebasiertes Variabilitätsmanagement
A1	Additionspunkt-modellierung	ja
A2	Modifikationspunkt-modellierung	ja
A3	Hierarchische Dekomposition	⊕
A4	Abhängigkeitstypen	Implikation, Ausschluss, Beeinflussung
A5	Bindung	⊕
A6	Konsistenz	⊕

Tabelle 10.1: Erfüllung der Anforderungen an Variabilitätsmanagement durch das featurebasierte Variabilitätsmanagement

Um die praktische Anwendbarkeit des in dieser Arbeit vorgestellten Ansatzes zu demonstrieren, wurde eine Fallstudie mit dem Industriepartner arvato services durchgeführt. Darüber hinaus erfolgte die Entwicklung einer prototypischen Werkzeugunterstützung für das featurebasierte Variabilitätsmanagement, welche die Spezifikation von Variabilität in Testfällen und deren Abbildung auf ein Featuremodell ermöglicht. Mit Hilfe der Werkzeugunterstützung ist die Ableitung von Testfällen für Produkte möglich.

10.2 Ergebnisse der Arbeit

Die Notwendigkeit der Definition des in dieser Arbeit vorgestellten Ansatzes wurde mit Hilfe der in den Kapiteln 3 und 4 vorgestellten Anforderungen und der durchgeführten Evaluation existierender Ansätze begründet. Diese Anforderungen sollen nun dazu dienen, den Ansatz hinsichtlich ihrer Erfüllung zu untersuchen.

10.2.1 Featurebasiertes Variabilitätsmanagement

Durch die Nutzung des Metamodells des Variabilitätsmanagements als Grundlage für den in dieser Arbeit vorgestellten Ansatz werden alle an Variabilitätsmanagement gestellten Anforderungen erfüllt, wie Tabelle 10.1 dargestellt. Die Bewertungen jeder einzelnen Anforderung werden im Folgenden detailliert dargestellt:

A1: Additionspunktmodellierung

Durch die Erweiterung von existierenden Modellierungssprachen mit Hilfe des Metamodells des Variabilitätsmanagements ist die Modellierung von Additionspunkten in Modellen möglich. Das Featuremodell ermöglicht die Modellierung von Additionspunkten über die Domänenbeziehung, wie bereits in Abschnitt 4.3.3 festgestellt wurde.

A2: Modifikationspunktmodellierung

Im Featuremodell werden Modifikationspunkte durch Attribute realisiert. Variabilitätsmodellierungssprachen erben das Konzept der Modifikationspunkte vom Metamodell des Variabilitätsmanagements. Die Anforderung wird dadurch erfüllt.

A3: Hierarchische Dekomposition

Das Featuremodell bietet über die Domänenbeziehung die Möglichkeit der hierarchischen Dekomposition von Features. Bei der Spezifikation von Modellen mit Variabilität erlaubt das Metamodell des Variabilitätsmanagements die hierarchische Dekomposition von Additionspunkten und Varianten. Somit wird die Anforderung durch das featurebasierte Variabilitätsmanagement erfüllt.

A4: Abhängigkeitstypen

Das Featuremodell unterscheidet alle drei geforderten Abhängigkeitstypen. Die Abhängigkeiten werden im Featuremodell definiert und über das Abbildungsmodell auf die Additions- und Modifikationspunkte in Modellen abgebildet.

A5: Bindung

Die Bindung von Variabilität und die dabei existierenden Einschränkungen werden durch den Ansatz erfüllt.

A6: Konsistenz

Die Konsistenz von Featuremodellen wird durch die in [Maß07] vorgestellten Maßnahmen sichergestellt. Diese Maßnahmen und die dabei adressierten Inkonsistenzen gehen über die in Kapitel 4 definierten Anforderungen an Konsistenz hinaus.
Die Sicherung der Konsistenz der übrigen Modelle sowohl in der Plattform als auch nach der Ableitung in Produkten sowie die der Abbildung zwischen Featuremodell und übrigen Modellen, wird durch die Maßnahmen in Kapitel 6 adressiert.
Die Anforderung der Konsistenz wird somit durch den Ansatz erfüllt.

Durch die Nutzung des Metamodells des Variabilitätsmanagements sowie des Featuremodells können alle Anforderungen an Variabilitätsmanagement erfüllt werden. Der Vergleich mit existierenden Ansätzen zeigt die Vorteile des gewählten Ansatzes (vgl. Abschnitt 4.3.3).
Neben den hier vorgestellten Anforderungen an Variabilitätsmanagements existieren weitere, wie zum Beispiel die Software-Produktlinienevolution, die in dieser Arbeit nicht adressiert worden sind (vgl. Abschnitt 4.2.5).

10.2.2 Anforderungs- und Testfallspezifikation für Software-Produktlinien

Die Anforderungen an die Spezifikation von Anforderungsartefakten und Testfällen mit Variabilität wurden in Kapitel 3 definiert. Diese werden nun dazu genutzt, den in dieser Arbeit entwickelten Anforderungs- und Testfallspezifikationsprozess zu bewerten. Die Ergebnisse der Bewertung sind in Tabelle 10.2 dargestellt und werden im Folgenden erläutert.

Anforderung	**Beschreibung**	**Eigener Ansatz**
A1	Modellierung von Variabilität in Anforderungen	⊕
A2	Behandlung von Abhängigkeiten in Anforderungen	⊕
A3	Erweiterung existierender Anforderungsartefakte um Variabilität	ja
A4	Äquivalenzklassenanalyse mit Variabilität	ja
A5	Grenzwertanalyse mit Variabilität	nein
A6	Ablaufanalyse mit Variabilität	ja
A7	Modellierung Testschritte mit Variabilität	ja
A8	Modellierung Vor- und Nachbedingung mit Variabilität	⊙
A9	Modellierung Ein- und Ausgabeparameter mit Variabilität	ja
A10	Modellierung Parametereigenschaften mit Variabilität	⊕
A11	Modellierung Testdaten mit Variabilität	⊕
A12	Unterstützung der Auswahl von Anforderungen aus der Plattform	⊕
A13	Abbildung von variablen Anforderungen auf variable Testfälle	⊕
A14	Automatisierte Ableitung von Testfällen auf Basis ausgewählter Anforderungen	⊕

Tabelle 10.2: Erfüllung der Anforderungen an das Variabilitätsmanagement in Anforderungs- und Testfallspezifikation für SPL

A1: Modellierung von Variabilität in Anforderungen

Diese Arbeit bietet einen Ansatz für die Modellierung von Variabilität in Anwendungsfallbeschreibungen, basierend auf einem Anwendungsfallmetamodell. Dieses Modell wurde im Rahmen der Arbeit um Variabilität erweitert. Alle Bestandteile der Anwendungsfallbeschreibung können somit als variabel definiert werden.

A2: Behandlung von Abhängigkeiten in Anforderungen

Die Modellierung und das Management von Abhängigkeiten zwischen variablen Anforderungen wird im eigenen Ansatz mit Hilfe des featurebasierten Variabilitätsmanagements realisiert. Dabei werden die drei Typen *Implikation*, *Ausschluss* und *Beeinflussung* unterschieden.

A3: Erweiterung existierender Anforderungsartefakte um Variabilität

Im Rahmen der Definition der Anforderungsspezifikation mit Variabilität wurde ein Verfahren für die Identifikation von implizit modellierter Variabilität in existierenden Anforderungsartefakten definiert.

A4: Äquivalenzklassenanalyse mit Variabilität

Die Überdeckung von Parametern mit Variabilität macht eine Äquivalenzklassenanalyse mit Variabilität notwendig. Der in dieser Arbeit vorgestellte Ansatz definiert einen einfachen Algorithmus für die kombinatorische Äquivalenzklassenanalyse mit Variabilität.

A5: Grenzwertanalyse mit Variabilität

Für die Grenzwertanalyse während der Spezifikation von Testdaten mit Variabilität sind die Parameterbestandteile notwendig. Das Vorkommen von Modifikationspunkten in Parameterbestandteilen macht die Analyse von Grenzwerten aufwendiger und wurde in dieser Arbeit nicht betrachtet.

A6: Ablaufanalyse mit Variabilität

Die Analyse und Abdeckung von Abläufen in der Testbasis wird durch den Testfallspezifikationsprozess unterstützt. Das Vorkommen von Variabilität wird differenziert betrachtet und die für die Testfallspezifikation resultierende Variabilität beschrieben.

A7: Modellierung von Testschritten mit Variabilität

Die Spezifikation von Testschritten und die dabei explizite Berücksichtigung von Variabilität wurde im Testfallspezifikationsprozess dieser Arbeit berücksichtigt. Dabei wurden die beiden Typen Additions- und Modifikationspunkt unterschieden und je nach Art der Variabilität der gesamte resultierende Testfall oder nur Teile von ihm als variabel spezifiziert.

A8: Modellierung von Vor- und Nachbedingungen mit Variabilität

Der Testfallspezifikationsprozess unterstützt die Modellierung von variablen Vor- und Nachbedingungen, die aus Vor- oder Nachbedingungen der jeweiligen Testbasis oder aber auch darin enthaltenen Schritten ohne externen Akteur basieren. Die Anforderung wird aber nur teilweise erfüllt, da andere existierende Ansätze eine formale Beschreibung von Vor- und Nachbedingungen mit Variabilität anbieten.

A9: Modellierung von Ein- und Ausgabeparameter mit Variabilität

Diese Anforderung wird erfüllt, da Ein- und Ausgabeparameter und die Modellierung von Variabilität Bestandteil des Testfallmetamodells mit Variabilität sind. Ein Parameter kann somit als variabel deklariert werden oder auch Modifikationspunkte enthalten.

A10: Modellierung von Parametereigenschaften mit Variabilität

Die Eigenschaften von Parametern, wie zum Beispiel Pflichtbedingung und Wertebereich, werden durch den Ansatz unterstützt. Die Modellierung von Variabilität wird dabei explizit vorgesehen. Die Modellierung wird möglich, da Informationen aus den Implementierungsmodellen der Produktlinienplattform genutzt werden.

A11: Modellierung von Testdaten mit Variabilität

Das dem Ansatz zugrunde liegende Testfallmetamodell berücksichtigt explizit die Modellierung von Testdatensätzen und deren Testdaten. Durch die Spezifikation von Parametereigenschaften und der darin vorhandenen Variabilität ist es möglich, Testdaten mit Variabilität zu spezifizieren.

A12: Unterstützung der Auswahl von Anforderungen aus der Plattform

Die einfache und konsistente Auswahl von variablen Bestandteilen aus der Produktlinienplattform durch den Auftraggeber wird in dieser Arbeit durch das featurebasierte Variabilitätsmanagement realisiert. Einfach wird die Auswahl dadurch, dass der Auftraggeber die gewünschten Bestandteile in einem zentralen Modell auswählt, nämlich dem Featuremodell. Die Konsistenz kann dabei durch den Ansatz zur Konsistenzsicherung aus [Maß07] realisiert werden.

A13: Abbildung von variablen Anforderungen auf variable Testfälle

Mithilfe des featurebasierten Variabilitätsmanagement ist die Abbildung von variablen Bestandteilen aus Anforderungsartefakten und Testfällen möglich. Die Spezifikation von Abbildungen wurde in den Testfallspezifikationsprozess mit Variabilität integriert.

A14: Automatisierte Ableitung von Testfällen auf Basis ausgewählter Anforderungen

Durch die Auswahl von Features im Featuremodell werden automatisch auch die zu den gewählten Features passenden Testfälle ausgewählt. Die Abbildung zwischen Features und Testfällen wird im eigenen Ansatz durch das Abbildungsmodell realisiert.

Bis auf die Grenzwertanalyse mit Variabilität werden alle gestellten Anforderungen durch den Ansatz erfüllt. Neben den hier definierten und sehr auf Variabilität ausgerichteten Anforderungen können weitere, testspezifische Anforderungen an einen Testfallspezifikationsprozess formuliert werden. Darunter sind zum Beispiel Anforderungen hinsichtlich der Automatisierung und Abdeckung. Diese Anforderungen standen bei der Erstellung des Ansatzes in dieser Arbeit nicht im Fokus und

bedürfen weiterer Evaluation und Erweiterungen. Neben diesen möglichen Erweiterungen werden im nun folgenden Abschnitt weitere Maßnahmen zur Erweiterung oder Verbesserung des in dieser Arbeit vorgestellten Ansatzes diskutiert.

10.3 Ausblick

Das featurebasierte Variabilitätsmanagement und die darauf basierenden Anforderungs- und Testfallspezifikationsprozesse unterstützen die Entwicklung von Software-Produktlinien und damit als zentrales Ziel die organisierte Wiederverwendung von Modellen.
Der präsentierte Ansatz zeigt Abhängigkeiten zu anderen Bereichen der Entwicklung von Software-Produktlinien und kann daher auf unterschiedliche und vielfältige Art und Weise verbessert werden. Dieser Abschnitt gibt daher einen Ausblick auf mögliche Erweiterungen und Verbesserungen. Einige davon adressieren die in Abschnitt 3.1.2 beschriebenen Problemstellungen, die für diese Arbeit ausgeschlossen worden sind.

Die Formalisierung von Anforderungen mit Hilfe von Metamodellen bietet die Möglichkeit einer nachvollziehbaren Evaluation, inwiefern existierende Ansätze die formalisierten Anforderungen erfüllen. In dieser Arbeit wurde eine solche Evaluation exemplarisch für das Variabilitätsmanagement durchgeführt. Um diesen Ansatz praktisch einsetzbar zu machen, müsste eine Formalisierung des Vergleichs sowie die Definition dessen Voraussetzungen stattfinden und dieser durch ein Werkzeug unterstützt werden. Dabei könnten Techniken wie Graphtransformationen [HEWC01] für die Feststellung eines Isomorphismus sowie Ontologien [KS05, SK06] für den ontologischen Abgleich eingesetzt werden.
Die Modellierung von Anforderungen im vorgestellten Ansatz fokussiert funktionale Anforderungen. Die Modellierung von nicht-funktionalen Anforderungen [ISO] mit Variabilität wurde bisher nicht berücksichtigt. In diesem Kontext sollte vor allem die Featuremodellierung auf Modellierbarkeit von nicht-funktionalen Anforderungen untersucht werden.

Im featurebasierten Variabilitätsmanagement wird bisher nur der Bindungszeitpunkt „Produktableitung“ explizit unterstützt. Die Literatur [PM06] definiert weitere Bindungszeitpunkte für Variabilität. Weiterführende Arbeiten könnten die Erweiterung des Ansatzes für weitere Bindungszeitpunkte untersuchen.

Der vorgestellte Testfallspezifikationsprozess kann auch für den Test der Plattform eingesetzt werden. Zu diesem Zweck könnte er mit existierenden Ansätzen, die zum Beispiel den kombinatorischen Test von Produktlinienplattformen ermöglichen, verknüpft werden. Solche Ansätze sind zum Beispiel [Oli08] und [OSW08]. Ein erster Beitrag für eine solche Verknüpfung wurde bereits in [WO10] beschrieben.
Durch eine weitere Formalisierung der Testbasis, z. B. durch visuelle Kontrakte [Loh06, GMWE09] für die Modellierung von Vor- und Nachbedingungen oder den Einsatz von Verhaltensmodellen basierend auf der UML, welche eine formale Syntax definieren, könnten Techniken des modellbasierten Testens eingesetzt werden, um eine automatisierte Ableitung von Testfällen zu ermöglichen [PP04, Mly10].
Neben der Formalisierung der Syntax für die Modellierung von Variabilität wäre auch die Formalisierung ihrer Semantik eine interessante Erweiterung. In [Hau05] wird die Formalisierung der Semantik in UML-Verhaltensdiagrammen untersucht. In [EW10] wird ein erster Schritt für die Formalisierung der Semantik von Variabilität beschrieben. Beide Ansätze könnten den in dieser Arbeit präsentierten Ansatz erweitern.
Im Kontext des Testprozesses wäre auch eine genaue Beleuchtung der Grenzwertanalyse mit Variabilität, vor allem hinsichtlich von Modifikationspunkten interessant. Durch eine solche Erweiterung könnte die Wiederverwendbarkeit weiter verbessert werden.
Durch das in dieser Arbeit gewählte Testfallmetamodell ist der Aufbau von Testfällen und Testschritten gleich. Dies könnte die Nutzung von Testfällen einer niedrigeren Teststufe als Testschritte für eine darüber liegende Teststufe ermöglichen. Diese Wiederverwendung von Testfällen auf unterschiedlichen Teststufen steht orthogonal zur Wiederverwendung von Testfällen für unterschiedliche Produkte. Weitere Forschungen in diesem Umfeld könnte das Vorhaben der Wiederverwendung von Testartefakten auf unterschiedlichen Teststufen ermöglichen.

Die Werkzeugunterstützung des Ansatzes sollte weiter ausgebaut werden. Zunächst wäre ein Modul für die Erweiterung von Modellierungssprachen um Variabilität notwendig, welches den Sprachingenieur bei seiner Arbeit unterstützt. Damit wäre die Erweiterung aller Sprachen, die bei der Entwicklung von Software-Produktlinien eingesetzt werden, möglich. Korrelierend dazu könnte der in dieser Arbeit vorgestellte Ansatz zur Erweiterung von Modellierungssprachen auf praktische Einsetzbarkeit evaluiert werden.
Eine Werkzeugunterstützung für die Spezifikation von Testfällen würde den Testdesigner bei seiner Arbeit unterstützen. Hierbei könnten auch existierende Testfallspezifikationswerkzeuge erweitert werden.

Weiterhin ist der Bereich der Software-Produktlinienevolution im Rahmen dieser Arbeit ausgeschlossen worden [Bos00, Pus02]. In weiteren Arbeiten könnte die Auswirkung der Evolution auf das Variabilitätsmanagement untersucht werden.

Eine weitere Form der Konsistenzsicherung kann im Rahmen des Variabilitätsmanagements zur Vermeidung von Fehlern beitragen. Beispielsweise sind Additionspunkte mit Varianten in einem Anforderungsmodell vorhanden, welches als Testbasis dient. In den zu diesem Anforderungmodell erstellten Testfällen müssen alle Varianten des Anforderungsmodells berücksichtigt werden. Allgemeiner formuliert muss die Variabilität zwischen verschiedenen Entwicklungsphasen einer SPL konsistent sein (Phasenkonsistenz).

Die angeführten Verbesserungen und Erweiterungen schließen diese Arbeit ab. Diese Liste lässt sich sicherlich um weitere Punkte ergänzen.

10.4 Schlussbemerkung

Das Software-Produktlinienparadigma stellt einen vielversprechenden Ansatz für die organisierte Wiederverwendung von Artefakten bei der Entwicklung von softwareintensiven Systemen dar. Die Modellierung und das Management von Variabilität ermöglichen hierbei die Anpassung der Produkte an die Bedürfnisse der Kunden. Trotzdem ist gerade diese Variabilität für den Anstieg der Komplexität der Software-Produktlinienentwicklung im Vergleich zur Entwicklung von Einzelsystemen verantwortlich. Um die Vorteile des Software-Produktlinienparadigmas nutzbar zu machen, sind Konzepte notwendig, die diese gesteigerte Komplexität beherrschbar machen.

Das featurebasierte Variabilitätsmanagement in Anforderungs- und Testfallspezifikation stellt ein Konzept dar, welches dieses Problem adressiert. Der in dieser Arbeit vorgestellte Ansatz bietet somit einen Beitrag für die Umsetzung des Software-Produktlinienparadigmas in der Praxis und kann sicherlich durch die Kombination mit anderen Ansätzen weiter verbessert werden.

Literaturverzeichnis

[AEMW09] Assmann, M. ; Engels, G. ; Massen, T. von d. ; Wübbeke, A.: Identifying Software Product Line Component Services. In: *Proceedings of International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE '09)*, 2009, S. 45–56

[Bac01] Bachmann, F.: Managing Variability in Software Architectures. In: *Proceedings of the 2001 Symposium on Software Reusability*, 2001, S. 126–132

[Bal01] Balzert, H.: *Lehrbuch der Software-Technik. Bd.1. Software-Entwicklung.* Spektrum Akademischer Verlag, 2001

[BBM05] Berg, K. ; Bishop, J. ; Muthig, D.: Tracing Software Product Line Variability: From Problem to Solution Space. In: *Proceedings of the 2005 Annual Research Conference of the South African institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT '05)*, 2005, S. 182–191

[BFGL04] Bertolino, A. ; Fantechi, A. ; Gnesi, S. ; Lami, G.: Product Line Use Cases / Istituto di Scienze e Technologie dell'Informazione A.Faedo, C.N.R. Pisa, 2004. – Forschungsbericht

[BFGL06] *Kapitel* Product Line Use Cases: Scenario-Based Specification and Testing of Requirements. In: Bertolino, A. ; Fantechi, A. ; Gnesi, S. ; Lami, G.: *Software Product Lines: Research Issues in Engineering and Requirements.* Springer Verlag, 2006, S. 425–445

[BG03a] Bertolino, A. ; Gnesi, S.: PLUTO: A Test Methodology for Product Families. In: *Lecture Notes in Computer Science* Bd. 3014, Springer Verlag Heidelberg, 2003, S. 181–197

[BG03b] Bertolino, A. ; Gnesi, S.: Use Case-based Testing of Product Lines. In: *SIGSOFT Software Engineering Notes* 28 (2003), Nr. 5, S. 355–358

[Bin99] Binder, R. V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Longman, 1999

[Boe79] Boehm, B. W.: Guidelines for Verifying and Validating Software Requirements and Design Specification. In: *Proceedings of EURO IFIP 1979*, 1979, S. 711–719

[Bos00] Bosch, J.: *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach.* Addison-Wesley Longman, 2000

[Bos02] Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: *Proceedings of the Second International Conference on Software Product Lines (SPLC '02)*, 2002, S. 257–271

[BPSM+08] Bray, T. ; Paoli, J. ; Sperberg-McQueen, C. M. ; Maler, E. ; Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fifth Edition) / W3C. http://www.w3.org/TR/2008/REC-xml-20081126/, 2008. – Forschungsbericht

[Bro09] Brockmann, J.: *Entwicklung eines Baukasten zur Modellierung von Geschäftsprozessvarianten im Bereich von Kundenbindungsprogrammen.* Bielefeld, Fachhochschule der Wirtschaft (FHDW), Bachelorarbeit, 2009

[BS04] Broy, M. ; Steinbrüggen, R.: *Modellbildung in der Informatik.* Springer, Berlin Heidelberg, 2004

[Béz01] Bézivin, J.: From Object Composition to Model Transformation with the MDA. In: *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS '01)*, 2001, S. 348–350

[CDS06] Cohen, M. B. ; Dwyer, M. B. ; Shi, J.: Coverage and Adequacy in Software Product Line Testing. In: *Proceedings of the ISSTA 2006*

Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA '06), 2006, S. 53–63

[CN98] Cusumano, M.A. ; Nobeoka, K.: *Thinking Beyond Lean - How Multi-Project Management is Transforming Product Development at Toyota and Other Companies.* New York : Free Press, 1998

[CN01] Clements, P. ; Northrop, L.: *Software Product Lines : Practices and Patterns.* Addison-Wesley Professional, 2001

[Coc00] Cockburn, A.: *Writing Effective Use Cases.* Addison-Wesley Professional, 2000

[Dij70] Dijkstra, E. W.: Notes On Structured Programming / T.H.-Report 70-WSK-03. 1970. – Forschungsbericht

[Don01] Donaldson, A. J. M.: A Case Narrative of the Project Problems with the Denver Airport Baggage Handling System (DABHS) / Software Forensics Centre Technical Report TR 2002-01. 2001. – Forschungsbericht

[DW00] Dröschel, W. ; Wiemers, M.: *Das V-Modell 97.* Oldenbourg Verlag München Wien, 2000

[Dym02] Dymond, K. D.: *CMM Handbuch. Das Capability Maturity Model für Software.* Springer, Berlin, 2002

[EBB05] Eriksson, M. ; Börstler, J. ; Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In: *Proceedings of the 9th International Conference on Software Product Lines (SPLC '05)*, 2005, S. 33–44

[Ecl] Eclipse Feature Modeling Plugin: *http://gsd.uwaterloo.ca/projects/fmp-plugin/ (Stand: 05.07.2010)*

[EE95] Ebert, J. ; Engels, G.: Specialization of Object Life Cycle Definitions / University of Koblenz-Landau. 1995 (19/95). – Fachbericht Informatik

[Emk09] Emken, A.: *Identifkation und Modellierung von Variabilität in Anwendungsfällen und deren Testartefakten in einer Software-Produktlinie*, Universität Paderborn, Diplomarbeit, 2009

[Erl06] Erl, T.: *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice-Hall, 2006

[ES04] Ehrig, M. ; Staab, S.: QOM - Quick Ontology Mapping. In: *Proceedings of Third International Semantic Web Conference (ISWC '04)*, 2004, 7-11

[Esc09] Eschenlohr, J.: *Automatisierte Ableitung von Testskripten für eine Software-Produktlinie*, Universität Paderborn, Bachelorarbeit, 2009

[EW10] Erwig, M. ; Walkingshaw, E.: *The Choice Calculus: A Representation for Software Variation.* 2010. – School of Electrical Engineering and Computer Science Oregon State University

[FFVM04] Fuentes-Fernández, L. ; Vallecillo-Moreno, A.: An Introduction to UML Profiles. In: *European Journal for the Informatics Professional* 7 (2004), S. 6–13

[FGLN04] Fantechi, A. ; Gnesi, S. ; Lami, G. ; Nesti, E.: A Methodology for the Derivation and Verification of Use Cases for Product Lines. In: *Proceedings of Third International Conference on Software Product Lines (SPLC '04)*, 2004, S. 255–265

[FP05] Friske, M. ; Pirk, J.: Werkzeuggestützte interaktive Formalisierung textueller Anwendungsfallbeschreibungen für den Systemtest. In: *Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (Band2)*, 2005

[FS05] Friske, M. ; Schlingloff, H.: Von Use Cases zu Test Cases: Eine systematische Vorgehensweise. In: *Tagungsband Dagstuhl-Workshop MBEES: Model Based Engineering of Embedded Systems*, 2005, S. 1–10

[GAd98] Griss, D. ; Allen, R. ; d'Allesandro, M.: Integrating Feature Modelling with the RSEB. In: *Proceedings of the 5th International Conference of Software Reuse (ICSR '98)*, 1998, S. 76–85

[Gen] Generic Modeling Environment (GME): *http://www.isis.vanderbilt.edu/projects/gme (Stand 15.07.2010)*

[GLRW04] Geppert, B. ; Li, J. ; Rössler, F. ; Weiss, D. M.: Towards Generating Acceptance Tests for Product Lines. In: *Proceeding of 8th International Conference on Software Reuse : Methods, Techniques, and Tools (ICSR '04)*, 2004, S. 35–48

[GMWE09] Güldali, B. ; Mlynarski, M. ; Wübbeke, A. ; Engels, G.: Model-Based System Testing Using Visual Contracts. In: *Proceedings of Euromicro SEAA Conference 2009, Special Session on Model Driven Engineering*, 2009, S. 121–124

[GO08] Gomaa, H. ; Olimpiew, E. M.: Managing Variability in Reusable Requirement Models for Software Product Lines. In: *Proceedings of the 10th International Conference on Software Reuse (ICSR '08)*, 2008, S. 182–185

[Hau05] Hausmann, J. H.: *Dynamic Meta Modeling - A Semantics Description Technique for Visual Modeling Languages*, Universität Paderborn, Doktorarbeit, 2005

[HC01] Heineman, G. T. ; Councill, W. T.: *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley Longman, 2001

[HDHMM06] Hörmann, K. ; Dittmann, L. ; Hindel, B. ; M. Müller, M: *SPICE in der Praxis. Interpretationshilfe für Anwender und Assessoren.* dpunkt, Heidelberg, 2006

[HEWC01] Heckel., R. ; Ehrig., H. ; Wolter, U. ; Corradini, A.: Double-Pullback Transitions and Coalgebraic Loose Semantics for Graph Transformation Systems. In: *ACPS* 9 (2001), Nr. 1, S. 83–110

[Hor06] Horn, L. R.: *Contradiction.* Stanford Encyclopedia of Philosophy. `http://plato.stanford.edu/entries/contradiction/ (Stand:13.06.2010)`. Version: 2006

[HP04] Halmans, G. ; Pohl, K.: Communicating the Variability of a Software-Product Family to Customers. In: *Informatik - Forschung und Entwicklung* 2 (2004), Nr. 18, S. 113–131

[HU94] Hopcroft, J. E. ; Ullman, J. D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Addison Wesley, Bonn, 1994

[HVF05] Hartmann, J. ; Vieira, M. ; Foster, H.: A UML-based Approach to System Testing. In: *Innovations in Systems and Software Engineering* 1 (2005), S. 12–24

[IEE90] IEEE Computer Society: IEEE Standard Glossary of Software Engineering Terminology / IEEE Std 610.12-1990. New York, 1990. – Forschungsbericht

[IEE98] IEEE Computer Society: IEEE Recommended Practice for Software Requirements Specifications / IEEE Std 830-1998. New York, 1998. – Forschungsbericht

[Int05] International Standards Organization: *Quality Management Systems - Fundamentals and Vocabulary (ISO 9000:2005).* 2005

[ISO] ISO9000: *http://www.iso.org (Stand: 10.07.2010)*

[JKB08] Jaring, M. ; Krikhaar, R. L. ; Bosch, J.: Modeling Variability and Testability Interaction in Software Product Line Engineering. In: *International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, 2008, S. 120–129

[JM02] John, I. ; Muthig, D.: Tailoring Use Cases for Product Line Modeling. In: *Proceedings of the International Workshop on Requirements Engineering for Product Lines*, 2002, S. 26–32

[KCH+90] Kang, K. C. ; Cohen, S. ; Hess, J. H. ; Novak, W. E. ; Peterson, A. S.: Feature Oriented Domain Analysis (FODA) Feasibility Study / CMU/SEI-90-TR21. 1990. – Forschungsbericht

[KKL+98] Kang, K. C. ; Kim, S. ; Lee, J. ; Kim, K. ; Kim, G. J. ; Shin, E.: FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. In: *Annals of Software Engineering* 5 (1998), S. 143–168

[KL07] Korherr, B. ; List, B.: A UML 2 Profile for Variability Models and Their Dependency to Business Processes. In: *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA '07)*, 2007, S. 829–834

[KLKL07] Kang, S. ; Lee, J. ; Kim, M. ; Lee, W.: Towards a Formal Framework for Product Line Test Development. In: *Proceedings of the 7th IEEE International Conference on Computer and Information Technology (CIT '07)*, 2007, S. 921–926

[KNK05] Kishi, T. ; Noda, N. ; Katayama, T.: Design Verification for Product Line Development. In: *Proceedings of the 9th International Conference on Software Product Lines (SPLC '05)*, 2005, S. 150–161

[KPRR03] Kamsties, E. ; Pohl, K. ; Reis, S. ; Reuys, A.: Testing Variabilities in Use Case Models. In: *Proceedings of the 5th International Workshop on Software Product-Family Engineering*, 2003, S. 6–18

[KPRR04] Kamsties, E. ; Pohl, K. ; Reis, S. ; Reuys, A.: Szenario-basiertes Systemtesten von Software-Produktfamilien mit ScenTED. In: *Proceedings of Modellierung '04*, 2004, 169-186

[Kru99] Kruchten, P.: *Der Rational Unified Process. Eine Einführung.* Addison-Wesley, 1999

[KS05] Kalfoglou, Y. ; Schorlemmer, M.: Ontology Mapping: The State of the Art. In: *Proceedings of Dagstuhl Seminar on Semantic Interoperability and Integration*, 2005

[KT03] Kauppinen, R. ; Taina, J.: RITA Environment for Testing Framework-based Software Product Lines. In: *Proceedings of the Eighth Symposium on Programming Languages and Software Tools (SPLST '03)*, 2003, S. 58–69

[KV03] Kropfitsch, D. ; Vigenschow, U.: Das Fehlermodell: Aufwandsschätzung und Planung von Tests. In: *Objektspektrum* 6 (2003), S. 30–35

[LGFM98] Laboratory, M. G. ; Griss, M. L. ; Favaro, J. ; Methodologist, C.: Integrating Feature Modeling with the RSEB. In: *Proceedings of the Fifth International Conference on Software Reuse (ICSR '05)*, 1998, S. 76–85

[Lig02] Liggesmeyer, P.: *Software-Qualität*. Spektrum Akademischer Verlag, Heidelberg, 2002

[Lin02] Linden, F. van d.: Software Product Families in Europe: The Esaps & Cafe Projects. In: *Software, IEEE* 19 (2002), Nr. 4, S. 41–49

[LKK04] Lee, J. ; Kang, K. C. ; Kim, S.: A Feature-Based Approach to Product Line Production Planning. In: *Proceedings of the Third Software Product Lines Conference (SPLC '04)*, 2004

[LMNW03] Lichter, H. ; Massen, T. von d. ; Nyssen, E. ; Weiler, T.: Vergleich von Ansätzen zur Featuremodellierung bei der Softwareproduktlinienentwicklung / RWTH Aachen. 2003. – Forschungsbericht

[Loh06] Lohmann, M.: *Kontraktbasierte Modellierung, Implementierung und Suche von Komponenten in serviceorientierten Architekturen*, Universität Paderborn, Doktorarbeit, 2006

[Maß07] Massen, T. von d.: *Feature-basierte Modellierung und Analyse von Variabilität in Produktlinienanforderungen*, RWTH Aachen, Doktorarbeit, 2007

[Mba09] Mbayiha, P.: *Testdesign und Testdatengenerierung für Schnittstellen-tests unter Berücksichtigung von Variabilität*, Universität Paderborn, Diplomarbeit, 2009

[McG01] McGregor, J. D.: Testing a Software Product Line / Carnegie Mellon Software Enineering Institute. 2001. – Forschungsbericht

[McG02] McGregor, J. D.: Building Reusable Test Assets for a Product Line. In: *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, 2002, S. 345–346

[McG03] McGregor, J. D.: The Evolution of Product Line Assets / Carnegie Mellon Software Enineering Institute. 2003. – Forschungsbericht

[Meta] Metaedit: *http://www.metacase.com (Stand: 15.07.2010)*

[Metb] Metamodeling with DOME: *http://www.htc.honeywell.com/dome (Stand: 15.07.2010)*

[MH03] Muccini, H. ; Hoek, A. van d.: Towards Testing Product Line Architectures. In: *International Workshop on Test and Analysis of Component-Based Systems*, 2003, 111-121

[Mis06] Mishra, S.: Specification Based Software Product Line Testing: A Case Study. In: *Proceedings of the Conference on Concurrency, Specification and Programming (CS&P '06)*, 2006, S. 243–254

[ML97] Meyer, M. ; Lehnerd, A.: The Power of Product Platforms. In: *Free Press*. New York, 1997

[Mly10] Mlynarski, M.: Holistic Model-Based Testing for Business Information Systems. In: *Proceedings of 3rd International Conference on Software Testing, Verification and Validation*, 2010, S. 327–330

[MW08] Massen, T. von d. ; Wübbeke, A.: Modellierung von Variabilität in der Geschäftsanalyse - eine industrielle Fallstudie. In: *Proceedings of Software Engineering 2008, Workshopband, Produktlinien im Kontext: Technologie, Prozesse, Business und Organisation (PiK '08)*, 2008, S. 285–296

[MW09] Massen, T. von d. ; Wübbeke, A.: Lösungsorientierte Software Produktlinienentwicklung in heterogenen Systemlandschaften. In: *Hildesheimer Informatikberichte - Tagungsband der PIK 2009 Produktlinien im Kontext*, 2009, S. 15–23

[NFLJ03] Nebut, C. ; Fleurey, F. ; Le Traon, Y. ; Jézéquel, J.-M.: A Requirement-based Approach to Test Product Families. In: *Proceedings of the 5th Workshop on Product Families Engineering (PFE '05)*, 2003, S. 198–210

[NPLJ02] Nebut, C. ; Pickin, S. ; Le Traon, Y. ; Jézéquel, J.-M.: Reusable Test Requirements for UML-Modeled Product Lines. In: *Proceedings of Workshop on Requirements Engineering for Product Lines (REPL '02)*, 2002, S. 51–56

[NPLJ03] Nebut, C. ; Pickin, S. ; Le traon, Y. ; Jézéquel, J.-M.: Automated Requirements-Based Generation of Test Cases for Product Families. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03)*, 2003, S. 263–266

[NTJ06] *Kapitel* System Testing of Product Families: from Requirements to Test Cases. In: Nebut, C. ; Traon, Y L. ; Jézéquel, J.-M.: *Software Product Lines*. Springer Verlag, 2006, S. 447–478

[NUOT01] Nakatani, T. ; Urai, T. ; Ohmura, S. ; Tamai, T.: A Requirements Description Metamodel for Use Cases. In: *Proceedings of the Asia-Pacific Software Engineering Conference*, 2001, S. 251

[Obe09a] Oberhokamp, C.: *Konzeption und Vergleich von Testdesign-Strategien für Software-Produktlinien*, Universität Paderborn, Diplomarbeit, 2009

[Obe09b] Oberhokamp, C.: *Verfolgbarkeit von Variabilität in Software-Produktlinien*, Universität Paderborn, Seminararbeit, 2009

[OG05] Olimpiew, E. M. ; Gomaa, H.: Model-Based Testing for Applications Derived from Software Product Lines. In: *Proceedings of the 1st International Workshop on Advances in Model-Based Testing*, 2005, S. 1–7

[Oli08] Olimpiew, E. M.: *Model-Based Testing for Software Product Lines*, George Mason University, Doktorarbeit, 2008

[OMG03] OMG: *Object Management Group - Object Constraint Language Version 2.2.* http://www.omg.org/spec/OCL/2.2 : http://www.omg.org/spec/OCL/2.2, 2003

[OMG08] OMG: *Object Management Group - Software Process Engineering Meta-Model, Version 2.0.* http://www.omg.org/technology/documents/formal/spem.htm, 2008

[OMG09] OMG: *Object Management Group - Unified Modeling Language, Superstructure 2.2.* http://www.omg.org/cgi-bin/doc?formal/09-02-02.pdf, 2009

[OSW08] Oster, S. ; Schürr, A. ; Weisemöller, I.: Towards Software Product Line Testing Using Story Driven Modelling. In: *Proceedings of the 6th International Fujaba Days*, 2008, S. 48–51

[OWES10] *Kapitel* Model-Based Software Product Lines Testing Survey. In: Oster, S. ; Wübbeke, A. ; Engels, G. ; Schürr, A.: *Model-Based Testing for Embedded Systems.* CRC Press, 2010

[PBKS04] Pohl, K. ; Böckle, G. ; Knauber, P. ; Schmid, K.: *Software-Produktlinien: Methoden, Einführung und Praxis.* dpunkt.Verlag, 2004

[PBL05] Pohl, K. ; Böckle, G. ; Linden, F. J. d.: *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, 2005

[Pic07] Pichler, R.: *Scrum - Agiles Projektmanagement erfolgreich einsetzen.* dpunkt.verlag, 2007

[PKS00] Pol, M. ; Koomen, T. ; Spiller, A.: *Management und Optimierung des Testprozesses.* dpunkt, Heidelberg, 2000

[PM06] Pohl, K. ; Metzger, A.: Software Product Line Testing. In: *Commun. ACM* 49 (2006), Nr. 12, S. 78–81

[Poh08] Pohl, K.: *Requirements Engineering: Grundlagen, Prinzipien, Techniken.* dpunkt.Verlag, 2008

[PP04] Pretschner, A. ; Philipps, J.: Methodological Issues in Model-Based Testing. In: *Model-Based Testing of Reactive Systems*, 2004, S. 281–291

[Pus02] Pussinen, M.: A Survey on Software Product Line Evolution / Institute of Software Systems, Tampere University of Technology. 2002. – Forschungsbericht

[RG99] Richters, M. ; Gogolla, M.: A Metamodel for OCL. In: *Proceedings of the Second International Conference on Unified Modeling Language. Beyond the Standard*, 1999, S. 156–171

[RKPR05a] Reuys, A. ; K., Erik ; Pohl, K. ; Reis, S.: Szenario-basierter Systemtest von Software-Produktfamilien. In: *Inform., Forsch. Entwickl.* 20 (2005), Nr. 1-2, S. 33–44

[RKPR05b] Reuys, A. ; Kamsties, E. ; Pohl, K. ; Reis, S.: Model-Based System Testing of Software Product Families. In: *Proceedings of the International Conference on Advanced Information Systems Engineering*, 2005, S. 519–534

[RMP06] Reis, S. ; Metzger, A. ; Pohl, K.: A Reuse Technique for Performance Testing of Software Product Lines. In: *Proceedings of Third International Workshop on Software Product Line Testing (SPLiT '05)*, 2006, S. 5–10

[RMP07] Reis, S. ; Metzger, A. ; Pohl, K.: Integration Testing in Software Product Line Engineering: A Model-Based Technique. In: *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE '07)*, 2007, S. 321–335

[RP06] Rechenberg, P ; Pomberger, G.: *Informatik Handbuch*. Hanser, 2006

[RR99] Robertson, S. ; Robertson, J.: *Mastering the Requirements Process*. Addison-Wesley, 1999

[RRKP03] Reuys, A. ; Reis, S. ; Kamsties, E. ; Pohl, K.: Derivation of Domain Test Scenarios from Activity Diagrams. In: *Proceedings of*

the International Workshop on Product Line Engineering - The Early Steps - Planning, Modeling and Managing (PLEES '03), 2003, S. 35–41

[Rup09] Rupp, C.: *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis.* Hanser Fachbuch, 2009

[Sch07] Scheidemann, K.: *Verifying Families of System Configurations*, Technical Univerity of Munich, Diss., 2007

[SGB01] Svahnberg, M. ; Gurp, J. van ; Bosch, J.: On the Notion of Variability in Software Product Lines. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, 2001, S. 45–54

[Sie03] Siedersleben, J.: *Softwaretechnik: Praxiswissen für Softwareingenieure.* Hanser, 2003

[SJ03] Schmid, K. ; John, I.: A Practical Approach to Full Life-Cycle Variability Management. In: *International Workshop on Software Variability Management (SVM'03) at International Conference on Software Engineering (ICSE '03)*, 2003, S. 41–46

[SK06] Saeki, M. ; Kaiya, H.: On Relationships Among Models, Meta Models and Ontologies. In: *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006, 140-149

[SL05] Spillner, A. ; Linz, T.: *Basiswissen Softwaretest.* dpunkt.verlag Heidelberg, 2005

[SM09] Suhl, L. ; Mellouli, T.: *Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen.* Springer Berlin, 2009

[Sof] Software Engineering Institute (SEI): *http://www.sei.cmu.edu/productlines/ (Stand: 24.06.2010)*

[Som08] Somé, S. S.: A Meta-Model for Textual Use Case Description. In: *Journal of Object Technologie* 8 (2008), Nr. 7, S. 87–106

[SP07] *Kapitel* Variability Modeling and Product Derivation in E-Business Process Families. In: SCHNIEDERS, A. ; PUHLMANN, F.: *Technologies for Information Systems*. Springer-Verlag Berlin, 2007, S. 63–74

[Sta74] STACHOWIAK, H.: *Allgemeine Modelltheorie*. Springer, Wien, 1974

[UGKB07] UZUNCAOVA, E. ; GARCIA, D. ; KHURSHID, S. ; BATORY, D.: A Specification-based Approach to Testing Software Product Lines. In: *Proceedings of the 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC-FSE companion '07)*, 2007, S. 525–528

[UL07] UTTING, M. ; LEGEARD, B.: *Practical Model-Based Testing. A Tools Approach*. Morgan Kaufmann, 2007

[Wüb08] WÜBBEKE, A.: Towards an Efficient Reuse of Test Cases for Software Product Lines. In: *Proceedings of the 12th International Software Product Line Conference (SPLC '08)*, 2008, S. 361–368

[WO10] WÜBBEKE, A. ; OSTER, S.: Verknüpfung von kombinatorischem Plattform- und individuellem Produkt-Test für Software-Produktlinien. In: *Software Engineering 2010 - Workshopband*, 2010, S. 361–372

[WSS08] WEISSLEDER, S. ; SOKENOU, D. ; SCHLINGLO, B.: Reusing State Machines for Automatic Test Generation in Product Lines. In: *Proceedings of the 1st Workshop on Model-based Testing in Practice (MoTiP2008)*, 2008

Anhang A

Algorithmen für Spezifikationsprozesse mit Variabilität

A.1 Anforderungsspezifikation mit Variabilität

Zum Verständnis sind an die Anweisungen der Algorithmen die Schritte aus Abbildung 7.7 als Kommentare in geschweiften Klammern eingefügt.

Der Algorithmus enthält als Schritt 8 die Erweiterung des Featuremodells um fehlende Features. In diesem Fall werden neue Features in das Featuremodell eingefügt. Für jedes Feature muss das passende Vaterfeature bestimmt werden. Darüber hinaus müssen die Abhängigkeiten zu anderen Features spezifiziert werden, sodass das Features zusammen mit seiner Position im Featuremodell Variabilitätseigenschaften der Variante modelliert.

Algorithmus 2 Additionspunktspezifikation in Anforderungen

Vorbedingung: Featuremodell ist initial spezifiziert und das Abbildungsmodell ist initialisiert

1: Erstelle einen neuen Variationspunkt vp_x mit x als eindeutigem Variationspunktindex {Schritt 1}
2: **repeat**
3: Erstelle neue Varianten v_y von vp_x mit y als eindeutigem Index jeder Variante {Schritt 2}
4: **until** Keine weitere Variante notwendig
5: **for all** $v_y \in vp_x$ **do**
6: **if** $\exists BK_z \in$ *Abbildungsmodell* mit passender *Featurebedingung* **then**
7: Erweitere die Variante um die *Bindungskonfiguration*: $v_y|BK_z$ {Schritt 4}
8: **else**
9: Erzeuge neue *Abbildungsimplikation* mit BK_s und s als eindeutiger Index im *Abbildungsmodell* {Schritt 3}
10: Erweitere die Variante um die *Bindungskonfiguration*: $v_y|BK_s$ {Schritt 4}
11: **for all** *Features* $f_i \in$ *Featuremodell* **do**
12: **if** f_i abstrahiert Variabilitätseigenschaften von v_y **then**
13: Füge f_i zur *Featurebedingung* hinzu {Schritt 5}
14: **end if**
15: **end for**
16: **for all** $f_j \in$ Menge fehlender *Features* $\in$ *Featuremodell* **do** {Schritt 8}
17: Bestimme Vaterfeature $f_k \in$ *Featuremodell*
18: Füge f_j als Kind-Feature zu f_k hinzu
19: Spezifiziere Abhängigkeiten von f_j zu anderen Features
20: Füge f_j zur *Featurebedingung* hinzu
21: **end for**
22: **end if**
23: Notiere den Variationspunkt vp_x mit Variante v_y in der Form $vp_x(v_y)$ unter *Varianten* in der *Abbildungsimplikation*. {Schritt 6}
24: Spezifiziere den Inhalt der Variante v_y {Schritt 7}
25: **end for**

Nachbedingung: Alle Varianten des Variationspunktes vp_x und ihre Abhängigkeiten sind spezifiziert

Algorithmus 3 Modifikationspunktspezifikation in Anforderungen

Vorbedingung: Featuremodell ist initial definiert und Abbildungsmodell ist initialisiert

1: Erstelle einen neuen Variationspunkt vp_x mit x als eindeutigem Variationspunktindex {Schritt 1}
2: **if** $\exists BK_z \in$ Abbildungsmodell mit passender *Wertefunktion* **then**
3: Erweitere den *Modifikationspunkt* um die *Bindungskonfiguration* mit *Wertefunktion*: $vp_x|f_w(BK_z)$ {Schritt 4}
4: **else**
5: Erzeuge neue *Abbildungsbeeinflussung* mit BK_s und s als eindeutiger Index im *Abbildungsmodell* {Schritt 2}
6: Spezifiziere *Wertefunktion* mit Attribut t aus Feature f {Schritt 3}
7: Erweitere den *Modifikationspunkt* um die *Bindungskonfiguration* mit *Wertefunktion*: $vp_x|f_w(BK_s)$ {Schritt 4}
8: **end if**
9: Notiere den Variationspunkt vp_x in der *Modifikationspunktmenge* in der Abbildungsbeeinflussung. {Schritt 5}

Nachbedingung: Modifikationspunkt vp_x, Wertefunktion und die Abhängigkeiten sind spezifiziert

A.2 Spezifikation von logischen Testfällen

Die Position des Variationspunktes im Ablauf der Anwendungsfallbeschreibung ist für die Spezifikation von logischen Testfällen von Bedeutung. Die relevanten Positionen der Variationspunkte und ihre Auswirkungen auf die Spezifikation wurden in Abbildung 8.10 beschrieben.

In dieser Arbeit wird beispielhaft eine Schrittüberdeckung für Anwendungsfallbeschreibungen gewählt. Die Algorithmen 4 und 5 beschreiben das Vorgehen für die Spezifikation von Variabilität während der Spezifikation von Testfall und Testschritten durch den Testdesigner.

Algorithmus 4 betrachtet schrittweise alle Abläufe im Anwendungsfall. Ein passendes Überdeckungskriterium für Abläufe kann in [MH03] gefunden werden. Zu Beginn werden alle Schritte der Anwendungsfallbeschreibung als nicht besucht markiert.
Die eigentliche Spezifikation der Testschritte geschieht in Algorithmus 5.
Schritte mit internen Akteuren werden bei der Spezifikation von Testschritten nicht berücksichtigt, da sie nicht durch den Testtreiber durchgeführt werden können. Für jeden Schritt werden die darin vorhandenen Variationspunkte mit ihren Varianten betrachtet und Testschritte bzw. Variationspunkte mit Testschrittvarianten erstellt. Jeder verarbeitete Schritt wird anschließend als besucht markiert.
Das Besuchen eines bereits als besucht markierten Schrittes führt zum Abbruch der Spezifikation des betrachteten Testfalls. Ein Abbruch kann dann sinnvoll sein, wenn sich im Ablauf der Anwendungsfallbeschreibung Schleifen befinden, die unter Umständen nicht terminieren. Die in diesem Beispiel implementierte Markierung der bereits besuchten Schritte ist eine Möglichkeit einen Abbruch herbeizuführen. Je nach Anforderungen muss eine andere Strategie definiert werden.

Algorithmus 4 Spezifikation von logischen Testfällen mit Variabilität

Vorbedingung: Testbasis tb mit Anwendungsfallbeschreibung $anwf$

1: **for all** Schritte $s \in anwf$ **do**
2: besucht = $false$
3: **end for**
4: **if** $anwf$ ist Variante **then**
5: Erstelle vp_x an tb
6: **end if**
7: **for all** Ablauf $a \in anwf$ **do**
8: **if** $anwf$ ist Variante **then**
9: **if** $NormalerAblauf \in a$ ist Variante **then**
10: Erstelle neue Abbildungsimplikation ai mit mit Featurebedingung $fb_{anwf} \wedge fb_{NormalerAblauf}$
11: Assoziiere neuen Testfall tc an ai
12: **else**
13: Assoziiere neuen Testfall tc mit Featurebedingung fb_{anwf} an vp_x
14: **end if**
15: **else**
16: **if** $NormalerAblauf \in a$ ist Variante **then**
17: Erstelle vp_x an tb
18: Assoziiere neuen Testfall tc mit Featurebedingung $fb_{NormalerAblauf}$ an vp_x
19: **else**
20: Erstelle neuen Testfall tc
21: **end if**
22: **end if**
23: $\rightarrow SVN$:Spezifikation Vor- und Nachbedingung
24: **for all** Schritt $s \in a$ **and** abbruch = $false$ **do**
25: **if** s.besucht = $false$ **then**
26: $\rightarrow STS$:Spezifikation von Testschritten
27: **else**
28: abbruch = $true$
29: **end if**
30: **end for**
31: **end for**

Nachbedingung: Alle logischen Testfälle für die Anwendungsfallbeschreibung sind erstellt.

Algorithmus 5 STS: Spezifikation von Testschritten

1: **if** s ist erster Schritt einer Variante „Alternativ-“ (alt) oder „Ausnahme-Szenario“ (aus) **then**
2: **if** tc ist Variante **then**
3: Erweitere fb_{tc} um $\wedge\{fb_{alt}, fb_{aus}\}$
4: **else**
5: Definiere vp_x an tb
6: Assoziiere tc and vp_x mit $\{fb_{alt}, fb_{aus}\}$
7: **end if**
8: **end if**
9: **if** s ist Variationspunkt vp_y **then**
10: Erstelle vp_x an tc
11: **for all** Variante $v \in vp_y$ **do**
12: **if** s hat externen Akteur **then**
13: Erstelle Testschrittvariante tsv
14: Assoziiere tsv als Variante mit Featurebedingung fb_v an vp_x
15: **for all** Modifikationspunkt $mp_s \in v$ **do**
16: Erstelle mp_{tsv} in tsv mit Abbildungsbeeinflussung $abfl_{mp_v}$
17: **end for**
18: **end if**
19: **end for**
20: s.besucht = $true$
21: **else**
22: **if** s hat externen Akteur **then**
23: Erstelle Testschritt ts an tc
24: **for all** Modifikationspunkt $mp_s \in s$ **do**
25: Erstelle mp_{ts} in ts mit Abbildungsbeeinflussung $abfl_{mp_s}$
26: **end for**
27: s.besucht = $true$
28: **end if**
29: **end if**
30: $\rightarrow SEA$:Spezifikation Ein- und Ausgabeparameter
31: $\rightarrow SVN$:Spezifikation Vor- und Nachbedingung

Algorithmus 6 SVN: Spezifikation von Vor- und Nachbedingungen

1: **for all** Verfeinerung $ve \in \{anwf, s\}$ mit Typ $Vor-$ oder $Nachbedingung$ **do**
2: **if** ve ist Variationspunkt **then**
3: Erstelle vp_x an $\{tc, ts\}$
4: **for all** Variante $v \in ve$ **do**
5: Erstelle Vor- (vb) oder Nachbedingung (nb)
6: Assoziiere $\{vb, nb\}$ als Variante mit Featurebedingung fb_v an vp_x
7: **for all** Modifikationspunkt $mp_v \in v$ **do**
8: Erstelle mp_b in $\{vb, nb\}$ mit Abbildungsbeeinflussung $abfl_{mp_v}$
9: **end for**
10: **end for**
11: **else**
12: Erstelle Vor- (vb) oder Nachbedingung (nb) an $\{tc, ts\}$
13: **for all** Modifikationspunkt $mp \in ve$ **do**
14: Erstelle mp_b in $\{vb, nb\}$ mit Abbildungsbeeinflussung $abfl_{mp}$
15: **end for**
16: **end if**
17: **end for**

Bei der Spezifikation von Vor- und Nachbedingungen (Algorithmus 6) werden alle an den Anwendungsfall bzw. Schritt assoziierten Verfeinerungen von Typ „Vor- und Nachbedingung“ berücksichtigt. Falls eine solche Verfeinerung Additions- und Modifikationspunkte enthält, werden diese auch im Testfall berücksichtigt.

Algorithmus 7 SVN: Spezifikation von Ein- und Ausgabeparametern

1: **for all** Verfeinerung $ve \in s$ mit Typ *Ein−* oder *Ausgabeparameter* **do**
2: **if** ve ist Variationspunkt **then**
3: Erstelle vp_x an ts
4: **for all** Variante $v \in ve$ **do**
5: Erstelle Ein-(ep) oder Ausgabeparameter(ap)
6: → SPB:Spezifikation Parameterbestandteile
7: Assoziiere {ep, ap} als Variante mit Featurebedingung fb_v an vp_x
8: **for all** Modifikationspunkt $mp_v \in v$ **do**
9: Erstelle mp_p in {ep, ap} mit Abbildungsbeeinflussung $abfl_{mp_v}$
10: **end for**
11: **end for**
12: **else**
13: Erstelle Ein-(ep) oder Ausgabeparameter(ap) an ts
14: **for all** Modifikationspunkt $mp \in ve$ **do**
15: Erstelle mp_p in {ep, ap} mit Abbildungsbeeinflussung $abfl_{mp}$
16: **end for**
17: **end if**
18: **end for**

Die Spezifikation von Ein- und Ausgabeparametern (Algorithmus 7) erfolgt analog zur Spezifikation von Vor- und Nachbedingungen. Die dabei relevanten Verfeinerungen tragen den Typ „Ein-“ bzw. „Ausgabeparameter“. Jeder dabei vorkommende Additions- bzw. Modifikationspunkt wird bei der Spezifikation der Ein- und Ausgabeparameter an einen Testschritt berücksichtigt. Während der Spezifikation eines Ein- bzw. Ausgabeparameters wird die Spezifikation der Parameterbestandteile aufgerufen (Algorithmen 8 und 9.

Algorithmus 8 SPB: Spezifikation von Parameterbestandteilen Teil 1

1: **for all** Verfeinerung $ve \in \{ep, ap\}$ mit Typ $Pflichtbedingung$ **do**
2: **if** ve ist Variationspunkt **then**
3: Erstelle vp_x an $\{ep, ap\}$
4: **for all** Variante $v \in ve$ **do**
5: Erstelle Pflichtbedingung (pb)
6: Assoziiere pb als Variante mit Featurebedingung fb_v an vp_x
7: **for all** Modifikationspunkt $mp_v \in v$ **do**
8: Erstelle mp_p in pb mit Abbildungsbeeinflussung $abfl_{mp_v}$
9: **end for**
10: **end for**
11: **else**
12: Erstelle Pflichtbedingung (pb) an $\{ep, ap\}$
13: **for all** Modifikationspunkt $mp \in ve$ **do**
14: Erstelle mp_p in pb mit Abbildungsbeeinflussung $abfl_{mp}$
15: **end for**
16: **end if**
17: **end for**
18: **for all** Verfeinerung $ve \in \{ep, ap\}$ mit Typ $Format$ **do**
19: **if** ve ist Variationspunkt **then**
20: Erstelle vp_x an $\{ep, ap\}$
21: **for all** Variante $v \in ve$ **do**
22: Erstelle Format (f)
23: Assoziiere f als Variante mit Featurebedingung fb_v an vp_x
24: **for all** Modifikationspunkt $mp_v \in v$ **do**
25: Erstelle mp_p in f mit Abbildungsbeeinflussung $abfl_{mp_v}$
26: **end for**
27: **end for**
28: **else**
29: Erstelle Format (f) an $\{ep, ap\}$
30: **for all** Modifikationspunkt $mp \in ve$ **do**
31: Erstelle mp_p in f mit Abbildungsbeeinflussung $abfl_{mp}$
32: **end for**
33: **end if**
34: **end for**

Algorithmus 9 SPB: Spezifikation von Parameterbestandteilen Teil 2

1: **for all** Verfeinerung $ve \in \{ep, ap\}$ mit Typ $Wertebereich$ **do**
2: **if** ve ist Variationspunkt **then**
3: Erstelle vp_x an $\{ep, ap\}$
4: **for all** Variante $v \in ve$ **do**
5: Erstelle Wertebereich (wb)
6: Assoziiere wb als Variante mit Featurebedingung fb_v an vp_x
7: **for all** Modifikationspunkt $mp_v \in v$ **do**
8: Erstelle mp_p in wb mit Abbildungsbeeinflussung $abfl_{mp_v}$
9: **end for**
10: **end for**
11: **else**
12: Erstelle Wertebereich (wb) an $\{ep, ap\}$
13: **for all** Modifikationspunkt $mp \in ve$ **do**
14: Erstelle mp_p in wb mit Abbildungsbeeinflussung $abfl_{mp}$
15: **end for**
16: **end if**
17: **end for**
18: **for all** Verfeinerung $ve \in \{ep, ap\}$ mit Typ Typ **do**
19: **if** ve ist Variationspunkt **then**
20: Erstelle vp_x an $\{ep, ap\}$
21: **for all** Variante $v \in ve$ **do**
22: Erstelle Typ (typ)
23: Assoziiere typ als Variante mit Featurebedingung fb_v an vp_x
24: **for all** Modifikationspunkt $mp_v \in v$ **do**
25: Erstelle mp_p in typ mit Abbildungsbeeinflussung $abfl_{mp_v}$
26: **end for**
27: **end for**
28: **else**
29: Erstelle Typ (typ) an $\{ep, ap\}$
30: **for all** Modifikationspunkt $mp \in ve$ **do**
31: Erstelle mp_p in typ mit Abbildungsbeeinflussung $abfl_{mp}$
32: **end for**
33: **end if**
34: **end for**

A.3 Spezifikation von konkreten Testfällen

Algorithmus 10 SKT: Spezifikation von konkreten Testfällen

1: **if** $\exists p$ mit vp_x (Parameter mit Additionspunkt) **then**
2: Erstelle vp_y an tc
3: **end if**
4: **for all** Parameterkombination pk der Überdeckungsstrategie **do**
5: Erstelle Testdatensatz td für pk
6: **if** $\exists p \in pk$ mit Additionspunkt **then**
7: Definiere Abbildungsimplikation ai mit Featurebedingung $fb_{td} = fb_{p_1} \wedge fb_{p_2} ... \wedge fb_{p_n}$
8: Assoziiere td an vp_y und ai
9: **end if**
10: $\rightarrow \ddot{U}NP$:Überprüfung der Notwendigkeits- und Pflichtbedingungen
11: **if** td nicht gelöscht **then**
12: $\rightarrow STD$ Spezifikation Testdatum für Testdatensatz
13: **end if**
14: **end for**

Die Spezifikation von konkreten Testfällen (Algorithmus 10) wird durch die Spezifikation von Testdatensätzen realisiert. Dazu wird ein Überdeckungskriterium genutzt, welches die Variabilität in Parametern berücksichtigt. Jeder Testdatensatz, der einen variablen Parameter enthält, ist von der Auswahl dieser Parameter für Produkte abhängig.
Für jede erstellte Parameterkombination muss überprüft werden, ob Notwendigkeits- und Pflichbedingungen verletzt werden (Algorithmus 11). Diese Verletzungen werden korrigiert und der Testdatensatz ggf. verworfen, falls die Korrektur in einem bereits existierenden oder verworfenen Testdatensatz resultiert.

Für alle nach diesem Schritt noch vorhandenen Testdatensätze werden anschließend Testdaten erstellt (Algorithmus 12). Dabei wird für jede Kombination an Varianten und Modifikationspunkten ein Testdatum definiert, welches von der Konjunktion der jeweiligen Featurebedingungen bzw. Abbildungsbeeinflussungen abhängt.

Algorithmus 11 ÜNP: Überprüfung der Notwendigskeits- und Pflichtbedingungen

1: **for all** Notwendigkeitsbedingung nb ist verletzt **do**
2: Korrigiere verletzenden Parameter (Löschen oder Hinzufügen) in Testdatensatz tds
3: **end for**
4: **if** $tds \in$ Menge der Testdatensätze **or** bereits verworfen **then**
5: Verwerfe td
6: **end if**
7: **for all** Pflichtbedingung pb ist verletzt und negiert keine Notwendigkeitsbedingung **do**
8: **if** pb ist Variationspunkt **then**
9: **if** tds ist Variante an Variationspunkt **then**
10: Erweitere Featurebedingung von tds mit $\wedge\neg fb_{pb}$
11: **else**
12: Erstelle neuen Variationspunkt vp_x an Testfall tc
13: Erstelle neue Abbildungsimplikation ai mit Featurebedingung $\neg fb_{pb}$
14: Assoziiere tds an vp_x und ai
15: **end if**
16: **else**
17: Korrigiere verletzenden Parameter (Löschen oder Hinzufügen) in Testdatensatz td
18: **end if**
19: **end for**
20: **if** $tds \in$ Menge der Testdatensätze **or** bereits verworfen **then**
21: Verwerfe td
22: **end if**

Algorithmus 12 STD: Spezifikation eines Testdatums

1: Bestimme Äquivalenzklassen für Parameter p
2: Wähle Äquivalenzklasse unter Berücksichtigung der Notwendigkeitsbedingungen

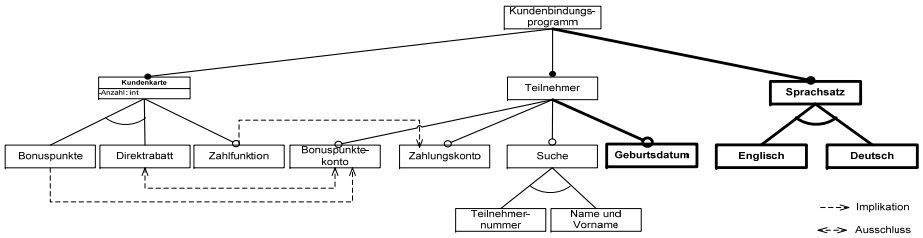
3: **if** Additionspunkt in *Typ*, *Format* oder *Wertebereich* von p **then**
4: **if** Modifikationspunkt mp in *Typ*, *Format* oder *Wertebereich* von p **then**
5: Erstelle vp_x in Testdatensatz tds
6: **for all** Variantenkombination vk in Parameterbestandteil pb **do**
7: Erstelle Abbildungsimplikation ai mit Featurebedingung $fb_{td} = fb_{pb_1} \wedge fb_{pb_2} ... \wedge fb_{pb_n}$
8: Assoziierte Testdatum td an vp_x und ai
9: Erstelle Abbildungsbeeinflussung ab mit Wertefunktion basierend auf $\forall mp \in pb$
10: Erstelle Modifikationspunkt mp_x in td
11: Assoziiere mp_x mit ab
12: **end for**
13: **end if**
14: **else**
15: **if** Modifikationspunkt mp in *Typ*, *Format* oder *Wertebereich* von p **then**
16: Erstelle Testdatum td in Testdatensatz tds
17: Erstelle Abbildungsbeeinflussung ab mit Wertefunktion basierend auf $\forall mp \in pb$
18: Erstelle Modifikationspunkt mp_x in td
19: Assoziiere mp_x mit ab
20: **else**
21: Erstelle Testdatum td in Testdatensatz tds
22: **end if**
23: **end if**

Anhang B

Laufendes Beispiel aus dem Testfallspezifikationsprozess

B.1 Analyse der Testbasis

Um das Beispiel aus Abbildung 8.7 konsistent mit dem Feature- und Abbildungsmodell aus Abbildung 7.9 zu halten, wird das Featuremodell um die Features Sprachsatz mit den Sub-Features Deutsch und Englisch, sowie Geburtsdatum erweitert und alle Abhängigkeiten zu den im Prototyp der Benutzerschnittstelle vorhandenen Varianten über das Abbildungsmodell spezifiziert. Die neuen Features sowie Elemente des Abbildungsmodells sind in Abbildung B.1 hervorgehoben.



Abbildungsimplikation:

Bindungs-konfiguration	Featurebedingung	Bestandteile
BK_1	Suche	$vp_1(v_1)$, $vp_6(v_1)$
BK_2	Zahlfunktion ∧ Zahlungskonto	$vp_2(v_1)$, $vp_3(v_1)$, $vp_5(v_1)$, $vp_6(v_1)$, $vp_9(v_1)$
BK_3	Teilnehmernummer	$vp_4(v_1)$, $vp_{11}(v_1)$
BK_4	Name und Vorname	$vp_4(v_2)$, $vp_{11}(v_2)$
BK_5	**Geburtsdatum**	**$vp_{24}(v_1)$**
BK_6	**Englisch**	**$vp_{25}(v_1)$**
BK_7	**Deutsch**	**$vp_{25}(v_2)$**

Abbildungsbeeinflussung:

Bindungs-konfiguration	Wertefunktion	Modifikationspunktmenge
BK_{10}	Kundenkarte.Anzahl	vp_7, vp_{10}

Abbildung B.1: Featuremodell und Abbildungsmodell mit Berücksichtigung der zusätzlichen Artefakte der Testbasis

B.2 Spezifikation von logischen Testfällen

Verbliebene Pfade aus der Anwendungsfallbeschreibung (vgl. Abbildung 8.12):

$1 \rightarrow 1.1 \rightarrow vp_4 \rightarrow 1.4 \rightarrow 1.5 \rightarrow 1.6$
$1 \rightarrow 1.1 \rightarrow vp_4 \rightarrow 1.3$
$1 \rightarrow 1.1 \rightarrow vp_4 \rightarrow 1.4 \rightarrow 1.5 \rightarrow 1.6 \rightarrow 2$

In Abbildung B.2 sind die übrigen drei Testfälle für die Anwendungsfallbeschreibung „UC_01_Teilnehmer_suchen“ abgebildet. Das dazugehörige Feature- und Abbildungsmodell ist in Abbildung B.3 dargestellt. Die im Rahmen der Testfallspezifikation hinzugefügten Variationspunkte sowie die Abbildungsimplikation sind hervorgehoben.

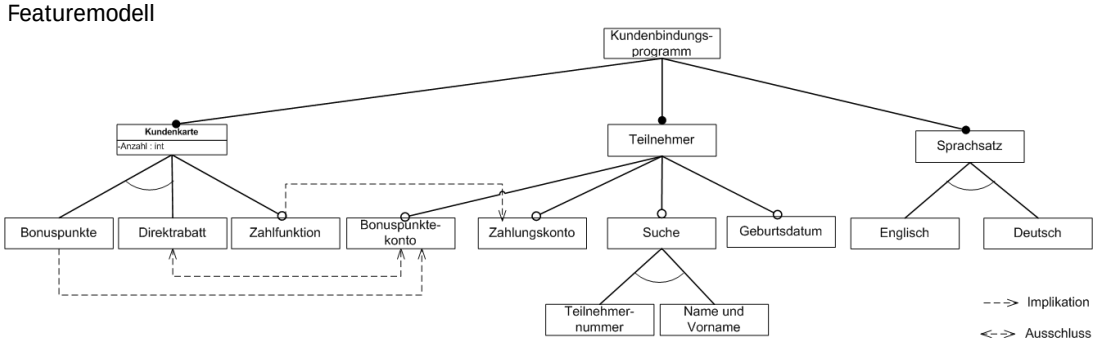
<vp$_{12}$><v$_1$|BK$_1$>	
Testfallname	TC_01:Teilnehmer_erfolgreich_suchen
Nr.	**Testschritte**
1	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
<vp$_{13}$><v$_1$|BK$_3$>	
2	Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.
</v$_1$> <v$_2$|BK$_4$>	
2	Der SCA sucht den Teilnehmer anhand des Namens .
</v$_2$></vp$_{13}$>	
3	Der SCA wählt einen Teilnehmer aus der Liste der Teilnehmer aus
</v$_1$></vp$_{12}$>	

<vp$_{14}$><v$_1$|BK$_1$>	
Testfallname	TC_02:Teilnehmer_erfolglos_suchen
Nr.	**Testschritte**
1	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
<vp$_{15}$><v$_1$|BK$_3$>	
2	Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.
</v$_1$> <v$_2$|BK$_4$>	
2	Der SCA sucht den Teilnehmer anhand des Namens .
</v$_2$></vp$_{15}$>	
</v$_1$></vp$_4$>	

<vp$_{16}$><v$_1$|BK$_1$>	
Testfallname	TC_03:Teilnehmer_erfolgreich_suchen_Stammdaten ändern
Nr.	**Testschritte**
1	Der SCA wählt die Maske zur Suche und Pflege von Teilnehmern aus.
<vp$_{17}$><v$_1$|BK$_3$>	
2	Der SCA sucht den Teilnehmer anhand der Teilnehmernummer.
</v$_1$> <v$_2$|BK$_4$>	
2	Der SCA sucht den Teilnehmer anhand des Namens .
</v$_2$></vp$_{17}$>	
3	Der SCA wählt einen Teilnehmer aus der Liste der Teilnehmer aus
4	Der SCA ändert die Stammdaten des Teilnehmers
</v$_1$></vp$_6$>	

Abbildung B.2: Testschritte der logischen Testfälle zum UC_01_Teilnehmer_suchen

Featuremodell



Abbildungsimplikation:

Bindungskonfiguration	Featurebedingung	Bestandteile
BK_1	Suche	$vp_1(v_1)$, $vp_6(v_1)$, $\mathbf{vp_{12}(v_1)}$, $\mathbf{vp_{14}(v_1)}$, $\mathbf{vp_{16}(v_1)}$
BK_2	Zahlfunktion ∧ Zahlungskonto	$vp_2(v_1)$, $vp_3(v_1)$, $vp_5(v_1)$, $vp_6(v_1)$, $vp_9(v_1)$
BK_3	Teilnehmernummer	$vp_4(v_1)$, $vp_{11}(v_1)$, $\mathbf{vp_{13}(v_1)}$, $\mathbf{vp_{15}(v_1)}$, $\mathbf{vp_{17}(v_1)}$, $\mathbf{vp_{19}(v_1)}$
BK_4	Name und Vorname	$vp_4(v_2)$, $vp_{11}(v_2)$, $\mathbf{vp_{13}(v_2)}$, $\mathbf{vp_{15}(v_2)}$, $\mathbf{vp_{17}(v_2)}$, $\mathbf{vp_{19}(v_2)}$
BK_5	Geburtsdatum	$vp_{24}(v_1)$
BK_6	Englisch	$vp_{25}(v_1)$
BK_7	Deutsch	$vp_{25}(v_2)$
BK_8	**Suche ∧ Zahlfunktion ∧ Zahlungskonto**	**$vp_{18}(v_1)$**

Abbildungsbeeinflussung:

Bindungskonfiguration	Wertefunktion	Modifikationspunktmenge
BK_{10}	Kundenkarte.Anzahl	$vp_{7,}$ vp_{10}

Abbildung B.3: Abbildungsmodell nach Spezifikation der logischen Testfälle zu UC-_01_Teilnehmer_suchen

B.3 Technikmodell mit Variabilität

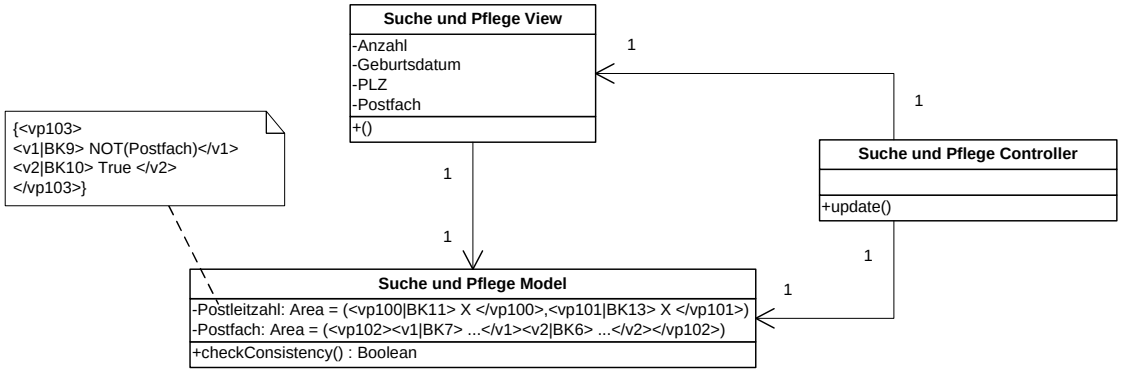


Abbildung B.4: Technikmodell: Klassendiagramm für die GUI Spezifikation nach MVC-Muster

Abbildungsverzeichnis

Tabellenverzeichnis