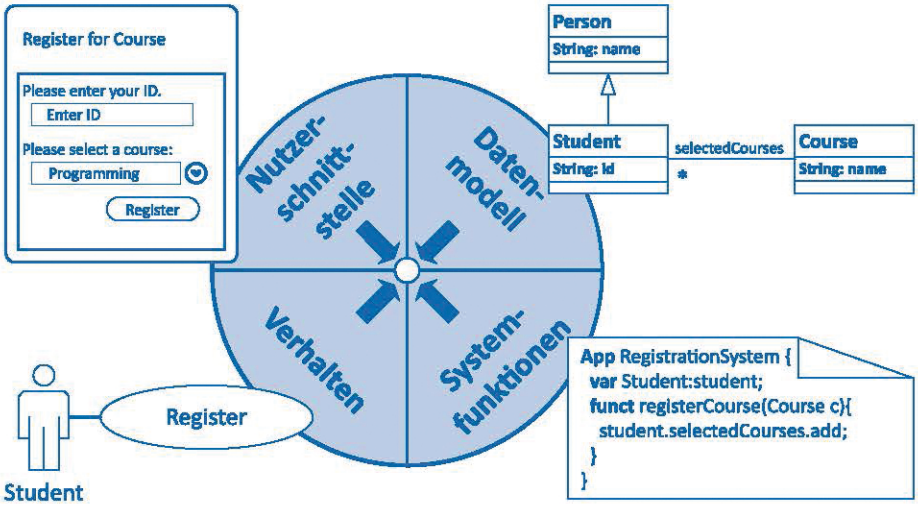
Veit Hoffmann

Rapid Prototyping in der Use-Case-zentrierten Anforderungsanalyse



Aachener Informatik-Berichte,
Software Engineering Band 15

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe
Prof. Dr. rer. nat. Horst Lichter

Rapid Prototyping in der Use-Case-zentrierten Anforderungsanalyse

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. Veit Hoffmann
aus Aachen

Berichter: Prof. Dr. rer. nat. Horst Lichter
Prof. Dr. Sc. Tech. Dirk Riehle M.B.A.

Tag der mündlichen Prüfung: 11. Juli 2013

Diese Dissertation ist auf den Internetseiten der
Hochschulbibliothek online verfügbar.

[Hof13] V. Hoffmann:
Rapid Prototyping in der Use-Case-zentrierten Anforderungsanalyse.
Shaker Verlag, ISBN 978-3-8440-2291-9. Aachener Informatik-Berichte, Software Engineering, Band 15. 2013.
www.se-rwth.de/publications/

D 82 (Diss. RWTH Aachen University, 2013)

Nancy gewidmet

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 15

Veit Hoffmann

Rapid Prototyping in der Use-Case-zentrierten Anforderungsanalyse

Shaker Verlag
Aachen 2013

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: D 82 (Diss. RWTH Aachen University, 2013)

Printed in Germany.

ISBN 978-3-8440-2291-9
ISSN 1869-9170

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 95 96 - 0 • Telefax: 02407 / 95 96 - 9
Internet: www.shaker.de • E-Mail: info@shaker.de

Kurzdarstellung

Die Anforderungsanalyse ist einer der zentralen Erfolgsfaktoren bei der Durchführung von Softwareentwicklungsprojekten. Zur Analyse der Anforderungen an interaktive Informationssysteme hat sich in den letzten Jahrzehnten die Use-Case-zentrierte Analyse etabliert. Hierbei bildet ein Use Case Modell das zentrale Artefakt des gesamten Anforderungsentwicklungsprozesses. Alle anderen Anforderungsartefakte werden mit diesem Use Case Modell verbunden.

Verschiedene Studien zeigen, dass Prototyping deutlich zur Verbesserung der Ergebnisse von Anforderungsanalysen beitragen kann. Die frühe Verfügbarkeit eines ausführbaren Modells des Zielsystems vereinfacht die Kommunikation aller Projektbeteiligten und die Validierung von Anforderungen. Außerdem verbessert Prototyping die Gebrauchsqualitäten der Software und reduziert Akzeptanzprobleme. Dennoch werden die Potentiale von Prototypingtechniken in der Use-Case-zentrierten Anforderungsanalyse derzeit nicht ausreichend ausgeschöpft. Gründe hierfür sind die unzureichende Einbettung der verwendeten Modellierungsnotationen, das Fehlen eines durchgängigen methodischen Vorgehens und eine ungeeignete Werkzeugunterstützung.

Die in dieser Arbeit entwickelten Notationen, Methoden und Werkzeuge zielen darauf ab die Lücke zwischen der Use-Case-zentrierten Anforderungsanalyse und dem Prototyping zu schließen. Kern der Lösung ist SUPrA, ein modellbasierter Prototypingprozess. Hier werden die Anforderungen auf Basis von UCSM, einer Use-Case-zentrierten Anforderungsspezifikationsnotation, spezifiziert. UCSM besitzt eine formal definierte Semantik. Diese ermöglicht es aus UCSM Modellen automatisch verschiedene Arten von Nutzerschnittstellenprototypen zu generieren. SUPrA ist in eine durchgängige Werkzeugumgebung eingebettet. OpenUMF unterstützt die Spezifikation von UCSM Modellen, die Generierung von Prototypen sowie deren Bewertung. Dies ermöglicht es Prototypen prozessbegleitend einzusetzen, da der manuelle Anpassungsaufwand der Prototypen bei Änderungen der Spezifikation entfällt.

Weiterhin wird ProDUCE, ein methodisches Vorgehen, das Prototypingaktivitäten in die Use-Case-zentrierte Analyse integriert, vorgestellt. Hier werden Use-Case-zentrierte Spezifikationen in mehreren Phasen iterativ und inkrementell entwickelt. Dabei wird Prototyping sowohl analytisch als auch konstruktiv zur Qualitätsverbesserung der Analyse eingesetzt.

Die Anwendbarkeit und Skalierbarkeit der entwickelten Ansätze wurde in verschiedenen industriellen Projekten und in Forschungsprojekten demonstriert.

Danksagung

Diese Dissertation währe ohne die Hilfe vieler Kollegen und Freunde nicht möglich gewesen. Hiermit bedanke ich mich herzlich bei allen, die mich während der spannenden Phase meiner Promotion unterstützt haben.

Mein erster Dank gilt Prof. Dr. Horst Lichter für die konstruktive inhaltliche Begleitung dieser Arbeit. Vielen Dank, dass Sie immer ein offenes Ohr für Ideen und Probleme hatten. Prof. Dr. Dirk Riehle danke ich für die bereitwillige Übernahme des Zweitgutachtens.

Die langjährige Zusammenarbeit mit der KISTERS AG lieferte viele Ideen für diese Arbeit sowie Teile der praktischen Evaluierung. Mein Dank gilt Dr. Heinz-Josef Schlebusch stellvertretend für alle Mitarbeiter, mit denen ich zusammenarbeiten konnte. Ich danke auch Klaus Kisters für die Unterstützung des Kooperationsprojekts.

Wichtige Beiträge zu verschiedenen Forschungsfragen und dem Werkzeug OpenUMF wurden in einer Reihe von Diplom- und Masterarbeiten bearbeitet. In solchen Arbeiten lernt man voneinader und es bilden sich Freundschaften. Ich danke Philipp Fonteyn, Nizar Jeribi, Thomas Klapdor, Bora Kiliclar, Jan Krause, Mark Lehmacher, Robert Morys, Michael Mussil, Holger Naussed, Eduard Renz, Tobias Vößing, Daniel Watermeyer und Christoph Weidmann. Besonders danke ich Andreas Walter, dessen Abschlussarbeit die Grundlage für das Metamodell UCSM gelegt hat.

Meinen Kollegen Ana Dragomir, Andreas Ganser, Simona Jeners, Bärbel Kronewetter, Dr. Alexander Nyßen, Chayakorn Piyabunditkul, Tanya Sattaya-aphitan, Dr. Holger Schackmann, Matthias Vianden und Marion Zinner danke ich für die konstruktive Zusammenarbeit und freundschaftliche Athmosphäre. Die Disskussionen mit Euch und die vielen provokativen Fragen waren eine wichtige Unterstützung.

Besonders danke ich meiner Famile. Nancy, Vera, Mama und Georg halfen nicht nur als Lektoren beim Polieren der Sprache in dieser Dissertation; meine Familie ist in meinem Leben die zentrale Stütze. Besonders danke ich meiner Freundin Nancy für ihre Liebe und ihre Unterstützung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Software bildet heutzutage in fast allen Industriesegmenten die Grundlage für die Realisierung von innovativen Produkten und Diensten. Dabei hat die Komplexität der entwickelten Systeme in den letzten Jahren signifikant zugenommen. Zusätzlich herrschen ein hoher Innovations-, Termin- und Kostendruck.

Verschiedene Studien zeigen, dass die Anforderungsanalyse einer der zentralen Erfolgsfaktoren bei der Softwareentwicklung ist. Die direkt auf Unzulänglichkeiten in der Analyse zurückführbaren Probleme in der Softwareentwicklung schwanken je nach Studie zwischen 65% [Rup07, 10] und 50% [HBR02]. Zusätzlich steigen die Kosten für die Beseitigung von Anforderungsfehlern deutlich an, je später sie im Entwicklungsprozess entdeckt werden [BB01].

Eine strukturierte Anforderungsanalyse ist deshalb essentiell für erfolgreiche Softwareprojekte und das Fundament moderner Software-Entwicklungsprozesse [Poh08].

Das Ziel der Anforderungsanalyse ist es, die Anforderungen, die von den unterschiedlichen Projektbeteiligten an das zu entwickelnde System gestellt werden, zu identifizieren, zu dokumentieren und so zu modellieren, dass eine Realisierung dieser Anforderungen geplant und umgesetzt werden kann [Dum03, 36ff.].

Die Wahl geeigneter Modellierungstechniken und eines geeigneten Vorgehens sind hier zwei zentrale Erfolgsfaktoren. Die verwendeten Notationen müssen die fachlichen Belange des Zielsystems adäquat abdecken können und die Diskussion verschiedener Projektbeteiligter über das System ermöglichen. Abgesehen von diesen rein fachlichen Aspekten müssen im Spannungsfeld der Anforderungsanalyse verschiedene weitere menschliche und unternehmenspolitische Einflussfaktoren berücksichtigt werden [Som07b].

Bei der Entwicklung von interaktiven Anwendungssystemen hat sich die Use Case Modellierung als Technik für die Anforderungsanalyse etabliert. Empirische Untersuchungen zeigen, dass sich die Use Case Modellierung seit ihrer Einführung in den späten 80er Jahren zu einer der am weitesten verbreiteten Techniken für die Dokumentation von funktionalen Anforderungen entwickelt hat [Nei03, WHL09, Ebe08]. Aufgrund ihrer ablauforientierten Struktur und ihrer semiformalen Natur eignen sich Use Case Modelle gut, um Systeme mit starkem Nutzerbezug zu beschreiben. Zusätzlich unterstützen sie verschiedene Aspekte der Projektplanung wie Priorisierung, Aufwandsschätzung und das Zuteilen von Verantwortlichkeiten. Mittlerweile ist die Use Case Modellierung ein integraler

Bestandteil vieler moderner Softwareentwicklungsprozesse und Use Case Modelle werden im Sinne der Use Case getriebenen Entwicklung [Jac04] als zentrales Steuerungsdokument für den gesamten Entwicklungsprozess verwendet.

Zusätzlich werden heutzutage häufig Prototypen zur Anforderungsanalyse eingesetzt [Som07b]. Prototypen sind für alle Arten von Projektbeteiligten leicht verständlich und können deren Projektbeteiligung (Involvement) deutlich erhöhen [Sut97]. Bei der Entwicklung von interaktiven Anwendungssystemen werden im Sinne des Usage Centered Design [Con09] insbesondere Prototypen der Nutzerschnittstelle verwendet. Die Darstellung von Informationen bildet bei diesen Systemen ein zentrales Akzeptanzkriterium und wird deshalb bereits in der Anforderungsanalyse modelliert.

Verschiedene Studien zeigen, dass Projekte, die Szenariobeschreibungen und Nutzerschnittstellenprototypen kombinieren, signifikant erfolgreicher sind als Vergleichsprojekte [HL01, Hal97].

Dennoch werden die Potentiale, die aus der Kombination von Use Case Modellen und Nutzerschnittstellenprototypen entstehen, nicht optimal genutzt. Derzeit werden Use Case Modelle und Nutzerschnittstellenprototypen weitestgehend unabhängig voneinander entworfen und weiterentwickelt. Die unzureichende Integration der Notationen führt zu unnötig hohen Entwicklungskosten der Anforderungsspezifikation und Inkonsistenzen zwischen den beiden Modellierungsartefakten.

1.1 Ziele der Arbeit

Die in dieser Arbeit vorgestellten Methoden, Notationen und Werkzeuge zielen darauf ab, die Integration von Prototypen mit textuellen Use Case zentrierten Anforderungsspezifikationen zu verbessern. So soll die Gesamtqualität der Anforderungsspezifikation erhöht werden.

Dazu wird ein durchgängiges Metamodell eingeführt, das es ermöglicht, textuelle Use Case Modelle mit Nutzerschnittstellenbeschreibungen zu kombinieren. So können diese Modelle zur wechselseitigen Validierung eingesetzt und in einem interaktiven, inkrementellen Anforderungsentwicklungsprozess leichter konsistent gehalten werden.

Zusätzlich wird ein generativer Ansatz zur Erzeugung von ausführbaren Nutzerschnittstellenprototypen aus diesen Modellen vorgestellt. Diese können zur Analyse der modellierten Anforderungen eingesetzt werden.

Die entwickelten Methoden werden durch eine spezielle Werkzeugumgebung unterstützt, die sowohl Mechanismen zur Spezifikation der verschiedenen Anforderungsartefakte als auch zur Generierung, Ausführung und Analyse der Prototypen enthält.

Diese Arbeit enthält also die folgenden Beiträge zum Gebiet des Software-Engineerings:

- eine metamodell-basierte Notation für die Beschreibung von Use Case zentrierten Anforderungsmodellen,
- einen generativen Ansatz zur Erzeugung von interaktiven Prototypen aus diesen Modellen,
- ein Vorgehen zur Modellierung und Qualitätssicherung der Use Case zentrierten Anforderungsmodelle auf Basis der erzeugten Prototypen.

Die entwickelten Methoden, Modellierungsnotationen und Werkzeuge wurden in mehreren Fallstudien evaluiert.

1.2 Überblick über die Arbeit

Die Arbeit besteht aus insgesamt 14 Kapiteln. Nach der Einleitung in Kapitel 1 führt Kapitel 2 die relevanten Grundbegriffe ein und stellt die Forschungsgebiete im Umfeld dieser Arbeit vor. In Kapitel 3 wird die Problemstellung dieser Arbeit motiviert und die Ziele dieser Arbeit werden vorgestellt. Verwandte Arbeiten werden anschließend in Kapitel 4 präsentiert. Kapitel 5 gibt eine Zusammenfassung der Anforderungen an einen Prototypingansatz für die Use-Case-zentrierte Analyse. Zum Verständnis der präsentierten Lösungen wird in Kapitel 6 ein Beispielsystem vorgestellt, an dem die in den folgenden Kapiteln vorgestellten Lösungsaspekte erklärt werden. Anschließend werden die einzelnen Aspekte der Lösung diskutiert. Die Spezifikation von Use-Case-zentrierten Modellen, die als Basis des Prototypings geeignet sind, beschreibt Kapitel 7. Kapitel 8 stellt dar, wie aus diesen Modellen Prototypen generiert werden können. Die Bewertung der Prototypen diskutiert anschließend Kapitel 9. Ein methodisches Vorgehen zur Use-Case-zentrierten, Prototyp getriebenen Anforderungsanalyse wird in Kapitel 10 eingeführt. Die zugehörige Werkzeugunterstützung präsentiert Kapitel 11. Eine Bewertung des präsentierten Ansatzes geben Kapitel 12 und 13. Zunächst werden Erfahrungen aus der Praxis vorgestellt. Anschließend werden die Ergebnisse anhand der in Kapitel 5 formulierten Anforderungen bewertet. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 14.

2 Grundlagen

In diesem Kapitel werden die begrifflichen und methodischen Grundlagen dieser Arbeit erläutert. Dazu wird zunächst eine Eingrenzung der betrachteten Softwaresysteme gegeben. Anschließend werden die zentralen Konzepte der Anforderungsanalyse, der Use Case Modellierung und des Prototyping vorgestellt. Die für das Verständnis der Arbeit nötigen Begriffe werden jeweils begleitend eingeführt.

2.1 Softwaresysteme

Grob lässt sich Software in System- und Anwendungssoftware unterteilen. Systemsoftware dient dem Betrieb und der Wartung von Hardware sowie der Schaffung technischer Infrastruktur für Anwendungssoftware. Zur Systemsoftware zählt man Hardwaretreiber und Betriebssysteme sowie Kommunikationsprogramme, Datenbanken und Middleware. Anwendungssoftware hingegen ist Software für die Lösung bzw. Unterstützung bestimmter fachlicher Aufgaben eines Nutzers mit Hilfe eines Computersystems [Bal96, 26ff.].

Bei Anwendungssoftware kann wiederum zwischen Batchsoftware und interaktiver Software unterschieden werden. Batchsoftware führt weitestgehend autark einen definierten Auftrag aus. Interaktive Software hingegen wird während der Ausführung aktiv durch einen Nutzer gesteuert [RG97, 645].

Weiterhin teilt Dumke Anwendungssoftware bezüglich technischer Aspekte und Anwendungsaspekte in fünf verschiedene Systemklassen ein [Dum03, 242ff.]:

- Informations- und Datenbanksysteme
- Funktionsorientierte Systeme und Konstruktionssyteme
- Ereignisgesteuerte Systeme und reaktive Systeme
- Kommunikations- und Interaktionssysteme
- Wissensbasierte Systeme und Entscheidungshilfesysteme

2.1.1 Interaktive Informationssysteme

Unter Informationssystemen versteht man Systeme zur Verarbeitung und effizienten Speicherung von Daten. Sie dienen dazu, bestimmte Informationsbedürfnisse innerhalb einer fachlichen Anwendungsdomäne zu befriedigen. Diese Kategorie umfasst verschiedene Businessapplikationen, z.B. Bank- oder Versicherungssysteme oder Data-Warehousing-Systeme [Dum03, 242ff.].

Der Begriff *interaktives Informationssystem* bzw. auch *interaktives System* bezeichnet solche Informationssysteme, die ein signifikantes Maß an Nutzerinteraktion erfordern. Wasserman & Shewmake definieren den Begriff *interaktives Informationssystem* folgendermaßen:

> „An interactive information system (IIS) provides its users with conversional access to data. Such systems typically comprise a human-computer interface and a set of operations (transactions), many of which include access to or modification of a database.“ [WS90]

Interaktive Informationssysteme helfen fachliche Abläufe zu vereinfachen bzw. zu beschleunigen und dienen der Unterstützung betriebswirtschaftlicher Entscheidungsprozesse. Solche Systeme sind immer Teil eines betriebswirtschaftlichen, sozioökonomischen Gesamtsystems [Krc09]. Deshalb ist allen diesen Systemen gemein, dass Handhabung und Informationspräsentation eine zentrale Rolle spielen. Diese beiden Aspekte „bestimmen wesentliche Merkmale der Gebrauchsqualität und der Software Ergonomie“ [Rec06, 804]. Deshalb müssen sie beim Entwurf von interaktiven Informationssystemen besonders beachtet werden.

2.2 Anforderungsanalyse, Anforderungen & Anforderungsspezifikation

Der Begriff *Anforderungsanalyse* (Requirements Engineering) bezeichnet ein methodisches Vorgehen zur Identifikation und Verwaltung relevanter Rahmenbedingungen an ein Softwareprodukt. Die Anforderungsanalyse besteht aus Tätigkeiten zur Ermittlung, Festlegung, Beschreibung, Analyse und Verabschiedung von Anforderungen (vgl. [Sep90]). Entsprechend unterscheidet Partsch [Par10] drei wichtige Teilaufgaben der Anforderungsanalyse:

- Ermittlung von Anforderungen
- Dokumentation von Anforderungen
- Analyse der Anforderungsbeschreibung

Unter *Anforderungen* (Requirement) versteht man verschiedene Arten von Rahmenbedingungen für die Entwicklung eines Softwaresystems. In den Standards

der IEEE wird der Begriff wie folgt definiert:

> „A well-formed requirement is a statement of system functionality (a capability) that must be met or possessed by a system to satisfy a customer's need or to achieve a customer's objective, and that is qualified by measurable conditions and bounded by constraints.“ [IEE98]

Laut Pohl definieren Anforderungen also „sowohl (1) Wünsche und Ziele von Benutzern als auch (2) Bedingungen und Eigenschaften des zu entwickelnden Systems“ [Poh08, 13]. Dies schließt sowohl Anforderungen in dokumentierter Form als auch nicht dokumentierte Bedingungen und Eigenschaften ein.

Als *Anforderungsspezifikation* bezeichnet man ein Dokument, das die wesentlichen Anforderungen an eine Software und ihre Schnittstellen präzise, vollständig und überprüfbar dokumentiert [LL10, 375]. Dabei sollte die Anforderungsspezifikation eine Reihe grundsätzlicher Qualitäten erfüllen [II98]. Zur Strukturierung von Anforderungsspezifikationen werden verschiedene Templates verwendet, die Struktur und Aufbau des Inhalts festlegen [II98, Vol]. Abhängig von Zielsystem und Entwicklungsprozess kann der Inhalt von Anforderungsspezifikationen aber teilweise deutlich variieren.

2.2.1 Perspektiven der Anforderungsanalyse

Oftmals werden Anforderungen in *funktionale* und *nicht-funktionale Anforderungen* unterteilt.

Alle Anforderungen, die Systemfunktionalität beschreiben, werden als funktionale Anforderungen bezeichnet. Alle anderen Arten von Anforderungen sind nicht-funktionale Anforderungen. Hierzu zählen laut Pohl Qualitätsanforderungen und Rahmenbedingungen an das zu erstellende Produkt oder den Entwicklungsprozess [Poh08, 14 ff.].

Bei der Analyse von funktionalen Anforderungen müssen die vier in Abbildung 2.1 dargestellten Anforderungsperspektiven und ihre Zusammenhänge betrachtet werden [Bal96, 96ff.]:

- **Datenperspektive:** Die Datenperspektive betrachtet die statische Struktur von Daten. Sie enthält Datentypen und deren Attribute sowie Beziehungen zwischen Datentypen. In der objektorientierten Analyse werden dazu typischerweise Klassendiagramme [OMG07] eingesetzt.
- **Funktionsperspektive:** Die Funktionsperspektive beschreibt die Funktionen des Systems, indem sie darstellt, wie Daten manipuliert werden. Hier werden häufig Datenflussdiagramme [DeM78] oder Funktionsbäume eingesetzt.

- **Verhaltensperspektive:** Die Verhaltensperspektive (auch dynamische Perspektive) stellt dar, wie das System auf Stimuli von außen reagiert. Dazu werden Systemzustände, Zustandsübergänge und Systemausgaben beschrieben. Gebräuchliche Modellierungsnotationen sind Zustandsautomaten [HMU07] oder Petrinetze [Pet62] und insbesondere Use Case Modelle.

- **Benutzungsperspektive:** Die Benutzungsperspektive beschreibt das Aussehen und die Ergonomie der Nutzerschnittstelle. Ihr kommt bei Systemen mit graphischer Nutzerschnittstelle und hohem Interaktionsgrad eine besondere Bedeutung zu. Zur Beschreibung der Benutzungsperspektive werden oft skizzenhafte Darstellungen der einzelnen Bildschirmseiten und der Kontrollstrukturen verwendet.

Wichtig ist hier die Beobachtung, dass die einzelnen Perspektiven der Anforderungsanalyse nicht überschneidungsfrei sind. Stattdessen sind sie teilweise stark miteinander verzahnt, und verschiedene Modellierungsaspekte beeinflussen sich gegenseitig.

Weiterhin variiert die Bedeutung der einzelnen Perspektiven von Anwendung zu Anwendung abhängig von der Art des Zielsystems. Bei der Spezifikation von interaktiven Anwendungssystemen kommt i.d.R. insbesondere der Daten- und Benutzungsperspektive eine besondere Bedeutung zu [Dum03].

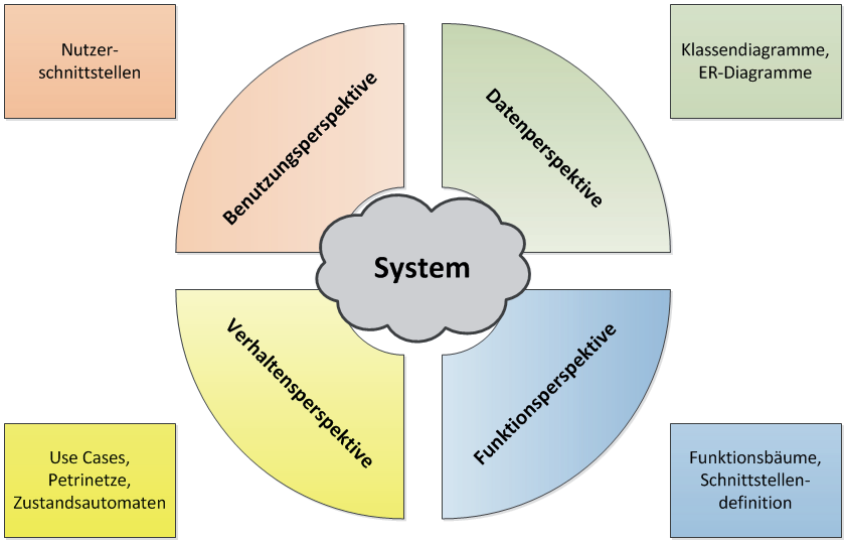


Abbildung 2.1: Anforderungsperspektiven nach Balzert [Bal96]

2.2.2 Vorgehensweisen bei der Anforderungsanalyse

Abhängig vom gewählten Entwicklungsprozess sind die Vorgehensweisen bei der Anforderungsanalyse durchaus unterschiedlich [Par10, 20 ff.]. In klassischen Phasenmodellen, wie dem V-Modell [Drö00], bildet die Anforderungsanalyse normalerweise die erste Phase des Entwicklungsprozesses. Sie endet i.d.R. mit der Abnahme einer Anforderungsspezifikation, die alle wichtigen Anforderungen enthalten soll.

Bei inkrementellen oder evolutionären Vorgehensweisen, wie dem Unified Process [JBR99], wird die Anforderungsanalyse normalerweise prozessbegleitend bis zum Ende der Entwicklung durchgeführt. Dabei werden die Anforderungen, wie alle anderen Entwicklungsartefakte, schrittweise vervollständigt.

Ein ähnliches Vorgehen findet man auch in den meisten agilen Prozessen. Hier werden die Anforderungen typischerweise jeweils vor dem Start eines Entwicklungszyklusses, z.B. eines Sprints in Scrum [PQ10], definiert und anschießend umgesetzt. In agilen Prozessen können sich die Anforderungen während der Projektlaufzeit jederzeit ändern. Deshalb muss auch hier die Anforderungsanalyse kontinuierlich während der gesamten Projektlaufzeit durchgeführt werden.

2.2.3 Rollen in der Anforderungsanalyse

Software wird im Allgemeinen von einem *Softwarehersteller* für einen *Kunden* entwickelt [LL10]. Dies gilt für Individualentwicklung und Produktentwicklung gleichermaßen. In der Produkteentwicklung, in der Software für einen Markt entwickelt wird, übernehemn oft Marketing- oder Salesabteilungen die Rolle des Kunden. Sowohl der Softwarehersteller als auch der Kunde sind meist Firmen oder Unternehmensabteilungen und stellen lediglich juristische Personen dar. Konkret gibt es eine Vielzahl verschiedener Projektbeteiligter, die entweder Mitarbeiter des Softwareherstellers oder des Kunden sind. Diese lassen sich bezüglich ihrer Rolle im Projekt in mehrere Gruppen einteilen. Auf Seiten des Softwareherstellers gibt es verschiedene *Softwareleute*. Dies sind im Wesentlichen *Entwickler* (Analysten, Architekten und Programmierer) und *Manager*. Die Personen, die den Kunden repräsentieren, bezeichnen Ludewig und Lichter als *Klienten*. Hierzu zählen *Fachexperten* und *Nutzer* sowie weitere Menschen, die direkt oder indirekt vom entwickelten System betroffen sind.

2.3 Modell, Modellierung & Metamodell

Modelle spielen in der Softwareentwicklung eine zentrale Rolle. Insbesondere in der Anforderungsanalyse können sie ein wichtiges Hilfsmittel darstellen. Partsch [Par10] identifiziert hier die folgenden Vorteile:

- Modelle ermöglichen Abstraktion.
- Modelle sind i.d.R. leichter zu verstehen als die Realität.
- Modelle fokussieren bestimmte Modellierungsperspektiven.
- Modelle erfassen Expertenwissen.
- Modelle dienen als Diskussionsgrundlage und verbessern so die Kommunikation.

Ein *Modell* ist ein beschränktes Abbild der Wirklichkeit. Es wird auf der Basis von Funktions-, Struktur- oder Verhaltensähnlichkeiten bzw. Verhaltensanalogien zu einem Original speziell für Analysen entwickelt, deren Durchführung am Original nicht möglich oder zu aufwendig wäre [Ali05]. Nach Stachowiak [Sta73, 131ff.] ist es durch drei Merkmale gekennzeichnet:

- **Abbildung:** Ein Modell ist eine Repräsentation eines natürlichen oder künstlichen Originals. Dieses Original kann auch selbst ein Modell sein.
- **Verkürzung:** Ein Modell bildet nicht alle Eigenschaften des Originals ab. Stattdessen stellt es lediglich die Eigenschaften dar, die für die Nutzung des Modells relevant scheinen.
- **Pragmatismus:** Modelle werden immer für einen bestimmten Einsatzzweck entwickelt. Dabei sind sie ziel-, adressaten- und zeitgerichtet. D.h. sie sind ein Abbild einer Sache, das einem bestimmten Anwender zur Beantwortung einer bestimmten Frage innerhalb eines bestimmten Zeitintervalls dienen soll.

Der Prozess, bei dem ein Modell zielgerichtet aus einem Original erzeugt wird, wird als *Modellieren* bezeichnet.

Weiterhin identifiziert Selic [Sel03] fünf zentrale Qualitäten guter Modelle von Softwaresystemen:

- **Abstraktion** (abstraction): Das Verkürzungsmerkmal sagt aus, dass ein Modell immer eine reduzierte Darstellung eines Originals ist. Ein gutes Modell ist dabei ausschließlich auf die relevanten Aspekte fokussiert.
- **Verständlichkeit** (understandability): Ein gutes Modell präsentiert die modellierten Eigenschaften des Originals in einer Form, die für die Anwender des Modells intuitiv verständlich ist. Ein gutes Modell bildet hier eine Abbildung, deren Verständnis weniger intellektuellen Aufwand verlangt als das Original.
- **Präzision** (accuracy): Ein gutes Modell muss eine „lebensechte“ Abbildung der relevanten Aspekte des modellierten Systems anbieten. Es muss sich also in Bezug auf die untersuchten Aspekte wie sein Original verhal-

ten.

- **Vorhersagefähigkeit** (predictiveness): Ein gutes Modell muss dazu genutzt werden können, die interessanten, nicht offensichtlichen Eigenschaften des modellierten Systems, entweder durch Experimente oder durch eine formale Analyse, korrekt vorherzusagen. Diese Eigenschaft hängt offensichtlich von der Präzision des Modells ab.

- **Kosteneffizienz** (inexpensive): Ein gutes Modell muss signifikant billiger zu erzeugen und zu analysieren sein als sein Original.

Der Begriff *Metamodell* besteht aus dem Wort Modell und der griechischen Vorsilbe *meta*. Meta bedeutet dabei soviel wie „über", „nach", „neben" oder „zwischen". Ein Metamodell ist ein Modell, das eine Modellierungssprache definiert [Poh08, 291]. Dabei definiert das Metamodell die abstrakte und die konkrete Syntax sowie die Semantik der Modellierungssprache [CSW08, 31]. Dass ein Modell ein Metamodell ist, ist also keine absolute Eigenschaft des Modells selbst, sondern eine Rolle, die das Modell gegenüber anderen Modellen einnimmt. Dabei stehen Elemente dieser Modelle mit Elementen des Metamodells in einer „ist-Instanz-von"-Beziehung.

Grundsätzlich kann ein Modell auch selbst sein eigenes Metamodell definieren. Diesen Ansatz bezeichnet man als Bootstrapping. Er wird beispielsweise bei der Definition von Meta-Object-Facility (MOF) [OMG11] verwendet, die der UML2 Spezifikation [Obj11] zu Grunde liegt.

2.4 Use Cases

Die Modellierung von Use Cases hat sich in den letzten 20 Jahren zu einer der am weitesten verbreiteten Techniken im Requirements Engineering entwickelt. Die Idee der Use Case Modellierung ist es, das nach außen sichtbare Systemverhalten in Form einer Reihe von „Geschichten" aus Nutzersicht, den sogenannten Use Cases, zu beschreiben. Die Gesamtheit der modellierten Informationen bezeichnet man dabei als *Use Case Modell*:

> „use case model. (1) a model that describes a system's functional requirements in terms of use cases." [Int10]

Im Folgenden werden kurz die zentralen Elemente von Use Case Modellen vorgestellt. Anschließend werden Einsatz und Nutzen von Use Cases diskutiert, bevor verschiedene gängige Notationen für die Use Case Modellierung vorgestellt werden.

2.4.1 Grundbegriffe

Der strukturelle Aufbau von Use Case Modellen ist sehr einfach. Im Wesentlichen werden zwei Arten von Entitäten, nämlich Akteure und Use Cases sowie Beziehungen zwischen diesen modelliert. Im Folgenden werden Akteure, Use Cases und deren Beziehungen kurz eingeführt. Anschließend werden die allgemeinen Konzepte der in Use Cases enthaltenen Ablaufbeschreibungen vorgestellt.

2.4.1.1 Akteur

Allgemein beschreiben Use Cases die Interaktion des modellierten Systems mit verschiedenen Akteuren. Ein Akteur ist dabei eine Rolle, die Nutzer des Systems oder externe Fremdsysteme gegenüber dem modellierten System einnehmen.

> „Actors represent the people or things that interact in some way with the system; by definition, they are outside the system.“ [BS02]

2.4.1.2 Use Case

Eine eindeutige Definition des Begriffs Use Case (Anwendungsfall, Nutzungsfall) zu geben, fällt deutlich schwerer. Der Begriff Use Case wird in der Literatur und Praxis synonym für sehr unterschiedliche Konzepte verwendet. Eingeführt wurde er von Ivar Jacobson bereits im Jahr 1986. In seinem Buch „Object-oriented Software Engineering“ gibt er die folgende Definition an:

> „A use case is a special sequence of transactions, performed by a user and a system in a dialogue. A transaction is performed by either the user or the system and is a response initiated by a stimulus. When no more stimuli can be generated, all the transactions will finally ebb away. The use case is ended and a new use case can be initiated.“ [Jac04]

Bis heute haben sich jedoch viele verschiedene Interpretationen entwickelt, die teilweise sehr stark von Jacobsons initialer Begriffsdefinition und den zugehörigen Konzepten abweichen. So identifizierte Cockburn gut 10 Jahre später bereits mehr als 18 verschiedene Definitionen des Begriffes Use Case [Coc00]. Diese Zahl hat bis heute sicherlich noch zugenommen. Cockburn selbst definiert den Use Case Begriff wie folgt:

> „A use case captures a contract between the stakeholders of a system about its behavior. The use case describes the system's behavior under various conditions as it responds to a request from one of the stakeholders, called the primary actor.“ [Coc00]

Ähnliche zielorientierte Interpretationen werden in vielen anderen Standardwerken zu Use Cases verwendet:

> „Use cases represent the things of value that the system performs for its actors. ...They also have detailed descriptions that are essentially stories about how the actors use the system to do something they consider important, and what the system does to satisfy these needs.“ [BS02]

> „A use case represents a series of interactions between an outside entity and the system, which ends by providing business value.“ [KG03]

Für diese Arbeit wird eine leichte Abwandlung von Cockburns Definition verwendet, um zu betonen, dass Use Cases Systemverhalten in Form eines Dialogs zwischen dem System, Nutzern und Fremdsystemen beschreiben. Wir definieren Use Cases also folgendermaßen:

> „Ein Use Case bezeichnet eine Menge von Interaktionssequenzen, die ein System gemeinsam mit Fremdsystemen und Nutzern durchführen kann, um ein spezielles Ziel zu erreichen und somit einen Mehrwert für einen speziellen Nutzer, den primären Akteur, zu erzeugen. Hierbei beschreibt jeder Use Case sowohl solche Interaktionssequenzen, bei denen das Ziel erreicht wird, als auch solche, bei denen das Ziel nicht erreicht werden kann.“

2.4.1.3 Beziehungen zwischen Use Cases und Akteuren

In Use Case Modellen werden neben den Use Cases und Akteuren auch Beziehungen zwischen diesen beschrieben. Hier haben sich vier verschiedene Beziehungsarten etabliert. Diese sind auch Bestandteil der UML2 Spezifikation [OMG07]:

- **Inklusion**: Die Inklusionsbeziehung ist eine gerichtete Beziehung zwischen zwei Use Cases, einem *inkludierenden Use Case* und einem *inkludierten Use Case*. Stehen zwei Use Cases in einer Inklusionsbeziehung, wird das Verhalten, das im inkludierten Use Case beschrieben ist, an einer definierten Stelle im inkludierenden Use Case eingefügt.

- **Erweiterung:** Bei einer Inklusionsbeziehung wird das Verhalten des inkludierten Use Case in allen Fällen in den inkludierenden Use Case eingefügt. Die Erweiterungsbeziehung hingegen beschreibt eine bedingte Einfügesemantik. Hier wird das Verhalten des *erweiternden Use Case* nur dann in den *erweiterten Use Case* eingefügt, wenn während der Durchführung der Verhaltenssequenz zu einer bestimmten Zeit eine definierte *Erweiterungsbedingung* erfüllt ist. Im Unterschied zur Inklusionsbeziehung wird der *Erweiterungspunkt*, an dem das Verhalten des erweiternden Use Case

in den erweiterten Use Case eingefügt werden kann, i.d.R. explizit gekennzeichnet.

- **Generalisierung:** Die Generalisierungsbeziehung kann sowohl zwischen Akteuren als auch zwischen Use Cases modelliert werden. Eine Generalisierungsbeziehung zwischen zwei Akteuren bedeutet, dass der *spezielle Akteur* eine spezielle Unterrolle des *generellen Akteurs* beschreibt.

  Bei Generalisierungsbeziehungen zwischen Use Cases ist das im *speziellen Use Case* beschriebene Verhalten eine spezielle Variante des im *generellen Use Case* beschriebenen Verhaltens. Wie sich diese Beziehung in der Verhaltensbeschreibung des speziellen Use Cases konkret manifestiert, ist allerdings nicht klar festgelegt.

- **Assoziation:** Eine Assoziation wird immer zwischen einem Use Case und einem Akteur beschrieben. Sie zeigt an, dass der Akteur an der Durchführung des im Use Case beschriebenen Verhaltens beteiligt ist. Manche Notationen unterscheiden zwischen primären und sekundären Akteuren. Die UML2 Spezifikation macht allerdings einen solchen Unterschied nicht.

2.4.1.4 Use Case Beschreibung

Unabhängig von der konkreten Notation sind Use Cases zielorientierte Beschreibungen. Sie modellieren Interaktionssequenzen zwischen Akteuren und dem System, die im Kontext der Erreichung eines bestimmten Ziels eines primären Akteurs auftreten können. Jeder Use Case beschreibt hier eine Menge verschiedener Interaktionssequenzen vom Anfang eines Use Cases bis zu einem bestimmten Ende. Diese Interaktionssequenzen werden als *Szenarien* bezeichnet [Coc00, 42]. Der *Normalablauf* beschreibt hier das einfachste mögliche erfolgreiche Szenario. Er bildet die Ausgangsbasis für alle anderen Verhaltensvarianten, die *Alternativszenarien*.

Use Cases enthalten zwei verschiedene Arten von Alternativszenarien:

Bei *erfolgreichen Szenarien* (Success Szenario) [MB03] wird am Ende der Interaktionssequenz das Ziel des primären Akteurs erreicht. Modelliert also beispielsweise der Use Case „`Geld abheben`" Szenarien, in denen der primäre Akteur „`Kunde`" einen bestimmten Geldbetrag von einem Geldautomaten abheben will, so werden diejenigen Szenarien, in denen der Kunde am Ende der Interaktionssequenz den gewünschten Betrag erhält, als erfolgreiche Szenarien bezeichnet. Alle anderen Szenarien nennt man *Ausnahmeszenarien* (Exception).

Normalerweise werden die Szenarien eines Use Cases nicht einzeln unabhängig voneinander modelliert. Stattdessen enthalten Use Cases den sogenannten *Ereignisfluss* (Flow of Events). Dieser besteht aus *Flüssen* (Flow) [BS02], aus denen die einzelnen Szenarien des Use Cases zusammengesetzt werden können.

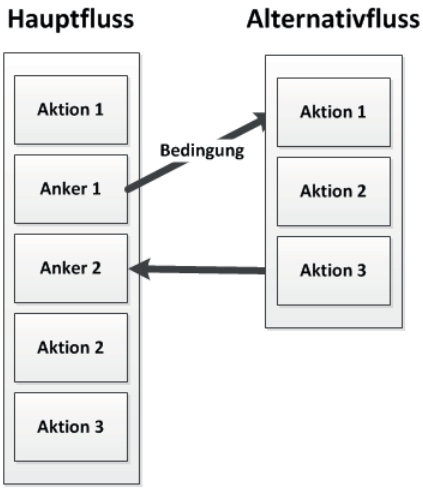


Abbildung 2.2: Ereignisfluss – Schematische Darstellung

Jeder Fluss besteht wiederum aus einer zusammenhängenden Sequenz von atomaren Schritten, den so genannten *Ereignissen* (Events) (vgl Abb. 2.2). Hier können zwei Arten von Ereignissen unterschieden werden:

- *Aktionen* (Action) beschreiben ein diskretes in sich abgeschlossenes Verhalten, das entweder vom System oder von einem der am Use Case beteiligten Akteure durchführt wird. Dabei ist jede Aktion unabhängig von allen anderen Aktionen.

- *Anker* (Anchor) hingegen realisieren Punkte, an denen Flüsse miteinander verbunden werden können. Sie modellieren also Stellen, an denen der Kontrollfluss der Use Cases während der Ausführung von einem Fluss an einen anderen Fluss übergeben wird. Weiterhin werden sie verwendet, um Inklusions- bzw. Erweiterungsbeziehungen in der Verhaltensbeschreibung eines Use Cases zu manifestieren.

Der Ereignisfluss spannt also einen gerichteten Graphen auf, der durch die Verbindung von Flüssen mit Hilfe von Ankern entsteht. Ein Szenario kann nun als Pfad auf dem Ereignisfluss definiert werden [MB03]. Demzufolge enthält der Ereignisfluss alle möglichen Szenarien eines Use Cases. Der Zusammenhang zwischen Ereignisfluss, Szenarien und Flüssen wird in Abbildung 2.3 verdeutlicht. Hier können weiterhin zwei Arten von Flüssen unterschieden werden: Der *Hauptfluss* (basic flow) beschreibt den kompletten Normalablauf. Alle anderen Flüsse werden als *Alternativflüsse* (alternative flows) bezeichnet [Int10]. Sie enthalten nie ein komplettes Szenario. Stattdessen entstehen alle Alternativszenarien aus der Kombination des Hauptflusses mit einem oder mehreren Alternativflüssen.

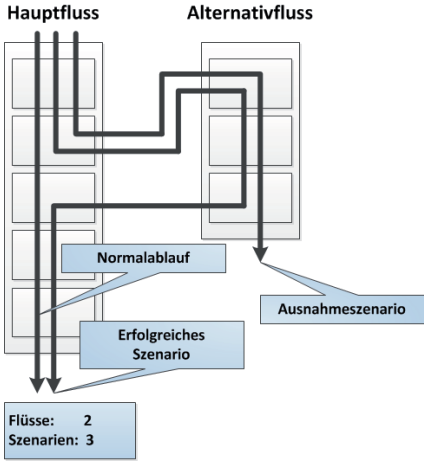


Abbildung 2.3: Ereignisfluss, Flüsse und Szenarien [MB03]

2.4.2 Use Case getriebene Entwicklung

In vielen Softwareprojekten, in denen Use Case Modelle eingesetzt werden, spielen diese in weiten Teilen des Entwicklungsprozesses eine wichtige Rolle [Coc00]. Sie werden, abgesehen von der Anforderungsanalyse, auch in einer Reihe weiterer Modellierungsaktivitäten eingesetzt. Hierzu zählen beispielsweise die Erstellung der Nutzerdokumentation, die Architekturanalyse oder das Testen.

In diesem Zusammenhang hat Jacobson den Begriff *Use Case getriebene Entwicklung* bzw. Use Case Driven Design geprägt [Jac04, 129]. In der Use Case getriebenen Entwicklung bilden Use Case Modelle die zentralen Dokumente zur Koordination des gesamten Entwicklungsprozesses. Alle anderen Artefakte des Softwareentwicklungsprozesses werden auf Basis von Use Case Modellen entwickelt und sind direkt oder indirekt mit ihnen verbunden. Somit können alle Änderungen an der Software immer vom Use Case Modell aus getrieben werden. Diese Idee hat sich in vielen verschiedenen Softwareentwicklungsprozessen etabliert. Erwähnenswert sind hier neben dem Unified Process [JBR99] beispielsweise OEP [OV06], ROPES [Dou99], ICONIX [RS07] oder das V-Modell-XT [RBH+08]

2.4.3 Use Case zentrierte Anforderungsanalyse

Der Begriff *Use Case zentrierte Anforderungsanalyse* beschreibt ein spezielles Vorgehen zur Anforderungsanalyse. In der Use-Case-zentrierten Anforderungsanalyse bilden Use Case Modelle das zentrale Artefakt des gesamten Anforderungsentwicklungsprozesses. Alle anderen Anforderungsartefakte werden, wie in

Abbildung 2.4 dargestellt, mit den Use Cases verbunden.

Abbildung 2.4: Use Cases – Nabe-Speiche Modell

Heutzutage wird in den meisten Lehrbüchern ein inkrementelles Vorgehen bei der Anforderungsanalyse empfohlen. Dieses ist auch in fast allen Use Case getriebenen Prozessen etabliert. Während der Analyse durchlaufen die einzelnen Use Cases mehrere aufeinanderfolgende Modellierungsphasen, die jeweils einen speziellen Modellierungsfokus besitzen. Die Use Cases selbst werden in jeder dieser Phasen bis zu einem bestimmten Detaillierungsgrad spezifiziert. Je nach Autor werden zwischen drei [Coc00, KG03] und sechs [BS02] verschiedene Phasen genannt, die sich teilweise leicht unterscheiden. Unabhängig vom Autor lassen sich jedoch in allen Use-Case-zentrierten Anforderungsanalyseansätzen drei generelle Phasen identifizieren:

- **Produktabgrenzung:** In dieser Phase wird der Umfang der Entwicklung festgelegt. Ziel dieser Phase ist es, Platzhalter für alle wichtigen Interaktionen zwischen Akteuren und dem System zu identifizieren. Hierzu werden *Use Case Kurzbeschreibungen* verwendet. Jede Kurzbeschreibung besitzt einen Namen und fasst das Ziel des Use Cases in einigen wenigen Sätzen zusammen. Sie sind normalerweise nicht weiter strukturiert und geben keine detaillierte Beschreibung des Ablaufes.

- **Konzeptmodellierung:** Ziel dieser Phase ist es, den Normalablauf festzulegen und mögliche Alternativabläufe zu identifizieren. Hierzu werden *Use Case Konzeptskizzen* eingesetzt. Diese enthalten eine kurze Beschreibung der Essenz eines Use Cases. In der Regel beschreiben sie lediglich den Hauptfluss des Use Cases komplett. Wichtige Ausnahmen und Alternativen werden zwar genannt, aber i.d.R. nicht vollständig modelliert. Zusätzlich beschreiben Konzeptskizzen oftmals die Vor- und Nachbedin-

gungen eines Use Cases.

- **Detailmodellierung:** In der Detailmodellierung werden die Details der einzelnen Abläufe beschrieben und in ihren Kontext gesetzt. Ziel ist die Definition eines vollständigen, konsistenten Anforderungsmodells der modellierten Applikation. Hierzu werden *Use Case Detailbeschreibungen* verwendet. Jede Use Case Detailbeschreibung modelliert den Normalablauf eines Use Cases sowie alle Alternativen und Ausnahmen. In der Regel wird hier der Kontrollfluss des Use Cases in Form einer strukturierten Beschreibung dargestellt. Detaillierte Use Case Beschreibungen bilden das Ende der Use Case Modellierung. Sie dienen als Vorlage für detaillierte technische Analysen, für den Softwaretest, die Nutzerdokumentation und weitere Aktivitäten im Softwareentwicklungsprozess. [KG03].

Wichtig ist hier anzumerken, dass viele der vorgeschlagenen Use Case getriebenen Anforderungsanalyseansätze Teil einer übergeordneten iterativen Softwareentwicklungsmethode sind. Das heißt, dass die Use-Case-zentrierte Anforderungsanalyse mehrfach durchlaufen wird (vgl. Abb. 2.5). Oftmals wird dabei in den ersten beiden Phasen ein breiterer Ausschnitt der Problemwelt betrachtet als in der Detailmodellierungsphase. Deshalb enthalten Use Case Modelle i.d.R. gleichzeitig Use Cases auf mehreren verschiedenen Detaillierungsstufen.

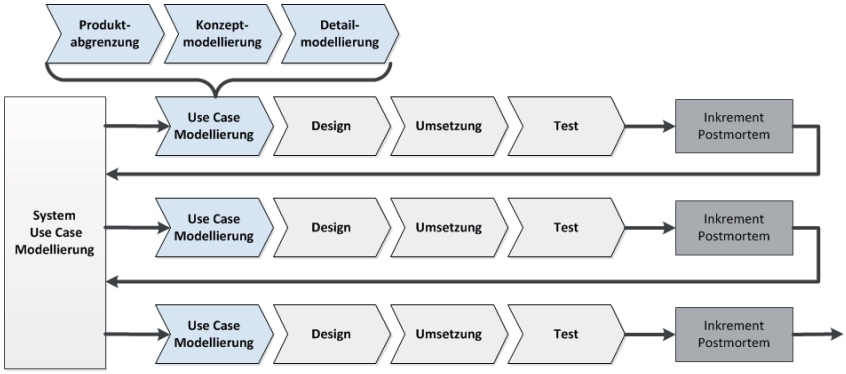


Abbildung 2.5: Use Case Entwicklungszyklus (adaptiert von Armour [AM01])

2.4.4 Notationen für Use Case Modelle

Wie Use Cases in der Praxis dokumentiert werden, variiert sehr stark. Abhängig von der Art des durchgeführten Projekts, des geplanten Einsatzes der Use Case Modelle, dem angestrebten Detaillierungsgrad der Beschreibungen und persönlichen Präferenzen des Entwicklungsteams können sehr unterschiedliche Formate gewählt werden. Eine empirische Studie, die Weidmann, Hoffmann & Lichter im Jahr 2009 [WHL09] durchgeführt haben, zeigt jedoch, dass bis heute deutlich mehr textorientierte Notationen (ca. 70%) als Diagramme (ca. 30%)

eingesetzt werden und dass UML Use Case Übersichtsdiagramme in Projekten, die Use Cases verwenden, sehr verbreitet sind. Hierbei sei jedoch bemerkt, dass in der Praxis oftmals verschiedene graphische und textuelle Notationen in einem Projekt parallel oder nacheinander verwendet werden. Im Folgenden werden die gängigsten Use Case Notationen vorgestellt und Vor- und Nachteile diskutiert.

2.4.4.1 Textuelle Notationen

Unter textuellen Notationen versteht man Formate, die Use Case Modelle entweder in reiner Textform oder in einem Tabellenformat beschreiben. Dabei unterscheiden sich verschiedene textuelle Notationen deutlich in der Darstellungsform der modellierten Informationen.

Unstrukturierte textuelle Use Cases: Bei unstrukturierten textuellen Use Cases wird das Verhalten in einem zusammenhängenden Text beschrieben, der alle relevanten Informationen enthält.

Diese Darstellungsform eignet sich sehr gut für eine knappe, initiale Beschreibung eines Use Cases in frühen Phasen der Anforderungsanalyse. Sie sind auch für nicht-technische Projektbeteiligte leicht zu lesen, bieten uneingeschränkte Flexibilität bei der Beschreibung und ermöglichen eine kompakte Darstellung abstrakter Zusammenhänge. Deshalb werden sie in der Literatur empfohlen, um in der Produktabgrenzung den Problem-Scope abzustecken. Für eine detaillierte Beschreibung des Systemverhaltens inklusive verschiedener Alternativszenarien und variierender Vor- und Nachbedingungen sind unstrukturierte Use Cases allerdings ungeeignet. Das beschriebene Verhalten wird schon für wenige verschiedene Interaktionsszenarien unübersichtlich, und einzelne Abläufe sind schwer zu identifizieren. Deshalb fällt eine konsistente Beschreibung sowie die Qualitätssicherung von komplexen unstrukturierten Use Cases schwer.

Use Case Templates: Use-Case-Template-basierte Notationen sind in der Literatur mit Abstand am weitesten verbreitet. Sie sind Bestandteil aller modernen Use Case Entwicklungsmethoden. Use Case Templates definieren ein festes tabellarisches Format, das Art und Form der in einem Use Case zu beschreibenden Informationen festlegt. Template-basierte Notationen schließen sowohl einfache tabellenorientierte Templates, wie sie z.B. Cockburn [Coc00], Bittner und Spence [BS02] oder andere [AM04, Got03, CL01] vorschlagen, als auch strukturierte Use Case Metamodelle [HLNW09, SS11, Wil01, LJ08, NJJ+96, Som09, NUOT01] ein, da diese sich lediglich bezüglich ihres Formalisierungsgrades (vgl. Kap. 2.4.5) unterscheiden.

Alle gängigen Use Case Templates bestehen aus zwei Teilen. Der erste Teil beschreibt verschiedene Übersichtsinformationen, welche sich je nach Template teilweise deutlich unterscheiden. Gängige Informationen sind z.B. Trigger, Vor-

und Nachbedingungen oder Ziele. Aber auch exotischere Informationen, wie z.B. Profile potentieller Angreifer in Misuse Cases (vgl. [SO01]), werden modelliert.

Der zweite Teil des Use Case Templates enthält die Beschreibung des Ereignisflusses. Hierbei wird typischerweise, wie in Abbildung 2.6 dargestellt, der Normalablauf in einer ausgezeichneten Sektion als Folge von nummerierten Schritten beschrieben. Zusätzlich werden Alternativabläufe in einer oder mehreren zusätzlichen Sektionen modelliert. Dabei wird i.d.R. lediglich der Teil des Alternativverhaltens beschrieben, der vom Normalverhalten abweicht. Die Nummerierung der Schritte in Alternativabläufen und die Art, wie Beziehungen zwischen Alternativen dargestellt werden, unterscheidet sich von Template zu Template.

Name	<Prägnanter Bezeichner>	
Ziel	<Welches Ziel soll erreicht werden?>	
Einordnung	<Übergeordnete Funktion, Hauptfunktion, Basisfunktion>	
Vorbedingung	<Welcher Zustand muss vor Ablauf des Use Cases gelten?>	
Nachbedingung	<Welcher Zustand tritt ein, wenn der Use Case erfolgreich durchgeführt werden konnte?>	
Nachbedingung im Fehlerfall	<Welcher Zustand tritt ein, wenn das Ziel nicht erreicht werden konnte?>	
Haupt-Neben-Akteure	<Wer sind die wichtigen Akteure?>. <Welche Akteure sind sonst noch involviert?>	
Auslöser	<Welches Ereignis führt zum Start des Use Case?>	
Normalablauf	**Schritt**	**Aktion**
	1	<Auflistung aller Aktionen vom Start des Use Cases bis zum Erreichen des Ziels>
	2	<...>
	3	
Alternativabläufe	**Schritt**	**Aktion**
	1a	<Bedingung für eine Verzweigung> : <Aktionen der Verzweigung>

Abbildung 2.6: Use Case Template - Beispiel

Use Case Templates sind gut geeignet, um komplizierte Zusammenhänge verschiedener Alternativabläufe innerhalb eines Use Cases übersichtlich darzustellen, da sie die einzelnen Interaktionsfragmente eines Use Cases explizit modellieren. Die standardisierte Form der Use Case Darstellung ermöglicht es, zum einen methodische Unterstützung für die unterschiedlichen Informationen in den einzelnen Template-Feldern anzubieten, zum anderen erleichtert sie die Prüfung der modellierten Informationen besonders im Hinblick auf Vollständigkeit.

Auf der anderen Seite schränkt die fixe Struktur von Use Case Templates die Modellierungsflexibilität deutlich ein, und der Mehraufwand für die Beschreibung diverser Übersichtsinformationen ist speziell für Use Case Skizzen zu groß. Außerdem verleitet das schrittorientierte Beschreibungsformat zu einer starken Fragmentierung der einzelnen Abläufe im Use Case, was die Analyse des Verhaltens insbesondere für nicht-technische Projektbeteiligte erschwert.

Use Case Templates können grundsätzlich für alle Detaillierungsgrade eingesetzt werden. Dabei sind spezielle Use Case Templates jedoch nur für bestimmte Detaillierungsgrade geeignet. Deshalb empfiehlt Cockburn, während der Use Case Entwicklung mehrere verschiedene Templates für unterschiedliche Detaillierungsgrade einzusetzen [Coc00].

Use Case Metamodelle haben deutliche Vorteile gegenüber reinen Tabellen-Templates, da es aufgrund der formalisierten Struktur der Metamodelle leichter möglich ist, automatische Qualitätsanalysen durchzuführen. Auf diesen Aspekt wird in Kapitel 2.4.5 näher eingegangen.

Syntaktische Anforderungsmuster: Syntaktische Anforderungsmuster legen eine definierte Textstruktur, i.d.R. mit Hilfe von Satzmustern, für die in Use Cases modellierten Informationen fest [Poh08, 245]. In der Praxis werden sie oftmals mit Use Case Templates kombiniert. Syntaktische Anforderungsmuster definieren also, wie die textuelle Struktur der modellierten Informationen auszusehen hat. Dabei legen sie ausschließlich die textuelle Struktur der modellierten Informationen fest, machen aber keine Annahmen zum verwendeten Vokabular. Syntaktische Anforderungsmuster, wie z.B. Crews-Savre [Mai98] oder [RBA02, Sch02], sind relativ generisch und können in unterschiedlichen Anwendungsdomänen eingesetzt werden. Ihr größter Vorteil ist, dass sie konstruktive Regeln zur strukturierten Beschreibung von Anforderungen, wie z.B. das Sophist REgelwerk [GKKL10], formalisieren und so die Qualität der beschriebenen Anforderungen erhöhen. Allerdings schränken sie die Modellierungsmöglichkeiten sehr stark ein, so dass bestimmte Zusammenhänge nicht mehr wie gewünscht darstellbar sind.

Domänenspezifische Sprachen: Domänenspezifische Sprachen (oder Normsprachen [Poh08, 246]), wie z.B. [PMP11] oder [ZD09], schränken zusätzlich zur Satzstruktur auch das verwendbare Vokabular ein. Da das verwendbare Vokabular normalerweise Konzepte aus der Anwendungsdomäne enthält, ermöglichen sie eine sehr genaue Beschreibung von Verhalten innerhalb einer bestimmten Domäne. Durch das vorgegebene Vokabular ist aber die Ausdrucksmächtigkeit der domänenspezifischen Sprache auf die Anwendungsdomäne beschränkt. Weiterhin sind viele domänenspezifische Sprachen nicht als Kommunikationsmittel mit Klienten geeignet, da ihr Verständnis nur mit Kenntnis der Modellierungsregeln und des verwendeten Domänenvokabulars möglich ist.

Formale Sprachen: Bei formalen Sprachen handelt es sich nicht um textuelle Notationen im engeren Sinne. Stattdessen sind sie textuelle Darstellungen mathematischer Formalismen. Hier werden eine Reihe verschiedener formaler Sprachen zur Modellierung von Use Cases vorgeschlagen. Zu diesen Formalismen zählen z.B. Z [KC08, GL00, SS00, vdPKS+03], Pro-Cases [Men04b, Men04a], aussagenlogische Konstrukte [LLH01] oder die Modellierungssprache LOTOS

[Tuo98]. Allen diesen Notationen ist gemein, dass sie eine definierte, mathematisch fundierte Ausführungssemantik haben. Deshalb sind sehr genaue automatisierte Analysen des beschriebenen Verhaltens möglich (vgl. 2.4.5). Andererseits ist ihre Erstellung sehr aufwendig, und sie sind als Kommunikationsmittel nur in sehr wenigen Projekten einsetzbar, da sie nur für Experten verständlich sind.

2.4.4.2 Graphische Notationen

Graphische Notationen definieren im Unterschied zu textuellen Notationen neben der Semantik der Modellierungselemente eine graphische Darstellung der Konzepte. Im Unterschied zu anderen Domänen gibt es für die Beschreibung von Use Case Verhalten keine spezielle Notation. Im Folgenden werden gängige graphische Notationen zur Beschreibung von Verhalten und ihr Einsatz zur Use Case Modellierung vorgestellt.

Use Case Maps: Use Case Maps (UCM) sind eine graphische Notation für die Beschreibung der Zusammenhänge verschiedener Verhaltensfragmente eines komplexen Systems [Buh98]. Dabei liegt der Fokus auf einer Beschreibung der Zusammenhänge grober funktionaler Blöcke und nicht auf der Beschreibung von Verhaltensdetails.

> „The notation is not a behavior specification technique in the ordinary sense, but a notation for helping a person to visualize, think about, and explain the overall behavior of a complex system. The focus is on the big picture, not on details.“ [Buh98]

Use Case Maps basieren auf einer einfachen graphischen Notation, die Szenarien in Form von sogenannten *kausalen Pfaden* (causal path) beschreibt. Diese Pfade stellen dar, wie bestimmtes Verhalten durch eine Folge kausaler Beziehungen zwischen Verantwortlichkeiten (responsibilites) von Komponenten (components) realisiert wird. Komponenten sind generische Platzhalter für Entitäten. Sie können sowohl Systemteile als auch externe Akteure beschreiben. Verantwortlichkeiten repräsentieren in UCM abstrakte Aufgaben, die zur Umsetzung eines bestimmten Szenarios durchgeführt werden müssen. Verantwortlichkeiten sind dabei unabhängig von Komponenten definiert. Deshalb ist es möglich, sogenannte unbound-Maps zu beschreiben, in denen Folgen von Verantwortlichkeiten lediglich definiert werden, ohne sie Komponenten zuzuweisen. Weiterhin enthalten UCM verschiedene Möglichkeiten, um Laufzeitkonstrukte durch dynamische Stubs und dynamische Verantwortlichkeiten zu beschreiben. Außerdem enthalten sie Sprachelemente für die Beschreibung zeitbehafteten Verhaltens, wie Timeouts oder Timed-Triggers.

Buhr [Buh98] empfiehlt zur Formalisierung von Use Cases unbound-Maps, die auch in verschiedenen anderen Publikationen zu diesem Zweck eingesetzt werden [AL00]. UCM erlauben eine sehr genaue formalisierte Spezifikation des Use

Case Verhaltens. Außerdem lässt sich mit UCM gut die Verteilung von Use Case Verhalten auf die Komponenten des Zielsystems darstellen. Andererseits ist die Notation für viele Arten von Projektbeteiligten zu komplex. UCM sind insbesondere dann unübersichtlich, wenn viele verschiedene Szenarien auf wenige Systemkomponenten abgebildet werden.

Message Sequence Charts: Message Sequence Charts (MSC) sind eine durch die ITU-T [Int04] standardisierte Notation für die Beschreibung sequentieller Abläufe. Diese werden in MSC durch den Nachrichtenaustausch zwischen kommunizierenden Systemkomponenten modelliert. Basis MSC (bMSC) stellen dabei die kommunizierenden Komponenten durch vertikale Linien und die gesendeten Nachrichten durch horizontale Pfeile zwischen den Komponenten dar. Jeder MSC wird hierbei durch eine Folge sogenannter Events vollständig charakterisiert. Diese Events können lokale Aktionen, verschiedene Arten von Nachrichten-Events und einige weitere Arten von Events sein (vgl.[Int98]).

Message Sequence Charts eignen sich besonders zur Beschreibung detaillierter linearer Abläufe innerhalb eines Use Cases und zur Verdeutlichung der Akteure. Deshalb sind sie für Essential Use Cases [Con95] relativ verbreitet. Sie sind jedoch weniger gut geeignet, um Use Cases mit einer großen Zahl an Alternativen oder mit zustandsbehaftetem Verhalten zu modellieren, da sie schon für wenige Fallunterscheidungen unübersichtlich werden.

UML2 Sequenzdiagramme: UML2 Sequenzdiagramme können als eine spezielle Variante von MSC gesehen werden. Sie werden typischerweise zur Modellierung von objekt-orientierten Systemen eingesetzt. Sie modellieren dementsprechend die Kommunikation zwischen Objekten (vgl. [Obj11]). Neben den Standardkonzepten von MSC enthalten sie zusätzliche Modellierungskonzepte, z.B. zur Beschreibung von Schachtelungen oder erweiterten Kontrollstrukturen wie Switches. Sie sind eines der neun Verhaltensdiagramme der UML2 und sind als Format zur Beschreibung von Use Case Details verbreitet [RKW95, GF03, Li99]. Allerdings weisen sie die gleichen Probleme bei der Darstellung von Fallunterscheidungen wie MSC auf.

Life Sequence Charts: Life Sequence Charts (LSC) wurden von Damm und Harel [DH01] eingeführt, um einige Nachteile von MSC auszugleichen. Sie bieten Möglichkeiten an, um zwischen obligatorischem und optionalem Verhalten zu unterscheiden. Damit wird es möglich, zwischen Szenarien, die passieren müssen, Szenarien, die passieren können und Szenarien, die nicht passieren dürfen, zu unterscheiden [Has08]. Außerdem unterscheiden sie Bedingungen, die entweder erfüllt sein müssen (hot condition) oder erfüllt sein können (cold condition). Weiterhin definieren sie sogenannte *pre-charts*, spezielle Aktivierungsszenarien, die dazu führen, dass ein zugehöriges Szenario ausgeführt werden muss. Diese

Definition wird von Harel und Marelly [HM03] um weitere Konzepte wie Variablen, Zuweisungen und Schleifen erweitert.

Mit LCS ist eine genauere Spezifikation des modellierten Verhaltens als mit MSC möglich. Insbesondere können auch die Zusammenhänge und Rollen verschiedener Szenarien eines Systems beschrieben werden. Hierfür ist aber ein deutlich erhöhter Modellierungsaufwand nötig. Die Stärke von LSC liegt in der automatischen Analysierbarkeit der Modelle. Andererseits sind sie aufgrund ihrer Komplexität für nicht-technische Projektbeteiligte nur schwer verständlich und sind deshalb aus unserer Sicht in vielen Projekten als primäre Modellierungsnotation ungeeignet. Zusätzlich leiden LSC unter denselben Skalierungsproblemen wie MSC.

Zustandsautomaten: Zustandsautomaten (Finite State Maschines) dienen zur Beschreibung des Verhaltens von Systemen. Ihnen liegt ein mathematischer Formalismus zu Grunde, der um eine graphische Notation angereichert ist. Zustandsautomaten bestehen aus Zuständen, Zustandsübergängen und Aktionen. Sie sind eine verbreitete Notation zur Beschreibung von Use Cases [KP05, SPW07, SCK09]. In der Regel werden hier die Ereignisse eines Use Case Modells als Zustände modelliert. Die Transitionen stellen den möglichen Ereignisfluss dar. Diese Art der Darstellung ermöglicht eine kompakte Darstellung des beschriebenen Verhaltens. Allerdings sind Zustandsautomaten nicht für automatisierte Analysen des Gesamtverhaltens eines Use Case Modells geeignet, weil sie nicht in der Lage sind, kontextabhängiges Verhalten exakt darzustellen. Diese Einschränkung wird in Kapitel 8.1.1 noch einmal erörtert.

UML Statemachines: UML Statemachines sind eine von der OMG standardisierte Notation für Harel-Automaten [Har87]. Sie bieten zusätzlich zu Zuständen und Übergängen einige weitere Modellierungsmöglichkeiten, z.B. zur Schachtelung von Zuständen. Sie werden von verschiedenen Autoren zur Modellierung von Use Case Verhalten vorgeschlagen [GF03, HKP05]. Allerdings gelten auch für UML Statemachines dieselben Einschränkungen hinsichtlich der Aussagemächtigkeit wie bei allgemeinen Zustandsautomaten.

Use Case Charts: Use Case Charts sind eine spezielle graphische Notation zur formalisierten Use Case Beschreibung [WJ06, JW07]. Sie kombinieren hierarchische Zustandsautomaten mit UML2 Sequenzdiagrammen. Vorteil dieser Notation ist, dass sie es erlaubt, einzelne in Sequenzdiagrammen beschriebene Szenarien zu einer konsistenten Sicht des Gesamtsystemverhaltens zu kombinieren. Haupteinsatzzweck von Use Case Charts ist die Simulation des in den Use Cases beschriebenen Verhaltens. Deshalb müssen Use Case Charts sehr detailliert modelliert werden. Somit sind sie als primäre Notation in vielen Projekten ungeeignet, da sie für viele Arten von Projektbeteiligten unverständlich sind.

Petrinetze: Petrinetze wurden 1962 von Carl Adam Petri als Modellierungssprache für diskrete, nebenläufige Systeme entwickelt. Sie sind im Unterschied zu vielen anderen graphischen Notationen zur Verhaltensmodellierung streng mathematisch fundiert und ermöglichen deshalb eine Reihe mächtiger, qualitativer und quantitativer Analysen. Petrinetze bestehen aus Stellen und Transitionen, die einen bipartiten Graph bilden. Die Semantik von Petrinetzen wird im Kapitel 8.1.2 noch einmal ausführlich diskutiert. Neben den von Petri vorgestellten Standardpetrinetzen gibt es eine Reihe erweiterter Formalismen, wie z.B. Prädikat/Transitions-Netze oder hierarchische Netze. Alle vorgestellten Arten von Petrinetzen sind zur direkten Beschreibung des Verhaltens von Use Cases ungeeignet. Sowohl das zu Grunde liegende mathematische Modell, als auch die zugehörigen graphischen Notationen sind zu abstrakt, um Use Cases kompakt zu modellieren. Dennoch werden sie in mehreren verschiedenen Ansätzen als Formalisierungstechnik [XH07, LMCS98, Som07a] verwendet, um vollautomatisch bestimmte Eigenschaften, wie interne Konsistenz oder Terminierung, von Use Cases zu bestimmen.

Flussdiagramme: Flussdiagramme dienen zur Beschreibung von Kontroll- und teilweise auch von Datenfluss in einem System. Sie bestehen aus Aktionen, die mit gerichteten Kanten verbunden sind. Jede dieser Kanten zeigt einen Kontroll- bzw. einen Datenfluss von einer Aktion zu einer anderen an. Werden Flussdiagramme zur Beschreibung von Use Cases eingesetzt, so beschreiben die Aktionen des Diagramms die Ereignisse des Use Cases und die Kanten den Ereignisfluss zwischen diesen Ereignissen. Ansätze, die UML2 Aktivitätsdiagramme zur Ablaufbeschreibung verwenden, finden sich in [GF03, YBL10]. Zhang [ZXZ01] beschreibt eine Flussdiagrammnotation, die nicht auf der UML2 basiert.

Flussdiagramme sind sehr gut geeignet, die Konzepte flussorientierter Use Case Beschreibungen (vgl. 2.4.1.4) abzubilden. Außerdem sind Flussdiagramme für viele Arten von Projektbeteiligten intuitiv verständlich. Deshalb sind sie im Unterschied zu vielen anderen Notationen als Kommunikationsmittel gut geeignet. Andererseits haben sie, verglichen mit z.B. Sequenzdiagrammen, Schwächen bei der Darstellung der interagierenden Komponenten.

2.4.4.3 Diskussion

Aus der Diskussion der Vor- und Nachteile der einzelnen Use Case Notationen in den vorhergehenden Abschnitten wird klar, dass es keine ideale Modellierungsnotation für Use Cases gibt. Abhängig von der Art des Projektes, dem Detaillierungsgrad und den beteiligten Personen sind die verschiedenen Modellierungsnotationen unterschiedlich gut geeignet. Flexible Notationen, wie unstrukturierte textuelle Use Cases oder einfache Flussdiagramme, sind in frühen Phasen gut geeignet um Use Case Kurzbeschreibungen zu modellieren. Ihnen fehlt aber die notwendige Formalität und Strukturiertheit für die Modellierung

von komplexen Use Case Detailbeschreibungen. Andererseits sind formal definierte Notationen wie DSL oder LSC nicht für frühe Phasen der Use Case Analyse geeignet. Zum einen sind sie zu unflexibel und zum anderen sind sie für viele Arten von Projektbeteiligten unverständlich.

Deshalb ist es i.d.R. nötig, in einer Use-Case-zentrierten Anforderungsanalyse mehrere verschiedene Modellierungsnotationen der Reihe nach oder sogar gleichzeitig einzusetzen. Dies erhöht den Modellierungsaufwand und birgt die Gefahr von Inkonsistenzen.

Derzeit fehlt es also an einer durchgängigen Notation, mit der Use Cases auf unterschiedlichen Detaillierungsgraden beschrieben werden können und deren Instanzen auf unterschiedliche Weise textuell und graphisch dargestellt werden können.

2.4.5 Formalisierungsgrade von Use Cases

Der Formalisierungsgrad eines Use Cases bezeichnet, inwieweit die im Use Case modellierten Informationen „semi"-automatisch durch eine geeignete Werkzeugunterstützung analysierbar sind. Hierbei lassen sich unabhängig von der verwendeten Notation (2.4.4) fünf verschiedene aufeinander aufbauende generelle Formalisierungsgrade unterscheiden. Im Folgenden werden diese Formalisierungsgrade vorgestellt und Aussagen dazu gemacht, welche Informationen jeweils automatisch analysierbar sind. Die vorgestellten Formalisierungsgrade sind dabei im Prinzip unabhängig vom Detaillierungsgrad der modellierten Use Cases.

Laborwerte beifügen

Das System zeigt jedem Arzt eine Liste aller verfügbaren Laborwerte. Dieser wählt alle relevanten Laborwerte aus. Falls Laborwerte fehlen, stellt er diese manuell ein. Zum Abschluss bestätigt er seine Eingaben.

Abbildung 2.7: Beispiel – Nichtformalisierter Use Case

2.4.5.1 Nichtformalisierte Use Cases

Bei nicht formalisierten Use Cases handelt es sich um Repräsentationen von Use Cases, die nicht für eine automatische Analyse aufbereitet sind. Bei dieser Art von Darstellung sind keine strukturellen oder semantischen Informationen formalisiert. Deshalb ist es nicht möglich, Aussagen über das in den Use Cases modellierte Verhalten abzuleiten. Es ist lediglich möglich, standardisierte Textanalyseverfahren, wie Rechtschreibprüfungen oder die Ermittlung von Lesbarkeitsindizes (vgl. [MP82]), durchzuführen.

		Author:	**VHoff**
ID	0001		
Titel	Laborwerte beifügen		
Primary Actor	Arzt		
	...		
Standard Flow	1. Das System zeigt eine Liste aller verfügbaren Laborwerte an. 2. Der Arzt wählt alle relevanten Laborwerte aus. 3. Der Arzt bestätigt seine Eingaben.		
Alternatives & Exceptions	Bei 2.: Falls Laborwerte fehlen: Der Arzt stellt die Laborwerte manuell ein. Weiter bei 3.		

Abbildung 2.8: Beispiel – Formalisiertes Use Case Template

2.4.5.2 Formalisierte Use Case Templates

Bei dieser Art der Darstellung ist zwar die Struktur eines Use Case Templates formalisiert, nicht jedoch die in den einzelnen Feldern des Templates enthaltene Information. Somit sind diese Inhalte der Templatefelder für ein Analysesystem eine Black-Box und können nicht weitergehend analysiert werden. Bei formalisierten Use Case Templates können also die primitiven Textanalysen, die bereits bei unstrukturierter Information möglich waren, verfeinert werden. Es ist zudem möglich, die Vollständigkeit der Use Case Beschreibung gegenüber den definierten Templatefeldern zu bestimmen. Außerdem können Komplexitätsmetriken auf der Templatestruktur erhoben werden (vgl. [BD04]).

		Author:	**VHoff**
ID	0001		
Titel	Laborwerte beifügen		
Primary Actor	Arzt		
	...		
Flow 0001.01	Laborwerte beifügen - Hauptfluss		
Contexts	0001.01.C01 Interaktion mit Arzt.		
Events	Action: Das System zeigt eine Liste aller verfügbaren Laborwerte an. Action: Der Arzt wählt alle relevanten Laborwerte aus. Extension: Laborwerte fehlen --> Fehlende Laborwerte Action: Der Arzt bestätigt seine Eingaben.		
Flow 0001.02	Laborwerte manuell einstellen		
Contexts	0001.02.C01 Extension Fehlende Laborwerte: wichtige Laborwerte fehlen		
Events	Action: Der Arzt stellt die fehlenden Laborwerte manuell ein.		

Abbildung 2.9: Beispiel – Strukturierter Use Case

2.4.5.3 Strukturierte Use Cases

Bei strukturierten Use Cases sind zusätzlich zum Use Case Template die Informationen zur Struktur des Use Case Modells selbst formalisiert. Genauer ist der Kontrollflussgraph des Ereignisflusses formalisiert modelliert. Somit ist der Zusammenhang zwischen einzelnen Elementen des Use Case Modells explizit beschrieben. Bei dieser Use Case Darstellung sind atomare Schritte mit den Akteuren, die sie ausführen, verbunden und Beziehungen zwischen Use Cases explizit in den Verhaltensbeschreibungen der einzelnen Use Cases modelliert.

Damit wird es möglich, die Beziehungen zwischen den Use Cases und den in den Use Cases modellierten Verhaltensfragmenten zu analysieren. So können z.B. unerreichbare Use Cases oder strukturelle Fehler, z.B. zwei Use Cases, die sich gegenseitig inkludieren, identifiziert werden. Außerdem können Komplexitätsmaße über die Struktur des Ablaufgraphs erhoben werden. Das in den Use Case Schritten und Verzweigungen modellierte Verhalten bleibt aber weiter rein textuell beschrieben und kann somit nicht automatisch analysiert werden.

		Author:	**VHoff**
ID	0001		
Titel	Laborwerte beifügen		
Primary Actor	Arzt		
	...		
Flow 0001.01	Laborwerte beifügen - Hauptfluss		
Contexts	0001.01.C01 Interaktion mit Arzt.		
Events	Action	Das System zeigt eine Liste aller verfügbaren Laborwerte an.	
	Action	Der Arzt wählt alle relevanten Laborwerte aus.	
	Extension	Laborwerte fehlen --> Fehlende Laborwerte	
	Action	Der Arzt bestätigt seine Eingaben.	
Flow 0001.02	Laborwerte manuell einstellen		
Contexts	0001.02.C01 Extension Fehlende Laborwerte: wichtige Laborwerte fehlen		
Events	Action	Der Arzt stellt die fehlenden Laborwerte manuell ein.	

Abbildung 2.10: Beispiel – Integrierter, strukturierter Use Case

2.4.5.4 Integrierte strukturierte Use Cases

Bei integrierten strukturierten Use Cases ist zusätzlich die Beziehung zwischen in den Use Cases modellierten Informationen und Elementen aus anderen Anforderungsartefakten formalisiert. So werden z.B. in einem Use Case Schritt explizit Elemente des Domänenmodells verwendet.

Damit wird es möglich, die Benutzung von Domänenbegriffen im Use Case Modell zu beobachten und daraus Metriken für z.B. die Lesbarkeit oder die Voll-

ständigkeit des Modells abzuleiten (vgl. [PVC07]).

		Author:	**VHoff**
ID	0001		
Titel	Laborwerte beifügen		
Primary Actor	Arzt		
	...		
Flow 0001.01	Laborwerte beifügen - Hauptfluss		
Contexts	0001.01.C01 Interaktion mit Arzt.		
Events	Action	Das System zeigt eine Liste aller verfügbaren Laborwerte an. table.show(labValues);	
	Action	Der Arzt wählt alle relevanten Laborwerte aus. selected = table.selection();	
	Extension	Laborwerte fehlen --> Fehlende Laborwerte	
	Action	Der Arzt bestätigt seine Eingaben. button.click();	
Flow 0001.02	Laborwerte manuell einstellen		
Contexts	0001.02.C01 Extension Fehlende Laborwerte: wichtige Laborwerte fehlen if(!selected.containsAll(expected));		
Events	Action	Der Arzt stellt die fehlenden Laborwerte manuell ein. selected+= manualList;	

Abbildung 2.11: Beispiel – Formalisierter Use Case

2.4.5.5 Formalisierte Use Cases

Bei formalisierten Use Cases ist, zusätzlich zur Struktur der Use Cases, das in den Use Cases modellierte Verhalten formalisiert. Die Formalisierung dieser Informationen wird hierbei dargestellt, indem die Ereignisse des Ereignisflusses eines Use Cases mit Elementen der modellierten Domäne und Systemzuständen verknüpft werden. Ein Ereignis enthält also beispielsweise Informationen dazu, welche Entitäten der modellierten Domäne in diesem Ereignis gelesen bzw. verändert werden und in welchem Zustand sich das System nach Durchführung eines Ereignisses befindet.

Damit ist es möglich, zu analysieren, wie der Datenfluss im beschriebenen Use Case Modell aussieht. Somit lassen sich z.B. Inkonsistenzen zwischen dem in den Use Cases modellierten Verhalten und Vor- bzw. Nachbedingungen identifizieren oder *Dead Ends* im Use Case Modell aufdecken, an denen das modellierte System einen Zustand erreicht, an dem kein Use Case mehr durchführbar ist.

2.4.5.6 Formalisierungsgrad & Prototyping

Prinzipiell ist es möglich, Prototypen aus strukturierten Use Cases oder höheren Formalisierungsgraden „semi“-automatisch abzuleiten. Dabei unterscheiden sich aber die Simulationsmöglichkeiten der Prototypen abhängig vom Formalisierungsgrad.

- **(Integrierte) strukturierte Use Cases:** In Prototypen strukturierter Use Cases ist es ausschließlich möglich, den Kontrollfluss der Use Cases zu analysieren. Dabei liegt das beschriebene Verhalten allerdings nur verbal vor. Die Simulationsumgebung ist deshalb nicht in der Lage, an Verzweigungspunkten über den korrekten Fortgang des Szenarios selbstständig zu entscheiden. Deshalb muss der Nutzer die Simulation manuell steuern. Um den Nutzer dabei zu unterstützen, kann jeweils der aktuelle Ablauf des Szenarios präsentiert werden. Bei integrierten strukturierten Use Cases kann die Entscheidungsfindung weiterhin unterstützt werden, indem lediglich die Schritte des aktuellen Simulationslaufes gezeigt werden, die mit denselben Daten wie die aktuelle Entscheidung verknüpft sind. Obwohl der Ablauf der Simulation manuell kontrolliert werden muss, hat sich gezeigt, dass schon diese Form der Verhaltenssimulation das Verständnis von Anwendern deutlich erhöht, da es die Einzelszenarien in Use Case Beschreibungen erlebbar macht (vgl. [Nau11]).

- **Formalisierte Use Cases:** Bei formalisierten Use Case Simulationen kann die Steuerung der Simulation komplett von der Simulationsumgebung übernommen werden. Diese setzt während der Durchführung von Einzelschritten Systemvariablen. Zusätzlich kann die Simulationsumgebung Entscheidungen über den Systemablauf durch Auswerten von Variablenbelegungen selbstständig treffen. Hierdurch wird es möglich, eine vollständige Analyse des beschriebenen Ablaufes durchzuführen und zusätzlich zum Kontrollfluss auch den Datenfluss zu beobachten.

- **Partiell formalisierte Modelle:** In der Praxis, insbesondere in iterativen, inkrementellen Entwicklungsprozessen, liegen oft Modelle vor, die sowohl strukturierte als auch formalisierte Use Cases enthalten. In diesen Modellen sind bestimmte Modellteile bereits formalisiert, wohingegen andere Teile des Modells lediglich in einer strukturierten Form vorliegen. Hier muss die Analyse in den strukturierten Teilen des Modells manuell durchgeführt werden. Damit die formalen Teile des Modells automatisch simuliert werden können, muss der Nutzer zusätzlich zur Steuerung des Kontrollflusses in der Lage sein, die im System enthaltenen Daten manuell zu verändern.

2.5 Prototyping

Unvalidierte Anforderungen führen zu häufigen, teuren Änderungen in späteren Phasen des Softwareentwicklungsprozesses. Deshalb ist die Anforderungsvalidierung eine der Kernaufgaben der Softwareentwicklung, und sie sollte so früh wie möglich im Entwicklungsprozess durchgeführt werden [WK92]. Hier gibt es allerdings ein gravierendes Problem mit der Validierung durch Nutzer. Der DSB Report fasst dies folgendermaßen zusammen:

> "We believe that users cannot, with any amount of effort and wisdom, accurately describe the operational requirements for a substantial software system without testing by real operators in an operational environment, and iteration on the specification. The systems built today are just too complex for the mind of man to foresee all the ramifications purely by the exercise of the analytic imagination." [Def87]

Prototyping versucht dieses Problem zu adressieren. Der Begriff Prototyping taucht in der Softwareentwicklung erstmals in den frühen achtziger Jahren auf. Er bezeichnet ein methodisches Vorgehen, bei dem ablauffähige Modelle bestimmter Systemaspekte zur Systemanalyse entwickelt werden. Kieback et. al. [KLSHZ91] definieren den Begriff wie folgt:

> „*Prototyping* ist eine Vorgehensweise, die ein bestimmtes Verständnis der Software-Entwicklung voraussetzt und die Auswirkungen auf den gesamten Entwicklungsprozess hat. Prototyping bedeutet bei der Systementwicklung frühzeitig ablauffähige Modelle („Prototypen") des zukünftigen Anwendungssystems zu erstellen und mit diesen zu experimentieren."

Prototyping hat dabei zwei zentrale Ziele:

- Schaffung einer Kommunikationsplattform für alle Arten von Projektbeteiligten
- Reduktion von Entwicklungsrisiken durch eine experimentelle, erfahrungsbasierte Vorgehensweise

2.5.1 Prototyp

Der Begriff Prototyp (gr. prototypos) bedeutet wörtlich übersetzt „Urbild" oder „erster seiner Art". In der Softwareentwicklung bezeichnet er eine „partielle" Implementierung eines Anwendungssystems zu Analysezwecken. Sommerville definiert den Begiff wie folgt:

> „Ein Prototyp ist eine initiale Version eines Softwaresystems, die

> verwendet wird, um Konzepte zu demonstrieren, Entwürfe zu erproben und -grundsätzlich- Erkenntnisse über das Problem und seine möglichen Lösungen zu gewinnen.“ [Som07b, 445]

Kieback et. al. [KLSHZ91] identifizieren drei wesentliche Merkmale eines Prototypen:

1. Prototypen sind ablauffähige Modelle, die bestimmte Aspekte eines Informationssystems realisieren.
2. Prototypen dienen als anschaulicher Diskussionsgegenstand in der Diskussion zwischen verschiedenen Projektbeteiligten. Sie werden zielgerichtet zur Diskussion eines Problems, der Klärung einer Frage oder der Vorbereitung einer Entscheidung entwickelt.
3. Jeder Prototyp dient als Spezifikation für weitere Prototypen oder das Anwendungssystem selbst.

2.5.2 Klassifikation von Prototypen

Wie bereits erwähnt, werden Prototypen immer zielgerichtet für bestimmte Einsatzzwecke erstellt. Dabei können die Prototypen nach verschiedenen Aspekten klassifiziert werden.

2.5.2.1 Klassifikation nach der Zielsetzung

Prototypen werden zur Beantwortung unterschiedlicher Fragen eingesetzt. Floyd [Flo84] unterscheidet hier drei Arten von Prototypingansätzen anhand ihrer Zielsetzung:

Exploratives Prototyping: Ziel des explorativen Prototyings ist primär die Klärung fachlicher Anforderungen. Hier soll vor allem die Kommunikation zwischen Anwender und Entwickler verbessert werden. In der Literatur wird häufig der Begriff *Rapid Prototyping* synonym für diese Art des Prototypings verwendet. Die verwendeten Prototypen werden als *Prototypen im engeren Sinne* bezeichnet.

Experimentelles Prototyping: Ziel des experimentellen Prototypings ist es, die Tauglichkeit einer technischen Lösung zu bewerten bzw. zwischen mehreren Lösungsmöglichkeiten abzuwägen. Die in diesem Zusammenhang eingesetzten Prototypen bezeichnet man als *Labormuster*.

Evolutionäres Prototyping: Beim explorativen und beim experimentellen Prototyping werden Prototypen ausschließlich zur Unterstützung der Systemanalyse eingesetzt und anschließend verworfen. Beim evolutionären Prototyping hingegen wird der Prototyp nicht verworfen, sondern als Kernsystem verwendet. Dieses Kernsystem wird in einem inkrementellen Prozess in mehreren Schritten zu einem Produktivsystem ausgebaut. Der entstehende Prototyp selbst wird als *Pilotsystem* bezeichnet.

Zusätzlich identifizieren Kieback et al. [Kie91] eine weitere Klasse von Prototypen, die zu Demonstrationszwecken während der Projektaquiese entwickelt wird. Diese bezeichnen sie als *Aquisitions-* bzw. *Demonstrationsprototypen*.

2.5.2.2 Klassifikation anhand des Detaillierungsgrades

Bei der Entwicklung von Prototypen kann der modellierte Aspekt auf verschiedenen Detaillierungsgraden betrachtet werden. Bischofberger und Pomberger [BP92] unterscheiden zunächst *vollständige* und *unvollständige Prototypen*. Ein vollständiger Prototyp modelliert dabei alle Aspekte des Zielsystems, wohingegen ein unvollständiger Prototyp lediglich bestimmte Aspekte des Zielsystems betrachtet. Allerdings weisen sie darauf hin, dass vollständige Protoytpen fast ausschließlich beim evolutionären Prototyping entwickelt werden.

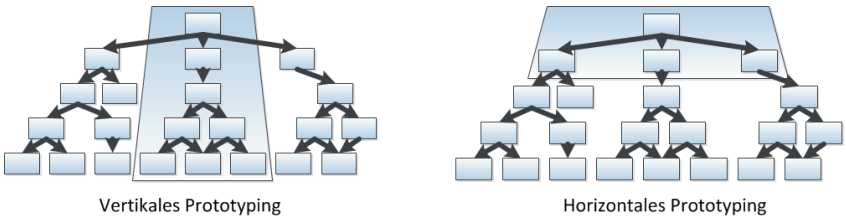


Abbildung 2.12: Modellierungsfokus beim Prototyping (Adaptiert von [WK92])

Nielsen [Nie94] unterscheidet die zwei in Abbildung 2.12 dargestellten Arten von unvollständigen Prototypen anhand ihres Detaillierungsgrades.

- *Vertikale Prototypen* konzentrieren sich auf die detaillierte Analyse eines kleinen Teils eines Systems oder Subsystems. Hierbei werden alle technischen Aspekte der Implementierung dieses Teils „in die Tiefe“ betrachtet.
- *Horizontale Prototypen* dienen einer Analyse des Systems in der Breite. Sie modellieren einen bestimmten technischen Aspekt eines Systems oder Subsystems, wie. z.B. die Nutzerschnittstelle, oder funktionale Aspekte, wie z.B. Datenbanktransaktionen [Kie91].

2.5.2.3 Klassifikation anhand des Modellierungsfokus

Grundsätzlich können Prototypen jeden Aspekt eines Softwaresystems modellieren. Dennoch haben sie i.d.R. einen bestimmten Modellierungsfokus. Lichter [Lic93] identifiziert hier drei typische Kategorien:

- *Nutzerschnittstellenprototypen* stellen einen Entwurf der geplanten Nutzerschnittstelle dar. Mit ihnen können zukünftige Nutzer die Nutzung und Ergonomie der geplanten Lösung evaluieren.
- *Funktionale Prototypen* realisieren einen Teil der geplanten Funktionen eines Systems. So kann untersucht werden, ob die geplanten Funktionen die Anforderungen adäquat widerspiegeln.
- *Architekturprototypen* definieren die Architektur eines Zielsystems, d.h. ihre Bausteine und Beziehungen, bevor diese vollständig implementiert sind. Sie dienen also der Analyse der Zielarchitektur.

Weiterhin nennen Wood und Kang [WK92] *Performanceprototypen*, die primär einer Analyse des Laufzeitverhaltes bestimmter Teile des Zielsystems, wie z.B. der Kommunikation zwischen Datenbank und Applikationslogik, dienen.

Nutzerschnittstellenprototypen: Nutzerschnittstellenprototypen spielen in dieser Arbeit eine besondere Rolle. Tabelle 2.1 stellt neun gängige Techniken zum Nutzerschnittstellenprototyping auf unterschiedlichen Detaillierungsleveln dar.

Technik	**Kurzbeschreibung & Einsatz**
Card sorting	Card sorting ist eine partizipative Methode. Sie dient zur Identifikation von geeigneten Navigationsstrukturen eines interaktiven Systems.
Wireframes	Wireframes sind skizzenhafte Darstellungen von Nutzerschnittstellenseiten. Sie werden verwendet, um einfache Designentscheidungen der Nutzerschnittstelle festzulegen.
Storyboards	Storyboards sind sequentielle Bildschirm-für-Bildschirm Abläufe. Sie werden verwendet, um Benutzungsszenarien z.B. eines Anwendungsfalls darzustellen.
Papierprototypen	Papierprototypen sind interaktive Prototypen auf Papier. Sie bestehen aus einem Mockup der Nutzerschnittstelle. Die Funktion des Prototypen wird manuell simuliert. Sie dienen dazu, konzeptionelle Designs mit Nutzern zu evaluieren.

Digitale Prototypen	Digitale Prototypen sind Nutzerschnittstellenprototypen, die mit Navigationspfaden angereichert sind. Sie erlauben es, verschiedene Szenarien eines Systems durchzuspielen. Dabei sind sie aber in der Regel nicht voll interaktiv, sondern stellen ähnlich wie Storyboards lediglich einen möglichen Ablauf dar. Sie werden ähnlich wie Papierprototypen primär zur Evaluierung von Konzepten mit Nutzern eingesetzt.
Blank Model Prototypen	Blank Model Prototypen dienen speziell zum Design von Hardwarekomponenten für die Nutzerinteraktion. Dazu werden im Wesentlichen typische Bastelmaterialen eingesetzt.
Videoprototypen	Videoprototypen dienen dazu, im Innovationsdesign Interaktionen mit einem System zu simulieren, das noch nicht entwickelt wurde. Dazu werden Videos aufgezeichnet, in denen Nutzer so tun, als würden sie das zu entwickelnde System benutzen. Oftmals dienen sie auch dazu, Nutzerrollen und den Benutzungskontext eines Systems zu dokumentieren.
Wizard-of-Oz Prototypen	Wizard-of-Oz Prototypen dienen der Benutzbarkeitsanalyse insbesondere bei taktilen Nutzerschnittstellen, wie z.B. Spracherkennung. Sie sind interaktive Prototypen, bei denen der Nutzer denkt, er bediene ein funktionales System. Allerdings werden die Systemreaktionen, wie beim Papierprototyping, manuell simuliert. Der Simulator ist dabei i.d.R. vor dem Nutzer verborgen.
Codierte Prototypen	Ein codierter Prototyp ist eine partielle Implementierung des Zielsystems. Er wird oft entweder in der Zielsprache des Systems oder mit einer 4th-Generation-Sprache entworfen. Für kodierte Prototypen gibt es drei Haupteinsatzgebiete: Sie werden explorativ für detaillierte Benutzbarkeitsstudien verwendet, sie werden experimentell genutzt, um technische Probleme zu analysieren, oder sie werden evolutionär zu Produktivsystemen weiterentwickelt verwendet.

Tabelle 2.1: Nutzerschnittstellenprototypen nach Arnowitz et. al. [AAB06]

Diese unterschiedlichen Arten von Nutzerschnittstellenprototypen lassen sich weiterhin grob in drei Kategorien einteilen:

- **Statische Nutzerschnittstellenprototypen:** Statische Nutzerschnittstellenprototypen stellen die Nutzerschnittstelle des Systems oder einen

Teil von ihr lediglich visuell dar. Sie bieten keine Möglichkeit zur Interaktion an. Hierzu zählen Wireframes und Storyboards.

- **Navigierbare Nutzerschnittstellenprototypen:** Digitale Prototypen und Papierprototypen werden als navigierbare Nutzerschnittstellenprototypen bezeichnet. Sie bieten die Möglichkeit, verschiedene Szenarien zu emulieren, indem die zugehörigen Bildschirmseiten der Reihe nach angezeigt werden. Weiterhin ist es möglich, den Ablauf der Szenarien durch Interaktion mit der Nutzerschnittstelle zu beeinflussen. Die dargestellten Nutzerschnittstellen selbst sind aber fest.

- **Funktionale Nutzerschnittstellenprototypen:** Bei funktionalen Nutzerschnittstellenprototypen sind zusätzlich zu Bildschirmfolgen auch Teile der Systemfunktionalität implementiert. Sie ermöglichen die Eingabe von Daten und reagieren dynamisch auf diese, indem die dargestellten Nutzerschnittstellen angepasst werden.

2.5.3 Einsatz von Prototypen

Grundsätzlich kann Prototyping in allen Arten von Softwareentwicklungsprozessen eingesetzt werden. Dies macht, abhängig vom Entwicklungsprozess, verschiedene Anpassungen des Vorgehens nötig. Diese werden beispielsweise von Bischofberger und Pomberger [BP92] im Detail erörtert. Typischerweise werden Prototypen schwerpunktmäßig in zwei Entwicklungsphasen, nämlich der Anforderungsanalyse und dem Softwaredesign, eingesetzt.

Prinzipiell kann Prototyping in allen Arten von Softwareprojekten eingesetzt werden. Allerdings ist Prototyping mit einem nicht unerheblichen Aufwand verbunden. Deshalb muss sein Einsatz sorgfältig abgewogen werden. Besonders effektiv ist Prototyping, wenn das Zielsystem eine große interaktive Komponente enthält (vgl. [Cri92]). Andere Systeme mit sehr wenig Nutzerinteraktion profitieren oft weit weniger von Prototypingansätzen. Bei solchen Systemen, wie z.B. Batchsystemen, die im wesentlichen Berechnungen durchführen, muss genau abgewogen werden, ob sich der Entwicklungsaufwand für Prototypen lohnt.

2.5.4 Ablauf des Prototypings im Entwicklungsprozess

Unabhängig vom konkreten Entwicklungsprozess in dem Prototypingtechniken eingesetzt werden sollen, folgt das Prototyping dem in Abbildung 2.13 dargestellten allgemeinen Ablauf. Ausgangspunkt für das Prototyping sind stets Situationen, in denen wichtige Informationen für die Weiterentwicklung der Software fehlen, unklar sind oder nur vage beschrieben werden können. Aus diesen Situationen leiten sich *offene Fragen* ab, die mit dem Prototyp beantwortet werden sollen.

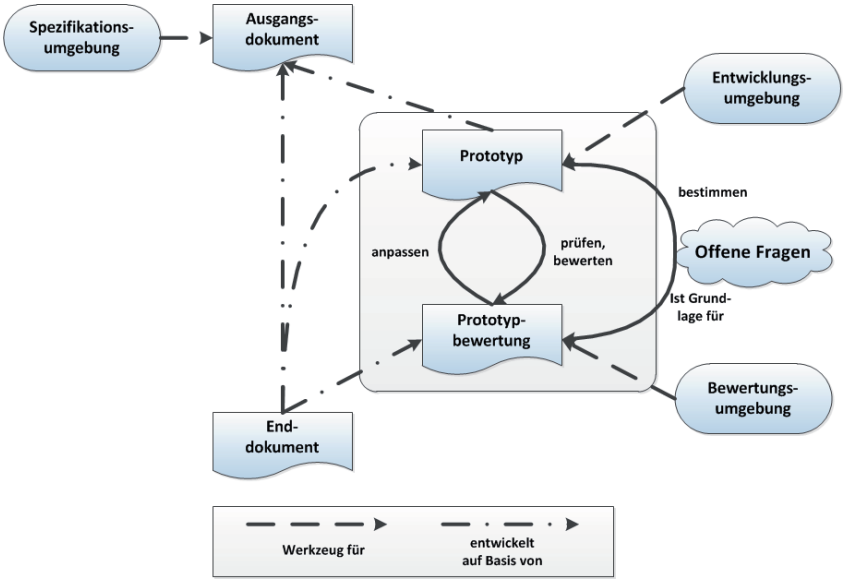


Abbildung 2.13: Prototypingablauf (adaptiert von Lichter [Lic93, 33])

Ausgangspunkt des Prototypings bildet ein *Ausgangsdokument*, das den aktuellen Wissensstand vor dem Prototyping fixiert. Beim explorativen Prototyping ist dieses Ausgangsdokument normalerweise eine Anforderungsspezifikation oder ein Teil davon.

Auf Basis dieses Ausgangsdokuments und der Rahmenbedingungen wird zunächst ein initialer *Prototyp* entworfen und implementiert.

Anschließend durchläuft der Prototyp einen Zyklus aus Ausführung, Bewertung und Anpassung. Bei der Ausführung wird der Prototyp anhand der formulierten Fragestellungen bewertet. Die Bewertungsergebnisse werden in der *Prototypbewertung* dokumentiert. Können nicht alle offenen Fragen adäquat beantwortet werden, wird der Prototyp entsprechend angepasst und erneut bewertet. Dieser *Prototypingzyklus* wird so lange durchlaufen, bis der Prototyp entweder den modellierten Aspekt angemessen darstellt oder ein pragmatisches Abbruchkriterium (z.B. Zeit oder Geld) erreicht wird. Am Ende des Prototypingzyklus erhält man ein *Enddokument*, das aus dem Prototypen, der Prototypbewertung und einer angepassten Version des Anfangsdokuments besteht.

Prototyping ist im Allgemeinen nur mit einer geeigneten Werkzeugunterstützung effizient durchführbar. Diese Werkzeugunterstützung besteht aus mehreren verschiedenen Softwarewerkzeugen, die während des Prototypings eingesetzt werden. Das Ausgangsdokument wird in einer *Spezifikationsumgebung* entwickelt und verändert. Der Prototyp wird in einer *Entwicklungsumgebung* entworfen, implementiert und ggf. angepasst. Die Ausführung und Bewertung der

Prototypen findet in einer *Bewertungsumgebung* statt. Diese enthält oftmals zusätzlich einen *Ablaufrecorder*, der Informationen über die Prototypausführung, die sogenannten *Traces*, aufzeichnet. Die Gesamtheit der zum Prototyping verwendeten Werkzeuge, also die Kombination von Spezifikations-, Entwicklungs- und Bewertungsumgebung, wird im weiteren Verlauf als *Prototypingumgebung* bezeichnet.

Zusätzlich wird jedes der Dokumente in einer speziellen Notation formuliert. Das Ausgangsdokument wird in der *Spezifikationsnotation* beschrieben. Der Prototyp wird in der *Prototypnotation* modelliert. Die Bewertungen werden in der *Bewertungsnotation* dokumentiert.

Rollen beim Prototyping: Während des Prototypings nehmen die Projektbeteiligten verschiedene Rollen ein. Im weiteren Verlauf dieser Arbeit sind besonders zwei Rollen interessant. Der *Prototypentwickler* erzeugt aus dem Spezifikationsdokument einen Prototyp. Außerdem passt er das Spezifikationsdokument bzw. den Prototyp im Prototypingzyklus auf Basis der Änderungsvorschläge aus der Prototypbewertung an. Die Rolle des Prototypentwicklers wird typischerweise von den Softwareleuten des Softwareherstellers eingenommen.

Bewertet wird der Prototyp vom *Prototypnutzer*. Dieser setzt den Prototyp in der Einsatzumgebung ein und dokumentiert seine Befunde. Prototypnutzer sind abhängig vom Einsatzzweck des Prototyps in der Regel sowohl Softwarehersteller als auch Klienten. Insbesondere während der Entwicklung des Prototyps wird die Rolle des Prototypnutzers oftmals von den Softwareherstellern übernommen. Die eigentliche Bewertung des Prototyps erfolgt aber i.d.R. durch Prototypnutzer des Klienten.

Modellgetriebenes Prototyping In dieser Arbeit wird ein modellgetriebener Ansatz zum Prototyping präsentiert. Dabei werden alle zum Prototyping verwendeten Dokumente in Form von Modellen spezifiziert.

Da es sich um einen Ansatz zum Rapid Prototyping handelt, in dem eine Anforderungsspezifikation das Ausgangsdokument bildet, ist das *Spezifikationsmodell* das Ausgangsdokument des Ansatzes. Analog werden die Begriffe *Prototypmodell* und *Bewertungsmodell* verwendet.

Jedes dieser Modelle wird durch ein Metamodell definiert. Die Ausgangsnotation wird durch das *Spezifikationmetamodell* beschrieben. Das *Prototypmetamodell* legt die Prototypingnotation fest und das *Bewertungsmetamodell* definiert die Bewertungsnotation.

2.5.5 Vor- und Nachteile des Prototypings

Verschiedene Publikationen diskutieren Vor- und Nachteile des Prototypings [GB95, BP92, And94]. Der Einsatz von Prototypen bringt mehrere Verbesserungen für den Softwareentwicklungsprozess mit sich. Die frühe Verfügbarkeit eines ausführbaren Modells des Zielsystems vereinfacht insbesondere die Kommunikation zwischen Softwareleuten und Klienten und die Validierung von Anforderungen und Konzepten. Außerdem verbessert Prototyping wichtige Gebrauchsqualitäten der Software und reduziert so das Risiko von Akzeptanzproblemen.

Allerdings müssen auch einige potentielle Nachteile in Kauf genommen werden [AH93]. Hier werden vor allem übersteigerte Erwartungen der zukünftigen Nutzer und Probleme mit der Qualität von Architektur und Implementierung des Zielsystems genannt. Letzteres Problem wird allerdings primär bei evolutionärem Prototyping beobachtet.

Weiterhin gibt es oft das Problem, dass bei Anpassungen des Prototyps Inkonsistenzen zwischen dem Ausgangsdokument und dem Prototypen entstehen. Grund hierfür ist, dass in vielen Prototypingansätzen eine methodische und werkzeugseitige Unterstützung für die Integration des Ausgangsdokuments und des Prototyps fehlen. Dies kann zu Fehlern im Enddokument und zu einem unnötig hohen Entwicklungsaufwand führen.

3 Problemstellung, Ziele & Lösungsideen

In diesem Kapitel werden die Ziele dieser Arbeit vorgestellt. Dazu wird zunächst in den Abschnitten 3.1 und 3.2 die Problemstellung motiviert. Anschließend stellt Kapitel 3.3 die Forschungsziele vor. Abschließend wird in Abschnitt 3.4 der entwickelte Lösungsansatz skizziert und somit ein Ausblick auf die weitere Arbeit gegeben.

3.1 Motivation

Wie bereits mehrfach erwähnt, hat sich die Use Case Modellierung als eine wichtige Technik für die Modellierung von funktionalen Anforderungen etabliert. Die Use Case Modellierung ist ein integraler Bestandteil vieler moderner Entwicklungsprozesse. Use Case Modelle bilden dabei nicht nur eine wichtige Säule für die Anforderungsmodellierung, sondern sie bilden im Sinne der Use Case getriebenen Entwicklung ein zentrales Artefakt für den gesamten Entwicklungsprozess (vgl. Kap. 2.4.2).

Rein textuelle Use Cases haben allerdings drei prinzipielle Schwächen, die die Qualität der Spezifikationen verschlechtern [HL10]:

1. Use Cases beschreiben das Verhalten eines Systems in Form einer abstrakten Interaktionssequenz. Genauer gesagt werden in der Beschreibung nicht konkrete Eingabewerte, sondern abstrakte verhaltensäquivalente Platzhalter modelliert (z.B. `Gültige PIN-Nummer` statt `PIN 1234`). Diese Art der Modellierung ist für viele Projektbeteiligte, die typischerweise in konkreten Szenarien denken, schwierig nachzuvollziehen.

2. Jeder Use Case beschreibt mehrere verschiedene, teils optionale Szenarien gleichzeitig. Zusätzlich sind diese Szenarien i.d.R. in mehrere Teile zersplittert, die möglicherweise sogar in andere Use Cases ausgelagert sein können. Deshalb haben insbesondere Fachexperten Probleme, die Korrektheit von Use Cases zu beurteilen, weil hierzu die einzelnen Szenarien manuell identifiziert und analysiert werden müssen.

3. Weil jeder Use Case nur eine Funktion des Systems bzw. einen Teil einer Systemfunktion beschreibt, ist es schwierig, das gesamte Systemverhalten, welches durch die Kombination verschiedener Use Cases entsteht, zu evaluieren. Denn Systemzustände und mögliche Übergänge zwischen Use

> Cases werden nur implizit in den Vor- und Nachbedingungen modelliert. Deshalb sind traditionelle Inspektionstechniken nur wenig geeignet, um Inkonsistenzen im globalen Systemverhalten, die durch Kombination von Use Cases entstehen, oder fehlende Anforderungen zu identifizieren.

Prototyping ist eine etablierte Technik, um diese Probleme zu bewältigen [DP04, SS97]. Es bietet einen intuitiven Zugang zum beschriebenen Verhalten, indem es dieses erlebbar macht (vgl. [Poh08, 475]). Verschiedene empirische Untersuchungen [Gem03, RST+, RST+10, Mus10] zeigen, dass schon einfache Animationsansätze das Verständnis der Projektbeteiligten signifikant erhöhen. So kann das Risiko von Fehlern in der Spezifikation deutlich reduziert werden [GB95], [You04, 102], [SS97, 94-98] [LV01]. Deshalb wird der Einsatz von Prototypen in praktisch allen neueren Veröffentlichungen zur Use-Case-zentrierten Analyse empfohlen.

3.2 Probleme & Herausforderungen

Obwohl die Use-Case-zentrierte Anforderungsanalyse seit über 20 Jahren eine etablierte Technik ist und Jacobson [Jac04, 166] bereits 1992 auf die Nützlichkeit von Prototypen zur Verbesserung von Use-Case-zentrierten Spezifikationen hingewiesen hat, werden die Potentiale des Prototypings in der Use-Case-zentrierten Anforderungsanalyse immer noch nur unzureichend genutzt. Hauptgrund hierfür ist die mangelnde konzeptionelle und methodische Integration von Prototypingtechniken in die Use-Case-zentrierte Anforderungsanalyse. Im Folgenden werden die Schwächen gebräuchlicher Ansätze einzeln vorgestellt. Darauf aufbauend werden im nächsten Abschnitt die Ziele dieser Arbeit identifiziert.

Schwäche 1: Diskontinuierliche Notation

Bis heute werden Use-Case-zentrierte Spezifikationen und Prototypen weitestgehend unabhängig voneinander entworfen und weiterentwickelt. Ferner werden jeweils separate Notationen, die nicht miteinander integriert sind, verwendet. Deshalb müssen Prototypen bzw. Use Cases bei Änderungen, wie sie in einem iterativen, inkrementellen Vorgehen auftreten, manuell angepasst werden. Dieses Vorgehen ist fehleranfällig, zeitintensiv und teuer. Außerdem verschlechtern auftretende Inkonsistenzen zwischen Prototypen und Use Cases die Qualität der Gesamtspezifikation. Aktuell fehlt es also an einer Integration der verschiedenen verwendeten Prototypingnotationen (Spezifikations-, Prototyping- und Ergebnisnotation) und den Notationen der Use-Case-zentrierten Anforderungsanalyse, die die Ausgangsnotation für das Prototyping bilden. Eine geeignete Integration muss es ermöglichen, Prototypen und Use-Case-zentrierte Spezifikationen unterschiedlicher Detaillierungsstufen miteinander konsistent zu halten und gemeinsam weiterzuentwickeln.

Schwäche 2: Unzureichende methodische Einbettung

In der Use-Case-zentrierten Anforderungsanalyse werden die Anforderungsartefakte schrittweise in unterschiedlichen Modellierungsphasen von kurzen Skizzen zu einer detaillierten Spezifikation weiterentwickelt. Hierbei sind verschiedene Arten von Prototypen unterschiedlich gut geeignet, die behandelten Modellierungsfragen zu adressieren.

Damit Prototyping effektiv in einem solchen Prozess eingesetzt werden kann, ist ein strukturiertes Vorgehen nötig. Dieses muss definieren, welche unterschiedlichen Arten von Prototypen in der Anforderungsanalyse erzeugt werden, und den Einsatz und Nutzen von Prototypen für jede Phase klar festlegen.

Ein solches Vorgehen fehlt in aktuellen Use-Case-zentrierten Anforderungsanalyseansätzen. Zwar gibt es einige Ansätze, die definieren, wie bestimmte Arten von Prototypen strukturiert aus Use Cases abgeleitet werden können, sie machen aber keinerlei Angaben dazu, wie diese Prototypen effektiv im Entwicklungsprozess eingesetzt werden können.

Schwäche 3: Ungeeignete Werkzeugunterstützung

Wie auch im Falle der Notationen fehlt es an einer Integration der verschiedenen Werkzeuge für das Prototyping in der Use-Case-zentrierten Anforderungsanalyse. Aktuelle Werkzeuge sind auf die Entwicklung von Use Cases oder Prototypen beschränkt. Insbesondere gibt es derzeit keine direkte Verbindung zwischen Spezifikationswerkzeugen für Use Cases und Prototypingwerkzeugen. Zusätzlich sind viele Prototypingwerkzeuge (Entwicklung und Einsatz) auf eine bestimmte Art von Prototyp beschränkt.

Aktuelle Werkzeuge sind also aus drei Gründen für den Einsatz in der Use-Case-zentrierten Anforderungsanalyse ungeeignet:

- Die ausführungsrelevanten Informationen müssen mehrfach modelliert werden, da sie sowohl Teil der Use-Case-zentrierten Spezifikation als auch der verschiedenen Prototypen sind.
- Es gibt keine Möglichkeit, die Konsistenz von Prototyp und zugehöriger Spezifikation automatisiert sicherzustellen. Außerdem ist es nicht möglich, Analyseergebnisse semi-automatisch zwischen den Prototypen und der Use-Case-zentrierten Spezifikation zu übertragen.
- Ein kontinuierlicher Übergang zwischen den verschiedenen Arten von Prototypen wird nicht unterstützt.

Zusammenfassend bleiben aufgrund dieser Mängel viele der beschriebenen Synergieeffekte ungenutzt. Der manuelle Anpassungsaufwand macht den Einsatz

von Prototypen in der Use-Case-zentrierten Analyse teuer, Inkonsistenzen zwischen Prototyp und Use Cases verschlechtern die Qualität der Spezifikation, und eine ungeeignete Werkzeuglandschaft verringert die Entwicklungsgeschwindigkeit.

Bis heute ist zusätzlich unklar, welche Arten von Prototypen in unterschiedlichen Phasen der Use-Case-zentrierten Anforderungsanalyse eingesetzt werden sollten, wie diese unterschiedlichen Prototypen systematisch gemeinsam mit Use-Case-zentrierten Spezifikationen entwickelt werden müssen, und wie Ergebnisse aus dem Einsatz der Prototypen zur Verbesserung der Use-Case-zentrierten Anforderungsanalyse verwendet werden können. Außerdem fehlt es an einer geeigneten Werkzeugumgebung, die den Einsatz von Prototypen in der Use-Case-zentrierten Anforderungsanalyse unterstützt.

3.3 Forschungsfragen

Aus der Motivation der Problemstellung und der Analyse der aktuellen Situation wird klar, dass es zum effektiven Einsatz von Prototyping in der Use-Case-zentrierten Anforderungsanalyse an der konzeptionellen, methodischen und technischen Einbettung mangelt. Ziel dieser Arbeit ist es deshalb, Prototyping in die Use-Case-zentrierte Anforderungsanalyse zu integrieren. Dazu sollen die folgenden Forschungsfragen beantwortet werden.

Frage 1: Wie können Prototypingnotationen in die Use-Case-zentrierte Anforderungsanalyse integriert werden?

Damit Prototyping effektiv in der Use-Case-zentrierten Anforderungsanalyse eingesetzt werden kann, müssen alle Notationen des Prototypingprozesses (Use-Case-zentrierte Spezifikationsnotation, Prototypnotation und Ergebnisnotation) miteinander integriert werden. Dies ist nur möglich, wenn eine Notation für die Beschreibung Use-Case-zentrierter Spezifikationen gefunden werden kann, die hinreichend formal ist, sodass sich Prototypen semi-automatisch ableiten lassen. Es gilt also die Frage zu beantworten, wie sich die Elemente der Use-Case-zentrierten Spezifikation so miteinander kombinieren lassen, dass ausführbare Prototypen abgeleitet werden können, die zum einen das Verhalten präzise wiedergeben und zum anderen geeignet sind, das Verhalten mit Klienten zu evaluieren.

Frage 2: Wie können Prototypingtechniken in die Use-Case-zentrierte Anforderungsanalyse eingebettet werden?

Die Use-Case-zentrierte Anforderungsanalyse wird iterativ, inkrementell in mehreren Phasen durchgeführt. Ein Ziel dieser Arbeit ist es, zu beschreiben, wie verschiedene Prototypingtechniken strukturiert zur Verbesserung der Analyse in diesen Entwicklungsansatz eingebettet werden können. Hier sollen insbesondere die folgenden zwei Fragen beantwortet werden:

- Welche Arten von Prototypen sollen in welchen Analysephasen entwickelt werden?
- Welche Analysetätigkeiten sollen mit diesen Prototypen unterstützt werden?

Frage 3: Wie können Entwicklung und Einsatz von Prototypen in der Use-Case-zentrierten Analyse unterstützt werden?

Prototyping ist nur mit einer geeigneten Werkzeugunterstützung ökonomisch sinnvoll [Plo04a]. Deshalb ist, abgesehen von den konzeptionellen und methodischen Fragestellungen zum Use-Case-zentrierten interaktiven Prototyping, zu klären, wie eine angemessene Werkzeugunterstützung aussehen kann. Eine geeignete Werkzeugunterstützung muss es zum einen ermöglichen, die unterschiedlichen Artefakte der Use-Case-zentrierten Spezifikation zu entwickeln, zu verändern und miteinander konsistent zu halten, zum anderen muss es möglich sein, verschiedene Prototypen kostengünstig zu entwickeln und anzupassen. Außerdem müssen diese ausgeführt und analysiert werden können.

3.4 Überblick über die entwickelte Lösung

Nachdem nun die zentralen Probleme, Herausforderungen und Ziele definiert sind, soll eine Skizze der entwickelten Lösung vorgestellt werden, um eine Orientierungshilfe für die folgenden Kapitel zu geben.

*SUPrA (**S**tructured **U**se Case centered **Pr**ototyping **A**pproach)* ist ein prototypingorientierter, modellgetriebener Ansatz zur Use-Case-zentrierten Analyse. Er besteht aus den drei in Abbildung 3.1 skizzierten zentralen Teilen.

Grundlage des Ansatzes ist *ProDUCE (**Pro**totyping **D**riven **U**se **C**ase **E**ngineering)*, eine leichtgewichtige Methode zur Use-Case-zentrierten Anforderungsanalyse. ProDUCE integriert unterschiedliche Prototypingtechniken in die iterative, inkrementelle Use-Case-zentrierte Analyse. Ziel ist es, Prototypen so früh wie möglich und prozessbegleitend einzusetzen, um die Qualität der Analyseergebnisse zu verbessern.

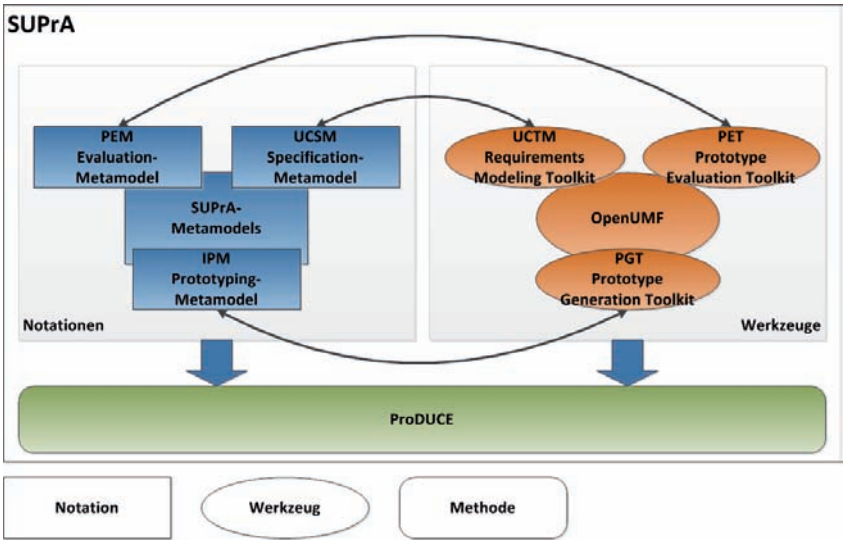


Abbildung 3.1: SUPrA – Überblick

ProDUCE ist in eine durchgängige Werkzeugunterstützung eingebettet. Die Prototypingumgebung *OpenUMF (**Open** **U**se Case **M**odeling **F**ramework)* besteht aus der Spezifikationsumgebung *UCMT (**U**se Case **C**entered **M**odeling **T**oolkit)* zur Entwicklung Use-Case-zentrierter Anforderungsspezifikationen, dem Prototypgenerator *PGT (**P**rototype **G**eneration **T**oolkit)* zur Erzeugung verschiedener Arten von Prototypen aus diesen Spezifikationen sowie der Bewertungsumgebung *PET (**P**rototype **E**valuation **T**oolkit)*.

Basis für diese Werkzeugunterstützung sind die SUPrA-Metamodelle. Das *UCSM (**U**se Case **C**entered **S**pecification **M**odel)* ist ein Use-Case-zentriertes Spezifikationsmetamodell. Im Unterschied zu vielen anderen Anforderungsmodellierungsnotationen bietet das UCSM Modellierungselemente für alle Perspektiven der Use-Case-zentrierten Anforderungsanalyse auf unterschiedlichen Detaillierungsgraden an. In SUPrA bildet es die Spezifikationsnotation.

Auf Basis einer formalen Petrinetzausführungssemantik für UCSM können verschiedene Arten von Nutzerschnittstellenprototypen automatisch generiert werden. Die Ergebnisse aus der Prototypbewertung werden dann in die SUPrA-Modelle eingepflegt und im nächsten Prototypingzyklus weiterverwendet.

Nachdem in Kapitel 4 verwandte Arbeiten diskutiert wurden, werden in den folgenden Kapiteln die einzelnen Aspekte von SUPrA der Reihe nach vorgestellt. Zunächst werden in Kapitel 5 die Anforderungen an SUPrA vorgestellt. Kapitel 6 präsentiert ein Anwendungsbeispiel, das durchgängig zur Einführung der Konzepte verwendet wird. In Kapitel 7 wird das UCSM und die zugehörige Spezifikationsumgebung vorgestellt.

Anschließend wird die Prototypgenerierung erläutert (Kap. 8). Dazu wird die Ausführungssemantik von UCSM erklärt, das Prototypingmetamodell *IPM (**I**ntegrated **P**rototyping **M**odel)* eingeführt und der Prototyp-Generierungsalgorithmus präsentiert.

In Kapitel 9 wird die Prototypbewertung diskutiert. Hierzu wird die Bewertungsumgebung PEF und das Bewertungsmetamodell *PEM (**P**rototype **E**valuation **M**odel)* vorgestellt und die Verbindung zwischen Bewertungsergebnissen und Spezifikationsmodell erörtert.

Die ProDUCE Methode wird in Kapitel 10 erläutert. Technische Details der Referenzimplementierung von OpenUMF bietet Kapitel 11.

Abschließend stellt Kapitel 12 Erfahrungen aus dem praktischen Einsatz vor. In Kapitel 13 wird eine Evaluierung des Ansatzes unter Berücksichtigung der formulierten Ziele vorgenommen.

4 Verwandte Arbeiten

Exploratives Prototyping bzw. dynamische Verhaltenssimulationen sind als Maßnahme zur Erhebung und Qualitätssicherung von Anforderungen bereits seit den frühen achtziger Jahren etabliert. Praktisch alle Textbücher zur Anforderungsanalyse empfehlen den Einsatz von Prototypen zur Ermittlung und Validierung von unklaren Anforderungen und zur Analyse von Ablaufbeschreibungen [Som07b, 443ff.]. Dementsprechend gibt es eine Vielzahl unterschiedlicher Ansätze, die das Prototyping in der Anforderungsanalyse unterstützen sollen. Betrachtet man das Prototyping aus dem Kontext einer Use-Case-zentrierten Anforderungsanalyse, lassen sich grob vier verschiedene Arten von Ansätzen unterscheiden:

- Vollautomatische Verhaltensanalyse
- Nicht Use-Case-basiertes Prototyping
- Use Case Simulation ohne Nutzerschnittstellen
- Use-Case-basiertes Prototyping mit Nutzerschnittstellen

Im Folgenden werden diese Ansätze kurz vorgestellt und jeweils einige wichtige Aspekte diskutiert. Anschließend werden einige dem in dieser Arbeit vorgestellten Ansatz verwandte Arbeiten diskutiert und abgegrenzt.

4.1 Vollautomatische Verhaltensanalyse

Vollautomatische Verhaltensanalysen sind keine Prototypingtechniken im engeren Sinne. Sie werden lediglich der Vollständigkeit halber erwähnt, da auch ihnen eine Formalisierung von Use-Case-basierten Spezifikationen zu Grunde liegt.

Vollautomatische Verhaltensanalysen dienen der „formalen“ Verifikation bestimmter Eigenschaften einer Use-Case-zentrierten Spezifikation. Dazu werden formale Use-Case-zentrierte Spezifikationen in einer speziellen Analyseumgebung vollautomatisch geprüft. Es werden symbolische Ausführungen des beschriebenen Verhaltens [Tuo98, SCK09] oder statische Analysen [ZD09, LMCS98, HHT02, DTK03] verwendet, um bestimmte Aspekte der internen Korrektheit einer Use-Case-zentrierten Anforderungsspezifikation zu prüfen.

Keiner dieser Ansätze bietet dem Nutzer die Möglichkeit, Szenarien einzeln der

Reihe nach durchzuspielen. Sie sind deshalb nicht zum explorativen Prototyping geeignet. Sie können aber zusätzlich zu Prototypen eingesetzt werden, um die Qualität von Use-Case-zentrierten Spezifikationen weiter zu verbessern, da die zu Grunde liegenden Verifikationsansätze auch auf hinreichend formale Prototypen übertragen werden können.

4.2 Nicht Use-Case-basierte Prototypingansätze

Insbesondere im Usage Centered Design [CL99] werden verschiedene Arten von Nutzerschnittstellenprototypen (vgl. Kap. 2.5.2.3) eingesetzt, um Funktionen zu evaluieren. Einen Überblick über gebräuchliche Techniken geben Arnowitz et. al. [AAB06]. Allerdings werden diese Prototypen unabhängig von Use Cases entwickelt und sie sind i.d.R. auch völlig ohne den Einsatz von Use Cases verwendbar. Wenn sie gemeinsam mit Use Cases eingesetzt werden, so müssen die Prototypen manuell aus dem Ereignisfluss der Use Cases aufgebaut werden. Hierzu müssen alle relevanten Szenarien in den Prototyp übertragen werden. Dies führt, wie bereits mehrfach erwähnt, zu erheblichem Mehraufwand in iterativen Prozessen, in denen sich die Anforderungen ändern.

4.3 Use Case Prototyping ohne Nutzerschnittstellen

Ein häufig verwendeter Ansatz zum Use Case Prototyping ist die Use Case Simulation. Hier wird das in der Use Case Spezifikation beschriebene Verhalten simuliert, anstatt einen eigenständigen Prototypen zu erzeugen. Dies hat laut Seybold et al. zwei zentrale Vorteile [SMG04]:

1. Die Verhaltenssimulation ist kostengünstiger als das Erzeugen eines Prototyps. Dies gilt insbesondere dann, wenn sich die Anforderungen ändern.

2. Inkonsistenzen zwischen Prototyp und Spezifikation sind ausgeschlossen.

Als Grundlage der Simulation werden formale Ausführungsmodelle aus der Use-Case-zentrierten Spezifikation erzeugt. Dazu werden die Use Cases und teilweise auch Modelle der Daten- und Funktionsperspektive verwendet.

Die Simulation läuft dann schrittweise in einer speziellen Laufzeitumgebung ab. Der Ablauf der Simulation kann von außen beeinflusst werden, indem ein Nutzer bestimmte Stimuli eingibt.

Hierbei gibt es sowohl Ansätze, bei denen direkt in einer formalen Notation spezifiziert wird, als auch Ansätze, bei denen eine semi-formale Spezifikation in eine formale Form überführt werden muss.

Zur Simulation von Use Case Verhalten werden verschiedene Arten von formalen Ausführungsmodellen verwendet. Am gebräuchlichsten sind Zustandsautomaten [JW07, Som05, Gli95, GLST01, KP05]. Es werden aber auch andere Formalismen verwendet. Amyot und Logrippo schlagen beispielsweise einen Ansatz auf Basis von LOTOS [AL00] vor. Mansurov und Zhukov verwenden MSC [MZ99].

Erwähnenswert ist weiterhin ADORA [SMG04]. ADORA enthält eine geschlossene formale Modellierungsnotation für Struktur und Verhalten von Softwaresystemen und einen simulationsgetriebenen Modellierungsansatz. ADORA ermöglicht die inkrementelle Verfeinerung von Verhalten und die Simulation von partiellen Lösungen durch den Einsatz von sogenannten Stubs und Drivern.

Allen diesen Ansätzen ist gemein, dass zur Visualisierung der Simulationsläufe das unterliegende formale Modell präsentiert wird. Nachteil dieser Präsentationsform ist, dass sie für wichtige Zielgruppen wie Klienten unverständlich ist, da diese die verwendeten Formalismen i.d.R. nicht kennen. Deshalb sind Use Case Simulationen als primäres Prototypingwerkzeug ungeeignet. Dennoch können die beschriebenen Ansätze zur Verbesserung der Anforderungsqualität beitragen, da sie Experten wichtige Hinweise auf Inkonsistenzen und Probleme im Ablauf des beschriebenen Verhaltens geben.

4.4 Mit Nutzerschnittstellen angereichertes Use Case Prototyping

Bei den bisher vorgestellten Ansätzen wird auf die Modellierung der Nutzerschnittstelle vollkommen verzichtet. Bomsdorf und Sinnig [BS09, SMK10] oder Mizouni [MSK10] schlagen den Einsatz von Task Modellen zur Beschreibung der Nutzerschnittstelle in einem Use-Case-basierten Simulationsansatz vor. Dabei werden knappe Use Case Beschreibungen durch Task Modelle verfeinert. Diese können dann in einer Simulationsumgebung analysiert werden. In beiden Ansätzen stehen allerdings Konformitätsprüfungen zwischen Task Modell und den Use Cases im Vordergrund.

Weiterhin werden verschiedene Ansätze vorgeschlagen, wie UML Modelle um Nutzerschnittstelleninformationen angereichert werden können. Phillips et al. beschreiben eine Erweiterung des UML Metamodells um abstrakte UI-Elemente [PK01]. Diese ermöglicht zwar die Kombination von UI Skizzen und Use Cases, sie bietet aber keine formale Ausführungssemantik. Deshalb ist der Ansatz für die Generierung von ausführbaren Prototypen ungeeignet.

Da Silva & Patton benutzen Standard UML-Modelle, insbesondere Objektdiagramme, zur Spezifikation eines Softwaresystems inklusive seiner Nutzerschnittstelle [dSP00]. Die entstehenden Modelle sind formal genug, um ausgeführt zu werden, diese Idee wird aber von den Autoren nicht weiter verfolgt.

4.5 Visuelle Use Case Prototypen

Verschiedene Ansätze verwenden eine Kombination von Nutzerschnittstellenprototypen und Use Cases zur Erzeugung von Prototypen. Diese visuellen Prototypen haben den Vorteil, dass sie für viele Projektbeteiligte deutlich leichter verständlich sind als Prototypen, die keine Darstellung der Nutzerschnittstelle anbieten. Am verbreitetsten sind Ansätze, bei denen Schritten im Use Case Ablauf Nutzerschnittstellen zugeordnet werden. Nawrocki [NO05] beschreibt beispielsweise einen solchen Ansatz. Dieser erlaubt es Use Cases mit UI-Skizzen zu assoziieren und dann html-Storyboards abzuleiten. Ähnliche Funktionalitäten bieten auch einige kommerzielle Use Case Werkzeuge, wie z.B. CaseComplete [Cas]. Allen diesen Ansätzen ist gemein, dass sie lediglich statische Nutzerschnittstellen anzeigen können. Sie sind nicht in der Lage, funktionale Nutzerschnittstellenprototypen zu erzeugen.

4.5.1 Verwandte Ansätze

In diesem Abschnitt werden artverwandte Ansätze beschrieben. Sie werden kurz charakterisiert und anschließend vom hier vorgestellten Ansatz abgegrenzt. Es wurden lediglich Ansätze ausgewählt, die funktionale Nutzerschnittstellenprototypen in iterativen, inkrementellen Use-Case-basierten Vorgehensmodellen prozessbegleitend zur Verbesserung der Analyseergebnisse einsetzen.

4.5.1.1 Virtual Windows

Laurensen und Harning beschreiben einen Ansatz, der Use Cases, Datenmodelle und Nutzerschnittstellen-Skizzen integriert. Zusätzlich enthält er ein phasenbasiertes, methodisches Vorgehen [LH01]. Primäres Ziel des Ansatzes ist die Entwicklung einer geeigneten Nutzerschnittstelle, die möglichst wenige verschiedene Darstellungen für Informationen enthält. Dazu werden zunächst Tasks, das sind Use Case Kurzbeschreibungen, und ein Datenmodell entwickelt. Anschließend werden sogenannte *virtuelle Fenster* entworfen, die ausschließlich die dargestellten Daten, aber keine Interaktionskomponenten enthalten. Anschließend werden diese virtuellen Fenster mit den Tasks verbunden und zu *physischen Fenstern* verfeinert, die die fehlenden Interaktionskomponenten enthalten. Auch Laurensen und Harning empfehlen den Einsatz von Prototypen. Allerdings sieht der Virtual Windows Ansatz keinen Einsatz von codierten Prototypen vor. Stattdessen werden Storyboards bzw. Papier-Prototypen verwendet.

Abgrenzung: Der Virtual Windows Ansatz zielt primär auf die Modellierung von konsistenten, benutzbaren Nutzerschnittstellen ab. Er enthält weder eine formale Notation noch eine Werkzeugunterstützung. Dennoch ist das be-

schriebene Vorgehen dem hier vorgeschlagenen sehr ähnlich. Insbesondere die Idee der virtuellen Fenster wurde in dieser Arbeit aufgegriffen.

4.5.1.2 Executable Use Cases

Jorgenson [Jor04] beschreibt einen simulationsgetriebenen Entwicklungsansatz. Use-Case-basierte Spezifikationen werden in drei aufeinander folgenden Phasen (Tier) spezifiziert. Zuerst werden sie verbal beschrieben (Tier 1). Anschließend wird die Spezifikation manuell in ein formales Modell überführt (Tier 2). Jorgenson schlägt hierzu gefärbte Petrinetze vor (vgl. Kap. 8.1.2). Anschließend wird diese Spezifikation automatisch in einen Prototypen übertragen, der mit UI-Informationen angereichert ist (Tier 3). Dieser kann dann zur Validierung des Verhaltens eingesetzt werden.

Abgrenzung: Im Unterschied zum in dieser Arbeit präsentierten Ansatz ist bei Executable Use Cases eine manuelle Transformation von textuellen Use Cases in ein formales Modell notwendig. Dies verursacht einen erheblichen Mehraufwand bei der Modellierung. Hier bezweifelt selbst der Autor, ob dieser rentabel ist.

4.5.1.3 Play In/Play Out

Play In/Play Out ist ein simulationsgetriebener Ansatz zur Analyse von Systemverhalten. In diesem von Harel und Marelly [HM03] vorgestellten Ansatz wird zunächst ein UI-Design entwickelt. Anschließend wird das Verhalten des modellierten Systems nicht spezifiziert, sondern in einer speziellen Umgebung aufgezeichnet (Play In). Dazu werden verschiedene Szenarien aufgenommen, indem ein Nutzer Eingaben und erwartete Ausgaben auf der Nutzerschnittstelle emuliert. Parallel erzeugt die Umgebung jeweils ein LSC (vgl. Kap. 2.4.4), das die durchgeführten Abläufe enthält.

Die Validierung des beschriebenen Verhaltens wird dann ebenfalls in einer Simulationsumgebung durchgeführt (Play out). Dazu werden beliebige Interaktionssequenzen durchgespielt und geprüft, ob das simulierte Verhalten den Nutzeranforderungen entspricht.

Abgrenzung: Der Play in/Play out Ansatz unterscheidet sich in zwei Aspekten deutlich vom hier vorgestellten. Erstens verzichtet er auf die Modellierung einer detaillierten textuellen Anforderungsspezifikation. Stattdessen wird die Spezifikation in Form von LSC dokumentiert. Zweitens ist die Modellierung der Nutzerschnittstelle nicht Teil des Ansatzes. Play In/Play Out nimmt an, dass bereits eine vollständige Nutzerschnittstelle existiert, die zur Steuerung des Verhaltens verwendet werden kann.

4.5.1.4 SCORE+

SCORE+ [HSW02] ist ein prototypgetriebener Ansatz zur Anforderungsmodellierung. Er kombiniert Use Cases, Domänenmodelle und UI-Prototypen. Dazu integriert SCORE+ SCORE, einen Ansatz zur Modellierung von Use Cases und Klassen, mit der FLUID UI-Modellierungsumgebung. In SCORE+ werden Use Cases und Domänenmodelle UML-konform mit Aktivitäts- und Klassendiagrammen modelliert. Die Nutzerschnittstelle wird mit abstrakten UI-Elementen der FLUID Methodik dargestellt. Zusätzlich werden Objektkollaborationen zur Modellierung von Systemverhalten verwendet. Weiterhin bietet SCORE+ die Möglichkeit, Abhängigkeiten zwischen den verschiedenen Modellierungselementen zu spezifizieren. Dadurch wird es möglich, funktionale, visuelle Prototypen automatisch aus der Spezifikation abzuleiten und Ergebnisse aus der Prototypenanalyse in die Spezifikation zurückzuspiegeln (Round Trip Prototyping).

Abgrenzung: SCORE+ ist dem hier vorgestellten Ansatz ähnlich. Er unterscheidet sich aber in zwei wichtigen Aspekten deutlich. SCORE+ basiert im Unterschied zum hier vorgestellten Ansatz nicht auf textuellen Use Case Beschreibungen, stattdessen werden spezielle Aktivitätsdiagramme verwendet. Außerdem verwendet SCORE+ zwei verschiedene Spezifikationsnotationen: eine Use-Case-basierte Grundnotation und UIDs als Detailmodellierungsnotation zur Vorbereitung des Prototypings. Da die UIDs manuell angepasst werden müssen, besteht hier wieder die Gefahr von Inkonsistenzen.

4.6 Diskussion

Zusammenfassend lassen sich zwei verschiedene Ansätze zur Erzeugung von Prototypen unterscheiden. Entweder wird eine formale Notation zur Spezifikation der Use Cases eingesetzt, die gleichzeitig auch als Prototypingnotation dient, oder der Prototyp wird semi- oder voll-automatisch aus einer semiformalen Notation abgeleitet.

Eine Anreicherung der Prototypen mit Nutzerschnittstelleninformationen verbessert die Zugänglichkeit des Prototypen deutlich und ermöglicht es auch Projektbeteiligten, wie Nutzern, Prototypen zu bewerten.

In den meisten vorgestellten Ansätzen bildet das Prototyping einen zusätzlichen Analyseschritt. Ziel des Prototypings ist primär die Validierung vollständiger detaillierter Use-Case-zentrierter Spezifikationen. Dazu wird zunächst ein Prototyp aus der Spezifikation abgeleitet. Anschließend werden die im Use Case enthaltenen Szenarien einzeln ausgeführt und auf Korrektheit geprüft. Solche Ansätze eignen sich nicht für ein prozessbegleitendes Prototyping, da Prototypen nur auf Grundlage einer detaillierten Spezifikation erzeugt werden können.

Iterative, inkrementelle Ansätze ermöglichen es, Prototyping prozessbegleitend während der Anforderungsanalyse gemeinsam mit einer Use-Case-zentrierten Spezifikation zu entwickeln und zu bewerten. Hier kann der Prototyp automatisch aus einer Use-Case-zentrierten Spezifikation erzeugt werden und Änderungen am Prototyp, die aus der Prototypbewertung entstehen, können automatisch in die Use-Case-zentrierte Spezifikation reflektiert werden. Dazu ist eine eingebettete Werkzeuglandschaft aus Spezifikations-, Prototyping- und Bewertungsumgebung notwendig. Außerdem müssen die Prototypingaktivitäten in ein strukturiertes Analysevorgehen integriert sein.

Allerdings bietet keiner der aktuellen Ansätze ein solches werkzeuggestütztes methodisches Vorgehen zum Einsatz von Prototyping in der Use-Case-zentrierten Anforderungsanalyse, das schon ab frühen Phasen prozessbegleitend eingesetzt werden kann.

5 Anforderungen an einen Use-Case-zentrierten Prototypingansatz

Wie in den vorherigen Abschnitten diskutiert, birgt der Einsatz von Prototypingtechniken große Potentiale zur Qualitätsverbesserung von Use-Case-zentrierten Anforderungsspezifikationen. Dazu ist aber ein geeigneter Modellierungsansatz notwendig. Dieser muss es ermöglichen

- die benötigten Modelle der Use-Case-zentrierten Anforderungsanalyse zu entwickeln,
- aus Use-Case-zentrierten Anforderungsspezifikationen effizient und effektiv geeignete Prototypen abzuleiten,
- aus der Bewertung von Prototypen wichtige Erkenntnisse zur Qualitätsverbesserung und Weiterentwicklung der Use-Case-zentrierten Anforderungsspezifikation zu gewinnen.

Im Folgenden sollen nun Anforderungen an einen solchen Ansatz diskutiert werden. Dazu werden zunächst wichtige Rahmenbedingungen festgelegt und zentrale Vorüberlegungen erörtert. Anschließend werden die Einsatzszenarien für den Prototypingansatz in der Use-Case-zentrierten Analyse vorgestellt und die Anforderungen an eine Werkzeugunterstützung für den Ansatz diskutiert.

5.1 Annahmen & Rahmenbedingungen

Ein Use-Case-zentriertes Vorgehen impliziert eine Reihe grundsätzlicher Rahmenbedingungen an Prototypingansätze, die parallel mit diesem Vorgehen in der Anforderungsanalyse eingesetzt werden sollen. Diese Rahmenbedingungen sollen nun diskutiert werden, bevor genauer auf die Anforderungen eines Use-Case-zentrierten Prototypingansatzes eingegangen wird.

Use-Case-zentriertes Vorgehensmodell: Iterative, inkrementelle, phasenorientierte Vorgehensmodelle sind seit vielen Jahren etablierte Praxis in der Use-Case-zentrierten Analyse. Sie haben sich in vielen Projekten bewährt und werden von allen gängigen Lehrbüchern empfohlen. Der in diesen Vorgehensmodellen angewendete Breadth-First-Ansatz hilft, die Projektrisiken strukturiert der Reihe nach zu adressieren, und wird von Analysten und Klienten als in-

tuitives Vorgehen akzeptiert. Deshalb muss ein geeigneter Prototypingansatz ebenfalls in ein solches Vorgehensmodell einbettbar sein. Wir setzen also für diese Arbeit ein iteratives, inkrementelles, phasenorientiertes Vorgehen bei der Anforderungsanalyse voraus.

Textuelles Spezifikationsdokument: In vielen Entwicklungsprojekten bildet eine textuelle Anforderungsspezifikation das Ergebnis der Anforderungsanalyse und eine wichtige Vertragsgrundlage zwischen Softwarehersteller und Kunden. Diese textuelle Anforderungsspezifikation enthält typischerweise weder den Prototyp noch die Bewertung des Prototyps. Deshalb müssen alle Ergebnisse aus dem Prototyping am Ende der Anforderungsanalyse in die textuelle Anforderungsspezifikation eingepflegt werden. Es ist also nötig, Ergebnisse aus dem Prototyping in die Use-Case-zentrierte Spezifikation zu übertragen.

Entwicklung von interaktiven Informationssystemen: Interaktive Informationssysteme sind heutzutage in allen Bereichen der Wirtschaft ein wichtiger Treiber für Innovation und die Optimierung von betrieblichen Abläufen. Sie spielen spätestens seit Verbreitung des Internets auch im privaten Bereich eine zentrale Rolle. Der hier beschriebene Ansatz soll insbesondere die Anforderungsanalyse dieser Systeme unterstützen. Deshalb muss er zusätzlich zur Beschreibung der Systemfunktionen und Abläufe in der Lage sein, die Nutzerschnittstelle des Systems zu modellieren. Diese beeinflusst bei interaktiven Systemen die Brauchbarkeit. Außerdem ist sie oft das zentrale Kommunikationsmittel in der Analyse, da Klienten i.d.R. ihre Anforderungen an das System am besten über die Beschreibung der Nutzerschnittstelle kommunizieren können.

Fokus auf funktionale Anforderungen: Primäres Ziel des hier vorgestellten Ansatzes ist es, die Analyse von funktionalen Anforderungen zu unterstützen. Deshalb werden beim Prototyping, abgesehen von den Anforderungen an die Nutzerschnittstelle, keine weiteren nicht-funktionalen Anforderungen betrachtet. Insbesondere Anforderungen an das Leistungsverhalten und die Datensicherheit sind nicht Teil des Ansatzes und spielen daher keine Rolle.

Wegwerfprototypen: Ziel dieses Ansatzes ist es, durch den Einsatz explorativen Prototypings fehlende Anforderungen aufzudecken und Unklarheiten auszuräumen. Dazu reicht es aus, die Außensicht des Systems zu modellieren. Eine saubere Definition der Systemarchitektur und der internen Datenrepräsentation ist für diesen Einsatzzweck nicht nötig. Solche Prototypen eignen sich nicht für die Weiterentwicklung in ein Produktivsystem, da die zu Grunde liegende Architektur unzureichend ist. Deshalb gehen wir davon aus, dass im vorgestellten Ansatz Wegwerfprototypen entwickelt werden.

Durchgehender Einsatz von Prototypen: Prototypen können bereits in frühen Phasen der Anforderungsanalyse wichtige Impulse geben und zur Beseitigung von Unklarheiten beitragen. Deshalb soll der entwickelte Ansatz den Einsatz von Prototypen prozessbegleitend und so früh wie möglich in der Anforderungsanalyse ermöglichen. Wie bereits in der Motivation diskutiert, muss er deshalb verschiedene Arten von Prototypen unterstützen und die sukzessive Weiterentwicklung von Prototypen ermöglichen.

5.2 Vorüberlegungen

Das Hauptproblem beim Einsatz von Prototypen in der Use-Case-zentrierten Analyse ist die mangelnde Integration der verschiedenen verwendeten Notationen. Diskontinuität zwischen der Spezifikations- und der Prototypingnotation als auch zwischen den Prototypingnotationen für verschiedene Prototyparten führt zu manuellem Aufwand und ist eine Quelle für Inkonsistenzen und Modellierungsfehler.

Diesem Problem kann man konzeptionell nur begegnen, indem man die Spezifikationsnotation und die Prototypingnotation eng verzahnt. Dies ist prinzipiell auf zwei Arten möglich:

1. **Vereinheitlichter Ansatz:** Beim vereinheitlichten Ansatz wird dieselbe Notation zur Beschreibung des Ausgangsmodells und als Prototypingnotation verwendet.

2. **Generativer Ansatz:** Im generativen Ansatz verwendet man eine Spezifikationsnotation, die es erlaubt, Prototypen automatisch aus Spezifikationsdokumenten zu erzeugen.

Ein vereinheitlichter Ansatz, wie ihn beispielsweise ADORA [Joo00] verfolgt, kommt für die hier adressierte Problemstellung nicht in Frage. Eine der zentralen Voraussetzungen ist die Verwendung einer texuellen Use-Case-zentrierten Notation als Spezifikationsnotation für den Prototypingansatz. Use-Case-zentrierte Spezifikationen eignen sich aber nicht als Prototypingnotation, da es Probleme gibt, die Ablaufsemantik abzubilden (vgl. [Leh08]). Deshalb wird beim Prototyping in der Use-Case-zentrierten Analyse ein generativer Ansatz verwendet. Im Folgenden wird also von einem generativen Prototypingansatz ausgegangen.

Offensichtlich ist ein solcher Ansatz nur mit einer geeigneten Werkzeugunterstützung praktikabel. Diese muss die Spezifikation von Use-Case-zentrierten Anforderungsspezifikationen, die Generierung von Prototypen sowie deren Bewertung unterstützen.

5.3 Anforderungen an einen Prototypingansatz

Zur Analyse der Anforderungen an einen Prototypingansatz für die Use-Case-zentrierte Anforderungsanalyse werden zunächst Einsatzszenarien identifiziert, in denen Prototyping zur Unterstützung der Analyse eingesetzt werden kann. Aus diesen Einsatzszenarien werden anschließend Anforderungen an die Einsatzumgebung, die Spezifikations-, die Prototyp- und die Bewertungsnotation, sowie an die zugehörigen Werkzeuge abgeleitet.

5.3.1 Einsatzszenarien für Prototyping in der Use-Case-zentrierten Anforderungsanalyse

In einem typischen Use-Case-zentrierten Vorgehen (vgl.2.4) können Prototypen sowohl konstruktiv zur Unterstützung der Modellierung als auch analytisch zur Qualitätssicherung verwendet werden. Dabei gibt es eine Reihe verschiedener Modellierungsaktivitäten, in denen Prototyping nutzbringend eingesetzt werden kann. Im Folgenden werden diese typischen Einsatzszenarien kurz umrissen und der Nutzen von Prototyping motiviert.

E1 – Inspektionen von Use Cases: Bei Inspektionen [Lai02] wird die Qualität einer Use-Case-zentrierten Spezifikation analysiert, indem die zugehörigen Dokumente manuell geprüft werden (vgl. [AS02]). Es gibt eine Reihe verschiedener Arten, diese Inspektionen zu organisieren (vgl. [KK09]). Grundsätzlich kann Prototyping bei allen Arten von Inspektionen zur Unterstützung der Verhaltensanalyse eingesetzt werden. Dazu werden verschiedene zu analysierende Szenarien der Reihe nach in der Bewertungsumgebung ausgeführt und Auffälligkeiten notiert. Zusätzlich kann mit einem solchen Prototypingansatz die Qualität der Inspektionen selbst analysiert werden. Dazu können, ähnlich wie beim White-Box-Testen, aus den durchgeführten Simulationsläufen Überdeckungsmetriken auf dem Ablaufgraphen der Use-Case-zentrierten Spezifikation ermittelt werden.

E2 – Inkrementelle Use Case Verfeinerung: Bei der inkrementellen Verfeinerung von Use Cases werden abstrakte verbale Use Case Beschreibungen sukzessive in detaillierte, strukturierte, formalisierte Beschreibungen überführt. Dabei werden im Wesentlichen zwei Arten von Verfeinerungen durchgeführt: Entweder wird ein „abstrakter“ Use Case Schritt in mehrere detaillierte Schritte zerlegt (vgl. Bsp. 1) oder ein verbal beschriebener Schritt wird in eine formale Darstellung überführt (vgl. Bsp. 2).

```
BSP 1.
Nutzer gibt Daten ein --> Nutzer gibt Namen ein;
                          Nutzer gibt Adresse ein;

BSP 2.
Nutzer gibt Namen ein  --> nutzer.setName(<<VALUE>>);
```

Prototyping kann dabei prinzipiell in beiden Fällen eingesetzt werden. Es ist aber insbesondere für den zweiten Fall nützlich. Dazu wird ein Use Case Szenario schrittweise ausgeführt, und die enthaltenen Schritte werden der Reihe nach mit einer formalen Darstellung angereichert. Dabei wird die Formalisierung dadurch erleichtert, dass der Systemzustand bzw. die Belegung relevanter Variablen in der Simulation angezeigt wird.

E3 – Testfallerzeugung: Zur Erzeugung von Systemtests schlagen Ryser und Glinz vor [RGJ00], die Testauswahl anhand einer Analyse des Ablaufgraphs der Use-Case-zentrierten Spezifikation zu treffen. Dabei kann interaktives Prototyping eingesetzt werden, um konkrete Testfälle zu erzeugen. Dazu werden die aus der Analyse des Kontrollflusses abgeleiteten abstrakten Testszenarien (vgl. [Kil10]) einzeln, der Reihe nach in der Simulationsumgebung ausgeführt und durch Eingabe von Werten konkretisiert. Zusätzlich können beim Prototyping aufgezeichnete Traces zur Generierung von automatisierten Testfällen eingesetzt werden, falls eine Beziehung zwischen der Nutzerschnittstelle des Prototyps und der Nutzerschnittstelle des Zielsystems hergestellt werden kann (vgl. [Con12]).

E4 – Automatisierte Verhaltensanalyse: Drenger und Paech [DP04] weisen darauf hin, dass bestimmte Defekte, wie Inkonsistenzen im globalen Systemverhalten, aufgrund der dezentralen Struktur von Use Case Modellen nur extrem schwierig mit Hilfe von manuellen Inspektionen zu identifizieren sind. Hier können vollautomatische Verhaltensanalysen (vgl. 4.1) eingesetzt werden, um derartige Defekte aufzudecken. Prototypen zielen zwar nicht primär auf einen solchen Einsatz ab, vollautomatische Verhaltensanalysen lassen sich aber auch auf Basis von Prototypen ausführen, wenn die unterliegende Prototypingnotation hinreichend formal ist (vgl. 2.4.5).

E5 – Datenanalyse: Bei der Datenanalyse werden aus einer Analyse des geforderten Verhaltens geeignete Datenstrukturen zur Repräsentation der zentralen Geschäftsobjekte (Business Objects) abgeleitet. Hierbei ist zum einen die statische Analyse der Datenentitäten, die für die Prototypausführung verwendet werden, hilfreich, zum anderen können mit Hilfe von interaktiven Prototypen Effekte der Datenmodellierung auf das Verhalten untersucht werden.

E6 – Generative Entwicklung partieller Lösungen: Bei der generativen Softwareentwicklung werden Artefakte des Zielsystems aus speziellen Modellen automatisch erzeugt (vgl. [SVEH07]). Dabei können die interaktiven Prototypen als Ausgangsmodelle für die Codegenerierung dienen. Hier ist es bei starken Annahmen an die technische Infrastruktur der Lösung prinzipiell sogar möglich, komplette Applikationen zu generieren. In der Praxis ist dies aber meist nicht gewünscht, da die Qualität der Architektur der entstehenden Lösung oftmals unzureichend ist (vgl. [BP92]). Bei geschickter Strukturierung der Zielarchitektur ist es aber möglich, weite Teile des Zielsystems zu generieren (vgl. [Löw11]).

E7 – Usability Analysen: Usability Analysen werden eingesetzt, um die Eignung einer Software im Einsatz zu bewerten. Dazu führen Nutzer bestimmte Aufgaben mit Hilfe des Systems aus. Anschließend werden verschiedene Parameter zur Eignung der Software (vgl. [Nie94]), z.B. die Zeit, die ein Nutzer benötigt, um eine bestimmte Aufgabe zu lösen, beobachtet. Viele dieser Usability Analysen können auch auf den interaktiven Prototypen durchgeführt werden. Dies ermöglicht sehr frühe Usability Analysen. Dadurch können Entwicklungskosten für ungeeignete Lösungen eingespart werden.

5.3.2 Anforderungen an die Prototypingwerkzeuge

Wie bereits erwähnt, ist Prototyping nur dann effizient einsetzbar, wenn es von geeigneten Werkzeugen unterstützt wird. Diese müssen es erlauben, Prototypen zu entwickeln, anzupassen und auszuführen. Soll Prototyping prozessbegleitend während der Anforderungserhebung und -validierung eingesetzt werden, müssen die Prototypingwerkzeuge von verschiedenen Projektbeteiligten für unterschiedliche Zwecke eingesetzt werden können. Zusätzlich müssen sie die Synchronisation von Prototypen und Ausgangsdokumenten, also der Use-Case-zentrierten Spezifikationen, unterstützen. Im Einzelnen werden also an eine Prototypingumgebung die folgenden Erwartungen gestellt.

W1 – Unterstützung verschiedener Prototyparten: In typischen iterativen, inkrementellen Use-Case-zentrierten Anforderungsanalyseansätzen werden unterschiedliche Modellierungsphasen durchlaufen. Jede dieser Phasen hat einen eigenen Modellierungsfokus und behandelt spezielle Fragestellungen. Außerdem liegen die modellierten Informationen in unterschiedlichen Detaillierungsgraden vor. Soll in einem solchen Prozess Prototyping effektiv eingesetzt werden, müssen unterschiedliche Arten von Prototypen verwendet werden (vgl. [WK92]). Deshalb muss eine geeignete Prototypingumgebung in der Lage sein, verschiedene Arten von Prototypen aus Use-Case-zentrierten Spezifikationen mit unterschiedlichem Detaillierungsgrad zu erzeugen und auszuführen.

W2 – Integration mit textueller Dokumentation: Damit Prototyping eingesetzt werden kann, müssen Prototypen kosteneffizient entwickelt werden können. Prototypen müssen einfach aus einer Anforderungsspezifikation abgeleitet und weiterentwickelt werden können. Außerdem muss es möglich sein, Erkenntnisse aus dem Prototyping strukturiert in die Use-Case-zentrierte Spezifikation zurückzuspiegeln. Deshalb ist eine enge Integration zwischen der Use-Case-zentrierten Spezifikation und den ausführbaren Prototypen notwendig.

W3 – Ausführung unvollständiger Spezifikationen: Da Prototyping prozessbegeleitend und so früh wie möglich eingesetzt werden soll, müssen ausführbare Prototypen auch aus unvollständigen Use-Case-zentrierten Spezifikationen erzeugt und analysiert werden können. Dies gilt für Spezifikationen, die nicht alle Use Cases des Zielsystems enthalten, für Spezifikationen, in denen Teile des Verhaltens nur verbal beschrieben sind, sowie für Spezifikationen, die eine Kombination aus beidem aufweisen.

W4 – Adressatenspezifische Anpassung: Die Prototypingumgebung muss von verschiedenen Projektbeteiligten benutzt werden können. Hierzu zählen Anforderungsanalysten, Softwaredesigner und Tester, aber auch insbesondere Klienten. Damit diese unterschiedlichen Projektbeteiligten Prototyping effektiv einsetzen können, ist es nötig, spezielle Perspektiven auf den Prototypen bzw. das repräsentierte Spezifikationsdokument für unterschiedliche Rollen bereitzustellen. Außerdem muss es insbesondere möglich sein, die Prototypingumgebung ohne Kenntnis unterliegender formaler Ausführungsmodelle zu benutzen.

W5 – Integration mit anderen Modellierungswerkzeugen: Während der Anforderungsentwicklung werden Use Cases i.d.R. parallel mit einer Reihe verschiedener anderer Anforderungsdokumente, wie z.B. Domänenmodellen, Business-Regeln oder technischen Anforderungen, entwickelt. Zusätzlich werden sie später im Entwicklungsprozess für weitere Zwecke wie Testen oder Architekturdesign eingesetzt. Konsequenterweise muss eine geeignete Prototypingumgebung die Einbindung von Modellierungswerkzeugen und Modellen für spezielle Entwicklungsaufgaben in allen Phasen des Entwicklungsprozesses ermöglichen.

W7 – Simulation des globalen Systemverhaltens: Wie bereits erwähnt, sind Defekte im globalen Systemverhalten mit manuellen Inspektionsverfahren nur schwer aufzudecken. Deshalb sollte die Prototypingumgebung in der Lage sein, das Gesamtverhalten eines Systems zu simulieren. Dazu muss es möglich sein, mehrere Use Cases hintereinander auszuführen und dabei den Systemzustand kontinuierlich weiterzuentwickeln.

W8 – Wiederverwendung von Simulationsdaten: Simulationsdaten wie z.B. Traces, die aus Simulationsläufen entstehen, können in anderen Modellierungstätigkeiten nutzbringend eingesetzt werden. Sie können z.B. für statische Analysen, Dokumentation von Szenarien oder zum Testentwurf dienen. Deshalb sollte es möglich sein, Simulationsläufe aufzuzeichnen.

Außerdem sollte die Prototypingumgebung aufgezeichnete Simulationsdaten abspielen können, beispielsweise um diese zur Präsentation bestimmter Abläufe in Kundengesprächen zu verwenden.

W9 – Analyse von Traces: Die Prototypingumgebung sollte es ermöglichen, statische Analysen auf ausgeführten Traces durchzuführen. Dies schließt insbesondere drei Arten von Analysen ein:

- Quantitative Messungen, z.B. Überdeckungsmessungen zur Bewertung der Analysequalität
- Strukturanalysen des Ausführungsmodells, z.B. zur Identifikation von Inkonsistenzen oder zur Analyse der Auswirkungen von Änderungen im Ablauf
- Revalidierung von Traces nach Änderungen am spezifizierten Verhalten

5.3.3 Anforderungen an die Spezifikationsnotation

Eine geeignete Spezifikationsnotation sollte den Entwickler darin unterstützen, eine leicht verständliche, kompakte, vollständige und korrekte Spezifikation zu erstellen. Außerdem muss sie im hier vorgestellten Ansatz formal genug sein, damit sich Prototypen ableiten lassen. Im Kontext dieser Arbeit sollte sie dazu die folgenden Eigenschaften aufweisen:

S1 – Ausdrucksstärke: Die Spezifikationsnotation muss in der Lage sein, alle relevanten Konzepte auszudrücken. Dies bedeutet, dass die Spezifikationsnotation Use Cases, Daten, Funktionen und Nutzerschnittstellen modellieren können muss.

S2 – Simplizität: Die Spezifikationsnotation muss benutzerfreundlich sein. Sie sollte leicht zu erlernen und anzuwenden sein. Dazu müssen die modellierten Konzepte auf eine intuitive Art in der Spezifikationsnotation abgebildet werden können.

S3 – Knappheit: Die Spezifikationsnotation sollte eine kompakte Beschreibung der modellierten Zusammenhänge ermöglichen. Es sollte möglich sein, die

zentralen Modellierungselemente knapp darzustellen. Außerdem sollte die redundante Beschreibung von Aspekten vermieden werden.

S4 – Strukturiertheit: Eine Einteilung der spezifizierten Informationen in klar abgrenzbare Bereiche sollte gefördert werden. Die Spezifikationsnotation sollte sowohl eine Strukturierung nach verschiedenen Aspekten der Anforderungsmodellierung als auch nach strukturellen Lösungsaspekten wie Entwurfseinheiten des Zielsystems ermöglichen.

S5 – Unterstützung inkrementeller Entwicklung: Die Spezifikationsnotation sollte es ermöglichen, Informationen auf unterschiedlichen Detaillierungsgraden zu beschreiben. Zusätzlich sollten die unterschiedlichen Aspekte der Anforderungsmodellierung sukzessive entwickelt und schrittweise verfeinert werden können.

S6 – Formalität: Soll die Spezifikationsnotation zur Generierung von Prototypen effektiv einsetzbar sein, muss sie formal genug sein. Es muss möglich sein, die für die Ableitung eines Prototyps benötigten Informationen eindeutig aus Spezifikationen abzuleiten. Dazu müssen Syntax und Semantik der Spezifikationsnotation klar definiert sein.

S7 – Erweiterbarkeit: Die konzeptionelle Erweiterbarkeit der Spezifikationsnotation ist wünschenswert. Es sollte sowohl möglich sein, neue Modellierungselemente einzuführen als auch existierende Modellierungselemente zu erweitern.

5.3.4 Anforderungen an die Prototypingnotation

Eine geeignete Prototypingnotation muss die technischen Grundlagen für den Prototypeinsatz bereitstellen. In unserem Fall muss sie in der Lage sein, verschiedene Arten von Prototypen zu beschreiben. Sie muss also die folgenden Anforderungen erfüllen.

P1 – Ausdrucksstärke: Wie auch die Spezifikationsnotation muss die Prototypingnotation in der Lage sein, alle relevanten Konzepte auszudrücken. Genauer gesagt, es müssen alle im Use Case Modell möglichen Arten von Szenarien abbildbar sein. Hierzu muss die Prototypingnotation den modellierten Kontroll- und Datenfluss exakt darstellen können.

P2 – Endliche, skalierende Struktur: Die Prototypingumgebung soll in Industrieprojekten einsetzbar sein. Deshalb muss die Struktur der Prototypingnotation so gewählt sein, dass alle syntaktisch korrekten Use Case Spezifikationen in einem endlichen Prototypen korrekt abgebildet werden können. Dies muss auch dann gewährleistet sein, wenn das spezifizierte Verhalten Schleifen oder rekursive Ausdrücke enthält. Zusätzlich müssen die Größe des Prototypmodells, sowie die Zeit- und Platzkomplexität des Erzeugungsalgorithmus für reale Anwendungen skalieren.

P3 – Komposition: Das Prototypingmetamodell sollte Mechanismen anbieten, um verschiedene Teilmodelle zu einem Gesamtmodell zu integrieren bzw. ein Gesamtmodell in mehrere Teilmodelle zu dekomponieren. Idealerweise sollte die Komposition durchführbar sein, ohne Teilmodelle in das Gesamtmodell zu kopieren.

P4 – Robuste Topologie: Wird die Prototypingumgebung prozessbegleitend und schon ab frühen Phasen der Anforderungsanalyse benutzt, so sind Änderungen an den Prototypmodellen zu erwarten. Dies kann Einfluss auf bereits durchgeführte Analysen und Traces haben. Damit in einem solchen Umfeld effektives Prototyping möglich ist, sollten hiervon so wenig Traces wie möglich betroffen sein. Deshalb muss die Topologie des Prototypmodells robust gegen Änderungen in den Spezifikationsmodellen sein. D.h., Änderungen an der Struktur des Spezifikationsmodells, z.B. an Bedingungen oder der Ablaufreihenfolge der Szenarien, dürfen nur lokale Effekte auf die Struktur des Prototypmodells haben.

5.3.5 Anforderungen an die Bewertungsnotation

Eine geeignete Bewertungsnotation muss die Möglichkeit bieten, Ergebnisse aus dem Prototypeneinsatz effektiv zu dokumentieren und effizient zur Verbesserung der Prototypen und Spezifikationsdokumente einzusetzen. In unserem Fall muss sie insbesondere in der Lage sein, Ergebnisse mit der Use-Case-zentrierten Spezifikation zu verknüpfen. Sie muss also die folgenden Anforderungen erfüllen:

B1 – Verbindung mit der Spezifikationsnotation: Ergebnisse des Prototypeinsatzes müssen in die Use-Case-zentrierte Spezifikationsnotation übertragen werden können. Dazu muss eine direkte Verbindung zwischen der Bewertungsnotation und der Spezifikationsnotation bestehen.

B2 – Dokumentation des Simulationsablaufs: Aufgezeichnete Traces können wichtige Hinweise zur Beseitigung von Fehlern in den Spezifikationsdoku-

menten geben. Sie können zur inkrementellen Verfeinerung der Spezifikation und zur Bewertung der Qualität von Analyseläufen herangezogen werden. Deshalb sollte die Bewertungsnotation ein Konzept enthalten, um diese Traces zu beschreiben.

6 CTPlaner ein Beispielsystem

Im folgenden Kapitel wird eine Use-Case-zentrierte Spezifikation eines Beispielsystems vorgestellt. Diese wird im weiteren Verlauf der Arbeit verwendet, um die Konzepte, Metamodelle und Werkzeuge des Ansatzes zu verdeutlichen.

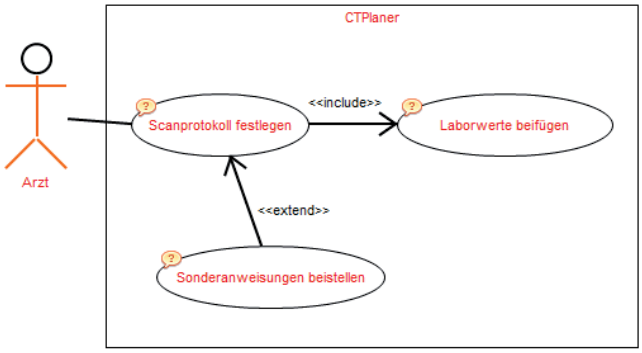


Abbildung 6.1: CTPlaner – Use Case Diagramm

Das Anwendungsbeispiel *CTPlaner* beschreibt einen stark vereinfachten, angepassten Ausschnitt aus der Projektstudie CTSherpa, einem Informationssystem für die Durchführung von CT-Scans im UK Aachen (vgl. Kap. 12). Beschrieben wird das Festlegen von Scanprotokollen. In einem Scanprotokoll definiert ein Radiologe Art und Ablauf von CT Untersuchungen. Hierzu wählt er relevante Untersuchungsparameter, wie die Lage des Patienten im CT, Art und Menge des gegebenen Kontrastmittels und die Rekonstruktionen, die aus den aufgenommenen Rohdaten berechnet werden sollen. Zusätzlich müssen relevante Laborwerte des Patienten beigestellt werden. Außerdem kann der Radiologe spezielle Sonderanweisungen zur Behandlung des Patienten angeben.

Das dargestellte Verhalten in diesem Beispiel wurde für die Präsentation der Konzepte von SUPrA angepasst. Es entspricht demnach nicht immer dem analysierten Verhalten der Projektstudie CTSherpa.

Abbildung 6.1 zeigt ein Use Case Diagramm von CTPlaner. Es enthält die drei Use Cases `Scanprotokoll festlegen`, `Laborwerte beifügen` und `Sonderanweisungen beistellen`. Die Abläufe der Use Cases sind in den Abbildungen 6.2 bis 6.4 dargestellt.

		Author:	**VHoff**
ID	0001		
Titel	Laborwerte beifügen		
Primary Actor	Arzt		
	...		
Flow 0001.01	Laborwerte beifügen - Hauptfluss		
Contexts	0001.01.C01 Interaktion mit Arzt.		
Events	Action	Das System zeigt eine Liste aller verfügbaren Laborwerte an. table.show(labValues);	
	Action	Der Arzt wählt alle relevanten Laborwerte aus. selected = table.selection();	
	Extension	Laborwerte fehlen --> Fehlende Laborwerte	
	Action	Der Arzt bestätigt seine Eingaben. button.click();	
Flow 0001.02	Laborwerte manuell einstellen		
Contexts	0001.02.C01 Extension Fehlende Laborwerte: wichtige Laborwerte fehlen if(!selected.containsAll(expected));		
Events	Action	Der Arzt stellt die fehlenden Laborwerte manuell ein. selected+= manualList;	

Abbildung 6.2: CTPlaner – Laborwerte beifügen (incl. formaler Darstellung)

Abbildung 6.5 stellt die Nutzerschnittstelle von CTPlaner dar. Sie besteht aus einer Übersichtsseite, die anstehende Untersuchungen zeigt, sowie einer Detailseite zur Festlegung des Scanprotokolls und einer Seite zur Beschreibung von Sonderanweisungen. Das Domänenmodell des Systems ist in Abbildung 6.6 dargestellt. Die Systemfunktionen, die für das Prototyping definiert wurden, stellt Abbildung 6.7 dar.

		Author:	**VHoff**
ID	0002		
Titel	Scanprotokoll festlegen		
Primary Actor	Arzt		
	...		
Flow 0002.01	Scanprotokoll festlegen - Hauptfluss		
Contexts	0002.01.C01 Interaktion mit Arzt.		
Events	TryBlock Werte einstellen	Action	Das System zeigt die Stammdaten des Patienten an.
		Action	Das System zeigt die Laborwerte an.
		Action	Der Arzt gibt die Patientenlage an.
		Loop	while (Der Arzt möchte eine Rekonstruktion hinzufügen) Action: Der Arzt wählt eine Rekonstruktion.
		Action	Der Arzt wählt Art und Menge des Kontrastmittels.
		Extension	if (Arzt will Sonderanweisungen einstellen) --> Sonderanweisungen beistellen
	Inclusion	--> Laborwerte beifügen	
	Action	Der Arzt gibt das Scanprotokoll frei.	
Flow 0002.02	Arzt bricht ab.		
Contexts	0002.02.C01 Exception Arzt wählt Abbruch: at Werte einstellen if (Der Arzt wählt abbrechen) return if (Der Arzt bestätigt abbrechen nicht) Alternative End: Das Scanprotokoll wird nicht freigegeben. Alle Eingaben werden gespeichert.		
Events	...		

Abbildung 6.3: CTPlaner – Scanprotokoll festlegen

		Author:	**VHoff**
ID	0003		
Titel	Sonderanweisungen beistellen		
Primary Actor	Arzt		
	...		
Flow 0003.01	Sonderanweisungen beistellen - Hauptfluss		
Contexts	0003.01.C01 Extension Sonderanweiungen einstellen: at Sonderanweisungen hinzufügen if (Arzt will Sonderanweisungen einstellen)		
Events	Action	Das System zeigt eine Liste aller verfügbaren Anmerkungen.	
	Action	Der Arzt wählt eine Reihe von Anmerkungen aus.	
	Action	Der Arzt bestätigt seine Eingaben.	
	Action	Das System fügt der Untersuchung die Sonderanweiungen hinzu.	

Abbildung 6.4: CTPlaner – Sonderanweisungen beistellen

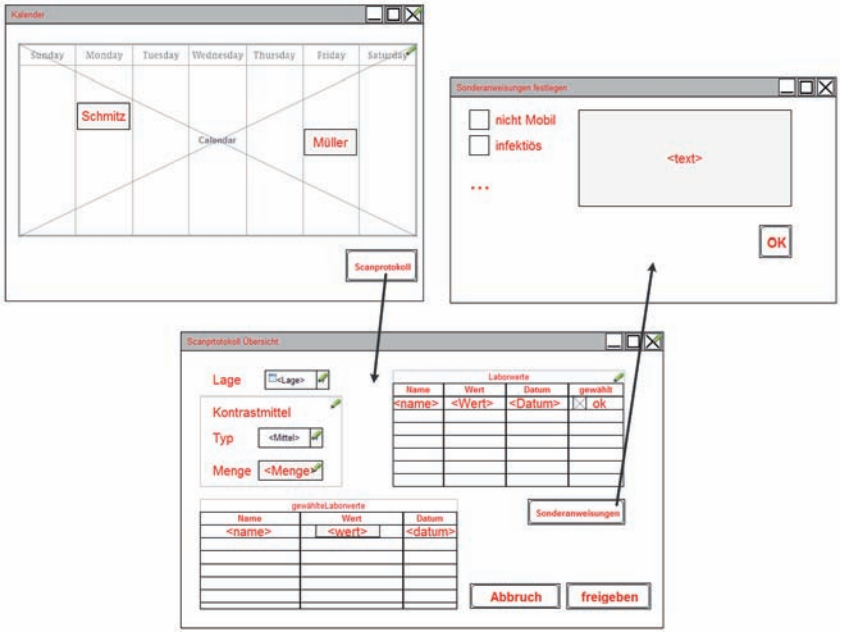


Abbildung 6.5: CTPlaner – Nutzerschnittstelle

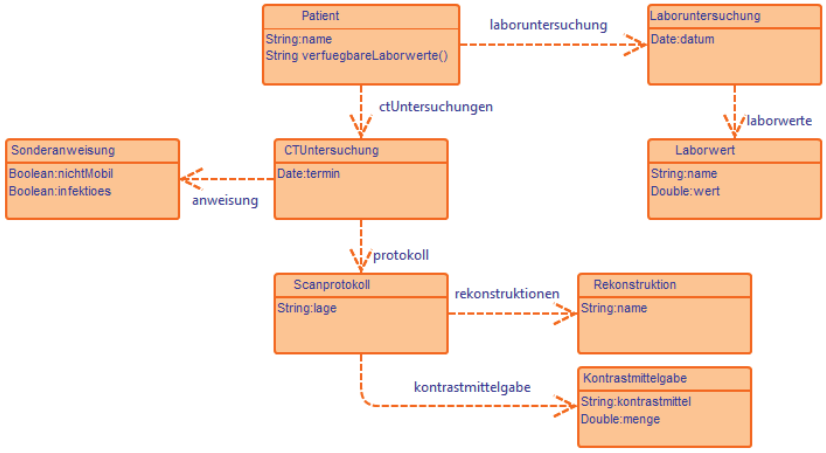


Abbildung 6.6: CTPlaner – Domänenmodell

```
App DataSource {
 // Systemvariablen

 // Basisvariablen: Müssen vor der Ausführung initialisiert werden
 var unteruchungen : java.util.List <ctplaner.CTUntersuchung>;
 var gefoderteLaborwerte : java.util.List <String>;

 // Laufzeitvariablen: Speichern den Systemzustand
 var aktuellesScanprotokoll : ctplaner.Scanprotokoll;
 var gewaehlteUntersuchung : ctplaner.CTUntersuchung;
 var gewaehlteLaborwerte : java.util.List <ctplaner.Laborwert>;
 var selectedLaborwert : ctplaner.Laborwert;

 //Prüft, ob dass Scanprotokoll vollständig ist und freigegeben werden kann
 funct protokollVollstaendig( ) : Boolean {
   aktuellesScanprotokoll.lage != null &&
   aktuellesScanprotokoll.kontrastmittelgabe != null;
 }

 // Fügt dem gewählten Scanprotokoll einen Laborwert hinzu
 funct laborwertWaehlen(ctplaner.Laborwert labv) : void {
   gewaehlteLaborwerte.add(labv);
 }

 // Entfernt einen Laborwert aus dem gewählten Scanprotokoll
 funct laborwertAbwaehlen(ctplaner.Laborwert labv) : void {
   gewaehlteLaborwerte.remove(labv);
 }

 // Hilfsfunktion, die alle verfügbaren Laborwerte für einen Patienten sammelt
 funct alleVerfuegbarenWerte(ctplaner.CTUntersuchung untersuchung) :
 java.util.List <String> {
   val a = new java.util.ArrayList<String>( );
   for(ctplaner.Laboruntersuchung lab : untersuchung.patient.laboruntersuchungen) {
     for (ctplaner.Laborwert lv : lab.laborwerte) {
      if (! a.contains(lv.name)) {
        a.add(lv.name);
      }
     }
   }
   return a ;
 }

}
```

Abbildung 6.7: CTPlaner – Systemfunktionen

7 Spezifikation von Prototypen

Für den Einsatz von Prototyping in der Use-Case-zentrierten Analyse wird zunächst eine geeignete Spezifikationsnotation benötigt. Im Folgenden stellen wir UCSM, ein Metamodell für Use-Case-zentrierte Spezifikationen, vor. Parallel zu den einzelnen Modellierungselementen wird die zugehörige Spezifikationsumgebung UCMT präsentiert.

7.1 UCSM : Ein Metamodell für Use-Case-zentrierte Anforderungsspezifikationen

UCSM (Use Case Centered Specification Model) ist ein Metamodell für die Use-Case-zentrierte Anforderungsmodellierung. Es bietet eine integrierte Modellierungssicht auf die verschiedenen Aspekte der Use-Case-zentrierten Analyse und ermöglicht es, unterschiedliche textuelle und graphische Sichten konsistent zu beschreiben und semi-automatisch zu prüfen. In SUPrA dient es als Spezifikationsmetamodell für das Prototyping.

Strukturell ist UCSM in eine Reihe von Submodellen aufgeteilt. Diese modellieren, ähnlich wie die Spracheinheiten der UML2 Spezifikation, jeweils einen Aspekt der Use-Case-zentrierten Analyse (vgl. 2.4.3). UCSM besteht aus vier Submodellen:

- **UDM:** Das *Use Case Description Model* (UDM) dient zur Modellierung des Systemverhaltens. Hierzu werden flussorientierte Use Case Modelle verwendet.
- **GDM:** Im *GUI Description Model* (GDM) wird die Nutzerschnittstelle des Systems beschrieben. Dazu enthält das GDM eine Reihe von typischen Widgets, aus denen graphische Nutzerschnittstellen aufgebaut werden können.
- **DCM:** Zur Modellierung von Domänenkonzepten wird das *Domain Concept Model* (DCM) eingesetzt. Es definiert Elemente der Klassenmodellierung wie Entitäten und deren Beziehungen.
- **SFM:** Das *System Function Model* (SFM) beschreibt die Funktionssicht des modellierten Systems. Es enthält Funktionen und Variablen des Systems.

Die Verbindung zwischen den einzelnen Aspekten vom UCSM wird dabei, wie in Abbildung 7.1 skizziert, durch ein spezielles Basismodell, das *Basic Concepts Model* (BCM), hergestellt.

Im Folgenden wird zunächst das Basismodell vorgestellt. Anschließend werden der Reihe nach die Konzepte der vier Submodelle eingeführt. Zusätzlich wird UCTM eine Spezifikationsumgebung für UCSM Modelle präsentiert.

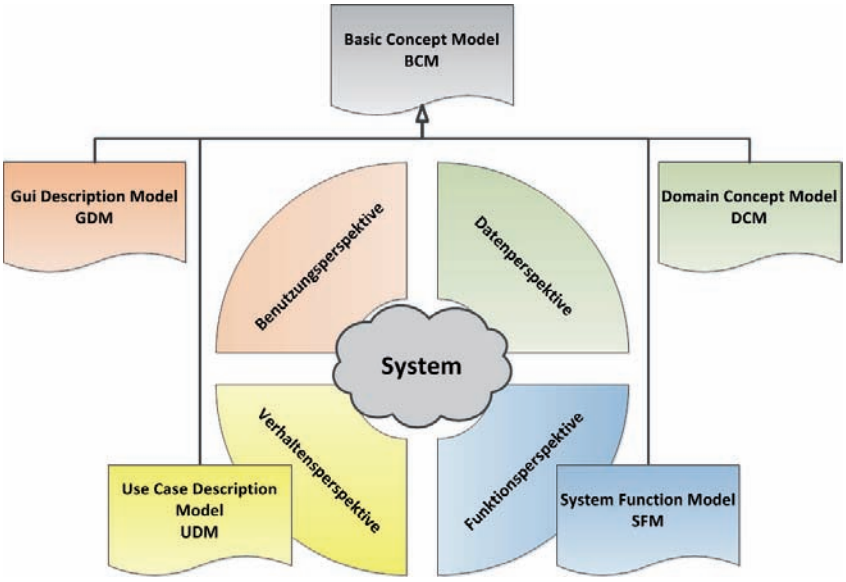


Abbildung 7.1: UCSM Submodelle - Überblick

7.2 UCMT : Eine Spezifikationsumgebung für UCSM

Die Use Case Centered Modeling Toolbox (UCMT) (vgl. Abb. 7.2) ist eine Spezifikationsumgebung für UCSM. Sie besteht aus graphischen, text- und formularbasierten Werkzeugen für die Entwicklung von UCSM Modellen. UCMT enthält jeweils ein oder mehrere Werkzeuge für jedes UCSM Submodell, sowie Ansichten für die Validierung von UCSM Modellen. Eine Beschreibung der technischen Umsetzung gibt Kapitel 11. Im Folgenden werden die einzelnen Werkzeuge von UCMT jeweils gemeinsam mit den zugehörigen UCSM Submodellen vorgestellt:

- **UDM:** Für die Spezifikation von UDM Modellen enthält UCMT den *UDM Diagramm Editor* zur Bearbeitung von Use Case Diagrammen und den *UDM Flow Editor* zur Beschreibung von flussorientierten Abläufen.

- **GDM:** Zur Modellierung von GDM Modellen enthält UCMT den *GDM*

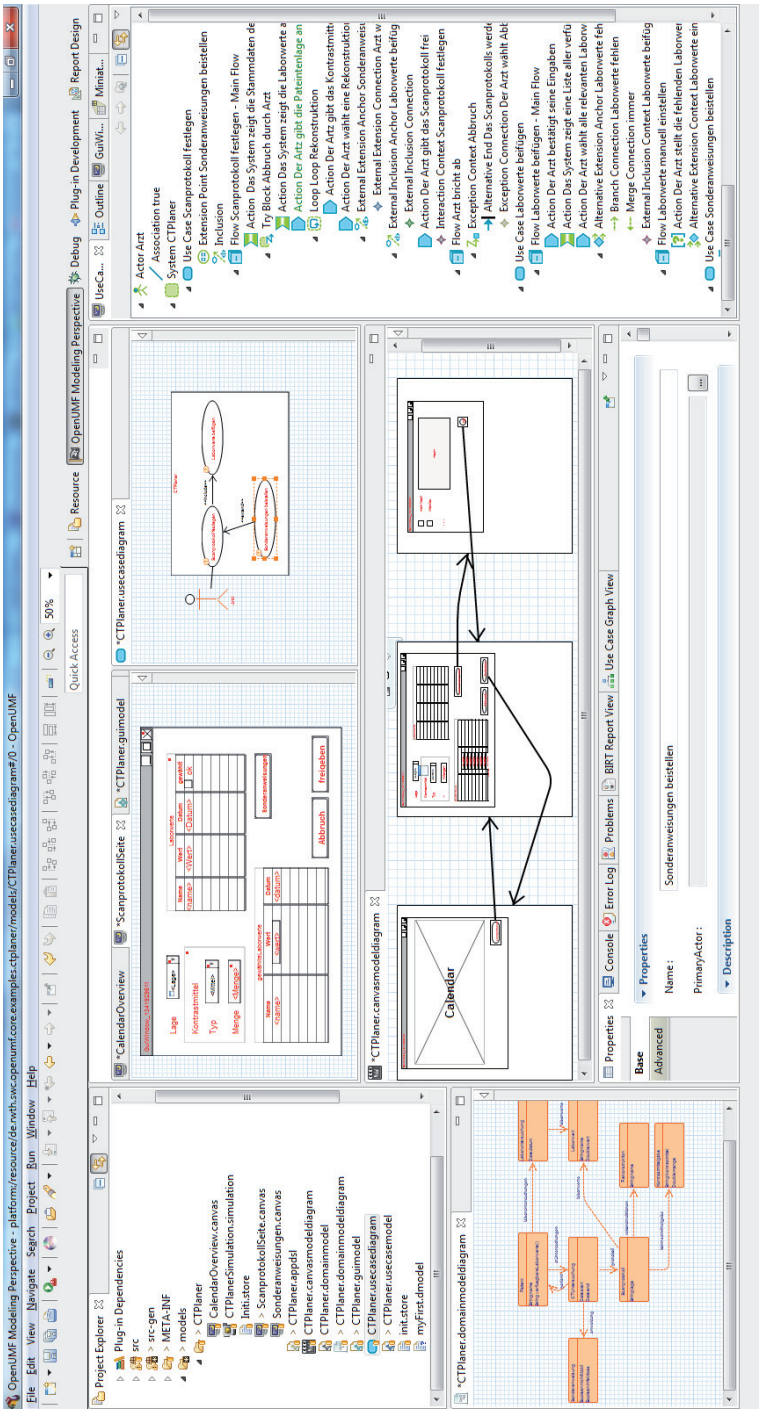


Abbildung 7.2: UCMT – Screenshot

Screen Editor zur Beschreibung einzelner Nutzerschnittstellenseiten sowie den *GDM Overview Editor* zur Beschreibung von deren Zusammenspiel in der modellierten Applikation.

- **DCM:** Mit dem *DCM Diagramm Editor* können DCM Modelle mit einer an UML2 Klassendiagramme angelehnten graphischen Notation modelliert werden.
- **SFM:** SFM Modelle werden rein textuell in einer domänenspezifischen Sprache im *SFM DSL Editor* modelliert.

7.3 BCM - Basic Concepts Model

Das in Abbildung 7.3 dargestellte Basic Concepts Model (BCM) bildet die Grundlage des UCSM Metamodells. Ähnlich wie die Meta-Object-Facility (MOF) [OMG11] definiert es allgemeine Konzepte, von denen alle anderen Elemente des UCSM Metamodells abgeleitet werden müssen. Außerdem stellt es die Infrastruktur für Querverweise zwischen Elementen unterschiedlicher UCSM Modelle zur Verfügung.

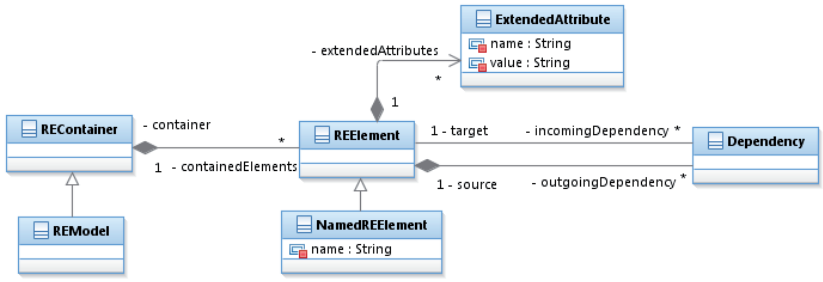


Abbildung 7.3: BCM Metamodell

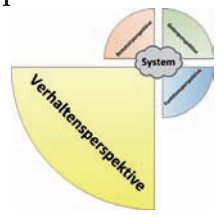
REElement & REContainer: Kernelemente des BCM sind *REElement*, ein abstraktes Oberkonzept für alle Konzepte, die in UCSM beschrieben werden können, sowie *NamedREElement* für REElemente mit Namen. Zusätzlich definiert das BCM den *REContainer* als Oberkonzept für die hierarchische Strukturierung von UCSM Modellen.

Dependency: Zur Modellierung von beliebigen Referenzen zwischen REElementen definiert das BCM das abstrakte Konzept *Dependency*. Von diesem werden alle spezielle Beziehungen zwischen bestimmten Arten von REElementen abgeleitet.

ExtendedAttribute: Mit *ExtendedAttribute* definiert das BCM einen leichtgewichtigen Annotationsmechanismus zur Erweiterung der Modellelemente um spezielle Informationen ohne Anpassung des Metamodells. Dazu kann jedem REElement eine beliebige Menge von *ExtendedAttributes* assoziiert werden, in denen beliebige textuelle Informationen abgelegt werden können.

Dieser Mechanismus ist zwar nicht so mächtig wie eine volle Stereotypunterstützung, wie sie z.B. die UML2 Spezifikation [Obj11] enthält. Unsere Erfahrungen zeigen aber, dass der Annotationsmechanismus in der Praxis ausreicht, da semantikreichere Erweiterungen leicht durch Definition eines neuen Submodells, also einer White-Box-Erweiterung des BCM Metamodells, bewerkstelligt werden können.

7.4 UDM - Use Case Description Model



Das Use Case Description Model (UDM) enthält die Elemente zur Verhaltensmodellierung mit Use Cases. Es bildet das zentrale Modell bei der Use Case zentrierten Analyse und ist das Ausgangsmodell für die Prototypengenerierung. UDM basiert auf dem Narrative Model [HLNW09], erweitert dieses aber um einige Konzepte, die die Beschreibung des Kontrollflusses erleichtern. Im Folgenden werden die Konzepte des UDM kurz vorgestellt.

7.4.1 Struktur

Im strukturellen Teil des UDM wird der Aufbau von Use Case Modellen modelliert. Dazu werden Akteure, Use Cases und deren Beziehungen beschrieben. Abbildung 7.4 stellt den strukturellen Teil des UDM Metamodells dar. In Abbildung 7.5 werden die Modellelemente am CTPlaner-Beispiel verdeutlicht.

Wie auch sein Vorgänger, das Narrative Model [HLNW09], ist das UDM 100%ig kompatibel mit den Mechanismen der UML2 Spezifikation [Obj11]. Anders als im Narrative Model wird dies aber nicht durch Referenzierung eines UML2 Use Case Modells erreicht, sondern durch Modellierung von äquivalenten Konzepten in UDM. Dies hat den Vorteil, dass aufwendige Resynchronisationen zwischen UDM und einem zugehörigen UML2 Use Case Modell entfallen. Dennoch ist es aufgrund der Kompatibilität der Konzepte leicht möglich, den strukturellen Teil eines UDM Modells automatisch in ein UML2 Use Case Modell zu überführen und umgekehrt.

Die Wurzel jedes UDM Modells bildet ein *UseCaseModel*. Dieses enthält eine Reihe verschiedener *Namespaces*. Das abstrakte Element Namespace ist von NamedREElement abgeleitet und stellt so die Verbindung zum BCM her. Ins-

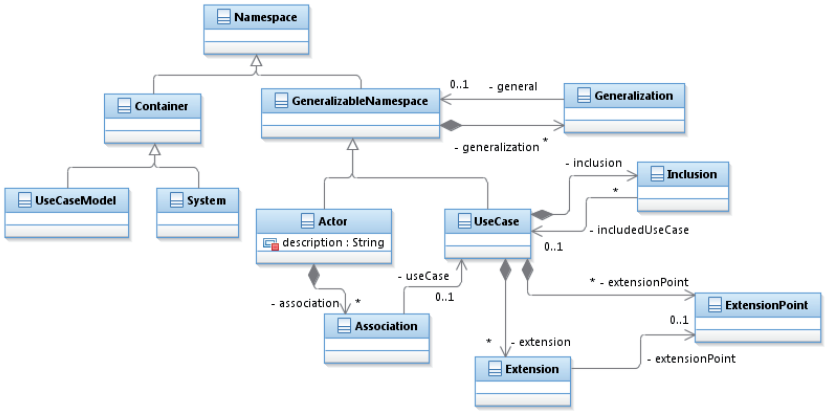


Abbildung 7.4: UDM Metamodell – Struktur

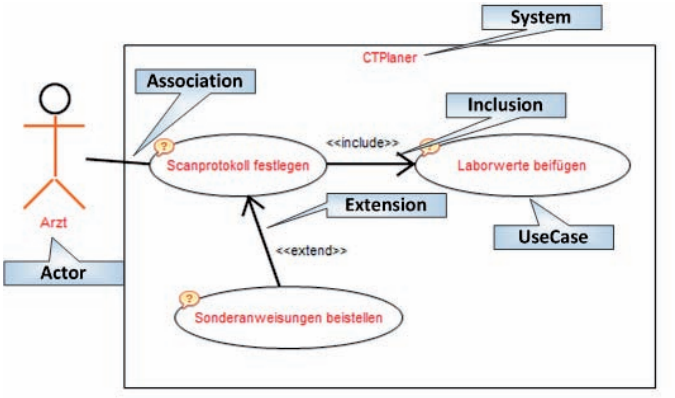


Abbildung 7.5: Beispiel – UDM Struktur

gesamt werden vier verschiedene Namespaces unterschieden.

- **Container:** *Container* sind Strukturierungselemente zur Gruppierung anderer Elemente. Sie dienen beispielsweise dazu, Use Cases, die zu einem bestimmten Subsystem oder einem bestimmten Geschäftsfall gehören, zu gruppieren.

- **System:** Das Element *System* ist von *Container* abgeleitet. Es stellt die Systemgrenze dar. Somit bildet es also einen Container für alle Use Cases.

- **Actor:** Zur Darstellung von Systemakteuren wird das Element *Actor* eingesetzt. Jeder Actor enthält neben einem Namen eine kurze textuelle Beschreibung, die den Akteur näher klassifiziert.

- **Use Case:** *UseCases* modellieren die Anwendungsfälle eines Systems. Neben einer Verhaltensbeschreibung enthalten Use Cases verschiedene Informationen zum Kontext, wie z.B. Vor- und Nachbedingungen.

Dependencies: In UDM können alle Beziehungen zwischen Namespaces, die in der UML2 definiert sind, modelliert werden. Die Beziehung zwischen Akteuren und Use Cases wird durch das Element *Association* dargestellt. Allerdings wird auf die Modellierung von Assoziationsenden verzichtet, die in UML2 Use Case Diagrammen keine Semantik haben. Inklusionsbeziehungen werden mit dem Element *Inclusion* modelliert. Erweiterungsbeziehungen werden durch *Extensions* realisiert, die an einen *ExtensionPoint* gebunden werden. Die Generalisierungsbeziehung zwischen Use Cases bzw. Akteuren kann durch das Element *Generalization* dargestellt werden. Alle Arten von Beziehungen werden vom Element *Dependency* aus dem BCM abgeleitet.

7.4.1.1 UCMT – UDM Diagramm Editor

Der strukturelle Teil des UDM wird mit dem in Abbildung 7.6 dargestellten graphischen Editor bearbeitet. Alle Elemente können aus einer Palette gewählt und frei auf der Zeichenfläche positioniert werden.

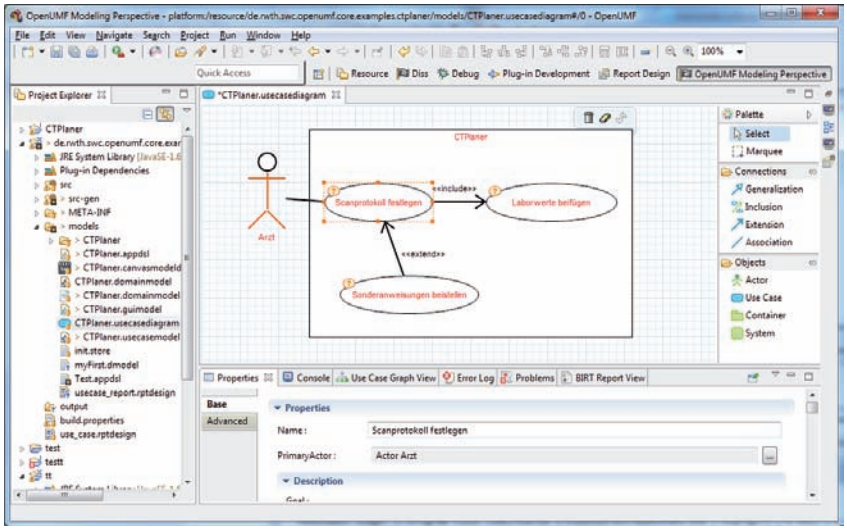


Abbildung 7.6: UCMT – UDM Diagram Editor

7.4.2 Ablaufbeschreibungen

Der Strukturteil des UDM gibt, ebenso wie die Use Case Diagramme der UML2, lediglich einen Überblick über die Struktur eines Use Case Modells und die Zusammenhänge von Modellelementen. Er bietet aber keinerlei Möglichkeit, Abläufe zu beschreiben. Diese werden im UDM als detaillierte flussorientierte Verhaltensbeschreibungen modelliert. UDM orientiert sich hierzu an der von Bittner und Spence [BS02] vorgeschlagenen Notation. Jeder Use Case besteht also, wie in Abbildung 7.7 dargestellt, aus einer Menge von Flüssen. Diese werden durch das Konzept *Flow* dargestellt.

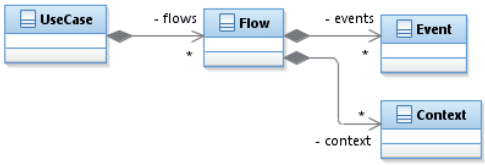


Abbildung 7.7: UDM Metamodell – Flussorientierte Beschreibungen

Der Idee von Bittner und Spence folgend, besteht jeder Flow aus einer Folge von konsekutiven *Events*, die jeweils einen „atomaren" Schritt in der Verhaltensbeschreibung darstellen. Zusätzlich ist jeder Flow mit einer Reihe von *Contexts* ausgestattet. Diese beschreiben die Benutzung eines Flows im Ereignisfluss (vgl. [HL10]).

7.4.2.1 UCMT – UDM Flow Editor

Die Ablaufbeschreibungen des UDM werden in einem formularbasierten Editor bearbeitet. Dieser bietet eine Baumansicht aller Elemente, in der auch neue Elemente eingefügt werden können. Zusätzlich enthält der UCMT Flow Editor für jeden Elementtyp des UDM eine Detailseite, auf der die Attribute von Elementen gesetzt werden können. Der UDM Flow Editor ist in Abbildung 7.8 dargestellt. Zusätzlich lassen sich die Beschreibungen in ein Textformat exportieren. Abbildung 7.9 zeigt eine vereinfachte Darstellung des exportierten Use Cases `Laborwerte beifügen`. Diese Darstellung wird im weiteren Verlauf zur Präsentation der einzelnen Konzepte von UDM verwendet.

7.4.3 Events

Wie in Abbildung 7.10 dargestellt, unterscheidet UDM eine Reihe verschiedener *Events*, die entweder Verhalten beschreiben oder zur Steuerung des Kontrollflusses dienen.

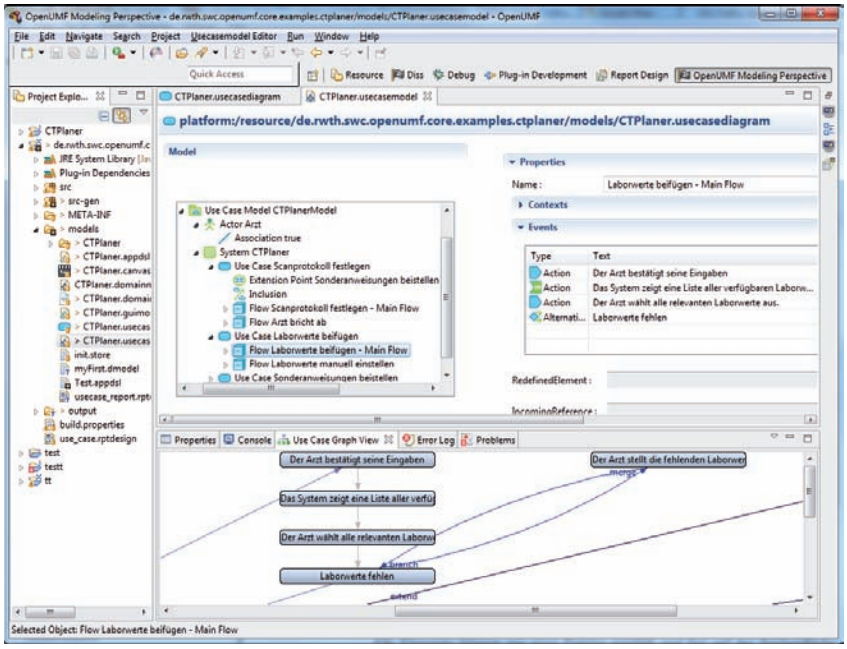


Abbildung 7.8: UCMT – UDM Flow Editor

		Author:	**VHoff**
ID	0001		
Titel	Laborwerte beifügen		
Primary Actor	Arzt		
	...		
Flow 0001.01	Laborwerte beifügen - Hauptfluss		
Contexts	0001.01.C01 Interaktion mit Arzt.		
Events	Action	Das System zeigt eine Liste aller verfügbaren Laborwerte an. table.show(labValues);	
	Action	Der Arzt wählt alle relevanten Laborwerte aus. selected = table.selection();	
	Extension	Laborwerte fehlen --> Fehlende Laborwerte	
	Action	Der Arzt bestätigt seine Eingaben. button.click();	
Flow 0001.02	Laborwerte manuell einstellen		
Contexts	0001.02.C01 Extension Fehlende Laborwerte: wichtige Laborwerte fehlen if(!selected.containsAll(expected));		
Events	Action	Der Arzt stellt die fehlenden Laborwerte manuell ein. selected+= manualList;	

Abbildung 7.9: Exportierter Use Case – Laborwerte beifügen

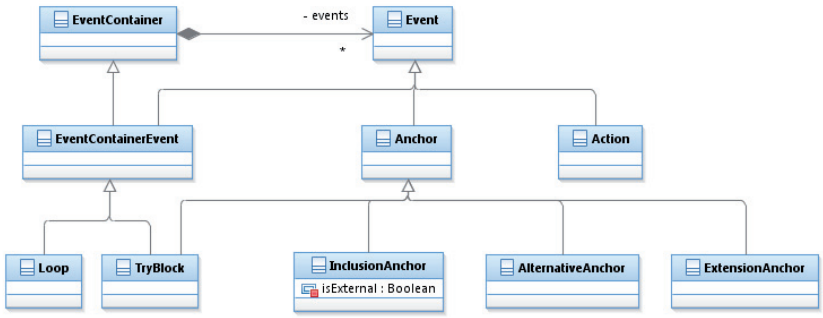


Abbildung 7.10: UDM Metamodell – Events

Action: Eine *Action* beschreibt einen atomaren Verhaltensschritt, den entweder ein Akteur oder das System selbst ausführt. Dazu enthält die Aktion eine textuelle Beschreibung, die nicht weiter interpretiert wird. Diese kann aber zusätzlich, wie in Abbildung 7.12 skizziert, mit einer *FormalDescription*, einer formalen Darstellung, weiter präzisiert werden. In Abschnitt 7.8 wird auf das Konzept der FormalDescriptions im Detail eingegangen. Zusätzlich kann der Ausführende, also ein dem Use Case assoziierter Akteur oder das System selbst, referenziert werden. Um diese Beziehung darzustellen, wurde die abstrakte Metaklasse *ActionExecutor* eingeführt (vgl. Abb. 7.11).

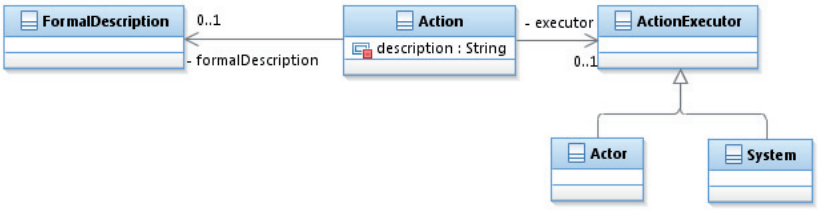


Abbildung 7.11: UDM Metamodell – Action

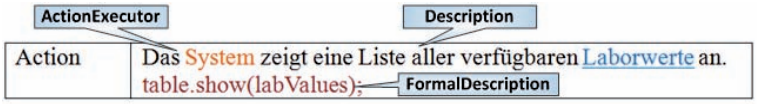


Abbildung 7.12: Beispiel – UDM Action

Anchor: Neben den Actions gibt es eine Reihe verschiedener *Anchors*, die in Abbildung 7.13 dargestellt werden. Diese modellieren Ankerpunkte, an denen

unterschiedliche Beziehungen zwischen Flows sowohl eines Use Cases als auch über Use Case Grenzen hinweg realisiert werden.

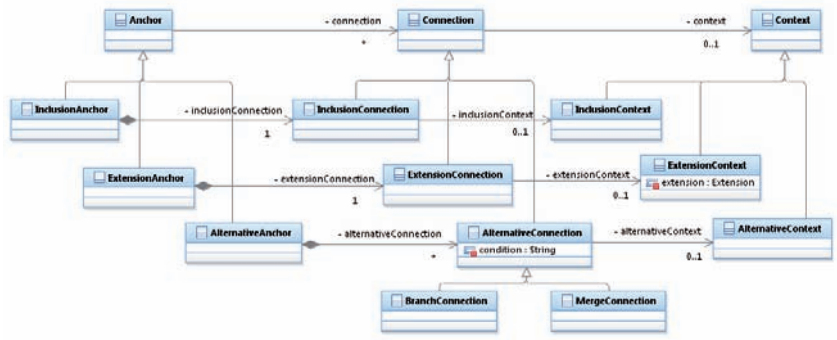


Abbildung 7.13: UDM Metamodell – Ankerpunkte

InclusionAnchor: Ein *InclusionAnchor* modelliert einen Ankerpunkt für eine Inklusionsbeziehung. Er beschreibt also eine Stelle, an der Events eines inkludierten Flusses in einen inkludierenden Fluss eingefügt werden. Es können zwei Arten von *InclusionAnchors* unterschieden werden. Interne *InclusionAnchors* realisieren den Anker einer Inklusionsbeziehung zwischen zwei Flüssen desselben Use Cases. *InclusionsAnchors*, die hingegen einen Anker für die Inklusion von Verhalten aus einem Fluss eines anderen Use Cases darstellen, werden als externe InclusionAnchors bezeichnet. Die Unterscheidung zwischen diesen beiden Arten von Inklusionsankern wird jedoch nicht, wie im Narrative Model [HLNW09], explizit über einen Typ gemacht, sondern aus dem inkludierten Fluss abgeleitet und lediglich durch ein transientes Attribut repräsentiert.

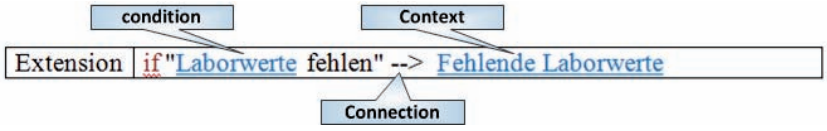


Abbildung 7.14: Beispiel – UDM ExtensionAnchor

ExtensionAnchor: Ein *ExtensionAnchor* dient zur Realisierung einer Erweiterungsbeziehung zwischen zwei Flüssen verschiedener Use Cases. Er ist also ein Ankerpunkt, an dem Verhalten eines erweiternden Flusses unter bestimmten Bedingungen in einen erweiterten Fluss eingefügt wird. Jeder ExtensionAnchor ist immer einem bestimmten ExtensionPoint des zugehörigen Use Cases assoziiert. Er manifestiert diesen in der Verhaltensbeschreibung. Aus Kompatibilitätsgründen zur UML2, in der die Erweiterungsbeziehung als bedingtes Einfügen von Verhalten definiert ist (vgl. [Obj11]), wird auch für ExtensionAnchors diese Se-

mantik angenommen. Folglich realisiert jeder ExtensionAnchor immer sowohl einen Aussprung- als auch einen Einsprungpunkt.

AlternativeAnchor: Ein *AlternativeAnchor* realisiert eine bedingte Beziehung zwischen zwei Flüssen desselben Use Cases. Bei einer derartigen Beziehung gibt es keine Einschränkungen von Seiten der UML2. Deshalb kann hier alternatives Verhalten modelliert werden. Ein AlternativeAnchor kann also einen Einsprung- bzw. Aussprungpunkt oder beides realisieren. Abbildung 7.15 stellt alle drei verschiedenen Möglichkeiten dar.

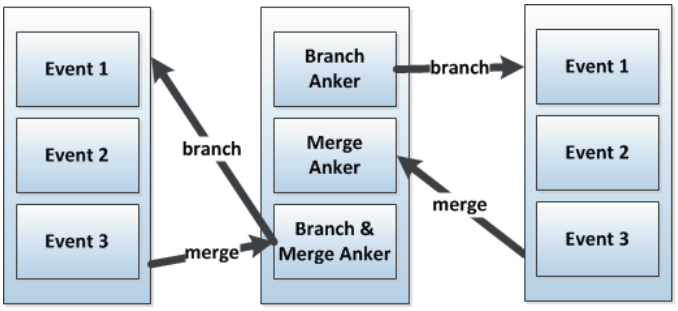


Abbildung 7.15: UDM – Ankerpunkte: Einsprung und Aussprung

Loop: Im Prinzip ist es zwar möglich, Iteration mit Hilfe von alternativen Flüssen zu formulieren. Dies ist aber in der Praxis oft unhandlich und führt zu unübersichtlichen Konstrukten aus mehreren Flüssen mit verschiedenen Rücksprungbedingungen. Deshalb wurde das Element *Loop* als expliziter Schleifenknoten eingeführt. Eine Loop enthält dazu, wie in Abbildung 7.16 dargestellt, eine Menge von Events, die den Schleifenkörper bilden und eine *loopCondition*, die die Abbruchbedingung der Schleife beschreibt.

TryBlock: In Use Cases gibt es oftmals alternatives Verhalten, das zu jeder Zeit während einer Folge von Schritten auftreten kann. Wie auch im Falle der Iteration ist es zwar grundsätzlich möglich, dieses Verhalten mit alternativen Flüssen abzubilden, die entstehenden Konstrukte sind aber i.d.R. nicht überschaubar, da eine große Zahl von Ankerpunkten modelliert werden muss, an denen aus- bzw. eingesprungen werden kann. Um dieses Problem zu lösen, wurde das Element *TryBlock* eingeführt. Dieses Element definiert ähnlich wie ein „try"-Block in Programmiersprachen (Java) eine geschützte Region (protected Region), in der eine Ausnahmesituation, die eine spezielle Reaktion erfordert, jederzeit auftreten kann. Ein TryBlock bildet dabei ebenfalls einen Ankerpunkt, an den spezielle Kontexte gebunden werden können, die Ausnahmesituationen beschreiben.

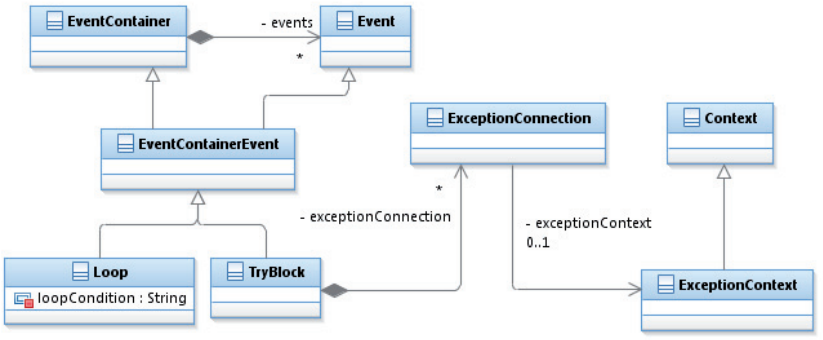


Abbildung 7.16: UDM Metamodell – Loop & TryBlock

7.4.4 Contexts

Wie bereits erwähnt, wird die Verwendung von Flüssen im Gesamtablauf modelliert, indem Flüssen Benutzungskontexte assoziiert werden. Diese Benutzungskontexte werden mit dem Konzept *Context* modelliert. Die unterschiedlichen Verwendungsarten werden jeweils durch einen speziellen Kontexttyp repräsentiert.

InteractionContext: Ein *InteractionContext* zeigt an, dass ein Fluss direkt von einem Akteur angestoßen wird. Er klassifiziert also einen Fluss als Hauptfluss (vgl. 2.4.1.4). Jeder InteraktionContext hält deshalb eine Referenz zum primären Akteur des Use Cases und definiert seinen Auslöser (Trigger). Zusätzlich beschreibt er Vorbedingungen, die erfüllt sein müssen, um den Fluss bzw. den zugehörigen Use Case auszuführen, und Nachbedingungen für den Normalablauf.

ConnectionContext: Abgesehen vom InteractionContext, der eine Sonderrolle einnimmt, dienen alle weiteren Kontexte zur Modellierung einer Verbindung zweier Flüsse. Während Ankerpunkte definieren, an welchen Stellen Verhalten in einen Fluss eingefügt werden kann, definieren Kontexte die Situation aus Sicht des einfügenden Flusses. Sie beschreiben jeweils eine Situation, in der das Verhalten des zugehörigen Flusses an referenzierten Ankern eines anderen Flusses eingefügt wird. Dabei wird die konkrete Verbindung zwischen Context und Anchor, die die jeweiligen Elemente repräsentieren, jeweils durch eine *Connection* hergestellt (vgl. Abb. 7.13). Diese enthält auch die Bedingungen, unter denen das Verhalten eingefügt wird.

InclusionContext: Ein *InclusionContext* bildet das Gegenstück zu einem InclusionAnchor. Er realisiert also eine Inklusionsbeziehung aus Sicht des inkludierten Flusses. Analog zu den Ankern können ebenfalls zwei Arten von InclusionContexts unterschieden werden. Interne InclusionContexts realisieren eine Inklusionsbeziehung zwischen zwei Flüssen desselben Use Cases. Externe InclusionContexts hingegen stellen eine Inklusionsbeziehung zwischen zwei Flüssen verschiedener Use Cases dar. Wie auch bei InclusionAnchors ist diese Unterscheidung lediglich implizit modelliert. Die Verbindung zwischen InclusionAnchor und InclusionContext wird dabei durch eine *InclusionConnection* hergestellt.

ExtensionContext: Analog zum InclusionContext, der eine Inklusionsbeziehung beschreibt, stellt ein *ExtensionContext* eine Erweiterungsbeziehung aus Sicht des erweiternden Flusses dar. Wie bei der Inklusionsbeziehung wird auch bei der Erweiterungsbeziehung die Verbindung zwischen Anker und Kontext durch ein explizites Verbindungselement, die *ExtensionConnection*, realisiert.

Die UML2 Spezifikation [Obj11, 601] erlaubt es, bei Erweiterungsbeziehungen mehrere verschiedene Verhaltensfragmente eines erweiternden Use Cases an unterschiedlichen Stellen des erweiterten Use Cases sogar mehrfach einzufügen. Dieser Semantik folgt UDM nicht. Das Binden von mehreren Ankern an einen Erweiterungskontext führt zu sehr stark zersplitterten Szenarien, einer unklaren Ausführungssemantik und deutet i.d.R. auf einen Modellierungsfehler hin. Deshalb wird in UDM auf ein solches Konstrukt verzichtet.

AlternativeContext: Ein *AlternativeContext* realisiert die Verbindung von zwei Flüssen innerhalb desselben Use Cases. Auch dieser wird explizit mit einer *AlternativeConnection* an einen AlternativeAnchor gebunden. Hierbei müssen allerdings mehrere Fälle unterschieden werden, da die Beziehung alternatives Verhalten modelliert. Deshalb müssen Einsprung- und Aussprungpunkte eines AlternativeContexts unterschieden werden. Dazu werden zwei verschiedene Arten von AlternativeConnections eingesetzt.

BranchConnections beschreiben, dass ein AlternativeAnchor als Aussprungpunkt an einen AlternativeContext gebunden ist.

MergeConnections hingegen modellieren eine Verbindung zu einem Rücksprungpunkt. Die Bedingung für Aus- bzw. Rücksprung wird dabei jeweils an der AlternativeConnection modelliert. Zusätzlich ist es möglich, Situationen zu beschreiben, in denen der Ausführungskontext nicht zum aufrufenden Fluss zurückkehrt und stattdessen ein alternatives Ende erreicht.

Solche alternativen Enden werden mit dem Element *AlternativeEnd* modelliert. Jedes AlternativeEnd ist mit einer Bedingung versehen, die beschreibt, wann es eintritt. Zusätzlich enthalten sie eine *postCondition*, die die Nachbedingungen beim Erreichen des Endes modelliert, da alternative Enden typischerweise nicht die Nachbedingungen im Erfolgsfall herstellen.

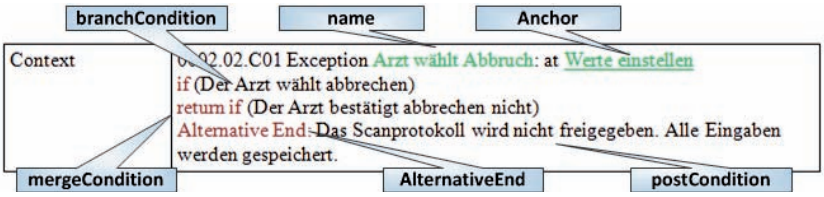


Abbildung 7.17: Beispiel – UDM ExceptionContext

ExceptionContext: Beschreiben AlternativeContexts alternatives Verhalten, das an einer einzelnen Stelle auftritt, dienen *ExceptionContexts* der Beschreibung von Alternativverhalten, das zu jeder Zeit während einer Schrittfolge auftreten kann. ExceptionContexts werden mit Hilfe einer *ExceptionConnection* an einen TryBlock gebunden. Neben der Aussprungbedingung (*branchCondition*), kann zusätzlich eine Rücksprungbedingung (*mergeCondition*) definiert werden. Außerdem kann, wie auch beim AlternativeContext, ein *AlternativeEnd* gebunden werden, um ein alternatives Ende darzustellen.

Generalisierung: Wie bereits erwähnt, ist es auf struktureller Ebene des Use Case Modells möglich, eine Generalisierungsbeziehung zwischen zwei Use Cases zu modellieren. Wenn ein Use Case eine Spezialisierung eines anderen ist, erbt er auf Ebene der flussorientierten Beschreibung alle Flüsse des generellen Use Cases.

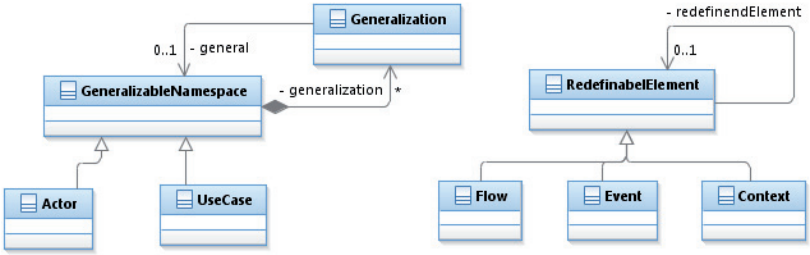


Abbildung 7.18: UDM Metamodell – Generalisierung & Redefinition

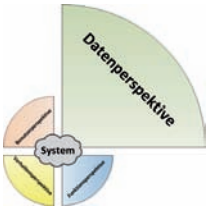
Um die Spezifika des speziellen Use Cases abzubilden, können Flows, Events und Contexts des generellen Use Cases im speziellen Use Case redefiniert werden. Dazu werden diese vom abstrakten Element *RedefinableElement* abgeleitet (vgl. Abb. 7.18). Die Redefinition des Verhaltens wird dabei in zwei Schritten durchgeführt:

- Zunächst kann ein Flow des generellen Use Cases im speziellen Use Case redefiniert werden. Dabei werden Kopien von allen Events und Contexts

des redefinierten Flows im redefinierenden Flow angelegt.

- Anschließend können die kopierten Flows angepasst werden, um spezielles Verhalten zu beschreiben. Zusätzlich können in den redefinierten Flows neue Events und Contexts hinzugefügt werden. Außerdem ist es möglich, komplett neue Flows im speziellen Use Case zu definieren.

7.5 DCM - Domain Concept Model



Das Domain Concept Model (DCM) dient zur Modellierung der Datensicht in der Use Case getriebenen Anforderungsspezifikation. Es ähnelt stark UML2 Klassendiagrammen und ist zu 100% mit diesen kompatibel. Es erweitert Klassendiagramme aber zusätzlich um die Möglichkeit, Elemente mit textuellen Informationen anzureichern, um Domänenglossare (vgl. [Poh08, 244]) in frühen Phasen der Datenmodellierung zu unterstützen. Außerdem kann die Implementierung von Methoden modelliert werden, um das Systemverhalten für das Prototyping zu beschreiben.

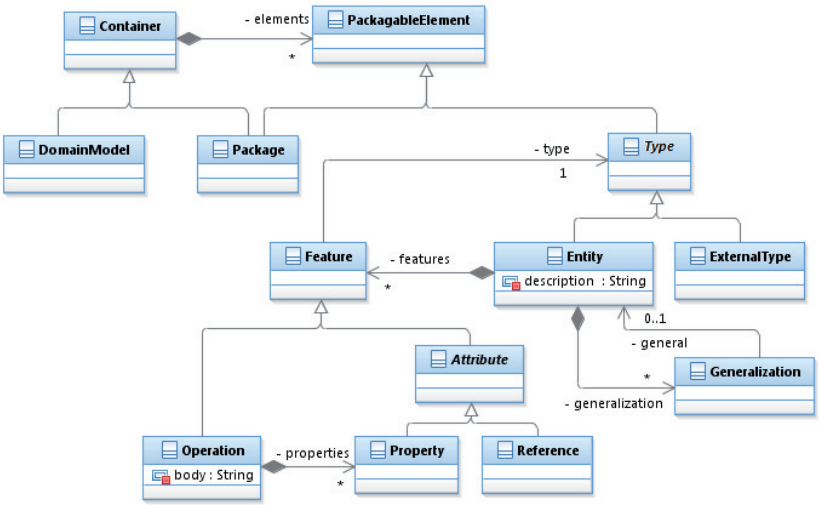


Abbildung 7.19: DCM Metamodell

7.5.1 Aufbau

Die Wurzel eines jeden DCM bildet das *DomainModel*. Es enthält *Entities*, die in Paketen hierarchisch geschachtelt sein können. Diese Pakete werden durch das Element *Package* repräsentiert.

7.5.2 Entities & Relations

Entity: Das Element *Entity* beschreibt ein bestimmtes Domänenkonzept. Es enthält eine kurze textuelle Beschreibung des Konzepts, wie sie in einem Domänenglossar vorkommen würde. Zusätzlich können Entitäten, wie in Abbildung 7.19 dargestellt, Attribute und Operationen enthalten.

Attribute: Attribute von Entitäten werden mit *Attributes* modelliert. Sie enthalten jeweils einen Namen, einen Typ und eine Kardinalität, die angibt, ob es sich um einen Einzelwert oder eine Menge von Werten handelt. In DCM werden zwei unterschiedliche Arten von Attributes unterschieden. *Properties* beschreiben primitive Attribute einer Entität. *References* hingegen stellen Referenzen zu einer anderen *Entity* dar.

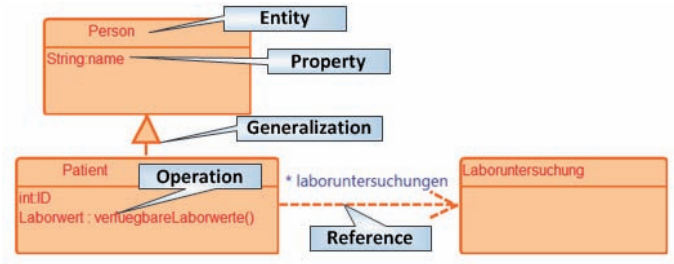


Abbildung 7.20: Beispiel – DCM Entities & Properties

Operation: *Operations* beschreiben Funktionen, die eine Entity anbietet. Jede Operation definiert eine Signatur bestehend aus einer Menge von Eingabeparametern und einem Rückgabetyp. Zusätzlich kann die Implementierung der Operation in deren *body* angegeben werden.

Type: Typen werden im DCM durch das abstrakte Konzept *Type* repräsentiert. DCM unterscheidet dabei, wie in Abbildung 7.19 dargestellt, zwei Arten von Typen.

Zusätzlich zu Entitäten, die offensichtlich Typen modellieren, gibt es die sogenannten *ExternalTypes*. Diese stellen externe Typen dar, die im DomainModel verwendet werden können. Sie ermöglichen es, zum einen primitive Typen wie Integer oder String im DomainModel zu verwenden, zum anderen kann mit Hilfe dieses Mechanismus auf vorgefertigte Typen, deren Implementierung vorliegt, zurückgegriffen werden. Dieser Mechanismus bricht zwar in gewissem Maße die Abgeschlossenheit der Domänenmodellierung und sollte deshalb mit Bedacht verwendet werden, er hat sich aber vor allem bei der Implementierung von Ope-

rationen für die Simulation als extrem nützlich erwiesen, weil damit leicht existierende APIs eingebunden werden können.

Generalization: Das Konzept *Generalization* dient zur Modellierung einer Generalisierungsbeziehung zwischen zwei Entities. Analog zur Generalisierungsbeziehung in UDM (vgl. Abs. 7.4) wird auch hier ein spezielles Element mit einem generellen Element verbunden. Zusätzlich ist es, anders als in vielen Programmiersprachen, möglich, dass eine Entity eine Spezialisierung mehrerer anderer Entities ist.

7.5.2.1 DCM Diagram Editor

Das DCM wird komplett mit einem graphischen Editor bearbeitet. Alle Elemente können aus einer Palette gewählt und frei auf der Zeichenfläche positioniert werden. Der DCM Diagram Editor ist in Abbildung 7.6 noch einmal dargestellt. Auf der Abbildung erkennt man, dass die Details der einzelnen Entitäten in einem Formular gesetzt werden können.

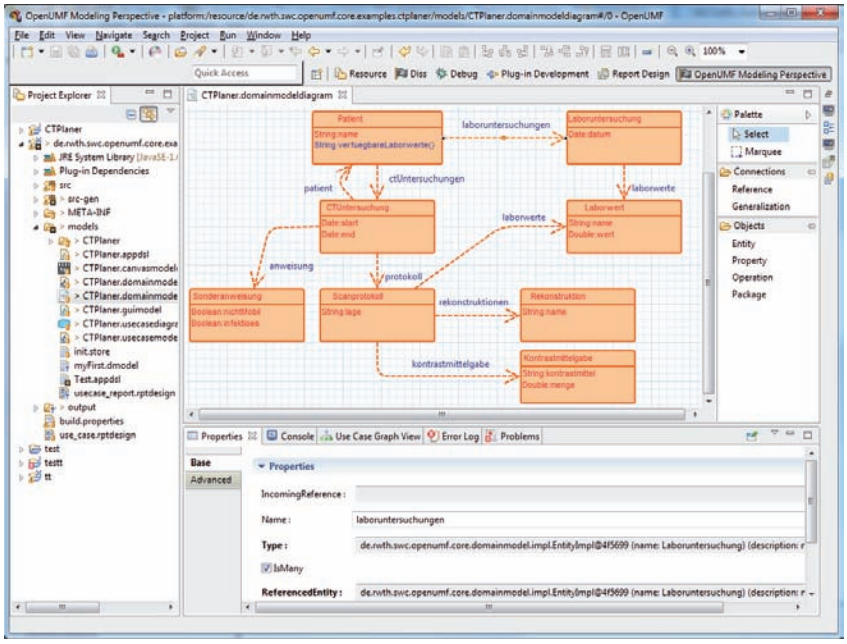
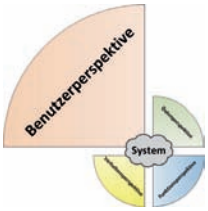


Abbildung 7.21: UCMT – DCM Editor

7.6 GDM - GUI Description Model



Das GUI Description Model (GDM) ist ein Metamodell zur abstrakten Beschreibung des strukturellen Aufbaus von graphischen Nutzerschnittstellen und zur Modellierung von Kontrollelementen. Dabei sollen Elemente der Nutzerschnittstellen unabhängig von der Art der Implementierung beschrieben werden. Im GDM werden ausschließlich die Interaktionselemente der Nutzerschnittstellen modelliert. Auf die Modellierung von Elementen, die nur zur Gestaltung der Nutzerschnittstelle dienen, wie Layouts oder Stylesheets, wird verzichtet.

7.6.1 Struktur

Typischerweise besteht die Nutzerschnittstelle eines interaktiven Systems aus mehreren verschiedenen Bildschirmseiten (Screens). Selbst einfache interaktive Systeme präsentieren dem Nutzer während der Nutzung, abhängig vom Ausführungskontext des Systems, verschiedene Informationen. Abbildung 7.22 beispielsweise zeigt einen Teil der Nutzerschnittstelle des CTPlaners.

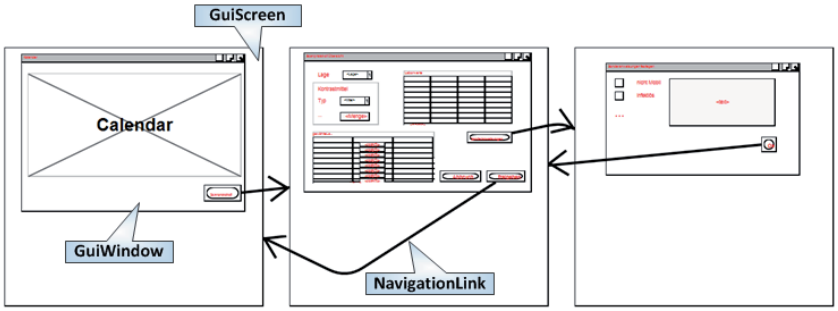


Abbildung 7.22: Beispiel – GDM Struktur

Deshalb enthält jedes *GuiModel*, wie in Abbildung 7.23 dargestellt, eine Reihe verschiedener *GuiScreens*. Jeder dieser GuiScreens stellt eine Bildschirmseite dar, die aus einer Menge visueller Komponenten aufgebaut ist. Zusätzlich können diese GuiScreens hierarchisch in *GuiPackages* strukturiert werden. Dazu sind beide Arten von Elementen vom Konzept *GuiPackageElement* abgeleitet. Weiterhin enthalten GuiModels sogenannte *Components* zur Wiederverwendung von Teilen der Nutzerschnittstelle. Diese werden später noch einmal gesondert behandelt.

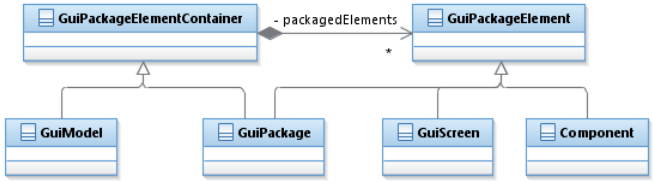


Abbildung 7.23: GDM Metamodell – Container

7.6.2 GDM Overview Editor

Wie auch beim UDM werden für die Modellierung des GDM zwei Editoren verwendet. Der in Abbildung 7.24 dargestellte GDM Overview Editor dient zur Modellierung der Nutzerschnittstellenstruktur. Er bietet eine graphische Ansicht, in der GuiScreens, Components und Beziehungen zwischen diesen modelliert werden können.

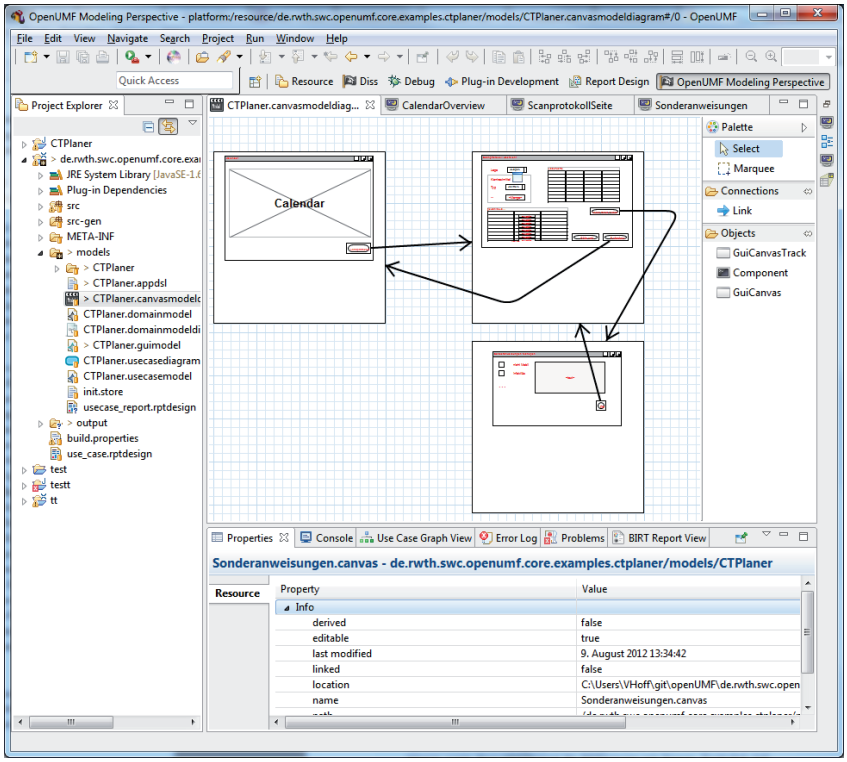


Abbildung 7.24: UCMT – GDM Overview Editor

7.6.3 Widgets

Wie bereits erwähnt, enthalten GuiScreens alle visuellen Komponenten der Nutzerschnittstelle, die zu einem bestimmten Zeitpunkt sichtbar sind. Analog zu den meisten Grafikbibliotheken, wie z.B. SWT [SWT], GTK [gtk] oder PrimeFaces [pri], sind die Nutzerschnittstellen aus verschiedenen Einzelkomponenten aufgebaut. Das Element *GuiWidget* bildet dabei ein abstraktes Oberkonzept für alle Arten von primitiven Nutzerschnittstellenelementen. Jedes GuiWidget hat einen Namen, eine Größe und eine Position in seinem Container.

GDM definiert hier eine Reihe verschiedener primitiver Widgets, sogenannter *GuiControls*. Diese werden im Folgenden kurz vorgestellt. Der entsprechende Ausschnitt des Metamodells wird in Abbildung 7.25 dargestellt.

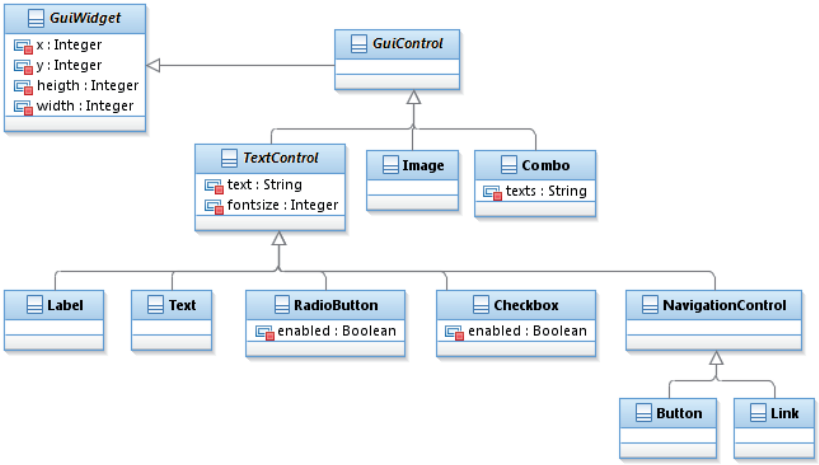


Abbildung 7.25: GDM Metamodell – Widgets

- **Label:** *Labels* stellen einen nicht änderbaren Text dar. Sie können sowohl ein als auch mehrzeilig sein. Neben dem *text*, den sie darstellen, haben sie eine *fontSize*, die die Textgröße darstellt.
- **Text:** Das Element *Text* stellt im Unterschied zu Label ein änderbares Textfeld dar. Texts enthalten wie Labels einen Text und eine Schriftgröße.
- **Button:** Ein *Button* stellt einen Clickbutton dar. Auch Buttons enthalten einen Text und eine Schriftgröße.
- **CheckBox:** Eine *Checkbox* stellt einen Auswahlbutton dar. Checkboxen enthalten zusätzlich zu Text und Fontgröße noch ein Attribut *enabled*, das darstellt, ob die Checkbox standardmäßig angewählt ist oder nicht.
- **RadioButton:** Ein *RadioButton* dient einer 1-n-Auswahl. Er hat eben-

falls ein *enabled* Attribut.

- **Hyperlink:** Der *Hyperlink* dient der Navigation zwischen zwei Seiten.
- **Image:** Das Element *Image* dient als Platzhalter für eine Grafik. Damit lassen sich sowohl Hintergrundbilder realisieren als auch beliebige nicht unterstützte Kontrollelemente simulieren.
- **Combo:** Das *Combo* dient zur Auswahl eines Elements aus einer Liste. Dazu enthält es eine Menge von *optionStrings*, aus denen ausgewählt werden kann.
- **Callout:** Im Unterschied zu den anderen vorgestellten Widgets stellen *Callouts* keine klassischen Controls dar. Stattdessen dienen sie dazu, bestimmte Informationen, wie z.B. Designüberlegungen oder spezielle Anforderungen, graphisch darzustellen.

7.6.4 GuiComposites

GuiComposites dienen dazu, Widgets in GuiScreens hierarchisch zu strukturieren. Sie sind also Container für GuiWidgets. GuiComposites sind im Sinne des Kompositionsmusters [Gam09] ebenfalls von GuiWidget abgeleitet und ermöglichen es, verschiedene Teile einer Nutzerschnittstelle zu verschachteln. Der Aufbau eines GuiWindows wird in Abbildung 7.26 dargestellt.

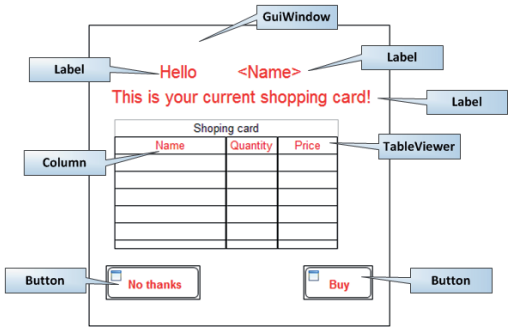


Abbildung 7.26: Beispiel – GDM GuiWindow

GDM unterscheidet zwei verschiedene Arten von GuiContainern: *GuiWindows* und *GuiComposites*. Ein GuiWindow modelliert dabei ein eigenständiges Fenster. Ein GuiComposite hingegen, dient lediglich zur Gruppierung anderer Widgets in einem GuiWindow. Hier gibt es wiederum einige verschiedene Arten, wie z.B. *GuiTabs* oder *GuiGroups*. Den entsprechenden Ausschnitt des GDM Metamodells zeigt Abbildung 7.27.

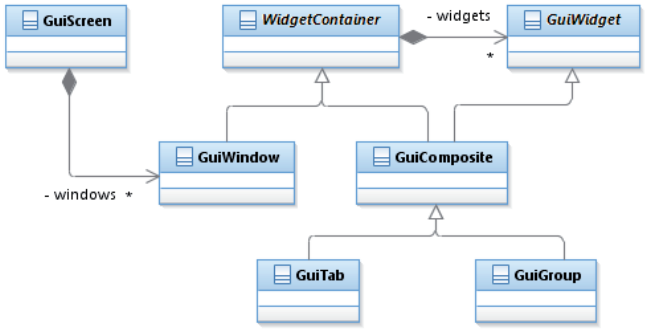


Abbildung 7.27: GDM Metamodell – GuiComposites

7.6.5 NavigationLinks

Oftmals müssen Situationen modelliert werden, in denen durch ein bestimmtes Kontrollelement die Navigation von einer bestimmten Bildschirmseite zu einer anderen ausgelöst wird. Hierzu können, wie in Abbildung 7.28 dargestellt, *NavigationLinks* verwendet werden.

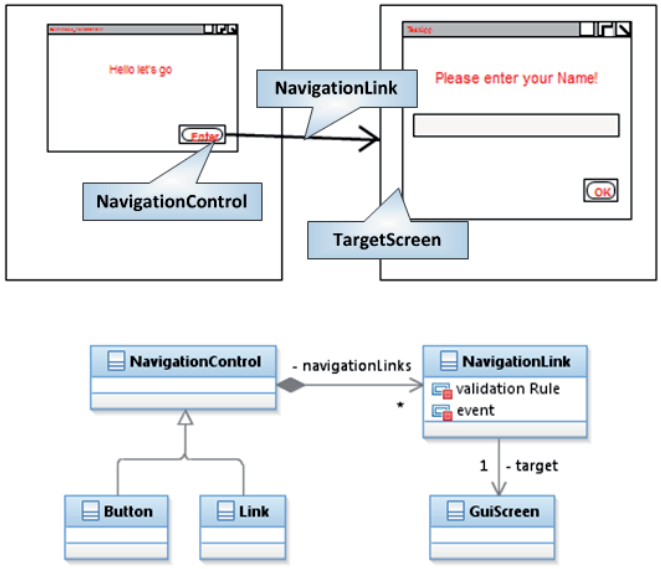


Abbildung 7.28: GDM – Seitennavigation & Links

Dabei ist während der Systemausführung nicht immer jede Navigation erlaubt. Deshalb halten NavigationLinks ein optionales Attribut *validationRule*. Dieses

modelliert eine Validierungsregel, die darstellt, ob eine Navigation möglich ist. So kann die Navigation abhängig vom Systemzustand an- bzw. abgeschaltet werden. Außerdem kann die Validierungsregel in Situationen, in denen einem GuiWidget mehrere NavigationLinks assoziiert sind, genutzt werden, um zu entscheiden, welche Navigation durchgeführt werden soll.

NavigationLinks entsprechen in den meisten Fällen einer Aktion aus dem Ereignisfluss des Use Case Modells. Deshalb kann ihnen eine Action assoziiert werden.

7.6.6 Viewer

Abgesehen von den primitiven Widgets, die normalerweise jeweils eine einzelne Variable darstellen, können im GDM auch kompliziertere Elemente dargestellt werden. Diese sogenannten Viewer stellen gleichzeitig verschiedene Attribute mehrerer unterschiedlicher Typen dar. Das GDM unterstützt Listen, Tabellen, Bäume und Kalender.

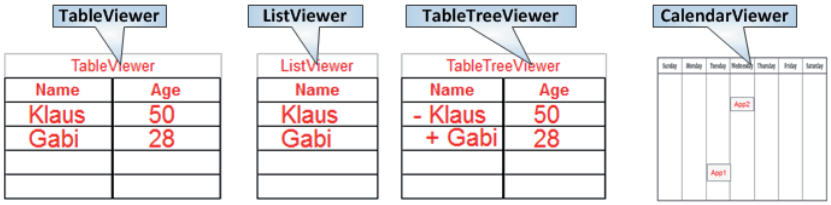


Abbildung 7.29: Beispiel – GDM Viewer

ListViewer: Listen werden durch das Metakonzept *ListViewer* repräsentiert. Dieses ist von GuiWidgetContainer abgeleitet, da es primitive Widgets zur Darstellung von Informationen verwendet.

TableViewer: Tabellen werden durch das Metakonzept *TableViewer* modelliert. Jeder TableViewer enthält eine Menge von *ViewerColumns*, die jeweils eine Spalte der Tabelle darstellen.

TableTreeViewer: Ein *TableTreeViewer* stellt einen sogenannte TableTree dar. TableTrees sind in allen gängigen Grafikframeworks gebräuchlich. Sie sind eine Kombination von einer Tabelle mit einer Baumstruktur. Dabei wird die Datenstruktur wie bei normalen Baumstrukturen in Form einer Vater-Kind-Hierarchie dargestellt. Zusätzlich enthält jeder Datensatz, wie bei einer Tabelle, eine Menge von Spalten, die jeweils unterschiedliche Informationen zum angezeigten Element präsentieren.

ViewerColumn: Eine *ViewerColumn* modelliert eine Spalte eines TableViewers bzw. eines TableTreeViewers. ViewerColumns sind wie Lists von GuiWidgetContainer abgeleitet, da sie ebenfalls an ein primitives Widget deligieren, das die Darstellung des jeweiligen Datensatzes übernimmt. Der Zusammenhang der einzelnen Elemente des Metamodells wird in Abbildung 7.30 verdeutlicht.

CalendarViewer: Ein *CalendarViewer* dient der Darstellung eines Terminkalenders. CalendarViewer zeigen ein oder mehrere Tage und stellen Termine an diesen Tagen graphisch dar.

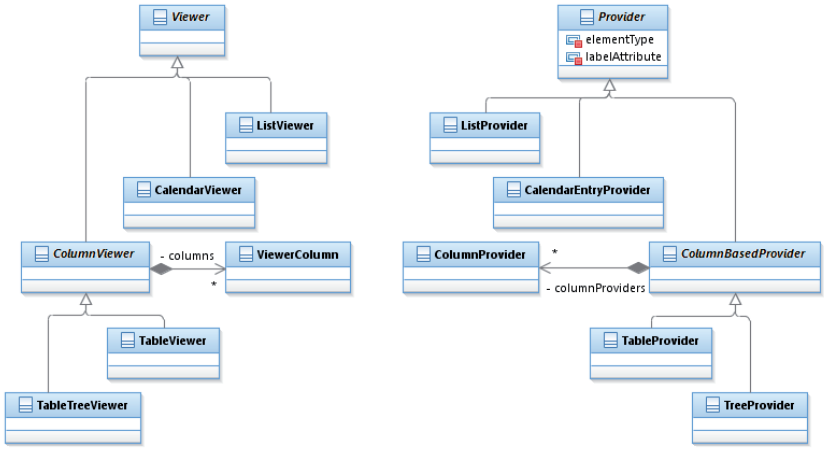


Abbildung 7.30: GDM Metamodell – Viewer & Provider

7.6.7 Provider

Ähnlich wie im EMF Rahmenwerk [emf] wird die Darstellung von einzelnen Elementen in Viewern typweise mit Hilfe von sogenannten *Providern* realisiert. Diese Provider beschreiben jeweils, wie Datenentitäten (vgl. 7.5) eines bestimmten Typs dargestellt werden. Dabei werden, z.B. in einer Tabelle, den primitiven Widgets in den einzelnen Spalten der Tabelle Attribute der angezeigten Daten assoziiert. Dieser Zusammenhang wird in Abbildung 7.31 noch einmal verdeutlicht. GDM unterscheidet fünf Arten von Providern:

ListProvider: *ListProvider* definieren, wie ein bestimmtes Element in einer Liste dargestellt wird. Dazu ist ihnen jeweils ein Typ (*type*) und ein *labelAttribute*, das definiert, wie ein Objekt des Typs dargestellt wird, assoziiert.

TableProvider: *TableProvider* definieren, wie Elemente eines bestimmten Typs in einer Tabelle dargestellt werden. Dazu enthalten sie eine Menge von *ColumnProviders*, die wiederum definieren, wie ein Attribut eines Elements des entsprechenden Typs in einer Spalte dargestellt wird. Dazu ist ihnen neben dem *labelAttribute* eine *Column* assoziiert.

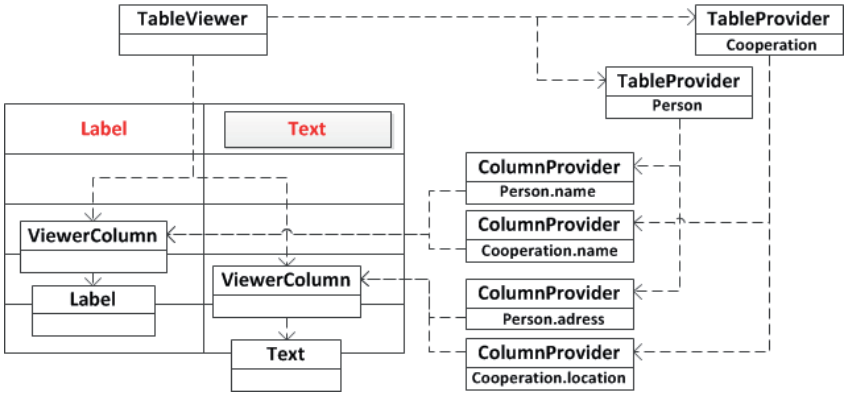


Abbildung 7.31: Zusammenhang Typ, Viewer und Provider

TreeProvider: Wie auch TableProvider verwenden *TreeProvider* ColumnProvider zur Darstellung von Elementen eines bestimmten Typs. Zusätzlich enthält das Attribute *children* eine Liste von Attributen des referenzierten Typs, die Kinder (Child-Elements) des dargestellten Typs enthalten.

CalendarEntryProvider: CalendarEntryProvider unterscheiden sich grundsätzlich von den anderen Providern. Sie enthalten neben der Referenz auf den dargestellten Typ drei Methoden zur Berechnung eines Termins aus einem Objekt des referenzierten Typs:

- *calculateStart* dient zur Berechnung des Startzeitpunkts,
- *calculateEnd* berechnet den Endzeitpunkt,
- *calculateDisplayText* bestimmt den Text, der im Kalender angezeigt wird.

7.6.8 GDM Screen Editor

Wie der GDM Overview Editor ist auch der in Abbildung 7.32 dargestellte GDM Screen Editor ein graphisches Werkzeug. Alle Arten von GuiElementen – GuiContainer, GuiWidgets und Viewer – können aus der Palette auf einen GuiScreen hinzugefügt und frei positioniert werden. Zusätzlich bietet der GDM

Screen Editor die Möglichkeit, GuiElements per Drag & Drop mit UDM Elementen zu verbinden.

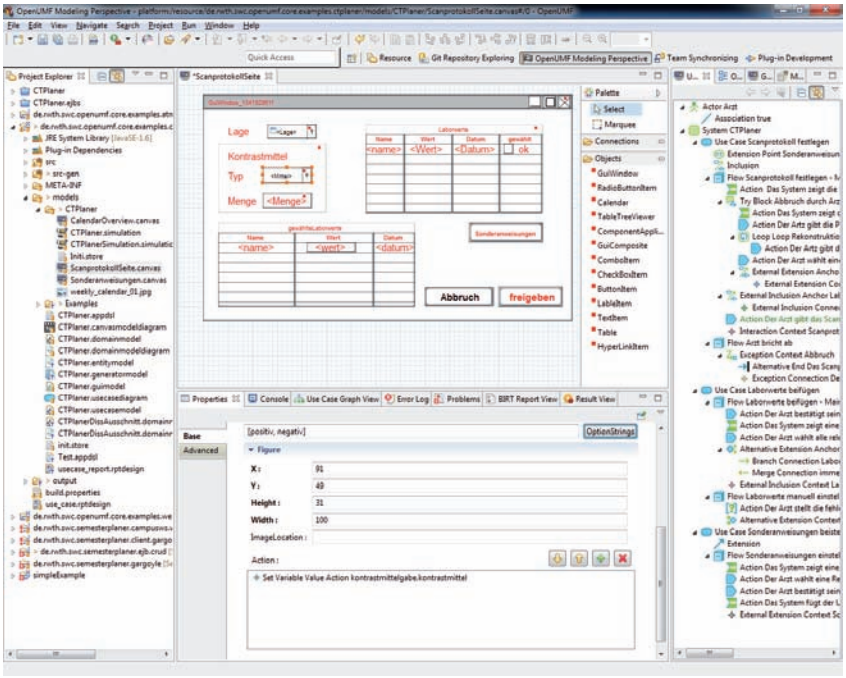
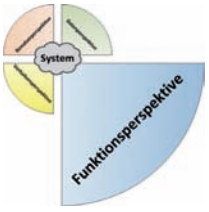


Abbildung 7.32: UCMT – GDM Screen Editor

7.7 SFM - System Function Model



Das System Function Model (SFM) dient der Modellierung der Funktionssicht des entwickelten Systems (vgl. [Bal96, 96 ff.]). Es enthält sämtliche Systemfunktionen, die während des Prototypings aufgerufen werden können, und alle verfügbaren Systemvariablen.

7.7.1 ApplicationFacade

Die Wurzel eines jeden SFM bildet die *ApplicationFacade*. Sie modelliert die fachliche Schnittstelle des Systems. Sie enthält alle Variablen und Funktionen des Systems.

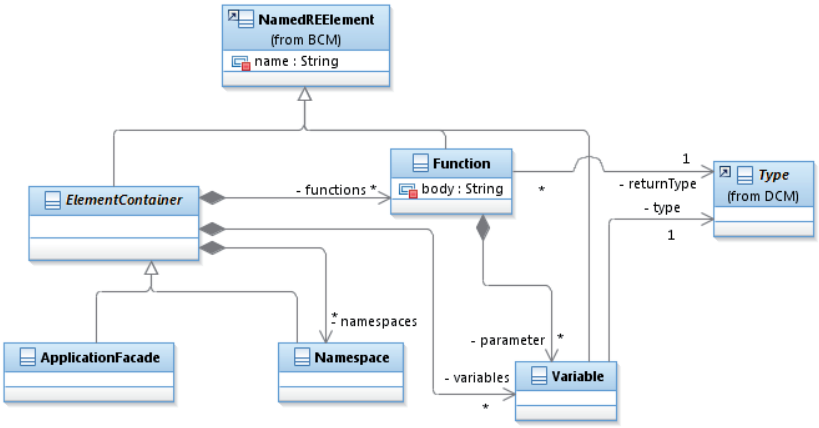


Abbildung 7.33: SFM – Metamodell

7.7.2 Variable

Die Variablen des Sytems werden mit dem Element *Variable* modelliert. Jede Variable hat einen Namen (*name*) und einen Datentyp (*type*). Dabei erlaubt das SFM sowohl die Verwendung von verschiedenen primitiven Datentypen, wie `Integer`, `String` oder `Boolean`, als auch von Kollektionstypen, wie `List` oder `Map`. Zusätzlich können alle Typen verwendet werden, die im DCM (vgl. 7.5) deklariert sind.

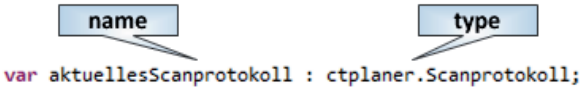


Abbildung 7.34: Beispiel – SFM Variable

7.7.3 Function

Jede *Function* stellt eine Systemfunktion dar. Sie beschreibt sowohl die Schnittstelle der Funktion, bestehend aus aus einem Rückgabetyp (*returnType*) und einer Menge von Parametern (*parameter*), als auch dem Funktionskörper (*body*). Das Attribut *body* beschreibt also die Implementierung der Funktion.

Zusätzlich sind für alle Variablen der ApplicationFacade eine Setter- und Getter-Funktion und eine Fabrikfunktion (Factory-Method) definiert, um auf Variablen zuzugreifen.

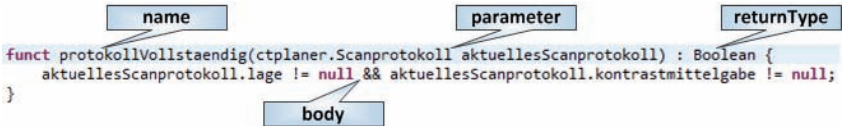


Abbildung 7.35: Beispiel – SFM Function

7.7.4 Namespace

Grundsätzlich reicht es aus, eine ApplicationFacade mit einer Menge von globalen Variablen zu definieren, um das Verhalten eines Systems emulieren zu können. Dies ist in der Praxis oft unhandlich, da eine Vielzahl von Variablen deklariert werden muss und so die Übersichtlichkeit leidet. Deshalb wurde das Konzept *Namespace* als Namensraum für Variablen eingeführt. Ähnlich, wie in Programmiersprachen sind Variablen nur innerhalb eines bestimmten Namespaces definiert. Somit kann die Anzahl, der zu einem Zeitpunkt im System verfügbaren Variablen deutlich reduziert werden, was Verständlichkeit und Analysierbarkeit des SFM deutlich erhöht.

7.7.5 SFM DSL Editor

Das SFM wird mit dem in Abbildung 7.36 dargestellten SFM DSL Editor, einem speziellen Texteditor, spezifiziert. Hier können die Variablen und Funktionen des modellierten Systems definiert werden. Dabei ist es möglich, auf alle im DCM definierten Entitäten, sowie auf Standard Java API zuzugreifen. Der Editor verwendet eine an Java angelehnte Syntax. Somit ist die Sprache leicht für Entwickler zugänglich. Außerdem bietet der SFM DSL Editor automatische Vervollständigung und Validierung an.

7.8 Zusammenhang der verschiedenen UCSM Submodelle

Nachdem die einzelnen Submodelle des UCSM separat betrachtet worden sind, soll nun der Zusammenhang der einzelnen Submodelle diskutiert werden. Dazu werden die relevanten Konzepte der einzelnen Submodelle noch einmal vorgestellt und ihre Beziehungen erörtert. Einen Überblick bietet Abbildung 7.37.

Den Ausgangspunkt für die Integration der verschiedenen Submodelle bilden die Actions im UDM. Sie enthalten, wie bereits erwähnt, neben der textuellen Beschreibung der Aktion, Referenzen auf die GuiWidgets, die das Verhalten darstellen. Zusätzlich können Actions eine FormalDescription enthalten. Diese FormalDescription beschreibt, welche Systemfunktionen bei der Durchführung

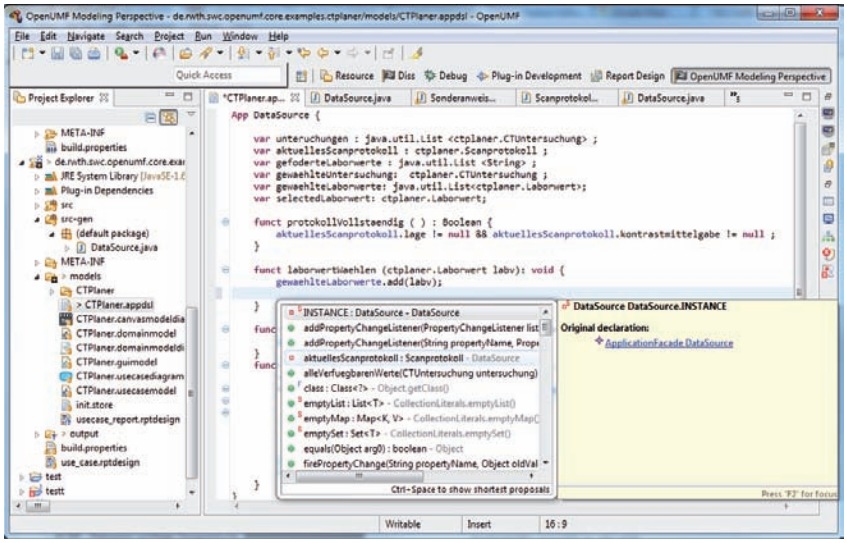


Abbildung 7.36: UCMT – SFM DSL Editor

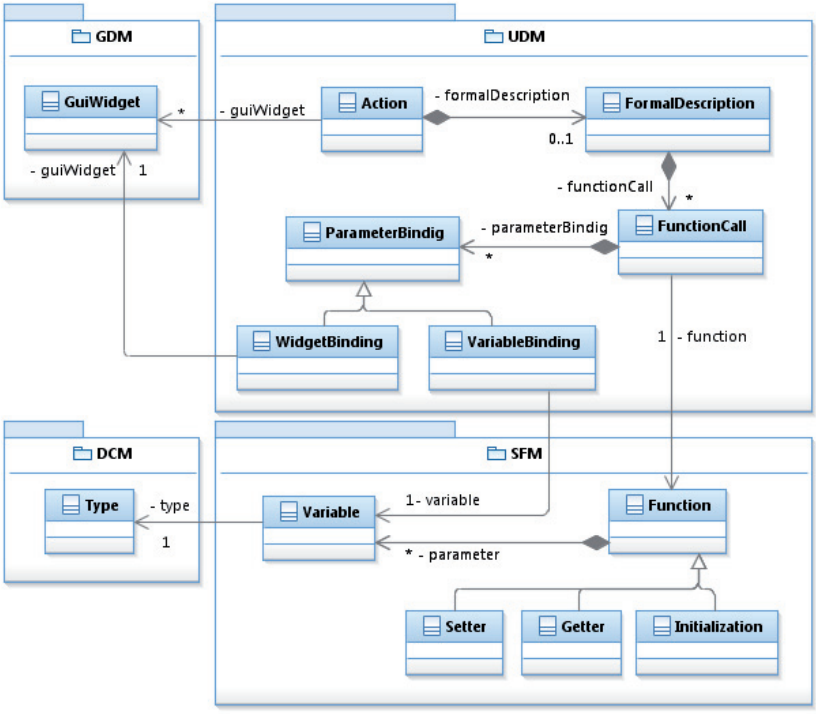


Abbildung 7.37: UCSM Metamodell – Beziehungen zwischen Submodellen

der Aktion ausführt werden. Dazu referenziert jede FormalDescription eine geordnete Liste von *FunctionCalls*. Diese werden in der Simulation der Reihe nach ausgeführt.

Jeder FunctionCall ist mit einer Funktion aus dem SFM verknüpft. Außerdem beschreibt der FunctionCall, welche Variablen der Laufzeitumgebung als Parameter an die Funktion übergeben werden, dazu werden *ParameterBindings* verwendet. Hier gibt es zwei verschiedene Varianten, entweder wird der Funktion eine Variable der ApplicationFacade als Parameter übergeben –hierzu wird ein *VariableBinding* verwendet– oder ein Parameter wird von außen in das System eingegeben. Dazu wird ein *WidgetBinding* verwendet. Dieses ist mit einem der Aktion assoziierten GuiWidget verbunden, das den Parameter bereitstellt. Analog werden auch Systemausgaben mit WidgetBindings modelliert.

Alle Variablen und Parameter des SFM sind mit einem Typ aus dem DCM verbunden.

Eine Beziehung zwischen textuellen Beschreibungen der Action des UDM und den Typen aus dem DCM wird nicht explizit hergestellt, obwohl die Modellierung einer solchen Beziehung von verschiedenen Autoren empfohlen wird. Es hat sich gezeigt, dass eine explizite Modellierung dieser Beziehung von Anwendern als unhandlich empfunden wird. Eine solche Beziehung kann aber leicht durch Vergleich zwischen Beschreibungstext und den Namen der Entities im DCM von einem Werkzeug hergeleitet werden.

7.9 UCSM - Diskussion wichtiger Designentscheidungen

Bei der Entwicklung von UCSM gab es an verschiedenen Stellen Designalternativen. Nun sollen wichtige Entscheidungen diskutiert und mögliche Alternativen erörtert werden.

Erweiterungssemantik: Anders als in der UML2 Spezifikation werden in UCSM Erweiterungsbeziehungen aus Sicht des erweiterten Flusses spezifiziert. Genauer wird die Beziehung zwischen Anker und Kontext sowie die Bedingung, unter der eine Erweiterungsbeziehung realisiert wird, am Erweiterungsanker spezifiziert. Diese Art der Spezifikation wird von den meisten Nutzern präferiert, da alle Use-Case-zentrierten Vorgehensmodelle eine Spezifikation vom allgemeinen zum speziellen Fall empfehlen.

Bedingte Rücksprünge bei alternativen Flüssen: UCSM ermöglicht es, verschiedene Rücksprunganker mit ein und demselben Alternativkontext zu assoziieren. Damit wird ein bedingter Rücksprung möglich. Diese Art der Model-

lierung ist sehr flexibel und in vielen ähnlichen Ansätzen üblich. Allerdings hat sie zwei Nachteile: Entscheidungen über den Fortgang eines Szenarios werden im aufgerufenen Fluss und nicht im aufrufenden Fluss behandelt. Außerdem können mit dieser Art der Beschreibung sehr unübersichtliche Konstruktionen aus verschiedenen Alternativflüssen modelliert werden. Dennoch haben wir uns dafür entschieden, bedingte Rücksprünge in alternativen Flüssen zuzulassen, um flexibel modellieren zu können. Den genannten Problemen begegnen wir mit Hilfe von Regeln und Metriken in der Werkzeugumgebung, die den Anwender auf Probleme hinweisen.

Modellierung von Systemzuständen: Im UCSM wird der Zustand des modellierten Systems, beziehungsweise seiner Teile, ausschließlich implizit durch die Belegung von Variablen sowie Vor- und Nachbedingungen an den Use Cases modelliert. Es gibt kein globales Zustandsmodell, das die Zustände und Übergänge im System explizit darstellt. Grund hierfür ist, dass in der Use-Case-zentrierten Analyse nicht zustandsorientiert spezifiziert wird. Stattdessen werden lediglich lokale Effekte, also die Belegung einiger weniger Variablen zu bestimmten Zeitpunkten, während der Durchführung, modelliert. Konsequenterweise wird auf die Modellierung eines Zustandsmodells verzichtet, da es für die Beschreibung des Systemverhaltens nicht notwendig ist. Nichtsdestotrotz kann ein Zustandsmodell wichtige Erkenntnisse für die weitere Analyse des modellierten Verhaltens liefern. Einige interessante Überlegungen dazu, wie sich ein globales Zustandsmodell aus Use Case Beschreibungen ableiten lässt, geben z.B. Knauss et al. [KLM09].

7.10 UCSM vs. UML

Die UML [OMG07] hat sich in den letzten Jahren als ein defakto Standard für die Modellierung von Softwaresystemen etabliert. Verschiedene Modellierungsmethodiken bauen auf der UML auf und eine große Zahl unterschiedlicher Werkzeuge unterstützen die Modellierung mit der UML. Obwohl sich UCSM an den Konzepten der UML orientiert, z.B. sind die Notationen für Use Cases und Domänenmodelle des UCSM von Notationen der UML abgeleitet, ist das UCSM nicht in das Metamodell der UML, die UML Superstructure, eingebettet. Eine solche Einbettung in die UML Superstructure wäre zwar grundsätzlich möglich. Sie ist allerdings nicht sinnvoll. Hierfür gibt es vier wesentliche Gründe:

- Die UML Superstructure enthält keine Modellierungselemente für die Beschreibung von Nutzerschnittstellen. Diese sind aber ein zentrales Konzept von SUPRA.
- Die Semantik der UML Superstructure ist in vielen Stellen zu unpräzise. Insbesondere der Zusammenhang von Use Cases, Verhaltens- und Klassenmodellen ist lückenhaft.

- In der UML Superstructure fehlt eine pragmatische, kompakte Notation für die Modellierung von Systemverhalten. Präzise kann das Verhalten eines Systems in der UML nur durch die Kombination von Verhaltens- und Objektdiagrammen beschrieben werden. Dies ist für unseren Anwendungsfall unnötig komplex.

- UCSM soll eine kompakte, semantikreiche Modellierung von Use-Case-zentrierten Anforderungsspezifikationen ermöglichen. Hierfür werden weite Teile der UML Superstructure nicht benötigt. Also ist das Metamodell unnötig komplex.

8 Generierung von Prototypen

Wie bereits erwähnt, verfolgt SUPrA einen generativen Ansatz zur Erzeugung von Prototypen. Die Prototypingmodelle werden automatisch aus UCSM Modellen erzeugt. Dazu muss die Ausführungssemantik des UCSM klar definiert sein und ein automatischer Transformationsalgorithmus gefunden werden, mit dem Instanzen des UCSM in Instanzen eines Prototypingmetamodells überführt werden können.

Im Folgenden wird zunächst eine formale Definition der Ausführungssemantik von UCSM auf Basis von gefärbten Petrinetzen vorgestellt. Anschließend wird das IPM, das Prototypingmetamodell von SUPrA, präsentiert und ein Transformationsalgorithmus erklärt.

8.1 Eine formale Ausführungssemantik für UCSM

Um die Ausführung von UCSM Modellen zum Prototyping der Anforderungen zu ermöglichen, muss eine formale Ausführungssemantik für die einzelnen Elemente eines UCSM Modells festgelegt werden. Eine solche formale Semantik ist eine strikt mathematische Definition der Bedeutung der Laufzeitkonzepte der verwendeten Elemente. Sie legt also fest, wie Programme einer spezifischen Sprache, in unserem Fall UCSM, zu interpretieren sind. Für die Definition einer solchen Semantik gibt es prinzipiell mehrere Möglichkeiten:

- **Axiomatische Semantik:** Axiomatische Semantiken definieren Konzepte einer Sprache mit Hilfe von formaler Logik [Hoa69]. Typischerweise wird Prädikatenlogik verwendet. Dabei wird die abstrakte Semantik von Elementen einer Sprache und ihren Relationen in einem Axiomensystem beschrieben. Die speziellen Ausführungseigenschaften der einzelnen Elemente werden als sogenannte Zusicherungen (Assertions) modelliert. Jede Zusicherung besteht dabei jeweils aus einem Prädikat mit Variablen, wobei die Variablen Systemzustände symbolisieren.

- **Operationelle Semantik:** Operationale Semantiken definieren Sprachen in Form von Zustandsänderungen einer abstrakten Maschine [Plo04b]. Dabei werden Systemzustände und mögliche Übergänge zwischen diesen Zuständen modelliert. Zustände definieren i.d.R. eine Menge von Variablenbelegungen. Übergänge zwischen diesen Zuständen werden entweder mit Zustandsübergangsfunktionen oder mit Inferenzregeln modelliert.

- **Denotationelle Semantik:** Denotationelle Semantiken definieren die Eigenschaften einer Sprache, indem Ausdrücke der Sprache durch mathematisch definierte Konstrukte dargestellt werden [Mos90]. Dabei werden i.d.R. Relationen zwischen Eingabe- und Ausgabewerten von Programmen durch mathematische Funktionen dargestellt.

- **Translatorische Semantik:** Translatorische Semantiken [LL81] definieren die Elemente einer Sprache, indem sie diese in eine andere Sprache mit einer definierten Semantik überführen. Dabei werden die Konstrukte der Ausgangssprache einzeln in die Konzepte der *Zielsprache* überführt. Translatorische Semantiken können also als Spezialform der denotationellen Semantiken angesehen werden.

- **Pragmatische Semantik:** Bei pragmatischen Semantiken wird auf eine abstrakte mathematische Definition einer Sprache verzichtet. Stattdessen wird die Semantik der Sprache durch eine Referenzimplementierung festgelegt.

Eine pragmatische Implementierung der Ausführungssemantik des UCSM ist durch die Referenzimplementierung in OpenUMF (vgl. Kap. 11) gegeben.

Zusätzlich wurde eine translatorische Semantik definiert, die die Elemente von UCSM auf gefärbte Petrinetze zurückführt. Diese Art der Darstellung ermöglicht eine sehr kompakte Beschreibung der Semantik, da sich die flussorientierten Konzepte von UCSM sehr einfach mit gefärbten Petrinetze abbilden lassen. Weiterhin haben die entstehenden Petrinetze spezielle topologische Eigenschaften, die die inkrementelle Entwicklung von Use Case Prototypen ermöglichen. Bevor die translatorische Semantik im Folgenden detailliert vorgestellt wird, soll zunächst diskutiert werden, warum gefärbte Petrinetze zur Definition der Semantik gewählt wurden. Zusätzlich werden die für das Verständnis der Semantik relevanten Konzepte gefärbter Petrinetze eingeführt.

8.1.1 Diskussion möglicher Ausführungssemantiken für UCSM

Die Ausführungssemantik von UCSM wird mit einer translatorischen Semantik definiert. Dies hat einen entscheidenden Vorteil: Wenn eine geeignete Zielsprache verwendet wird, kann diese auch als Prototypingnotation dienen. So kann also gleichzeitig eine Semantik für UCSM definiert und die Grundlage für die automatische Generierung von Prototypen gelegt werden. Damit eine Semantikdefinition möglich ist, muss also die Zielsprache der translatorischen Semantik den in Kapitel 5.3.4 gestellten Anforderungen an die Prototypingnotation genügen. Sie muss also eine ausreichende Ausdrucksstärke (P1) besitzen und für große UCSM Modelle skalieren (P2). Außerdem muss sie Möglichkeiten zur Komposition von Modellteilen anbieten (P3) und eine Topologie aufweisen, die robust gegen Änderungen im UCSM ist (P4).

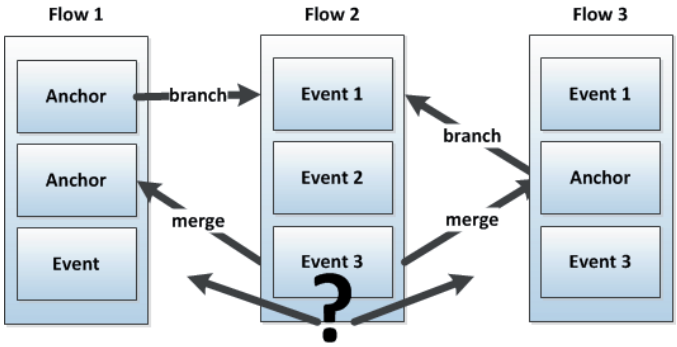


Abbildung 8.1: Rücksprung bei mehreren Kontexten

Kritisch ist hier vor allem die Ausdrucksstärke der Zielsprache (P1). Im UCSM ist der Rücksprungpunkt am Ende eines Flusses abhängig davon, in welchem Kontext er aufgerufen wurde. Hat also ein Fluss, wie in Abbildung 8.1 angedeutet, mehr als einen Kontext (z.B. Flow 2 in Abb. 8.1), so muss am Ende der Ausführung des Flusses abhängig vom Kontext der Rücksprunganker ermittelt werden. Dieser Zusammenhang ist kontextabhängig (vgl. [RG97]). Demzufolge lässt er sich nicht exakt durch eine kontextfreie Grammatik darstellen. Somit sind Formalismen, die kontextfreie Grammatiken darstellen können, als Zielsprache für eine translatorische Semantik ungeeignet.

Es muss also eine mächtigere, kontextsensitive Zielsprache gewählt werden. Prinzipiell sind allgemeine Petrinetze [Pet62] zwar in der Lage, kontextsensitive Sprachen exakt zu beschreiben, allerdings bieten sie weder Kompositionsmöglichkeiten (P3) noch ist es möglich, ein deterministisches Petrinetz zu beschreiben, das eine robuste Topologie (P4) aufweist. Somit kann dieser Formalismus ebenfalls nicht verwendet werden. Mit gefärbten Petrinetzen ist hingegen eine Darstellung möglich, die alle geforderten Eigenschaften aufweist. Deshalb wurde dieser Formalismus als Zielsprache gewählt.

8.1.2 Exkurs: Gefärbte Petrinetze

Die von Carl Adam Petri 1961 [Pet62] eingeführten Petrinetze sind ein weit verbreiteter Formalismus zur Spezifikation von schrittweise ablaufenden Prozessen. Sie definieren Mechanismen zur Darstellung von bedingter Auswahl, Iteration und Nebenläufigkeit. Petrinetze sind bipartite gerichtete Graphen. Sie bestehen aus Stellen, Transitionen und gerichteten Kanten, die jeweils von einer Stelle zu einer Transition oder umgekehrt laufen.

Die Ausführungssemantik von Petrinetzen basiert auf Marken (Tokens), die entlang den Kanten des Netzes weitergegeben werden. Im Unterschied zu vielen an-

deren populären Ausführungsmodellen, wie z.B. UML2 Aktivitätsdiagrammen [OMG07], haben sie eine exakte mathematisch definierte Ausführungssemantik und eine fundierte mathematische Theorie für die Prozessanalyse.

Gefärbte Petrinetze sind eine von Kurt Jensen eingeführte Erweiterung allgemeiner Petrinetze [Jen91]. Sie wurden entwickelt, um auf die Skalierungsprobleme allgemeiner Petrinetze zu reagieren. Ziel war es, einen Formalismus zu entwickeln, der gleichzeitig theoretisch fundiert und flexibel genug für die Anwendung in industriellen Problemen ist. Gefärbte Petrinetze kombinieren die Stärken von Petrinetzen mit allgemeinen Programmiersprachenkonzepten, wie der Definition von Datentypen und dem Verändern von Datenwerten. Dadurch wird eine deutlich kompaktere Beschreibung der modellierten Zusammenhänge möglich, ohne die Ausdrucksstärke der Notation einzuschränken.

Die Grundidee gefärbter Netze ist die Verwendung sogenannter gefärbter Marken. Diese sind im Unterschied zu Marken in allgemeinen Petrinetzen unterscheidbar. Dadurch kann das Schaltverhalten von Transitionen abhängig von der Art der Marken gemacht werden. Genauer gesagt, tragen die Marken Attribute. Die Werte dieser Attribute werden hinzugezogen, um zu entscheiden, ob eine bestimmte Transition feuern kann oder nicht. Dazu werden im gefärbten Petrinetz zusätzlich Typen definiert, die Marken und ihren Attributen zugeordnet werden.

Weiterhin führt Jensen ein Modulkonzept ein. Dieses ermöglicht es, zum einen gefärbte Petrinetze hierarchisch zu strukturieren und zum anderen Teilnetze an unterschiedlichen Stellen eines gefärbten Petrinnetzes wiederzuverwenden.

8.1.2.1 Informelle Definition

Im Folgenden soll eine kurze informelle Zusammenfassung der Semantik von gefärbten Petrinetzen gegeben werden. Eine formale Definition der Konzepte liefert Jensen [Jen91]. Bei der Betrachtung gefärbter Petrinetze unterscheiden wir zwei Aspekte: Als *Topologie* bezeichnen wir die Struktur eines gefärbten Netzes bestehend aus Stellen, Transitionen, Kanten, Inskriptionen und Typen. Das *Verhalten* hingegen bezeichnet die dynamische Schaltlogik eines gefärbten Petrinetzes. Sie besteht aus gefärbten Marken und ihren Positionen im Netz.

Topologie: Wie in allen anderen Petrinetzarten sind gefärbte Petrinetze aus *Stellen* und *Transitionen* aufgebaut, die mit *Kanten* verbunden sind. Stellen und ihre *Marken* repräsentieren Systemzustände. Transitionen stellen Übergänge zwischen diesen dar. Jede Stelle kann hierbei mehrere Marken halten, die jeweils eine Instanz eines beliebigen Datentyps sind. Dabei wird der Datentyp einer Marke als ihre *Farbe* bezeichnet.

Die Topologie gefärbter Petrinetze besteht aus drei Teilen: der *Netzstruktur*

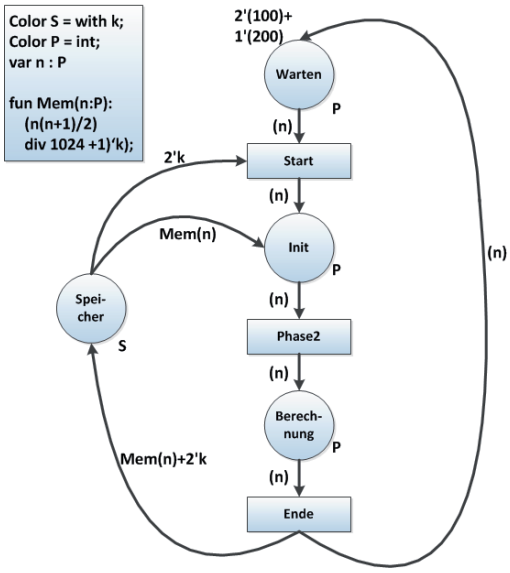


Abbildung 8.2: Gefärbtes Petrinetz – Beispiel

(net structure), dem *Deklarationsblock* (declaration) und den *Netzinskriptionen* (inscriptions).

- Die *Netzstruktur* entspricht der Struktur von ungefärbten Petrinetzen. Sie ist ein gerichteter bipartiter Graph bestehend aus Stellen (place) und Transitionen (transition) sowie Kanten (arcs), die sie verbinden.

- Der *Deklarationsblock* enthält drei Arten von Deklarationen. Zunächst werden *Farbmengen* (color set) beschrieben. Diese definieren analog zu Typen in Programmiersprachen nicht nur die Farben, die die Elemente der Farbmenge sind, sondern auch Operationen, die auf den Farben ausgeführt werden können. Zusätzlich können Variablen (variable) definiert werden. Diese sind jeweils einer bestimmten Farbmenge zugeordnet. Sie haben also einen Typ. Außerdem enthält der Deklarationsblock Funktionen (function), die in Netzinskriptionen verwendet werden können.

- *Netzinskriptionen* beschreiben zusätzliche Informationen über das Schaltverhalten des Netzes. Sie können Stellen, Transitionen oder Kanten zugeordnet werden. Dabei haben die unterschiedlichen Elemente der Netzstruktur jeweils unterschiedliche Arten von Inskriptionen:

  – Stellen haben drei Arten von Inskriptionen:

    Namen, die keine Ausführungssemantik haben und lediglich der Identifikation dienen,

Farbmengen, die definieren, welche Art von Marken in einer Stelle vorkommen darf,

Initialisierungsausdrücke (initialization expressions), die jeweils eine Menge von Marken beschreiben, die bei der Initialisierung des Netzes an der Stelle vorhanden sind.

- Transitionen haben neben dem Namen noch einen *Kontrollausdruck* (guard expression). Jeder Kontrollausdruck formuliert einen booleschen Ausdruck, der erfüllt sein muss, damit die Transition feuern kann.

- Arcs haben lediglich *Kantenausdrücke* (arc expression). Diese beschreiben Funktionen, die jeweils zu einer Farbe oder einer Multimenge von Farben ausgewertet werden. Dazu können, wie auch bei Kontrollausdrücken, Variablen und Funktionen aus dem Deklarationsblock verwendet werden.

Verhalten: Die Definition des Verhaltens von gefärbten Petrinetzen ist relativ einfach. Grundsätzlich wird die Dynamik des Netzes, wie auch bei ungefärbten Petrinetzen, durch das Feuern von Transitionen beschrieben. Dies bedeutet, dass durch Aktivieren einer Transition Marken von bestimmten Stellen im Netz (Vorbereich) entfernt und stattdessen Marken in anderen Stellen des Netzes (Nachbereich) hinzugefügt werden. Dabei wird eine Verteilung von Marken auf Stellen im Netz als Markierung bezeichnet. Die initiale Markierung wird hierbei durch Auswertung aller Initialisierungsfunktionen bestimmt.

Eine Transition ist aktiviert, wenn alle Stellen im Vorbereich der Transition eine ausreichende Anzahl von Marken halten. Zusätzlich muss eine legale *Bindung* vorliegen. Als Bindung bezeichnet man eine Zuweisung von Farben einer Marke zu den Variablen einer Transition. Wenn dabei alle Kantenausdrücke eingehender Kanten und der Kontrollausdruck erfüllt sind, bezeichnet man dies als legale Bindung.

Eine Transition kann grundsätzlich auf so viele verschiedene Arten feuern, wie legale Bindungen möglich sind; also auf so viele Arten, wie die Variablen der Marken, unter Berücksichtigung der Kantenausdrücke und Kontrollausdrücke, gebunden werden können. Jensen merkt allerdings an, dass normalerweise nur relativ wenige verschiedene Bindungen zu einer Markierung aktiviert sind [Jen91].

Wenn also eine Transition für eine Bindung aktiviert ist, kann sie feuern (occur). Dabei werden Marken von den *Eingangsstellen* (input places) im *Vorbereich* der Transition entfernt und Marken in den *Ausgangsstellen* (output places) im *Nachbereich* hinzugefügt. Anzahl und Farbe der gelöschten und hinzugefügten Marken hängen dabei von der Auswertung der jeweiligen Kantenausdrücke und den zugehörigen Bindungen ab.

8.1.3 Translatorische Semantik von UCSM

Im Folgenden wird die translatorische Semantik von UCSM detailliert beschrieben. Dazu wird zunächst die Grundidee der Transformation eingeführt. Anschließend werden die Transformationsregeln für alle ausführungsrelevanten Elemente des UCSM vorgestellt. Parallel stellen knappe Beispiele dar, wie die verschiedenen Einzelkomponenten kombiniert werden können. Zur Darstellung der Petrinetzelemente, die das Ergebnis der einzelnen Transformationsregeln sind, wird die in Abbildung 8.3 dargestellte graphische Notation von gefärbten Petrinetzen verwendet.

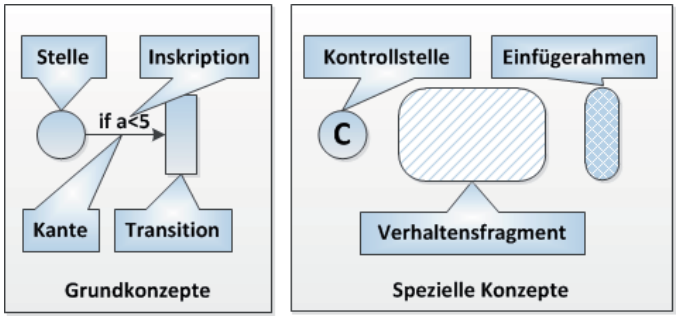


Abbildung 8.3: Semantikdefinitionsnotation – Überblick

Bevor nun einige Vorüberlegungen zur Topologie einer geeigneten Ausführungssemantik gemacht werden können und die translatorische Semantik präsentiert werden kann, müssen zunächst drei Begriffe eingeführt werden:

- **Ausführungsrelevantes Element:** Als *ausführungsrelevante Elemente* werden diejenigen Elemente des UCSM bezeichnet, die Einfluss auf die Ablaufsemantik des modellierten Systems haben. Dies sind zum einen Flüsse, Ereignisse und Kontexte, zum anderen werden die Daten und Funktionsbeschreibungen des UCSM zu den ausführungsrelevanten Elementen gezählt.

- **Ablaufelement:** *Ablaufelemente* sind die Elemente des Anforderungsmodells, die den Kontrollfluss des modellierten Systems spezifizieren. Genauer sind Flüsse, Ereignisse und Kontexte Ablaufelemente.

- **Ausführungsmodell:** Das gefärbte Petrinetz, das die Ausführungssemantik eines UCSM Modells definiert, wird im Folgenden als das *Ausführungsmodell* des UCSM Modells bezeichnet.

Wichtig ist hier die Beobachtung, dass die Elemente des GDM keine ausführungsrelevanten Elemente sind. Demzufolge sind sie auch in der Ausführungssemantik nicht repräsentiert. Sie werden lediglich während der Prototypbewertung zur Erzeugung der Nutzerschnittstelle des Prototyps verwendet. Hierauf wird in Kapitel 9 detailliert eingegangen.

8.1.4 Vorüberlegungen

Wie bereits in Kapitel 5.3 diskutiert, muss das Ausführungsmodell das Verhalten des UCSM exakt wiedergeben und zusätzlich eine kompakte, robuste Topologie aufweisen.

Eine wichtige Forderung an das Ausführungsmodell ist, dass es auch für große Modelle, wie sie in der industriellen Praxis vorkommen, skaliert. Es muss insbesondere auch dann verwendbar sein, wenn das UCSM Modell Schleifen enthält und deshalb potentiell unendlich viele Szenarien besitzt. Damit die Struktur auch in diesen Fällen, in denen unendliche viele Pfade durch den Ereignisfluss möglich sind, sicher eine endliche Struktur aufweist, muss die Semantik so aufgebaut sein, dass jedes ausführungsrelevante Element nur endlich oft vorkommt. Die Semantik muss also direkt gegen diese Elemente spezifiziert werden und kann nicht aus den im UCSM Modell enthaltenen Szenarien aufgebaut werden.

Weiterhin ist gefordert, dass jede Änderung am UCSM Modell lediglich lokale Effekte auf das Ausführungsmodell haben darf. Deshalb darf jedes ausführungsrelevante Element maximal einmal im Petrinetz repräsentiert sein.

Also folgt aus diesen beiden Forderungen, dass jedes ausführungsrelevante Element *genau* einmal im Petrinetz dargestellt werden muss.

Somit muss eine Petrinetztopologie entstehen, die der flussorientierten Darstellung der Use Case Spezifikation sehr ähnlich ist. Sie muss ebenfalls aus Verhaltensblöcken bestehen, aus denen die Einzelszenarien kombiniert werden. Damit eine exakte Abbildung des im UCSM spezifizierten Verhaltens möglich ist, muss der Ausführungszustand im Netzverhalten auf den Marken abgebildet werden. Dies wird durch eine spezielle Netztopologie erreicht, die im Folgenden vorgestellt wird.

8.1.5 Grundidee

Allgemein ist jedes Ausführungsmodell aus *Verhaltensfragmenten* (behavioral fragment) aufgebaut. Jedes dieser Verhaltensfragmente besteht aus einer Menge von Petrinetzelementen und realisiert die Semantik eines Ablaufelements. Dabei ist der Grundaufbau aller Verhaltensfragmente gleich. Für die weitere Beschreibung sind vier Elemente des Verhaltensfragments wichtig. Diese werden auch in Abbildung 8.4 dargestellt:

- **Einstiegsstelle:** Die Einstiegsstelle (entry place) verknüpft ein Verhaltensfragment mit seiner Außenwelt. Nur wenn sie eine Marke enthält, kann das Verhalten im Verhaltensfragment ausgeführt werden.
- **initiale Transition:** Die initiale Transition (inital transition) ist die erste Transition eines Verhaltensfragments. Sie bildet den Einstieg in das

modellierte Verhalten. Sie enthält i.d.R. einen Kontrollausdruck, der den Ausführungskontext kontrolliert.

- **finale Transition:** Die finale Transition (final transition) ist die letzte Transition einen Verhaltensfragments. Mit ihr wird das Verhalten eines Verhaltensfragments abgeschlossen.

- **Ausstiegsstelle:** Die Ausstiegsstelle (exit place) bildet das Ende des Verhaltensfragments. Sie enthält die Ergebnisse der Ausführung des Subnetzes.

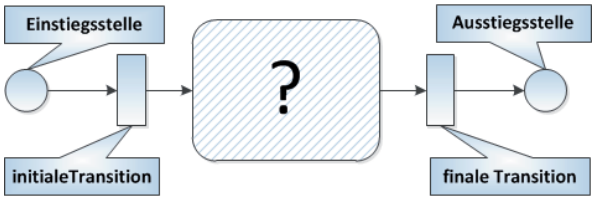


Abbildung 8.4: Verhaltensfragmente – Allgemeiner Aufbau

Wie bereits erwähnt, realisiert jedes Verhaltensfragment ein Ausführungselement. Im Folgenden werden die Verhaltensfragmente jeweils analog zu ihren zugehörigen Ausführungselementen bezeichnet. Das Verhaltensfragment einer Action heißt also beispielsweise *ActionFragment*.

Die Grundidee des Aufbaus des Ausführungsmodells ist, dass ausschließlich Actions *Verhaltensatome* (behavioral atoms) sind. Alle anderen Arten von Elementen werden als sogenannte *Verhaltensrahmen* (behavioral frames) betrachtet. Diese Verhaltensrahmen enthalten jeweils mindestens ein weiteres Verhaltensfragment (vgl. Abb. 8.5). Genauer enthalten *FlowFragments* die *EventFragments* der Events im Flow. *ContextFragments* enthalten die FlowFragments der assoziierten Flows und *AnchorFragments* enthalten ihre referenzierten Context-Fragments. Folglich ist das gesamte Ausführungsmodell aus Verhaltensrahmen aufgebaut.

Die enthaltenen Verhaltensfragmente bilden aus Sicht der Verhaltensrahmen eine Black-Box. Dabei wird das Verhaltensfragment lediglich durch die initiale Transition und die finale Transition des Verhaltensrahmens eingefasst. Genauer formuliert, ist die Einstiegsstelle des Verhaltensfragments immer im Nachbereich der initialen Transition des Verhaltensrahmens. Analog ist die Ausstiegsstelle des Verhaltensfragments im Vorbereich der finalen Transition.

Ausführungsstack: Wie eingangs erwähnt, muss der Ausführungskontext, der für eine exakte Verhaltenssimulation nötig ist, wegen der Anforderungen an das Ausführungsmodell im Netzverhalten modelliert werden. Dies lässt sich auf der gegebenen Netztopologie leicht durch Einführung eines Kellerspeichers errei-

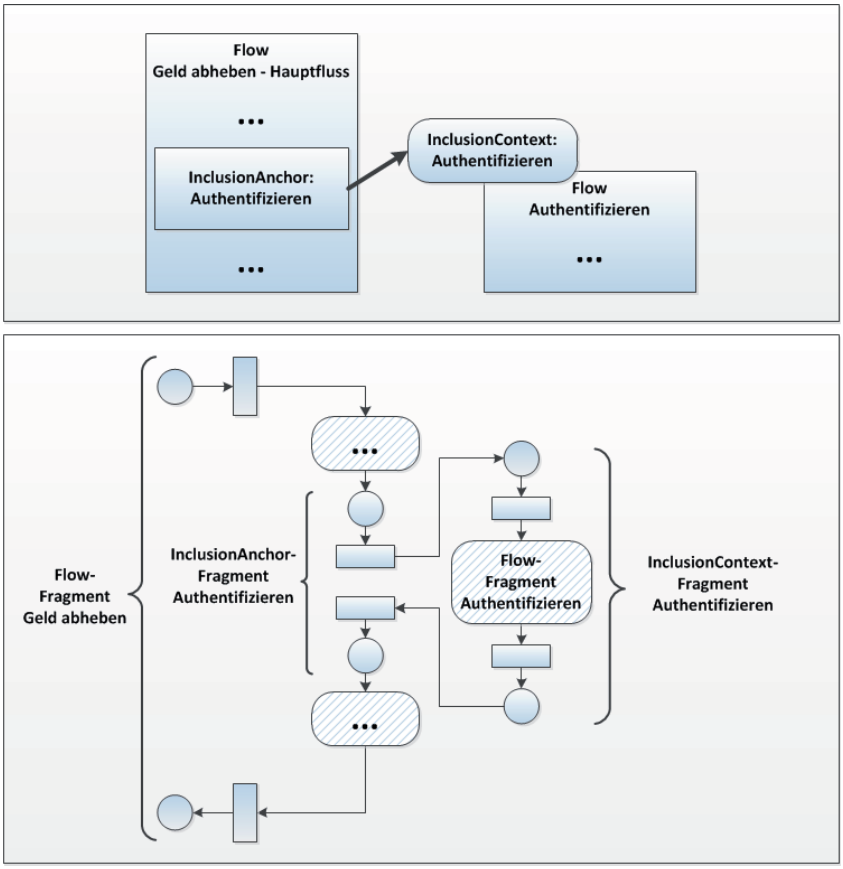


Abbildung 8.5: Verhaltensrahmen

chen. Jede Marke im Netz erhält dazu ein spezielles Attribut, den *Ausführungsstack* (ExecutionStack), das den aktuellen Ausführungskontext speichert.

Der Ausführungsstack einer Marke wird dabei bei der Netzausführung von den Transitionen der Verhaltensrahmen folgendermaßen gefüllt:

Jede initiale Transition schreibt eine Information über das Erreichen des zugehörigen Ausführungselements in den Kellerspeicher. Diese Information wird nach Ausführung des Verhaltensfragments von der finalen Transition wieder vom Ausführungsstack entfernt. Damit wird die Entscheidung, ob ein bestimmtes Verhalten ausführbar ist, vom Zustand des Ausführungsstacks abhängig gemacht. Eine Transition, die beispielsweise einen Kontext darstellt, kann nur dann feuern, wenn vorher der verbundene Anker seine Information auf den Ausführungsstack gelegt hat. Diese Schaltsemantik wird durch Kontrollausdrücke an den Transitionen modelliert. Dieser Zusammenhang wird in Abbildung 8.6 noch

einmal verdeutlicht. Der Ausführungsstack selbst wird durch einen klassischen Kellerspeicher realisiert, auf den mit `push` und `pop` zugegriffen wird.

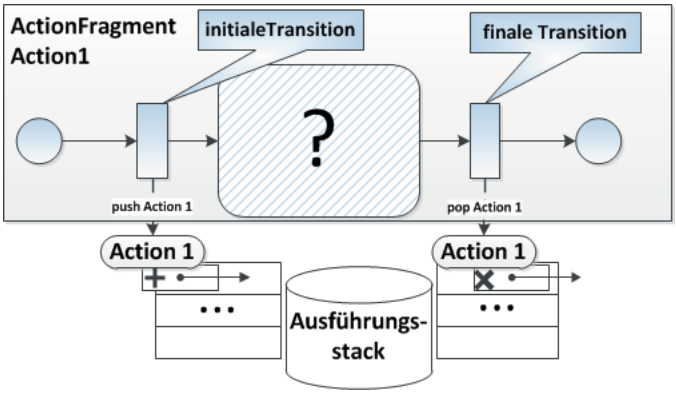


Abbildung 8.6: Verhaltensfragmente & Ausführungskontext

Typen, Variablen & Funktionen: Wie bereits in Abschnitt 7.1 diskutiert, kann das Verhalten eines Systems im UCSM teilweise oder komplett formal beschrieben werden. Dazu können im UCSM Typen, Variablen und Funktionen definiert werden. Diese bilden den Deklarationsblock des Ausführungsmodells. Abbildung 8.7 stellt die entsprechenden Konzepte noch einmal graphisch dar. Konkret werden die Elemente folgendermaßen abgebildet:

- **Datentypen:** Alle Datentypen, die im DCM definiert sind, werden durch Farbmengen dargestellt.
- **Variablen:** Die im SBM definierten Variablen werden analog als Variablen im Ausführungsmodell repräsentiert. Dabei werden die den Datentypen entsprechenden Farben als Typ verwendet.
- **Funktionen:** Alle in der ApplicationFassade des SMB definierten Funktionen sind auch im Ausführungsmodell verfügbar. Analog zu den Variablen werden die Typen der Parameter durch die entsprechenden Farben repräsentiert. Die Verwendung der Funktion und die Übergabe von Parametern wird im Abschnitt zu ActionFragments diskutiert.

Damit die Variablen des Systems während der Netzausführung verfügbar sind, enthält jede Marke neben dem Ausführungsstack für jede definierte Variable ein Attribut. Bei dieser Konstruktion enthalten die Marken zwar potentiell zu viele Attribute, dies ist jedoch vertretbar, da sich immer nur relativ wenige Marken im Netz befinden.

Außerdem ist diese Konstruktion sehr flexibel. Wenn es dem Nutzer beispielsweise erlaubt sein soll, Variablen in nicht formalisierten Aktionen manuell zu

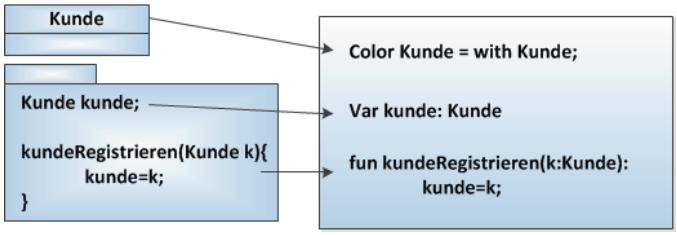


Abbildung 8.7: Ausführungsmodell – Deklarationsblock

manipulieren, müssen alle definierten Variablen zu jeder Zeit verfügbar sein.

Nutzereingaben: Nutzereingaben spielen bei der Ausführung von UCSM eine besondere Rolle. Während der Ausführung eines UCSM Modells muss es möglich sein, Nutzereingaben von außen in das Ausführungsmodell einzulesen. Hierzu werden, analog zur Idee von Holloway et al. [HK94] für ungefärbte Netze, sogenannte *Kontrollstellen* verwendet. Genauer gesagt haben die Transitionen der ActionFragments eine spezielle Kontrollstelle in ihrem Vorbereich. Diese Stellen enthalten Marken, die während der Simulation von außen initialisiert werden, sobald sie in der Transition konsumiert werden sollen. Die Marken dieser Kontrollstellen enthalten dann, wie in Abbildung 8.8 dargestellt, jeweils ein Attribut für jedes WidgetBinding der zugehörigen Aktion. Somit können Nutzereingaben im Ausführungsmodell realisiert werden, ohne dass ihre konkrete Belegung zu Beginn der Ausführung festgelegt werden muss. Der Vorteil dieser Art der Modellierung ist, dass die Lebendigkeit (liveness) des Netzes weiterhin lokal bewertet werden kann.

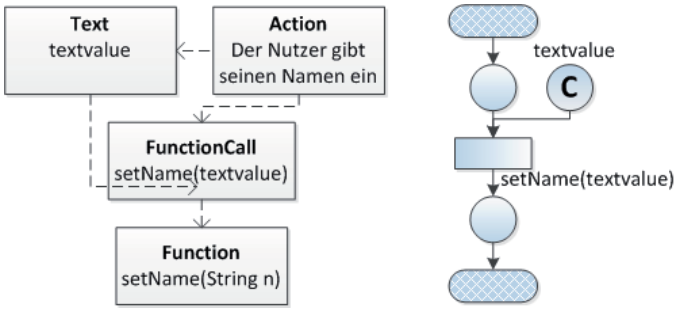


Abbildung 8.8: Ausführungsmodell – Kontrollstellen

8.1.6 Transformation von Elementen

Nachdem die allgemeinen Konzepte der Aussführungssemantik diskutiert wurden, soll nun die Definition der Ausführungssemantik gegeben werden. Dazu

werden der Reihe nach die Transformationsregeln für die verschiedenen Arten von Ausführungselementen vorgestellt.

8.1.6.1 Flow

Jeder Flow stellt einen Verhaltensrahmen für die enthaltenen Events dar. In Abbildung 8.9 erkennt man, dass die initiale und die finale Transition des FlowFragments die EventFragments des Flows einrahmen. Das FlowFragment selbst wird wiederum von seinen ContextFragments eingerahmt.

Abbildung 8.9: FlowFragment

8.1.6.2 Event

Alle Events sind Elemente des Ereignisflusses. Sie haben genau einen Vorgänger und einen Nachfolger. Diese Struktur muss im Ausführungsmodell erhalten bleiben. Deshalb ist jedes EventFragment zwischen ein Vorgänger- und ein Nachfolgerfragment eingebettet (vgl. Abb. 8.10). Damit ist die Ausstiegsstelle des Vorgängerfragments die Einstiegsstelle des EventFragments. Analog ist die Ausstiegsstelle des EventFragments die Einstiegsstelle des Nachfolgerfragments.

Vorgänger und Nachfolger eines Events hängen dabei von dessen Position im Ereignisfluss ab. Hier gibt es grundsätzlich drei Konstellationen:

- **Erstes Event:** Der Vorgänger des ersten Events eines Flows ist der Flow selbst. Somit ist die Einstiegsstelle des EventFragments im Nachbereich der initialen Transition des Flows.
- **Letztes Event:** Analog ist der Flow der Nachfolger des letzten Events eines Flows. Also ist die Ausstiegsstelle des EventFragments im Vorbereich der finalen Transition des FlowFragments.
- **Inneres Event:** Alle anderen Events sind innere Events. Der Vorgänger ist das Event, das vor ihnen im Ereignisfluss steht. Nachfolger eines inneren Events ist das Event hinter ihm im Fluss.

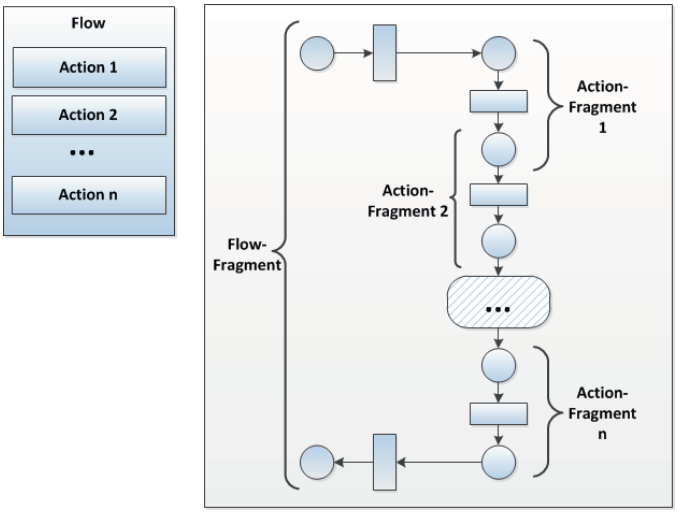


Abbildung 8.10: Ereignisfluss – Vorgänger & Nachfolger

Action: Actions sind, wie bereits erwähnt, Verhaltensatome. Actions können durch eine einzelne Transition repräsentiert werden. Genauer gesagt fallen also die initiale und die finale Transition des *ActionFragments* aufeinander. Der FunktionCall, der einer Action assoziiert sein kann, wird, wie in Abbildung 8.8 dargestellt, durch Kontrollausdrücke realisiert. Zusätzlich befindet sich im Vorbereich der Transition eine Kontrollstelle, über die Parameter mit einem WidgetBinding von außen eingelesen werden können (vgl. Abb. 8.11). Alle anderen Parameter der Funktion werden aus den Marken des Bindung gefüllt.

Anchor: Anchors dienen dazu, die Verbindung zwischen einem aufrufenden Flow mit einem oder mehreren aufrufbaren Flows herzustellen. Um diese Schaltsemantik darstellen zu können, enthalten alle AnchorFragmente zwei zusätzliche Elemente. Der *DecisionPlace* eines jeden AchorFragments modelliert die Stelle, an der in einen aufrufbaren Flow verzweigt wird. Am *MergePlace* kehrt der Kontext während der Ausführung eines Szenarios vom aufgerufenen in den aufrufenden Flow zurück (vgl. Abb. 8.11).

InclusionAnchor: Jeder InclusionAnchor ist genau an einen Kontext gebunden und der zugehörige Fluss muss immer ausgeführt werden, wenn der InclusionAnchor erreicht wird. Deshalb sind die initiale und die finale Transition des InclusionAnchorFragments nicht verbunden. Weiterhin zeigt Abbildung 8.11, wie der Ausführungsstack manipuliert wird.

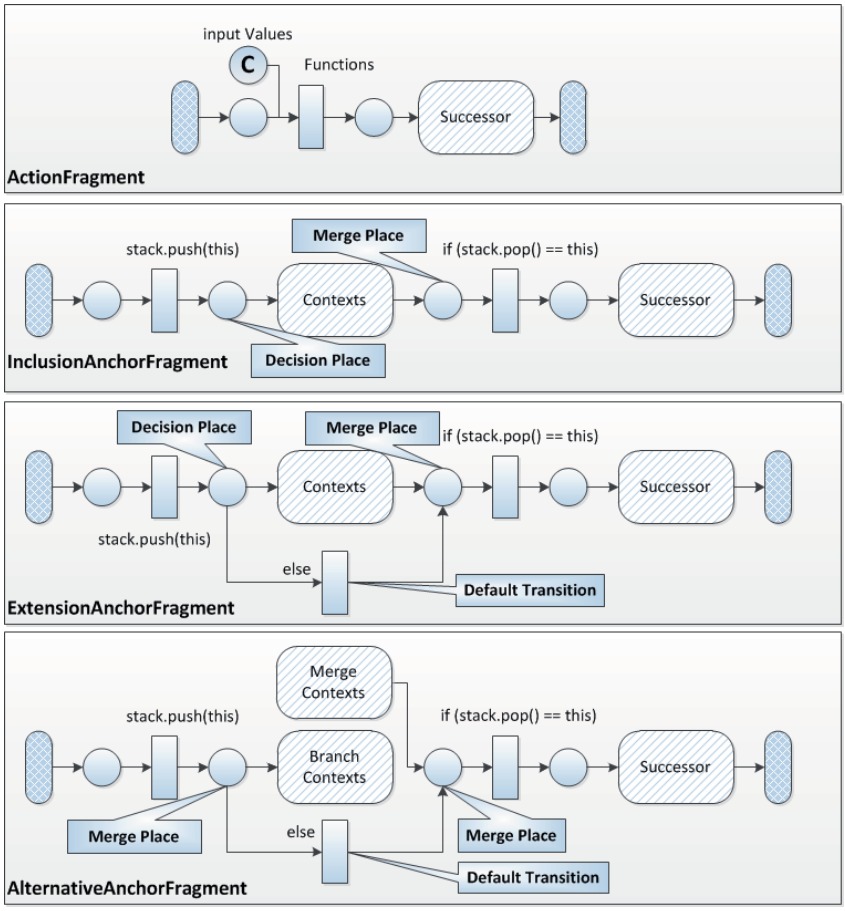


Abbildung 8.11: EventFragments

ExtensionAnchor: ExtensionAnchors können im Unterschied zu InclusionAnchors gleichzeitig an mehrere ExtensionContexts gebunden sein. Deshalb enthalten sie potentiell mehrere Verhaltensfragmente, die jeweils einen dieser Kontexte enthalten. Zusätzlich kann es vorkommen, dass keiner der gebundenen ExtensionContexts ausgeführt wird, da keine der modellierten Bedingungen zutrifft. Deshalb enthält das ExtensionAnchorFragment die sogenannte *Default-Transition*, die die initiale und die finale Transition des ExtensionAnchorFragments verbindet.

Damit die Schaltsemantik der Erweiterungsbeziehung korrekt abgebildet wird, wird hier eine Ordnung der am *DecisionPlace* hängenden Transitionen angenommen. Die Verhaltensfragmente der einzelnen ExtensionContexts werden dabei in der Reihenfolge, in der sie im UCSM definiert sind, evaluiert (vgl. Abb. 8.12).

Sollte keines der Verhaltensfragmente aktivierbar sein, wird die DefaultTransition gefeuert.

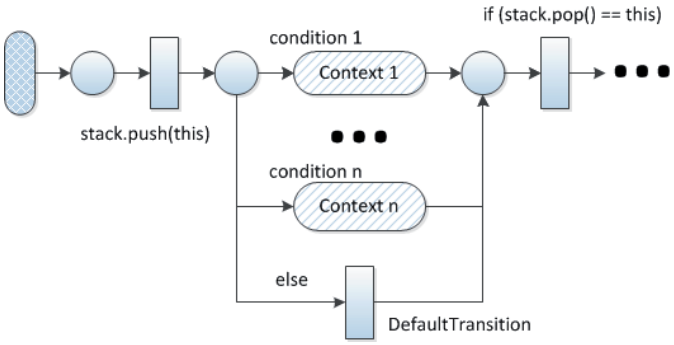


Abbildung 8.12: Schaltsemantik Erweiterung

Somit benötigt die DefaultTransition keinen Kontrollausdruck. Außerdem wird der Nichtdeterminismus aufgelöst, der durch nicht-überdeckungsfreie Bedingungen an den verbundenen Kontexten entsteht. Dies entspricht zwar nicht 100%ig dem spezifizierten Verhalten, es stellt aber die Konzepte einer möglichen Implementierung expliziter dar. In der Implementierung entsteht durch die Reihenfolge der Fallunterscheidungen im Quellcode ebenfalls eine Ordnung.

AlternativeAnchor: Wie bereits erwähnt, wird mit AlternativeAnchors alternatives Verhalten modelliert. Dementsprechend können AlternativeAnchors je nach Kontext die Rolle von Verzweigungs- oder Vereinigungsankern annehmen. Dem muss im AlternativeAnchorFragment Rechnung getragen werden.

Strukturell entspricht der Aufbau von AlternativeAnchorFragments exakt dem von ExtensionAnchorFragments. Sie weisen jedoch bezüglich der angebundenen Kontexte einen deutlichen Unterschied auf. Aufgrund der Tatsache, dass AlternativeAnchors entweder Verzweigungs- oder Vereinigungsanker sein können, dürfen am DescitionPlace des AlternativeAnchorFragments nur AlternativeContextFragments angebunden werden, die über eine BranchConnection mit dem AlternativeAnchor verbunden sind (vgl. Abb. 8.11). Analog sind am MergePlace nur diejenigen AlternativeContextFragments gebunden, die eine MergeConnection zum Anchor haben.

Zusätzlich haben auch AlternativeAnchorFragments eine DefaultTransition. Diese modelliert sowohl den Fall, dass an einem Verzweigungsanker keine der Bedingungen für eine Verzweigung erfüllt ist, als auch den Fall, dass ein AlternativeAnchor ausschließlich als Vereinigungsanker dient. Im zweiten Fall ist offensichtlich kein AlternativeContextFragment an den DescitionPlace gebunden.

Loop: Abbildung 8.13 stellt das Verhaltensfragment eines Loops dar. Loops bilden wie Flows Verhaltensrahmen für die Events, die sie enthalten. Demzufolge bestehen LoopFragments aus einer initialen und einer finalen Transition. Zusätzlich hat die initiale Transition einen Kontrollausdruck, der die Schleifenbedingung prüft. Die finale Transition ist statt mit der Ausstiegsstelle des LoopFragments mit seiner Einstiegsstelle verknüpft, um Iterationen durchführen zu können. Zusätzlich enthält das LoopFragment eine DefaultTransition, um den Schleifenaussprung zu modellieren. Hier wird jeweils die initiale Transition der Schleife zuerst ausgewertet.

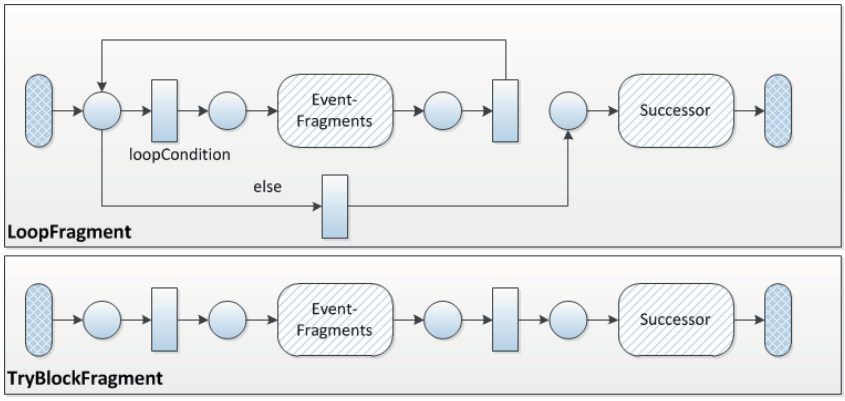


Abbildung 8.13: LoopFragment & TryBlockFragment

TryBlock: Auch TryBlocks sind einfache Verhaltensrahmen. TryBlockFragments bestehen, wie InclusionFragments, aus einer initialen und einer finalen Transition, die nicht miteinander verbunden sind. Sie bilden einen Rahmen für die im TryBlock enthaltenen Elemente. Die verschiedenen Verhaltensvarianten, die an einem TryBlock auftreten können, werden beim Ausnahmekontext beschrieben.

8.1.6.3 Contexts

Strukturell sind die Verhaltensfragmente aller Contexts gleich. Sie bestehen aus einer initialen und einer finalen Transition, die jeweils den assoziierten Flow einrahmen. Weiterhin stellt Abbildung 8.14 dar, dass die initiale - und die finale Transition jeweils einen Kontrollausdruck haben, der den Ausführungsstack prüft. Dabei ist insbesondere die finale Transition des ContextFragments interessant. Da einem Flow oftmals mehrere verschiedene Contexts assoziiert sind, muss nach Ende des Flows der Context gefunden werden, unter dem das Verhalten gestartet worden ist. Deshalb schreibt die initiale Transition den Context auf den Ausführungsstack, und die finale Transition prüft, ob dieser auf dem Ausführungsstack liegt.

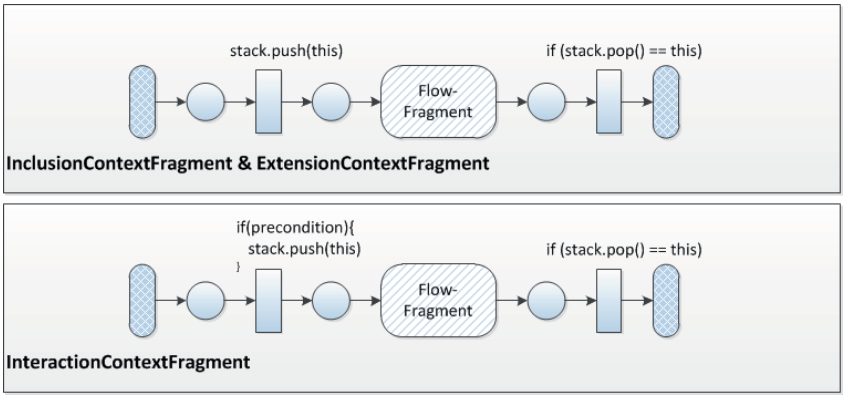


Abbildung 8.14: ContextFragments

Inclusion- und ExtensionContext: Inclusion- und ExtensionContexts entsprechen exakt der bereits beschriebenen Struktur. Da sie nur mit einem Anchor verbunden sind, kommen sie ohne Kontrollausdruck zur Identifikation des Zielankers an der finalen Transition aus.

Bei einem ExtensionContext muss allerdings an der initialen Transition ein Kontrollausdruck modelliert werden, der die Verzweigungsbedingung prüft.

AlternativeContext: Wie bereits erwähnt, ist die Schaltlogik von AlternativeContexts deutlich komplizierter als bei anderen Context-Arten. Zunächst hat die initiale Transition, wie bei ExtensionContexts, einen Kontrollausdruck zur Prüfung der Verzweigungsbedingung.

Allerdings ist ein AlternativeContext oftmals mit mehreren MergeAnchors verknüpft, die abhängig vom Zustand des Systems, also der Variablenbelegung auf der Marke, angesprungen werden können. Diese Schaltsemantik wird modelliert, indem in der finalen Transition, neben dem AlternativeContext, der aufrufende AlternativeAnchor vom Ausführungsstack entfernt wird. Anschließend wird der neue MergeAnchor auf den Ausführungsstack geschrieben. Dies wird, wie in Abbildung 8.14 dargestellt, durch Kontrollausdrücke erreicht.

Weiterhin können mit AlternativeContexts Ausnahmeszenarien (vgl. Kap. 2.4.1.4), in denen das Ziel des Use Cases nicht erreicht wird, modelliert werden. Hierzu kann der AlternativeContext mit einem *AlternativeEnd* verbunden werden.

Wie auch bei ExtensionAnchors wird eine Ordnung bei der Auswahl der verschiedenen Rücksprungoptionen verwendet, um Determinismus zu gewährleisten.

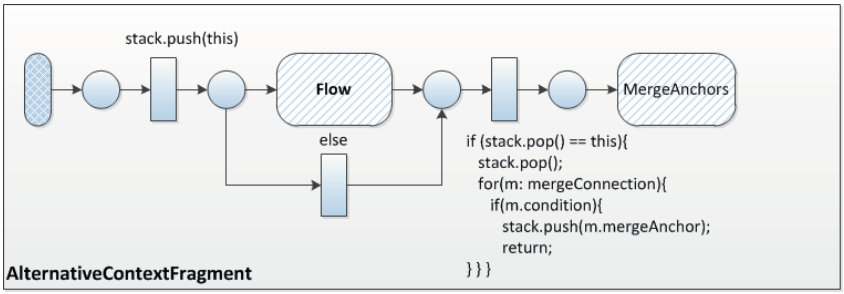


Abbildung 8.15: AlternativeContextFragment

InteractionContext: Bisher haben wir uns lediglich mit der Kombination von Szenarien aus Flüssen beschäftigt. Zusätzlich muss jedoch der mögliche Einsprung in einen Use Case durch InteractionContexts beschrieben werden. Diese haben an der initialen Transition einen Kontrollausdruck, der die Vorbedingungen des Use Cases prüft, falls diese formal spezifiziert sind. Die Nachbedingungen hingegen sind im Ausführungsmodell nicht formalisiert, da sie nicht direkt Einfluss auf die ausführbaren Szenarien haben. Ihre Einhaltung kann aber im Simulationswerkzeug untersucht werden.

ExceptionContext: Der ExceptionContext stellt einen Sonderfall dar. Er beschreibt im Unterschied zu allen anderen Contexts keine Situation, die eindeutig an einer Position des Kontrollflusses festgemacht werden kann. Stattdessen stellt er, wie bereits in Kapitel 7.4 diskutiert, einen Platzhalter für eine Menge gleichartiger Verhaltensalternativen, die an mehreren aufeinanderfolgenden Events eines Flows auftreten können, dar. Deshalb kann er nicht durch nur zwei Transitionen, nämlich einer initialen und einer finalen, dargestellt werden. Da die Schaltsemantik von gefärbten Petrinetzen das Feuern einer Transition nur dann zulässt, wenn eine Marke an jeder Stelle ihres Vorbereichs vorliegt, müssen Hilfstransitionen eingesetzt werden, um den Aussprung aus dem Normalablauf eines TryBlockes zu modellieren. Wie in Abbildung 8.16 dargestellt, wird eine derartige Hilfstransition an jede finale Stelle jedes EventFragments innerhalb des zum ExceptionContext gehörenden TryBlocks angehängt.

Hier wird analog zu allen bedingten Entscheidungen die Reihenfolge der am TryBlock registrierten Ausnahmekontexte zur Auswertung der Aussprungbedingungen verwendet.

AlternativeEnd: AlternativeEnds beschreiben, wie bereits diskutiert, Situationen, in denen der Ablauf eines Use Cases abgebrochen werden muss. Neben einem Kontrollausdruck, der die entsprechende Verzweigungsregel beschreibt, haben sie einen Kantenausdruck, der alle Elemente des Ausführungsstacks entfernt. Auch hier sind zugehörige Nachbedingungen nicht im Netz formalisiert,

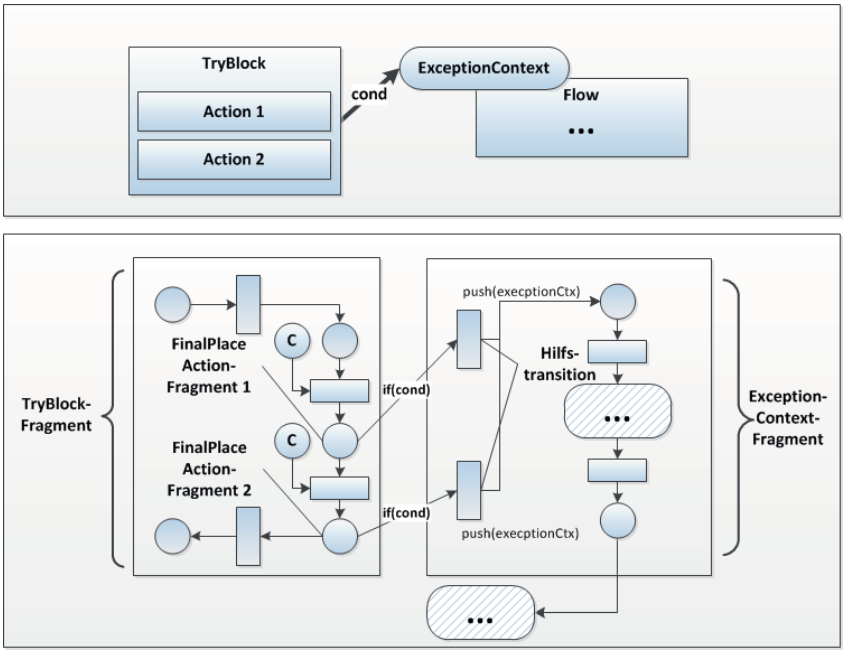


Abbildung 8.16: Verhaltensfragment – ExceptionContext

können aber im Werkzeug geprüft werden.

8.1.6.4 Zusätzliche Konzepte

Simulationsrahmen: Eine wichtige Forderung an die Simulationsumgebung ist die Möglichkeit, mehrere Use Cases nacheinander ausführen zu können, um das globale Systemverhalten evaluieren zu können. Dies wird durch den Simulationsrahmen erreicht. Dieser bildet einen globalen Verhaltensrahmen, der alle Szenarien des UCSM enthält. Seine initiale Transition wird als *Startbegrenzer* (start delimiter) und seine finale Transition als *Endbegrenzer* (end delimiter) des Ausführungsmodells bezeichnet. Wie in Abbildung 8.17 dargestellt, ist der Startbegrenzer mit der sogenannten *Ausgangsstelle* (primary place) verknüpft, die die ersten Marken erhält, wenn ein Simulationslauf gestartet wird. Der Endbegrenzer ist ebenfalls mit dieser Stelle verbunden. Somit werden aufeinanderfolgende Use Case Läufe möglich.

Fehlende Elemente: Wie bereits erwähnt, soll die Simulationsumgebung in der Lage sein, Simulationsläufe auf unvollständig spezifizierten UCSM Modellen durchzuführen. Deshalb enthält die Ausführungssemantik die sogenannten *Platzhalterstellen* (null place). Diese werden immer dann eingeführt, wenn eine

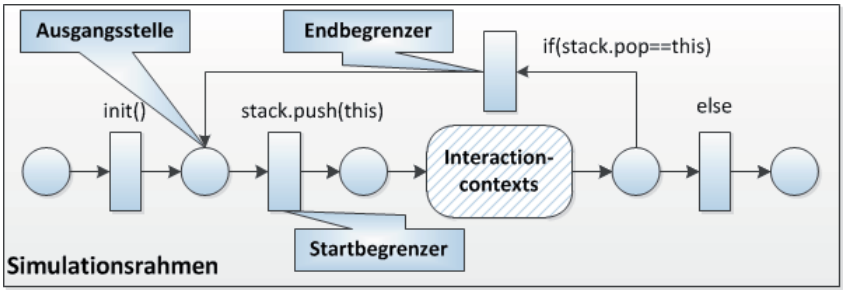


Abbildung 8.17: Simulationsrahmen

wichtige Ablaufinformation im UCSM fehlt. So wird beispielsweise ein fehlender InclusionContext an einem InclusionAnchor oder ein leerer Ereignisfluss in einem Flow durch eine Platzhalterstelle verbunden. Damit ist gewährleistet, dass das Ausführungsmodell keine Datensenken enthält, an denen Marken verloren gehen.

Hierdurch kann allerdings die Lebendigkeit des Ausführungsmodells nicht sichergestellt werden. In Situationen, in denen am Ende eines Use Case Durchlaufes keine der Vorbedingungen eines Use Cases mehr erfüllt ist, ist das Ausführungsmodell tot. Dies ist aber aus unserer Sicht ein gewünschtes Verhalten, das oft Fehler im UCSM aufdeckt.

8.1.7 Diskussion wichtiger Designentscheidungen

Die hier vorgestellte Ausführungssemantik ist nicht die einzig mögliche auf Basis von gefärbten Petrinetzen, die alle in Abschnitt 5.3 geforderten Anforderungen umsetzt. Im Folgenden werden einige interessante Aspekte möglicher Lösungsalternativen sowie deren Vor- und Nachteile diskutiert.

Formale Nachbedingungen: Wie bereits diskutiert, werden im vorgestellten Ausführungsmodell die Nachbedingungen nicht explizit modelliert. Hier gibt es zwei mögliche Alternativen:

- **Nachbedingungen als Kontrollausdrücke:** Die Nachbedingungen eines Use Cases hätten als Kontrollausdrücke an der finalen Transition des InteractionContextFragments des Use Cases bzw. eines AlternativeEnd-Fragments modelliert werden können. Dies hat aber aus unserer Sicht den Nachteil, dass das zugehörige Netz in allen Situationen, in denen die Nachbedingung nicht erfüllt ist, tot wäre. Insbesondere bei der inkrementellen Weiterentwicklung kommt uns dies zu streng vor. Man könnte allerdings explizit einen „strengen Simulationsmodus“ einführen, bei dem Kontrollausdrücke für Nachbedingungen verwendet werden.

- **Statusmarken:** Anstatt die Simulation bei Nichteinhaltung einer Nachbedingung abzubrechen, wäre es möglich, spezielle Statusmarken einzuführen. Dazu hätte die finale Transition eines InteractionsContexts bzw. eines AlternativeEnds eine spezielle *Statusstelle* in ihrem Nachbereich, in die, wie in Abbildung 8.18 dargestellt, eine *Statusmarke* geschrieben wird. Diese enthält dann ein Attribut, das die Einhaltung der Nachbedingung, z.B. durch einen booleschen Wert, codiert.

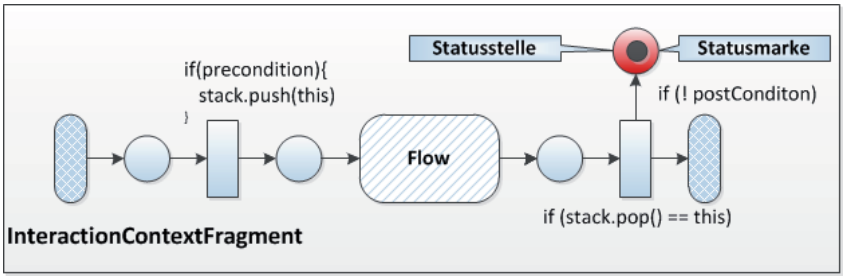


Abbildung 8.18: Statusmarken

Analog könnte man auch über die Art und Weise diskutieren, wie Vorbedingungen modelliert werden. Wir haben uns hier dafür entschieden, diese Vorbedingungen formal in Kontrollausdrücken einzuschreiben, da sie im Unterschied zu Nachbedingungen i.d.R. auch in der Implementierung der Lösung kontrolliert werden.

Simulation mit mehreren Nutzern: Im hier vorgestellten Ausführungsmodell sind zwar prinzipiell Simulationsläufe mit mehreren parallel arbeitenden Nutzern möglich, indem die Ausgangsstelle mit mehreren Marken initialisiert wird, das Ausführungsmodell selbst ist hierfür aber nicht explizit ausgelegt, da der gesamte Ausführungszustand lokal auf den Marken im Netz gespeichert ist. Somit gibt es keine Möglichkeit, z.B. eine Änderung des globalen Systemzustands von einem Nutzer an einen anderen zu kommunizieren. Wir haben uns gegen einen solchen Mechanismus entschieden, weil er einen erheblichen Mehraufwand auf der Modellierungsseite erfordert. Dieser lohnt aus unserer Sicht nicht, da Mehrnutzerbetrieb für unseren Hauptanwendungsfall, nämlich die Analyse der Richtigkeit der Ablaufstruktur eines UCSM Modells, nur eine untergeordnete Rolle spielt. Zusätzlich muss für die Mehrnutzersimulation ein relativ hoher Formalisierungsgrad im Ausgangsmodell vorliegen, der in vielen von uns betrachteten Projekten nie erreicht wurde.

Dennoch sollen hier kurz zwei Möglichkeiten skizziert werden, wie das vorgestellte Ausführungsmodell um Konzepte für geteilte Variablen im Mehrnutzerbetrieb erweitert werden kann:

- **Call by Reference Variablen:** Eine Möglichkeit, mit der sich der Aus-

tausch von zustandsrelevanten Variablen zwischen verschiedenen Nutzern eines Simulationslaufes umsetzen lässt, ist die Unterscheidung zwischen Call by Reference und Call by Value Variablen. Hier könnte man analog zu Programmiersprachen Variablen als Call by Reference markieren. Diese würden dann im Unterschied zu Call by Value Variablen nicht auf jede Marke kopiert. Stattdessen enthalten die Marken eine Referenz auf eine Instanz der Variable, die jeweils gelesen bzw. geschrieben wird.

- **Yield Transitions:** Die Call by Reference Modellierung nimmt an, dass Variablen jederzeit zwischen zwei Nutzern getauscht werden können. Dies ist i.d.R. nicht der Fall. Eine weitere Möglichkeit Daten, zwischen Nutzern auszutauschen ist, analog zum wechselseitigen Ausschluss (vgl. [RG97, 481]), explizite *Yield Transitions* zu modellieren, in denen die Variablen zwei parallel arbeitender Nutzer sychronisiert werden können. Dies ermöglicht es zudem, Probleme wie „dirty read“ und „lost update“ zu identifizieren.

8.2 IPM - Ein petrinetzbasiertes Prototypingmetamodell

Das Integrated Prototype Model (IPM) ist ein Metamodell zur Modellierung der Ausführungskonzepte funktionaler Nutzerschnittstellenprototypen. Es dient im hier vorgestellten Ansatz als Grundlage für die Generierung verschiedener Arten von Nutzerschnittstellenprototypen. Die Konzepte des IPM sind wie auch das UCSM vom BasicConceptModel abgeleitet. Dies vereinfacht die Verbindung mit den Modellelementen des IPM mit denen des UCSM.

Das IPM modelliert die topologischen Elemente gefärbter Petrinetze. Es enthält Stellen, Transitionen, Kanten, den Deklarationsblock und Inskriptionen.

Das IPM wird für die Darstellung der Ausführungslogik der Use Case Prototypen eingesetzt. Deshalb enthält es neben den normalen Schaltelementen gefärbter Petrinetze, Elemente, die die Verbindung zwischen den Petrinetzelementen und den zugehörigen Ausgangsdokumenten, wie Use Cases und Nutzerschnittstellenbeschreibung, herstellen. Im Folgenden werden die Elemente des Metamodells lediglich kurz vorgestellt. Eine detaillierte Beschreibung der Semantik der einzelnen Elemente ist bereits in Kapitel 8.1 enthalten.

PetriNet: Das Element *PetriNet* bildet die Wurzel eines Petrinetzes. Es enthält alle *TopologicElements*, sowie die Elemente des Deklarationsblocks und referenziert immer das UCSM, dessen Ausführungsmodell es bildet.

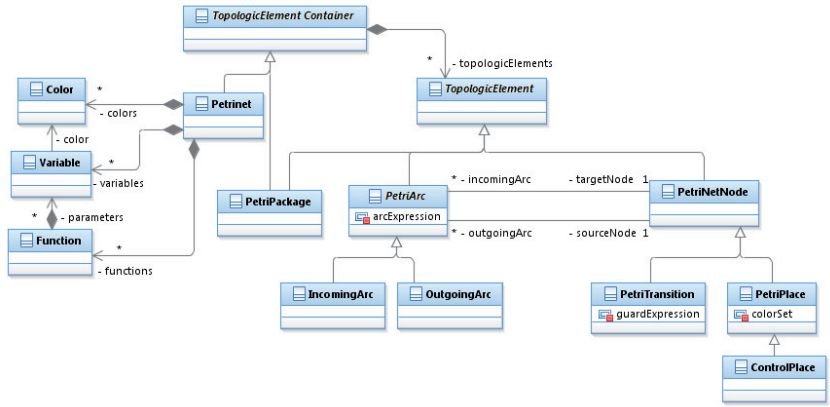


Abbildung 8.19: IPM – Metamodell

PetriPackage: *PetriPackage* ist ein Konzept, das in der Definition von gefärbten Petrinetzen nicht vorkommt. Es dient als Container für eine Menge von *TopologicElements*. PetriPackages haben selbst keine Schaltsemantik und werden deshalb bei der Ausführung ignoriert. Sie helfen aber, die Verständlichkeit des Netzes zu erhöhen, da sie es ermöglichen, das Netz entlang der Struktur der Ausgangsmodelle zu partitionieren.

PetriPlace: Das Element *PetriPlace* repräsentiert Stellen im Petrinetz. Da es sich um ein gefärbtes Netz handelt, kann jede PetriPlace einen Reihe von Colors referenzieren, um das Colorset des PetriPlaces zu beschreiben. *ControlPlaces* modellieren die in Abschnitt 8.1.5 beschriebenen Kontrollstellen.

PetriTransition: *PetriTransitions* stellen Transitionen im Petrinetz dar. Ihnen kann ein Kontrollausdruck *guardExpression* assoziiert werden, der das Schaltverhalten der Transition modelliert.

PetriArc: PetriPlaces werden mit PetriTransitions durch *PetriArcs* verbunden. Hierbei können zwei Arten von PetriArcs unterschieden werden. *InputArcs* stellen eine Verbindung von einem PetriPlace zu einer PetriTransition dar. *OutputArcs* hingegen beschreiben eine Verbindung von einer Transition zu einem PetriPlace. Beiden Arten von PetriArcs können jeweils einen Kantenausdruck *arcExpression* enthalten.

Declarationsblock: Neben den TopologicElements enthält das PetriNetModel die Elemente des Deklarationsblocks. Diese werden exakt, wie in Abschnitt 8.1.2 beschrieben, modelliert.

- **Color:** Das Element *Color* definiert einen Typ.
- **Variable:** Jede *Variable*, definiert eine Variable des PetriNets. Das Attribut color definiert deren Typ.
- **Function:** Eine *Function* modelliert eine Funktion des Petrinetzes. Analog zum SFM enthalten auch die Functions des IPM Parameter. Dazu referenziert das Attribut *parameters* eine Menge von Variablen.

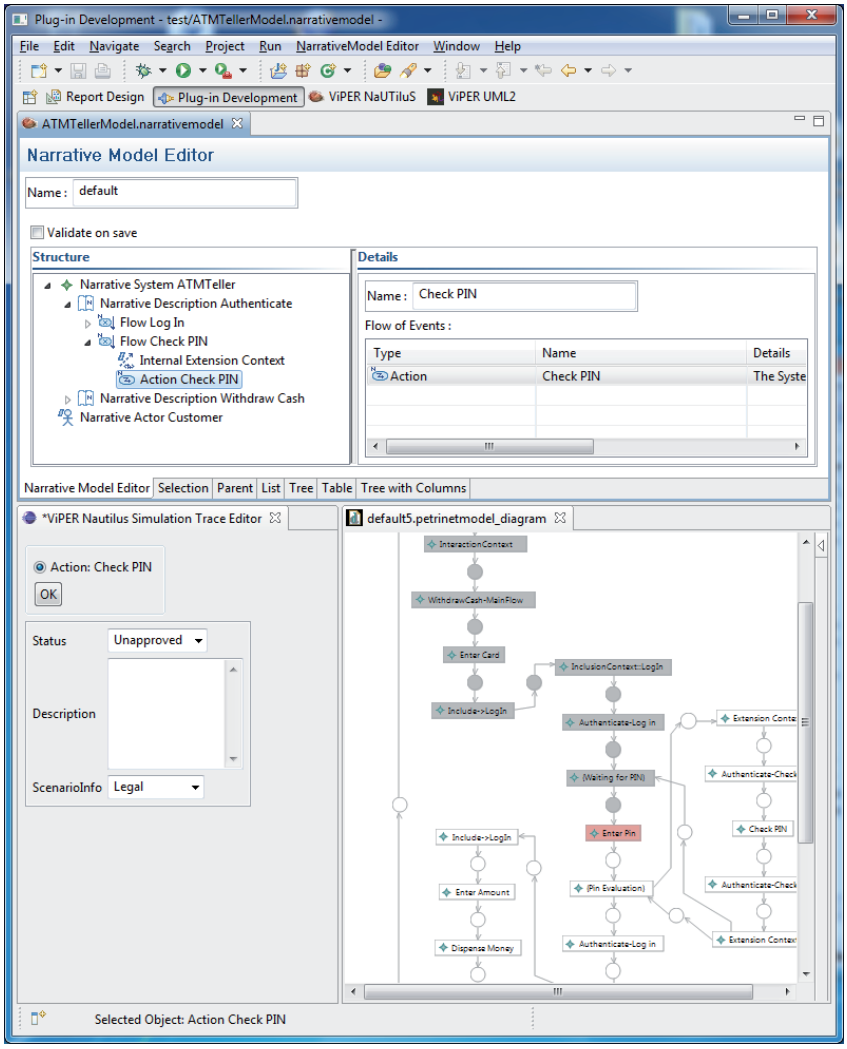


Abbildung 8.20: OpenUMF – IPM Editor

8.2.1 IPM Editor

In der Regel werden Instanzen des IPM vollautomatisch aus UCSM Modellen erzeugt. Deshalb ist eine manuelle Anpassung der IPM Modelle nicht notwendig. Dennoch kann es nützlich sein, IPM Modelle für spezielle Detailanalysen oder zur Weiterentwicklung des Prototypgenerators direkt zu analysieren und zu verändern. Deshalb enthält OpenUMF den in Abbildung 8.20 dargestellten IPM Editor. Dieser erlaubt es, IPM Modelle zu analysieren und zu verändern.

8.3 Generierung von Prototypen

Nachdem im vorangehenden Abschnitt die Sprachen UCSM und IPM eingeführt worden sind, kann nun der zugehörige Prototypgenerierungsansatz beschrieben werden. Die Grundidee des nachfolgenden Ansatzes ist es, ein UCSM Modell vollautomatisch in ein IPM Modell zu überführen, das die Struktur der in Kapitel 8.1 beschriebenen Ausführungssemantik aufweist. Dazu wird ein sehr einfacher Generierungsalgorithmus verwendet. Dieser erzeugt ein IPM Modell, indem er die beschriebenen Transformationsregeln der Reihe nach auf die ausführungsrelevanten Elemente des UCSM anwendet.

Der Algorithmus baut das IPM Modell auf, indem er zunächst die FlowFragments des IPM einzeln aufbaut und dann zusammenschaltet. Er besteht aus den vier in Abbildung 8.21 dargestellten Schritten. Im folgenden werden diese vier Schritte einzeln der Reihe nach beschrieben. Zusätzlich verdeutlichen die Abbildungen 8.22 und 8.23 die Schritte des Algorithmus noch einmal graphisch anhand eines Beispiels.

```
generateIPM(UCSM ucsm){
 fragmentMap = new HashMap<REElement,Fragment>();
 //Schritt 1
 flowCreation(ucsm, fragmentMap);
 //Schritt 2
 contextPreparation(ucsm, fragmentMap);
 //Schritt 3
 contextConnection(ucsm, fragmentMap);
 //Schritt 4
 frameSetup(ucsm, fragmentMap);
}
```

Abbildung 8.21: Generierungsalgorithmus – Aufbau

Schritt 1 – FlowCreation: Im ersten Schritt werden zunächst alle FlowFragments einzeln erzeugt. Dazu werden der Reihe nach alle Events eines Flows in ihre IPM Repräsentation transformiert und in den vom Flow aufgespannten Verhaltensrahmen eingefügt. Dabei werden gleichzeitig für alle AnchorFragments Platzhalterstellen generiert. Diese dienen dazu, auch dann ein ablauffähiges Modell zu garantieren, wenn ein Anchor nicht mit einem Context verbunden ist.

```
createFlows(UCSM ucsm, HashMap fragmentMap){
 for(Flow flow : ucsm.flows){
  flowFragment = createFlowFragment(flow);
  fragmentMap.put(flow,flowFrame);
  predecessor=flowFragement;
  for(event : f:events){
   eventFragment = createEventFragment(event);
   fragmentMap.put(event,eventFragment);
   setSuccessor(predecessor,eventFragment);
   predecessor = eventFragment;
  }
  setSuccessor(predecessor,flowFragement);
}}
```

Schritt 2 – ContextPreparation: Im ContextPreparationStep werden alle Contexts eines Flows eingefügt. Dazu wird jeder Context in ein ContextFragment transformiert, das das FlowFragment enthält.

```
prepareContexts(UCSM ucsm, HashMap fragmentMap){
 for(Context context : ucsm.contexts){
  contextFragment = createContextFragment(context);
  fragmentMap.put(context,contextFrame);
  flowFragment = fragmentMap.get(context.getFlow())
  insert(contextFragment,flowFragment);
}}
```

Schritt 3 – ContextConnection: Im ContextConnectionStep werden nun alle AnchorFragments mit den zugehörigen ContextFragments verbunden, indem die Platzhalterstelle durch das ContextFragment ersetzt wird.

```
connectContexts(UCSM ucsm, HashMap fragMap){
 for(Anchor anchor: ucsm.anchors){
  anchorFragment = fragMap.get(anchor);
  for(Connection connection : anchor.connections){
   ctxFrag = fragMap.get(connection.getTarget());
   link(anchorFragment,ctxFrag);
}}}
```

Schritt 4 – FrameSetup: Als letzten Schritt in der Transformation wird der Simulationsrahmen generiert. Alle InteraktionContextFragments und AlternativeEndFragments werden mit diesem verbunden. Damit ist die Transformation abgeschlossen.

```
setupFrame(UCSM ucsm, HashMap fragmentMap){
 frameFragment = createFrameFragment(ucsm);
 for(InteractionContext context : ucsm.intCtxs){
  contextFragment = fragmentMap.get(context);
  insert(frameFragement,contextFragment);
 }
 for(End end: ucsm.ends){
  endFragment = fragmentMap.get(end);
  link(frameFragement,endFragment);
}}
```

Dieser Generierungsalgorithmus terminiert für alle Arten von Modellen in linearer Zeit relativ zur Anzahl der ausführungsrelevanten Elemente im UCSM. Er läuft jeden Fluss des UCDM im ersten Schritt genau einmal ab. In den weiteren Schritten werden lediglich die Anchors bzw. Contexts betrachtet. Außerdem erzeugt der Generierungsalgorithmus in allen Fällen ein ablauffähiges Modell, da er in Situationen, in denen Ablaufinformationen fehlen, Platzhalterstellen erzeugt.

Zusätzlich kann er auch eingesetzt werden, um Simulationsmodelle aus Teilen der Use-Case-zentrierten Spezifikation abzuleiten. Dazu muss lediglich ausgewählt werden, welche Use Cases bei der Generierung berücksichtigt werden sollen. Alle anderen Use Cases werden dann bei der Generierung ignoriert und falls nötig durch Platzhalterstellen subsummiert.

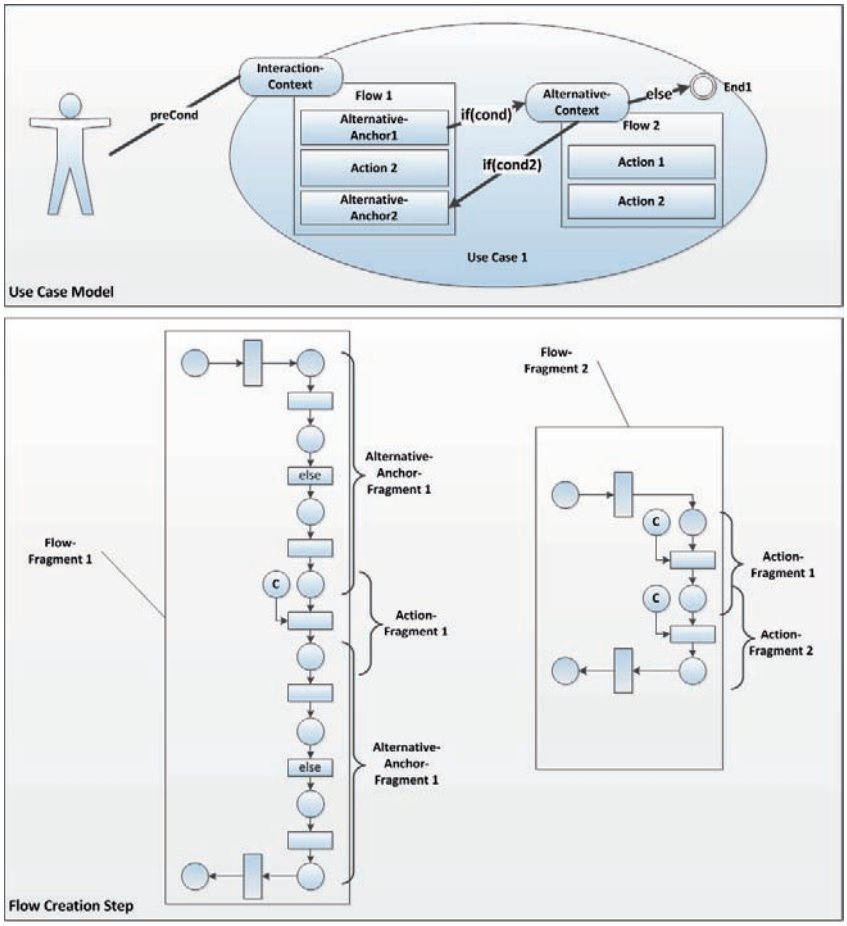


Abbildung 8.22: Generierung – Beispiel Teil 1

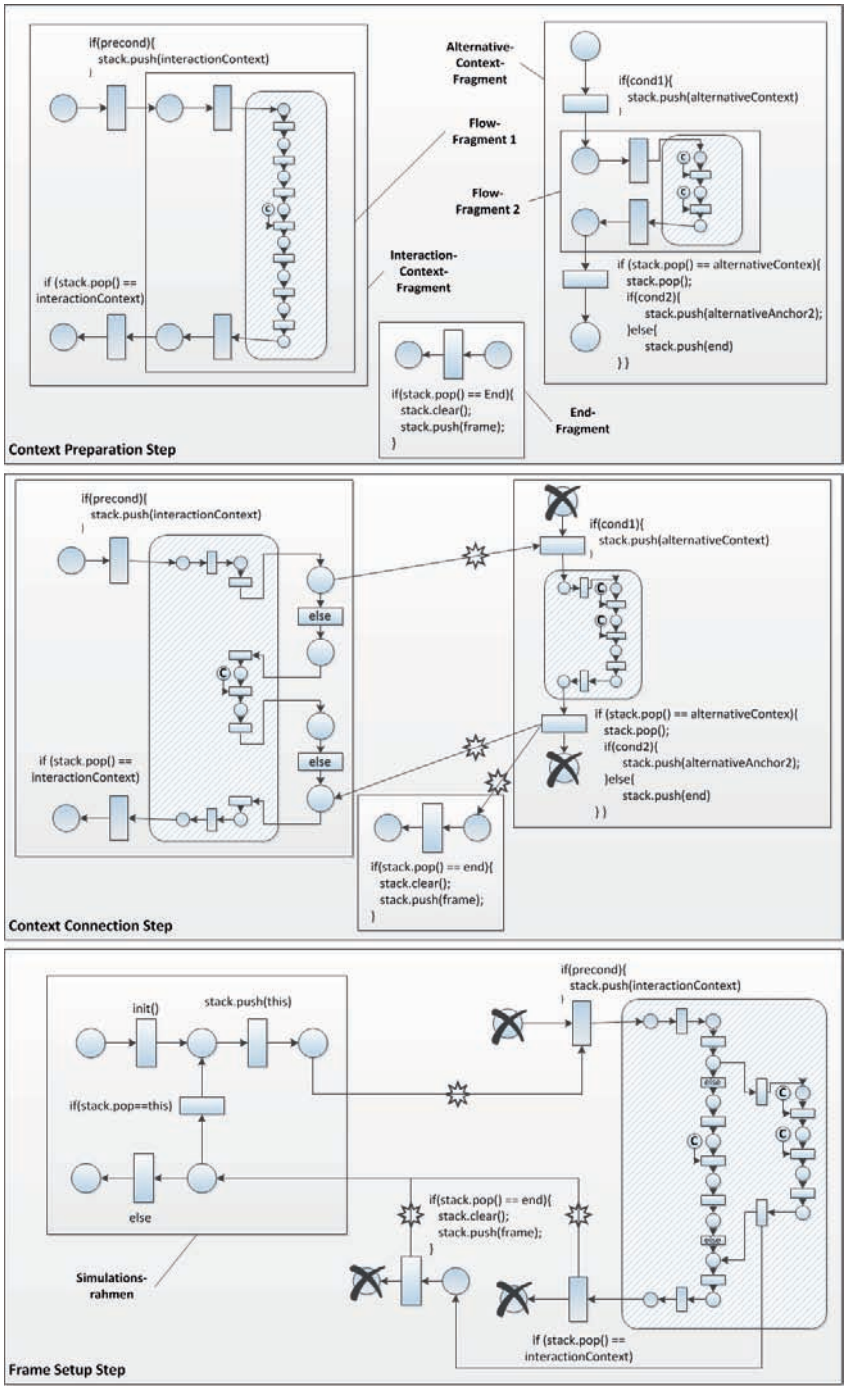


Abbildung 8.23: Generierung – Beispiel Teil 2

8.3.1 PGF – Ein Prototypgenerator für UCSM

Die Generierung eines Prototypen kann aus verschiedenen Werkzeugen des UCMT angestoßen werden. Sie ist für den Nutzer vollkommen transparent. Es ist also nicht nötig, dass der Nutzer die Ausführungssemantik des UCSM oder das IPM kennt. Abbildung 8.24 zeigt, wie die Prototypgenerierung aus dem GDM Overview Editor angestoßen wird.

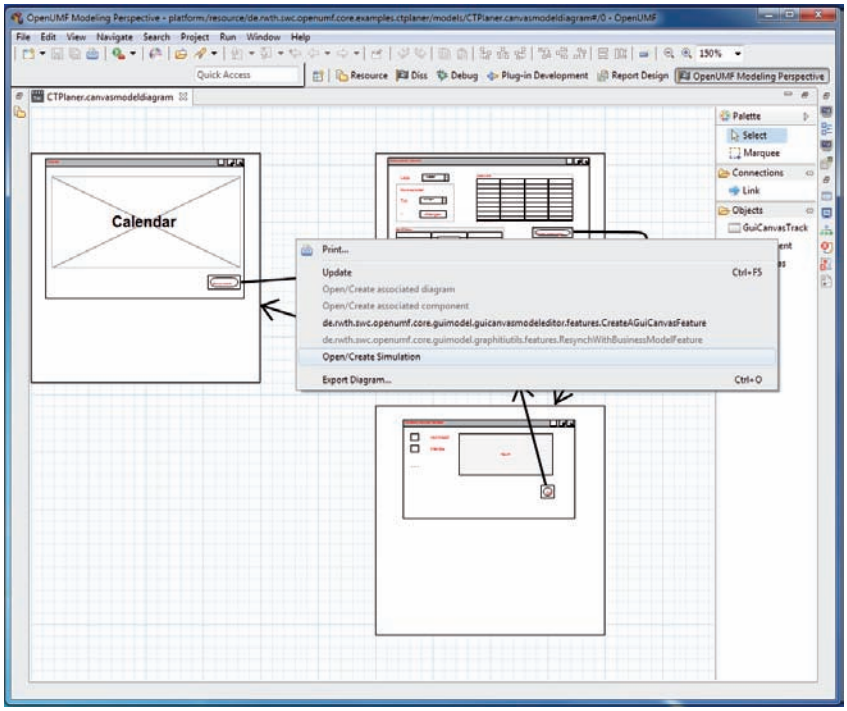


Abbildung 8.24: OpenUMF – Prototypgenerierung

9 Bewertung von Prototypen

Nachdem in den vorherigen Kapiteln die Spezifikation von UCSM Modellen und die Generierung von Ablaufprototypen aus diesen Modellen vorgestellt worden sind, soll nun die Bewertung der Prototypen diskutiert werden. Hierzu werden zunächst einige Vorüberlegungen gemacht. Anschließend wird PEM das Bewertungsmetamodell von SUPrA vorgestellt und die Bewertungsumgebung präsentiert.

9.1 Vorüberlegungen

Primäres Ziel der Prototypbewertung in unserem Anwendungsumfeld ist es, die modellierten Abläufe zu analysieren bzw. weiterzuentwickeln. Dies ist offensichtlich aufgrund der Komplexität der modellierten Informationen und der zahlreichen Zusammenhänge zwischen den einzelnen Submodellen von UCSM nur mit Hilfe einer geeigneten Werkzeugunterstützung möglich. Zur Bewertung der Abläufe ist es nötig, aus UCSM Modellen wie im letzten Kapitel beschrieben IPM Prototypen zu generieren und diese in einer geeigneten Bewertungsumgebung zu simulieren. Simulieren bezeichnet dabei das Ausführen von einem oder mehreren Szenarien in der Bewertungsumgebung. Um die in Kapitel 5 geforderten Anforderungen an eine geeignete Werkzeugumgebung erfüllen zu können, müssen drei Aspekte besonders betrachtet werden:

- **Steuerung der Simulation aus der Nutzerschnittstelle:** Die Simulation soll in den verschiedenen Phasen der ProDUCE Analyse (vgl. Kap. 10) von unterschiedlichen Nutzergruppen verwendet werden können. Wie bereits erwähnt, haben insbesondere Kunden i.d.R. keine Erfahrung mit der Analyse von formalen Modellen. Sie können eine Bewertung des Verhaltens eines modellierten Systems am einfachsten anhand der modellierten Nutzerschnittstelle durchführen. Zudem sind Benutzbarkeitsanalysen (E7) nur möglich, wenn die geplante Nutzerschnittstelle zur Simulation verwendet wird.

  Die Bewertung des modellierten Systems durch Experten des Kunden ist ein zentrales Anwendungsszenario für das Prototyping in SUPrA. Deshalb muss die Bewertungsumgebung in der Lage sein, die Simulationsläufe aus der im GDM modellierten Nutzerschnittstelle zu steuern. Da das IPM keine Informationen zu dieser Nutzerschnittstelle enthält, muss sie während der Simulation dynamisch aus dem zum simulierten IPM Modell gehören-

den UCSM Modell generiert werden.

- **Aufzeichnen von Traces:** Zusätzlich zur Analyse des Verhaltens können wichtige Informationen über das modellierte System und den Analyseprozess gewonnen werden, indem Simulationsläufe aufgezeichnet und später statisch analysiert werden (W7,W8). Deshalb muss die Bewertungsumgebung Traces aufzeichnen können. Zusätzlich müssen diese Traces mit speziellen Informationen angereichert werden, die die Bewertung des Prototyps hinsichtlich besonderer Aspekte unterstützen. So könnten beispielsweise Befunde während eines Reviews eines UCSM Modells dokumentiert werden.

- **Flexible Anpassung der Bewertungsumgebung:** Die Bewertungsumgebung muss in der Lage sein, Simulationen von UCSM Modellen in unterschiedlichen Detaillierungsgraden zur Beantwortung unterschiedlicher Analysefragen und mit unterschiedlichen Nutzergruppen zu unterstützen. Offensichtlich muss die Bewertungsumgebung dazu jeweils für den speziellen Einsatz angepasst werden. Sie muss also so aufgebaut sein, dass sie schnell für verschiedene Einsatzszenarien konfiguriert werden kann. Zusätzlich muss sie leicht um spezielle Werkzeuge für neue Einsatzszenarien erweitert werden können.

9.2 PEM - Prototype Evaluation Model

Das Prototype Evaluation Model (PEM) ist das Bewertungsmetamodell des SUPrA Ansatzes. Das IPM beschreibt lediglich die Netztopologie, d.h., es modelliert die Schaltsemantik eines gefärbten Petrinetzes, nicht aber das Netzverhalten, also die Marken des Netzes und ihre Position zu einer bestimmten Zeit. Diese Informationen werden im PEM abgelegt. Es dient sowohl zur Beschreibung des Netzverhaltens als auch zur Aufzeichnung von Traces für die spätere Analyse.

Simulation: Das Element *Simulation* bildet die Wurzel des PEM. Es enthält eine Menge von Traces und eine Referenz auf das IPM Modell, auf dem die Simulationen ausgeführt werden, sowie auf das zugehörige UCSM Modell.

Trace: Jeder *Trace* beschreibt einen einzelnen Simulationslauf durch das IPM Modell bzw. die Simulation eines bestimmten Szenarios. Er dient zu deren Aufzeichnung für die spätere Analyse. Jeder Trace besteht aus einer geordneten Liste von SimulationSteps.

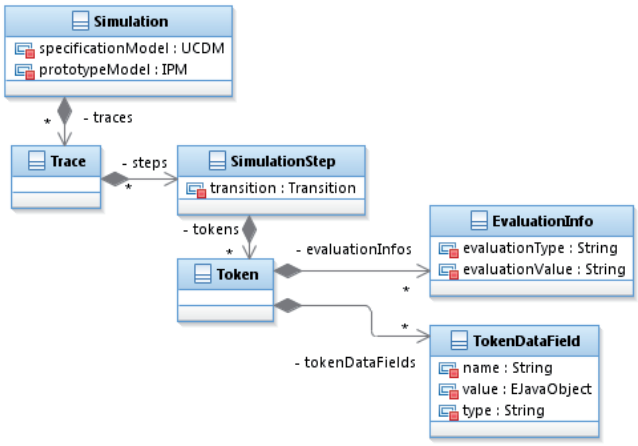


Abbildung 9.1: PEM Metamodell

SimulationStep: Jeder *SimulationStep* modelliert einen einzelnen Simulationsschritt. Ein SimulationStep enthält eine Referenz auf die Transition, die gefeuert hat, und die Bindung der Variablen der Transition sowie die Markierung des Netzes nach dem Feuern der Transition.

Token & TokenDataField: Ein *Token* modelliert eine Marke eines gefärbten Petrinetzes. Jeder Token enthält eine Reihe von Werten jeweils einer Farbmenge. Genauer gesagt, enthalten Marken in unserer objekt-orientierten Interpretation Variablen eines bestimmten Typs, die mit einem Wert belegt sind. Dieses Konzept wird durch *TokenDataFields* modelliert. Jedes TokenDataField enthält einen Namen (*name*), einen Typ (*type*) und einen Wert (*value*).

EvaluationInfo: Jede Ausführung eines Prototyps dient zur Bewertung des Prototyps unter einem bestimmten Aspekt. Deshalb muss es während eines Simulationslaufes möglich sein, Informationen zur Bewertung dieses Aspekts in SimulationSteps zu speichern. Diese Bewertungshilfen werden mit dem Element *EvaluationInfo* modelliert. Jedem SimulationStep kann eine Menge dieser EvaluationInfos zugeordnet werden. Ein EvaluationInfo enthält eine Beschreibung der relevanten Information (*evaluationValue*). Zusätzlich haben sie einen Typ (*evalationType*), der modelliert, welchem Aspekt das EvaluationInfo zugeordnet ist.

9.2.1 PEM Editor

Ähnlich wie IPM Modelle werden auch PEM Modelle nicht vom Nutzer explizit spezifiziert. Stattdessen werden sie während der Prototypbewertung von der Bewertungsumgebung automatisch erzeugt. Dennoch enthält OpenUMF den in Abbildung 9.2 dargestellten formular-basierten PEM Editor. Dieser wird allerdings lediglich zur Analyse von PEM Modellen durch Experten bei der Weiterentwicklung der Bewertungsumgebung benötigt.

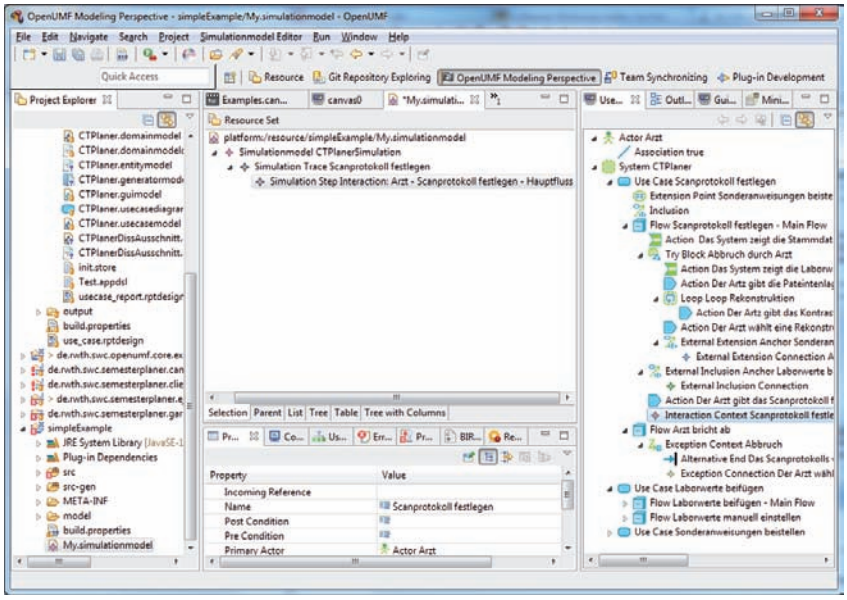


Abbildung 9.2: PEM Editor – Screenshot

9.3 Bewertungsumgebung

Nachdem das Spezifikationsmetamodell, das Prototypingmetamodell, das Bewertungsmetamodell sowie der Generierungsansatz zur Erzeugung von Prototypen eingeführt ist, soll als nächstes beschrieben werden, wie Prototypen bewertet werden können. Dies ist nur mit einer geeigneten Bewertungsumgebung möglich. Im Folgenden wird zunächst der generelle Ablauf des Prototypeinsatzes vorgestellt. Anschließend wird der Aufbau der Bewertungsumgebung beschrieben und das Zusammenspiel der einzelnen Komponenten beim Prototypeinsatz erläutert.

9.3.1 Ablauf der Prototypbewertung

Zur Bewertung der Prototypen wird das im UCSM beschriebene Verhalten auf Basis des aus dem UCSM generierten IPM Modell simuliert. Mit einer Simulation ist dabei der Durchlauf von einem oder mehreren Szenarien, also Pfaden durch das Ausführungsmodell, vom Startbegrenzer zum Endbegrenzer gemeint (vgl. Kap. 8.1.3). Dabei wird das Ausführungsmodell entsprechend seiner Schaltlogik schrittweise ausgeführt. Für den Prototypnutzer ist das Ausführungsmodell allerdings völlig transparent. Alle Informationen über den Ablauf der Simulation werden in Nutzerschnittstellenprototypen auf Basis des GDM oder auf dem UDM dargestellt. Über diese Nutzerschnittstellenprototypen können auch die Nutzereingaben eingelesen werden.

Jede Simulation beginnt damit, dass das Ausführungsmodell mit Marken initialisiert wird. Bei einer normalen Simulation wird ausschließlich dem Startbegrenzer des Ausführungsmodells eine Marke hinzugefügt. Es ist aber auch möglich, eine gespeicherte Simulation fortzusetzen. Dazu wird das Ausführungsmodell mit einer in einem SimulationStep gespeicherten Markierung initialisiert.

Nach Initialisierung des Ausführungsmodells, also Erzeugen von Marken an bestimmten Stellen im Ausführungsmodell, läuft die Simulation schrittweise ab. Dabei wandern die Marken abhängig von der Netztopologie und den Attributwerten der Marken durch das Ausführungsmodell bis zum Endbegrenzer. Jeder Simulationsschritt besteht dabei wiederum aus den drei in Abbildung 9.3 dargestellten Teilschritten:

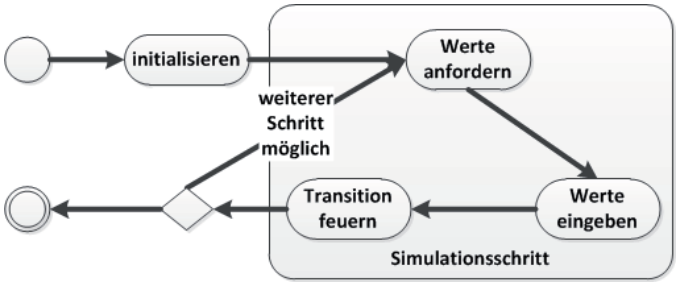


Abbildung 9.3: Struktureller Ablauf der Simulation

1. **Werte anfordern:** Die Bewertungsumgebung fordert vom Prototypnutzer eine Reihe von Nutzereingaben, um die Kontrollmarken im Ausführungsmodell zu erzeugen. Welche Nutzereingaben benötigt werden, ist dabei abhängig von der Kontrollstelle der zum Simulationsschritt gehörenden Transition.

2. **Werte einlesen:** Der Nutzer setzt die geforderten Werte. Dazu wird in der Regel die graphische Nutzerschnittstelle des emulierten Systems verwendet.

3. **Transition feuern:** Die Bewertungsumgebung feuert die aktive Transition. Die Marken im Ausführungsmodell werden einen Schritt weiter geschaltet.

Sollte am Ende eines Simulationsschrittes ein weiterer Simulationsschritt ausführbar sein, wird die Ausführung mit dem nächsten Schritt fortgesetzt. Sonst wird der Simulationslauf beendet.

Sollten mehrere Transitionen im Ausführungsmodell aktiv sein, so verlangt die Simulationsumgebung eine Entscheidung des Prototypnutzers über den Fortgang des Simulationslaufes. Dies passiert in Situationen, in denen eine Verzweigung im UDM vorliegt, die nicht formal beschrieben ist.

Während der Simulation wird ein Trace erzeugt, in dem jeder Simulationsschritt mit einem SimulationStep aufgezeichnet wird (vgl. Kap. 9.2).

9.3.2 Aufbau der Nutzerschnittstelle

Die Nutzerschnittstelle bildet ein zentrales Kriterium für die Brauchbarkeit der Bewertungsumgebung. Sie muss, wie bereits erwähnt, für verschiedene Nutzergruppen zur Bearbeitung unterschiedlicher Aufgabestellungen angepasst werden können. Dazu ist ein flexibler Aufbau der Nutzerschnittstelle nötig, der es ermöglicht, verschiedene Spezialwerkzeuge und Ansichten je nach Zielsetzung der Bewertung anzuzeigen und flexibel um eine zentrale Komponente zur Ablaufsteuerung zu gruppieren.

Dementsprechend besteht die Nutzerschnittstelle der Bewertungsumgebung, wie in Abbildung 9.4 dargestellt, logisch aus zwei Arten von Elementen.

- **Ablaufsteuerungskomponente:** Die Ablaufsteuerungskomponente bildet das zentrale Element der Nutzerschnittstelle der Bewertungsumgebung und ist bei jedem Simulationslauf sichtbar. Sie dient zur Steuerung der Simulation. Die Ablaufsteuerungskomponente zeigt zu jedem Schritt die zugehörige Seite der Nutzerschnittstelle an. Außerdem liest sie die zur Ausführung von Simulationsläufen nötigen Nutzereingaben ein.

- **Bewertungswerkzeuge:** Um die Ablaufsteuerungskomponente können verschiedene Bewertungswerkzeuge angeordnet werden. Diese Bewertungswerkzeuge dienen dazu, Simulationsläufe mit einer bestimmten Zielsetzung durchzuführen. Sie zeigen entweder spezielle zielsetzungsabhängige Informationen zu einem Simulationslauf an oder sie ermöglichen es, dem Simulationslauf Informationen, die zur zielorientierten Bewertung eines Szenarios benötigt werden, beizustellen. Hier gibt es eine Reihe generischer Bewertungswerkzeuge, z.B. enthält die Bewertungsumgebung Ansichten zur Anzeige von Traces oder Tokens. Außerdem können Werkzeuge für spezifische Zielsetzungen registriert werden. Beispielsweise gibt es ein Be-

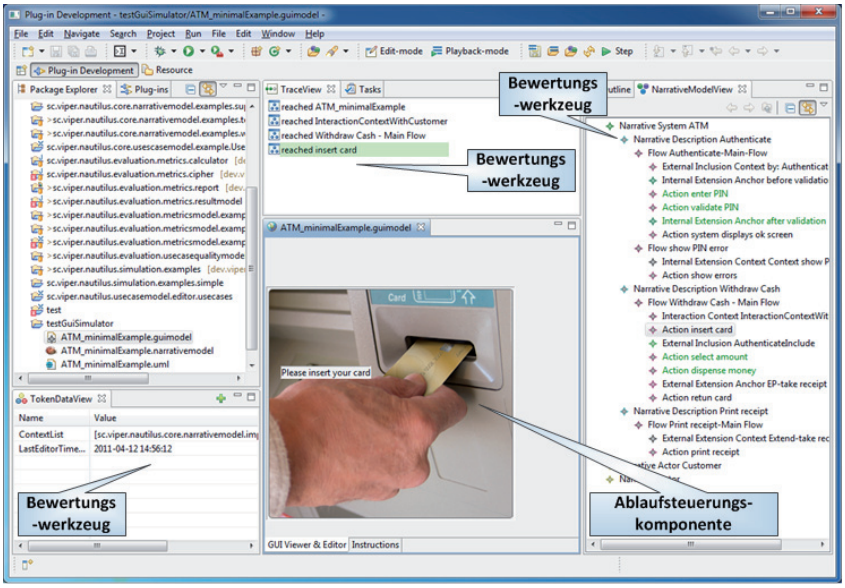


Abbildung 9.4: PEF – Aufbau

wertungswerkzeug zur Angabe von Befunden während eines simulationsgetriebenen Reviews.

9.3.3 Aufbau der Bewertungsumgebung

Eine Simulation des im UCSM Modell spezifizierten Verhaltens ist nur mit Hilfe einer geeigneten Bewertungsumgebung möglich. Diese muss in der Lage sein, die im Ausführungsmodell codierten Abläufe exakt auszuführen. Gleichzeitig muss sie den Simulationsablauf auf Basis der im UCSM spezifizierten Nutzerschnittstellen darstellen können und die für die Simulation benötigten Nutzereingaben einlesen.

Im Folgenden werden die wichtigsten Komponenten der Bewertungsumgebung vorgestellt und ihre Aufgaben während der Simulation erörtert. Hierbei werden lediglich die konzeptionellen Aspekte der Werkzeuginfrastruktur diskutiert. Die technische Umsetzung der Konzepte wird in Kapitel 11 vorgestellt.

Die Bewertungsumgebung besteht aus vier zentralen Komponenten:

- **PrototypeRunner:** Er führt die Ausführungsmodelle aus. Er realisiert also die Schaltsemantik des UCSM-Prototyps, indem er Marken durch das gefärbte Petrinetz schleust.

- **SimulationVisualization:** Sie dient dem Prototypnutzer als Steuerungskomponente. Dazu zeigt sie dem Prototypnutzer in jedem Simulationsschritt die relevanten Informationen an. Weiterhin bietet sie dem Prototypnutzer Möglichkeiten, die in einem Simulationsschritt geforderten Nutzereingaben zu machen.

- **TraceGenerator:** Der TraceGenerator bildet die Dokumentationskomponente der Bewertungsumgebung. Er zeichnet Simulationsläufe auf. Dazu legt er nach der Initialisierung der Simulation einen Trace an und speichert zu jedem Simulationsschritt einen SimulationStep, der die Markierung des Ausführungsmodells enthält.

- **SimulationController:** Er bildet die zentrale Steuerungskomponente der Simulation. Er dient als Mediator [Gam09] zwischen dem PrototypeRunner und der SimulationVisualization. Er übersetzt die Nutzereingaben aus der SimulationVisualization in die vom PrototypeRunner geforderten Marken und stößt am Ende eines Simulationsschrittes die Aktualisierung der Darstellung in der SimulationVisualization an. Außerdem steuert er das Persistieren von Informationen im TraceGenerator.

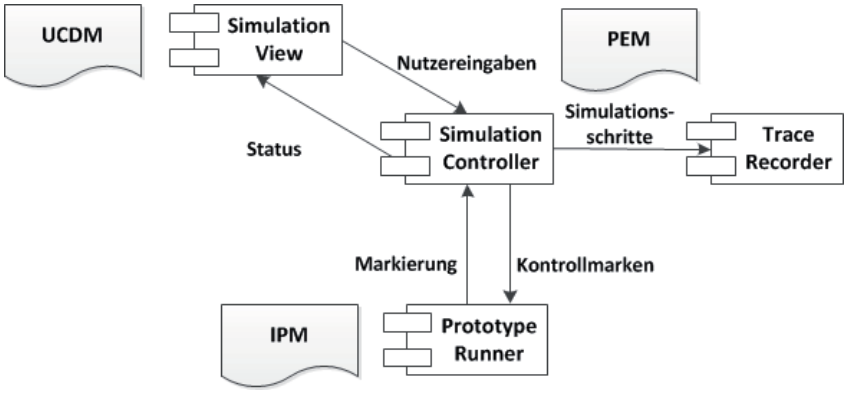


Abbildung 9.5: Bewertungsumgebung – Aufbau

9.3.4 Softwaretechnische Realisierung

Nachdem nun der allgemeine Aufbau der Bewertungsumgebung vorgestellt wurde, sollen einige wichtige Aspekte der softwaretechnischen Realisierung diskutiert werden. Hier werden im Wesentlichen die Mechanismen vorgestellt, die es ermöglichen, die Bewertungsumgebung flexibel zu erweitern.

Zeichnen der Nutzerschnittstelle: Während der Durchführung eines Simulationslaufes muss die graphische Nutzerschnittstelle des Systems emuliert

werden. Dabei muss die Darstellung bei jedem Simulationsschritt auf Basis des zum Simulationsschritt gehörenden GuiScreens neu erzeugt werden. Dazu wird in der SimulationVisualization jedes GuiElement des GuiScreens durch ein sogenanntes *Control* dargestellt. Um eine lose Kopplung zwischen Visualisierung und GDM zu gewährleisten, wird für die Erzeugung der Controls in der SimulationVisualization ein Strategiemechanismus verwendet. Dazu registriert die SimulationVisualization eine *GuiRenderingStrategyList*, die eine Menge von *GuiRenderingStrategies* enthält. Jede dieser GuiRenderingStrategies verfügt, wie in Abbildung 9.6 dargestellt, über zwei Methoden:

- **accepts:** Die accepts-Methode definiert, ob die GuiRenderingStrategy in der Lage ist, ein spezielles GuiElement zu zeichnen. Typischerweise wird diese Entscheidung auf Basis des Elementtyps gefällt. In speziellen Fällen werden zusätzlich bestimmte Eigenschaften des GuiElements überprüft.

- **draw:** Die draw-Methode zeichnet schließlich ein Control, das das zugehörige GuiElement darstellt, auf die Zeichenfläche der SimulationVisualization.

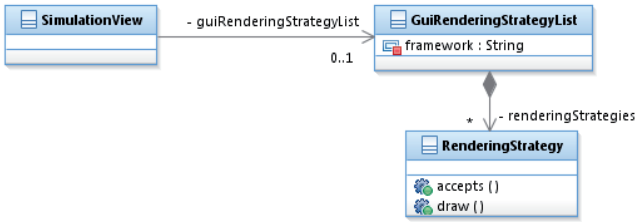


Abbildung 9.6: GuiRenderingStrategies

Mit diesen GuiRenderingStrategies kann während eines Simulationslaufes die Nutzerschnittstelle emuliert werden. Zu Beginn eines Simulationsschritts aktualisiert die SimulationVisualization die Ansicht. Dazu ermittelt der SimulationController das aktuelle Event und den zugehörigen GuiScreen. Der GuiScreen wird anschließend mit Hilfe der GuiRenderingStrategies in der SimulationView gezeichnet. Dazu werden alle GuiElements des GuiScreens der Reihe nach durchlaufen. Die SimulationVisualization sucht für jedes Element die passende RenderingStrategy. Wenn eine RenderingStrategy gefunden wurde, wird das zum GuiElement passende Control mit Hilfe der draw-Methode der RenderingStrategy auf der Zeichenfläche der SimulationVisualization angezeigt.

Einlesen von Nutzereingaben: Um Nutzereingaben aus den in der SimulationVisualization dargestellten Controls einlesen zu können, wird ein Observermechanismus (vgl. [Gam09]) verwendet. Beim Zeichnen eines Controls registriert sich der SimulationController als Observer beim Control. Das Control notifiziert dann den SimulationController über Nutzereingaben. Hierzu wird ein Eventme-

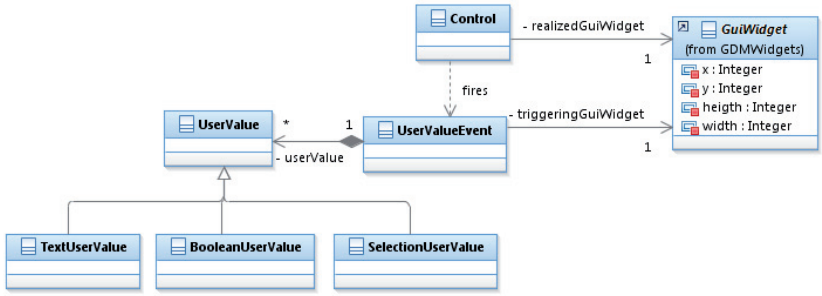


Abbildung 9.7: UserValueEvent

chanismus verwendet, um die flexible Erweiterbarkeit der Controls zu gewährleisten. Controls informieren den SimulationController, indem sie ein *UserValueEvent* erzeugen. Dieses enthält das Control und den Wert der Nutzereingabe. Mögliche Werte sind hier ein boolescher Wert für Buttons und Links, ein Text für TextWidgets oder eine Selektion, die eine oder mehrere Elemente enthält, für Viewer.

Ablaufsteuerung: Der Ablauf der Simulation wird auf Basis der Use Cases des UDM gesteuert. Konsequenterweise soll auch die Nutzerinteraktion in der Bewertungsumgebung entlang dieser Use Cases geführt werden. Allerdings sind die Nutzerschnittstellen interaktiver Systeme i.d.R. nicht deckungsgleich mit den Abläufen der Use Cases. Typischerweise werden die Nutzerschnittstellen anhand der Informationsbedürfnisse der Anwender, der Businessszenarien und spezieller Kriterien zur Benutzbarkeit entworfen. Deshalb enthalten Bildschirmseiten, die während der Ausführung eines Use Case gezeigt werden i.d.R. auch Controls, die nicht zum aktuellen Use Case gehören. Zusätzlich sind oftmals Controls, die mit unterschiedlichen Events des Ereignisflusses assoziiert sind, gleichzeitig sichtbar. Während der Simulation muss also unterschieden werden, welche Controls der gezeigten Bildschirmseite gerade aktiv sind. Genauer gesagt, muss entschieden werden, welche Controls die Erzeugung der aktuellen Kontrollmarken beeinflussen.

Dies entscheidet der SimulationController. Hierzu wird bei jedem eingehenden UserValueEvent überprüft, ob das erzeugende Control im aktuellen Schritt aktiv ist. Sollte dies der Fall sein, so wird der Wert des UserValueEvents in die Marke übertragen. Sonst wird das UserValueEvent im aktuellen Schritt ignoriert.

Der SimulationController unterstützt derzeit zwei Strategien, wie mit UserValueEvents von inaktiven Controls umgegangen wird. Entweder werden diese UserValueEvents verworfen und der Nutzer muss die Eingabe ggf. noch einmal machen, sobald das Control aktiv wird, oder UserValueEvents, die aus Controls kommen, die später im Ereignisfluss des Use Cases aktiv werden, werden

vorgehalten und automatisch verwendet, sobald das Control aktiv wird. Eingaben aus Controls, die im aktuellen Use Case nicht vorkommen, werden immer verworfen.

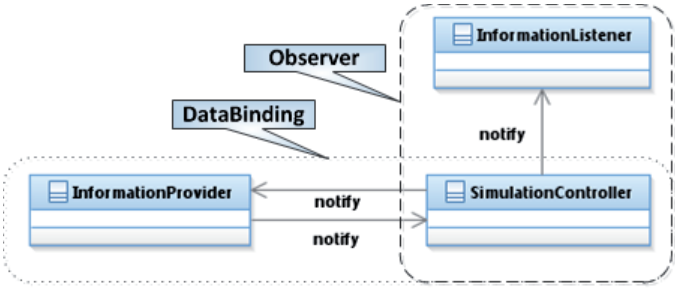


Abbildung 9.8: InformationProviders & InformationListeners

Zusatzinformationen zu Simulationsschritten: Simulationen werden immer zu einem bestimmten Zweck durchgeführt. Dabei ist es i.d.R. nötig, dass der Nutzer der Bewertungsumgebung neben den Nutzereingaben in die Simulation-View weitere einsatzzweck-spezifische Informationen angibt. Bei einem Review könnten dies beispielsweise Befunde oder Anmerkungen sein. Um diese anzugeben, werden je nach Einsatzzweck eine Reihe verschiedener Spezialwerkzeuge, die sogenannten *InformationProvider*, verwendet. Alle diese InformationProvider registrieren sich beim SimulationController. Dabei gehen der SimulationController und der InformationProvider ein DataBinding ein, d.h., sie registrieren sich gegenseitig als Observer. InformationProvider können dann ein EvaluationInfo erzeugen, das spezielle Informationen zu einem Simulationsschritt enthält. Die zu einem Schritt eingestellten EvaluationInfos werden dann gemeinsam mit dem SimulationStep im Trace der Simulation gespeichert.

Anzeigen von Ablaufinformationen während der Simulation: Analog zu den InformationProvidern, die dazu dienen, spezielle Informationen zur Simulation anzugeben, werden *InformationListener* verwendet, um spezielle Informationen über die Simulation anzuzeigen. Hier wird ebenfalls das Observer-Muster und ein Event-Mechanismus verwendet, um den SimulationController und die InformationListener zu entkoppeln. Der SimulationController verteilt die InfoEvents der InformationProvider an alle registrierten InformationListener.

10 Prototypying in der Use-Case-zentrierten Analyse

Typische Use-Case-zentrierte Ansätze bieten allgemeine Hinweise dazu, wie bestimmte Arten von Prototypen aus Use-Case-zentrierten Spezifikationen entwickelt werden können. Wie bereits in Kapitel 3.2 geschildert, fehlen aber detaillierte Angaben dazu, wie diese in der Analyse eingesetzt werden können.

Im Folgenden werden zunächst einige Vorüberlegungen zum Einsatz von Prototyping in der Use-Case-zentrierten Analyse gemacht. Anschließend werden die in SUPrA unterstützten Prototyparten vorgestellt und es wird ProDUCE, ein iteratives inkrementelles phasen-basiertes Vorgehen zur Use-Case-zentrierten Analyse, präsentiert.

10.1 Vorüberlegungen zum Einsatz von Prototyping

SUPrA erleichtert die Entwicklung verschiedener Arten interaktiver Nutzerschnittstellenprototypen und deren Einsatz in der Use-Case-zentrierten Analyse.

UCSM ermöglicht eine kompakte und präzise Spezifikation von Use-Case-zentrierten Spezifikationen und zugehörigen Prototypen. Die generative Prototypentwicklung vermeidet Inkonsistenzen zwischen Prototypen und der Spezifikation, und die enge Integration von Spezifikations- und Bewertungsnotation ermöglicht es, Prototypingergebnisse in die Use-Case-zentrierte Spezifikation zurückzuspiegeln.

Nun gilt es zu erörtern, wie die Produkte von SUPrA strukturiert zur Anforderungsanalyse eingesetzt werden können. Dazu müssen die folgenden Fragen beantwortet werden:

- Wann werden Prototypen im Prozess entwickelt?
- Welche Arten von Prototypen werden entwickelt?
- Wie werden diese aus UCSM erzeugt?
- Wozu werden diese Prototypen eingesetzt?
- Wie werden die Ergebnisse des Prototypings evaluiert?

10.2 Unterstützte Prototyparten

In SUPrA lassen sich abhängig vom Detaillierungsgrad des UCSM eine Reihe verschiedener Prototypen generieren. Im Folgenden werden die unterstützten Arten von Prototypen kurz vorgestellt und der minimal benötigte Detaillierungsgrad des UCSM diskutiert.

- **Wireframes:** Wireframes lassen sich ableiten, sobald GuiScreens im GDM spezifiziert wurden. Für das Erzeugen von Wireframes sind darüberhinaus keinerlei weitere Informationen nötig.

- **Storyboards:** Wie auch Wireframes lassen sich Storyboards prinzipiell komplett aus dem GDM erzeugen, ohne dass dieses mit Use Cases verbunden ist. Dazu müssen mehrere GuiScreens mit NavigationLinks verbunden sein. Diese Art von Storyboards bezeichnen wir im Folgenden als *Nutzerschnittstellen-Storyboards*.

  Alternativ lassen sich Storyboards aus der Ablaufstruktur des Ereignisflusses von Use Cases ableiten. Diese bezeichnen wir als *Use Case Storyboards*.

- **Papierprototypen:** Papierprototypen spielen im vorgestellten Ansatz nur eine untergeordnete Rolle. Nichtsdestotrotz können aus dem UCSM leicht Vorlagen für Papierprototypen erzeugt werden. Dazu können Bilder der GuiScreens bzw. der Wireframes ausgedruckt werden.

- **Digitale Prototypen:** Die wichtigste Art von Prototypen im vorgestellten Ansatz sind digitale Prototypen. Aus dem UCSM lassen sich eine Reihe verschiedener digitaler Prototypen ableiten.

  - **Screenflows:** Die einfachste Art von digitalen Prototypen sind die sogenannten *Screenflows*. Diese stellen mögliche Szenarien, z.B. den Ereignisfluss eines Use Cases dar, indem sie eine Reihe von GuiScreens verbinden. Fallunterscheidungen sind dabei rein verbal beschrieben. Verhaltensvarianten müssen also vom Nutzer explizit ausgewählt werden. Hier können analog zu den Storyboards *Nutzerschnittstellen Screenflows*, ohne Verbindung mit Use Cases, und *Use Case Screenflows* unterschieden werden.

  - **Semifunktionale Prototypen:** In *semifunktionalen Prototypen* sind zusätzlich Teile des Ablaufes formalisiert. Genauer gesagt, sind Variablen im UCSM modelliert und verschiedene Aktionen sind mit FunctionCalls angereichert. Also können bestimmte Aspekte der Ausführungslogik automatisch von der Bewertungsumgebung simuliert werden. Semifunktionale Prototypen bilden die wichtigste Art des Prototypings in SUPrA. Sie werden sowohl konstruktiv zur schrittweisen Verfeinerung des Verhaltens als auch analytisch zur Validierung von Abläufen eingesetzt.

  - **Use Case Simulation:** Eine Sonderform der digitalen Prototypen bilden *Use Case Simulationen*. Diese nutzen zur Darstellung des Ablaufes nicht, wie die anderen beschriebenen Prototypen, die graphische Nutzerschnittstelle, sondern blenden das Verhalten direkt auf der Use Case Beschreibung ein (vgl. [Leh08]). Wie auch bei den anderen digitalen Prototypen kann ihr Formalisierungsgrad variieren.

- **Wizard of Oz Prototypen:** Wizard of Oz Prototypen sind ebenso wie Papierprototypen ausschließlich ein Nebenprodukt des Ansatzes. Sie können aus denselben Modellen wie digitale Prototypen erzeugt werden. Der einzige Unterschied zu digitalen Prototypen ist, dass die Entscheidung über den Fortgang eines simulierten Szenarios nicht vom Prototypnutzer selbst, sondern von einem Experten, abhängig vom Verhalten des Prototypnutzers, getroffen wird.

- **Codierte Prototypen:** Codierte Prototypen lassen sich ableiten, wenn jeder Schritt der Ablaufbeschreibung durch einen FunctionCall formalisiert ist. Sie spielen im vorgestellten Ansatz allerdings nur eine untergeordnete Rolle, da die Anforderungsanalyse i.d.R beendet wird, bevor das Verhalten vollständig formalisiert ist. Lediglich für kleine Teile des spezifizierten Verhaltens wird gelegentlich ein ausreichender Formalisierungsgrad erreicht.

10.3 ProDUCE – Überblick

ProDUCE ist eine iterative, inkrementelle, phasen-orientierte Methode zur Entwicklung von Use-Case-zentrierten Anforderungsspezifikationen. ProDUCE besteht aus den drei in Abbildung 10.1 dargestellten Phasen: *Produktabgrenzung*, *Konzeptmodellierung* und *Detailmodellierung*. Jede dieser Phasen definiert eine Menge verschiedener Aktivitäten und wird jeweils durch ein Quality Gate abgeschlossen. Dabei werden in jeder Phase eine Reihe unterschiedlicher Projektrisiken adressiert.

Im Folgenden werden nun die einzelnen Phasen von ProDUCE der Reihe nach vorgestellt. Dazu werden zunächst jeweils die einzelnen Ziele der Phasen präsentiert. Anschließend werden die durchzuführenden Aktivitäten beschrieben. Der Schwerpunkt der Beschreibung liegt dabei auf den Prototypingaktivitäten. Alle anderen Aktivitäten werden lediglich kurz zusammengefasst, um den Prototypingaktivitäten einen Kontext zu geben. In ProDUCE gibt es zu jeder Aktivität eine kurze textuelle Beschreibung, die Ziel, Vorgehen und die erzeugten Artefakte festlegt. Ein Beispiel zeigt Abbildung 10.2.

In ProDUCE werden abhängig vom Detallierungsgrad der Beschreibungen und der Zielsetzungen der Phasen unterschiedliche Arten von Prototypen verwendet. Dabei werden sukzessive einfache statische Nutzerschnittstellenprototypen

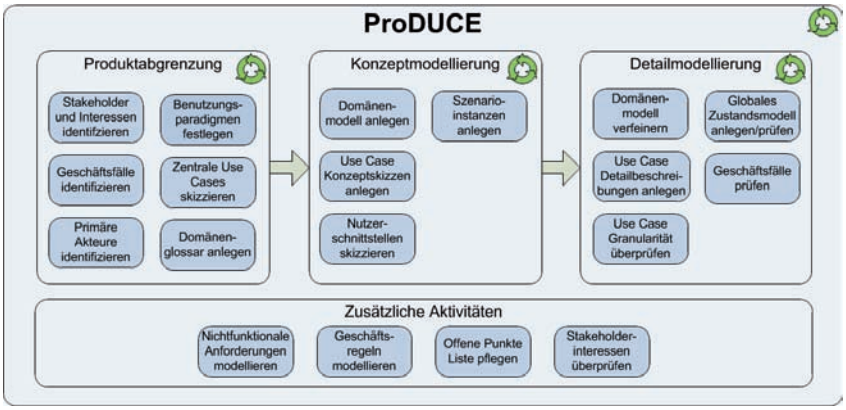


Abbildung 10.1: ProDUCE – Vorgehen

IDENTIFY ACTORS

All hands on deck

GOAL

Find the application's Actors. Actors are the people and applications that interact with this application.

GUIDELINES

Ask the **executive sponsor** who she thinks the actors are, and ask **each stakeholder**. Your definition of the actors may be fuzzy at this point, and that is OK. Define them as they arise and push concerns about duplication and definition into later iterations. At this point, **consider every stakeholder as a potential actor**.

Don't Confuse Actors with Organizational Roles or Job Titles. Often stakeholders tend to name those when asked for actors. Often the same business role or person may play the role of several different Actors in the System or several business roles are represented by one Actor.

Give Every Actor a Brief Description. This description should sum up the nature of the actor and state relevant facts about his knowledge considering the system.

Trace the Actors to the User Types, Stakeholders, and Stakeholder Roles. It is important to record the relationship between the actors and the user types, stakeholders, and stakeholder roles identified in the Vision document. Tracing the actors to the user types will help to capture and identify the actor's characteristics.

Don't use Generalization between Actors! It's usually wrong. Besides the information has little impact on the behavior.

Consider external Systems! External Systems that are needed for the execution of a use case are Actors too.

Ask yourself how the System is **set up** and **how information comes** into the System. This often leads to Actors that have been forgotten.

TEMPLATES :

Template_Actor_Goal_Priority.docx

Abbildung 10.2: ProDUCE – Aktivitätsbeschreibung

zu funktionalen Nutzerschnittstellenprotoytpen weiterentwickelt. Einen Überblick über die in den unterschiedlichen Phasen eingesetzten Prototypen gibt Abbildung 10.3.

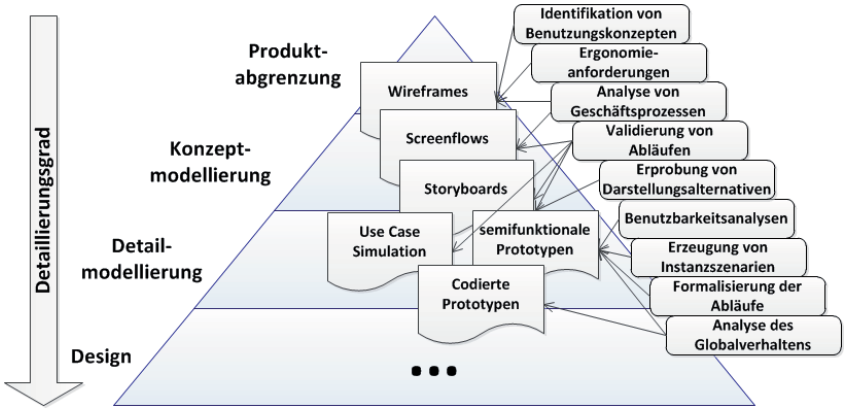


Abbildung 10.3: ProDUCE – Prototyparten nach Phasen und Zielen

10.4 Produktabgrenzung

Die Produktabgrenzungsphase bildet die erste Phase der Use-Case-zentrierten Analyse. Ihr Ziel ist es, die Grenzen des zu entwickelnden Systems zu identifizieren, Platzhalter für alle wichtigen Interaktionen zwischen Akteuren und dem System zu definieren und zentrale Konzepte festzulegen (vgl. [KG03]).

In der Produktabgrenzungsphase wird nur ein Minimum an Informationen modelliert, das nötig ist, um die Grenzen und Ziele des Produkts festzulegen und eine detailliertere Analyse vorzubereiten. Deshalb werden die Informationen im UCSM knapp gehalten. Use Cases werden ausschließlich in Form von Kurzbeschreibungen modelliert. Das Domänenmodell wird in Form eines unformalen Begriffslexikons beschrieben.

Modellierung: In der Produktabgrenzung soll der Kontext des entwickelten Systems abgesteckt und die Kernfunktionen identifiziert werden. Dazu schlägt ProDUCE die folgenden Aktivitäten vor:

- **Stakeholder und Interessen identifizieren:** Ziel der Entwicklung eines Softwaresystems ist es, die Bedürfnisse von Personen, Gruppen oder Institutionen zu befriedigen [Rup07]. Diese Stakeholder sollten möglichst früh in der Analyse identifiziert werden. Sie sind die Basis für detailliertere Analysen zu Nutzern und Funktionen des Systems. Anregungen zur

Durchführung einer Stakeholderanalyse geben beispielsweise Rupp et al. [Rup07].

- **Geschäftsfälle identifizieren:** Geschäftsfälle beschreiben die fachlichen Abläufe, an denen das modellierte System beteiligt ist. Teilweise werden sie auch als Business Use Cases [AM01] oder Summary Goals [Coc00] bezeichnet. Sie dienen der Einordnung des Systems in den geschäftlichen Kontext. Sie sind eine wichtige Quelle für die Analyse von Akteuren und Use Cases und dienen zur Abgrenzung des Systems gegen seine Umwelt. Wie Geschäftsfälle in der Use-Case-zentrierten Analyse erhoben werden können, diskutiert beispielsweise Cockburn [Coc00].

- **Primäre Akteure identifizieren:** Die primären Akteure sind die zentrale Quelle für die Identifikation von Use Cases. Deshalb müssen sie zu Beginn der Use Case Analyse identifiziert werden. Zur Identifikation der primären Akteure sind die Ergebnisse der Stakeholderanalyse und die identifizierten Geschäftsfälle hilfreich.

- **Zentrale Use Cases skizzieren:** In der Produktabgrenzung sollten die wichtigsten Systemfunktionen dokumentiert werden. Hierzu wird zu jeder dieser Funktionen eine Use Case Kurzbeschreibung (vgl. Abs. 2.4.3) angelegt. Als Basis für die Identifikation dienen Stakeholderlisten, Geschäftsfälle und die Ziele der primären Akteure.

- **Domänenglossar anlegen:** Parallel zur Analyse der Use Cases sollte ein Domänenglossar gepflegt werden. Im Domänenglossar werden die wichtigen Begriffe des Projektes gesammelt und eine Definition formuliert.

- **Benutzungsparadigmen festlegen:** In vielen Situationen empfiehlt es sich, die grundsätzlichen Interaktionsparadigmen für die zentralen Benutzer früh festzulegen. Hierzu werden Benutzungsmetaphern definiert und einige allgemeine Bedienkonzepte festgelegt.

Prototyping: Aufgrund der fehlenden Definition von Interaktionsabläufen spielen interaktive Prototypen in dieser Phase keine Rolle. Stattdessen werden statische Nutzerschnittstellenprototypen verwendet. Allerdings weisen Constantine & Lockwood [CL99] darauf hin, dass Prototypen in der Produktabgrenzung grundsätzlich sparsam eingesetzt werden und nicht zu detailliert sein sollten, da die Gefahr besteht, sich zu früh auf eine u.U. ungeeignete Lösung festzulegen. In der Produktabgrenzungsphase werden vor allem folgende Aktivitäten durch Prototyping unterstützt:

- **Identifikation von Benutzungskonzepten:** Zu Beginn der Analyse fällt es vielen Klienten schwer, sich das System lediglich anhand der angebotenen Funktionen vorzustellen. Deshalb werden für verschiedene Rollen, Gruppen von Aufgaben oder Systemteile Konzeptprototypen erzeugt. Konzeptprototypen sind sehr knappe Wireframes, die lediglich aus 1-3

wichtigen Interaktionselementen bestehen. Sie sollen Nutzern die Möglichkeit geben, sich vorzustellen, wie bestimmte Systemfunktionen angeboten werden könnten. Sie bieten früh die Möglichkeit, Benutzungskonzepte mit Nutzern zu evaluieren. Außerdem bilden sie den Ausgangspunkt für detailliertere Analysen der Nutzerschnittstelle.

- **Analyse von Geschäftsprozessen:** Frühe Business-Storyboards können zur Validierung der Geschäftsprozesse, die vom spezifizierten System unterstützt werden sollen, herangezogen werden. Dazu werden Konzeptprototypen mit den Schritten der Geschäftsabläufe verbunden. Dies erleichtert vielen Klienten den Zugang zu den Geschäftsprozessbeschreibungen und ermöglicht ihnen eine detailliertere Analyse der einzelnen Prozessschritte. Durch die Analyse werden oft fehlende Konzeptprototypen identifiziert. Diese weisen entweder auf einen fehlenden Prozessschritt, eine fehlende Rolle oder ein nicht beachtetes Fremdsystem hin.

- **Kommunikation von Ergonomieanforderungen:** Vielen Klienten fällt es leichter, Ergonomieanforderungen wie Drag&Drop Unterstützung, Direct Editing oder Copy&Paste Funktionen zu spezifizieren, wenn ein Konzeptprototyp vorliegt. Diese Ergonomieanforderungen führen zu einer erhöhten Akzeptanz der Lösung. Außerdem geben sie oft Hinweise auf neue innovative Systemfunktionen.

10.5 Konzeptmodellierung

Ziel der Konzeptmodellierungsphase ist es, die zentralen Systemfunktionen klar festzulegen. Dazu werden Use Case Kurzbeschreibungen sukzessive zu flussorientierten Beschreibungen weiterentwickelt. Diese enthalten den Normalablauf und wichtige Alternativabläufe sowie Vor- und Nachbedingungen. Parallel werden die Domänenkonzepte des Begriffslexikons zueinander in Beziehung gesetzt. Prototypen der Nutzerschnittstellen werden von Konzeptprototypen und „virtuellen Fenstern“ (vgl. [LH01]) zur vollständigen Darstellungen einzelner Nutzerseiten ausgebaut. Angebotene Systemfunktionen werden identifiziert und ihre Funktionen verbal beschrieben.

Modellierung: In der Konzeptmodellierungsphase werden auf Basis der Normalabläufe des Systemverhaltens die Grundkonzepte des Zielsystems modelliert. Hierzu beschreibt ProDUCE die folgenden Aktivitäten:

- **Domänenmodell anlegen:** In der Konzeptmodellierung wird ein fachliches Modell der Domäne angelegt. Dieses legt die Entitäten der Domäne und deren Beziehungen fest. Grundlage hierfür bildet das Begriffslexikon.

- **Use Case Konzeptskizzen anlegen:** Die Use Case Kurzbeschreibungen

werden zu Use Case Skizzen verfeinert. Dabei entstehen oft auch neue Use Cases. Zusätzlich werden Beziehungen zwischen Use Cases modelliert.

- **Nutzerschnittstellen skizzieren:** Wie bereits erwähnt, bildet die Nutzerschnittstelle bei interaktiven Informationssystemen einen Teil der Anforderungsspezifikation. In der Konzeptmodellierung sollten die Nutzerschnittstellenseiten für alle wichtigen Use Cases angelegt werden. Hierzu empfiehlt sich ein Vorgehen nach dem Virtual Windows Ansatz (vgl. [LH01]).

- **Szenarioinstanzen anlegen:** Szenarioinstanzen der wichtigsten Use Cases sollten ebenfalls in der Konzeptmodellierung angelegt werden. Sie dienen zur Validierung der Use Cases, zur Erprobung der Nutzerschnittstelle, insbesondere im Hinblick auf zu erwartende Datenmengen, und als Input für die Erzeugung von Testspezifikationen.

Prototyping: In der Konzeptmodellierungsphase spielen Prototypen in verschiedenen Modellierungsaktivitäten eine zentrale Rolle. Es werden im Wesentlichen Storyboards und digitale Prototypen eingesetzt. Hier werden insbesondere die folgenden Tätigkeiten unterstützt:

- **Validierung der Ablaufbeschreibungen:** Zur Validierung der Ablaufbeschreibungen werden die in den flussorientierten Beschreibungen enthaltenen Szenarien einzeln mit Klienten durchgespielt und Inkonsistenzen und Fehler gesucht. Dazu werden Storyboards oder digitale Prototypen eingesetzt. Für kompliziertes oder zustandsbehaftetes Verhalten können teilweise auch bereits codierte Prototypen entwickelt werden.

- **Benutzbarkeitsanalysen:** Die Benutzbarkeit der entwickelten Lösungskonzepte kann mit Prototypen überprüft werden. Dazu simulieren Prototypnutzer festgelegte Szenarien mit Hilfe von digitalen Prototypen. Gleichzeitig werden bestimmte Benutzbarkeitsmetriken (vgl. [TA08]) bestimmt. Dabei kann das Szenario sowohl vom Prototypnutzer selbst durchgespielt werden als auch im Sinne von Wizard-of-Oz-Prototyping von einem Experten simuliert werden.

- **Erprobung von Darstellungsformen:** Angemessene Darstellungsformen für bestimmte Datensätze hängen signifikant von der Menge der darzustellenden Datensätze ab. Mit Hilfe von digitalen Prototypen können verschiedene Darstellungsformen ausprobiert werden. Dazu werden dieselben Szenarien mit typischen Datensätzen mehrfach mit unterschiedlichen Darstellungsformen durchgespielt.

- **Identifikation wichtiger Instanzszenarien:** Armour und Miller [AM01] empfehlen die Entwicklung von Instanzszenarien als Basis für die Evaluierung der Spezifikation und das Testen. Diese können mit Hilfe von

digitalen Prototypen leicht aufgezeichnet werden, indem Nutzereingaben und gewünschte Systemausgaben während der Ausführung notiert werden (vgl. [HK02]).

10.6 Detailmodellierung

Ziel der Detailmodellierungsphase ist es, eine vollständige, konsistente Sicht auf die spezifizierten Anforderungen zu erzeugen. Dazu werden Modellierungslücken geschlossen. Beispielsweise werden fehlende Alternativszenarien im UCSM spezifiziert. Außerdem werden die unterschiedlichen Modellierungsperspektiven harmonisiert.

Modellierung: In der Detailmodellierungphase wird eine vollständige Spezifikation des Systems erstellt, indem die bisher modellierten Systemaspekte verfeinert werden. Dazu schlägt ProDUCE die folgenden Aktivitäten vor:

- **Domänenmodell verfeinern:** In der Detailmodellierung sollte das Domänenmodell zu einem vollständigen Modell der fachlichen Entitäten, deren Beziehungen und Verantwortlichkeiten ausgebaut werden. Hierzu wird das vorhandene Domänenmodell sukzessive verfeinert. Insbesondere werden die wichtigsten Verantwotlichkeiten der Entitäten hinzugefügt.

- **Use Case Detailbeschreibungen anlegen:** Um die fachlichen Abläufe vollständig zu spezifizieren, werden in der Detailmodellierung die Use Cases zu Detailbeschreibungen ausgebaut. Dazu werden alle Alternativen und Ausnahmen beschrieben.

- **Use Case Granularität prüfen:** Ziel ist es, in der Detailmodellierung eine gleichmäßige, durchgehende Spezifikation zu bekommen. Deshalb sollte die Granularität der einzelnen Use Case Beschreibungen geprüft werden. Zu große Use Cases sollten in mehrere kleinere Use Cases aufgebrochen werden. Use Cases, die nur aus wenigen Schritten bestehen, sind entweder Teil eines anderen Use Cases und haben keinen eigenständigen fachlichen Nutzen oder sie sind nicht detailliert genug beschrieben und müssen deshalb verfeinert werden.

- **Globales Zustandsmodell anlegen/prüfen:** In der Detailmodellierung sollte ein globales Zustandsmodell angelegt werden. Ein solches Zustandsmodell lässt sich aus den Use Case Vor- und Nachbedingungen ableiten. Mit diesem kann überprüft werden, ob das System nach jedem möglichen Ende eines Use Cases in einem Zustand ist, in dem es weiter arbeiten kann. Genauer gesagt, kann geprüft werden, ob das System in einen Zustand versetzt werden kann, an dem kein Use Case mehr ausführbar ist.

- **Geschäftsfälle prüfen:** Am Ende der Anforderungsmodellierung sollte

ein System spezifiziert sein, mit dem die geforderten Geschäftsfälle unterstützt werden können. Dies sollte in der Detailmodellierung geprüft werden. Hierzu werden die Geschäftsfälle anhand der Use Cases durchgespielt.

Prototyping: In der Detailmodellierungsphase werden alle Use Cases in eine vollständig flussorientierte Form überführt. Alle Alternativ- und Ausnahmeszenarien werden beschrieben. Zusätzlich wird eine Verbindung zwischen Use Cases und Nutzerschnittstelle sowie Use Cases und ApplicationFacade hergestellt. Durch diese Kombination wird die automatische Generierung von codierten und semifunktionalen Prototypen möglich. Diese werden für die folgenden Zwecke eingesetzt:

- **Formalisierung von Verhaltensbeschreibungen:** Digitale oder partiell codierte Prototypen können konstruktiv zur Formalisierung des Verhaltens eingesetzt werden. Dazu werden einzelne Szenarien durchlaufen und für jeden Schritt eine Formalisierung angegeben. Für Schritte des Akteurs werden GuiWidgets mit Systemvariablen verknüpft. Für Systemschritte werden Funktionsaufrufe der ApplicationFacade und die übergebenen Parameter spezifiziert.

- **Validierung von Geschäftsprozessen:** Mit Hilfe von codierten Prototypen kann untersucht werden, ob das spezifizierte System in der Lage ist, die adressierten Geschäftsprozesse zu unterstützen. Hierzu werden wichtige Geschäftsprozesse durchgespielt. Externe Applikationen, die im Geschäftsprozess eine Rolle spielen, werden i.d.R. durch Stubs, z.B. mit Hilfe von Skripten, gekapselt.

- **Analyse des Use Case Zusammenspiels:** Durch Ausführung mehrerer Use Cases nacheinander, können diese im Kontext evaluiert werden. Hierbei wird überprüft, ob Situationen herstellbar sind, in denen kein Use Case mehr ausführbar ist. Solche Situationen deuten i.d.R auf fehlende Szenarien oder auf zu enge Vorbedingungen hin.

10.7 Zusätzliche Aktivitäten

Zusätzlich zu den Aktivitäten in den einzelnen Phasen werden in der ProDUCE Methode noch eine Reihe weiterer Aktivitäten vorgeschlagen, die in den meisten Projekten eine Rolle spielen. Diese sind aber nicht Teil des phasen-orientierten Vorgehens, sondern werden parallel zu allen Phasen durchgeführt.

- **Nicht-funktionale Anforderungen modellieren:** Während der gesamten Modellierung sollten die verschiedenen nicht-funktionalen Anforderungen dokumentiert werden. Hierzu gibt es sehr unterschiedliche An-

sätze. Eine gute Zusammenfassung geben Rupp et al. [Rup07].

- **Geschäftsregeln modellieren:** Geschäftsregeln werden, wie auch nichtfunktionale Anforderungen, in typischen Use-Case-zentrierten Ansätzen nicht in einer speziellen Modellierungsaktivität identifiziert. Stattdessen entstehen sie als Nebenprodukt bei anderen Modellierungstätigkeiten. Sie sind dennoch ein wichtiges Analyseergebnis und müssen dokumentiert werden. Ansätze hierzu beschreibt beispielsweise Pohl [Poh08].

- **Offene-Punkte-Liste pflegen:** Bei einem Breadth-First Ansatz können in allen Modellierungstätigkeiten Informationen auftauchen, die in der aktuellen Phase nicht behandelt werden. Damit diese Informationen nicht verloren gehen, werden sie in einer Offenen-Punkte-Liste dokumentiert. Zu Beginn einer jeden Phase sollte die Offene-Punkte-Liste analysiert und relevante Punkte abgearbeitet werden.

- **Stakeholderinteressen überprüfen:** Neben den Systemakteuren, die direkt an der Durchführung der Use Cases des Systems beteiligt sind, gibt es i.d.R. noch eine Reihe weiterer Stakeholder, die ebenfalls Interessen gegenüber dem System haben. Dies können beispielsweise andere Nutzerrollen sein, die die Ergebnisse, die in einem Use Case erzeugt werden, in einem anderen Use Case weiterverarbeiten müssen. Es sind aber auch externe Stakeholder, wie Manager oder gesetzgebende Behörden, zu betrachten. Bei der Analyse des Systemverhaltens ist es wichtig darauf zu achten, dass auch die Interessen dieser Stakeholder gewahrt werden. Eine umfassende Diskussion hierzu gibt Cockburn [Coc00].

10.8 Prototyping beim Übergang zum Systemdesign

Neben den Anwendungsszenarien während der Anforderungsanalyse, die in der ProDUCE Methode behandelt werden, können die entwickelten Prototypen für weitere Aktivitäten im Entwicklungsprozess eingesetzt werden. Hierzu werden primär die Analyseergebnisse des Prototypeneinsatzes verwendet:

- **Testspezifikation:** Die Bewertungsumgebung kann ähnlich wie ein Testrecorder zur Spezifikation von funktionalen System- bzw. Abnahmetests verwendet werden. Dazu werden Testszenarien durch Eingaben in die Nutzerschnittstelle spezifiziert. Zusätzlich werden die Ausgaben des Systems für die Beschreibung von Sollwerten verwendet. Wie Testspezifikationen strukturiert aus UCSM abgeleitet werden können, diskutiert Kiliclar [Kil10]. Conrad [Con12] stellt einen Ansatz vor, wie sich aus solchen Testspezifikationen automatisierte Testfälle ableiten lassen.

- **Nutzerdokumentation:** Prototypen können als Basis für die Nutzerdokumentation sehr hilfreich sein. Dazu werden Storyboards für relevante

Szenarien aufgezeichnet. Diese helfen bei der Beschreibung einer funktionsorientierten Nutzerdokumentation insbesondere dann, wenn die Dokumentation von Personen erstellt wird, die nicht an der Systementwicklung beteiligt waren. Dies ist beispielsweise der Fall, wenn sie von einer speziellen Dokumentationsabteilung oder einer externen Firma, erzeugt wird.

- **Generative Entwicklung von partiellen Lösungen:** Die Prototypingnotationen sind formale Sprachen. Deshalb ist es möglich, aus diesen generativ Teile eines evolutionären Prototyps oder des Zielsystems zu erzeugen. Im Falle von UCSM bietet sich insbesondere die Nutzerschnittstellenbeschreibung an. Es ist uns aber auch gelungen, eine komplette EJB-Applikation mit Hilfe eines speziellen Codegenerators (vgl. [Löw11]) aus einem für das Prototyping formalisierten UCSM-Modell zur in Abschnitt 6 vorgestellten Beispielapplikation zu erzeugen.

11 Werkzeugunterstützung

Das Open Use Case Modeling Framework (OpenUMF) ist eine integrierte Prototypingumgebung für die Use-Case-zentrierte Anforderungsmodellierung. OpenUMF wurde im Rahmen dieser Dissertation und in mehreren Diplomarbeiten entwickelt. Die Entwicklung von OpenUMF begann als Erweiterung des Modellierungswerkzeugs ViPER. 2007 wurde die metamodell-basierte Use Case Modellierungskomponente NaUTILUS [Wal07, HLNW09], der Vorgänger von OpenUMF, zu ViPER hinzugefügt. Seit Ende 2008 wird ViPER-NaUTILUS als eigenständiges Produkt betrieben, das Ende 2011 in OpenUMF umbenannt wurde. Während dieser Zeit wurde OpenUMF kontinuierlich weiterentwickelt. Dem ursprünglichen Use Case Modellierungswerkzeug wurden sukzessive Modellierungsperspektiven für Domänenmodelle [Fon10], graphische Nutzerschnittstellen und Systemfunktionen hinzugefügt. Außerdem wurden Werkzeuge zur Simulation der spezifizierten Modelle [Leh08, Ren10] entwickelt. Weiterhin enthält OpenUMF heute Komponenten für die Erzeugung von Testspezifikationen auf Basis von Use-Case-zentrierten Spezifikationen [Kil10], zur Beschreibung und Bewertung von Metriken für Use-Case-zentrierte Spezifikationen [Kla09] und zur Beschreibung von Modellierungsmethoden [Kra11], die in dieser Arbeit nicht weiter behandelt werden.

OpenUMF baut auf Eclipse-Technologie [ecl] und dem Plattformrahmenwerk ViPER-Platform [Nys09] auf, das Funktionaläten zur Verwaltung von EMF-Ressourcen [emf] enthält. Im Folgenden wird der softwaretechnische Aufbau vom OpenUMF kurz skizziert. Die einzelnen Werkzeuge, aus denen OpenUMF besteht, sind bereits in den Kapiteln 7 bis 9 diskutiert worden.

Zur Entwicklung von UCSM Modellen beinhaltet OpenUMF die bereits eingeführten graphischen und formularbasierten Werkzeuge von UCMT (vgl. 7.2). Außerdem enthält OpenUMF eine Implementierung des Prototypgenerators PGT (vgl. 8.3) und der Bewertungsumgebung PET (vgl. 9.3).

11.1 Das Prototypingwerkzeug OpenUMF

Wie bereits erwähnt, ist OpenUMF eine integrierte Entwicklungsumgebung (Integrated Developent Environement IDE) für die UCSM Modellierung, Prototypgenerierung und Simulation. Neben den Werkzeugen zur Use-Case-zentrierten Analyse enthält sie eine Reihe weiterer Eclipse-Basiskomponenten. Dazu zählt z.B ein kompletter *Eclipse SDK* [ecl], bestehend aus der *Eclipse Plattform* der

Java Entwicklungsumgebung *JDT* und der Plugin Entwicklungsinfrastruktur *PDE*. Somit ist OpenUMF auch eine voll ausgestattete Java Entwicklungsumgebung. Weiterhin bietet OpenUMF generische Unterstützung für die modellgetriebene Entwicklung inklusive Modellvalidierung, Modell-zu-Modell und Modell-zu-Text-Transformation. Dazu sind verschiedene Komponenten aus dem Rahmenwerk *Eclipse Generative Modeling Technologies GMT* in OpenUMF integriert. Außerdem enthält OpenUMF eine Reihe weiterer Basisrahmenwerke, wie z.B. Graphiti [gra] zur Entwicklung von graphischen Editoren. Dieser Aufbau macht es möglich, OpenUMF als seine eigene Entwicklungsumgebung einzusetzen und alle von OpenUMF angebotenen Funktionen in OpenUMF weiterzuentwickeln.

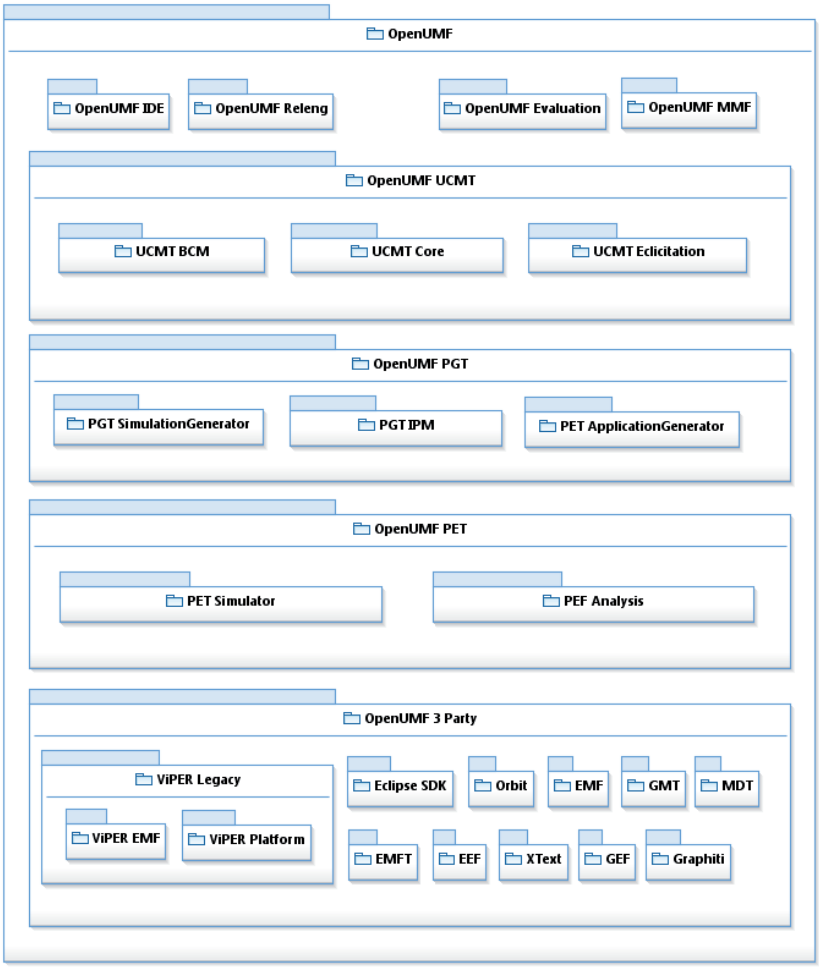


Abbildung 11.1: OpenUMF Architekturübersicht

Die Kernfunktionalität von OpenUMF besteht aus der Spezifikationssumgebung UCST, dem Prototypgenerator PGT und einer Bewertungsumgebung PET. Diese werden jeweils durch ein Feature realisiert. Der Begriff Feature ist dabei aus der OSGI-Spezifikation [OSG09] entnommen. Im Eclipse-Umfeld bezeichnet er eine Menge von Plugins[1], die eine bestimmte Funktion bereitstellen. OpenUMF besteht also aus den folgenden Features:

- OpenUMF Use Case Centered Modeling Toolkit (UCMT)
- OpenUMF Prototype Generation Toolkit (PGT)
- OpenUMF Prototype Evaluation Toolkit (PET)

Weiterhin enthält OpenUMF ein Plattform Feature. Dieses stellt keine Endnutzerfunktionalität bereit, sondern dient als Rahmenwerk für die Entwicklung aller anderen OpenUMF Features.

Abbildung 11.1 stellt eine Übersicht der Architektur von OpenUMF dar. Neben den bereits vorgestellten Features enthält die OpenUMF zwei weitere Plugins:

- *OpenUMF IDE* enthält die Produktdefinition von OpenUMF in Form eines Eclipse RCP (Rich Client Platform).
- *OpenUMF Releng* enthält das Release Engineering von OpenUMF, es definiert eine automatische Build- und Testinfrastruktur.

11.1.1 OpenUMF UCMT

Das OpenUMF UCMT Feature bündelt alle Funktionalitäten für die Modellierung von Use-Case-zentrierten Anforderungsspezifikationen. Ein Überblick über die Pluginstruktur von OpenUMF UCMT ist in Abbildung 11.2 dargestellt. Hier erkennt man, dass OpenUMF UCMT aus drei Teilen besteht:

OpenUMF UCMT BCP definiert, ähnlich wie das Plattform Feature, eine Basisinfrastrukur für die UCSM Modellierung. Es enthält eine Implementierung des Basic Concept Models und das Plugin BCP Infrastructure. Dieses bietet eine Reihe von Basisklassen für die verschiedenen Werkzeuge von UCSM an. Außerdem definiert es Dienste für die Navigation zwischen den verschiedenen Sichten von UCSM Core und UCSM Extension und enthält Funktionen für die Validierung der Gesamtmodelle. Das Plugin BCP GraphitiTools enthält generische Funktionalitäten für die Entwicklung graphischer Editoren für BCM Modelle.

OpenUMF UCMT UCSM enthält jeweils ein separates Feature für alle Modellierungssichten von UCSM. Jedes dieser Features definiert, neben einer Implementierung eines der in Kapitel 7.1 vorgestellten Submodelle, die zugehörigen

[1] Dieser Begriff wird in Eclipse synonym zu Bundle in der OSGI-Spezifikation verwendet.

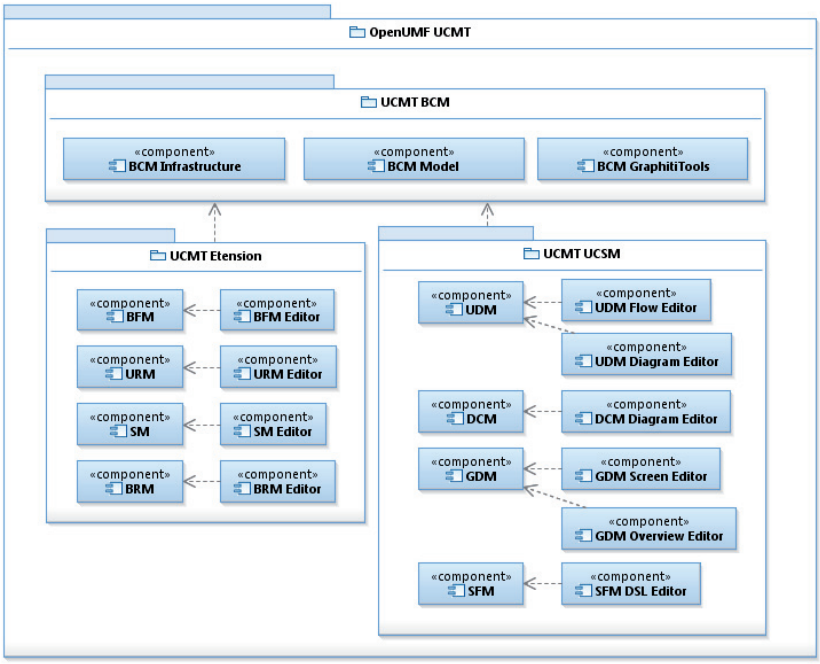


Abbildung 11.2: OpenUMF UCMT

Modellierungswerkzeuge. Einige dieser Werkzeuge sind in Abbildung 11.3 noch einmal dargestellt.

OpenUMF UCMT Extension enthält Modellierungswerkzeuge für die Modellierung von weiteren vom BCM abgeleiteten Anforderungsmodellen, insbesondere für frühe Phasen der Anforderungsanalyse. Diese unterstützen die in dieser Arbeit nicht behandelten Modellierungsaspekte von ProDUCE. OpenUMF UCMT Extension enthält Modelle für unstrukturierte Anforderungslisten (URM), Stakeholdermodelle (SM), Business Cases (BCM) und Geschäftsregeln (BRM). Diese Modelle werden in dieser Arbeit nicht weiter betrachtet, da sie für das Prototyping keine Rolle spielen.

11.1.2 OpenUMF PGT

Das OpenUMF PGT Feature enthält die Funktionalität für die Generierung von Prototypen aus UCSM Modellen. Seine Struktur ist in Abbildung 11.4 dargestellt. Es besteht aus drei Teilen.

OpenUMF PGT IPM enthält die Implementierung des IPM sowie den in Abschnitt 8.2.1 vorgestellten Editor zum Anlegen und Verändern der Modelle zu

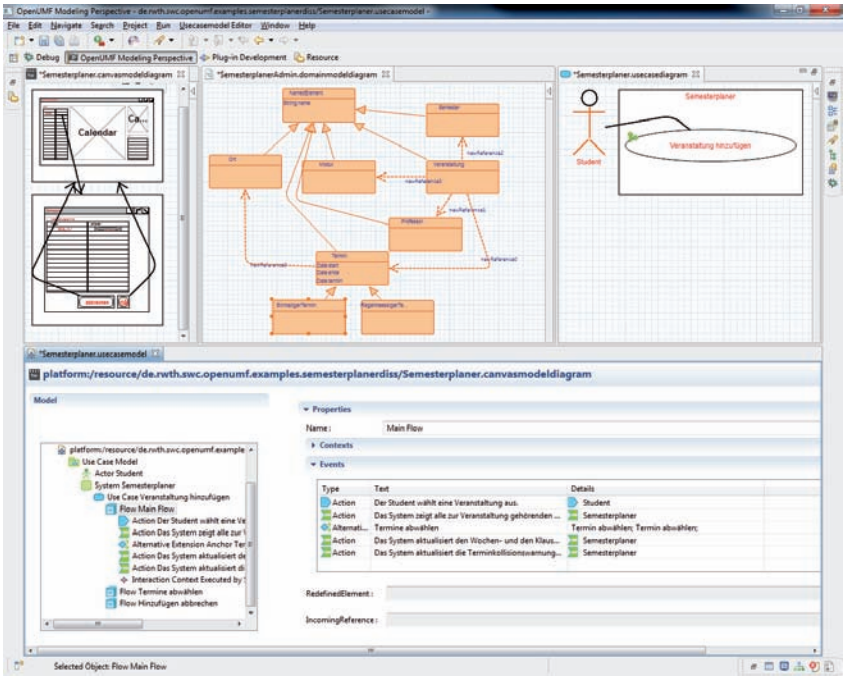


Abbildung 11.3: OpenUMF UCMT Werkzeuge (Screenshot)

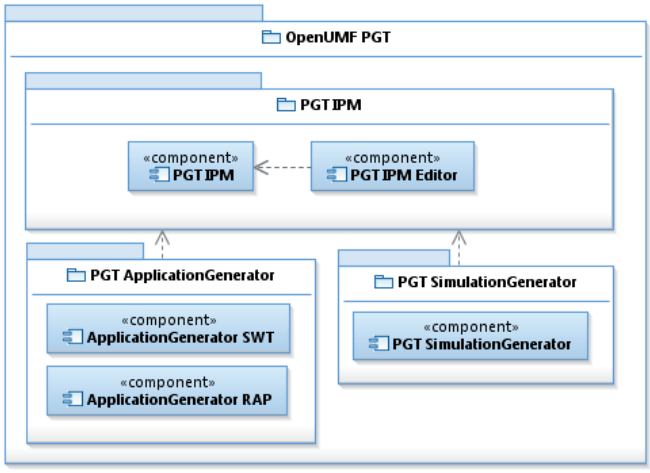


Abbildung 11.4: OpenUMF PGT

Testzwecken.

OpenUMF PGT SimulationGenerator enthält eine Referenzimplementierung des UCSM Modellgenerators (vgl. 8.3). Der Modellgenerator bietet dabei derzeit die Generierung von unterschiedlichen Ausführungssemantiken an. Zum einen realisiert er den `Strict Mode`. Dieser implementiert den in Kapitel 8.3 beschriebenen Generierungsalgorithmus. Zum anderen bietet der Generator einen `Simple Mode`. Dieser erzeugt ein IPM Modell mit einer ausschließlich nutzerschnittstellenorientierten Ausführungssemantik. Dieses IPM Modell erlaubt es, alle sichtbaren Kontrollelemente eines GuiScreens zu verwenden, unabhängig davon, ob sie derzeit in einem Use Case aktiviert sind.

OpenUMF PGF ApplicationGenerator ist ein Generator für Standalone Simulationsapplikationen. Er ist in der Lage, aus einem UCSM Modell verschiedene Arten von Prototypen zu generieren, die ohne die Bewertungsumgebung lauffähig sind. *PGT ApplicationGenerator SWT* erzeugt Applikationen auf Basis von SWT und Java. *PGT ApplicationGenerator RAP* erzeugt RAP-Applikationen [rap]. Alle diese Prototypen können Simulationsläufe wie im Simulator ausführen. Dabei sind die Prototypen auch in der Lage, Traces aufzuzeichnen.

11.1.3 OpenUMF PET

Der Aufbau des Features OpenUMF PET ist in Abbildung 11.5 skizziert. Es bietet die Infrastruktur zur Simulation und Bewertung von UCSM-Prototypen. Es besteht aus drei Teilen:

OpenUMF PET PEM enthält eine Implementierung des PEM (vgl. 9.2) sowie den zugehörigen Editor.

OpenUMF PET Simulator enthält die Simulationsumgebung, in der OpenUMF Prototypen ausgeführt werden können. Das Plugin *PET Simulator Core* implementiert die in Abschnitt 9.3 beschriebene Architektur. Es enthält einen graphiti-basierten graphischen SimulatorViewer, einen IPMPrototypeExecutor, einen SimulationController und einen TraceRecorder. Außerdem bietet PET Simulator Core Erweiterungspunkte für die Anbindung von InformationProvidern und InformationListenern. Das Plugin *PET Simulator Views* enthält eine Sammlung von allgemeinen InformationListenern. Das Plugin *PET Simulator Review* enthält eine Erweiterung für die Durchführung von Reviews.

OpenUMF PET Analysis bietet verschiedene Werkzeuge zur Analyse von Simulationsläufen. Es enthält Mechanismen für Überdeckungsmetriken und für die Revalidierung von Traces nach Änderungen in einem UCSM Modell.

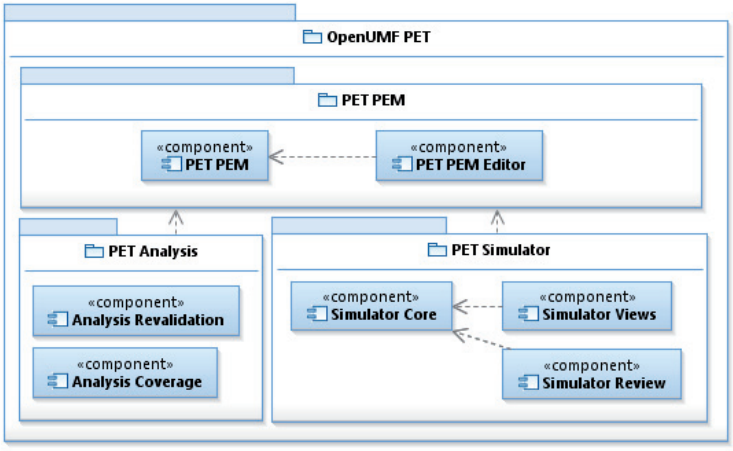


Abbildung 11.5: OpenUMF PET

12 Erfahrungen aus der Praxis

Die Notation UCSM, die ProDUCE Methode und die zugehörige Werkzeugunterstützung OpenUMF sind in verschiedenen Projekten in der Forschungsgruppe Softwarekonstruktion der RWTH Aachen (SWC) und in industriellen Kooperationen eingesetzt worden. Der folgenden Abschnitt berichtet von praktischen Erfahrungen mit der Anwendung der Ansätze und Werkzeuge. Dazu werden zunächst drei charakteristische Projekte vorgestellt. In jedem dieser Projekte wurden abhängig vom Zustand der Entwicklung von SUPrA und den Rahmenbedingungen des Projekts unterschiedliche Teile von SUPrA eingesetzt.

In den Fallbeispielen werden zunächst jeweils kurz die betrachteten Fragestellungen diskutiert. Anschließend wird der Aufbau des Projekts, der Ablauf der Analyse und die eingesetzten Aspekte von SUPrA vorgestellt. Zuletzt wird der Einsatz von SUPrA im Projekt bewertet. Zusätzlich werden in diesem Kapitel Erfahrungen aus einer Reihe weiterer Fallbeispiele zusammengefasst und eine Gesamteinschätzung zu SUPrA gegeben.

12.1 Eingruppierung der Fallbeispiele

Bei den hier vorgestellten Fallbeispielen handelt es sich ausschließlich um Erfahrungsberichte aus dem praktischen Einsatz von SUPrA. Es wurden keine kontrollierten Experimente (controlled experiment) oder Fallstudien (case study) durchgeführt (vgl. [MAKS12]).

Der Einsatz von kontrollierten Experimenten bot sich aufgrund der Fragestellungen der Arbeit nicht an. Mit kontrollierten Experimenten lassen sich nur einzelne Teilaspekte eines Vorgehens oder Ansatzes analysieren. Zudem sind sie sehr aufwendig. Ziel dieser Arbeit war es, einen durchgängigen Ansatz für Prototyping in der Use-Case-zentrierten Analyse zu entwickeln. Das Hauptaugenmerk lag dabei auf der praktischen Anwendbarkeit in kleinen bis mittelgroßen Industrieprojekten und auf der Durchgängigkeit des Ansatzes. Zur Beurteilung dieser beiden Aspekte sind kontrollierte Experimente ungeeignet, da es unmöglich ist, alle unabhängigen Variablen zu kontrollieren. Deshalb wurde auf ihre Durchführung verzichtet.

Ziel von Case Studies ist die quantitiative bzw. qualitative Beobachtung einiger wichtiger Parameter. Grundsätzlich sind Case Studies zwar geeignet, um Industrieprojekte zu bewerten, da sie nur einen relativ kleinen Mehraufwand verlangen

und als unabhängige Erweiterung zu einem Projekt betrieben werden können. Da nicht alle unabhängigen Variablen kontrolliert werden, ist es bei Einzelbeobachtungen aber schwierig, Rückschlüsse auf die Gründe von Beobachtungen zu machen. Deshalb ist es, um belastbare Aussagen zu bekommen, nötig, mehrere ähnliche Projekte, idealerweise mit kleinen Änderungen am Vorgehen durchzuführen (vgl. [MAKS12]). Dies war aus zwei Gründen für die hier vorgestellten Arbeitsergebnisse nicht möglich. Zum einen gab es nicht die Möglichkeit, eine Reihe ähnlicher Projekte, insbesondere im industriellen Umfeld durchzuführen, zum anderen wurde SUPrA und insbesondere die Werkzeugunterstützung parallel zum praktischen Einsatz weiterentwickelt. Deshalb gab es zwischen den verschiedenen beschriebenen Projekten teilweise deutliche Unterschiede in der Reife von SUPrA, sodass sie nicht direkt vergleichbar sind.

12.2 Fallbeispiel 1 – MeDIC

MeDIC ist ein unternehmensweites Metrikmanagementsystem zur Verwaltung und Auswertung von Prozess- und Produktmetriken bei der Generali Deutschland Informatik Services GmbH (GDIS). Entwickelt wird MeDIC im Rahmen eines Kooperationsprojektes zwischen der GDIS und dem SWC. MeDIC besteht aus vier Komponenten:

- **Metrikinformationssystem:** Das Metrikinformationssystem ist eine Erfahrungsdatenbank. Es enthält Definitionen relevanter Metriken und Anleitungen zu deren Interpretation.
- **Metrikkonfigurationssystem:** Das Metrikkonfigurationssystem dient zur Erzeugung einer projektspezifischen Messstrategie durch Auswahl und Anpassung von vorgefertigten Standardmetriken.
- **Messinfrastruktur:** Die Messinfrastruktur implementiert eine technische Infrastruktur zur Aggregation und Weiterverarbeitung von Messwerten verschiedener Messwerkzeuge.
- **Metrikdashboard:** Das Metrikdashboard bietet eine rollenspezifisch anpassbare Präsentation von Messergebnissen.

12.2.1 Betrachtete Fragestellungen

Im MeDIC-Projekt sollte der Einsatz von ProDUCE als Modellierungsmethode erprobt werden. In der Analyse wurde aufgrund der fehlenden Reife auf den Einsatz von OpenUMF verzichtet. Wegen der fehlenden technischen Unterstützung wurden auch die SUPrA-Metamodelle nicht eingesetzt. Allerdings wurden ähnliche Notationen zur Spezifikation verwendet.

Bei der Bewertung der Fallstudie sollten entsprechend die relevanten Aspekte von SUPrA untersucht werden. Es gilt also die folgenden Fragen zu beantworten:

- Ist eine Kombination von Use Cases und Nutzerschnittstellenprototypen sinnvoll?
- Ist ProDUCE zur Analyse der Anforderungen an MeDIC geeignet?
- Ist die verwendete Werkzeugumgebung geeignet?

12.2.2 Ablauf der Anforderungsanalyse

Die Entwicklung von MeDIC unterliegt einem featuregetriebenen Lebenszyklus. Die verschiedenen Komponenten von MeDIC wurden in mehreren Releases sukzessive entwickelt. Die Anforderungen wurden dabei nach der ProDUCE-Methode in mehreren Iterationen analysiert. Das Analyseteam bestand jeweils aus 1-3 Domänenexperten des Kunden (GDIS) sowie einem Analysten des Softwareherstellers (SWC). Zusätzlich waren jeweils 2-3 studentische Entwickler des Softwareherstellers beteiligt.

Benutzer hinzufügen

Ziel	Benutzer zu dem Projekt hinzufügen.	
Verbindlichkeit	kann	Für den ersten Piloten sind immer alle Benutzer allen Projekten zugeordnet, daher wird dieser Use Case nicht implementiert, sollte aber vorgesehen werden. Die Benutzer werden für diese Pilotierung von den Metrikkoordinatoren angelegt.
Akteure	Metrik Kunde	
Screens	12, im Erfolgsfall Screen 12a, im Fehlerfall Screen 12b	
Vorbedingung	Der Benutzer existiert im GDIS LDAP	
Standardablauf	1. V-Key des Benutzers eingeben 2. *System*: Benutzer im LDAP suchen 3. Gefundenen Benutzer auswählen 4. *System*: Den Benutzer dem Projekt hinzufügen 5. {include Rolle eines Benutzers bearbeiten}	
Alternativablauf	7. Alternative (Benutzer nicht gefunden) a. *System*: Hinweis ausgeben und abbrechen.	

Abbildung 12.1: MeDIC – Use Case Konzeptskizze

- **Produktabgrenzung:** Zum Start eines jeden Releases wurden zunächst in einer Planungssitzung der Umfang der im nächsten Release zu entwickelnden Funktionalität in Form einer kurzen Use Case Liste bzw. eines Use Case Diagramms festgelegt. Zusätzlich wurden im Anschluss Projektstakeholder und ihre Ziele modelliert. Weiterhin wurden einzelne Wireframes entwickelt.

- **Konzeptmodellierung:** In der Konzeptmodellierung wurden die Anforderungen iterativ und inkrementell entwickelt. Hierzu wurden Use Case Konzeptskizzen und Wireframes bzw. Storyboards entwickelt. Dabei gab es in MeDIC einen besonders starken Fokus auf die Intentionen verschiedener Nutzergruppen und die Gestaltung der Nutzerschnittstelle. Abbildung 12.1 stellt hier ein typisches Beispiel einer Use Case Dokumentation dar. Zur Qualitätssicherung der Anforderungen wurden regelmäßig Adhoc-Reviews mit den Softwareentwicklern und Metrikexperten des Kunden durchgeführt. Zu jedem Review wurden Use-Case-basierte Nutzerschittstellenprototypen eingesetzt. Dazu wurden manuell digitale Prototypen mit Powerpoint oder speziellen GUI-Prototypingwerkzeugen entworfen und mit den Use Case Abläufen annotiert. Ein Ausschnitt eines Powerpoint Prototypen für den Use Case `Benutzer hinzufügen` ist in Abbildung 12.2 dargestellt.

- **Detailmodellierung:** In der Detailmodellierung wurden lediglich wenige komplexe Abläufe zu detaillierten Use Case Beschreibungen ausspezifiziert und in interaktiven digitalen Prototypen evaluiert. Zusätzlich wurden codierte Prototypen in der Zieltechnologie JEE entwickelt.

12.2.3 Einsatz von SUPrA

Einsatz von Prototypen:

- **Produktabgrenzung:** In der Produktabgrenzung wurden nur wenige Prototypen verwendet. Ausschließlich zur *Identifikation von Benutzungskonzepten* wurden einige wenige Wireframes entworfen.

- **Konzeptmodellierung:** In der Konzeptmodellierung wurden zunächst Use Case Normalabläufe beschrieben. Anschließend wurden aus diesen manuell digitale Prototypen abgeleitet, die in regelmäßigen Adhoc-Reviews zur *Validierung der Ablaufbeschreibungen* und zur *Erprobung von Darstellungsformen* verwendet wurden.

- **Detailmodellierung:** In der Detailmodellierung wurden codierte Prototypen zur *Validierung der Ablaufbeschreibungen* und des *globalen Sytemverhaltens* eingesetzt. Außerdem wurde der Prototyp für *Benutzbarkeitsanalysen* eingesetzt. Zusätzlich wurden Teile des Prototyps für die evolutionäre Entwicklung des Zielsystems verwendet.

Eingesetzte Notationen:

- **Use Case Modell:** Zur Beschreibung von Use Cases wurde ein flussorientiertes Use Case Template verwendet. Use Case Kurzbeschreibungen wurden anhand einer Stakeholderanalyse identifiziert. Anschließend wur-

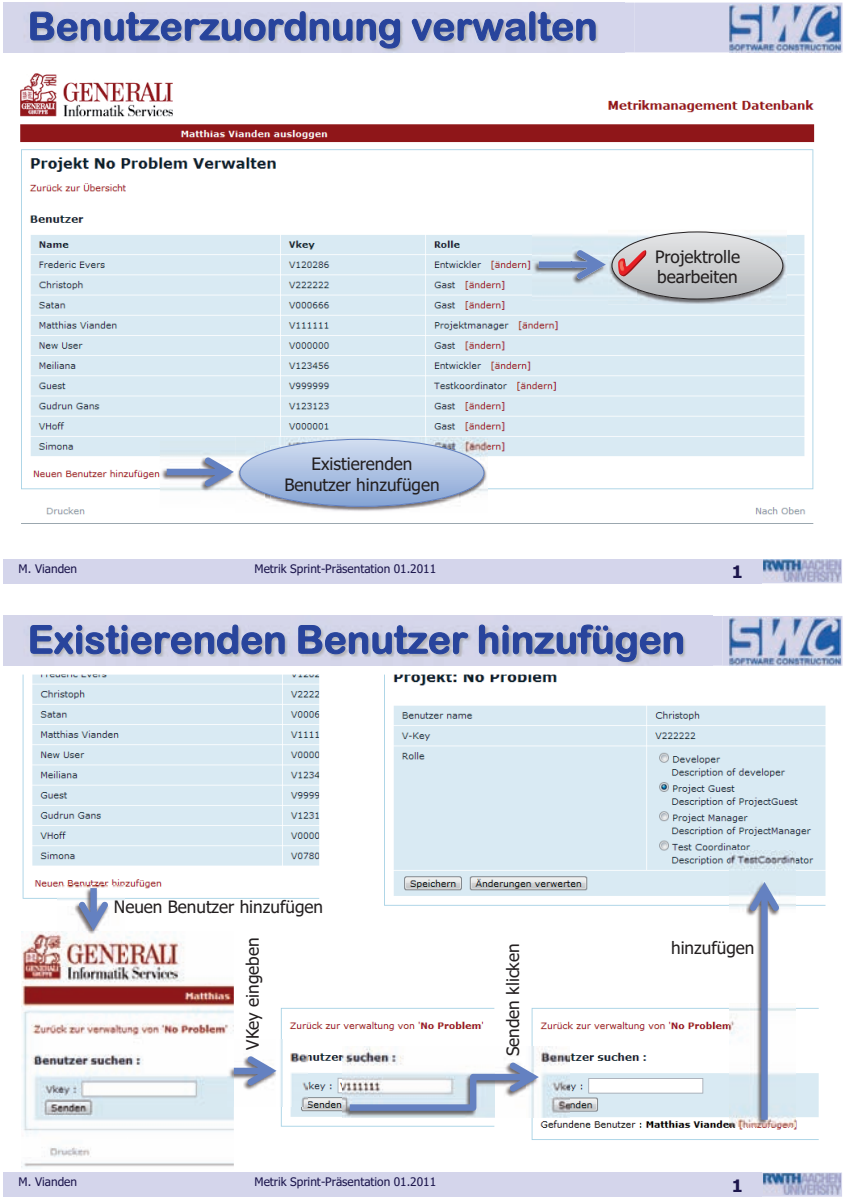


Abbildung 12.2: MeDIC – Powerpoint Prototyp

den sie in der Konzeptmodellierung zu knappen Use Case Konzeptskizzen weiterentwickelt. Eine Spezifikation von Detailbeschreibungen wurde nur bei komplexen Abläufen erstellt. Auf eine formalisierte Darstellung wurde verzichtet.

- **Domänenmodell:** Das Domänenmodell wurde in Form eines Klassendiagramms dokumentiert. Es spielte in der Analyse nur eine untergeordnete Rolle. Insbesondere wurde es in den Adhoc-Reviews nicht verwendet. In der Detailmodellierung wurde zusätzlich aus dem Klassendiagramm die Persistenzschicht des Prototyps generiert.

- **Nutzerschnittstellenmodell:** Die Nutzerschnittstelle wurde in der Konzeptmodellierung mit verschiedenen UI-Prototyping-Werkzeugen beschrieben. Sie wurde in diesen Phasen mehrfach angepasst und sukzessive verfeinert. In der Detailmodellierung wurde manuell eine Implementierung abgeleitet.

- **Systemfunktionsmodell:** Ein Systemfunktionsmodell wurde nicht erstellt. Stattdessen wurden die Systemfunktionen in der Detailmodellierungsphase direkt in den codierten Prototypen implementiert.

- **Weitere Notationen:** Zusätzlich zu den beschriebenen Notationen wurde eine Notation zur Stakeholderanalyse verwendet. In einem Word Template wurden Stakeholder und deren Ziele modelliert.

Eingesetzte Werkzeuge: Use Cases und Stakeholder wurden in Word Textdokumenten spezifiziert. Zusätzlich wurden teilweise Aktivitätsdiagramme entwickelt. Zur Beschreibung der Domäne wurden Klassendiagramme im Rational Software Architect [rsa] entworfen. Beim Prototyping der Nutzerschnittstelle kamen Powerpoint sowie verschiedene Spezialwerkzeuge zum GUI-Prototyping, wie beispielsweise Pencil [pen], zum Einsatz. Der codierte Prototyp, eine JEE Applikation, wurde in Eclipse entwickelt. Die Persistenzschicht konnte hierbei teilweise automatisch aus dem Klassendiagramm generiert werden.

12.2.4 Diskussion der Fallstudie

Konnten die Anforderungen mit den gewählten Notationen beschrieben werden? Die gewählten Notationen waren gut geeignet, um die Anforderungen an MeDIC zu modellieren. Alle funktionalen Anforderungen konnten strukturiert modelliert werden. Zusätzlich zu den im SUPrA Ansatz vorgeschlagenen Notationen wurde eine Stakeholder-Ziel-Modellierungsnotation verwendet. Diese half dabei, die Use Cases zu gruppieren und strukturiert zu analysieren.

Die verwendeten Notationen waren für alle Projektbeteiligten leicht zugänglich.

Insbesondere die Kombination von Use Cases und Nutzerschnittstellenprototypen ermöglichte es den Projektbeteiligten, sich schnell in die modellierten Systemaspekte einzuarbeiten und diese zu evaluieren.

Allerdings gab es teilweise Probleme mit der Use Case Technik. Es zeigte sich, dass flussorientierte Use Cases zur Beschreibung von CRUD-Funktionaliäten unnötig komplex sind. Für diese Funktionen wurden stattdessen ausschließlich Use Case Kurzbeschreibungen eingesetzt, die mit einer Skizze der Nutzerschnittstelle angereichert wurden.

Zusätzlich war die Einbettung von Use Cases und Nutzerschnittstelle zu gering. Für die Entwicklung der Nutzerschnittstellenprototypen mussten die Funktionsbeschreibungen der Use Cases manuell in die Prototypingnotation überführt werden.

War ProDUCE zur Analyse geeignet? Zusammenfassend ist die Einschätzung der ProDUCE Methode im MeDIC-Team durchweg positiv. Insbesondere die Kombination von Use Cases und Nutzerschnittstellenprototypen wird von allen befragten Projektbeteiligten als sehr nützlich bewertet, da sie den Adhoc-Reviews, die gleichzeitig die primäre Quelle für neue Anforderungen waren, Struktur gab.

MeDIC ist aufgrund erheblicher Probleme mit der Anforderungsanalyse im ersten Release von einem normalen Use-Case-zentrierten Ansatz auf die ProDUCE Methode umgestellt worden. Bevor ProDUCE eingesetzt wurde, gab es massive Probleme mit der Qualitätssicherung von Anforderungen. Bei Präsentationen von Softwareinkrementen wurden regelmäßig Fehler festgestellt, die auf missverstandene Anforderungen oder implizite Annahmen in den textuellen Use Cases zurückzuführen waren.

Alle Projektbeteiligten berichten, dass sich die Anzahl von Änderungswünschen, die auf Anforderungsfehler zurückzuführen sind, seit der Einführung von ProDUCE deutlich verringert hat. Zusätzlich berichten Projektbeteiligte, dass sie die Adhoc-Reviews als deutlich zielgerichteter empfinden.

Allerdings muss erwähnt werden, dass Inkonsistenzen zwischen Use Cases und Prototypen aufgrund der fehlenden Werkzeugunterstützung nur mit sehr viel Disziplin und deutlichem Mehraufwand vermieden werden konnten.

War die Werkzeugumgebung geeignet? Insgesamt war die verwendete Werkzeugumgebung nicht gut geeignet. Die fehlende Integration zwischen den verschiedenen Werkzeugen für unterschiedliche Modellierungsperspektiven machte viel manuellen Änderungsaufwand nötig. Insbesondere gab es keine Möglichkeit, die Use Cases mit den Nutzerschnittstellenprototypen zu resynchronisieren. Deshalb wurden Änderungen an den Nutzerschnittstellenprototypen nicht in die Use Cases nachgepflegt. Im MeDIC Projekt war diese Schwachstelle der

Werkzeugumgebung nicht problematisch, da die Use Cases nicht im Prozess weiterverwendet wurden. In anderen Projekten jedoch, in denen die Use Cases Vertragsbestandteil sind oder für Tätigkeiten wie Testen verwendet werden, wäre ProDUCE mit der verwendeten Werkzeugumgebung nicht praktikabel.

12.3 Fallbeispiel 2 – TCA-GUI

Das TCA-Gui-Projekt ist ein Kooperationsprojekt zwischen der Kisters AG und der Forschungsgruppe Softwarekonstruktion der RWTH Aachen. Im TCA-Gui-Projekt soll eine generische, eclipse-basierte Plattform für die Entwicklung verschiedener einsatzszenario-spezifischer Klienten für ein Zeitreihenmanagementsystem entwickelt werden. Ziel ist es, verschiedene bestehende Teillösungen und Altsysteme in einer gemeinsamen Plattform zu integrieren. Aus dieser sollen sich dann schnell spezielle Applikationen für Kundenprojekte ableiten lassen. TCA-Gui soll eine Vielzahl verschiedener graphischer und formularorientierter Werkzeuge zur Verarbeitung von Zeitreihen enthalten. Einige wichtige Funktionen sind:

- Das Ändern von Zeitreihenstammdaten
- Das Visualisieren von Zeitreihen
- Das Rollenmanagement
- Die Reporterzeugung
- Die Konfiguration von Serverinfrastrukturen
- Die Aggregation von Zeitreihendaten

Ein besonderes Augenmerk liegt hier auf einer generischen Kommunikationsinfrastruktur, mit der verschiedene graphische Werkzeuge und funktionale Komponenten integriert werden können.

Die in diesem Abschnitt beschriebene Fallstudie bezieht sich im Wesentlichen auf die 8-monatige Phase der Masterarbeit von Herrn Pholkit Manakitiwiphat. In dieser Phase wurde eine detaillierte Anforderungsanalyse durchgeführt. Außerdem wurde die allgemeine Konzeption der Plattform entwickelt und in weiten Teilen implementiert.

12.3.1 Betrachtete Fragestellungen

Im TCA-GUI-Projekt soll der Einsatz von SUPrA als Anforderungsansatz erprobt werden. In der Analyse wurde die ProDUCE Methode verwendet. Alle

Analyseartefakte wurden konform zu SUPrA modelliert. Da im TCA-GUI Projekt evolutionäres Prototyping eingesetzt werden sollte, wurden die Prototypen direkt für die Zielarchitektur Eclipse entworfen. Hierzu wurde WindowBuilder [Win], ein Spezialwerkzeug zum Rapid-Prototyping von Eclipse Nutzerschnittstellen, eingesetzt.

Bei der Bewertung der Fallstudie sollen nun die relevanten Aspekte von SUPrA untersucht werden. Es gilt also die folgenden Fragen zu beantworten:

- Ist ProDUCE zur Analyse der Anforderungen an TCA-GUI geeignet?
- Können die Anforderungen an TCA-GUI mit SUPrA modelliert werden?
- Ist die Werkzeugumgebung geeignet?

Ablauf der Anforderungsanalyse: Das TCA-Gui Projekt begann im Oktober 2011 mit einem Kernteam von zwei Experten aus der Entwicklungsabteilung des Kisters Zeitreihenmanagementsystems TCM (Time Series Managment System) sowie dem Masterstudenten Pholkit Manakitiwiphat.

- **Produktabgrenzung:** Gestartet wurde TCA-Gui mit einem Kickoff Meeting, in dem die relevanten Altsysteme präsentiert wurden. Außerdem wurden die wichtigsten Kernfunktionalitäten identifiziert. Anschließend wurden aus den Altsystemen die zentralen Use Cases des Systems destilliert.
- **Konzeptmodellierung:** Auf der Basis der Altsysteme wurden zunächst die Kernfunktionalitäten der wichtigsten Altsysteme mit textuellen Use Cases redokumentiert. Anschließend wurden Nutzerschnittstellenprototypen für diese Use Cases entworfen. Hier wurden alle zentralen Nutzerschnittstellen noch einmal nach den Nutzerzielen neu entworfen. Von dieser Maßnahme versprach man sich, Unzulänglichkeiten in der Nutzerinteraktion der Altsysteme beseitigen zu können. Zur Modellierung wurden Papierprototypen verwendet. Parallel wurde eine technische Machbarkeitsstudie durchgeführt. Hier wurde untersucht, wie Eclipse Standardmechanismen eingesetzt werden können, um die unterschiedlichen Werkzeuge von TCA-GUI zu integrieren. Eine Datenanalyse wurde nicht durchgeführt, da vom verwendeten Zeitreihensystem ein Datenmodell vorgegeben war.
- **Detailmodellierung:** Zur Detailmodellierung wurden codierte Prototypen eingesetzt, um eine mögliche Kommunikationsinfrastruktur zu evaluieren. Diese wurden im Sinne des evolutionären Prototypings zu einem Pilotsystem weiterverwendet. Dieser Ansatz bot sich im Projekt an, da bereits eine partielle Implementierung einer eclipse-basierten Plattform existierte.

12.3.2 Einsatz von SUPrA

Einsatz von Prototypen:

- **Produktabgrenzung:** Auf die Modellierung von Wireframes wurde wegen der existierenden Altsysteme verzichtet. Teilweise wurden jedoch Screenshots von Altsystemen zur Visualisierung von Lösungsideen verwendet.
- **Konzeptmodellierung:** In der Konzeptmodellierung wurden Wireframes bzw. Storyboards für die zentralen Use Cases entworfen. Sie dienten zur *Spezifikation und Validierung der Use Case Ablaufbeschreibungen.* Weiterhin wurden sie verwendet, um eine durchgängige Nutzerschnittstelle zu garantieren.
- **Detailmodellierung:** In der Detailmodellierung wurde ein codierter Prototyp entworfen. Er diente zur *Validierung der Ablaufbeschreibungen* und des *globalen Sytemverhaltens.* Insbesondere wurde er aber verwendet, um die *technische Realisierbarkeit* der Lösungsideen zu demonstrieren und die *Benutzbarkeit der Lösung zu bewerten.*

Eingesetzte Notationen:

- **Use Case Modell:** In der Produktabgrenzung wurden Use Case Diagramme entworfen. Use Cases wurde konform zum UDM in einem speziellen Word-Template spezifiziert. Use Case Kurzbeschreibungen wurden anhand der Analyse von Altsystemen in der Produktabgrenzung entwickelt. Diese wurden in der Konzeptmodellierung zu Use Case Konzeptskizzen weiterentwickelt. Auf eine Spezifikation von Detailbeschreibungen oder eine formalisierte Darstellung wurde verzichtet.
- **Domänenmodell:** Ein Domänenmodell wurde nicht entwickelt, da es vom Zeitreihenmanagementsystem vorgegeben war.
- **Nutzerschnittstellenmodell:** Die Nutzerschnittstelle in der Konzeptmodellierung wurde konform zum GDM mit Papierprototypen spezifiziert. In der Detailmodellierung bildeten diese Prototypen die Basis für die Implementierung eines codierten Prototyps.
- **Systemfunktionsmodell:** Ein Systemfunktionsmodell wurde nicht erstellt. Stattdessen wurden die Systemfunktionen in der Detailmodellierungsphase direkt in den codierten Prototyp implementiert.
- **Weitere Notationen:** Abgesehen von den vorgestellten Modellen wurden keine weiteren Notationen verwendet.

Eingesetzte Werkzeuge: Die Use Case Diagramme wurden mit Visio spezifiziert. Zur Beschreibung der Use Case Abläufe wurde ein Word-Template eingesetzt. Die Nutzerschnittstellen wurden zunächst auf Papier skizziert und anschließend mit WindowBuilder [Win] in eine Eclipse-Implementierung überführt. Hieraus wurde anschließend der codierte Prototyp –eine Eclipse Applikation– entwickelt. Das Datenmodell wurde als Bibliothek importiert.

12.3.3 Diskussion der Fallstudie

Konnten die Anforderungen mit UCSM modelliert werden? Die verwendeten Notationen waren für die Modellierung der Anforderungen sehr gut geeignet. Alle Anforderungen konnten klar in Use Cases strukturiert werden. Die Nutzerschnittstellenprototypen halfen bei der Kommunikation und dabei, eine durchgängige Nutzerschnittstelle zu entwerfen.

War ProDUCE zur Analyse geeignet? Die Erfahrungen aus dem TCA-Gui-Projekt sind insgesamt positiv. Die rigorose Anforderungsanalyse mit ProDUCE ermöglichte es, die Konzeption des Systems anhand von klaren Zielvorgaben zu treiben. Weiterhin waren die angereicherten Use Cases ein sehr gutes Kommunikationsmittel. Sie ermöglichten es Herrn Manakitiwiphat, sich trotz einiger Sprachprobleme schnell in die komplexe technische Infrastruktur der TSM Applikationslandschaft einzuarbeiten und eine fundierte Analyse der Anforderungen durchzuführen und zu dokumentieren.

Durch das Prototyping der Nutzerschnittstellen war es zudem möglich, die Konvergenz der einzelnen Seiten der Nutzerschnittstelle deutlich zu erhöhen und Redundanzen in der Nutzerschnittstelle zu reduzieren.

War die Werkzeugumgebung geeignet? Insgesamt war die Werkzeugumgebung für TCA-GUI gut geeignet. Die fehlende Integration zwischen Use Case Diagramm, Use Case Beschreibungen und Papierprototypen der Nutzerschnittstelle war im beschriebenen Projekt kein Problem, da die geforderten Abläufe schon zu Beginn des Projekts relativ klar waren. Deshalb gab es nur wenige Änderungen an den Use Cases. Allerdings berichten die Projektbeteiligten, dass die fehlende Integration für Projekte mit mehr Änderungen potentiell problematisch ist.

Die Infrastruktur für die Entwicklung codierter Prototypen war sehr gut geeignet. Alle Aspekte der Prototypen konnten in der evolutionären Entwicklung weiterverwendet werden.

12.4 Fallbeispiel 3 – CTSherpa

CTSherpa ist ein geplantes, intelligentes Patientenprozessierungssystem für die radiologische Abteilung des Universitätsklinikums Aachen (UKA). In der Fallstudie betrachten wir ausschließlich eine im Oktober 2011 gestartete fünf-monatige Vorstudie, in der die zentralen Anforderungen an das System erhoben und dokumentiert wurden.

CTSherpa ist ein System zur Steuerung der Patientenprozessierung in der radiologischen Abteilung. Grob besteht CTSherpa zwei Teilen: dem CT-Informationssystem und dem CT-Prozessierungssystem.

- Das CT-Informationssystem verbindet Informationen aus anderen medizinischen Informationssystemen wie z.B. Medico oder iSite [med]. Es bietet eine Reihe spezieller Sichten für unterschiedliche Aufgaben im CT-Umfeld, wie z.B. Scanprotokollfestlegung, Scandurchführung oder Befundung.
- Das CT-Prozessierungssystem dient der Planung des Ablaufes bei der Patientenprozessierung. Es plant die tägliche Auslastung von CTs auf Basis von Patientenparametern. Diese sind z.B. die Dringlichkeit der Untersuchung, bestehende Erkrankungen des Patienten oder Sprechzeiten der behandelnden Ärzte sowie spezielle Behandlungsparameter. Hierzu zählen die Behandlungsdauer, die Patientenlage oder der Vorlauf der Untersuchung. Hierbei können alle Planungsparameter von den Nutzern (Ärzte, MTA) in Form von Regeln dynamisch angepasst werden. Weiterhin kann die Qualität der Regeln automatisch evaluiert werden. Dazu soll sich das CT-Planungssystem selbst beobachten und die Qualität seiner Planungsvorschläge vermessen.

12.4.1 Betrachtete Fragestellungen

Im CTSherpa-Projekt sollte der Einsatz von SUPrA als Anforderungsansatz erprobt werden. In der Analyse wurden sowohl die UCSM Notationen, die OpenUMF Werkzeugumgebung als auch das ProDUCE eingesetzt.

Bei der Bewertung der Fallstudie sollen entsprechend auch alle Aspekte von SUPrA untersucht werden. Es gilt also die folgenden Fragen zu beantworten:

- Können die Anforderungen an CTSherpa mit UCSM modelliert werden?
- Ist ProDUCE zur Analyse der Anforderungen an CTSherpa geeignet?
- Ist OpenUMF geeignet, die Entwicklung und Dokumentation der Anforderungen zu unterstützen?

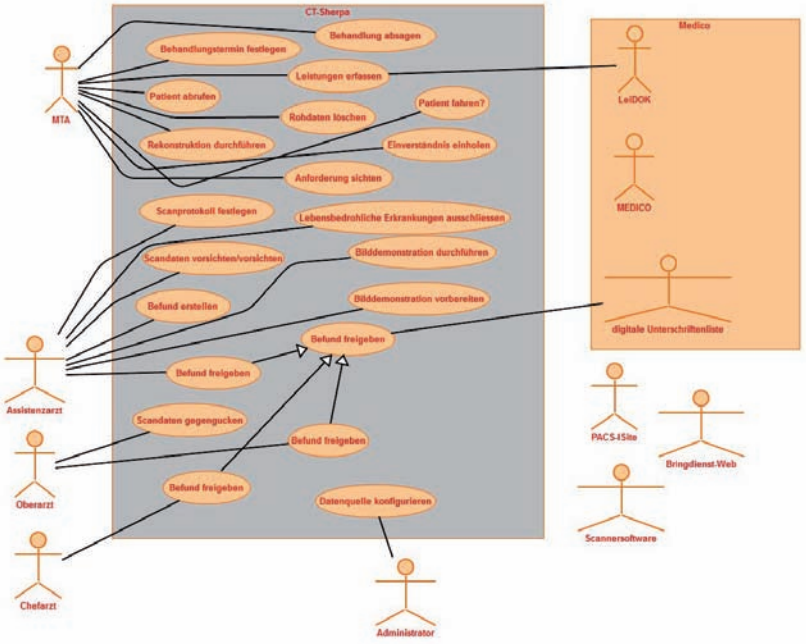


Abbildung 12.3: CTSherpa – Use Case Diagramm (Ausschnitt)

12.4.2 Ablauf der Anforderungsanalyse

CTSherpa wurde nach der ProDUCE-Methode in einer Iteration analysiert. Beteiligt war ein Kernteam aus 5-8 Domänenexperten des Kunden (UKA), die verschiedene Nutzerrollen repräsentierten. Diese spezifizierten die fachlichen Anforderungen. Die methodische Leitung der Anforderungsanalyse und die Anforderungsdokumentation übernahm ein Analyst des Softwareherstellers (SWC). Konkret wurde die Analyse folgendermaßen ausgestaltet:

- **Produktabgrenzung:** Gestartet wurde das Projekt mit einem gemeinsamen Kickoff-Meeting aller Projektbeteiligter, in dem die Projektvision präsentiert wurde. Anschließend wurden die Geschäftsabläufe in Geschäftsworkflows fixiert. Aus diesen wurden die zentralen Akteure und Use Cases abgeleitet. Zusätzlich wurden Bedienkonzepte für alle Nutzerrollen festgelegt. Parallel wurde ein Domänenglossar aufgebaut.

- **Konzeptmodellierung:** In der Konzeptmodellierung wurden die konkreten Anforderungen unterschiedlicher Nutzerrollen in Einzel- oder Kleigruppeninterviews on the Job spezifiziert. Dazu wurden Abläufe der Use Cases anhand des Geschäftsworkflows und der in der Produktabgrenzung

entwickelten Wireframes spezifiziert. Parallel wurden Benutzbarkeitsanforderungen dokumentiert. Aus dem Domänenglossar und einer Analyse der vorhandenen Informationen in existierenden Fremdsystemen wurde zusätzlich ein Domänenmodell spezifiziert. Einen Überblick über die Ergebnisse eines typischen Interviews gibt Abbildung 12.4.

- **Detailmodellierung:** Zur Evaluierung der Use Case Abläufe und des Domänenemodells wurde manuell ein codierter Prototyp entwickelt. An diesem wurde untersucht, ob die fachlichen Abläufe korrekt abgebildet wurden und ob das Datenmodell die Informationsbedürfnisse der Nutzer bedienen kann. Zusätzlich wurde der codierte Prototyp eingesetzt, um verschiedene Interaktionsformen auszuprobieren.

12.4.3 Einsatz von SUPrA

Einsatz von Prototypen:

- **Produktabgrenzung:** In der Produktabgrenzung wurden Wireframes bei der Spezifikation der Geschäftsworkflows zur *Analyse von Geschäftsprozessen* eingesetzt. Außerdem wurden sie zur *Identifikation von Benutzungskonzepten* für die unterschiedlichen Nutzerrollen verwendet.

- **Konzeptmodellierung:** In der Konzeptmodellierung wurden zunächst Screenflows aus Geschäftsworkflows und den in der Produktabgrenzung entwickelten Wireframeprototypen abgeleitet. Diese wurden zur *Spezifikation der Use Case Normalabläufe* eingesetzt. Anschließend wurden die Screenflows sukzessive zu Papierprototypen verfeinert, mit denen *Alternativabläufe spezifiziert und validiert* wurden.

- **Detailmodellierung:** In der Detailmodellierung wurde ein codierter Prototyp zur *Validierung der Ablaufbeschreibungen* und des *globalen Systemverhaltens* eingesetzt. Es wurde untersucht, ob die fachlichen Abläufe korrekt abgebildet wurden und ob das Datenmodell die Informationsbedürfnisse der Nutzer bedienen kann. Außerdem wurde der Prototyp für *Benutzbarkeitsanalysen* eingesetzt. Hier wurde insbesondere die Interaktion mit einer Kalenderkomponente und eine geeignete Notation für die Spezifikation von fachlichen Analyseregeln untersucht.

Eingesetzte Notationen:

- **Use Case Modell:** Zur Beschreibung von Use Cases wurde das UDM verwendet. Use Case Kurzbeschreibungen wurden anhand der Geschäftsworkflows in der Produktabgrenzung entwickelt. Diese wurden in der Konzeptmodellierung zu knappen Use Case Konzeptskizzen weiterentwickelt. Auf eine Spezifikation von Detailbeschreibungen oder eine formalisierte

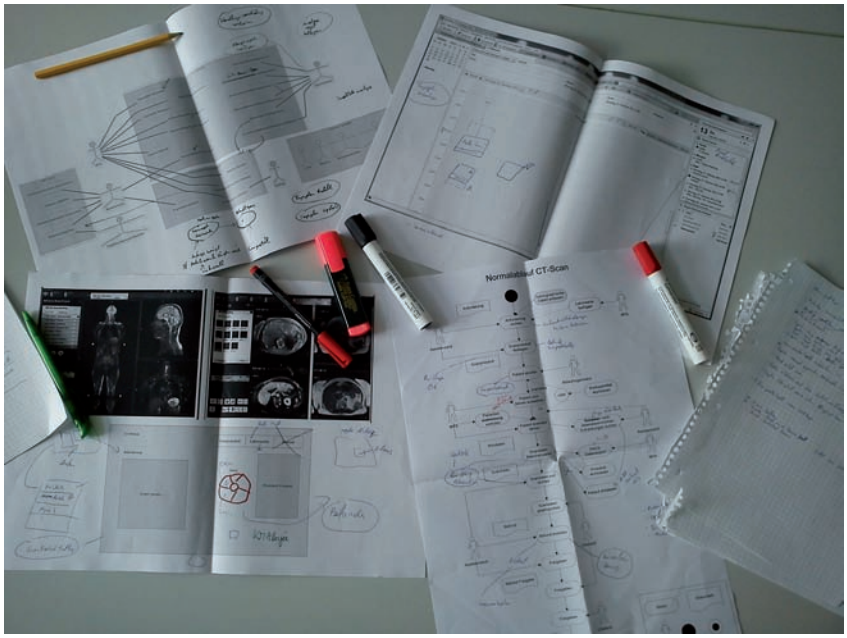


Abbildung 12.4: CTSherpa – Interviewdokumente

Darstellung wurde verzichtet.

- **Domänenmodell:** Ab der Produktabgrenzung wurden alle wichtigen Begriffe in einem Domänenenglossar gesammelt. Das Domänenmodell wurde in der Konzeptmodellierung in ein DCM-Modell überführt. Dieses wurde aber nicht für Interviews verwendet. In der Detailmodellierung wurde aus dem DCM-Modell eine Implementierung in Eclipse-EMF generiert.

- **Nutzerschnittstellenmodell:** Die Nutzerschnittstelle wurde komplett im GDM spezifiziert. Bereits in der Konzeptmodellierung wurden einfache Wireframes bzw. Screenshots ähnlicher Nutzerschnittstellen anderer Werkzeuge spezifiziert. Aus dem GDM Modell wurden in der Konzeptmodellierung Papierprototypen abgeleitet. Änderungen an diesen Prototypen wurden ebenfalls im GDM realisiert. In der Detailmodellierung bildete das GDM die Grundlage für die Implementierung des codierten Prototypen.

- **Systemfunktionsmodell:** Ein Systemfunktionsmodell wurde nicht erstellt. Stattdessen wurden die Systemfunktionen in der Detailmodellierungsphase direkt in den codierten Prototypen implementiert.

- **Weitere Notationen:** Zusätzlich zu den UCSM Modellen wurde eine Geschäftsworkflownotation eingesetzt. Diese enthielt Aktionen, Arbeitsergebnisse und Rollen. Außerdem wurde eine leichtgewichtige Notation zur

Beschreibung von nicht-funktionalen Anforderungen auf Basis des DCM verwendet. Diese enthielt Anforderungsarten, Anforderungen und Beziehungen zu UCSM Elementen.

Eingesetzte Werkzeuge: Die Geschäftsworkflows wurden mit Visio spezifiziert. Zur Spezifikation aller anderen Anforderungen wurde OpenUMF eingesetzt. Neben den vorgestellten Werkzeugen zur Entwicklung von UCSM Modellen wurde eine textuelle DSL für die Spezifikation der nicht-funktionalen Anforderungen verwendet.

Auch der codierte Prototyp, eine Eclipse Applikation, wurde mit Hilfe von OpenUMF entwickelt. Das Datenmodell konnte hierbei aus dem GDM generiert werden.

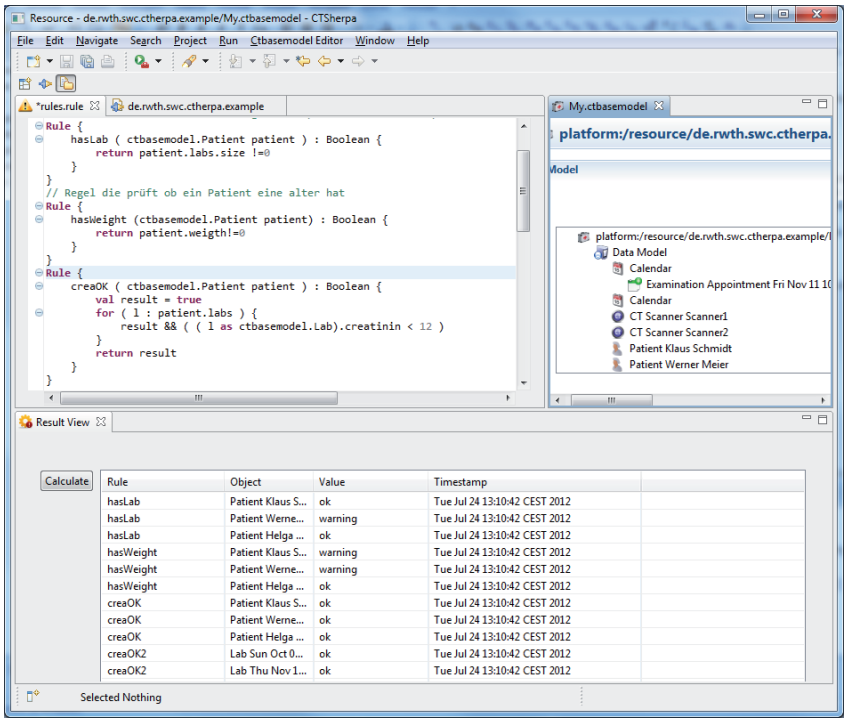


Abbildung 12.5: CTSherpa – Codierter Prototyp für die Regelspezifikation

12.4.4 Diskussion der Fallstudie

Konnten die Anforderungen mit UCSM modelliert werden? Insgesamt war UCSM sehr gut geeignet, um die Anforderungen im CTSherpa Projekt

aufzunehmen. Alle funktionalen Anforderungen konnten strukturiert abgebildet werden. Zusätzlich war eine White-Box Erweiterung von UCSM nötig, um nicht-funktionale Anforderungen insbesondere an die Benutzbarkeit zu spezifizieren, die im UCSM nur über verschiedene Use Cases verschmiert darstellbar gewesen wären. Außerdem wurde eine spezielle Notation für Geschäftsworkflows verwendet.

Die Notationen waren für alle Projektbeteiligten zugänglich. Insbesondere die graphische Modellierung von Nutzerschnittstellen wurde von den Domänenexperten sehr gut angenommen. Anforderungen konnten klar kommuniziert und validiert werden.

War ProDUCE zur Analyse geeignet? Das Vorgehen war insgesamt gut geeignet. Die Analyse aus Workflowsicht war für Domänenexperten intuitiv und wurde gut akzeptiert. Der Einsatz von Prototypen insbesondere in frühen Phasen der Analyse war ein voller Erfolg. Obwohl keiner der Domänenexperten Erfahrung in der Spezifikation von Anforderungen hatte, konnten sie die Anforderungen auf Basis einfacher Wireframes präzise spezifizieren. Auch die Papierprototypen wurden als sehr nützlich empfunden, da sie den Domänenexperten halfen, spezifizierte Abläufe zu validieren.

Auch die Erfahrungen mit dem Einsatz von codierten Prototypen ist positiv. Mit dem codierten Prototyp konnte die Tragfähigkeit des Domänenmodells gezeigt werden. Außerdem wurden wichtige Erkenntnisse für die Benutzerinteraktion einer technischen Umsetzung gewonnen. Insbesondere die Analyse geeigneter Interaktionsformen für die Spezifikation von fachlichen Regeln wäre ohne einen codierten Prototypen nicht möglich gewesen.

Positiv war zudem die gute Beteiligung der Domänenexperten. Dies ist aus unserer Sicht auf das schnelle Feedback im Vorgehen zurückzuführen, da Domänenexperten ihren Input in den Prototypen schnell wiederfanden.

Allerdings wies das angewandte Vorgehen auch zwei Schwächen auf. Da der Fokus des Ansatzes primär auf der Interaktion lag, wurde die Datenmodellierung nur wenig betrachtet. Die Qualität des Datenmodells konnte aber durch den Einsatz des codierten Prototypen gesichert werden. Außerdem gab es Probleme bei der Spezifikation der Systeminterna des CT-Prozessierungssystems. Grund hierfür war zum einen, dass das Vorgehen die Systeminterna nicht explizit adressiert hat, zum anderen war die Auswahl der Projektbeteiligten nicht optimal. Viele Domänenexperten konnten die geplante Funktionalität nicht spezifizieren, da das CT-Prozessierungssystems für ihre berufliche Rolle nicht wichtig ist.

War die Werkzeugumgebung geeignet? Insgesamt war OpenUMF als Werkzeugumgebung gut geeignet. UCSM Anforderungen konnten strukturiert spezifiziert werden und auch die nicht-funktionalen Anforderungen ließen sich

leicht integrieren. Wireframes konnten effektiv entwickelt werden und Papierprototypen ließen sich effizient erzeugen und anpassen.

Allerdings muss angemerkt werden, dass die Simulationsumgebung in OpenUMF nicht geeignet war, einen codierten Prototypen zu erzeugen. Im CTSherpa Projekt lag ein starker Fokus auf der Evaluierung spezieller Interaktionsformen, wie Drag&Drop, die in PEF nicht unterstützt werden. Deshalb musste manuell ein codierter Prototyp entwickelt werden.

12.5 Weitere Fallbeispiele

Abgesehen von den bereits vorgestellten Projekten ist SUPrA in einer Reihe weiterer Projekte eingesetzt worden. Alle diese Projekte sind nach der ProDUCE Methode entwickelt worden. Die Werkzeuginfrastruktur wurde in unterschiedlichem Umfang eingesetzt. Im Folgenden werden die Projekte kurz skizziert. Tabelle 12.1 gibt einen Überblick über den Einsatz von SUPrA. Abschließend wird eine zusammenfassende Einschätzung gegeben.

- **WfDesigner AdminInterface:**

  Ziel: Entwicklung einer speziellen Kommunikationsansicht, mit der ein eclipse-basiertes Workflowentwicklungswerkzeug mit einer server-seitigen Workflowausführungsmaschine kommunizieren kann.

  Projektdauer: 5 Monate

- **XAM:**

  Ziel: Entwicklung einer JEE Applikation zur Verwaltung von Klausur- und Übungsaufgaben

  Projektdauer: 1 Jahr, noch laufend

- **ViPER-Methodtoolbox:**

  Ziel: Entwicklung eines graphischen Werkzeugs zur Beschreibung und Anpassung von Entwicklungsprozessdokumentationen

  Projektdauer: 5 Monate

- **OpenUMF:**

  Ziel: Portierung und Weiterentwickung von ViPER Nautilus ([HLNW09]) auf das UCSM Metamodell

  Projektdauer: ca. 6 Monate

Projekt	OpenUMF	Wireframes/ Storyboards	digitale Prototypen	codierte Prototypen
WfDesigner	nein	ja	ja	ja
XAM	nein	ja	ja	ja
ViPER	ja	ja	ja	nein
OpenUMF	ja	ja	ja	ja

Tabelle 12.1: Eingesetzte Prototypen – Fallbeispiele

Erfahrungen: Insgesamt war der Einsatz von SUPrA in allen Projekten hilfreich. Es zeigt sich, dass die Kombination von Use Cases und Nutzerschnittstellenprototypen sehr gut als Kommunikationsmittel geeignet ist. Insbesondere in Diskussionen mit Projektbeteiligten, die nicht regelmäßig in die Entwicklung involviert sind, wie Kunden oder Managern, erleichtern die Nutzerschnittstellenprototypen den Zugang deutlich. Weiterhin konnte beobachtet werden, dass der Entwicklungsaufwand der Prototypen deutlich geringer ist, wenn OpenUMF zur Prototypgenerierung eingesetzt wird.

Allerdings sind auch einige Schwächen von SUPrA aufgedeckt worden. Globale Geschäftsregeln können nur schwer in die übrige Dokumentation integriert werden. Für CRUD-Funktionen sind flussorientierte Beschreibungen unnötig kompliziert. SUPrA bietet keine methodische Unterstützung für die Modellierung von vielen Arten nicht-funktionaler Anforderungen, wie z.B. Performanceanforderungen. Wie diese mit SUPrA integriert werden können, ist unklar, da sie in keinem der untersuchten Projekte eine Rolle spielten.

12.5.1 Erfahrungen in studentischen Projekten

SUPrA wurde 2012 zum zweiten Mal erfolgreich als Anforderungsentwicklungsmethodik im Softwarepraktikum des Bachelorstudiengangs Informatik der RWTH Aachen eingesetzt. Im Sommersemester 2012 wurde zusätzlich OpenUMF als Entwicklungswerkzeug verwendet. Die Erfahrungen sind hier durchweg positiv. Die Qualität der Anforderungsdokumente ist in Hinblick auf die Richtigkeit und Vollständigkeit deutlich besser als in den Praktika der vorhergehenden Jahre, in denen keine Analysemethode vorgegeben wurde. Weiterhin zeigen Befragungen, die mit den Studenten durchgeführt wurden, dass ihre Zufriedenheit und ihre Konfidenz in die entwickelte Spezifikation deutlich besser ist. Zugegebenermaßen sind diese Ergebnisse nicht besonders überraschend, da sich die entwickelten Systeme sehr gut für den Einsatz von Prototypingtechniken eignen. Dennoch kann der Einsatz von SUPrA auch in diesem Fall als Erfolg gesehen werden. Weiterhin demonstrieren dieses studentischen Projekte die Simplizität des Ansatzes, da Studierende im dritten Semester ohne praktische Erfahrung in der Anforderungsanalyse nach einer kurzen Einführung (ca. 30 Minuten) in der Lage waren, qualitativ hochwertige Spezifikationen zu erzeugen.

13 Evaluierung

Im Folgenden werden die Sprache UCSM und die Prototypingumgebung OpenUMF bewertet. Dazu werden verschiedene wichtige Aspekte einzeln, der Reihe nach beleuchtet.

13.1 Bewertung der Sprache UCSM

In diesem Abschnitt werden die Vor- und Nachteile der Spezifikationsnotation UCSM erörtert. Die Diskussion ist dabei anhand der in Kapitel 5 aufgestellten Forderungen strukturiert.

13.1.1 Ausdrucksstärke

UCSM bietet, wie in Abschnitt 8.1 beschrieben, eine integrierte Prototypingnotation für alle vier funktionalen Sichten der Use-Case-zentrierten Analyse. Im Folgenden wird diskutiert, ob für diese Sichten eine ausreichende Ausdrucksstärke gegeben ist und ob die Beziehungen zwischen den verschiedenen Sichten adäquat dargestellt werden können.

Use Cases: In UDM können alle in der UML2 Spezifikation festgelegten Use Case Konzepte dargestellt werden. UDM hat Notationselemente für Use Cases, Akteure und das modellierte System. Zusätzlich können diese in Pakete gruppiert werden. Alle in der UML2 beschriebenen Beziehungen können modelliert werden. UDM enthält Beschreibungselemente für Generalisierungs-, Inklusions- und Erweiterungsbeziehungen. Außerdem können Assoziationsbeziehungen zwischen Akteuren und Use Cases beschrieben werden. Hierbei kann im Unterschied zur UML2 der primäre Akteur explizit gekennzeichnet werden. Die Ausdrucksstärke für den strukturellen Teil des UDM ist also gegeben.

Im Beschreibungsteil des UDM können Use Case Ablaufbeschreibungen unterschiedlicher Detaillierungsgrade beschrieben werden. Es ist möglich, textuelle Kurzbeschreibungen anzugeben oder den Ablauf flussorientiert zu beschreiben. In der flussorientierten Sicht werden alle von Bittner und Spence [BS02] vorgeschlagenen Modellierungselemente unterstützt. Neben einfachen Aktionen, die mit einer formalen Beschreibung unterlegt werden, können Inklusions- und Erweiterungsbeziehungen sowohl zwischen Flüssen desselben Use Cases als auch

zwischen Flüssen verschiedener Use Cases beschrieben werden. Zusätzlich gibt es die Möglichkeit, Schleifen und Ausnahmefälle zu beschreiben. Mit UDM kann der Ablauf eines Use Cases komplett und konform zur Semantik von UML2 Aktivitätsmodellen beschrieben werden. Die Ausdrucksstärke des Beschreibungsteils kann also ebenfalls als hinreichend angesehen werden.

Nutzerschnittstelle: Das GDM bietet eine kompakte Beschreibungsform für graphische Nutzerschnittstellen. Es bietet die Möglichkeit, verschiedene Bildschirmseiten und deren Zusammenhang zu beschreiben. Die einzelnen Bildschirmseiten können aus frei positionierbaren Widgets zusammengestellt werden. Zusätzlich können diese Widgets mit dem UDM verbunden werden. So wird ihr Verhalten modelliert. Die Ausdrucksstärke des GDM kann also im Hinblick auf unseren Anwendungsfall als hinreichend betrachtet werden.

Allerdings ist das GDM auf die Beschreibung von formularorientierten Nutzerschnittstellen, wie sie in typischen Desktop- oder Webanwendungen vorkommen, beschränkt. Es bietet nicht die Konzepte an, um mobile Applikationen, graphische Editoren oder rein textuelle Schnittstellen zu modellieren. Diese Interaktionsformen spielen in typischen Informationssystemen nur eine untergeordnete Rolle, deshalb wurde zu Gunsten der Simplizität des GDM auf sie verzichtet. Weiterhin enthält das GDM nur die gängigsten Widgets, die benötigt werden, um die Funktionsweise der Nutzerschnittstelle zu demonstrieren. Allgemein enthalten Graphikbibliotheken wie z.B. PrimeFaces [pri] eine deutlich größere Zahl von Widgets. Diese unterscheiden sich aber hauptsächlich in der Darstellungsform –beispielsweise gibt es in Primefaces ca. 10 verschiedene Arten von Clickbuttons–, nicht aber in der Art der Nutzerinteraktion. Deshalb wurde auf diese Widgets verzichtet.

Domänenmodell: Das DCM enthält alle Konzepte der Begriffsmodellierung. Es ermöglicht die Beschreibung von Entitäten, deren Attribute und Umgangsformen. Es ermöglicht die Beschreibung von Generalisierungsbeziehungen und Referenzen zwischen Entitäten. Somit ist die Ausdrucksstärke hinreichend.

Auf die Modellierung von Interfaces wird im DCM verzichtet, da dieses Konzept technisch motiviert ist. Stattdessen ist es möglich, dass eine Entität eine Spezialisierung mehrerer anderer Entitäten ist. Dies wird bei der Implementierung von Domänenmodellen oft mit Interfaces umgesetzt.

Systemfunktionen: Das SFM bietet die Möglichkeit, getypte Systemvariablen und Systemfunktionen zu beschreiben. Dies reicht für unseren Einsatzzweck, das Rapid Prototyping, aus. Deshalb ist die Ausdrucksstärke des SFM ausreichend.

Im Allgemeinen fehlt die Möglichkeit, die Verteilung der Systemfunktionen auf Architektureinheiten zu beschreiben. Zwar gibt es ein Namespacekonzept, aber

dieses dient lediglich zur Verbesserung der Übersichtlichkeit des SFM. Das SFM ist für die Beschreibung von Systemarchitekturen nicht geeignet.

Zusammenhang: Im UCSM ist es möglich, die Beziehungen zwischen Use Case und Systemfunktionen, zwischen Use Cases und der Nutzerschnittstelle und zwischen Systemfunktionen und dem Domänenmodell zu beschreiben. Dies reicht aus, um in unserem Anwendungzweck das Verhalten eines modellierten Systems komplett zu simulieren. Somit ist die geforderte Ausdrucksstärke gegeben. Außerdem können zusätzlich beliebige Beziehungen zwischen Elementen modelliert werden, da alle Elemente vom Konzept REElement des BCM abgeleitet sind.

13.1.2 Simplizität

Das Qualitätskriterium der Simplizität sagt aus, wie leicht eine Sprache zu erlernen und zu benutzen ist.

13.1.2.1 Erlernbarkeit & Benutzbarkeit

Die Erlernbarkeit einer Sprache hängt davon ab, wie schwer es Nutzern fällt, sich ein mentales Modell der Sprache aufzubauen. Diese mentalen Modelle entstehen evolutionär und hängen von den Erfahrungen und dem technischen Hintergrund des Nutzers ab [Nor87]. Die Sprache UCSM verwendet ausschließlich Notationen und Konzepte, die bereits aus anderen Modellierungssprachen bekannt sind und für viele Softwareentwickler zum Handwerkszeug gehören. Im Folgenden soll die Erlernbarkeit von UCSM systematisch vermessen werden. Dazu verwenden wir die von Green & Petre [GP96] vorgeschlagenen kognitiven Dimensionen, die allgemeine Entwurfsprinzipien für textuelle oder graphische Sprachen darstellen.

Abstraktionsgrad: Die verschiedenen Abstraktionsgrade, auf denen in UCSM modelliert werden kann, sind seit vielen Jahren in anderen Modellierungsnotationen akzeptiert. Somit kann der Abstraktionsgrad als geeignet angesehen werden.

Nähe der Abbildung: Die Nähe der Abbildung entspricht der der einzelnen verwendeten Submodelle. Diese beschreiben alle etablierte Modellierungssichten. Ihre Abbildungsnähe kann als gut angesehen werden.

Konsistenz: UCSM verwendet eine relativ große Zahl von verschiedenen Elementen, insbesondere in verschiedenen Submodellen. Innerhalb der Submodelle

sind sich die einzelnen Elemente jedoch relativ ähnlich. So sind beispielsweise alle Arten von Widgets mit einem Namen, einer Größe und einer relativen Position ausgestattet. Insgesamt kann also die Konsistenz als gut angesehen werden.

Diffusität: UCSM enthält ein explizites Modellierungselement für alle zentralen Konzepte. Deshalb ist die Diffusität als sehr gering einzustufen.

Fehleranfälligkeit: Fehler betreffen in UCSM vor allem Beziehungen zwischen Elementen verschiedener Submodelle. Diese können konstruktiv nur schwer vermieden werden. Deshalb enthält OpenUMF automatische Validierungsregeln, die versuchen, diese Fehler aufzudecken.

Schwierige mentale Operationen: UCSM enthält an drei Stellen schwierige mentale Operationen. Dies sind Beziehungen von Flüssen in Ausnahmesituationen, Provider für Views und die Implementierung von Systemfunktionen. Alle diese Probleme sind jedoch auf komplexe fachliche Beziehungen zurückzuführen. Deshalb können sie aus unserer Sicht nicht vereinfacht werden, ohne Präzision zu verlieren.

Versteckte Abhängigkeiten: Generell gib es eine Reihe von Abhängigkeiten zwischen den verschiedenen Modellierungssichten von UCSM. Diese werden aber weitestgehend explizit modelliert und sind deshalb leicht erkennbar. Zusätzlich werden implizite Beziehungen zwischen Elementen, z.B. zwischen einem Widget und einem Attribut einer Entität, durch spezielle Sichten in der Werkzeugumgebung sichtbar gemacht.

Frühzeitige Festlegung: Während der Anforderungsanalyse ist es normal, Annahmen und Festlegungen zu treffen, die im späteren Verlauf der Analyse revidiert werden müssen. Diese Problematik lässt sich nicht konstruktiv umgehen und sie betrifft alle Submodelle des UCSM. OpenUMF versucht, einigen verbreiteten Problemen mit speziellen Werkzeugen zu begegnen. So bietet OpenUMF beispielsweise Refactorings zur Restrukturierung von Use Case Flows an.

Schrittweise Auswertung: Schrittweise Auswertbarkeit war eine der Hauptforderungen an die Sprache. Sie ist durch den generativen Prototypingansatz gegeben und wird in der Bewertung des Prototypingansatzes diskutiert.

Erkennbarkeit von Rollen: Jedem Sprachelement ist genau eine Rolle zugewiesen. Deshalb ist die Erkennbarkeit von Rollen gut.

Sekundäre Notation: Der in Abschnitt 7.3 beschriebene Erweiterungsmechanismus ermöglicht es, Kommentare zu beliebigen Notationselementen hinzuzufügen.

Viskosität: Die meisten Änderungen im Modell können leicht durchgeführt werden, da sie lediglich lokale Effekte haben. Ausnahmen hierzu bilden Änderungen am Ablauf von Use Cases, diese können Änderungen an der Nutzerschnittstelle nach sich ziehen. Diese Effekte sind jedoch der Komplexität der modellierten Probleme geschuldet. Deshalb kann auch für diese Fälle die Viskosität als angemessen betrachtet werden. Zusätzlich sind in der Werkzeugumgebung Regeln implementiert, die den Nutzer bei derartigen Änderungen unterstützen sollen.

Sichtbarkeit/Übersichtlichkeit: In UCSM sind die einzelnen Sichten klar identifizierbar. Eine Navigation zwischen zusammengehörigen Elementen verschiedener Sichten wird durch die Werkzeugunterstützung angeboten. Die Übersichtlichkeit der Sprache ist also gut.

13.1.3 Knappheit

UCSM hat explizite Modellierungselemente für alle wichtigen Konzepte. Fast alle atomaren Zusammenhänge können mit einem einzelnen Modellierungselement dargestellt werden. Somit ist die Knappheit der Sprache gegeben.

13.1.4 Strukturiertheit

UCSM ist in vier Submodelle aufgeteilt, die jeweils einen funktionalen Aspekt des modellierten Systems kapseln. Zusätzlich gibt es innerhalb aller Sichten die Möglichkeit, Elemente in Substrukturen zu gruppieren. Die Strukturiertheit der Sprache ist also gegeben. Einzig eine durchgängige Strukturierung anhand fachlicher Aspekte wird nicht unterstützt.

13.1.5 Unterstützung inkrementeller Entwicklung

UCSM bietet die Möglichkeit, Spezifikationselemente sowohl textuell als auch formal zu spezifizieren. Zusätzlich können textuelle Informationen sukzessive in eine formale Darstellung überführt werden. Eine inkrementelle Verfeinerung der Spezifikation wird also durch UCSM unterstützt.

13.1.6 Erweiterbarkeit

Der in Abschnitt 7.3 diskutierte Erweiterungsmechansimus bietet die Möglichkeit, beliebigen Elementen zusätzliche Informationen zuzuordnen. Außerdem können neue Modellierungsperspektiven durch Ableitung vom BCM realisiert werden. Die Erweiterbarkeit der Sprache ist also gut.

13.1.7 Formalität

Aus UCSM Modellen können vollautomatisch ausführbare Nutzerschittstellenprototypen abgeleitet werden. Die Formalität der Sprache ist also ausreichend.

13.2 Bewertung der Software OpenUMF

Dieser Abschnitt diskutiert die innere und äußere Qualität der OpenUMF Werkzeugumgebung. Dazu werden die Qualitätsmerkmale des ISO/IEC Standards 9126 [Int05] jeweils auf die Komponenten der Werkzeugsuite angewandt, für die sie relevant sind.

13.2.1 Funktionalität

13.2.1.1 Eignung

Das Qualitätsmerkmal Eignung beschreibt, inwieweit die Software spezifizierte Anforderungen umsetzt. Deshalb wird hier die Umsetzung der Forderungen an die Werkzeugsuite (vgl. 5.3.2) kurz diskutiert.

Unterstützung verschiedener Prototyparten: OpenUMF unterstützt verschiedene Arten von Nutzerschnittstellenprototypen (vgl. Kap. 10.2). Es bietet Werkzeugunterstützung für Wireframes, Storyboards, digitale- und codierte Prototypen. Außerdem bietet es die Möglichkeit, nicht-graphische Verhaltensprototypen zu erzeugen.

Integration mit textueller Dokumentation: OpenUMF basiert auf dem UCSM, einem Metamodell für Use-Case-zentrierte Spezifikationen. Aus diesem Metamodell können automatisch textuelle Spezifikationsdokumente in unterschiedlichen Formaten (doc, pdf, xsl, html) abgeleitet werden.

Präzise Simulation des Verhaltens: OpenUMF basiert auf der in Abschnitt 8.1 beschriebenen formalen Ausführungssemantik. Die dargestellten Simulationsläufe sind also präzise.

Ausführung unvollständiger Spezifikationen: OpenUMF ermöglicht es, wie in Abschnitt 8.3 skizziert, ausführbare Prototypen aus unvollständigen UCSM-Modellen zu erzeugen. Diese können auch in der Simulationsumgebung ausgeführt werden.

Adressatenspezifische Anpassung: OpenUMF enthält neben der allgemeinen Simulationsansicht eine Reihe weiterer Ansichten, die Zusatzinformationen zu Simulationsläufen geben können. Diese Ansichten können beliebig kombiniert werden. So kann OpenUMF nach den Bedürfnissen der Nutzer angepasst werden.

Integration mit anderen Modellierungswerkzeugen: OpenUMF ist in eine Eclipse Werkzeugsuite eingebettet. So können OpenUMF Werkzeuge leicht mit anderen Werkzeugen, wie z.B. einer Metrikkomponente [Kla09], oder einem Testcasegenerator [Kil10] integriert werden.

Simulation des globalen Systemverhaltens: OpenUMF ist in der Lage, ein vereinigtes Simulationsmodell aus allen Use Cases eines UCSM zu generieren. Durch die Einführung des Simulationsrahmens wird es so möglich, mehrere Use Cases nacheinander auszuführen und das globale Systemverhalten zu simulieren.

Wiederverwendung & Analyse von Simulationsdaten: OpenUMF enthält mit dem TraceRecorder eine Komponente, die Simulationsdaten aufzeichnet. Diese können abgespielt, zur Analyse der Simulation oder als Eingabe z.B. für die Erzeugung von Tests verwendet werden.

13.2.1.2 Genauigkeit

Die Genauigkeit der Simulation wird maßgeblich durch den Detaillierungsgrad der Use-Case-zentrierten Spezifikation determiniert. Wenn die Simulation auf Basis eines formalen UCSM generiert wird, ist sie entsprechend der formalen Spezifikation von UCSM (vgl. 8.1) genau. Bei nicht formalen UCSM ist die Genauigkeit von den Nutzereingaben, die Systemverhalten emulieren, abhängig.

13.2.1.3 Interoperabilität

OpenUMF verwendet weitestgehend Standardmechanismen von Eclipse. UCSM Modelle können deshalb in Eclipse Sichten, wie der Vorschau oder der Properties-Ansicht angezeigt werden. Durch Implementierung des Selektionsmechanismus können sich externe Werkzeuge über Aktionen in OpenUMF informieren lassen. Weiterhin ist es möglich, zusätzlich Funktionen über die Erweiterungsmechanismen von Eclipse in OpenUMF einzufügen.

Zusätzlich bietet OpenUMF die Möglichkeit, UML2 Modelle zu exportieren. So können UCSM Modelle auch in externen Werkzeugen verarbeitet werden.

13.2.1.4 Sicherheit

Datensicherheit und Zugriffskontrolle spielen in OpenUMF keine Rolle. Soll der Zugriff auf bestimmte Daten kontrolliert werden, muss dies durch Einsatz eines Versionsverwaltungssystems oder durch Rechtemanagement auf Dateisystemebene realisiert werden.

13.2.2 Zuverlässigkeit

13.2.2.1 Reife

Die Use Case Spezifikationsumgebung wird bereits seit 2008 als Teil der ViPER Werkzeugumgebung [Nys09] eingesetzt. Andere Teile der OpenUMF Werkzeugumgebung sind neueren Datums. Dennoch sind alle Teile von OpenUMF in verschiedenen Fallstudien eingesetzt worden. Aus diesen Fallstudien wurde eine Vielzahl von Verbesserungsvorschlägen abgeleitet, die größtenteils bereits in die Werkzeugumgebung eingepflegt wurden. Insgesamt ist OpenUMF allerdings ein Forschungsprototyp. Insbesondere die Nutzerdokumentation ist unzureichend.

13.2.2.2 Fehlertoleranz

Bei der Fehlertoleranz müssen die drei Komponenten Spezifikationsumgebung, Prototypgenerator und Bewertungsumgebung einzeln betrachtet werden.

Die Spezifikationsumgebung ist robust gegen falsche Eingaben. Es ist möglich, Elemente auf ungültige Weise miteinander zu verbinden und so beispielsweise eine unendliche Schleife im Ereignisfluss zu erzeugen. Die Spezifikationsumgebung versucht, solche ungültigen Modelle durch Validierungsregeln zu identifizieren. Sie ist aber in der Lage, diese zu persistieren und weiterzubearbeiten.

Der Prototypgenerator ist ebenfalls gegen illegale Eingaben robust. Er ist in

der Lage, ein ausführbares Modell für UCSM-Modelle mit illegalen Schleifen oder fehlenden Informationen zu erzeugen. Allerdings entsprechen diese Modelle nicht unbedingt den Erwartungen des Prototypnutzers. So wird beispielsweise für einen fehlenden Rücksprunganker standardmäßig ein Aussprung aus der Interaktionssequenz des Use Cases generiert.

Die Ausführungsumgebung ist ebenfalls gegen falsche Eingaben weitestgehend robust. Syntaktisch inkorrekte Skripte und Eingaben, die einen unerwarteten Datentyp haben, werden ignoriert. Sollte ein Simulationslauf doch aufgrund eines Fehlers abgebrochen werden müssen, so kann er immer ab dem letzten gültigen Schritt wieder aufgenommen werden, da der Trace jeden Schritts persistiert wird.

13.2.2.3 Wiederherstellbarkeit

Alle Elemente der Werkzeugsuite speichern Daten persistent in xml-Dateien. Hierbei können in seltenen Fällen, z.B. bei Speicherüberläufen, Fehler auftreten. Diese können von OpenUMF nicht verhindert werden, da sie auf Probleme im unterliegenden Eclipse Rahmenwerk zurückzuführen sind. Allerdings enthält OpenUMF eine Anbindung an alle gängigen Versionsverwaltungssysteme und mehrere spezielle Änderungsverfolgungswerkzeuge aus der Eclipse-Werkzeuglandschaft, die in diesen Fällen Abhilfe schaffen können.

13.2.3 Verwendbarkeit

13.2.3.1 Verständlichkeit

Die Simulationsumgebung ist für alle Arten von Projektbeteiligten intuitiv zu bedienen, da sie über eine graphische Schnittstelle bedient wird, die der des entwickelten Zielsystems sehr ähnlich ist. Einzige Ausnahme hiervon ist die Spezifikation von Systemfunktionen während der Ausführung. Dies erfordert Programmierkenntinsse in Java. Deshalb sollten kompliziertere Skripte nur von Experten spezifiziert werden. Es zeigt sich jedoch, dass einfache Zuweisungen auch von Kunden nach kurzer Einweisung spezifiziert werden können.

Für die Spezifikationsumgebung gilt ähnliches. Die Spezifikationsdokumente sind leicht für alle Arten von Projektbeteiligten verständlich, da sie in einer textuellen Darstellungsform angezeigt werden können. Allerdings erfordert die Spezifikation von flussorientierten Use Cases eine eingehendere Beschäftigung mit dem UDM und die Spezifikation des SFM Java Progammierkenntnisse.

13.2.3.2 Erlernbarkeit

Die Benutzung von OpenUMF ist im Allgemeinen leicht zu erlernen. Die Modellierung der Perspektiven von UCSM ist für Nutzer, die bereits Erfahrungen in der Modellierung von Use-Case-zentrierten Anforderungen haben, leicht verständlich.

Die Benutzung der Simulationsumgebung ist für alle Projektbeteiligten leicht erlernbar, da sie im Wesentlichen auf der Interaktion mit graphischen Kontrollelementen basiert. Einzig die Spezifikation von Skripten erfordert Grundkenntnisse in der Programmierung.

13.2.3.3 Bedienbarkeit

Die Bedienbarkeit von OpenUMF ist im Allgemeinen gut. Bei der Spezifikation von UCSM Modellen helfen verschiedene Übersichtsansichten dabei, zwischen den verschiedenen Modellierungsperspektiven zu navigieren. Allerdings gibt es hier noch Verbesserungspotential. Höherwertige Refactoring- und Autokorrekturfunktionen könnten die Bedienbarkeit weiter steigern.

13.2.3.4 Attraktivität

In den durchgeführten Fallstudien gibt es durchweg positive Resonanz auf das Werkzeug. Allerdings wurden einige Verbesserungspotentiale identifiziert. Beispielsweise wird die Interaktion bei der Modellierung von Contexts im UCST kritisiert.

13.2.4 Effizienz

13.2.4.1 Zeitverhalten

Bei der Betrachtung des Zeitverhaltens ist vor allem der Prototypengenerator interessant. Die Spezifikationsumgebung wie auch die Bewertungsumgebung führen im Wesentlichen einfache Lese- und Schreiboperationen auf den zugehörigen Modellen aus. Ihr Laufzeitverhalten ist fast ausschließlich von den verwendeten Basistechnologien EMF und Graphiti abhängig.

Der Prototypengenerator basiert auf dem in Abschnitt 8.1.6 beschriebenen Algorithmus. Dieser kann mit einer Laufzeit von O(3n) über die Anzahl der enthaltenen Events abgeschätzt werden. Messungen zeigen, dass die Generierung von Prototypen selbst aus Modellen mit über 10000 Events im Bereich weniger Sekunden liegt.

13.2.4.2 Resourcenverbrauch

Im Hinblick auf den Resourcenverbrauch muss vor allem die Hauptspeicherauslastung betrachtet werden. Diese liegt bei Einsatz von der Spezifikationsumgebung, dem Prototypengenerator und der Simulationsumgebung relativ hoch (ca. 300 MB). Der Hauptteil dieser Last entfällt jedoch bei den untersuchten Modellen (ca. 50 UC) auf Basisfunktionalität aus dem Eclipserahmenwerk. Die Größe der Spezifikationsmodelle selbst spielt nur eine untergeordnete Rolle. Es gilt allerdings zu überprüfen, ob diese Eigenschaft auch für größere Modelle gilt.

13.2.5 Wartbarkeit

13.2.5.1 Analysierbarkeit & Änderbarkeit

Das OpenUMF Build System erhob regelmäßig automatisch verschiedene Codemetriken mit Hilfe des Werkzeugs Sonar [son]. Hiermit wurde zum einen verifiziert, dass die Abhängigkeiten der einzelnen Pakete von OpenUMF konform zur in Abschnitt 11 vorgestellten Architekturdefinition sind, zum anderen wurde regelmäßig eine Reihe von Quellcodemetriken und Regeln vermessen. Abbildung 13.1 zeigt eine Zusammenfassung der aktuellen Zahlen. Die große Zahl an Regelverstößen ist auf generierte Codefragmente zurückzuführen.

13.2.5.2 Testbarkeit & Stabilität

OpenUMF ist in ein automatisiertes Build- und Releasemanagementsystem eingebunden (vgl. Kap. 11). Dieses enthält zusätzlich eine automatisierte Regressionstestsuite, die die wichtigsten Funktionen der Werkzeugsuite testet. Sie beinhaltet sowohl funktionale als auch UI-Tests für die nicht-automatisch generierten Teile von OpenUMF.

13.2.6 Portabilität

13.2.6.1 Anpassbarkeit

Alle Komponenten der Werkzeugumgebung sind plattformunabhängig in Java und Eclipse implementiert. Die Struktur der Implementierung des UCSM ermöglicht es, leicht neue Modellierungselemente einzufügen (vgl. 7.3). Gleiches gilt auch für den Prototypengenerator, der auf einem elementspezifischen Strategiemuster aufbaut.

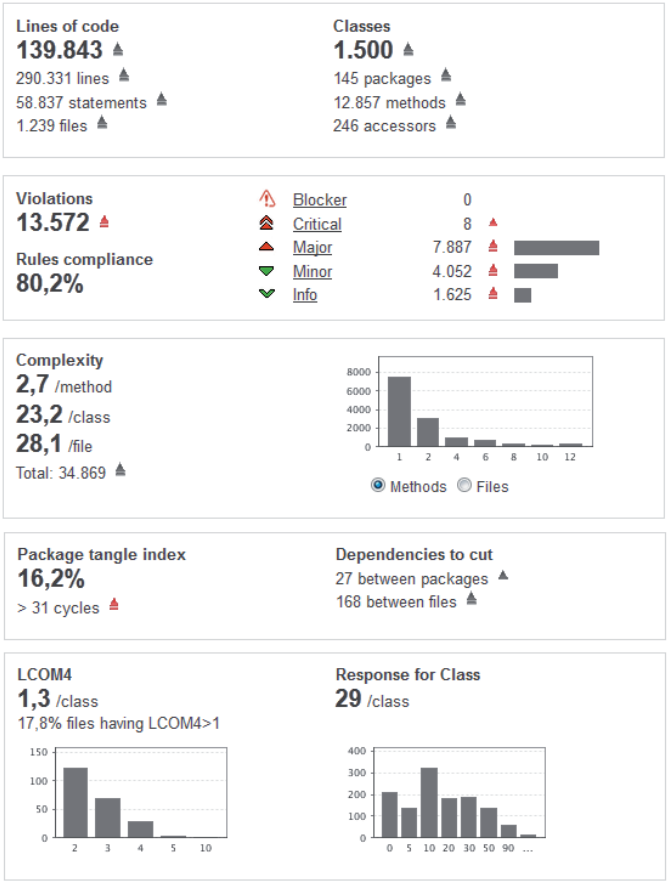


Abbildung 13.1: Codemetriken OpenUMF Sonar

13.2.6.2 Installierbarkeit

OpenUMF wird als lauffähige Eclipse RCP für die Betriebssysteme MS-Windows, Linux und MacOS ausgeliefert. Es ist ohne Installation nach einem Download sofort einsetzbar. Außerdem sind alle Komponenten einzeln als Installationspakete für die Eclipse Plattform 3.7 verfügbar. Sie können über den etablierten Updatemechanismus von Eclipse automatisch installiert und aktualisiert werden.

13.2.6.3 Koexistenz

OpenUMF bildet innerhalb von Eclipse Installationen eine abgeschlossene Komponente, bestehend aus mehreren Features. Sie hat abgesehen von Abhängigkeiten auf bestimmte Versionen der eingesetzten Basisfeatures keinerlei Seiteneffekte mit sonstigen Plugins des Eclipse Ökosystems.

13.2.6.4 Ersetzbarkeit

Die gesamte Werkzeugumgebung ist modular aus Plugins aufgebaut, die nur über schmale, definierte Schnittstellen miteinander kommunizieren. Zusätzlich sind alle Schnittstellen mit Interfaces gekapselt. Somit ist die Austauschbarkeit von Komponenten gewährleistet.

14 Zusammenfassung & Ausblick

Ausgangspunkt für diese Arbeit war die unzureichende Einbettung von Prototypingtechniken in die Use-Case-zentrierte Analyse. Gängige Ansätze bieten keine methodische und konzeptuelle Einbettung von Prototyping und Spezifikationsnotation. Sie eignen sich lediglich dazu, einmalig einen Prototypen aus einer vollständigen Use Case Spezifikation abzuleiten. Deshalb sind sie für einen Einsatz in einem inkrementellen und iterativen Vorgehen bei der Use-Case-zentrierten Analyse zu unflexibel, da Prototypen und Use Case Spezifikation bei Änderungen manuell angepasst werden müssen.

Diese Arbeit stellt Methoden, Notationen und Werkzeuge vor, welche die Einbettung von Prototypingtechniken in ein Use-Case-zentriertes Vorgehen verbessern. Die Spezifikationsnotation UCSM bietet die Möglichkeit, alle Perspektiven der Use-Case-zentrierten Analyse zu beschreiben und miteinander in Beziehung zu setzen. Zusätzlich kann aus UCSM Modellen automatisch ein ausführbarer Prototyp generiert werden. Dadurch entfällt die manuelle Anpassung von Prototypen nach Änderungen im Spezifikationsmodell. So können typische Übertragungsfehler vermieden werden. Der generative Ansatz ermöglicht ein schnelles Erzeugen von Prototypen in einem inkrementellen Vorgehen bei der Use Case Spezifikation.

Weiterhin wurde ein methodisches Vorgehen entwickelt, das beschreibt, wann welche Arten von Prototypen in einer Use-Case-zentrierten Anforderungsanalyse entwickelt werden sollten und wie diese zur Verbesserung der Analyse eingesetzt werden können.

Die entwickelten Sprachen und Methoden werden durch die Werkzeugsuite OpenUMF unterstützt. Zu dieser Werkzeugumgebung gehört eine eingebettete Entwicklungsumgebung für alle Submodelle des UCSM. Weiterhin enthält OpenUMF eine Referenzimplementierung des in Kapitel 8 vorgestellten Prototypgenerators und der in Kapitel 9 diskutierten Bewertungsumgebung.

Die vorgestellten Ansätze wurden in mehreren Fallstudien in industriellen Projekten und in der Forschung eingesetzt. SUPrA hat sich dabei in Use-Case-zentrierten Anforderungsanalysen als geeignet erwiesen. Es war möglich, schon ab frühen Phasen verschiedene Arten von Prototypen zur Validierung von UCSM Modellen einzusetzen und die UCSM Modelle auf Basis von Prototypen weiterzuentwickeln. Bei einem Teil der Fallstudien konnte bereits die Werkzeugumgebung erprobt werden. Allerdings müssen hier noch weitere Erfahrungen gesammelt werden. Insbesondere Fragen zur Ergonomie der Spezifikationsumgebung

– z.B. wo und wie die Systemfunktionen aus dem SFM mit den Use Case Flüssen des UDM verknüpft werden sollten – müssen noch einmal betrachtet werden.

14.1 Ausblick

Die in dieser Arbeit beschriebenen Ergebnisse stellen dar, wie sich Prototyping in ein iteratives, inkrementelles, phasen-orientiertes Use-Case-zentriertes Vorgehen einbetten lässt. Allerdings gibt es bei der Entwicklung und dem Einsatz von Use Case Prototypen eine Reihe offener Fragestellungen. Diese werden nun kurz vorgestellt. Zusätzlich werden einige Ideen und Vorbemerkungen zur Beantwortung dieser Fragen skizziert.

Mehrnutzersimulation: Derzeit konzentriert sich der UCPT Ansatz auf die Simulation von Abläufen mit einem einzelnen Benutzer. Hier könnte die Einsatzumgebung um die Möglichkeit erweitert werden, einen Simulationslauf mit mehreren gleichzeitig arbeitenden Nutzern auszuführen. Dazu müsste untersucht werden, wie in der ApplicationFacade zwischen privaten Variablen eines Nutzers und globalen Variablen des Systems, die zwischen allen Nutzern geteilt werden, unterschieden werden kann. Einige Vorüberlegungen dazu gibt Abschnitt 8.1.1. Zusätzlich muss untersucht werden, ob in einem Simulationslauf mit mehreren Nutzern spezielle Regeln für das Feuern von Transitionen festgelegt werden müssen, um das nebenläufige Verhalten des simulierten Systems korrekt abzubilden.

Spezifikation von nicht-funktionalen Anforderungen: Derzeit werden im UCPT Ansatz keinerlei nicht-funktionale Anforderungen, abgesehen vom Aufbau der Nutzerschnittstelle, betrachtet. Hier gilt es zu untersuchen, wie nicht-funktionale Anforderungen in der Spezifikationsnotation und vor allem in der Bewertungsumgebung unterstützt werden können. Interessant ist hier insbesondere das Laufzeitverhalten. Denkbar wäre es, die erwartete Dauer einer Aktion im Ereignisfluss, wie von Nyssen [Nys09] vorgeschlagen, abzuschätzen und dann die Dauer der Aktion in der Simulation beispielsweise aufgrund einer Verteilungsfunktion automatisch zu berechnen. Eine andere Möglichkeit wäre es, die Dauer von Schritten während der Simulation manuell von Prototypnutzern einstellen zu lassen.

Evolutionäres Prototyping: In UCPT werden zum Prototyping ausschließlich Wegwerfprototypen entwickelt. Hier wäre es nützlich, am Ende der Anforderungsanalyse die Möglichkeit anzubieten, einen Prototyp zu erzeugen, der sich zur Weiterentwicklung in ein Produktivsystem eignet. Dies wäre insbesondere bei der Modellierung von formalen UCSM Modellen sinnvoll, um den sehr

hohen Spezifikationsaufwand zu rechtfertigen. Hierzu müssen bestimmte Annahmen über die Architektur des Zielsystems gemacht werden. Es gilt also zu untersuchen, wie diese Annahmen im UCSM hinzugefügt werden können. Dazu gibt es mehrere Möglichkeiten. Denkbar wäre ein expliziter Modellierungsschritt, in dem die Systemfunktionen auf architekturelle Einheiten des Zielsystems gemappt werden. Nyssen [Nys09] beschreibt beispielsweise einen solchen Ansatz für komponentenbasierte, eingebettete Systeme. Alternativ könnte das UCSM Modell in ein spezielles Generatormodell für eine spezielle Zielarchitektur übersetzt werden. Einen ähnlichen Ansatz verfolgt beispielsweise Löwenthal [Löw11] für Klassendiagramme.

Verbreiterung der Zielsysteme: Das UCSM ist derzeit darauf ausgerichtet, die Nutzerschnittstelle von typischen Desktop-Applikationen darzustellen. Das GDM könnte für weitere Arten von Applikationstypen wie z.B. Applikationen für mobile Endgeräte oder eingebettete Systeme erweitert werden. Hier gibt es zwei Möglichkeiten. Entweder führt man eine Reihe spezieller GDM Versionen ein, die jeweils die Widgets für einen Applikationstyp enthalten, oder man erweitert das GDM um Informationen, mit denen man den Applikationstyp festlegen kann. Zusätzlich ist es in jedem Fall nötig, die Widgets für unterschiedliche Applikationstypen in der Bewertungsumgebung zu unterstützen. Dies lässt sich in der bestehenden Architektur leicht durch Implementierung spezieller RenderingStrategien (vgl. 9.3.3) erreichen.

Methodische Unterstützung: Im aktuellen Ansatz werden lediglich Vorschläge und Vorgaben zu Entwicklung und Einsatz von Prototypingtechniken in der Use-Case-zentrierten Analyse beschrieben. Um das Potential des Einsatzes von Prototypingtechniken ausschöpfen zu können, ist ein strukturiertes Vorgehen nötig, das zusätzlich die Modellierungsaktivitäten zur Entwicklung der UCSM Modelle klar festlegt. Hier muss untersucht werden, wie ein allgemeines Vorgehen für die prototyporientierte Use-Case-zentrierte Analyse aussehen muss und wie dieses für unterschiedliche Systemtypen angepasst werden kann. Außerdem wäre eine technische Unterstützung dieses Vorgehens in der Werkzeugsuite hilfreich. Einige Vorüberlegungen hierzu liefert Mussil [Mus11].

Literaturverzeichnis

[AAB06] Arnowitz, J., M. Arent und N. Berger: *Effective Prototyping for Software Makers*. Morgan Kaufmann Publishers Inc., 2006.

[AH93] Asur, Sujai und S. Hufnagel: *Taxonomy of rapid-prototyping methods and tools*. In: *[1993] Proceedings The Fourth International Workshop on Rapid System Prototyping*, Seiten 42–56. IEEE Comput. Soc. Press, 1993.

[AL00] Amyot, D und Luigi Logrippo: *Use Case Maps and LOTOS for the prototyping and validation of a mobile group call system*. Computer Communications, 23(12):1135–1157, Juli 2000.

[Ali05] Alisch, K.: *Gabler Wirtschaftslexikon*. Gabler, 2005.

[AM01] Armour, F. und G. Miller: *Advanced use case modeling: software systems*. Addison-Wesley object technology series. Addison-Wesley, 2001.

[AM04] Alexander, I. und Neil Maiden: *Scenarios, stories, use cases: through the systems development life-cycle*. John Wiley & Sons Inc, 2004.

[And94] Andriole, S.J.: *Fast, cheap requirements prototype, or else!* IEEE Software, 11(2):85–87, März 1994.

[AS02] Anda, Bente und Dag I. K. Sjøberg: *Towards an inspection technique for use case models*. In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering - SEKE '02*, Seite 127, New York, New York, USA, 2002. ACM Press.

[Bal96] Balzert, H.: *Lehrbuch der Software-Technik.: Software-Entwicklung*. Spektrum, Akad. Verl., 1996.

[BB01] Boehm, Barry und V.R. Basili: *Top 10 list [software development]*. IEEE Computer, 34(1):135–137, 2001.

[BD04] Bernárdez, Beatriz und Amador Durán: *Empirical Evaluation and Review of a Metrics – Based Approach for Use Case Verification **. Practice, 36(4):247–258, 2004.

[BP92] Bischofberger, W. und Gustav Pomberger: *Prototyping-oriented software development: concepts and tools*. Texts and monographs in computer science. Springer-Verlag, 1992.

[BS02] Bittner, Kurt und Ian Spence: *Use Case Modeling (Addison-Wesley Object Technology)*. Addison-Wesley Longman, Amsterdam, September 2002.

[BS09] Bomsdorf, Birgit und Daniel Sinnig: *Model-Based Specification and Validation of User Interface Requirements*. Human-Computer Interaction. New Trends, Seiten 587–596, 2009.

[Buh98] Buhr, R.J.A.: *Use case maps as architectural entities for complex systems*. IEEE Transactions on Software Engineering, 24(12):1131–1155, 1998.

[Cas] *CaseComplete Product Website*. `http://www.casecomplete.com/`.

[CL99] Constantine, Larry L. und Lucy A. D. Lockwood: *Software for use: a practical guide to the models and methods of usage-centered design*, Band 32. Addison-Wesley, 1999.

[CL01] Constantine, L.L. und L.A.D. Lockwood: *Structure and style in use cases for user interface design*. Object Modeling and User Interface Design. Addison-Wesley, Boston, 1(978):245–280, 2001.

[Coc00] Cockburn, Alistair: *Writing Effective Use Cases*. Addison-Wesley Professional, Oktober 2000.

[Con95] Constantine, Larry L.: *Essential modeling: use cases for user interfaces*. interactions, 2(2):34–46, April 1995.

[Con09] Constantine, Larry L: *Toward a Pragmatic Integration of Activity Theory with Usage-Centered Design*. In: Seffah, Ahmed, Jean Vanderdonckt und Michel C. Desmarais (Herausgeber): *Human-Centered Software Engineering*, Human–Computer Interaction Series, Seiten 27–51. Springer, London, 2009.

[Con12] Conrad, Steffen: *Ein Ansatz zum modellgetriebenen Test von EJB-basierten Informationssystemen*. Diplomathesis, RWTH Aachen University, 2012.

[Cri92] Crinnion, John: *Evolutionary Systems Development: A Practical Guide to the Use of Prototyping within a Structured Systems Methodology*. Perseus Publishing, 1992.

[CSW08] Clark, Tony, Paul Sammut und J. Willans: *Applied metamodelling: a foundation for language driven development*. Ceteva, Sheffield, 2008.

[Def87] Defence Science Board: *Report of the defense science board task force on military software*. Technischer Bericht, Defence Science Board, Washington DC, 1987.

[DeM78] DeMarco, T.: *Structured analysis and system specification*. Prentice-Hall software series. Yourdon, 1978.

[DH01] Damm, Werner und D. Harel: *LSCs: Breathing life into message sequence charts.* Formal Methods in System Design, 19(1):45–80, 2001.

[Dou99] Douglass, Bruce Powel: *ROPES : Rapid Object-Oriented Process for Embedded Systems.* Real-Time Systems, Seiten 1–40, 1999.

[DP04] Denger, Christian und Barbara Paech: *An integrated quality assurance approach for use case based requirements.* Rumpe, B. Hesse, W.(eds). Modellierung, 2004.

[Drö00] Dröschel, W.: *Das V-Modell 97: der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz.* Oldenbourg, 2000.

[dSP00] Silva, P.P. da und NW Paton: *User Interface Modelling with UML.* In: *In Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Representation*, Band 20, Seiten 203—-217. IOS Press, Juli 2000.

[DTK03] Dranidis, Dimitris, Kalliopi Tigka und Petros Kefalas: *Formal modelling of use cases with X-machines.* In: *Proc. 1st South East European Workshop on Formal Methods, SEEFM'03*, Seiten 72–83, Thessaloniki, Greece, 2003.

[Dum03] Dumke, R.: *Software engineering.* Vieweg Lehrbuch. Vieweg, 2003.

[Ebe08] Ebert, Christof: *Systematisches Requirements Engineering und Management - Anforderungen ermitteln, spezifizieren, analysieren und verwalten (2. Aufl.).* dpunkt.verlag, 2008.

[ecl] *Eclipse Project Website.* `http://www.eclipse.org/`.

[emf] *Eclipse-EMF Project Website.* `http://www.eclipse.org/modeling/emf/`.

[Flo84] Floyd, Christiane: *A systematic look at prototyping.* Approaches to prototyping, 1:1–18, 1984.

[Fon10] Fonteyn, Philipp: *Semantische Integration von Konzept- und Use Case Modellen.* Bachelorthesis, RWTH Aachen University, 2010.

[Gam09] Gamma, E.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series. Addison-Wesley, 2009.

[GB95] Gordon, V.S. und J.M. Bieman: *Rapid prototyping: lessons learned.* IEEE Software, 12(1):85–95, Januar 1995.

[Gem03] Gemino, Andrew: *Empirical comparisons of animation and narration in requirements validation.* Requirements Engineering, 9(3):153–168, November 2003.

[GF03] García, José Daniel und Rosa Filgueira: *Specifying use case behavior with interaction models.* Journal of Object Technology, 2(2), 2003.

[GKKL10] Geltinger, Georg, Oliver Kappes, Laura Kolenc und Frank Lehmann: *Das SOPHIST-REgelwerk: Anwendung und Bewertung einer Methode des Requirements Engineerings.* Technischer Bericht, Berufsakademie Ravensburg, Ravensburg, 2010.

[GL00] Grieskamp, Wolfgang und M. Lepper: *Using use cases in executable Z.* In: *icfem*, Seite 111. Published by the IEEE Computer Society, 2000.

[Gli95] Glinz, Martin: *An integrated formal model of scenarios based on statecharts.* In: Schäfer, Wilhelm und Pere Botella (Herausgeber): *Software Engineering - ESEC '95*, Nummer 1994 in *Lecture Notes in Computer Science*, Seiten 254–271. Springer Berlin Heidelberg, 1995.

[GLST01] Grieskamp, Wolfgang, Markus Lepper, W. Schulte und Nikolai Tillmann: *Testable use cases in the abstract state machine language.* In: *Quality Software, 2001. Proceedings. Second Asia-Pacific Conference on*, Seiten 167–172. IEEE, 2001.

[Got03] Gottesdiener, E: *Use Cases: best practices.* Rational Software white paper, 2003.

[GP96] Green, T. R. G. und M. Petre: *Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework.* JOURNAL OF VISUAL LANGUAGES AND COMPUTING, 7:131–174, 1996.

[gra] *Graphiti Project Website.* `http://www.eclipse.org/graphiti/`.

[gtk] *GTK Project Website.* `http://www.gtk.org/`.

[Hal97] Hall, Anthony: *Do interactive systems need specifications.* In: *DSV-IS*, Band 97, Seiten 1–12, 1997.

[Har87] Harel, David: *Statecharts: a visual formalism for complex systems.* Science of Computer Programming, 8(3):231–274, Juni 1987.

[Has08] Hassine, Jameleddine: *Formal Sematics and verification of Use Case Maps.* Doctoral Thesis, Concordia University, 2008.

[HBR02] Hall, T., S. Beecham und A Rainer: *Requirements problems in twelve software companies: an empirical analysis.* IEE Proceedings - Software, 149(5):153, 2002.

[HHT02] Hausmann, J.H., Reiko Heckel und G. Taentzer: *Detection of conflicting functional requirements in a use case-driven approach.* In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, Seiten 105–115. IEEE, 2002.

[HK94] Holloway, L E I und B H Krogh: *Controlled Petri Nets: A Tutorial Survey.* In: Cohen, Guy und Jean-Pierre Quadrat (Herausgeber): *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, Nummer c in *Lecture Notes in Control and Information Sciences*, Seiten 158–168. Springer, Berlin Heidelberg, 1994.

[HK02] Harel, David und Hillel Kugler: *Specifying and Executing Requirements- The Play-In / Play-Out Approach Summary.* Computer-Aided Design, Seiten 84–85, 2002.

[HKP05] Harel, David, Hillel Kugler und Amir Pnueli: *Synthesis revisited: Generating statechart models from scenario-based requirements.* In: Kreowski, Hans-Jörg, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg und Gabriele Taentzer (Herausgeber): *Formal Methods in Software and Systems Modeling*, Nummer 287, Seiten 309–324. Springer, Berlin Heidelberg, 2005.

[HL01] Hofmann, H.F. und F. Lehner: *Requirements engineering as a success factor in software projects.* IEEE Software, 18(4):58–66, Juli 2001.

[HL10] Hoffmann, Veit und Horst Lichter: *A Model Based Narrative Use Case Simulation Environment.* In: Cordeiro, Jose A. Moinhos, Maria Virvou und Boris Shishkov (Herausgeber): *ICSOFT 2010 - Proceedings of the Fifth International Conference on Software and Data Technologies, Volume 2, Athens, Greece, July 22-24, 2010*, Seiten 63–72. SciTePress, 2010.

[HLNW09] Hoffmann, Veit, Horst Lichter, Alexander Nyssen und Andreas Walter: *Towards the integration of uml-and textual use case modeling.* Journal of Object Technology, 8(3):85–100, 2009.

[HM03] Harel, David und Rami Marelly: *Specifying and executing behavioral requirements: the play-in/play-out approach.* Software and Systems Modeling, 2(2):82–107, Juli 2003.

[HMU07] Hopcroft, J.E., R. Motwani und J.D. Ullman: *Introduction to automata theory, languages, and computation.* Pearson/Addison Wesley, 2007.

[Hoa69] Hoare, C.A.R.: *An axiomatic basis for computer programming.* Communications of the ACM, 12(10):576–580, Oktober 1969.

[HSW02] Homrighausen, Andreas, Hans-Werner Six und Mario Winter: *Round-Trip Prototyping Based on Integrated Functional and User Interface Requirements Specifications*. Requirements Engineering, 7(1):34–45, April 2002.

[IEE98] IEEE Computer Society. Software Engineering Standards Committee: *IEEE Guide for Developing System Requirements Specifications*. Secretary, 1998.

[II98] IEEE Computer Society. Software Engineering Standards Committee und IEEE-SA Standards Board: *IEEE recommended practice for software requirements specifications*. In: *Electronics*, Band 1998. Institute of Electrical and Electronics Engineers, 1998.

[Int98] International Telecommunications Union (ITU-T): *ITU-T Recommendation Z.120 Annex B*. Technischer Bericht, 1998.

[Int04] International Telecommunications Union (ITU-T): *ITU-T Recommendation Z.120*. Technischer Bericht, 2004.

[Int05] International Organisation for Standardization (ISO): *ISO/IEC 9126. Software engineering – Product quality*. International Electronical Commission, 2005.

[Int10] International Organisation for Standardization (ISO): *Systems and software engineering – Vocabulary*. ISO/IEC/IEEE 24765:2010(E), Seiten 1—-418, 2010.

[Jac04] Jacobson, Ivar: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[JBR99] Jacobson, Ivar, Grady Booch und James Rumbaugh: *The Unified Software Development Process*. Addison-Wesley Professional, Februar 1999.

[Jen91] Jensen, Kurt: *Coloured Petri nets: A high level language for system design and analysis*. Advances in Petri nets 1990, Seiten 342–416, 1991.

[Joo00] Joos, S.: *Adora-L- eine Modellierungssprache zur Spezifikation von Software-Anforderungen*. Doktorarbeit, Universität Zürich, 2000.

[Jor04] Jorgensen, J.B.: *Executable use cases as links between application domain requirements and machine specifications*. "Third International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM04)"W5S Workshop - 26th International Conference on Software Engineering, 2004:8–13, 2004.

[JW07] Jayaraman, Praveen K. und Jon Whittle: *UCSIM: A Tool for Simulating Use Case Scenarios.* In: *29th International Conference on Software Engineering (ICSE'07 Companion)*, Seiten 43–44. IEEE, Mai 2007.

[KC08] Kakollu, Durga Prasad und B.D. Chaudhary: *A Z-Specification of Classification and Relationships between Usecases.* 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Seiten 779–784, 2008.

[KG03] Kulak, Daryl und Eamonn Guiney: *Use Cases: Requirements in Context.* Addison-Wesley Professional, August 2003.

[Kie91] Kieback, A.: *Prototyping in industriellen Software-Projekten.* GMD-Studien. Ges. für Mathematik u. Datenverarbeitung, 1991.

[Kil10] Kiliclar, Kamil Bora: *Entwicklung Meta-Modell basierter Testfall-Spezifikationen aus Narrative Use Case Modellen.* Diplomathesis, RWTH Aachen University, 2010.

[KK09] Kollanus, Sami und Jussi Koskinen: *Survey of Software Inspection Research.* The Open Software Engineering Journal, 3(1):15–34, Mai 2009.

[Kla09] Klapdor, Thomas Nicolas: *Werkzeugunterstützung für die Qualitätsbewertung von Use Case basierten Anforderungsspezifikationen.* Diplomathesis, RWTH Aachen University, 2009.

[KLM09] Knauss, Eric, D. Lubke und Sebastian Meyer: *Feedback-driven requirements engineering: the heuristic requirements assistant.* In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, Seiten 587–590. IEEE, 2009.

[KLSHZ91] Kieback, A., H. Lichter, M. Schneider-Hufschmidt und H. Züllighoven: *Prototyping in industriellen Software-Projekten: Erfahrung und Analysen.* Informatik-Spektrum, 15(184):51, 1991.

[KP05] Kanyaru, John und K. Phalp: *Requirements validation with enactable models of state-based use cases.* Empirical Assessment in Software Engineering, EASE, 2005.

[Kra11] Krause, Jan: *Entwicklung einer werkzeugunterstützten Tailoring-Methode für projektspezifische Use-Case-Modellierungsprozesse.* Bachelorthesis, RWTH Aachen University, 2011.

[Krc09] Krcmar, H.: *Informationsmanagement.* Springer, 2009.

[Lai02] Laitenberger, Oliver: *A Survey of Software Inspection Technologies.* In: Chang, S (Herausgeber): *Handbook of Software Engineering and Knowledge Engineering Volume II.* World-Scientific, 2002.

[Leh08] Lehmacher, Mark: *Eine Simulationsumgebung für strukturierte natürlichsprachliche Anwendungsfallbeschreibungen.* Diploma Thesis, RWTH-Aachen University, 2008.

[LH01] Lauesen, S. und M.B. Harning: *Virtual windows: linking user tasks, data models, and interface design.* IEEE Software, 18(4):67–75, Juli 2001.

[Li99] Li, Liwu: *A semi-automatic approach to translating use cases to sequence diagrams.* In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*, Seiten 184–193, Washington DC, 1999. IEEE Computer Society.

[Lic93] Lichter, H.: *Entwicklung und Umsetzung von Architekturprototypen für Anwendungssoftware.* Verl. d. Fachvereine an d. Schweiz. Hochschulen u. Techniken (VdF), 1993.

[LJ08] Lei, Ma und Wei Chang Jiang: *Research on Activity Based Use Case Meta-Model.* 2008 International Conference on Advanced Computer Theory and Engineering, Seiten 843–846, Dezember 2008.

[LL81] Lepore, Ernest und Barry Loewer: *Translational semantics.* Synthese, 48(1):121–133, Juli 1981.

[LL10] Ludewig, J. und H. Lichter: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* Dpunkt.Verlag GmbH, 2010.

[LLH01] Li, X., Z. Liu und J. He: *Formal and use-case driven requirement analysis in UML.* In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, Nummer 230, Seiten 215–224. IEEE, 2001.

[LMCS98] Lee, Woo Jin, Student Member, Sung Deok Cha und Ieee Computer Society: *Integration and analysis of use cases using modular Petri nets in requirements engineering.* IEEE Transactions on Software Engineering, 24(12):1115–1130, 1998.

[Löw11] Löwenthal, Tobias: *Generierung von web-basierten Prototypen für Geschäftsanwendungen.* Diplomathesis, RWTH Aachen University, 2011.

[LV01] Lauesen, Soren und Otto Vinter: *Preventing Requirement Defects: An Experiment in Process Improvement.* Requirements Engineering, 6(1):37–50, Februar 2001.

[Mai98] Maiden, Neil A M: *CREWS-SAVRE : Scenarios for Acquiring and Validating Requirements.* Automated Software Engineering, 446:419–446, 1998.

[MAKS12] Münch, J., O. Armbrust, M. Kowalczyk und M. Soto: *Software Process Definition and Management.* Systems- And Software-Engineering. Springer, 2012.

[MB03] Metz, Pierre und John O Brien: *Specifying Use Case Interaction : Types of Alternative Courses.* JOURNAL OF OBJECT TECHNOLOGY, 2(2):111–131, 2003.

[med] *Siemens Medical Systems Website.* `http://www.medical.siemens.com/`.

[Men04a] Mencl, V: *Deriving behavior specifications from textual use cases.* In: *Proc. of Workshop on Intelligent Technologies for Software Engineering (WITSE'04), Austria*, Seiten 331–341, Linz, 2004. Oesterreichische Computer Gesellschaft.

[Men04b] Mencl, Vladimir: *Converting Textual Use Cases into Behaviour Specifications.* Technischer Bericht, Citeseer, 2004.

[Mos90] Mosses, Peter D.: *Handbook of theoretical computer science (vol. B).* Kapitel Denotational semantics, Seiten 575–631. MIT Press, Cambridge, MA, USA, 1990.

[MP82] McCallum, Douglas R. und James L Peterson: *Computer-based readability indexes.* In: *Proceedings of the ACM '82 conference on - ACM 82*, Seiten 44–48, New York, New York, USA, 1982. ACM Press.

[MSK10] Mizouni, Rabeb, Daniel Sinnig und Ferhat Khendek: *Towards an Integrated Model for Functional and User Interface Requirements.* Ifip International Federation For Information Processing, Seiten 214–221, 2010.

[Mus10] Mustafa, B.A.: *An Experimental Comparison of Use Case Models Understanding by Novice and High Knowledge Users.* In: *Proceeding of the 2010 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the 9th SoMeT_10*, Band 416353, Seiten 182–199. IOS Press, 2010.

[Mus11] Mussil, Michael: *Entwicklung eines Methodenbaukastens für Use-Case-basierte Anforderungsmodelle.* Diplomathesis, RWTH Aachen University, 2011.

[MZ99] Mansurov, N und D Zhukov: *Automatic synthesis of SDL models in use case methodology.* In: *SDL'99, Proceedings of the Ninth SDL Forum.* Citeseer, 1999.

[Nau11] Naussed, Holger: *Entwicklung einer werkzeuggestützten Gui Prototyp basierten Review Methode für Anwendungsfälle.* Diplomathesis, RWTH-Aachen University, 2011.

[Nei03] Neill, C.J.: *Requirements engineering: The state of the practice.* Software, IEEE, 20(6):40–45, 2003.

[Nie94] Nielsen, Jakob: *Usability engineering.* Morgan Kaufmann Series in Interactive Technologies. AP Professional, 1994.

[NJJ+96] Nissen, H.W., M.A. Jeusfeld, M. Jarke, G.V. Zemanek und H. Huber: *Managing multiple requirements perspectives with metamodels.* Software, IEEE, 13(2):37–48, 1996.

[NO05] Nawrocki, Jerzy und Ł. Olek: *Use-cases engineering with uc workbench.* In: *Proceeding of the 2005 conference on Software Engineering: Evolution and Emerging Technologies*, Seiten 319–329. IOS Press, 2005.

[Nor87] Norman, D. A.: *Some observations on mental models.* In: Baecker, R. M. (Herausgeber): *Human-computer interaction*, Seiten 241–244. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[NUOT01] Nakatani, Takato, Tetsuya Urai, S. Ohmura und Tetsuo Tamai: *A requirements description metamodel for use cases.* In: *Proceedings Eighth Asia-Pacific Software Engineering Conference*, Seiten 251–258. IEEE Comput. Soc, 2001.

[Nys09] Nyssen, Alexander: *Model-Based Construction of Embedded A Methodology for Small Devices.* Doctoral Thesis, RWTH Aachen University, 2009.

[Obj11] Object Management Group (OMG): *OMG unified modeling language: superstructure.* (August), 2011.

[OMG07] OMG: *Unified Modeling Language : Superstructure.* 1(February), 2007.

[OMG11] OMG: *OMG Meta Object Facility (MOF) 2.0 Core Specification Version 2.0: Final Adopted Specification.* (August), 2011.

[OSG09] OSGI Alliance: *OSGi Service Platform, Core Specification, Release 4, Version 4.2.* Technischer Bericht, OSGI Alliance, September 2009.

[OV06] Oestereich, B. und U. Vigenschow: *Agiles Software-Projektmanagement (APM).* dpunkt, 2006.

[Par10] Partsch, H.A.: *Requirements-Engineering systematisch: Modellbildung für softwaregestützte Systeme.* Springer, 2010.

[pen] *Pencil Project Website.* `http://pencil.evolus.vn/`.

[Pet62] Petri, Carl Adam: *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[PK01] Phillips, C. und E. Kemp: *Extending UML use case modelling to support graphical user interface design.* Proceedings 2001 Australian Software Engineering Conference, 201:48–57, 2001.

[Plo04a] Ploesch, Reinhold: *Contracts, Scenarios and Prototypes - An Integrated Approach to High Quality Software.* Springer-Verlag Berlin Heidelberg New York, 2004.

[Plo04b] Plotkin, G.D.: *A structural approach to operational semantics.* Journal of Logic and Algebraic Programming, 60-61:17–139, Juli 2004.

[PMP11] Post, Amalinda, Igor Menzel und Andreas Podelski: *Applying Restricted English Grammar on Automotive Requirements — Does it Work ? A Case Study.* Lecture Notes in Computer Science, 6606:166–180, 2011.

[Poh08] Pohl, Klaus: *Requirements Engineering.* Dpunkt.Verlag GmbH, 2., korrigierte Auflage. Auflage, Juni 2008.

[PQ10] Pries, K.H. und J.M. Quigley: *Scrum Project Management.* Taylor & Francis, 2010.

[pri] *Primefaces Project Website.* `http://primefaces.org/`.

[PVC07] Phalp, Keith Thomas, Jonathan Vincent und Karl Cox: *Assessing the quality of use case descriptions.* Software Quality Journal, 15(1):69–97, Januar 2007.

[rap] *Eclipse-RAP Project Website.* `http://www.eclipse.org/rap/`.

[RBA02] Richards, D., K. Boettger und O. Aguilera: *A controlled language to assist conversion of use case descriptions into concept lattices.* AI 2002: Advances in Artificial Intelligence, Seiten 1–11, 2002.

[RBH+08] Rausch, A., M. Broy, R. Höhn, K. Bergner und S. Höppner: *Das V-Modell XT: Grundlagen, Methodik Und Anwendungen.* Springer, 2008.

[Rec06] Rechenberg, P.: *Informatik-Handbuch.* Hanser, 2006.

[Ren10] Renz, Eduard: *Nutzerzentrierte Use-case-basierte Simulation von Systemverhalten.* Diplomathesis, RWTH Aachen University, 2010.

[RG97] Rechenberg, P. und Pomberger G.: *Informatik-Handbuch.* Hanser, 1997.

[RGJ00] Ryser, Johannes, Martin Glinz und Johannes Ryser: *SCENT : A Method Employing Scenarios to Systematically Derive Test Cases for System Test.* Technischer Bericht, University of Zurich, Zurich, 2000.

[RKW95] Regnell, Björn, K. Kimbler und A. Wesslen: *Improving the use case driven approach to requirements engineering.* In: *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, Nummer March, Seiten 40–47. IEEE Comput. Soc. Press, 1995.

[RS07] Rosenberg, Doug und Matt Stephens: *Use case driven object modeling with UML: theory and practice.* apress, 2007.

[rsa] *Rational Software Architect Website.* `http://www-01.ibm.com/software/awdtools/swarchitect/`.

[RST+] Ricca, Filippo, Giuseppe Scanniello, Marco Torchiano, G. Reggio und E. Astesiano: *Usefulness of Screen Mockups in Use Case Descriptions-A Formal Experiment.*

[RST+10] Ricca, Filippo, Giuseppe Scanniello, Marco Torchiano, Gianna Reggio und Egidio Astesiano: *On the effectiveness of screen mockups in requirements engineering: Results from an internal replication.* In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Seiten 1–10. ACM, 2010.

[Rup07] Rupp, C.: *Requirements-Engineering und -Management: professionelle, iterative Anforderungsanalyse für die Praxis.* Hanser, 2007.

[Sch02] Schwitter, Rolf: *English as a formal specification language.* In: *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, Seiten 228–232. IEEE, 2002.

[SCK09] Sinnig, Daniel, Patrice Chalin und Ferhat Khendek: *LTS semantics for use case models.* In: *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*, Seite 365, New York, New York, USA, 2009. ACM Press.

[Sel03] Selic, B.: *The pragmatics of model-driven development.* IEEE Software, 20(5):19–25, September 2003.

[Sep90] September, A.: *IEEE Standard Glossary of Software Engineering Terminology.* Office, 121990(1), 1990.

[SMG04] Seybold, Christian, Silvio Meier und M. Glinz: *Evolution of requirements models by simulation.* In: *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, Seiten 43–48. IEEE, 2004.

[SMK10] Sinnig, Daniel, R. Mizouni und F. Khendek: *Bridging the gap: empowering use cases with task models.* In: *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, Seiten 291–296. ACM, 2010.

[SO01] Sindre, Guttorm und A.L. Opdahl: *Templates for misuse case description.* In: *Proc. Seventh International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'2001)*. Citeseer, 2001.

[Som05] Some, Stephane S.: *Enhancement of a use cases based requirements engineering approach with scenarios.* In: *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, Seiten 8–pp. IEEE, 2005.

[Som07a] Somé, Stéphane S.: *Petri nets based formalization of textual use cases.* University of Ottawa, Ottawa, OT, Canada, Technical Report TR-2007-11, Nov, Seiten 1–46, 2007.

[Som07b] Sommerville, Ian: *Software Engineering.* Pearson Studium, 8 Auflage, 2007.

[Som09] Somé, Stéphane S.: *A Meta-Model for Textual Use Case Description.* J. of Object Technology, 8(7):87–106, 2009.

[son] *Sonar Project Website.* `http://www.sonarsource.org/`.

[SPW07] Sinha, Avik, Amit Paradkar und Clay Williams: *On Generating EFSM models from Use Cases Information in a Use Case Specification.* System, 2007.

[SS97] Sommerville, Ian und Peter Sawyer: *Requirements Engineering: A Good Practice Guide.* John Wiley & Sons, 1 Auflage, 1997.

[SS00] Sendall, Shane und Alfred Strohmeier: *From Use Cases to System Operation Specifications.* System, Seiten 1–15, 2000.

[SS11] Siqueira, F.L. und P.S.M. Silva: *An Essential Textual Use Case Meta-model Based on an Analysis of Existing Proposals.* In: *Proc. Workshop on Requirements Engineering (WER)*, Seiten 419–430, 2011.

[Sta73] Stachowiak, H.: *Allgemeine Modelltheorie.* Springer-Verlag, 1973.

[Sut97] Sutcliffe, Alistair: *A Technique Combination Approach to Requirements Engineering.* Requirements Engineering, 1997.

[SVEH07] Stahl, T., M. Völter, S. Efftinge und A. Haase: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* Dpunkt.Verlag GmbH, 2007.

[SWT] *Eclipse-SWT Project Website.* `http://www.eclipse.org/swt/`.

[TA08] Tullis, T. und W. Albert: *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics.* Interactive Technologies. Elsevier/Morgan Kaufmann, 2008.

[Tuo98] Tuok, R: *Formal specification and use case generation for a mobile telephony system.* Computer Networks and ISDN Systems, 30(11):1045–1063, Juni 1998.

[vdPKS+03] Poll, J.A. van der, P. Kotzé, A. Seffah, T. Radhakrishnan und A. Alsumait: *Combining UCMs and formal methods for representing and checking the validity of scenarios as user requirements.* In: *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, Seiten 59–68. South African Institute for Computer Scientists and Information Technologists, 2003.

[Vol] *Volere Website.* `http://www.volere.co.uk//`.

[Wal07] Walter, Andreas: *Ein Use Case-Modellierungswerkzeug für die ViPER-Plattform.* Diploma Thesis, RWTH Aachen University, 2007.

[WHL09] Weidmann, Christoph, Veit Hoffmann und Horst Lichter: *Einsatz und Nutzen von Use Cases -Ergebnisse einer empirischen Untersuchung*, 2009.

[Wil01] Williams, Clay E: *Toward a Test-Ready Meta-model for Use Cases.* In: *Proceedings of the Workshop on Practical UML-based Rigorous Development Methods*, Seiten 270–287, 2001.

[Win] *WindowBuilder Project Website.* `http://www.eclipse.org/windowbuilder/`.

[WJ06] Whittle, J. und P.K. Jayaraman: *Generating Hierarchical State Machines from Use Case Charts.* In: *14th IEEE International Requirements Engineering Conference (RE'06)*, Seiten 19–28. IEEE, September 2006.

[WK92] Wood, D.P. und K.C. Kang: *A classification and bibliography of software prototyping.* Technischer Bericht, Carnegie Mellon University, Software Engineering Institute, 1992.

[WS90] Wasserman, Anthony I. und David T. Shewmake: *Human-computer interaction.* In: Preece, Jenny (Herausgeber): *The role of prototypes in the User Software Engineering (USE) methodology*, Seiten 385–401. Prentice Hall Press, Upper Saddle River, NJ, USA, 1990.

[XH07] Xu, Dianxiang und Xudong He: *Generation of test requirements from aspectual use cases.* In: *Proceedings of the 3rd workshop on Testing aspect-oriented programs - WTAOP '07*, Seiten 17–22, New York, New York, USA, 2007. ACM Press.

[YBL10] Yue, Tao, Lionel C Briand und Yvan Labiche: *An Automated Approach to Transform Use Cases into Activity Diagrams.* Quality Engineering, Seiten 1–26, 2010.

[You04] Young, RR: *The requirements engineering handbook.* Technology Management, 2004.

[ZD09] Zhao, Jinqiang und Zhenhua Duan: *Verification of Use Case with Petri Nets in Requirement Analysis.* In: *Proceedings of the International Conference on Computational Science and Its Applications: Part II*, Seiten 29–42, Berlin, Heidelberg, 2009. Springer.

[ZXZ01] Zhang, Lu, Dan Xie und Wei Zou: *Viewing Use Cases as Active Objects.* Software Engineering Notes, 26(2):44–48, 2001.

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] we show how this looks like and how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Many of our contributions build on UML/P described in the two books [Rum11] and [Rum12] are implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams (ADs) [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02] The question how to adapt and extend the UML in discussed in [PFR02] on product line annotations for UML and to more general discussions and insights on how to use meta-modeling for defining and adapting the UML [EFLR99], [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06], [KRV10], [Kra10] describes an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools

can be defined in modular forms [KRV08, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been examined in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a].

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory. [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied on class diagrams in [CGR08]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] embodies the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development, maintenance and [LRSS10] technologies for evolving models within a language and across languages and linking architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Many products exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures the commonalities as well as the differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are added (that sometimes also modify the core). A set of applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] describes an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. And we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on state machines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using state machines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts

[GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specifications concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [THR+13] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW12] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13] that perfectly fits Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus, enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development. Application classes like Cyber-Physical Systems [KRS12], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools. We tackle these challenges by perusing a model-based, generative approach [PR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g.,c collecting data from temperature, cars etc. are easily developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, October 2007.

[BCGR09a] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, 2009.

[BCR07a] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, February 2007.

[BCR07b] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, TUM-I9737, TU Munich, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In M. Broy and B. Rumpe, editors, *RTSE '97: Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, LNCS 1526, pages 43–68, Bernried, Germany, October 1998. Springer.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Proceedings of the 10th Workshop on Automotive Software Engineering (ASE 2012)*, pages 789–798, Braunschweig, Germany, September 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*. Springer, 2012.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, CfG Fakultät, TU Braunschweig, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*, LNCS 5795, pages 670–684, Denver, Colorado, USA, October 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluver Academic Publisher, 1999.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-Based Modeling of Buildings and Facilities. In *Proceedings of the 11th International Conference for Enhanced Building Operations (ICEBO' 11)*, New York City, USA, October 2011.

[FPPR12] Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Proceedings of the 7th International Conference on Energy Efficiency in Commercial Buildings (IEECB)*, Frankfurt a. M., Germany, April 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop*, Paderborn, Germany, October 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, Toulouse, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, Informatik Bericht 2008-01, pages 76–89, CFG Fakultät, TU Braunschweig, March 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TUM, Munich, Germany, 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Technical Report 2006-04, CfG Fakultät, TU Braunschweig, August 2006.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings der Modellierung 2006*, Lecture Notes in Informatics LNI P-82, Innsbruck, März 2006. GI-Edition.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TUM, Munich, Germany, 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems. 16th Monterey Workshop*, LNCS 6662, pages 17–32, Redmond, Microsoft Research, 2011. Springer.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality. 18th International Working Conference, Proceedings, REFSQ 2012*, Essen, Germany, March 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Model Driven Engineering Languages and Systems, Proceedings of MODELS*, LNCS 6394, Oslo, Norway, 2010. Springer.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC), Tokyo*, pages 22–31. ACM, September 2013.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab / Simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 11–18, New York, NY, USA, 2013. ACM.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Proceedings of the Third European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, pages 99–113. Springer, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In H. Arabnia and H. Reza, editors, *Proceedings of the 2009 International Conference on Software Engineeering in Research and Practice*, Las Vegas, Nevada, USA, 2009.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI) at ICSE 2012*, pages 61–66, Zurich, Switzerland, June 2012. IEEE.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, Oct 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *TOOLS 26, Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Proceedings of International Software Product Lines Conference (SPLC 2011)*. IEEE Computer Society, August 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, fortiss GmbH*, February 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208, Oxford, UK, March 2012. Springer.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik in Berlin*, Lecture Notes in Informatics LNI 198, pages 181–192, 27. Februar - 2. März 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.)*, Techreport B-108, Helsinki School of Economics, Orlando, Florida, USA, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Proceedings of the Modelling of the Physical World Workshop MOTPW'12, Innsbruck, October 2012*, pages 2:1–2:6. ACM Digital Library, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. P. Dini, IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag Aachen, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Germany, 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Proceedings of the first International Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 323–338. Chapmann & Hall, 1996.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006 (DSM'06), Portland, Oregon USA*, Technical Report TR-37, pages 150–158, Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07), Montreal, Quebec, Canada*, Technical Report TR-38, pages 8–10, Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), Nashville, TN, USA, October 2007*, LNCS 4735. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In R. F. Paige and B. Meyer, editors, *Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Zurich, Switzerland, 2008*, Lecture Notes in Business Information Processing LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 241–270. Springer, October 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proc. Euro. Soft. Eng. Conf. and SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 592–607, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Proc. 25th Euro. Conf. on Object Oriented Programming (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 153–167. Springer, 2011.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In G. J. Chastek, editor, *Software Product Lines - Second International Conference, SPLC 2*, LNCS 2379, pages 188–197, San Diego, 2002. Springer.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, LNCS 873. Springer, October 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In J. Davies J. M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In H. Kilov and K. Baclawski, editors, *Practical foundations of business and system specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[PR13] Antonio Navarro Perez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proc. of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing. Co-located with MODELS 2013, Miami*, Sun SITE Central Europe Workshop Proceedings CEUR 1118, pages 15–24. CEUR-WS.org, 2013.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technical Report TUM-I9510, Technische Universität München, 1995.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In N. Seyff and A. Koziolek, editors, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA), May 6-10, 2013, Karlsruhe, Germany*, pages 10–12, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, ISBN 3-89675-149-2, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, Hershey, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever, editor, *Formal Methods for Components and Objects*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future. 9th International Workshop, RISSEF 2002. Venice, Italy, October 2002*, LNCS 2941. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, second edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer, second edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Germany, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 57–76, October 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 461–466, Karlsruhe, Germany, May 2013. IEEE.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Germany, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, Aachen, Germany, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In D. Schaefer, editor, *Proceedings of the SESAR Innovation Days*. EUROCONTROL, November 2011.