

Helmut Balzert

Lehrbuch der Softwaretechnik

Basiskonzepte und Requirements Engineering

3. Auflage

Lehrbücher der Informatik

Spektrum
AKADEMISCHER VERLAG

Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering

Lehrbücher der Informatik

Herausgegeben von
Prof. Dr.-Ing. habil. Helmut Balzert

Heide Balzert
Lehrbuch der Objektmodellierung
Analyse und Entwurf mit der UML 2, 2. Auflage

Helmut Balzert
Lehrbuch Grundlagen der Informatik
Konzepte und Notationen in UML 2, Java 5, C# und C++,
Algorithmik und Softwaretechnik, Anwendungen, 2. Auflage

Klaus Zeppenfeld
Lehrbuch der Grafikprogrammierung
Grundlagen, Programmierung, Anwendung

Helmut Balzert
Objektorientierte Programmierung
mit Java 5

Helmut Balzert
Lehrbuch der Softwaretechnik:
Softwaremanagement
2. Auflage

Helmut Balzert

Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering

3. Auflage

Unter Mitwirkung von
Heide Balzert
Rainer Koschke
Uwe Lämmel
Peter Liggesmeyer
Jochen Quante

Autor

Prof. Dr. Helmut Balzert

E-Mail: hb@W3L.de

Wichtiger Hinweis für den Benutzer

Der Verlag und der Autor haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch zu publizieren. Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder ähnliches zurückzuführen sind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich. Der Verlag übernimmt keine Gewähr dafür, dass die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Der Verlag hat sich bemüht, sämtliche Rechteinhaber von Abbildungen zu ermitteln. Sollte dem Verlag gegenüber dennoch der Nachweis der Rechtsinhaberschaft geführt werden, wird das branchenübliche Honorar gezahlt.

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.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

3. Auflage 2009

© Spektrum Akademischer Verlag Heidelberg 2009

Spektrum Akademischer Verlag ist ein Imprint von Springer

09 10 11 12 13

5 4 3 2 1

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Planung und Lektorat: Dr. Andreas Rüdinger

Redaktion und Gestaltung: M.Sc. Kerstin Kohl, Witten; Dagmar Fraude, Witten

Umschlaggestaltung: SpieszDesign, Neu-Ulm

Titelbild: Anna Solecka-Zach: „Ohne Titel“ (1995)

Satz: W3L GmbH, Witten – automatischer Satz aus der W3L-Plattform.

ISBN 978-3-8274-1705-3

Vorwort zur 3. Auflage

Die Softwaretechnik bildet einen Grundpfeiler der Informatik. Sie hat sich zu einem umfassenden Wissenschaftsgebiet entwickelt. Um Ihnen, liebe Leserin, lieber Leser, ein optimales Erlernen dieses Gebietes zu ermöglichen, habe ich – zusammen mit Koautoren – ein dreibändiges Lehr- und Lernbuch über die Softwaretechnik geschrieben. Es behandelt die Kerngebiete »Basiskonzepte und Requirements Engineering« (Band 1), »Entwurf und Architekturen« (Band 2) und »Softwaremanagement« (Band 3) sowie die Abhängigkeiten zwischen diesen Gebieten.

Ich habe das Gebiet der Softwaretechnik in mehrere große Bereiche gegliedert – der Tempel der Softwaretechnik (Abb. 0.0-1) veranschaulicht diese Gliederung. Ziel jeder Softwareentwicklung ist es, ein lauffähiges Softwareprodukt zu erstellen, zu warten und zu pflegen. Im Mittelpunkt der Gliederung steht daher die Softwareentwicklung (mittlere Säule). Jede der Aktivitäten der Softwareentwicklung trägt dazu bei, Teilprodukte zu erstellen, die dann in ein Gesamtprodukt münden.

Aufbau &
Gliederung

Eine Softwareentwicklung läuft aber *nicht* von alleine ab. Sie basiert auf und nutzt Basistechniken, das sind Prinzipien, Methoden, Werkzeuge, *Best Practices*.

Die eigentliche Softwareentwicklung wird durch die zwei Säulen »Softwaremanagement« und »Prozess- und Qualitätsmodelle« und das Dach »Allgemeines Management« eingerahmt. Bei den Aktivitäten des Managements und der Prozess- & Qualitätssicherung handelt es sich um *begleitende* Aktivitäten, deren Ergebnisse aber selbst *nicht* Bestandteil des Endprodukts sind. Dennoch sind sie eminent wichtig – was die große Anzahl fehlgeschlagener Softwareprojekte deutlich zeigt.

In diesem Buch werden die Basistechniken, die Basiskonzepte und das *Requirements Engineering* behandelt. Dieses Buch besteht aus folgenden Teilen:

I Die Wissenschaftsdisziplin Softwaretechnik

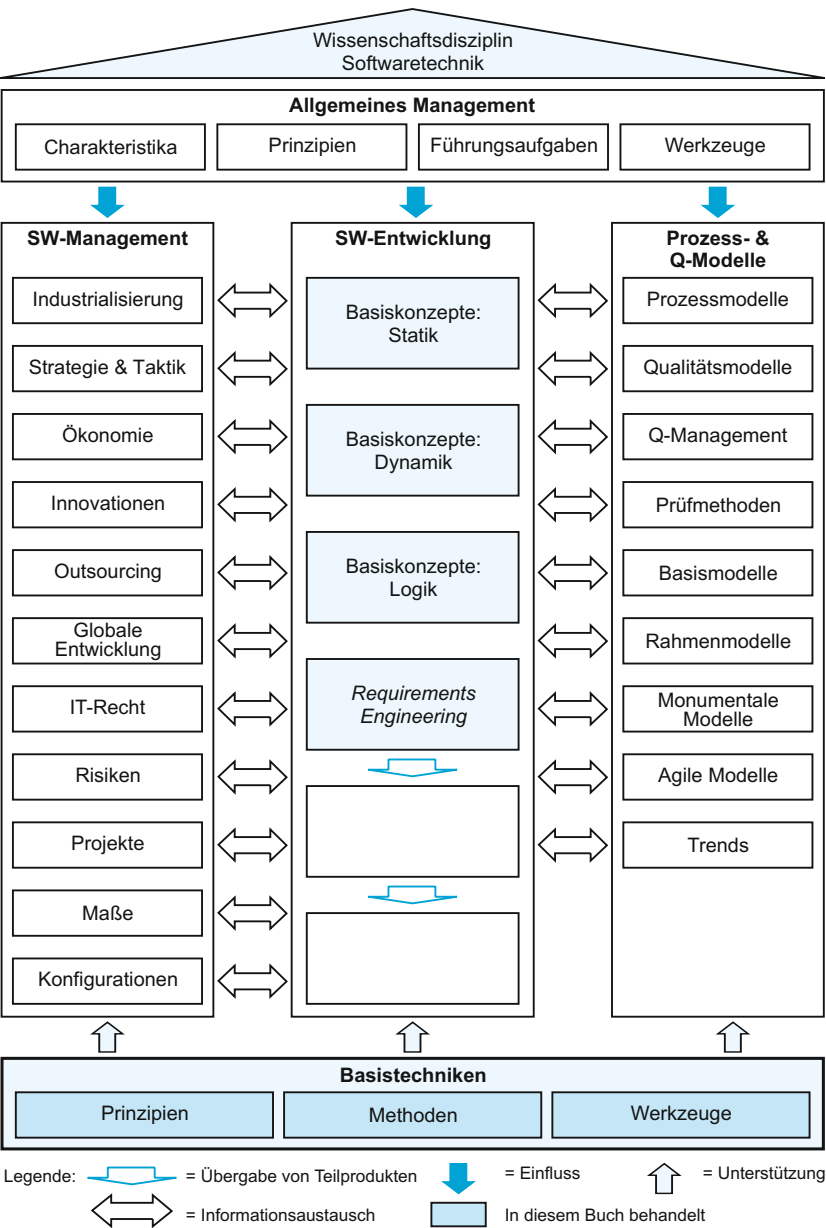
II Basistechniken

III Basiskonzepte

IV Requirements Engineering

Vorwort zur 3. Auflage

Abb. 0.0-1:
Gliederung der
Softwaretechnik.



Zu einem dreibändigen Lehrbuch der Softwaretechnik gehört natürlich auch eine Reflexion über die Wissenschaftsdisziplin:

■ »Die Wissenschaftsdisziplin Softwaretechnik«, S. 1

Wie in anderen dynamischen Disziplinen auch, so gibt es auch in der Softwaretechnik Trends und »Hypes«. Nach einer Euphoriephase erfolgt dann oft die Ernüchterung. Für die Lehre und das Lernen ist es

daher wichtig, das Konstante in der Softwaretechnik zu identifizieren – das unabhängig von aktuellen Trends gilt. Ich habe allgemeingültige Prinzipien, Methoden und Werkzeuge unter dem Oberbegriff Basistechniken zusammengefasst:

■ »Basistechniken«, S. 23

Analog habe ich versucht, bei den heute in der Softwaretechnik angewandten Konzepten die grundlegenden Konzepte zu isolieren und in die drei Kategorien Statik, Dynamik und Logik einzuordnen:

■ »Basiskonzepte«, S. 99

Softwaresysteme lassen sich grob in Informationssysteme und softwareintensive Systeme gliedern. Bei softwareintensiven Systemen ist die Software nur ein – wenn auch wichtiger – Bestandteil des Systems. Je nachdem, um was für eine Art von Softwaresystem es sich handelt, werden die Basiskonzepte unterschiedlich intensiv und in verschiedenen Kombinationen eingesetzt.

Um diese Unterschiede zu verdeutlichen, werden zwei Fallstudien verwendet, auf die sich in den Beispielen zu den Basiskonzepten immer wieder bezogen wird:

■ »Fallstudie: SemOrg – Die Spezifikation«, S. 107

■ »Fallstudie: Fensterheber – Die Spezifikation«, S. 117

Zu wissen, was der Kunde will, und dies zu ermitteln, zu beschreiben, zu spezifizieren, zu analysieren und in Form einer fachlichen Lösung zu modellieren ist die Aufgabe des *Requirements Engineering* – im Deutschen auch als Systemanalyse bezeichnet. Da im *Requirements Engineering* viele Basistechniken und Basiskonzepte benötigt und eingesetzt werden, habe ich die Basistechniken und Basiskonzepte an den Anfang des Buches gestellt. Das *Requirements Engineering* wird oft noch unterschätzt – hier werden jedoch die Grundlagen für den Erfolg oder Misserfolg einer Softwareentwicklung gelegt:

■ »Requirements Engineering«, S. 433

Die Inhalte der 2. Auflage wurden in zwei Bände aufgeteilt, um die Bücher handlicher und zielgruppenorientierter zu gestalten. Im Gegensatz zur 2. Auflage ist die 3. Auflage unabhängig von einem Vorgehensmodell. Die Basistechniken und Basiskonzepte sind prozessneutral beschrieben und nach sachlogischen Gesichtspunkten strukturiert – das gilt auch für das *Requirements Engineering*.

Die Methoden SA (*Structured Analysis*) und SA/RT (*Real Time Analysis*) werden nicht mehr behandelt, da ihr Einsatz in der Praxis abnimmt. Das gesamte Gebiet der Software-Ergonomie wird aus Umfangsgründen nicht mehr behandelt – obwohl es zunehmend wichtiger wird. Das Gebiet ist jedoch inzwischen so umfangreich, dass es dazu eigene Lehrbücher gibt und an Hochschulen oft auch in eigenen Lehrveranstaltungen unterrichtet wird.

2 Fallstudien

Zur 3. Auflage

Vorwort zur 3. Auflage



Kapitel aus der 2. Auflage, die nicht mehr in dieser neuen Auflage berücksichtigt werden konnten, finden Sie als *Open Content* auf der Website www.W3L.de/OpenContent.

Buchaufbau

Das Buch besteht aus 28 Kapiteln. Jedes Kapitel besteht aus Unterkapiteln. Am Ende der meisten Kapitel befindet sich eine Zusammenfassung des behandelten Lehrstoffs. Am Ende des Buches sind das umfangreiche Glossar sowie das Literaturverzeichnis angeordnet. Dadurch kann das Buch auch ideal als Nachschlagewerk benutzt werden.

Didaktik &
Methodik

Ziel der Didaktik ist es, einen Lehrstoff so zu strukturieren und aufzubereiten, dass der Lernende sich leicht ein mentales Modell von dem Lehrstoff aufbauen kann und genügend Übungsmöglichkeiten erhält, um zu überprüfen, ob er den Lehrstoff – nach der Beschäftigung mit ihm – entsprechend den vorgegebenen Lernzielen beherrscht. Dieses didaktische Ziel habe ich versucht in diesem Lehrbuch umzusetzen. Die Übungsmöglichkeiten befinden sich jedoch nicht im Lehrbuch, sondern in dem ebenfalls verfügbaren E-Learning-Kurs (siehe unten).

In den meisten Lehrbüchern wird die Welt so erklärt, wie sie ist – ohne dem Lernenden vorher die Möglichkeit gegeben zu haben, über die Welt nachzudenken. Ich stelle daher in vielen Kapiteln am Anfang an Sie eine Frage. Diese Frage soll Sie dazu anregen, über ein Thema nachzudenken. Erst nach dem Nachdenken sollten Sie weiter lesen. (Vielleicht sollten Sie die Antwort nach der Frage zunächst durch ein Papier abdecken).

Querverweise

Die Softwaretechnik ist komplex. Es gibt viele gegenseitigen Abhängigkeiten. Um diese Abhängigkeiten zu verdeutlichen, enthält dieses Buch viele (absolute) Querverweise auf andere Kapitel, damit Sie sich mit verschiedenen Perspektiven auf ein Themengebiet befassen können.

Einsatz des
Buches



Dieses Buch kann zur Vorlesungsbegleitung, zum Selbststudium und zum Nachschlagen verwendet werden. Um den Umfang und die Kosten des Buches zu begrenzen, enthält dieses Buch *keine* Tests und Aufgaben. Für diejenigen Leser unter Ihnen, die Ihr Wissen durch Tests und Aufgaben aktiv überprüfen möchten, gibt es einen (kostenpflichtigen) E-Learning-Online-Kurs. Mentoren und Tutoren betreuen Sie bei der Bearbeitung von Tests und Aufgaben. Das Bestehen eines Abschlusstests und einer Abschlussklausur wird durch Zertifikate dokumentiert. Dieser Online-Kurs ist Bestandteil des Online-Bachelor-Studiengangs »Web- und Medieninformatik« der FH Dortmund. Sie finden den Kurs auf der W3L-Plattform (<http://www.W3L.de>).

Kostenloser
E-Learning-Kurs

Ergänzend zu diesem Buch gibt es den kostenlosen E-Learning-Kurs »Prinzipien der Softwaretechnik«, der zusätzlich zahlreiche Tests enthält, mit denen Sie Ihr Wissen überprüfen können. Sie fin-

Vorwort zur 3. Auflage

den den Kurs auf der E-Learning-Plattform www.W3L.de. Bitte wählen Sie auf dem Reiter »Online-Kurse« den Link »Zur TAN-Einlösung«. Registrieren Sie sich als neuer Benutzer und geben Sie anschließend folgende Transaktionsnummer (TAN) ein: 3645112138.

Dieses Buch ist für folgende Zielgruppen geschrieben:

Zielgruppen

- Studierende der Informatik und Softwaretechnik an Universitäten, Fachhochschulen und Berufsakademien.
- Software-Ingenieure, Softwaremanager und Software-Qualitätssicherer in der Praxis.

Vorausgesetzt werden Kenntnisse, wie sie normalerweise in einer Einführungsvorlesung zur Informatik vermittelt werden.

Vorkenntnisse

Zur Vermittlung der Lerninhalte werden Beispiele und Fallstudien verwendet. Um Ihnen diese unmittelbar kenntlich zu machen, sind sie in blauer Schrift gesetzt.

Beispiele,
Fallstudien

Da ein Bild oft mehr aussagt als 1000 Worte, habe ich versucht, möglichst viele Sachverhalte zu veranschaulichen.

Visualisierung

In diesem Lehrbuch wurde sorgfältig überlegt, welche Begriffe eingeführt und definiert werden. Ziel ist es, die Anzahl der Begriffe möglichst gering zu halten. Alle wichtigen Begriffe sind im Text halbfett und blau gesetzt. Die so markierten Begriffe sind am Ende des Buches in einem Glossar alphabetisch angeordnet und definiert. Dabei wurde oft versucht, die Definition etwas anders abzufassen, als es im Text der Fall war, um Ihnen noch eine andere Sichtweise zu vermitteln.

Begriffe, Glossar
halbfett, blau

Um in einem Buch deutlich zu machen, dass Leser und Leserinnen gemeint sind, gibt es verschiedene Möglichkeiten für den Autor:

Weibliche vs.
männliche Anrede

- 1** Man formuliert Bezeichnungen in der 3. Person Singular in ihrer männlichen Form. In jüngeren Veröffentlichungen verweist man in den Vorbemerkungen dann häufig darauf, dass das weibliche Geschlecht mit gemeint ist, auch wenn es nicht im Schriftbild erscheint.
- 2** Man redet beide Geschlechter direkt an, z. B. Leserinnen und Leser, man/frau.
- 3** Man kombiniert die beiden Geschlechter in einem Wort, z. B. StudentInnen.
- 4** Man wechselt das Geschlecht von Kapitel zu Kapitel.

Aus Gründen der Lesbarkeit und Lesegewohnheit habe ich mich für die 1. Variante entschieden. Die Variante 4 ist mir an und für sich sehr sympathisch, jedoch steigt der Aufwand für den Autor beträchtlich, da man beim Schreiben noch nicht die genaue Reihenfolge der Kapitel kennt.

Vorwort zur 3. Auflage

Als Begleitunterlage & zum Selbststudium	Bücher können als Begleitunterlage oder zum Selbststudium ausgelegt sein. In diesem Buch versuche ich einen Mittelweg einzuschlagen. Ich selbst verwende das Buch als begleitende und ergänzende Unterlage zu meinen Vorlesungen. Viele Lernziele dieses Buches können aber auch im Selbststudium erreicht werden.
Englische vs. deutsche Begriffe	Ein Problem für ein Informatikbuch stellt die Verwendung englischer Begriffe dar. Da die Wissenschaftssprache der Softwaretechnik Englisch ist, gibt es für viele Begriffe – insbesondere in Spezialgebieten – keine oder noch keine geeigneten oder üblichen deutschen Fachbegriffe. Auf der anderen Seite gibt es jedoch für viele Bereiche der Softwaretechnik sowohl übliche als auch sinnvolle deutsche Bezeichnungen, z. B. Entwurf für <i>Design</i> . Da mit einem Lehrbuch auch die Begriffswelt beeinflusst wird, bemühe ich mich in diesem Buch, sinnvolle und übliche deutsche Begriffe zu verwenden. Ist anhand des deutschen Begriffs <i>nicht</i> unmittelbar einsehbar oder allgemein bekannt, wie der englische Begriff lautet, dann wird in Klammern und kursiv der englische Begriff hinter dem deutschen Begriff aufgeführt. Dadurch wird auch das Lesen der englischsprachigen Literatur erleichtert.
Englische Begriffe kursiv gesetzt	Gibt es noch keinen eingebürgerten deutschen Begriff, dann wird der englische Originalbegriff verwendet. Englische Bezeichnungen sind immer <i>kursiv</i> gesetzt, sodass sie sofort ins Auge fallen.
Lesen des Buches: sequenziell	Ziel der Buchgestaltung war es, Ihnen als Leser viele Möglichkeiten zu eröffnen, dieses Buch nutzbringend für Ihre eigene Arbeit einzusetzen. Sie können dieses Buch sequenziell von vorne nach hinten lesen. Die Reihenfolge der Kapitel ist so gewählt, dass die Voraussetzungen für ein Kapitel jeweils erfüllt sind, wenn man das Buch sequenziell liest.
Nach Teildisziplinen	Eine andere Möglichkeit besteht darin, jeweils eine der Teildisziplinen »Basistechniken«, »Basiskonzepte«, » <i>Requirements Engineering</i> « durchzuarbeiten. Auf Querbezüge und notwendige Voraussetzungen wird jeweils hingewiesen.
Themenbezogen	Außerdem kann das Buch themenbezogen gelesen werden. Möchte man sich in die Grundlagen der statischen Basiskonzepte einarbeiten, dann kann man die dafür relevanten Kapitel durcharbeiten. Will man sich auf das <i>Requirements Engineering</i> konzentrieren, dann kann man auch nur diese Kapitel lesen.
Punktuell	Durch das Buchkonzept ist es natürlich auch möglich, punktuell einzelne Kapitel durchzulesen, um eigenes Wissen zu erwerben, aufzufrischen und abzurunden, z. B. Durchlesen des Kapitels über Petrinetze.
Zum Nachschlagen	Durch ein ausführliches Sachregister und Glossar sowie durch Zusammenfassungen kann dieses Buch auch gut zum Nachschlagen verwendet werden.

Vorwort zur 3. Auflage

Soll das Buch begleitend zu einem Softwareprojekt eingesetzt werden, dann kann zunächst der »Requirements Engineering«, S. 433, behandelt werden. Parallel zum Projektverlauf können dann die für das Projekt relevanten Basiskonzepte behandelt werden.

Projekt-
bezogen

Ich habe versucht, ein innovatives wissenschaftliches Lehrbuch der Softwaretechnik zu schreiben. Ob mir dies gelungen ist, müssen Sie als Leser selbst entscheiden.

Ein Buch soll aber nicht nur vom Inhalt her gut sein, sondern Form und Inhalt sollten übereinstimmen. Daher wurde auch versucht, die Form anspruchsvoll zu gestalten. Da ich ein Buch als »Gesamtkunstwerk« betrachte, ist auf der Buchtitelseite ein Bild der Malerin Anna Solecka-Zach abgedruckt. Sie setzt den Computer als Hilfsmittel ein, um ihre künstlerischen Vorstellungen umzusetzen.

Die dynamische Entwicklung der Softwaretechnik und die Themenbreite machen es für einen Autor fast unmöglich, alleine ein Lehrbuch der Softwaretechnik zu schreiben. Ich habe daher einige Kollegen gebeten, Teilgebiete durch eigene Beiträge abzudecken.

Koautoren

Prof. Dr. rer. nat. Rainer Koschke, Universität Bremen, hat das Kapitel zum Thema »Software-Werkzeuge« verfasst:

■ »Werkzeuge«, S. 59

Prof. Dr.-Ing. Uwe Lämmel, Hochschule Wismar, hat ein Kapitel zum Thema »Regelbasierte Systeme« beigetragen:

■ »Regeln«, S. 404

Prof. Dr.-Ing. Peter Liggesmeyer, Technische Universität Kaiserslautern und Institutsleiter des Fraunhofer Instituts für Experimentelles Software Engineering, und seine Mitarbeiter Dr. Robert Eschbach, Dr. Mario Trapp, Johannes Kloos und Bastian Zimmer haben mit der »Fallstudie Fensterheber« und entsprechenden Beispielen in den Basiskonzepten sowie mit dem Kapitel »Formale Logik« zu diesem Buch beigetragen:

■ »Fallstudie: Fensterheber – Die Spezifikation«, S. 117

■ »Fallstudie: Fensterheber – Die fachliche Lösung«, S. 575

■ »Formale Logik«, S. 357

■ »Markov-Ketten«, S. 292

■ »Generalisierte stochastische Petrinetze«, S. 316

Dr.-Ing. Jochen Quante, Universität Bremen, hat das Thema »Aufwandsschätzmethoden« bearbeitet:

■ »Schätzen des Aufwands«, S. 515

Von meiner Frau, Prof. Dr. Heide Balzert, Fachhochschule Dortmund, habe ich aus ihrem »Lehrbuch der Objektmodellierung«, 2. Auflage, eine Reihe von Abschnitten übernommen [Balz05]. Die Boxen mit Checklisten stammen – in etwas abgewandelter Form – ebenfalls aus dem »Lehrbuch der Objektmodellierung«.

Vorwort zur 3. Auflage

Dank Ich danke allen Koautoren für ihre Beiträge zu diesem Lehrbuch. Zusätzlich danke ich Dr.-Ing. Olaf Zwintzsch, Geschäftsführer der W3L GmbH in Witten und Lehrbeauftragter für Softwaretechnik an der Ruhr-Universität Bochum, für die kritische Durchsicht des gesamten Buches und Prof. Dr. Roland Gabriel, Lehrstuhl für Wirtschaftsinformatik an der Ruhr-Universität Bochum, für Anregungen zum Kapitel »Multidimensionale Datenstrukturen«.

Die Grafiken erstellte meine Mitarbeiterin Frau Anja Scharlt. Danke!

Trotz der Unterstützung vieler Personen bei der Erstellung dieses Buches enthält ein so umfangreiches Werk sicher immer noch Fehler und Verbesserungsmöglichkeiten: »*Nobody is perfect*«. Kritik und Anregungen sind daher jederzeit willkommen. Eine aktuelle Liste mit Korrekturen und Informationen zu diesem Buch finden Sie im kostenlosen E-Learning-Kurs zu diesem Buch (siehe oben).

Nach soviel Vorrede wünsche ich Ihnen nun viel Spaß beim Lesen. Möge Ihnen dieses Buch – trotz der manchmal »trockenen« Materie – ein wenig von der Faszination und Vielfalt der Softwaretechnik vermitteln. Werden Sie ein guter Softwareingenieur – Sie werden gebraucht!

Ihr



Inhalt

I	Die Wissenschaftsdisziplin Softwaretechnik	1
1	Was ist Software?	3
2	Warum ist Software so schwer zu entwickeln?	9
3	Was ist Softwaretechnik?	17
II	Basistechniken	23
4	Prinzipien	25
4.1	Prinzip der Abstraktion	26
4.2	Prinzip der Strukturierung	34
4.3	Prinzip der Bindung und Kopplung	37
4.4	Prinzip der Hierarchisierung	38
4.5	Prinzip der Modularisierung	40
4.6	Geheimnisprinzip	42
4.7	Prinzip der Lokalität	45
4.8	Prinzip der Verbalisierung	46
4.9	Abhängigkeiten zwischen den Prinzipien	48
4.10	Zusammenfassung	50
5	Methoden	53
6	Werkzeuge	59
6.1	Menschen, Methoden, Werkzeuge	59
6.2	Klassifikation von Werkzeugen	60
6.2.1	Von Werkzeugen behandelte Artefakte	62
6.2.2	Von Werkzeugen unterstützte Operationen	62
6.2.3	Werkzeuge zur Kollaboration und Kommunikation	73
6.2.4	Unterstützung von Prozessmodellen und Methoden	75
6.3	Integrierte Entwicklungsumgebungen	76
6.4	Modellgetriebene Entwicklung	79
6.5	Auswahlkriterien bei der Anschaffung von Werkzeugen	87
6.6	Evaluationsverfahren für die Anschaffung	90
6.7	Zusammenfassung	97

III Basiskonzepte 99

7 Fallstudie: SemOrg – Die Spezifikation 107

8 Fallstudie: Fensterheber – Die Spezifikation 117

9 Statik 127

- 9.1 Funktionalität 127
- 9.1.1 Einzelne Funktionen 128
- 9.1.2 Zusammenfassung von Funktionen 131
- 9.1.3 Box: Klassen – Methode und Checkliste 137
- 9.2 Funktions-Strukturen 142
- 9.2.1 Funktionsbaum 143
- 9.2.2 Pakete 145
- 9.2.3 Box: Pakete – Methode und Checkliste 148
- 9.2.4 Vererbung 150
- 9.2.5 Box: Vererbung – Methode und Checkliste 155
- 9.2.6 Assoziation 158
- 9.2.7 Box: Assoziationen – Methode und Checkliste 166
- 9.2.8 Box: Multiplizitäten – Methode und Checkliste 169
- 9.2.9 Aggregation und Komposition 171
- 9.2.10 Box: Komposition und Aggregation – Methode und
Checkliste 175
- 9.2.11 Weitere Strukturen 177
- 9.3 Daten 181
- 9.4 Box: Attribute – Methode und Checkliste 187
- 9.5 Daten-Strukturen 190
- 9.5.1 XML, DTD und XML-Schemata 190
- 9.5.2 Entity-Relationship-Modell 199
- 9.5.2.1 ER-Konzepte und OO-Konzepte im Vergleich 200
- 9.5.2.2 Schlüssel, Tabellen und Dateien 204
- 9.5.2.3 Beispiele für semantische Datenmodelle 207
- 9.5.2.4 Unternehmensdatenmodelle und Weltmodelle 209
- 9.5.2.5 Zusammenfassung 213
- 9.5.3 Multidimensionale Datenstrukturen 214
- 9.5.3.1 *Data Warehouse* und *Data Marts* 215
- 9.5.3.2 OLAP und Hyperwürfel 217
- 9.5.3.3 Modellierungsansätze 222
- 9.5.3.4 Zusammenfassung 226

10 Dynamik 227

- 10.1 Kontrollstrukturen 227
- 10.1.1 Die Sequenz 229
- 10.1.2 Die Auswahl 229
- 10.1.3 Die Wiederholung 231
- 10.1.4 Der Aufruf 235

10.1.5	Die Nebenläufigkeit	235
10.1.6	Aktivitätsdiagramm	236
10.1.7	Box: Aktivität – Methode und Checkliste	245
10.1.8	Zusammenfassung	249
10.2	Geschäftsprozesse und <i>Use Cases</i>	250
10.2.1	Konzepte und Notationen	251
10.2.2	EKPs und Aktivitätsdiagramme	253
10.2.3	<i>Use Case</i> -Diagramme und -Schablonen	255
10.2.4	Box: <i>Use Case</i> – Methode und Checkliste	262
10.2.5	Zusammenfassung	268
10.3	Zustandsautomaten	269
10.3.1	Erstellung eines Zustandsautomaten	270
10.3.2	Notationen	272
10.3.3	Zustandsautomat mit Endzuständen	274
10.3.4	Mealy-Automat vs. Moore-Automat	275
10.3.5	Zustandsautomat nach Harel	277
10.3.6	Verhaltens- vs. Protokollzustandsautomaten	287
10.3.7	Markov-Ketten	292
10.3.8	Box: Zustandsautomat – Methode und Checkliste	295
10.3.9	Zusammenfassung	301
10.4	Petrinetze	303
10.4.1	Bedingungs/Ereignis-Netze	305
10.4.2	Stellen/Transitions-Netze	309
10.4.3	Prädikat/Transitions-Netze	311
10.4.4	Hierarchische Petrinetze	312
10.4.5	Zeitbehaftete Petrinetze	314
10.4.6	Generalisierte stochastische Petrinetze	316
10.4.7	Aktivitätsdiagramme und Petrinetze	317
10.4.8	Strukturelemente und Strukturen von Petri-Netzen	320
10.4.9	Box: Petrinetze – Methode	322
10.4.10	Analyse und Simulation von Petrinetzen	327
10.4.11	Wertung	328
10.4.12	Zusammenfassung	330
10.5	Szenarien	332
10.5.1	Sequenzdiagramm	333
10.5.2	Kommunikationsdiagramm	343
10.5.3	Box: Sequenz- und Kommunikationsdiagramm – Methode und Checkliste	346
10.5.4	Timing-Diagramm	352
10.5.5	Zusammenfassung	355
11	Logik	357
11.1	Formale Logik	357
11.1.1	Aussagenlogik	358
11.1.2	Prädikatenlogik	367

Inhalt

11.1.3	Temporale Logik	370
11.1.4	Zusammenfassung	377
11.2	<i>Constraints</i> und die OCL in der UML	377
11.2.1	<i>Constraints</i> in der UML	378
11.2.2	OCL	380
11.2.3	Zusammenfassung	386
11.3	Entscheidungstabellen und Entscheidungsbäume	386
11.3.1	Erstellung einer Entscheidungstabelle	387
11.3.2	Anwendung einer Entscheidungstabelle	389
11.3.3	Überprüfung und Optimierung von Entscheidungstabellen	391
11.3.4	Darstellungsformen für Entscheidungstabellen	393
11.3.5	Entscheidungstabellen-Verbunde	394
11.3.6	Erweiterte Entscheidungstabellen	399
11.3.7	Eintreffer- und Mehrtreffer-Entscheidungstabellen	400
11.3.8	Zusammenfassung & Bewertung	402
11.4	Regeln	404
11.4.1	Aufbau von Regeln	405
11.4.2	Auswahl von Regeln	407
11.4.3	Regelbasierte Software	409
11.4.4	Der Rete-Algorithmus	412
11.4.5	Verkettung von Regeln	414
11.4.6	Lösungssuche	417
11.4.6.1	Der Suchbaum	419
11.4.6.2	Tiefe-zuerst-Suche	420
11.4.6.3	Breite-zuerst-Suche	422
11.4.6.4	Heuristische Suche	425
11.4.7	Bewertete Regeln	426
11.4.8	Geschäftsregeln	428
11.4.9	Anwendungen	429
11.4.10	Zusammenfassung	430

IV *Requirements Engineering* 433

12 *Problem vs. Lösung* 437

13 *Bedeutung, Probleme und Best Practices* 439

14 *Aktivitäten und Artefakte* 443

15 *Der Requirements Engineering-Prozess* 449

16 *Anforderungen und Anforderungsarten* 455

16.1 Visionen und Ziele 456

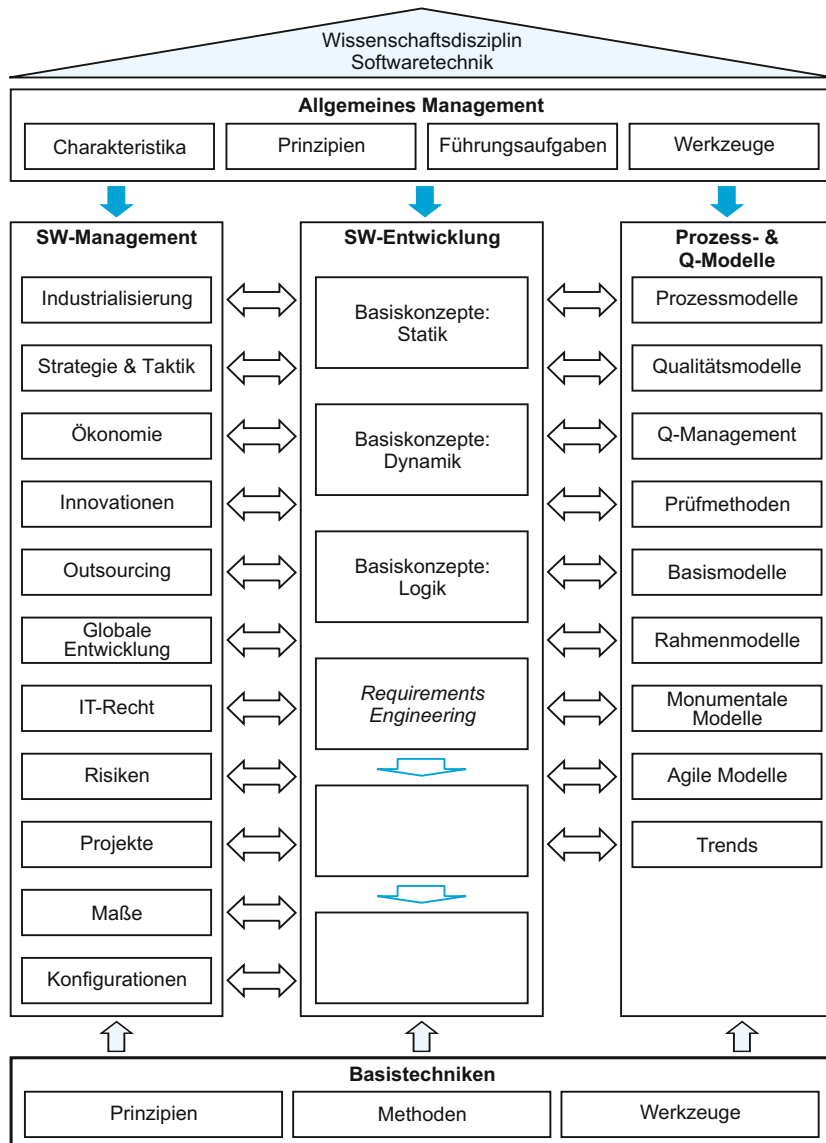
16.2 Rahmenbedingungen 459

16.3	Kontext und Überblick	461
16.4	Nichtfunktionale Anforderungen	463
16.5	Box: Qualitätsmerkmale nach ISO/IEC 9126-1	468
16.6	Abnahmekriterien	471
17	Anforderungen an Anforderungen	475
18	Anforderungsattribute	479
19	Natürlichsprachliche Anforderungen	481
20	Anforderungsschablonen	485
20.1	Anforderungsschablone der IEEE 830-1998	485
20.2	Anforderungsschablonen im V-Modell XT	487
20.3	Schablonen für Lastenheft, Pflichtenheft und Glossar	492
20.4	Schablonen für agile Entwicklungen	497
21	Anforderungen ermitteln und spezifizieren	503
22	Anforderungen analysieren, validieren und abnehmen	513
23	Schätzen des Aufwands	515
23.1	Voraussetzungen und Einflussfaktoren	515
23.2	Warum ist das Schätzen des Aufwands wichtig?	517
23.3	Warum eine Aufwandsschätzung schwierig ist	518
23.4	Schätzverfahren	522
23.4.1	Analogiemethode	523
23.4.2	Expertenschätzung	523
23.4.3	Bottom-up-Methode	524
23.4.4	Prozentsatzmethode	525
23.4.5	Algorithmische Schätzung	526
23.4.6	Faustregeln	526
23.5	Die Function-Points-Methode	527
23.6	<i>Object Points/Application Points</i>	535
23.7	COCOMO II	536
23.8	Bewertung und weitere Aspekte	539
23.9	Zusammenfassung	542
24	Anforderungen priorisieren	543
25	Anforderungen modellieren	547
25.1	Beispiel: Objektorientierte Analyse	548
25.1.1	Strukturierung der OOA-Konzepte	549
25.1.2	OOA-Muster	550

Inhalt

25.1.3	OOA-Methode	559
25.2	Domänenspezifische Sprachen	563
26	Fallstudie: SemOrg V1.0 – Die fachliche Lösung	565
27	Fallstudie: Fensterheber – Die fachliche Lösung	575
28	Modellierte Anforderungen analysieren, verifizieren und abnehmen	587
Glossar 589		
Literatur 605		
Sachindex 619		

I Die Wissenschaftsdisziplin Softwaretechnik



I I Die Wissenschaftsdisziplin Softwaretechnik

Die Wissenschaftsdisziplin Softwaretechnik gehört zur Kerninformatik und dort zur Praktischen Informatik, besitzt aber auch viele Bezüge zu Nachbarwissenschaften. Jede Wissenschaftsdisziplin lässt sich im Wesentlichen durch ihren Gegenstandsbereich, ihre Erkenntnisziele und ihre eingesetzten Methoden und Konzepte beschreiben.

i Um den Gegenstandsbereich zu definieren, wird daher zunächst der Begriff Software näher betrachtet:

■ »Was ist Software?«, S. 3

Die Erkenntnisziele ergeben sich zum Teil aus den Problemen der Softwareentwicklung:

■ »Warum ist Software so schwer zu entwickeln?«, S. 9

Nach diesen Vorbetrachtungen lässt sich dann die Disziplin Softwaretechnik genauer definieren:

■ »Was ist Softwaretechnik?«, S. 17

1 Was ist Software?

Grundsätzlich ist festzuhalten, dass ein Kernstück jeder Wissenschaft eine eigene einheitliche und allgemein anerkannte Begriffswelt ist. Wegen der starken Praxisorientierung der Softwaretechnik spielt die Begriffsbildung hier sogar eine herausragende Rolle.

Die hohe Innovationsgeschwindigkeit und die Praxisnähe behindern eine solide, konsistente und systematische Begriffsbildung. Daher gibt es heute für die Softwaretechnik noch keine allgemein anerkannte und klar definierte Terminologie.

Jeder Begriffssystematik liegt eine bestimmte Sicht auf die Softwaretechnik zugrunde. Hat man eine andere Sicht, dann ist die Begriffssystematik zumindest teilweise ungeeignet. Auf der anderen Seite sollte man eingebürgerte Begriffe – auch wenn sie einer Systematik zuwiderlaufen – nicht durch neue, theoretisch bessere Begriffe versuchen zu »überschreiben«. Man erhöht dadurch die Missverständnisse noch, anstatt sie zu reduzieren.

Glücklicherweise ist man davon abgekommen, den Begriff Software durch einen »deutschen« Begriff zu ersetzen.

Wie würden Sie den Begriff »Software« definieren?

Zu dem Begriff Software gibt es eine Reihe von Definitionen.

»**Software** (engl., eigtl. »weiche Ware«), Abk. SW, Sammelbezeichnung für Programme, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation« [Broc88].

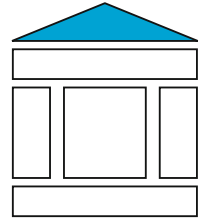
»**Software** (kurz **SW**): Menge von →Programmen oder Daten zusammen mit begleitenden Dokumenten, die für ihre Anwendung notwendig oder hilfreich sind« [HKL+84, S. 204].

»**software**: (1) **Computer programs, procedures, rules, and possibly associated documentation and data** pertaining to the operation of a **computer system** [...]« (IEEE Standard Glossary of Software Engineering Terminology [ANSI83, S. 31]).

Die Definitionen zeigen, dass Software ein umfassenderer Begriff als Programm ist. Außerdem gehören zu Software immer auch eine entsprechende Dokumentation und evtl. zugehörige Daten.

In Zusammenhang mit dem Begriff Software tauchen auch die Begriffe Softwareprodukt und Softwaresystem auf. Wie würden Sie diese Begriffe definieren und evtl. voneinander abgrenzen?

»Ein **Produkt** ist ein in sich abgeschlossenes, i.a. für einen Auftraggeber bestimmtes Ergebnis eines erfolgreich durchgeführten Projekts oder Herstellungsprozesses. Als Teilprodukt bezeichnen wir einen abgeschlossenen Teil eines Produkts. [...]«



Frage

Antwort

Frage

Antwort

I 1 Was ist Software?

SW-Produkt: Produkt, das aus →Software besteht. [...]

Unter einem *System* wird ein Ausschnitt aus der realen oder gedanklichen Welt, bestehend aus Gegenständen (z. B. Menschen, Materialien, Maschinen oder anderen Produkten) und darauf vorhandenen Strukturen (z. B. deren Aufbau aus Teileinheiten oder Beziehungen untereinander) verstanden.« [HKL+84, S. 204 ff.]

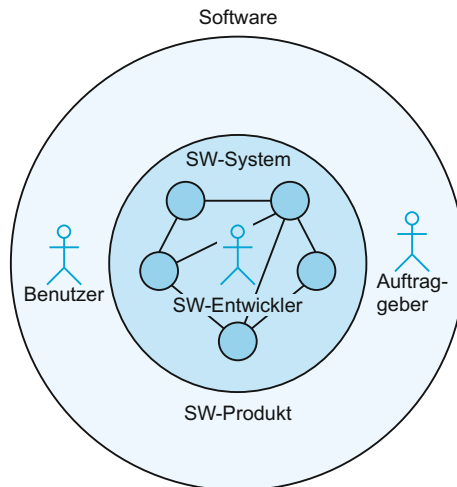
Systemteile, die nicht weiter zerlegbar sind oder zerlegt werden sollen, werden als Systemelemente bezeichnet [HBB+94].

Ein **Softwaresystem** ist dementsprechend ein System, dessen Systemkomponenten und Systemelemente aus Software bestehen.

Betrachtet man die drei Begriffe Software, Softwareprodukt und Softwaresystem im Zusammenhang, dann kann man feststellen, dass **Software** der allgemeinere, neutrale aber auch unbestimmtere Begriff ist.

Softwareprodukt betrachtet Software von »außen«, aus Käufer- oder Auftraggebersicht, während Softwaresystem die »innere« Sichtweise darstellt, d. h. so, wie ein Entwickler Software sieht (Abb. 1.0-1). »**software product**. A **software** entity designated for delivery to a user« [ANSI83, S. 32].

Abb. 1.0-1:
Software,
Softwareprodukt,
und
Softwaresystem.



Soll die spezifische Sicht auf Software *nicht* besonders betont werden, dann wird in diesem Buch der Begriff Software verwendet.

Softwaresysteme gliedert man oft noch in Anwendungssoftware und Systemsoftware.

System-SW

Systemsoftware, auch Basissoftware genannt, ist Software, die für eine Computer-**Plattform**, eine spezielle Hardware oder eine Hardwarefamilie entwickelt wurde, um den Betrieb und die Wartung dieser Hardware zu ermöglichen bzw. zu erleichtern. Eine Plattform ist gekennzeichnet durch den jeweils verwendeten Prozessor-

typ und das eingesetzte Betriebssystem. Zur Systemsoftware zählt man immer das Betriebssystem, in der Regel aber auch Compiler, Datenbanken, Kommunikationsprogramme und spezielle Dienstprogramme. Systemsoftware orientiert sich grundsätzlich an den Eigenschaften der Hardware, für die sie geschaffen wurde und ergänzt normalerweise die funktionalen Fähigkeiten der Hardware.

Anwendungssoftware (*application software*), auch Applikationssoftware genannt, ist Software, die Aufgaben des Anwenders mit Hilfe eines Computersystems löst. Anwendungssoftware setzt in der Regel auf der Systemsoftware der verwendeten Hardware auf bzw. benutzt sie zur Erfüllung der eigenen Aufgaben. Im Mittelpunkt dieses Buches steht die Anwendungssoftware.

Anwendungs-SW

Anwendungssoftware, Systemsoftware und Hardware bilden zusammen ein Computersystem.

Als **Anwender** werden alle Angehörigen einer Institution oder organisatorischen Einheit bezeichnet, die ein Computersystem zur Erfüllung ihrer fachlichen Aufgaben einsetzen. Sie benutzen die Ergebnisse der Anwendungssoftware oder liefern Daten, die die Anwendungssoftware benötigt.

Anwender

Benutzer sind nur diejenigen Personen, die ein Computersystem unmittelbar einsetzen und bedienen, oft auch Endbenutzer oder Endanwender genannt.

Benutzer

Folgende weitere Begriffe werden häufig verwendet:

Ein **technisches System** setzt sich aus dem Computersystem und sonstigen technischen Einrichtungen zusammen.

Technisches System

Bei einem Computersystem kann es sich zum Beispiel um einen Bordrechner handeln, bei einem technischen System um ein Flugzeug. Ein Zugangskontrollsystem besteht aus einem Ausweisleser (Computersystem) und der Tür mit dem Schließmechanismus (sonstige technische Einrichtung). Der Ausweisleser selbst besteht aus einem Rechner im Ausweisleser (Hardware) und der Software zum Datenabgleich Ausweis-Authentisierung. Bei dem Zugangskontrollsystem handelt es sich also um ein technisches System.

Beispiel

Die Mitarbeiter in ihrer Rolle als Aufgabenträger einschließlich Anwendern und Benutzern werden oft als **organisatorisches System** bezeichnet.

Organisatorisches System

Im Zusammenhang mit dem Begriff System ist oft noch die Unterscheidung zwischen Systementwicklung und Softwareentwicklung wesentlich. Worin besteht Ihrer Meinung nach der Unterschied?

Frage

Spricht man von Softwareentwicklung, dann meint man die ausschließliche Entwicklung von Software. Spricht man von Systementwicklung, dann meint man oft die Entwicklung eines Systems, das

Antwort

I 1 Was ist Software?

aus Hardware- und Software-Komponenten besteht. In einem solchen Fall müssen bei der Entwicklung zusätzliche Randbedingungen berücksichtigt werden.

In diesem Buch wird eine Systementwicklung, die eine Hardware-Entwicklung mit berücksichtigt, *nicht* betrachtet.

Eine wichtige wirtschaftliche Bedeutung haben heute **software-intensive Systeme**. In einem softwareintensiven oder softwarebasierten System werden wesentliche Eigenschaften durch die in das System eingebettete Software realisiert. Im Gegensatz zu reinen Softwarelösungen ist bei softwareintensiven Systemen die Software nur ein – wenn auch wichtiger – Systembestandteil. Softwareintensive Systeme werden in diesem Buch ebenfalls behandelt. Auf die besonderen Herausforderungen bei softwareintensiven Systemen wird in [Broy06] näher eingegangen.

Komplexitäts-
arten

Anwendungen lassen sich nach Komplexitätsarten gliedern. Sechs Dimensionen der Software-Komplexität können unterschieden werden:

■ Komplexität der Funktionen:

Softwaresysteme, die eine Vielzahl von Funktionen enthalten. Textverarbeitungssysteme, DTP-Systeme und integrierte Büroprogramme sind Beispiele dafür. Integrierte Büroprogramme mit Textverarbeitung, Tabellenkalkulation, Bürografik und Datenbank besitzen zwischen 1.000 und 1.500 Funktionen. Ein DTP-System hat einen Funktionsumfang von 700 bis 1.000 Funktionen.

■ Komplexität der Daten:

Softwaresysteme, die eine Vielzahl von Datenstrukturen oder sehr komplexe Datenstrukturen enthalten. Ein Beispiel hierfür sind Datenbanksysteme.

■ Komplexität der Algorithmen:

Softwaresysteme, die z. B. komplexe numerische Berechnungen durchführen.

■ Komplexität des zeitabhängigen Verhaltens:

Softwaresysteme, die sich durch nebenläufige Prozesse, gegenseitigen Ausschluss, Synchronisation, definierte Zeitbedingungen und ähnliches auszeichnen. Beispiele dafür sind Betriebssysteme, verteilte Systeme, Prozesssteuerungen, softwareintensive Systeme.

■ Komplexität der Systemumgebung:

Softwaresysteme, die in andere Systeme eingebettet sind (softwareintensive Systeme, *embedded systems*). Da diese Softwaresysteme nur Teilkomponenten des Gesamtsystems sind, kommt es hierbei ganz wesentlich auf das Zusammenwirken von Softwaresystem und Gesamtsystem an. Beispiele hierfür sind Flugzeugsteuerungen, Radaranlagen, Kraftwerkssteuerungen.

■ **Komplexität der Benutzungsoberfläche:**

Softwaresysteme mit komplexer Interaktion zwischen Benutzer und Computersystem. Beispiele dafür sind CAD- und CASE-Systeme, Spiele, Büroanwendungen, Grafiksysteme, aber auch moderne Web-Anwendungen.

Anwendungssoftware lässt sich nach diesen Komplexitätsarten klassifizieren:

- **Administrative, kaufmännische, betriebswirtschaftlich orientierte Software** ist gekennzeichnet durch eine Komplexität der Daten und/oder der Funktionen.
- **Technisch-wissenschaftliche Software** ist gekennzeichnet durch die Komplexität der Funktionen und der Algorithmen.
- **Softwareintensive Systeme** sind gekennzeichnet durch die Komplexität der Algorithmen, der Systemumgebung und des zeitabhängigen Verhaltens.
- **GUI-intensive Software** ist gekennzeichnet durch die Komplexität der Benutzungsoberfläche und -interaktion.

2 Warum ist Software so schwer zu entwickeln?

Seit Software entwickelt wird, beklagt man sich darüber, wie schwierig diese Tätigkeit ist. Immer wieder ist von der »Software-Krise« die Rede. Um diese Schwierigkeiten zu verstehen, muss man sich zunächst die Charakteristika von Software näher ansehen.

Außerdem muss man analysieren, wie sich Software in den letzten 30 Jahren entwickelt hat, um zu begreifen, dass, trotz wesentlicher Fortschritte in der Softwaretechnik, noch viel zu tun ist.

Überlegen Sie, durch welche Charakteristika sich Software von anderen technischen Produkten unterscheidet.

Software unterscheidet sich durch folgende Charakteristika von anderen technischen Produkten:

- Software ist ein immaterielles Produkt.

Software kann man nicht »anfassen«, nicht »sehen«. Das, was die Dokumentation beschreibt, stimmt oft nicht hundertprozentig mit dem lauffähigen Code der Software überein. Diese Immaterialität hat massive Konsequenzen für die Softwaretechnik und insbesondere für das Softwaremanagement. Der Entwicklungsfortschritt ist nicht objektiv zu ermitteln.

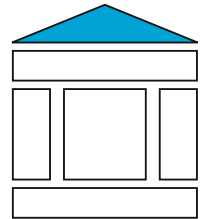
- Software unterliegt *keinem* Verschleiß.

Technische Produkte nutzen sich im Laufe der Zeit ab. Software kann beliebig oft ablaufen, ohne dass Abnutzungserscheinungen auftreten. Daher ist eine klassische Wartung von Software, bei der es darum ginge, verschlissene Teile auszutauschen, nicht notwendig.

- Software altert.

Auf den ersten Blick erscheint die Aussage unglaublich, zumal gerade festgestellt wurde, dass Software keinem Verschleiß unterliegt. Beachtet man jedoch, dass sich die Umgebung, in der eine Software eingesetzt wird, ständig ändert, dann wird diese Aussage verständlich. Software, die nicht ständig an die sich ändernde Umgebung angepasst wird, altert und ist irgendwann veraltet, d. h. sie kann nicht mehr für den ursprünglich vorgesehenen Zweck eingesetzt werden. Beispielsweise läuft sie auf einer neuen Hardware-Plattform nicht mehr oder sie kann nicht über das Web benutzt werden.

- Für Software gibt es *keine* Ersatzteile.



Frage

Antwort

I 2 Warum ist Software so schwer zu entwickeln?

Bei technischen Produkten werden Produktteile, die verschlissen oder fehlerhaft sind, ausgetauscht und durch in der Regel vorproduzierte Ersatzteile ersetzt. Für Software geht dies nicht. Es würde bedeuten, Software-Komponenten mehrfach zu entwickeln. Tritt ein Fehler in einer Komponente auf, dann müsste man darauf hoffen, dass in der ausgetauschten Komponente nicht zufällig derselbe Fehler auftaucht. Außerdem müssten die Schnittstellen und ihre Semantik exakt definiert sein. Experimente haben gezeigt, dass Entwicklungsteams, die Komponenten völlig unabhängig voneinander entwickeln, zwar nicht die exakt gleichen Fehler machen, aber ähnliche Probleme erzeugen. Das liegt daran, dass der Mensch aufgrund seiner mentalen Fähigkeiten dazu neigt, in bestimmte Fehlerfallen zu »tappen«.

- Software ist im Allgemeinen leichter und schneller änderbar als ein technisches Produkt.

Um technische Produkte zu ändern, müssen oft erst neue Werkzeuge hergestellt werden. Ist Software gut strukturiert und modularisiert, dann können Änderungen schnell und einfach durchgeführt werden.

- Software ist schwer zu »vermessen«.

Technische Produkte kann man mit Hilfe von Messgeräten in der Regel sehr exakt vermessen. Anhand der Maße können die Produkte dann sowohl untereinander als auch mit einem Standard oder einer Vorgabe verglichen werden. Das ist bei Software nur bedingt möglich. Erstens ist die Qualität für Software schwer definierbar und quantifizierbar. Zweitens ist die Korrelation zwischen dem, was man messen kann, und dem, was als Qualität gefordert ist, nicht oder nur in Ansätzen bekannt.

Veränderungen Software hat sich in den letzten 20 Jahren in verschiedener Hinsicht gravierend verändert, sodass die Fortschritte in der Softwaretechnik *nicht* ausreichen, um mit den Veränderungen der Software Schritt zu halten.

Frage Wenn Sie sich schon längere Zeit mit der Softwaretechnik befassen, welche Änderungen stellen Sie fest?

Antwort Die wesentlichen Veränderungen sind:

- Zunehmende Bedeutung
- Wachsende Komplexität
- Zunehmende Qualitätsanforderungen
- Nachfragestau und Engpassfaktor
- Mehr Standardsoftware
- Zunehmend »Altlasten«
- Zunehmend »Außer-Haus«-Entwicklung

2 Warum ist Software so schwer zu entwickeln? I

Diese Veränderungen sind nicht isoliert voneinander zu sehen, sondern sie stehen in Wechselwirkung und bedingen sich teilweise gegenseitig. Im Folgenden werden diese Veränderungen etwas näher betrachtet.

Zunehmende Bedeutung

Die große Bedeutung der Software verdeutlicht folgendes Zitat:

Zitat

»Software ist der fundamentale *Werkstoff des Informationszeitalters*. Innovative Produkte und Dienstleistungen sind ohne Software nicht mehr denkbar. Die Wettbewerbsfähigkeit der deutschen Wirtschaft hängt entscheidend von der Fähigkeit ab, Software-intensive Produkte und Dienstleistungen höchster Qualität zu erstellen. [...] Software wird in der Zukunft *integrierter* – in vielen Fällen sogar *dominierender* – Teil großer komplexer Systeme sein« [BJN+06, S. 210].

Die Abb. 2.0-1 zeigt als Beispiel den zunehmenden Anteil der Software im Automobilbereich. In einem PKW gehobener Ausstattung betrug im Jahr 2003 die Größe eingebetteter Software 70 MB, bis zum Jahr 2010 rechnet man mit 1 GB [BJN+06, S. 216].

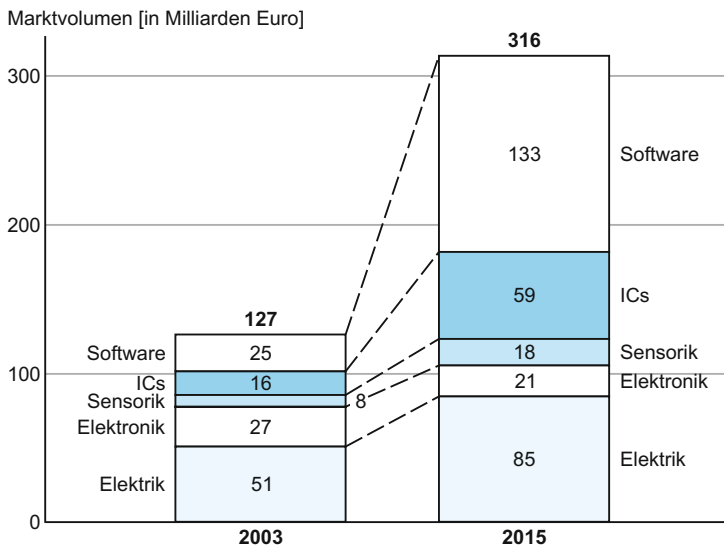


Abb. 2.0-1:
Marktvolumen von
Software im
Automobilbereich
für das Jahr 2003
mit der Prognose
für 2015 (in
Anlehnung an
[Hons05]).

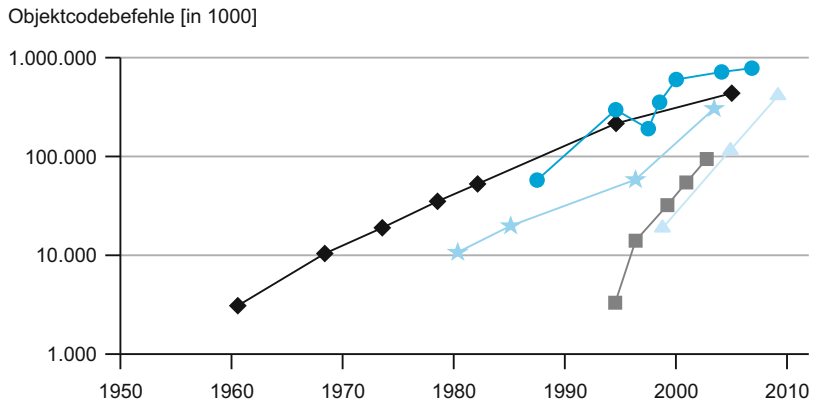
Wachsende Komplexität

Die Bedeutung der Software hat auch deshalb so stark zugenommen, weil Software in immer neuen Anwendungsgebieten eingesetzt wird und immer schwierigere Aufgaben unterstützt. Komplexere Aufgaben erfordern auch komplexere und umfangreichere Software. Die

I 2 Warum ist Software so schwer zu entwickeln?

Abb. 2.0-2 zeigt, wie sich der Umfang der Software für die Raumfahrt, für Vermittlungssysteme, für eingebettete Systeme und für Betriebssysteme entwickelt hat.

Abb. 2.0-2:
Zunehmender
Umfang von
Softwaresystemen
[Eber08, S.2].



Legende:

- ◆ Raumfahrt (Space Flight Control)
- ★ Vermittlungssysteme
- ▲ Eingebettete Software im Auto
- Windows-Betriebssystem
- Linux-Betriebssystem (Kernel)

In den 70er und 80er Jahren verdoppelte sich der Umfang nach 5–7 Jahren. Heute verdoppelt sich der Umfang bereits alle 2–4 Jahre, d. h. die Verdopplungsrate hat sich verdoppelt.

Trend Ein Trend der letzten 10 bis 20 Jahre ist die zunehmende **Multifunktionalität** von Systemen verbunden mit einer wachsenden **Heterogenität** der assoziierten technischen Komponenten [Broy06, S. 73]. Die Zunahme multifunktionaler Geräte zeigt folgendes Beispiel: Im Jahr 2005 wurden zum ersten Mal mehr in Handys integrierte Kameras verkauft als eigenständige Kameras.

Zunehmende Qualitätsanforderungen

Zitat »Für die Wertschöpfung im Produktionsgüter- und Dienstleistungsbereich ist Software Engineering die "Produktionstechnik des 21. Jahrhunderts" [...] Die künftigen neuen Produkte werden ohne hochgradige Qualitätszusicherungen, die nur durch *qualitätsorientierte Entwicklungs- und Produktionsprozesse* erreicht werden, nicht erfolgreich sein« [BJN+06, S. 212].

Nachfragestau und Engpassfaktor

»Es gibt einen geschätzten Bedarf von 385.000 Softwareentwicklern in Deutschland [Zahlen von 2002, Anm. des Autors]. Auch bei vorsichtiger Bewertung dieser Zahl ergibt sich daraus, dass ein weiterer Ausbau der Softwaretechnik an deutschen Universitäten dringend erforderlich ist [...] Durch den schnellen technologischen Wandel, gerade im Bereich Software Engineering, besteht ein wachsender Bedarf an kontinuierlicher *Weiterbildung* und auch eine steigende Nachfrage nach der Zertifizierung von Software-Engineering-Kompetenzen« [BJN+06, S. 215 f.] Zitat

Nach Schätzung des Branchenverbandes BITKOM liegt der Bedarf an Informatikabsolventen von Hochschulen bei etwa 20.000 pro Jahr. Nach Angaben des Statistischen Bundesamts schlossen 2006 15.400 Absolventen ihr Informatikstudium erfolgreich ab. Daraus ergibt sich eine jährliche Lücke von ca. 5000 Informatikabsolventen pro Jahr, das ist eine Lücke von ca. 25 Prozent pro Jahr.

Der inzwischen schon hohe Grad der Akademisierung nimmt weiter zu. Ein Drittel aller beschäftigten IT-Fachleute hat einen Hochschulabschluss. Bei allen Beschäftigten liegt dagegen der Akademikeranteil knapp unter 10 Prozent. Außerdem arbeiten nur wenige in der Informatik in Teilzeit (7 Prozent verglichen mit 17 Prozent bei allen Beschäftigten im Jahr 2006) [Hohn07].

Nach Erhebungen von BITKOM sind 2009 in Deutschland 820.000 Personen in der IT beschäftigt. Zwei Drittel davon sind Hochschulabsolventen. 2009 fehlen 45.000 IT-Fachleute in Deutschland (Quelle: Computer Zeitung, 12.1.2009, S. 21).

Mehr Standardsoftware

Der Nachfragestau bei Software, aber auch die hohen Kosten der Individualsoftware haben dazu geführt, dass der Anteil der Standardsoftware ständig zunimmt.

Für Standardsoftware spricht:

- Kann dort sinnvoll eingesetzt werden, wo die Anforderungen weitgehend standardisiert sind, wie z.B. im Rechnungswesen oder in der Finanzbuchhaltung.
- Individuelle Software, die mit konventionellen Werkzeugen entwickelt wird, ist *nicht* mehr bezahlbar.
- Individualsoftware ist in der Regel fehleranfällig, da zum ersten Mal eingesetzt.

I 2 Warum ist Software so schwer zu entwickeln?

Für Individualsoftware spricht:

- Die Kosten, um die Standardsoftware an die Firmenspezifika anzupassen, übersteigen deutlich die Kosten der Standardsoftware. In den meisten Fällen wird dann aus Kostengründen das Anwendungsunternehmen an die Standardsoftware angepasst und nicht umgekehrt.
- Vorhandene Anwendungen können nur schwer in die Standardsoftware integriert werden.
- Manche Branchen sind zu klein für Standardlösungen. Im Extremfall gibt es nur ein oder ganz wenige Unternehmen in einer Branche, z. B. nur wenige große Touristikanbieter.

Trend Trenduntersuchungen prognostizieren, dass der Anteil der Individualsoftware bis auf fünf Prozent sinken wird und dass sich solche Individualsoftware nur große Firmen leisten können. Aus diesem Trend ergibt sich, dass der Anteil vollständig neu geschriebener Software abnimmt. Zunehmend wichtiger werden Anpassungen.

Zunehmend »Altlasten«

In den letzten Jahrzehnten wurde viel Software entwickelt. Anwendungssoftware wird oft 20 Jahre und länger eingesetzt. Da sich die Einsatzumgebung dieser Anwendungssoftware ständig ändert, muss diese Software ebenfalls ständig angepasst werden. Diese permanenten Anpassungsprozesse verursachen oft 2/3 aller Software-Kosten. Bei diesen »Altlasten« stellt sich immer wieder die Frage, ob eine weitere Sanierung möglich und ökonomisch sinnvoll ist oder ob eine Ablösung durch ein neues Softwaresystem erforderlich ist.

Die Idee, sämtliche »alte Software« durch »neue Software« zu ersetzen, hat sich aus Kostengründen als *nicht* realisierbar herausgestellt. Oft wird daher alte Software »eingepackt« in eine neue Programmierschnittstelle (*Wrapping*), sodass z. B. moderne Benutzungsoberflächen realisiert werden können. Der alte Kern der Software wird aber weiterhin benutzt.

Trend Durch die zunehmende Verbreitung von Software werden die »Altlasten« weiter zunehmen, denn die Softwareprodukte von heute sind die Altlasten von morgen.

Zunehmend »Außer-Haus«-Entwicklung

Während früher Software in größeren Firmen durch eigene Software-Abteilungen entwickelt wurde, zeichnet sich seit einigen Jahren ein Trend ab, Software *nicht* selbst zu entwickeln, sondern bei Software-Häusern im In- und/oder Ausland in Auftrag zu geben (Outsourcing).

2 Warum ist Software so schwer zu entwickeln? I

Es wird davon ausgegangen, dass von den Softwareprodukten und den zugehörigen Dienstleistungen generell etwa 55 % intern und 45 % extern erbracht werden. Durch die zunehmende Produktintegration von Software (eingebettete Systeme) wird der Prozentsatz intern erstellter Software nicht drastisch zurückgehen.

intern 55 % extern
45 %

»[...] wenn es um den Fremdbezug von Anwendungen geht, dann liegt Software aus deutschen Ländern weit vorne [...] Mehr als 90 Prozent setzen heimische Software ein « (Quelle: Computer Zeitung, 11.3.2009, S. 11).

Zitat

»Je nachdem wie sich der Mix aus lokalen und Offshore-Ressourcen ergibt, können wir in typischen Global Sourcing-Projekten eine Reduzierung der Kosten in Höhe von 30 Prozent erreichen [...] Das gelte vor allem für Projekte, die eine Laufzeit von mindestens sechs Monaten haben und dabei zehn Mitarbeiter binden« (Quelle: Computer Zeitung, 11.3.2009, S. 11).

Offshoring

Zitat

3 Was ist Softwaretechnik?

Bevor ich eine eigene Definition vorschlage, möchte ich einige publizierte Definitionen aufführen.

»Unter **Softwaretechnik** (engl. *Software Engineering*) versteht man allgemein die (Ingenieur-) Wissenschaft, die die kosteneffiziente Entwicklung von qualitativ hochwertiger Software behandelt« (Fachgruppe Softwaretechnik der Gesellschaft für Informatik [GI08]).

»**software engineering**:

(1) *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*

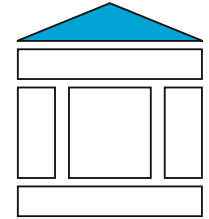
(2) *The study of approaches as in (1)*« (IEEE Standard Glossary of Software Engineering Terminology [IEEE90]).

»**Softwaretechnik, Softwaretechnologie, Software-Engineering** das, –, Teilgebiet der Informatik, das sich mit Methoden und Werkzeugen für das ingenieurmäßige Entwerfen, Herstellen und Implementieren von Software befasst« [Broc08].

Softwaretechnik (syn.: **Software Engineering**): Fachgebiet der Informatik, das sich mit der Bereitstellung und systematischen Verwendung von Methoden und Werkzeugen für die Herstellung und Anwendung von Software beschäftigt (Ein Begriffssystem für die Softwaretechnik [HKL+84, S. 204]).

In Erweiterung der letzten Definition erscheint folgende Definition das Gebiet am besten zu charakterisieren:

Softwaretechnik: Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen. Zielorientiert bedeutet die Berücksichtigung z. B. von Kosten, Zeit, Qualität.



Definition

Diese Definition betont wichtige Charakteristika der Softwaretechnik. Es handelt sich um umfangreiche Software, die arbeitsteilig und ingenieurmäßig entwickelt wird, wobei die Ziele des Software-Kunden zu erreichen sind. Diese Charakteristika treffen *nicht* zu, wenn beispielsweise ein Einzelner ein Programm entwickelt und selbst anwendet.

Umfangreiche SW

Bei der Softwaretechnik handelt es sich um eine Ingenieurdisziplin. In der Regel gibt es nicht »die« Lösung für ein Problem, sondern Ziel ist, marktorientiert den optimalen Kompromiss zu finden. Ingenieurmäßig bedeutet hier auch »kein Künstlertum«.

Ingenieur-
Disziplin

I 3 Was ist Softwaretechnik?

Nach über 40 Jahren Softwaretechnik – die erste Softwaretechnik-Tagung fand 1968 in Garmisch-Partenkirchen statt – gibt es etablierte und bewährte Prinzipien, Methoden und **Best Practices**. Diese können und sollen nicht von jedem Software-Entwickler neu erfunden werden. Es geht heute nicht darum, künstlerisch Software zu entwickeln, sondern die vorhandenen Erkenntnisse systematisch anzuwenden.

In der obigen Definition erscheinen die Begriffe Prinzip, Methode und Werkzeug. Diese Begriffe habe ich unter dem Oberbegriff Basistechniken zusammengefasst. Sie werden im Buchteil II definiert und erläutert:

■ »Basistechniken«, S. 23

Methoden geben an, welche Konzepte wie und wann verwendet werden, um die festgelegten Ziele zu erreichen.

Konzepte

Das Verb »konzipieren« bedeutet »eine Grundvorstellung von etwas entwickeln; verfassen, entwerfen.« Es wurde dem lateinischen Verb »capere« entlehnt, das »nehmen, fassen; begreifen« bedeutet (Duden-Herkunftswörterbuch).

Konzepte erlauben es also, definierte Sachverhalte unter einem oder mehreren Gesichtspunkten zu beschreiben, zu spezifizieren, zu modellieren.

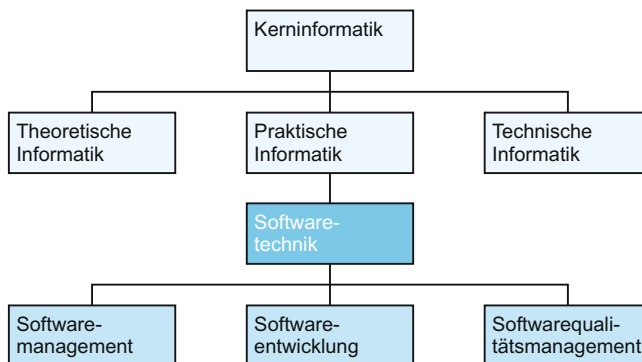
In der Softwaretechnik stehen inzwischen eine ganze Reihe von Basiskonzepten zur Verfügung, die im Buchteil III vorgestellt werden:

■ »Basiskonzepte«, S. 99

Softwaretechnik als Teilgebiet der Informatik

Die Softwaretechnik ist eine Teildisziplin der Informatik. Die Kerninformatik gliedert man in Theoretische, Praktische und Technische Informatik (Abb. 3.0-1). Die Softwaretechnik zählt zur Praktischen Informatik.

Abb. 3.0-1: Die Softwaretechnik innerhalb der Informatik.



Der Gegenstandsbereich der Softwaretechnik wird unterschiedlich breit definiert:

Breite vs. enge
Definition

- »Die *breite Definition* ordnet der Softwaretechnik alle Themengebiete bzw. Technologien zu, die für die Funktionsweise heute üblicher Softwaresysteme relevant sind. Da hierbei fast immer Betriebssysteme, Netzwerke, Datenbanken usw. eine Rolle spielen, umfängt das Themengebiet hier völlig aus und ist mehr oder weniger identisch mit der Praktischen Informatik.
- Die *fokussierte Definition* ordnet der Softwaretechnik in erster Linie Themen wie die Analyse von Anforderungen, die Gestaltung, Programmierung und Architektur von Programmen, Entwicklungsprozesse und Qualitätssicherung zu. Nicht als Teil der Softwaretechnik oder nur als Randbereich werden Technologien für Betriebssysteme, Netzwerke, Datenbanksysteme und andere wichtige Komponenten von Softwaresystemen verstanden; diese Technologien haben zu eigenen Fachgebieten der Praktischen Informatik geführt. Eine saubere Abgrenzung zu diesen Nachbargebieten ist aber oft schwierig.

Die breite Definition liegt u.a. Begriffen wie 'Software-Ingenieur' oder Studiengängen mit der Bezeichnung 'Softwaretechnik' zugrunde« [GI08]. In diesem Buch wird von der fokussierten Auffassung ausgegangen.

Die Softwaretechnik selbst lässt sich in drei Teildisziplinen gliedern:

Gliederung der
SWT

- Die Softwareentwicklung,
- das Softwaremanagement und
- das Softwarequalitätsmanagement.

Die beiden letzten Disziplinen werden in dem »Lehrbuch der Softwaretechnik – Softwaremanagement (Band 3)« behandelt.

Die (technische) **Softwareentwicklung** hat die Aufgabe, ein Produkt zu planen, zu definieren, zu entwerfen und zu realisieren, das die geforderten Qualitätseigenschaften besitzt und die Kundenwünsche erfüllt. Nach der Softwareerstellung befindet sich das Softwareprodukt in der Anwendung und muss gepflegt und gewartet werden. Wartung bedeutet, ein nach der Inbetriebnahme auftretendes Fehlverhalten zu beseitigen. Pflege bedeutet, ein Produkt an geänderte Bedingungen anzupassen oder aufgrund neuer oder geänderter Anforderungen weiterzuentwickeln.

SW-Entwicklung

Eine Softwareentwicklung läuft nicht von selbst ab, sondern **Softwaremanagement** ist erforderlich, um den technischen Entwicklungsprozess zu planen, zu organisieren, zu leiten und zu kontrollieren. Da Softwareentwicklungen oft in Form von so genannten Projekten durchgeführt werden, spricht man oft auch von Softwareprojektmanagement, obwohl dies nur eine Teilmenge des Softwaremanagements ist.

SW-Management

I 3 Was ist Softwaretechnik?

- SW-QM Die Sicherstellung einer geforderten Softwarequalität muss durch ein entwicklungsbegleitendes **Softwarequalitätsmanagement** (QM), früher oft Softwarequalitätssicherung genannt (QS), erreicht werden. Dazu sind eine ganze Reihe von konstruktiven und analytischen Maßnahmen durchzuführen.
- Abgrenzung »Das Software Engineering grenzt sich zu seinen *Grundlagendisziplinen* (Kerninformatik, Ingenieurwissenschaften, Mathematik, Psychologie, Soziologie, Betriebswirtschaft) dadurch ab, dass es sich vor allem auf Methoden konzentriert und dabei immer den Anspruch praktischer Anwendbarkeit vor Augen hat. [...] Weder zu den Grundlagen- noch zu den Anwendungsfeldern hin ist eine scharfe Abgrenzung möglich; Software Engineering ist inhärent *transdisziplinär*« [BJN+06, S. 211].
- Zitat
- Systemisch Softwaretechnik ist heute eine systemische Disziplin, d.h. eine Disziplin mit vielen Wechselwirkungen zu anderen Disziplinen.

Ist Softwaretechnik eine Wissenschaft?

- Die Softwaretechnik ist ohne Zweifel eine Ingenieurdisziplin. Aber ist sie auch eine Wissenschaft? [Prec99, Kapitel 3.1.1] schreibt dazu:
- Zitat »Im Mittelpunkt der Arbeit steht im Ingenieurwesen das konkrete Lösen eines praktischen Problems [...]. Deshalb bleibt meist keine Zeit, dabei zugleich die Grundlagen dieser Arbeit, nämlich das wissenschaftliche Wissen über das Fach, grundsätzlich zu verbessern. Eine Wissenschaft hat hingegen nur die Produktion und Organisation von Wissen zum Ziel [...]. Für eine Wissenschaft ist es unerheblich, ob die Produktion von Wissen überwiegend Selbstzweck ist, wie beispielsweise die Klassifikation von Pflanzen in der Biologie, oder stark zweckgerichtet wie in den meisten Teilen der Informatik. Bei Betrachtung der Wissenschaftsgeschichte stellt man ohnehin fest, daß fast sämtliche Wissenschaften in weiten Teilen zweckgerichtet waren und sind. [...] Zu jedem Ingenieurfach muß es eine zugehörige Wissenschaft geben, damit das Ingenieurwesen größere Fortschritte machen kann. Alle obigen Überlegungen gelten [...] auch insbesondere für die Softwaretechnik. Bei der Softwaretechnik tritt allerdings besonders der Aspekt der Interdisziplinarität mit in den Vordergrund: Da überall der Mensch so eng eingebunden ist, bekommen Überschneidungen mit der Psychologie, Soziologie und anderen Fächern, die sich mit dem Menschen befassen, eine verglichen mit anderen Teilen der Informatik besonders hohe Bedeutung. Jedenfalls hat die Softwaretechnik-Forschung die Aufgabe, wissenschaftliche Ergebnisse zu liefern, die den Wissensschatz vergrößern, auf den Softwareingenieure zur Lösung ihrer Probleme zurückgreifen. Da die Softwaretechnik eine Ingenieurwissenschaft ist, werden die Ergebnisse an ihrer Nützlichkeit gemessen.«

3 Was ist Softwaretechnik? I

Weiterführende Betrachtungen zum Gegenstand einer Wissenschaft und zur Entwicklung einer Wissenschaftsdisziplin enthält z. B. [Dahm00].

Literatur

Zu dem Grad, inwieweit die Softwaretechnik heute bereits eine Wissenschaft ist, resümiert [Prec99, Kapitel 3.1.3]:

»Die Softwaretechnik als praktische technische Disziplin hat heute wenig wissenschaftlichen Charakter. Das vorhandene Wissen stammt überwiegend aus einer Mischung von Intuition und Empirizismus und ist nur in wenigen Fällen wissenschaftlich abgesichert, denn die meisten Beiträge sind Entwürfe (von Modellen, Methoden oder Werkzeugen) und stellen insofern nur Hypothesen dar – und selbst diese werden meist nicht klar ausgesprochen. Geprüft werden diese Hypothesen bislang nur selten; nicht zuletzt deshalb, weil eine solche Prüfung in der Softwaretechnik meist enorm aufwendig ist. So wissen wir beispielsweise herzlich wenig darüber, welche Eigenschaften von z. B. Entwurfsmethoden, Programmiersprachen oder Entwicklungswerkzeugen welche Qualitätsattribute der Produkte in welcher Weise beeinflussen, obwohl diese Fragen den Kern unseres Faches betreffen.«

Zitat

Im letzten Jahrzehnt hat sich die medizinische Praxis als Ergebnis des evidenzbasierten Paradigmas dramatisch geändert. Unter einer evidenzbasierten Medizin (EbM) versteht man, dass jeder Arzt sich an empirisch belegten Ergebnissen aus der medizinischen Forschung orientieren soll und nicht an bloßer Expertenmeinung. Analog wird eine **evidenzbasierte Softwaretechnik** gefordert: EBSE (*Evidence-Based Software Engineering*).

Evidenzbasiert

»EBSE aims to improve decision making related to software development and maintenance by integrating current best evidence from research with practical experience and human values« [DKJ05, S. 59].

Zitat

Trotz dieser Mängel lässt sich Folgendes feststellen:

»Software kann Dinge leisten, die vormals nicht denkbar waren. Das allein spricht dafür, dass die Softwareentwicklung in den letzten Jahrzehnten enorm erfolgreich war, auch in der Art und Weise, wie Software gebaut wurde« [Dene08, S. 33].

Zitat

In den letzten Jahren geht der Trend hin zu einer **wertbasierten Softwaretechnik** (*Value-Based Software Engineering* – VBSE). Im Gegensatz zur wertneutralen Softwaretechnik, bei der alle Techniken und Konzepte gleich wichtig sind, werden bei der wertbasierten Softwaretechnik immer betriebswirtschaftliche Gesichtspunkte berücksichtigt. »The value-based approach to software development integrates value considerations into current and emerging software engineering principles and practices,[...]« [BoHu03].

Wertbasiert

Die Reife einer Wissenschaftsdisziplin spiegelt sich auch in ihrer Terminologie wider. Fachbegriffe müssen definiert werden, um eine effektive Kommunikation unter den Fachexperten erst möglich zu

Begriffe

I 3 Was ist Softwaretechnik?

machen. Da es sich bei der Softwaretechnik noch um eine junge Disziplin handelt, ist auch die Begriffswelt noch nicht stabil – aber sie wird immer stabiler und konsistenter. Ein Arbeitskreis der »Gesellschaft für Informatik« (GI) hat eine Begriffssammlung erstellt und zu einem kohärenten Begriffsnetz zusammengefügt: Website Begriffsnetz Informatik (<http://www.informatikbegriffsnetz.de/>) (siehe auch [BFH+07]). Am bekanntesten im englischsprachigen Raum ist das *IEEE Standard Glossary of Software Engineering Terminology* von 1990: Website IEEE SE Glossary (<http://w3.umh.ac.be/genlog/SE/SE-contents.html>).

Gliederung der Softwareentwicklung

Im Rahmen einer Softwareentwicklung sind eine ganze Reihe von Aktivitäten von vielen Personen koordiniert und systematisch durchzuführen. Es gibt eine Vielzahl von sogenannten Prozess- und Qualitätsmodellen, die festlegen, wie vorzugehen ist. Ein Überblick über diese Modelle wird in dem »Lehrbuch der Softwaretechnik – Softwaremanagement (Band 3)« gegeben.

Spezifizieren &
Konstruieren

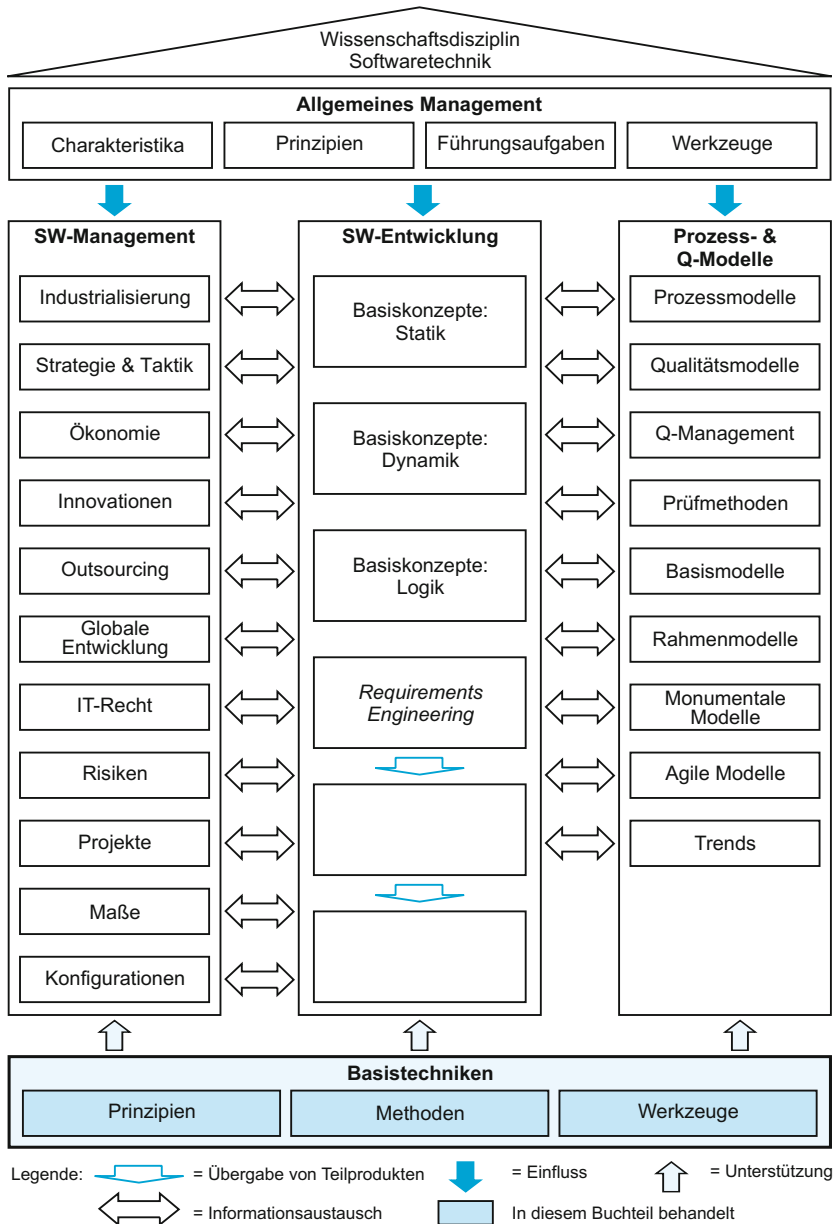
Will man eine grobe Untergliederung der Aktivitäten in der Softwareentwicklung vornehmen, die auch unterschiedliche Qualifikationen erfordern, dann kann man Spezifizieren und Konstruieren in der Softwareentwicklung unterscheiden.

Zum **Spezifizieren** zählen alle Aktivitäten, die ausgehend von den Kundenwünschen hin zu einer fachlichen Lösung führen. Alle diese Tätigkeiten werden unter dem englischen Begriff **Requirements Engineering** subsumiert – in Deutschland oft als Systemanalyse bezeichnet. Dieses Teilgebiet der Softwareentwicklung wird im Buchteil IV behandelt:

■ »Requirements Engineering«, S. 433

Ausgehend von der fachlichen Problemlösung wird beim **Konstruieren** die technische Problemlösung entwickelt, in Form von Programmen realisiert und beim Kunden installiert, gewartet und gepflegt. Dieses Teilgebiet der Softwareentwicklung wird im »Lehrbuch der Softwaretechnik, Band 2« behandelt.

II Basistechniken



II II Basistechniken

Die Softwaretechnik ist gekennzeichnet durch eine hohe Innovationsgeschwindigkeit. Insbesondere im Bereich der Methoden, Werkzeuge und *Best Practices* vergeht kaum eine Woche oder ein Monat, in dem nicht neue Methoden und neue Werkzeuge angekündigt werden. In dem rasanten Wandel fällt es schwer, das »Konstante«, das »Übergeordnete« zu erkennen. Im Folgenden werden einige Prinzipien, Methoden und Werkzeugkonzepte vorgestellt, die eine gewisse »Allgemeingültigkeit« für die Softwaretechnik besitzen – zusammengefasst unter dem Oberbegriff »Techniken«. Sie gelten auch in vielen anderen Disziplinen. Anhand von Beispielen werden ihre Relevanz und »Gültigkeit« für die Softwaretechnik gezeigt.

i Der Begriff »Prinzip« wird heute in vielfältigen Varianten verwendet. Umgangssprachlich spricht man von einem »Prinzipienreiter«, wenn eine Person gemeint ist, die an festen Regeln und Grundsätzen festhält. Auch von der Softwaretechnik lassen sich eine Reihe von Grundsätzen identifizieren:

■ »Prinzipien«, S. 25

Wie in jeder Wissenschaftsdisziplin so sollte auch in der Softwaretechnik immer »methodisch« vorgegangen werden:

■ »Methoden«, S. 53

Werkzeuge sind Hilfsmittel, die einem bei der Erledigung von Arbeiten unterstützen. Eine professionelle Softwareentwicklung ist ohne Werkzeuge nicht möglich. Da der Werkzeugmarkt jedoch sehr in Bewegung ist, wird nur ein Überblick über die prinzipiellen Unterstützungsmöglichkeiten durch Werkzeuge gegeben sowie auf typische Werkzeuge einer Kategorie hingewiesen:

■ »Werkzeuge«, S. 59

4 Prinzipien

Was ist für Sie ein Prinzip? Nach welchen Prinzipien handeln Sie? Welche Prinzipien fallen Ihnen zur Softwaretechnik ein?

Frage

Prinzipien sind Grundsätze, die man seinem Handeln zugrunde legt. Prinzipien sind allgemeingültig, abstrakt, allgemeinsten Art. Sie bilden eine theoretische Grundlage. Prinzipien werden aus der Erfahrung und Erkenntnis hergeleitet und durch sie bestätigt. Folgende Prinzipien werden vorgestellt:

Antwort

- »Prinzip der Abstraktion«, S. 26
- »Prinzip der Strukturierung«, S. 34
- »Prinzip der Bindung und Kopplung«, S. 37
- »Prinzip der Hierarchisierung«, S. 38
- »Prinzip der Modularisierung«, S. 40
- »Geheimnisprinzip«, S. 42
- »Prinzip der Lokalität«, S. 45
- »Prinzip der Verbalisierung«, S. 46

Die Prinzipien sind nicht unabhängig voneinander:

- »Abhängigkeiten zwischen den Prinzipien«, S. 48

Alle diese Prinzipien spielen für den Softwareentwicklungsprozess, insbesondere für die Spezifikation, den Entwurf und die Implementierung, eine wichtige Rolle. Aber auch in den Bereichen Softwaremanagement und Softwarequalitätsmanagement finden diese Prinzipien ihre Anwendung, oft in etwas modifizierter Form. Es gibt noch weitere Prinzipien, die aber meist nur spezielle Anwendungsbereiche abdecken. Sie sind in den entsprechenden Kapiteln aufgeführt.

Der **Wirkungsbereich** der einzelnen Prinzipien wird anhand des Systembegriffs erläutert.

Ein **System** ist ein Ausschnitt aus der realen oder gedanklichen Welt, bestehend aus Systemkomponenten bzw. Subsystemen, die untereinander in verschiedenen Beziehungen stehen.

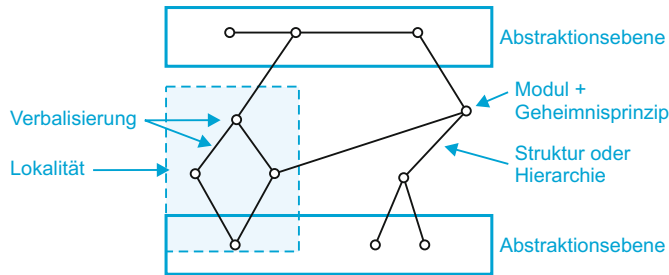
Definition

Systemteile, die nicht weiter zerlegbar sind oder zerlegt werden sollen, werden als **Systemelemente** bezeichnet [HBB+94].

Systeme können in allgemeiner Form durch Graphen dargestellt werden. Ein Graph stellt die Systemkomponenten bzw. Subsysteme durch (markierte) Knoten(punkte) und die Beziehungen (Relationen) durch verbindende (benannte) Linien (Kanten) dar. Die Abb. 4.0-1 zeigt ein Beispiel einer solchen grafischen Systemdarstellung und den Wirkungsbereich der einzelnen Prinzipien.

II 4 Prinzipien

Abb. 4.0-1:
Prinzipien und ihr
Wirkungsbereich.



Die Kontextabhängigkeit und der innere Aufbau einer Systemkomponente geben den Grad der Modularisierung an. Systemkomponenten, die sich auf demselben Abstraktionsniveau befinden, können zu Abstraktionsebenen zusammengefasst werden. Das Prinzip der Lokalität sorgt dafür, dass alle benötigten Informationen, die für eine Sichtweise oder für einen Zweck benötigt werden, lokal, d. h. räumlich nahe angeordnet sind. Gute Verbalisierung sorgt für die prägnante und fachlich geeignete Benennung von Systemkomponenten und Beziehungen.

Die Eigenschaften der Beziehungen bzw. Relationen zwischen den Systemkomponenten bestimmen die Struktur oder Hierarchie des Systems. Die Abb. 4.0-2 zeigt relevante Relationen und ihre Eigenschaften. Durch Graphen können beliebige Strukturen dargestellt werden. Gerichtete Graphen schränken die Strukturen ein. Weitere Einschränkungen ergeben sich, wenn die den Graphen zugrunde liegenden Relationen Ordnungsrelationen sind (Abb. 4.0-3).

4.1 Prinzip der Abstraktion

Die Abstraktion ist eines der wichtigsten Prinzipien der Softwaretechnik.

Frage Was verstehen Sie unter Abstraktion? Was ist das Gegenteil von Abstraktion?

**Antwort
Definition**

Unter **Abstraktion** versteht man Verallgemeinerung, das Absehen vom Besonderen und Einzelnen, das Loslösen vom Dinglichen. Abstraktion ist das Gegenteil von Konkretisierung.

Unter **Abstrahieren** versteht man dementsprechend das Abgehen vom Konkreten, das Herausheben des Wesentlichen aus dem Zufälligen, das Beiseite lassen von Unwesentlichem, das Erkennen gleicher Merkmale.

Abstrakt **Abstrakt** bedeutet also *nicht* gegenständlich, *nicht* konkret, *nicht* anschaulich, begrifflich verallgemeinert, theoretisch.

4.1 Prinzip der Abstraktion II

Die Eigenschaften der Relationen zwischen Systemkomponenten führen zu einer bestimmten Struktur oder Hierarchie eines Systems.

Definitionen

Das **kartesische Produkt** $A \times B$ zweier Mengen A, B besteht aus allen geordneten Paaren (a, b) mit $a \in A$ und $b \in B$.

Jede Teilmenge $R \subseteq A \times B$ heißt (zweistellige, binäre) Relation von A nach B , im Falle $R \subseteq A \times A$ (zweistellige, binäre) Relation in A . Ist $(a, b) \in R$, so schreibt man auch aRb .

Zuordnungen

Zweistellige Relationen in A können gedeutet werden als Zuordnungen:

Elementen aus A werden nach einer definierten Vorschrift (Ordnungsrelation " \leq ",

Gleichheitsrelation "=", Enthalten sein-Relation " \subseteq " im Mengensystem)

Elemente aus A zugeordnet. Elementpaare sind in der Relation, wenn sie dieser Vorschrift genügen.

Eigenschaften

Es sei R eine Relation in A . Dann heißt R

■ **reflexiv**, wenn für alle $a \in A$ aRa gilt,

■ **irreflexiv**, wenn für alle $a \in A$ aRa *nicht* gilt,

■ **symmetrisch**, wenn für alle $a, b \in A$ mit aRb folgt, dass auch bRa gilt,

■ **asymmetrisch**, wenn für alle $a, b \in A$ mit aRb folgt, dass bRa *nicht* gilt,

■ **identitiv**, wenn für alle $a, b \in A$ mit aRb und bRa folgt, dass $a=b$ gilt,

■ **transitiv**, wenn für alle $a, b, c \in A$ mit aRb und bRc folgt, dass (auch) aRc gilt,

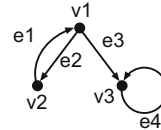
■ **konnex** (linear), wenn für alle $a, b \in A$ folgt, dass aRb oder bRa (oder beides) gilt.

Transitivität und Irreflexivität implizieren Asymmetrie. Die Eigenschaften der **Transitivität** liegen allen Ordnungsstrukturen zu Grunde.

Graphen

Die Relationen aus obiger Definition lassen sich eindeutig und anschaulich durch **gerichtete Graphen** darstellen.

Ein **gerichteter Graph** $G = (V, E)$ besteht aus einer Menge von Knoten $V = \{v_1, v_2, v_3, \dots\}$ und einer Menge von Kanten $E = \{e_1, e_2, e_3, \dots\}$.



Beispiel für einen gerichteten Graphen

Die Zuordnung einer Relation $R \subseteq A \times A$ zu einem gerichteten Graphen $G = (V, E)$ geschieht derart, dass alle $a \in A$ Knoten v in G werden und die Knoten v_1, v_2 , die je zwei Elemente $a, b \in A$ mit aRb entsprechen, durch eine (gerichtete) Kante (Pfeil) e von v_1 nach v_2 verbunden werden. Dabei wird oft vereinbart, dass die sich aus der Transitivität einer Relation ergebenden Pfeile unterdrückt werden. Anstelle $G = (V, E)$ schreibt man deshalb auch $G = (V, R)$.

v_1 nennt man den Ausgangsknoten, v_2 den Zielknoten der Kante e . Unter Ordnungsaspekten heißt v_1 (direkter) **Vorgänger** von v_2 und v_2 (direkter) **Nachfolger** von v_1 . Eine Folge von Knoten (v_1, v_2, \dots, v_n) heißt nun Pfad der Länge n , wenn es eine (gerichtete) Kante von v_i nach v_{i+1} für $i = 1, \dots, n-1$ gibt. v_n kann dann von v_1 aus erreicht werden.

Ein **Zyklus** (Schleife) ist ein Pfad (v_1, \dots, v_n) für den $v_1 = v_n$ gilt. Insbesondere ist (v_1, v_1) ein Zyklus der Länge 1. Schleifenfreie Graphen heißen **azyklisch**.

Die Zahl der Kanten, die in einem Knoten enden, heißt **Eingangsgrad**, die Zahl der Kanten, die aus einem Knoten entspringen, heißt **Ausgangsgrad**.

Abb. 4.0-2:
(Mathematische)
Relationen und
ihre
Eigenschaften.

Oft spricht man anstelle von Abstraktion auch von **Modellbildung**. Durch das Abstrahieren vom Konkreten erstellt man ein Modell der realen Welt, d.h. das Modell repräsentiert die reale Welt durch sein charakteristisches Verhalten.

»Modeling is the most important engineering technique; models help us to understand and analyze large and complex problems«
[Kram07, S. 41].

Modellbildung

Zitat

II 4 Prinzipien

Ungerichteter Graph

Beispiel: Assoziationen im ER-Modell

Gerichteter Graph

Beispiele: Assoziationen in UML-Klassendiagrammen, Kontrollstrukturen ohne Nebenläufigkeit, Petrinetze

Gerichteter azyklischer Graph mit partieller Ordnung

Eine **transitive** und **irreflexive** Relation R in einer Menge A heißt **partielle Ordnung** über A . Es gilt: Eine partielle Ordnung ist asymmetrisch. Wenn R partielle Ordnung über A ist, so ist $G = (V, R)$ ein gerichteter azyklischer Graph.

Rekursive Strukturen können nicht beschrieben werden: Die Irreflexivität verbietet direkte Rekursion, die zusammen mit der Transitivität herleitbare Asymmetrie indirekte Rekursion. Im Gegensatz zu Baumhierarchien (siehe unten) kann der Eingangsgrad von Knoten größer als 1 sein, d.h. es können von unterschiedlichen Zweigen Relationen zu einem Objekt bestehen. Beispiele: Mehrfachvererbung, Aggregation in UML-Klassendiagrammen

Gerichteter Graph mit reflexiver partieller Ordnung

Eine **transitive**, **reflexive** und **identitive** Relation R in einer Menge A heißt **reflexive partielle Ordnung** über A .

Gerichteter Graph mit linearer Ordnung

Eine **transitive**, **reflexive**, **identitive** und **konnexe** Ordnung R über A heißt **lineare Ordnung** von A .

Beispiel: Abstraktionsschichten mit linearer Ordnung

Gerichteter Graph mit strikter Ordnung

Eine **transitive** und **asymmetrische** Relation über A heißt **strikte Ordnung** über A .

Beispiel: Abstraktionsschichten mit strikter Ordnung

Gerichteter Baum

Ein (gerichteter) **Baum** ist ein gerichteter Graph $G = (A, R)$ mit (genau) einem speziellen Knoten $w \in A$, der sog. **Wurzel**, mit den folgenden Eigenschaften:

- 1 w hat Eingangsgrad 0, d.h. keine Kante endet in w ,
- 2 alle anderen Knoten haben den Eingangsgrad 1, d.h. in ihnen endet genau eine Kante,
- 3 jeder Knoten kann von w aus erreicht werden.

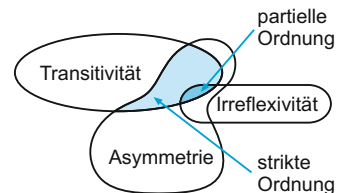
Ein Baum ist demnach azyklisch; der Pfad von w zu einem anderen Knoten ist eindeutig bestimmt.

Bei einer Baumstruktur haben alle Systemkomponenten einen eindeutigen **Abstand** von der Wurzel. Der Abstand ist dabei die Länge des minimalen Pfades.

Beispiele: Einfachvererbung, geschachtelte Pakete, XML-Dokumente, XML-Schemata

Ordnungsrelationen im Überblick

- **partielle Ordnung**: transitiv, irreflexiv, asymmetrisch.
- **reflexiv partielle Ordnung**: transitiv, reflexiv, identitiv.
- **vollständige Ordnung**: transitiv, reflexiv, konnex (nicht relevant).
- **lineare Ordnung**: transitiv, reflexiv, identitiv, konnex.
- **strikte Ordnung**: transitiv, asymmetrisch.



Hierarchie vs. Struktur

Gerichtete Graphen erzeugen Hierarchien, ungerichtete und/oder gerichtete Graphen erzeugen Strukturen.

Abb. 4.0-3:
Relevante Ordnungsrelationen.

Abstraktion und Konkretisierung sind nicht absolut, sondern relativ, d. h. es gibt mehr oder weniger Abstraktion und Konkretisierung. Daher wird der Begriff **Abstraktionsebenen** benutzt, um Abstufungen der Abstraktion zu bezeichnen.

4.1 Prinzip der Abstraktion II

Abstraktion und Modellierung werden ausführlich in [InLu83], [Luft84] und [Kram07] behandelt. [Literatur](#)

»Vom Konkreten zum Abstrakten« vs. »Vom Abstrakten zum Konkreten«

Schwierigkeiten entstehen oft dadurch, dass nicht die geeigneten Abstraktionen gefunden werden. Man sollte sich darüber im Klaren sein, dass das Abstrahieren eine äußerst anspruchsvolle Tätigkeit ist, denn es ist meist sehr schwierig, aus vielen konkreten Tatsachen das Wesentliche zu isolieren.

Bei der Softwareentwicklung findet ein ständiges Wechselspiel zwischen »Abstrahieren« und »Konkretisieren« statt. Am Anfang einer Softwareentwicklung steht in der Regel ein Abstraktionsprozess (Abb. 4.1-1).

Beispiel

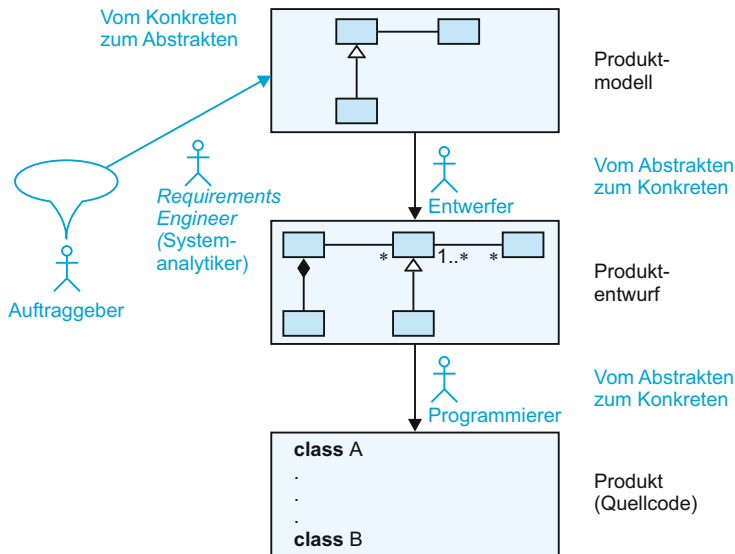


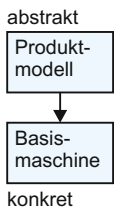
Abb. 4.1-1: »Vom Konkreten zum Abstrakten« und »Vom Abstrakten zum Konkreten« in der Softwareentwicklung.

Die Auftraggeber- bzw. Kundenwünsche müssen ermittelt, spezifiziert, analysiert und in Form einer abstrakten Beschreibung modelliert werden. Die Anforderungen an ein neues Produkt sind dabei meist unvollständig, zum Teil widersprüchlich, oft fallbezogen und bestehen aus einer Vielzahl von konkreten Details. Der *Requirements Engineer* bzw. Systemanalytiker muss »intellektuell« in der Lage sein, richtig zu abstrahieren, um vom »Konkreten« zum »Abstrakten« zu gelangen.

II 4 Prinzipien

Werden Mitarbeiter für diese Tätigkeit eingesetzt, die diese Fähigkeit *nicht* besitzen, dann liegt der weiteren Entwicklung ein falsches oder ungeeignetes Modell zugrunde. Ist ein Produktmodell erstellt worden, dann ist dieses der Ausgangspunkt für die »Konkretisierungen« im weiteren Softwareentwicklungsprozess. Im Entwurf und der Implementierung wird das ursprüngliche Produktmodell bis auf die Ebene der verwendeten Programmiersprache konkretisiert. Vom Entwerfer und Programmierer wird daher die Fähigkeit erwartet, vom »Abstrakten« zum »Konkreten« hin zu denken.

Da viele Mitarbeiter nicht gleich gut »Vom Konkreten zum Abstrakten« und »Vom Abstrakten zum Konkreten« denken können, ist eine Mitarbeiterspezialisierung für diese unterschiedlichen Tätigkeiten zu empfehlen.



Nach Erstellung des Produktmodells besteht also die Aufgabe der Softwareentwicklung darin, die Kluft zwischen dem abstrakten Produktmodell und der konkreten Basismaschine durch die schrittweise Konkretisierung des Produktmodells zu überbrücken. Die Basismaschine repräsentiert in der Regel die vom Betriebssystem und der verwendeten Programmiersprache bzw. deren Laufzeitsystem bereitgestellte Funktionalität.

Drei Abstraktionsebenen

In der objektorientierten Softwareentwicklung lassen sich prinzipiell folgende drei Abstraktionsebenen unterscheiden [HBK+94]:

- Die Exemplar-Ebene,
- die Typ-Ebene und
- die Meta-Ebene, genauer gesagt die Meta-Typen-Ebene.

Auf der **Exemplar-Ebene** werden menschliche Handlungen, spezifische Ereignisse, konkrete Sachverhalte in ihren Beziehungen und/oder zeitlichen Abläufen beschrieben. Passive Elemente, auf die Bezug genommen wird, sind konkrete Gegenstände, Personen oder begriffliche Artefakte wie »Haus Nr. 25«, »Guido Neumann« oder »Konto 2324«. Beziehungen verbinden diese miteinander, z. B. »Guido Neumann wohnt in Haus Nr. 25« oder »Guido Neumann besitzt Konto 2324«. Aktive Elemente sind einmalige, konkrete Handlungen, Aktivitäten, das Eintreten von Ereignissen wie »Guido Neumann zieht in Haus Nr. 25 ein«, »Guido Neumann eröffnet Konto 2324«.

Artefakt =
Kunsterzeugnis;
das durch
menschliches
Können
Geschaffene

Die **Typ-Ebene** erlaubt generalisierende Aussagen über Elemente der Exemplar-Ebene. Die passiven Elemente sind Typen von Gegenständen, Personen oder begrifflichen Artefakten wie »Haus«, »Kunde«, »Konto«. Die Klassifizierung gleichwertiger Beziehungen führt zu Beziehungstypen wie »Kunde besitzt Konto«. Durch die standar-

disierte Beschreibung von Handlungen, Aktivitäten oder des Eintretens von Ereignissen wie »Konto eröffnen«, entstehen aktive Elemente der Typ-Ebene.

Die **Meta-Ebene** entsteht durch die Zusammenfassung gleichartiger Elemente der Typ-Ebene. Die passiven Elemente »Haus«, »Kunde« und »Konto« werden unter dem Begriff »Klasse« subsumiert, »Attribute« fassen unstrukturierte passive Elemente, »Operationen« aktive Elemente zusammen. Die Elemente der Meta-Ebene dienen hauptsächlich zur Begriffsbildung auf der Typ-Ebene. Außerdem ist die **Meta-Ebene** für die Werkzeugunterstützung relevant. Viele Softwarewerkzeuge arbeiten intern mit einem Meta-Modell, ebenso basiert die **UML** auf einem Metamodell. Bei der UML (*Unified Modeling Language*) handelt es sich um eine grafische Modellierungssprache mit textuellen Annotationen, die in der Softwareentwicklung der heutige Industriestandard ist (siehe »Basiskonzepte«, S. 99, [OMG09b], [OMG09a]).

Alle drei Ebenen findet man in der objektorientierten Welt: Objektdiagramm (Exemplar-Ebene), Klassendiagramm (Typ-Ebene), Metamodell (Meta-Ebene) (Abb. 4.1-2).

Beispiel

Unabhängig von den objektorientierten Abstraktionen gibt es orthogonal dazu die **Abstraktionen der generischen Typen**, erstmals in der Programmiersprache Ada (1983) enthalten. Bei **generischen Typen** können Typparameter stellvertretend für konkrete Typen in Algorithmen verwendet werden. Dadurch können Algorithmen unabhängig von Typen geschrieben werden. Bei der Anwendung werden dann den Typparametern konkrete Typen zugeordnet. Neben der Typparametrisierung können Typen noch eingeschränkt werden.

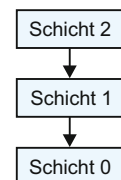
Generische Typen

Die Objektorientierung kann mit generischen Typen kombiniert werden. Dadurch entstehen mächtige Abstraktionen (Abb. 4.1-4). Beispielsweise ist es möglich, Typparameter mit Typeinschränkung bei Methoden und Klassen anzugeben (rechte Spalte der Abb. 4.1-4). Diese generischen Konzepte wirken in der Objektorientierung auch bei der Vererbung von Klassen und der Vererbung/Implementierung von Schnittstellen. In der UML gibt es für typparametrisierte Klassen eine eigene Notation (Abb. 4.1-5). Für »Abstraktionen« gibt es sonst in der Regel *keine* eigenständige Notation.

OO & generische Typen

Das verwendete Konzept, z. B. Klassen, die damit verbundene(n) Notation(en), z. B. die UML-Notation, und der Einsatz in der jeweiligen Entwicklungsphase geben indirekt das Abstraktionsniveau an. Zur Darstellung von Abstraktionsebenen bzw. -schichten wird oft pro Ebene ein Rechteck verwendet. Die Rechtecke sind getrennt

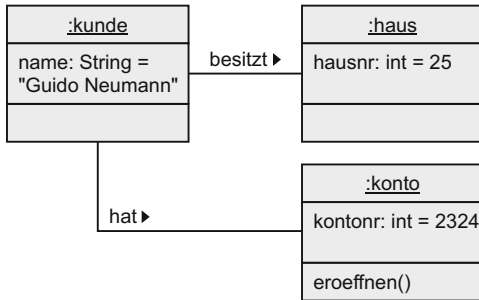
Zur Notation



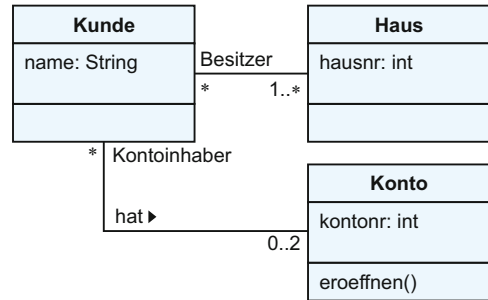
II 4 Prinzipien

durch Pfeile übereinander angeordnet, wobei das am höchsten oben angeordnete Rechteck die »abstrakteste« Schicht darstellt. Es ergibt sich eine lineare Ordnung (siehe »Prinzipien«, S. 25).

Exemplar-Ebene



Typ-Ebene



Meta-Ebene

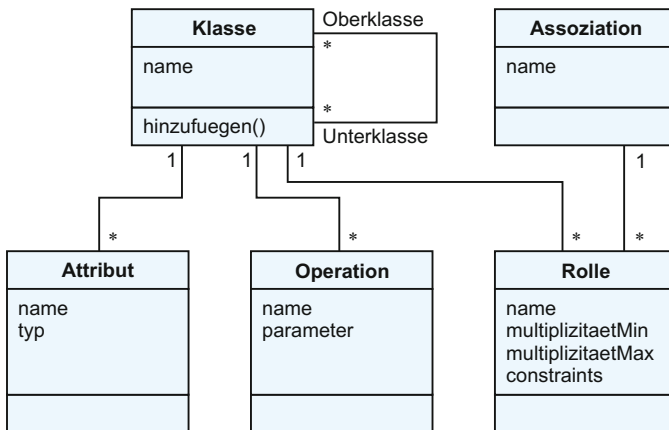


Abb. 4.1-2:
Beispiele für
objektorientierte
Abstraktions-
ebenen.

Eindimensionale vs. mehrdimensionale Abstraktion

Betrachtet man die in der Softwareentwicklung verwendeten Konzepte aus historischer Sicht, dann stellt man fest, dass neue Konzepte Sachverhalte stärker und mehrdimensional abstrahieren.

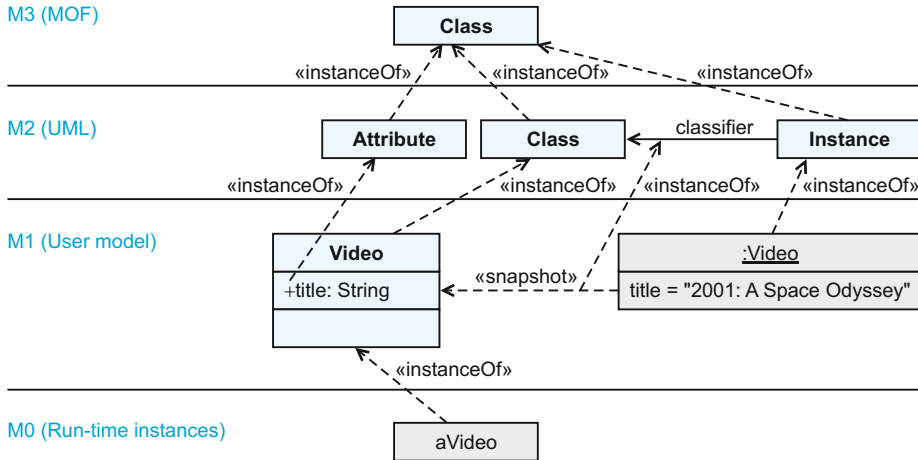
Bei Softwareentwicklungen spielen folgende Dimensionen bzw. Sichten eine wesentliche Rolle:

- Statik (Funktionen, Daten) (siehe »Statik«, S. 127)
- Dynamik (siehe »Dynamik«, S. 227)
- Logik (siehe »Logik«, S. 357)

Die folgenden Konzepte erlauben nur eine eindimensionale Modellierung bzw. Abstraktion:

- Funktionsbaum (Statik) (siehe »Funktionsbaum«, S. 143)
- XML (Statik) (siehe »XML, DTD und XML-Schemata«, S. 190)

4.1 Prinzip der Abstraktion II



- Entity-Relationship-Modell (Statik) (siehe »Entity-Relationship-Modell«, S. 199)
- Kontrollstrukturen (Dynamik) (siehe »Kontrollstrukturen«, S. 227)
- Geschäftsprozesse (Dynamik) (siehe »Geschäftsprozesse und Use Cases«, S. 250)
- Zustandsautomaten (Dynamik) (siehe »Zustandsautomaten«, S. 269)
- Petrinetze (Dynamik) (siehe »Petrinetze«, S. 303)
- Entscheidungstabellen (Logik) (siehe »Entscheidungstabellen und Entscheidungsbäume«, S. 386)
- Regeln (Logik) (siehe »Regeln«, S. 404)

Innerhalb dieser Konzepte gibt es nochmals Unterschiede im Abstraktionsgrad. Während z. B. ein Funktionsbaum und XML Sachverhalte nur »schwach« abstrahieren, ist ein Entity-Relationship-Modell eine »starke« Abstraktion (Typ-Ebene).

Eine zweidimensionale Modellbildung wird in folgendem Konzept ermöglicht:

- Klassen (Daten und Funktionen) (»starke« Abstraktion auf Typ-Ebene) (siehe »Funktionalität«, S. 127)

Vier verschiedene Dimensionen berücksichtigt das Konzept

- Objektorientierte Analyse (Statik mit Daten und Funktionen, Dynamik, Logik). Es führt zu einer »starken« Abstraktion (Typ-Ebene) (siehe »Beispiel: Objektorientierte Analyse«, S. 548).

Die Ermittlung geeigneter Abstraktionen ist um so leichter, je weniger Dimensionen berücksichtigt werden müssen, und je schwächer das zugrunde liegende Konzept abstrahiert. Der Einsatz neuer Konzepte erfordert daher höhere Abstraktionsfähigkeiten von den Mitarbeitern.

Abb. 4.1-3: Beispiel für das 4-Ebenen-Metamodell der UML [OMG09b, S.19].

II 4 Prinzipien

Abb. 4.1-4:
Kombinations-
möglichkeiten der
Objektorientie-
rung mit
generischen
Typen.

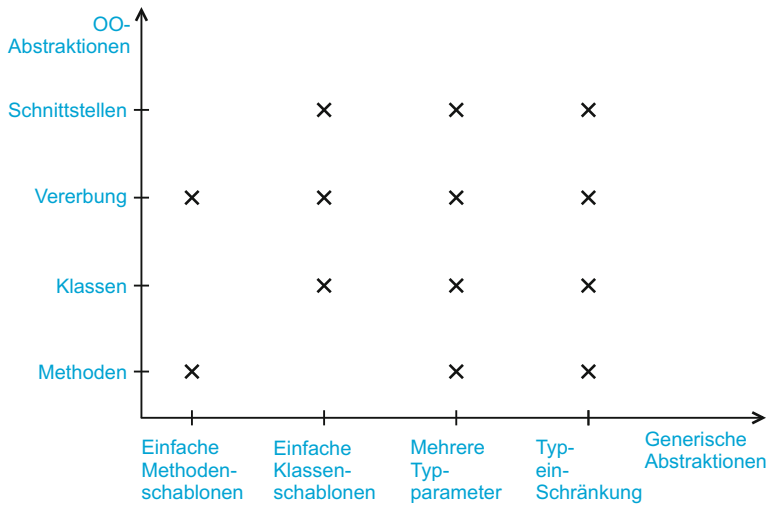
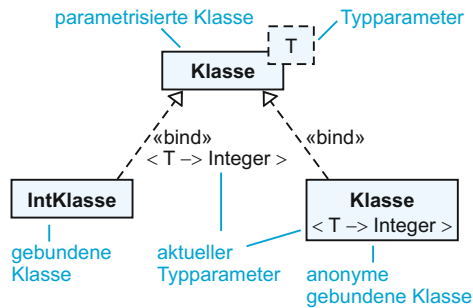


Abb. 4.1-5: UML-
Notation für
parametrisierte
Klassen (template
classes).



- Vorteile Die Anwendung des Prinzips der Abstraktion bringt folgende Vorteile:
- + Erkennen, Ordnen, Klassifizieren, Gewichten von wesentlichen Merkmalen.
 - + Erkennen allgemeiner Charakteristika (bildet Voraussetzung für Allgemeingültigkeit).
 - + Trennen des Wesentlichen vom Unwesentlichen.

4.2 Prinzip der Strukturierung

Frage Was ist für Sie eine Struktur? Wo in der Softwaretechnik spielen Strukturen eine Rolle?

Antwort Das Prinzip der Strukturierung hat sowohl für das fertige Softwareprodukt als auch für den Entwicklungs- und Qualitätssicherungsprozess eine große Bedeutung. Produkten und Prozessen soll eine geeignete Struktur aufgeprägt werden. Eine **Struktur** gibt die Anordnung der Teile eines Ganzen zueinander an. Etwas weitergehend ist folgende Definition:

Eine **Struktur** ist ein Gefüge, das aus Teilen besteht, die wechselseitig voneinander abhängen.

Definition

Im wissenschaftlichen Bereich findet man folgende Definitionen [Broc88]:

Eine **Struktur** ist ein (durch Relationen beschreibbares) Beziehungsgefüge und dessen Eigenschaften.

Eine **Struktur** ist ein nach Regeln aus Elementen zu einer komplexen Ganzheit aufgebautes Ordnungsgefüge.

Die folgende Definition berücksichtigt implizit die Abstraktion:

Unter der **Struktur** eines Systems versteht man die reduzierte Darstellung des Systems, die den Charakter des Ganzen offenbart. Losgelöst vom untergeordneten Detail beinhaltet die Struktur die wesentlichen Merkmale des Ganzen [Kope76, S. 39].

Definition

Strukturen können in allgemeiner Form durch Graphen dargestellt werden (siehe »Prinzipien«, S. 25). Die Semantik der Relation zwischen den Systemkomponenten bestimmt die Form der Struktur bzw. des Graphen.

In der Tab. 4.2-1 sind für eine Anzahl von Konzepten die Semantik der Relation und die sich daraus ergebende Form der Struktur angegeben. Strukturen lassen sich noch nach der Zeitspanne, in der sie existieren, klassifizieren. Drei verschiedene Zeitspannen lassen sich unterscheiden:

- Statische Struktur (Dokumentationsstruktur),
- dynamische Struktur (Laufzeitstruktur) und
- organisatorische Struktur (Entwicklungsstruktur).

Eine statische Struktur liegt vor, wenn eine Struktur ab einem Zeitpunkt vollständig vorliegt und erhalten bleibt. Die statische Struktur dokumentiert einen Sachverhalt. Bei den in Tab. 4.2-1 aufgeführten Strukturen handelt es sich, bis auf die Konfiguration, um statische Strukturen.

Statische Struktur

Entsteht während der Laufzeit eines Systems eine Struktur, dann liegt eine dynamische Struktur vor.

Dynamische Struktur

Durch rekursive Aufrufe entsteht während der Laufzeit eine dynamische Schachtelungsstruktur.

Beispiel

Eine organisatorische Struktur entsteht während einer Softwareentwicklung und existiert oft bis zur Fertigstellung des Produkts, in manchen Fällen während des gesamten Lebenszyklus des Produkts. Kennzeichnend für eine organisatorische Struktur ist, dass sie nicht »auf einen Schlag« da ist, sondern mit Fortschreiten der Entwicklung »wächst«.

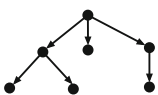
Organisatorische Struktur

II 4 Prinzipien

Konzept	Semantik der Relation	Unterschiedl. Komponententypen	Form der Struktur
Einfachvererbung in der OO	Attribute, Operationen und Assoziationen werden von Oberklasse A zu allen Unterklassen vererbt	1 (Klasse)	Gerichteter Baum
Mehrfachvererbung in der OO	Attribute, Operationen und Assoziationen werden von Oberklassen zu Unterklassen vererbt	1 (Klasse)	Gerichteter azyklischer Graph mit partieller Ordnung
Assoziationen in der OO	Zwischen Klassen bestehen Assoziationen	1 (Klasse)	Gerichteter Graph
UML-Aktivitätsdiagramm	Auf A folgt zeitlich B	> 10	Gerichteter Graph (transitiv)
Zustandsautomat	Von Zustand A wird in Zustand B übergegangen	1	Gerichteter Graph
Petrinetz	Transition	2	Gerichteter Graph
Netzplan	Auf A folgt zeitlich B	1	Gerichteter, azyklischer Graph
Konfiguration	Auf A folgt zeitlich B	1	Gerichteter, azyklischer Graph
Entwicklungs-Prozessmodell	Tätigkeit A erzeugt Teilprodukt B, Teilprodukt B wird verwendet von Tätigkeit C	2	Gerichteter Graph

Tab. 4.2-1:
Konzepte der
Softwaretechnik
und ihre
Strukturen.

Beispiel



Eine Konfigurationsstruktur eines Softwareprodukts entsteht während der Softwareentwicklung und setzt sich fort in der Wartungs- und Pflegephase. Die entstehende Struktur kann nicht von vornherein vollständig festgelegt werden. Sie ist nicht vorhersehbar.

Für die Projektorganisation wird von [Bake72, S. 72] folgendes Vorgehen beschrieben: »The general approach would be to begin a project with a single highlevel team to do overall system design and nucleus development. After the nucleus is functioning, programmers on the original team could become chief programmers on teams developing major subsystems. [...] The process could be repeated at lower levels if necessary.«

Diese Vorgehensweise impliziert eine gerichtete Baumstruktur.

Die Relation lautet: v_i **delegiert Aufgaben an** v_j .

Die Struktur kann *nicht* von vornherein festgelegt werden, da sich erst während der Entwicklung die Anzahl der Subsysteme ergibt, die wiederum die Anzahl der Teams determiniert.

Die Anwendung des Prinzips der Strukturierung bringt folgende Vorteile:

- + Erhöhung der Verständlichkeit.
- + Verbesserung der Wartbarkeit.

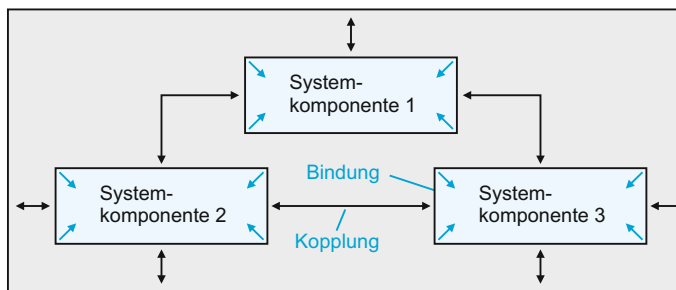
- + Erleichterung der Einarbeitung in ein fremdes Softwareprodukt.
- + Beherrschbarkeit der Komplexität eines Systems.

4.3 Prinzip der Bindung und Kopplung

Die Struktur eines Softwaresystems wird im Wesentlichen durch die Bindung jeder Systemkomponente und die Kopplungen zwischen den Systemkomponenten untereinander bestimmt (Abb. 4.3-1). In einem objektorientierten System werden die Systemkomponenten durch Klassen und Pakete realisiert. Die Kopplungen sind die Assoziationen und Vererbungsstrukturen.

Struktur

Umgebung



Legende: ↔ Beziehung

Abb. 4.3-1:
Bindung und
Kopplung.

Bindung (*cohesion*) ist ein qualitatives Maß für die Kompaktheit einer Systemkomponente. Es werden dazu die Beziehungen zwischen den Elementen innerhalb einer Systemkomponente betrachtet. Es wird untersucht, wie eng die Elemente verbunden sind und wie viele Aufgaben in der Systemkomponente erledigt werden. In einem stark gebundenen System ist jede Komponente (Methode, Klasse, Paket) für eine definierte Aufgabe verantwortlich (*responsible*).

Bindung

Das Gegenstück zur Bindung stellt die Kopplung dar. **Kopplung** (*coupling*) ist ein qualitatives Maß für die Schnittstellen zwischen Systemkomponenten. Es werden der Kopplungsmechanismus, die Schnittstellenbreite und die Kommunikationsart betrachtet. Der Kopplungsgrad bestimmt, wie einfach oder schwierig es ist, Änderungen an einem System vorzunehmen. Sind Klassen eng miteinander gekoppelt, dann erfordert die Änderung in einer Klasse gleichzeitig viele Änderungen in anderen Klassen. Außerdem ist es oft schwierig festzustellen, in welchen anderen Klassen Änderungen nötig sind. In einem lose gekoppelten System dagegen haben Änderungen in einer Klasse im Optimalfall keinerlei Auswirkungen auf andere Klassen.

Kopplung

II 4 Prinzipien

Ziel Zur Ausprägung einer Struktur lässt sich folgende These aufstellen:

- Die Struktur eines Systems ist um so ausgeprägter und die Modularität ist um so höher, je stärker die Bindungen der Systemkomponenten im Vergleich zu den Kopplungen zwischen den Systemkomponenten sind.

Je stärker die Ausprägung einer Struktur ist, desto geringer ist auch die Komplexität dieser Struktur. Geringe Komplexität bedeutet aber hoher Grad an Einfachheit, gute Verständlichkeit, leichte Einarbeitung.

- Die Forderung nach Einfachheit wird erfüllt, wenn die **Kopplungen minimiert und die Bindungen maximiert werden**.

Für die Produktqualität einer Systemkomponente spielt daher die Bindung der Systemkomponente eine entscheidende Rolle, für die Produktqualität eines Softwaresystems die Kopplung zwischen ihren Systemkomponenten.

4.4 Prinzip der Hierarchisierung

Nach [Simo62] verfügen viele in der Natur vorkommende komplexe Systeme über eine **hierarchische Struktur**. Unabhängig davon, ob es sich dabei um ein Ordnungsprinzip der Natur an sich handelt oder ob sie nur der menschlichen Erkenntnis der Natur entspringt, bietet sich eine solche Struktur offenbar auch als geeignetes Mittel für den Aufbau von künstlichen Systemen an.

Frage Woran denken Sie, wenn Sie den Begriff »Hierarchie« hören?

Antwort Der Begriff »Hierarchie« stammt aus der katholischen Kirche, wird heute aber in fast allen Lebens- und Wissenschaftsbereichen verwendet. Eine Hierarchie bezeichnet eine Rangordnung, eine Abstufung sowie eine Über- und Unterordnung.

Definition Ein System besitzt eine **Hierarchie**, wenn seine Elemente nach einer Rangordnung angeordnet sind. Elemente gleicher Rangordnung stehen auf derselben Stufe, sie bilden eine Ebene bzw. Schicht der Hierarchie.

Hierarchie vs. Struktur Genau betrachtet lässt sich das Prinzip der Hierarchisierung dem Prinzip der Strukturierung unterordnen (siehe »Prinzip der Strukturierung«, S. 34). Eine Hierarchie lässt sich nach denselben Kategorien klassifizieren wie eine Struktur. Die Semantik der Relation muss jedoch immer eine Rangfolge beinhalten.

Da hierarchische Systeme in der Softwaretechnik eine große Bedeutung besitzen, wird das Prinzip der Hierarchisierung hier als eigenständiges Prinzip aufgeführt, obwohl es ein Spezialfall des Prinzips der Strukturierung ist.

4.4 Prinzip der Hierarchisierung II

Hierarchien werden analog zu Strukturen dargestellt. Die Kanten müssen jedoch immer gerichtet sein. In der Regel sollten übergeordnete Elemente auch über den untergeordneten Elementen gezeichnet werden.

Zur Notation

Konzept/Methode	Semantik der Relation	Unterschiedl. Systemkomponenten-Typen	Hierarchieform
Funktionsbaum	A besteht aus B, A ruft B auf	1	Gerichteter Baum
Einfachvererbung	A vererbt an B	1	Gerichteter Baum
Mehrfachvererbung (OO)	A vererbt an B	1	Gerichteter azyklischer Graph mit partieller Ordnung
Aggregation (OO)	A besteht aus B, C ...	1	Gerichteter Graph
Entscheidungsbaum	A wird vor B entschieden	1	Gerichteter Baum
Pakete (OO)	A fasst B, C ... zusammen	2 (Paket, Klasse)	Gerichteter Baum
Zustandsautomat	B verfeinert A	1	Gerichteter Baum
UML-Aktivitätsdiagramm	B verfeinert A	1	Gerichteter Baum
UML-Sequenzdiagramm	B verfeinert A	1	Gerichteter Baum
Aufbauorganisation (außer Matrix)	A leitet B	1	Gerichteter Baum
Aufbauorganisation (Matrix)	C untersteht A und B	1	Gerichteter azyklischer Graph mit partieller Ordnung

In der Tab. 4.4-1 sind für eine Anzahl von Konzepten und Methoden die Semantik der Relation und die sich daraus ergebende Hierarchieform angegeben. Wie die Tabelle zeigt, überlagern sich bei Methoden mehrere Strukturen und/oder Hierarchien. Analog zu Strukturen lassen sich auch statische, dynamische und organisatorische Hierarchien unterscheiden.

Tab. 4.4-1: Konzepte / Methoden der Softwaretechnik und ihre Hierarchien.

Vergleichen Sie die Strukturierung mit der Hierarchisierung. Welche Vor- und Nachteile hat die Hierarchisierung?

Frage

Die Vorteile der Strukturierung gelten auch für die Hierarchisierung. Eine Hierarchie schränkt jedoch stärker ein als eine Struktur. Das hat sowohl Nachteile als auch gewünschte Vorteile. Der Hauptnachteil liegt in der Beschränkung auf definierte, gerichtete Strukturen. Dieser Nachteil ist zugleich auch der größte Vorteil. Er verhindert nämlich chaotische Strukturen.

Antwort

II 4 Prinzipien

4.5 Prinzip der Modularisierung

Frage Was assoziieren Sie mit dem Begriff »Modul«? Welche Eigenschaften hat ein Modul?

Antwort Das Prinzip der Modularisierung wird insbesondere in Ingenieurdisziplinen angewandt. Ein Computersystem oder ein Fernsehgerät wird aus Modulen aufgebaut, wobei jedes Modul eine weitgehend abgeschlossene Bau- oder Funktionsgruppe darstellt. Im Fehlerfall wird das komplette Modul gegen ein fehlerfreies ausgetauscht. Um dies zu ermöglichen, muss das Modul eine festgelegte Schnittstelle zu den anderen Modulen des Gerätes besitzen. Alle Informationen müssen über diese Schnittstelle laufen. Dieses Beispiel zeigt bereits einige Eigenschaften von Modulen. In der Softwaretechnik werden die Begriffe Modularität und Modul entweder in einem sehr weiten Sinne oder in einem sehr engen Sinne definiert. In diesem Abschnitt wird nur eine weit gefasste Begriffsbestimmung gegeben.

Beispiel Dokumente – insbesondere Referenz- und Benutzerhandbücher – sollen modular aufgebaut sein. Wird z.B. eine Funktion in einem Softwareprodukt geändert, dann soll es möglich sein, die zugehörige Funktionsbeschreibung im Referenzhandbuch gegen die neue auszutauschen. Solche partiellen Änderungen dürfen jedoch nicht dazu führen, dass das gesamte Referenzhandbuch gegen ein neues ersetzt werden muss. Beim partiellen Austauschen von Teilen müssen die Seitennummerierung und die Bezüge konsistent bleiben. Modularität von Dokumenten zeigt sich daher in folgenden Eigenschaften:

- Kapitel- oder abschnittsweise Seitenzählung z.B. Kapitel 2/Seite 5. Die Seitenzählung basiert jeweils auf dem Kapitelanfang. Bei Erweiterung eines Kapitels muss nicht das ganze Dokument neu nummeriert werden. Alle Abbildungsnummern werden jeweils kapitelweise durchgezählt.
- Die Bezüge auf andere Kapitel sollten möglichst gering sein, d. h. in den einzelnen Kapiteln sollten weitgehend abgeschlossene Themen behandelt werden.
- Die Kapitel dürfen nicht umfangreich sein, sonst geht der Vorteil der Modularität wieder verloren.
- Wünschenswert ist ein Referenzverzeichnis am Ende jedes Kapitels, damit auf einen Blick sichtbar ist, auf welche anderen Kapitel Bezug genommen wird.

Bildlich gesehen ist jedes Kapitel in sich abgeschlossen (Abb. 4.5-1). Die Bezüge zu anderen Kapiteln sind reduziert und leicht lokalisierbar. Bei Referenzhandbüchern ist oft sogar eine seitenweise Modularität möglich.

4.5 Prinzip der Modularisierung II

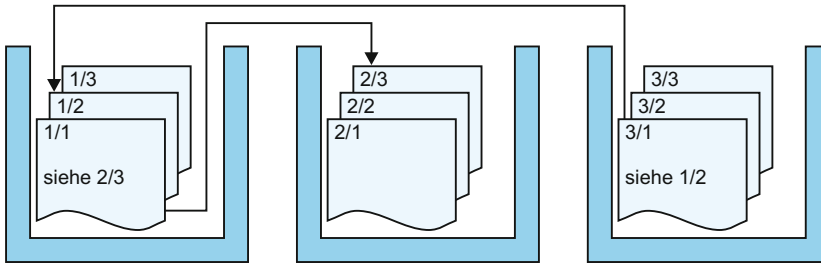


Abb. 4.5-1:
Dokument-Module.

Modularisierung bezeichnet die Konzeption von Modulen beim Softwareentwicklungsprozess. Modularisierung

Ein **Modul im weiteren Sinne** ist durch folgende Eigenschaften gekennzeichnet: Modul

- Darstellung einer funktionalen Einheit oder einer semantisch zusammengehörenden Funktionsgruppe.
- Weitgehende Kontextunabhängigkeit, d.h. ein Modul ist in sich abgeschlossen. Die Kontextunabhängigkeit beinhaltet, dass ein Modul von der Modul Umgebung weitgehend unabhängig entwickelbar, prüfbar, wartbar und verständlich ist.
- Definierte Schnittstelle für Externbezüge. Die externen Bezüge eines Moduls sollten klar erkennbar und möglichst in einer Schnittstellenbeschreibung zusammengefasst sein.
- Im qualitativen und quantitativen Umfang handlich, überschaubar und verständlich.

Unter »funktionaler Einheit« ist zu verstehen, dass in einem Modul zusammengehörende Dinge vereinigt sein sollen.

Werden in einem Benutzerhandbuch z. B. die Benutzungsoberfläche und die Druckausgaben in einem Kapitel zusammengefasst beschrieben, dann muss bei Änderung der Druckausgaben auch die Beschreibung der Benutzungsoberfläche ausgetauscht werden. Benutzungsoberfläche und Druckausgaben stehen in keinem so engen semantischen Zusammenhang, dass sie sinnvoll in einem Dokument-Modul zusammengefasst werden sollten. Beispiel

Die Kontextunabhängigkeit bedeutet, dass z. B. die Funktion »Suchen und Ersetzen eines Textes« in einem Textbearbeitungssystem unabhängig von der Funktion »Löschen eines Zeichens, eines Wortes, einer Zeile, eines Absatzes, eines Textbausteines« beschrieben ist und verstanden werden kann.

Betrachtet man die Konzepte der Softwaretechnik, dann findet man Modularität bei Systemkomponenten (Modularität im Kleinen) und/oder Subsystemen (Modularität im Großen).

Modularität im Großen wird durch folgende Konzepte unterstützt: Modularität im Großen

- Pakete
- Hierarchische Zustandsautomaten (Harel-Automat)
- Hierarchische Petrinetze

II 4 Prinzipien

Modularität im Kleinen	■ Hierarchische Netzpläne bei der Projektplanung
	Modularität im Kleinen wird durch folgende Konzepte unterstützt:
	■ Klassen
	■ Datenabstraktion
	■ Methoden, Prozeduren und Funktionen
	■ Entscheidungstabellen-Verbunde
	Diese Konzepte ermöglichen eine Modularisierung. Sie erzwingen sie in der Regel nicht.
Zusammenhang mit Abstraktion	Modularität im Großen ist eng verknüpft mit dem Prinzip der Abstraktion, da die Modularisierung gleichzeitig das Bilden von Abstraktionsebenen erfordert.
Zur Notation	Zur Kennzeichnung eines Moduls gibt es <i>keine</i> einheitliche Notation. Sie ergibt sich implizit durch die jeweils verwendete Notation für Systemkomponenten und Subsysteme.
	Das Prinzip der Modularisierung gilt nicht nur in der Softwareentwicklung, sondern ist auch bei der Gestaltung der Aufbauorganisation eines Unternehmens zu berücksichtigen. In Abhängigkeit von der Art der Interdependenzen zwischen Arbeitsabläufen ergibt sich eine »modulare« Aufbauorganisation.
Frage	Welche Vorteile besitzt die Modularisierung in der Softwaretechnik?
Antwort	Die Anwendung des Prinzips der Modularisierung bringt folgende Vorteile:
	+ Hohe Änderungsfreundlichkeit (leichter Austausch und leichte Erweiterbarkeit).
	+ Verbesserung der Wartbarkeit (leichte Lokalisierung, Kontextunabhängigkeit).
	+ Erleichterung der Standardisierung.
	+ Erleichterung der Arbeitsorganisation und Arbeitsplanung.
	+ Verbesserung der Überprüfbarkeit.

4.6 Geheimnisprinzip

Das **Geheimnisprinzip** (*information hiding*) [Parn71] stellt eine »Verschärfung« des Prinzips der Modularisierung dar. Das Geheimnisprinzip bedeutet, dass für den Anwender bzw. Benutzer einer Systemkomponente oder eines Subsystems die Interna der Systemkomponente bzw. des Subsystems verborgen, d.h. nicht sichtbar sind (Abb. 4.6-1).

Sinnvoll kann das Geheimnisprinzip nur in Verbindung mit der Modularisierung angewandt werden. Ein Modul soll kontextunabhängig sein und über eine definierte Schnittstelle mit der Umwelt kommunizieren. Die Anwendung des Geheimnisprinzips bedeutet dann, dass nur diese definierte Schnittstelle von außen sichtbar ist.

4.6 Geheimnisprinzip II

Das Geheimnisprinzip kann auch aus einer anderen Sicht definiert werden: Überflüssige Angaben, die zur Erledigung einer Aufgabe *nicht* benötigt werden, dürfen auch *nicht* sichtbar sein.

Historisch betrachtet entstand das Geheimnisprinzip im Rahmen des Softwareentwurfs und der Softwareimplementierung. Sowohl von Funktionen und Prozeduren als auch von abstrakten Datenobjekten und abstrakten Datentypen sollen nur die Schnittstellen für den Anwender sichtbar sein. Im Zusammenhang mit abstrakten Datenobjekten, abstrakten Datentypen und Klassen entstanden die Begriffe **Kapselung** (*encapsulation*), Verkapselung, Einkapselung bzw. Datenkapsel. Kapselung bedeutet, dass zusammengehörende Attribute bzw. Daten und Operationen in einer Einheit zusammengefasst sind. Die Kapselung stellt also eine spezielle Form eines Moduls dar. Die Einhaltung des Geheimnisprinzips bei einer Kapselung bedeutet, dass die Attribute bzw. Daten und die Realisierung der Operationen außerhalb der Kapselung *nicht* sichtbar sind. Das Geheimnisprinzip »verschärft« also eine Kapselung.

Es gibt zwei verschiedene Ausprägungen des Geheimnisprinzips.

In seiner liberalen Form bedeutet es, dass alle Interna einer Systemkomponente oder eines Subsystems für den Anwender zwar physisch sichtbar sind, dass aber Entwicklungsumgebungen oder Compiler unerlaubte Zugriffe auf Interna erkennen und als Fehler melden. Selbst wenn unerlaubte Zugriffe erkannt werden können, kann der Anwender doch die Interna sehen und dadurch vielleicht dieses Wissen implizit bei der Anwendung dieser Systemkomponente oder dieses Subsystems mitverwenden.

Solche impliziten Annahmen, die sich aus Kenntnissen der Interna ergeben, beeinflussen jedoch die Änderungsfreundlichkeit einer Systemkomponente oder eines Subsystems. Sollen die Interna einer Systemkomponente z. B. aus Effizienzgründen geändert werden, ohne dass die Schnittstelle davon betroffen ist, dann darf der Anwender davon nichts merken.

Diese Nachteile vermeidet man bei der strengen Anwendung des Geheimnisprinzips. Außer der definierten Schnittstelle sind sämtliche Interna für den Anwender völlig unsichtbar, d. h. beide Teile sind auch textuell völlig getrennt. Dies gilt beispielsweise für Pakete, Prozeduren und Funktionen in der Programmiersprache Ada und für öffentliche Klassenbibliotheken in Java. Das wiederum bedeutet, dass die Spezifikationen einer Systemkomponente bzw. eines Subsystems exakt und vollständig das Verhalten beschreiben müssen. Das kann sehr schwierig sein.

In der objektorientierten Welt wird das Geheimnisprinzip in verschiedenen Ausprägungen verwendet.

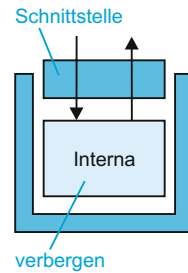


Abb. 4.6-1: Das Geheimnisprinzip.

Liberaler Ausprägung des Geheimnisprinzips

Strenge Ausprägung des Geheimnisprinzips

OO-Welt

II 4 Prinzipien

In einer Klasse sollte auf die Attribute nur über die Methoden zugegriffen werden. Der *Requirements Engineer* konzipiert eine Klasse einschließlich ihrer Attribute und sieht sie natürlich auch. Das Vererbungskonzept ermöglicht es, dass Unterklassen die Attribute der Oberklassen sehen (wenn protected). Wenn ein Objekt einer Unterklasse direkt auf die Attribute eines Objekts der Oberklasse zugreift, dann ist das Geheimnisprinzip verletzt. Es hängt von der verwendeten Programmiersprache ab, inwieweit das Geheimnisprinzip »durchlöchert« werden kann.

Die Programmiersprachen Java und C++ erlauben verschiedene Varianten des Geheimnisprinzips:

- Streng, durch das `private`-Konzept.
- Liberal, durch das `protected`-Konzept (innerhalb der Vererbungshierarchie kann auf die Attribute der Oberklassen direkt zugegriffen werden).
- Kein Geheimnisprinzip durch das `public`-Konzept (auf die Attribute kann frei zugegriffen werden).
- Beschränkt durch das `friendly`-Konzept in Java (Wird in Java nicht explizit eine Sichtbarkeit für Klassen, Methoden oder Attribute angegeben, so wird implizit als Sichtbarkeit `friendly` vereinbart. Das entsprechende syntaktische Konstrukt ist dann nur innerhalb des eigenen Paketes sichtbar).
- Beschränkt, durch das `friend`-Konzept in C++ (Methoden und Klassen können als »Freund« von anderen Klassen deklariert werden. Von den befreundeten Methoden und Klassen aus kann dann auf die privaten Anteile der Klasse zugegriffen werden, die die Freundschaft mit Hilfe des Schlüsselwortes `friend` deklariert).

Konflikte Die strenge Anwendung des Geheimnisprinzips führt in einigen Fällen zu Konflikten. Daten, die in Datenbanken gespeichert sind, will man in vielen Fällen mittels Prädikaten über Teile ihres Wertes (»... where Umsatz > 5000«) selektieren. Der Zugriff auf Daten über Datenbanksysteme erfordert daher eine differenzierte Anwendung des Geheimnisprinzips. Die Anwendung des Geheimnisprinzips bringt folgende Vorteile mit sich:

- + Die Anwendung einer Systemkomponente oder eines Subsystems wird zuverlässiger, da nur über die definierte Schnittstelle kommuniziert werden kann.
- + Der Anwender einer Systemkomponente oder eines Subsystems wird *nicht* mit unnötigen Informationen belastet.
- + Die Datenkonsistenz interner Daten kann besser sichergestellt werden, da direkte, unkontrollierbare Manipulationen nicht möglich sind.

Als Nachteile ergeben sich:

- Die Anwendungsschnittstelle muss vollständig und exakt beschrieben werden.

- In einigen Fällen kann das Geheimnisprinzip auch »hinderlich« sein, z. B. beim deklarativen Zugriff auf gespeicherte Daten. Generell sollte so viel wie möglich vom Geheimnisprinzip Gebrauch gemacht werden. Eine »Aufweichung« des Geheimnisprinzips sollte auf Sonderfälle beschränkt bleiben. Das Geheimnisprinzip sollte immer im Zusammenhang mit dem Prinzip der Modularisierung verwendet werden.

4.7 Prinzip der Lokalität

Zum Verstehen komplexer Probleme ist es notwendig, sich zu einem Zeitpunkt nur mit einer kleinen Anzahl von Eigenschaften zu beschäftigen, von denen man meint, dass sie wesentlich für den gegenwärtigen Gesichtspunkt sind. Von der konstruktiven Seite her ist es daher wichtig, alle relevanten Informationen für wichtige Gesichtspunkte lokal, d. h. an einem Platz, zur Verfügung zu stellen.

Durch gute Lokalität kann das Zusammensuchen benötigter Informationen, z. B. das Blättern auf verschiedenen Seiten eines Dokuments, das Suchen referenzierter globaler Objekte, vermieden werden.

Optimale **Lokalität** liegt vor, wenn zur Lösung eines Problems (z. B. Vorgaben für die Implementierung) oder zum Einarbeiten in einen Bereich (z. B. Fehlersuche in einem Modul) alle benötigten Informationen auf einer Seite zu finden sind [Wein71, S. 229]. Das bedeutet andererseits, dass *nicht* benötigte Informationen *nicht* vorhanden sind.

Lokalität

Viele Konzepte unterstützen von vornherein eine gute Lokalität, andere erschweren Lokalität.

Unterstützung der Lokalität:

Beispiele

- Entscheidungstabellen (ET) sind sehr kompakt und besitzen eine optimale Lokalität. Durch ET-Verbunde kann das Prinzip der Lokalität auch bei umfangreichen Tabellen eingehalten werden (siehe »Entscheidungstabellen und Entscheidungsbäume«, S. 386).
- Hierarchische Zustandsautomaten nach Harel erlauben pro Hierarchieebene eine lokale Sicht auf zusammengehörende Details (siehe »Zustandsautomat nach Harel«, S. 277).
- Hierarchische Petrinetze liefern pro Hierarchieebene alle Informationen, die für bestimmte Gesichtspunkte nötig sind (siehe »Hierarchische Petrinetze«, S. 312).
- Die objektorientierte Analyse (OOA) erlaubt durch Subsystem- bzw. Paketbildung eine Zusammenfassung von Klassen zu einer Einheit nach bestimmten Gesichtspunkten. Dadurch wird das Lesen und Verstehen eines Klassen-Diagrammes wesentlich erleichtert (siehe »Pakete«, S. 145).

II 4 Prinzipien

Behinderung der Lokalität:

- In der objektorientierten Welt wird das Verständnis einer Klasse erschwert, wenn sich die Klasse in einer Vererbungsbeziehung befindet. Informationen müssen entlang der Vererbungsbeziehungen zusammengesucht werden, um eine Klasse zu verstehen (siehe »Vererbung«, S. 150).
- Da in den Sprachen C++ und Java eine Schachtelung von Methoden *nicht* möglich ist, sind auch lokale Hilfsmethoden global in der jeweiligen Klasse angeordnet und sichtbar. Dadurch wird der Überblick und die Einarbeitung erschwert, da nicht benötigte Informationen sichtbar sind und wahrgenommen werden (siehe »Funktionalität«, S. 127).

Im Bereich der Implementierung hat das Lokalitätsprinzip zu folgender These geführt:

- Eine Gruppe von ungefähr 30 Anweisungen ist die oberste Grenze dessen, was beim ersten Lesen eines Listings einer Systemkomponente, eines Moduls oder einer Prozedur, Funktion bzw. Operation bewältigt werden kann.

Zur Notation Zur Kennzeichnung der Lokalität gibt es keine eigene Notation. Konzepte unterstützen meist implizit das Prinzip der Lokalität.

Die Beachtung des Prinzips der Lokalität bringt folgende Vorteile:

- + Ermöglicht die schnelle Einarbeitung.
- + Fördert die Verständlichkeit und Lesbarkeit.
- + Erleichtert die Wartung und Pflege.

Dem steht folgender Nachteil gegenüber:

- Nachteilig ist, dass die Lokalität das Geheimnisprinzip erschwert, da durch das Geheimnisprinzip öffentlich zugängliche und private Informationen getrennt werden müssen, obwohl sie inhaltlich zusammengehören und nach dem Lokalitätsprinzip daher an einer lokalen Stelle zusammenstehen sollten.

4.8 Prinzip der Verbalisierung

Frage Sie müssen sich in ein fremdes Softwareprodukt einarbeiten, um Wartungsmaßnahmen durchzuführen. Wodurch wird Ihre Arbeit erleichtert?

Antwort Sie erwarten sicher eine gute Dokumentation und Beschreibung des Softwareprodukts. Beides hängt wesentlich von einer guten Verbalisierung ab.

Verbalisierung bedeutet, Gedanken und Vorstellungen in Worten auszudrücken und damit ins Bewusstsein zu bringen.

4.8 Prinzip der Verbalisierung II

Dieses Prinzip wird schon lange für die Programmierung propagiert. Es hat heute jedoch eine wesentlich umfassendere Bedeutung. Insbesondere in den frühen Phasen der Softwareentwicklung kommt der Verbalisierung eine herausragende Bedeutung zu.

Die bei der Spezifikation gewählten Begriffe, Klassifizierungen und Namen beeinflussen alle weiteren Entwicklungsaktivitäten in ihrer Begrifflichkeit. Dieses Erkenntnis hat dazu geführt, dass viele Konzepte, die im *Requirements Engineering* eingesetzt werden, explizite Vorschriften und Regeln für die Namensgebung aufführen. Die Überprüfung der Einhaltung ist Aufgabe der Qualitätssicherung.

»Da der Computer keine Anschauungssemantik kennt, kann er auch nicht dazu benutzt werden sicherzustellen, daß bei der Wahl von Namen für Module, Funktionen, Typen oder Variablen zweckmäßige anschauungssemantische Bezüge hergestellt werden. Je komplexer die Systeme werden, um so undurchschaubarer wird der Namenswirrwarr, wenn den Entwicklern die Namenswahl freigestellt bleibt, d. h. wenn die Sicherstellung anschauungssemantischer Bezüge nicht bewußt als Engineering-Aufgabe wahrgenommen wird. Man bedenke, daß in der oben erwähnten Software in einer Million Zeilen C-Quellcode insgesamt rund 29 000 Namen vereinbart werden mußten« [Wend93, S. 36].

Zitat

Eine gute Verbalisierung kann erreicht werden durch:

- Aussagekräftige, mnemonische Namensgebung,
- geeignete Kommentare und
- selbstdokumentierende Konzepte und Sprachen.

Beispiele für explizite Regeln zur Namensgebung in den frühen Phasen einer Softwareentwicklung:

Beispiele

OOA (Objektorientierte Analyse) (siehe »Beispiel: Objektorientierte Analyse«, S. 548)

- Der Klassenname soll
 - ein Substantiv im Singular sein,
 - so konkret wie möglich gewählt werden,
 - dasselbe ausdrücken, wie die Gesamtheit der Attribute und/oder Operationen,
 - nicht die Rolle beschreiben, die diese Klasse in einer Beziehung zu einer anderen Klasse spielt.
- Ein Attributname soll
 - eindeutig und verständlich im Kontext der Klasse sein,
 - den Namen der Klasse nicht wiederholen (Ausnahmen sind feststehende Begriffe),
 - bei strukturierten Attributen der Gesamtheit der Komponenten entsprechen.
- Der Name einer Operation ist so zu wählen, dass er
 - ein Verb enthält,
 - dasselbe aussagt, wie die Spezifikation der Operation,

II 4 Prinzipien

- den Klassennamen nicht wiederholt (Ausnahme: feststehende Begriffe),
- die funktionale Bindung der Operation bestätigt.

Modellierung von Use Cases (siehe »Geschäftsprozesse und Use Cases«, S. 250)

■ Als Name eines Use Case sollte gewählt werden:

- Die Gerundiumform eines Verbs, z. B. Verkaufen, oder
- ein Substantiv gefolgt von einem Verb, z. B. Verkauf durchführen, oder
- der Anfangs- und Endpunkt des Prozesses, z. B. Interessent bis Auftrag.

Benutzungsoberfläche

- Die Aufgaben müssen mit den Fachbegriffen beschrieben werden, die der Benutzer kennt.
- Die für seine fachliche Arbeitstätigkeit relevanten Aufgabenbereiche und die dafür im Softwaresystem vorgesehenen Anwendungen muss der Benutzer ohne Schwierigkeiten identifizieren können.

Wichtig ist, dass die Werkzeuge die Konzepte unterstützen bzw. die Compiler für die Programmiersprachen sowohl lange Namen als auch die geeignete Strukturierung von Namen erlauben.

Im Zusammenhang mit der Verbalisierung ist auch der Grad der **Selbstdokumentation** einer Programmiersprache oder eines Konzepts nicht zu unterschätzen. Bei den Programmiersprachen spielen insbesondere die Schlüsselwörter und die Möglichkeiten der Programmstrukturierung eine Rolle. Die Programmiersprache Ada gilt hierbei als vorbildlich, C++ als das Gegenteil.

Bei Konzepten kommt es darauf an, dass die gewählten grafischen und textuellen Notationen intuitiv verständlich sind. Die jeweils relevanten Informationen müssen sichtbar, die unnötigen unsichtbar sein.

Die Einhaltung des Prinzips der Verbalisierung bringt folgende Vorteile mit sich:

- + Leichte Einarbeitung in fremde Modelle, Architekturen, Programme bzw. Wiedereinarbeitung in eigene Dokumente.
- + Erleichterung der Qualitätssicherung, der Wartung und Pflege.
- + Verbesserte Lesbarkeit der erstellten Dokumente.

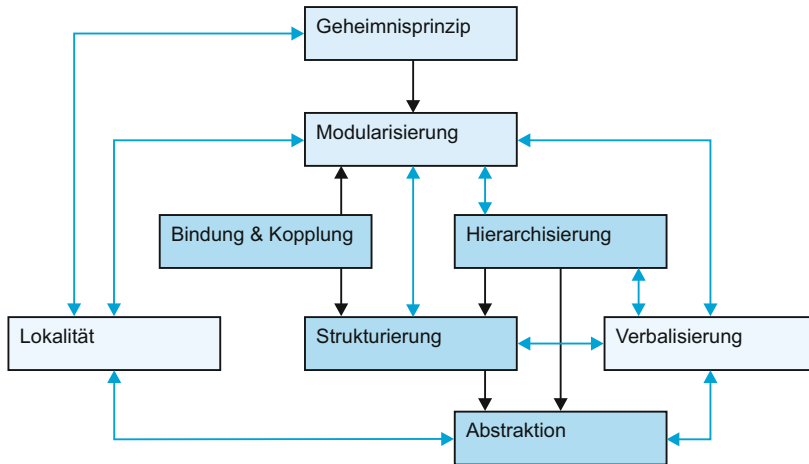
4.9 Abhängigkeiten zwischen den Prinzipien

Frage Welche Prinzipien setzen welche anderen Prinzipien voraus? Welche Prinzipien stehen in einer Wechselwirkung?

4.9 Abhängigkeiten zwischen den Prinzipien II

Viele der aufgeführten Prinzipien erscheinen trivial. Die Schwierigkeit ihrer Anwendung besteht jedoch darin, dass sie wechselseitig miteinander verwoben sind und sich zum Teil gegenseitig voraussetzen. Außerdem lässt sich keine kausale Kette herleiten, die besagt, in welcher Reihenfolge die Prinzipien anzuwenden sind. In der Abb. 4.9-1 wird versucht, die gegenseitigen Abhängigkeiten der Prinzipien darzustellen.

Antwort



Legende: A → B : A setzt B voraus
A ↔ B : A und B stehen in Wechselwirkung

Abb. 4.9-1:
Abhängigkeiten
zwischen den
Prinzipien.

Bevor ein System strukturiert werden kann, muss der Abstraktionsprozess abgeschlossen sein. Eine Hierarchiebildung prägt einer Struktur eine Rangordnung auf. Daher sind Strukturierung und Abstraktion Voraussetzungen für die Hierarchisierung.

Eine Modulbildung setzt entweder eine Strukturierung oder eine Hierarchisierung voraus. Es kann aber auch zunächst eine Modularisierung erfolgen, die dann zu einer impliziten oder expliziten Strukturierung oder Hierarchisierung führt. Daher ist hier eine wechselseitige Abhängigkeit angegeben.

Bindung und Kopplung setzen eine Modularisierung und Strukturierung voraus.

Das Geheimnisprinzip ist eine »Verschärfung« des Modularisierungsprinzips, daher ist die Modularisierung die Voraussetzung für das Geheimnisprinzip.

Das Lokalitätsprinzip steht in Wechselwirkung mit der Modularisierung und der Abstraktion. Beide Prinzipien sind erforderlich, um die notwendigen Informationen zu einem Gesichtspunkt (bestimmt durch die Abstraktion) lokal anzuordnen (möglich durch die Modulbildung). Durch das Geheimnisprinzip kann dann noch gesteuert werden, dass nicht relevante Informationen auch nicht sichtbar sind.

II 4 Prinzipien

Das Prinzip der Verbalisierung hat die meisten Wechselwirkungen zu anderen Prinzipien. Für Abstraktionsebenen, Strukturen, Hierarchien und Module müssen geeignete Namen gewählt werden. Bei der Namensgebung stellt man umgekehrt wieder fest, ob die Strukturen, Hierarchien, Abstraktionen und Module richtig gewählt wurden.

4.10 Zusammenfassung

Allgemeine Prinzipien verbinden die verschiedenen Bereiche und Aktivitäten der Softwaretechnik. Eines der wichtigsten Prinzipien der Softwaretechnik ist das Prinzip der Abstraktion. Zwischen den Polen »konkret« und »abstrakt« werden Abstraktionsebenen gebildet. Es entstehen unterschiedlich abstrakte Modelle der realen Welt. Prinzipiell lassen sich drei Abstraktionsebenen unterscheiden:

- die Exemplar-Ebene,
- die Typ-Ebene und
- die Meta-Ebene.

Sowohl Softwaresysteme als auch Softwareentwicklungsprozesse werden strukturiert oder hierarchisch gegliedert. Strukturen lassen sich klassifizieren durch

- die Semantik der Relation zwischen den Systemkomponenten,
- die Form der Struktur,
- die Anzahl unterschiedlicher Systemkomponenten-Typen und
- die Zeitspanne, in der die Struktur existiert (statisch, dynamisch, organisatorisch).

Einen Spezialfall der Struktur stellt die Hierarchie dar. Zwischen den Systemkomponenten einer Hierarchie besteht immer eine Rangordnung, d. h. eine Über-/Untergeordneten-Beziehung. Hierarchien lassen sich wie Strukturen klassifizieren.

Ein Modul ist eine Systemkomponente bzw. ein Subsystem mit folgenden Eigenschaften:

- Bereitstellung einer Funktion oder semantisch zusammengehörender Funktionen.
- Weitgehend kontextunabhängig.
- Definierte Schnittstellen.
- Handlich, überschaubar, verständlich.

Man kann Modularität im Großen (bezogen auf Subsysteme) und Modularität im Kleinen (bezogen auf Systemkomponenten) unterscheiden. Eng verbunden mit dem Modulprinzip ist das Geheimnisprinzip (*information hiding*). Während das Modulprinzip es ermöglicht, dass der Anwender eines Moduls die Modulinterna sieht – und damit dieses Wissen zumindest implizit verwenden kann – verbirgt das Geheimnisprinzip die Modulinterna.

4.10 Zusammenfassung II

Um die Struktur eines Systems einfach zu halten, müssen die Systemkomponenten in sich stark gebunden sein und die Schnittstellen zwischen den Systemkomponenten eine geringe Kopplung besitzen.

Die Verständlichkeit, Lesbarkeit und Einarbeitung in ein Softwareprodukt wird durch die Prinzipien Lokalität und Verbalisierung wesentlich unterstützt.

Zwischen den aufgeführten Prinzipien gibt es Wechselwirkungen und Voraussetzungen, die beachtet werden müssen.

5 Methoden

An was denken Sie, wenn Sie hören, dass eine Person »methodisch« arbeitet? Frage

Sie assoziieren mit dem Begriff methodisch sicher, dass jemand systematisch, planmäßig und zielgerichtet vorgeht. Mit diesen Adjektiven ist der Begriff »Methode« verknüpft. Antwort

Methoden sind planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen (im Allgemeinen im Rahmen festgelegter Prinzipien, siehe »Prinzipien«, S. 25).

Über welche Eigenschaften muss Ihrer Meinung nach eine Methode verfügen, damit man sie als »Methode« bezeichnen kann? Frage

In [Jack76] werden an eine »solide Methode« folgende Anforderungen gestellt: Antwort

- Sie baut *nicht* auf die Intuition des Entwicklers.
- Sie beruht auf durchdachten Prinzipien.
- Sie ist lehrbar.
- Sie ist in der Anwendung unabhängig vom jeweiligen Anwender.
- Sie ist einfach und leicht zu verstehen.

Es lassen sich zwei allgemeine, d. h. fachunabhängige, Methoden unterscheiden – man könnte auch von Klassen von Methoden sprechen:

- Die Top-down-Methode und
- die Bottom-up-Methode.

Beide Methoden orientieren sich vorwiegend an dem Begriffspaar »abstrakt/konkret«, d. h. sie stehen in enger Beziehung zu dem Prinzip der Abstraktion (siehe »Prinzip der Abstraktion«, S. 26).

Bei einer **Top-down-Methode** wird vom Abstrakten zum Konkreten vorgegangen bzw. vom Allgemeinen zum Speziellen.

Bei einer **Bottom-up-Methode** wird vom Konkreten zum Abstrakten bzw. vom Speziellen zum Allgemeinen vorgegangen.

Analog zu diesen Methoden sind die deduktive und induktive Vorgehensweise in der Didaktik. Bei der deduktiven Vorgehensweise wird aus dem Allgemeinen der Einzelfall bzw. das Besondere hergeleitet. Bei der induktiven Vorgehensweise wird vom Speziellen zum Allgemeinen hingeführt: Ausgangspunkt können mehrere Beispiele sein, aus denen dann auf eine allgemeine Regel geschlossen wird. Analogie

Wendet man die Top-down- bzw. Bottom-up-Methode in der Softwaretechnik an, dann ist jeweils festzulegen, was unter »Top« und »Bottom« zu verstehen ist. Einige Beispiele sind in der Tab. 5.0-1 zusammengestellt.

II 5 Methoden

Konzepte	Methode(n)	»Top«	»Bottom«
Softwareentwicklung			
■ Modellierung von Daten	Top-down / Bottom-up	abstrakte, strukturierte Daten	Datenelemente oder selbst definiertes Datum
■ Modellierung von Funktionen	Top-down / Bottom-up	abstrakte, strukturierte Funktionen	elementare Funktionen
■ Modellierung von Zuständen (durch hierarchische Zustandsautomaten)	Top-down	Oberzustand	Unterszustand
■ Modellierung durch hierarchische Petrinetze	Top-down	Kanäle & Instanzen	Stellen & Transitionen
■ OOA-Modellierung	Bottom-up	Pakete bzw. Komponenten	Klassen & ihre Beziehungen
■ Identifizieren von Klassen	Top-down / Bottom-up	Klassen	Attribute, Operationen
■ Generalisierung / Spezialisierung	Top-down / Bottom-up	allgemeingültige Klassen	spezielle / spezialisierte Klassen
■ Von OOA nach OOD	Top-down	OOA-Modell	OOD-Modell, das das OOA-Modell erweitert
■ Schrittweise Verfeinerung	Top-down	abstrakte Daten & Anweisungen	Daten & Anweisungen in der gewählten Programmiersprache
Softwaremanagement			
■ Planung (durch Netzpläne / Gantt-Diagramme)	Top-down / Bottom-up	abstrakter Vorgang	detaillierter Vorgang
Softwarequalitäts-sicherung			
■ Komponenten- & Integrationstest	Bottom-up	systemferne, benutzernahe Dienste	systemnahe Basisdienste

Tab. 5.0-1: Beispiele für Top-down- und Bottom-up-Methoden in der Softwaretechnik.

Einige Methoden der Softwaretechnik orientieren sich nicht an den Begriffspaaren »abstrakt/konkret« bzw. »allgemein/speziell«, sondern an dem Begriffspaar »außen/innen«.

Bei einer **Outside-in-Methode** wird zunächst die Umwelt eines Systems modelliert und davon ausgehend die Systeminterna.

Bei einer **Inside-out-Methode** werden zunächst die Systeminterna und dann die Schnittstellen zur Umwelt eines Systems modelliert. Die Tab. 5.0-2 zeigt einige Beispiele für diese Methoden aus dem Bereich der Softwaretechnik. Charakteristisch für diese Methoden ist, dass die Modellierung, zumindest beim ersten Vorgehensschritt, auf derselben Abstraktionsebene stattfindet. Bei einer Top-down- bzw. Bottom-up-Methode werden im Gegensatz dazu bereits im ersten Vorgehensschritt verschiedene Abstraktionsebenen modelliert.

Konzept	Methode	»Outside«	»Inside«
Softwareentwicklung			
■ Modellierung eines Petrinetzes	Outside-in	Schnittstellen des Systems zur Umwelt (Erzeugen & Löschen von Objekten)	Stellen & Transitionen
■ Modellierung zeitbasierter Vorgänge (durch Sequenzdiagramme)	Outside-in	Benutzeraktionen / externe Ereignisse	Botschaften zwischen Klassen & Objekten
Softwarequalitäts-sicherung			
■ Funktions- & Strukturtest	Outside-in	Schnittstelle	Datenstrukturen & Algorithmen
■ Integrationstest	Outside-in / Inside-out	höchste Schicht	niedrigste Schicht

Alle vier Methoden sind in der Abb. 5.0-1 dargestellt. In der Praxis werden die Methoden nicht in »Reinform« angewandt. Oft wird abwechselnd »Top-down« und »Bottom-up« vorgegangen, wobei darauf zu achten ist, dass man sich in der Mitte trifft. Außerdem ist es möglich, auch in der Mitte zu beginnen und dann von der Mitte »Bottom-up« zum »Top« hin und »Top-down« zum »Bottom« hin zu entwickeln (Middle-out). Analog kann dies auch bei Outside-in und Inside-out durchgeführt werden. Der abwechselnde Einsatz konträrer Methoden wird auch als »Jo-Jo-Methode« bezeichnet. Wie die Tabellen zeigen, überwiegen in der Softwaretechnik die Top-down-Methode und die Outside-in-Methode.

*Tab. 5.0-2:
Beispiele für
Outside-in- und
Inside-out-
Methoden in der
Softwaretechnik.*

Einige Methoden kombinieren diese beiden Methoden zu einer Outside-in/Top-down-Methode (Abb. 5.0-1), so z. B. die Modellierung hierarchischer Petrinetze. Beim Einsatz des Prozessmodells **XP** (*eXtreme Programming*) wird Bottom-up-/Inside-out vorgegangen.

Alle vier Methoden haben Vor- und Nachteile:

Top-down-Methode

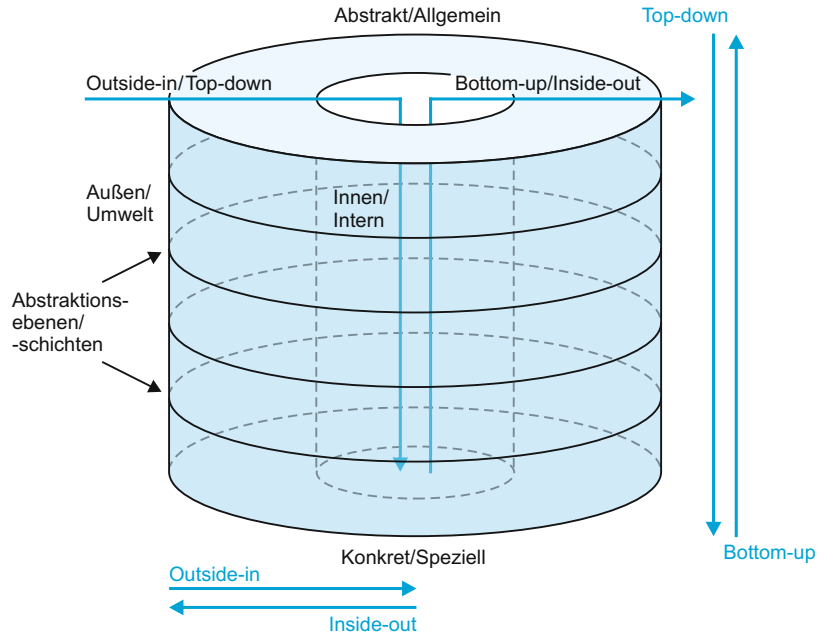
- + Konzentration auf das Wesentliche möglich, keine »Überschwemmung« mit Details.
- + Strukturelle Zusammenhänge werden leichter erkannt.
- Es wird ein hohes Abstraktionsvermögen benötigt.
- Entscheidungen werden u. U. vor sich hergeschoben, d. h. unbequeme Entscheidungen werden an tiefere Ebenen weitergereicht.
- Der »Top« ist oft nicht eindeutig zu bestimmen.

Bottom-up-Methode

- + Es ist eine konkrete Ausgangsbasis vorhanden.
- + Eine Begrenzung auf konkrete Teilgebiete ist möglich.
- + Wiederverwendbarkeit wird unterstützt.
- Übergeordnete Strukturen werden durch Details überdeckt.

II 5 Methoden

Abb. 5.0-1: Veranschaulichung der allgemeinen Methoden.



- Es muss eine breite Basis gelegt werden, um das Ziel sicher zu erreichen.

Outside-in-Methode

- + Konzentration auf die Umwelt bzw. den Kontext und die Schnittstellen des Systems.
- + Beziehungen zur Außenwelt werden nicht übersehen.
- Die Außenwelt lenkt von den internen Problemen ab.
- Die Wiederverwendbarkeit wird erschwert.

Inside-out-Methode

- + Strukturüberlegungen stehen im Mittelpunkt.
- Gefahr, dass Anforderungen der Umwelt übersehen oder zu spät erkannt werden.

Die in den Tabellen aufgeführten Beispiele zeigen, dass die Mehrzahl der **Konzepte** der Softwaretechnik entsprechend den aufgeführten allgemeinen Methoden angewandt werden (siehe »Basiskonzepte«, S. 99). Die Anwendung dieser allgemeinen Methoden sollte nicht dogmatisch, sondern tendenziell gesehen werden. Bei der Neuentwicklung von Softwareprodukten sollten die Top-down- und/oder die Outside-in-Methoden bevorzugt werden. In den seltensten Fällen betritt man bei einer Softwareentwicklung jedoch absolutes Neuland. Insbesondere wenn man die Möglichkeiten der Wiederverwendbarkeit intensiv ausnutzen will, ist es zumindest teilweise sinnvoll, Bottom-up und/oder Inside-out vorzugehen.

Unabhängig von den oben vorgestellten, allgemeinen Methoden, muss eine gute Softwareentwicklungsmethode folgende Eigenschaften besitzen:

- Das entwicklungsorientierte Vorgehen muss berücksichtigt werden.

Die einzelnen Teile eines Softwareproduktes entstehen nicht von heute auf morgen, sondern zeitlich hintereinander und/oder parallel. Eine Methode muss diesen Aspekt berücksichtigen. Eine Methode, die davon ausgeht, dass zu einem Zeitpunkt alle Erkenntnisse und Informationen vorhanden sind, ist *nicht* entwicklungsadäquat.

- Die Reihenfolge der Entwicklungsentscheidungen wird berücksichtigt.

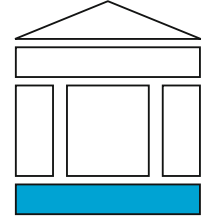
Der Softwareentwicklungsprozess erfordert ständig das Treffen von Entscheidungen. Mit jeder Entscheidung, die gefällt wird, wird der weitere Entwicklungsspielraum eingeengt. Eine gute Methode zeichnet sich dadurch aus, dass sie Entscheidungen *nur* zu den Zeitpunkten verlangt, zu denen sie auch aus fachlicher Sicht sinnvoll überhaupt möglich sind. Das bedeutet auch, dass Entscheidungen, die erst spät getroffen werden können, von der Methode auch erst zu einem späten Zeitpunkt gefordert werden.

6 Werkzeuge

Software ist Werkstoff und Werkzeug zugleich in der Softwareentwicklung. **Softwareentwicklungswerkzeuge** sind Programme, die die Entwicklung von Software unterstützen. Dieses Kapitel gibt einen Überblick über Softwareentwicklungswerkzeuge, diskutiert Kriterien und Verfahren bei der Auswahl geeigneter Werkzeuge, stellt integrative Lösungen kooperierender Werkzeuge in Form von **integrierten Entwicklungsumgebungen** vor und beschreibt Potenziale und Risiken des aktuellen Trends hin zur **modellgetriebenen Softwareentwicklung**:

- »Menschen, Methoden, Werkzeuge«, S. 59
- »Klassifikation von Werkzeugen«, S. 60
- »Integrierte Entwicklungsumgebungen«, S. 76
- »Modellgetriebene Entwicklung«, S. 79
- »Auswahlkriterien bei der Anschaffung von Werkzeugen«, S. 87
- »Evaluationsverfahren für die Anschaffung«, S. 90

[FMP+87]



i

Literatur

6.1 Menschen, Methoden, Werkzeuge

Die Verwendung von Werkzeugen ist ein wesentliches Kennzeichen des ingenieurmäßigen Vorgehens, das in der Softwaretechnik angestrebt wird. Sie sollen die Effektivität und Effizienz von Entwicklern erhöhen. Ein Werkzeug ist jedoch immer nur ein Hilfsmittel, nicht schon die Lösung selbst. Effizienz und Effektivität eines Werkzeugs wird bestimmt durch denjenigen, der das Werkzeug anwendet, die Adäquatheit für die Aufgabe, die derjenige damit lösen möchte, sowie die Methode, nach der derjenige dabei vorgeht. Die Abb. 6.1-1 setzt diese Aspekte in einen Kontext.

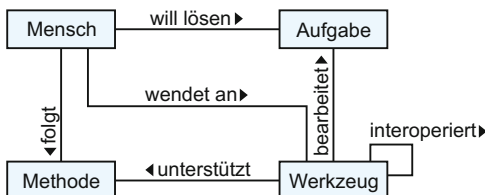


Abb. 6.1-1:
Zusammenhang
von Mensch,
Aufgabe, Methode
und Werkzeug.

II 6 Werkzeuge

Mensch & Werkzeug	Der Mensch wendet das Werkzeug an, um seine Aufgabe zu erledigen. Dazu muss er das Werkzeug richtig anwenden können. So sehr ein Hammer einen Handwerker in seiner Arbeit unterstützt, der Hammer ist nichts ohne das Geschick des Handwerkers.
Methode & Werkzeug	Ein Werkzeug ist nichts ohne die Methodik, die beschreibt, wann und wie man das Werkzeug sinnvoll einsetzt. Ein Werkzeug ist stets Teil einer Methode. Umgekehrt sind Methoden oft nur dann praktisch umsetzbar, wenn sie durch Werkzeuge geeignet unterstützt werden. Werkzeuge und Methoden bedingen sich also wechselseitig.
Aufgabe & Werkzeug	Ein Werkzeug muss zur Aufgabe passen. Wenn der Handwerker einen Balken absägen muss, wird ihm der Hammer nichts nutzen.
Werkzeug & Werkzeug	Bei der Lösung komplexer Aufgaben sind oft mehrere Werkzeuge beteiligt. Der Handwerker hat dazu einen ganzen Werkzeugkasten. Damit Werkzeuge bei der Softwareentwicklung zusammenwirken können, müssen sie interoperabel sein, das heißt, Daten untereinander austauschen und gegenseitige Kontrolle ausüben können. Mensch, Aufgabe, Methode und die Menge eingesetzter Werkzeuge müssen also immer als Ganzes betrachtet werden. Dieses Kapitel konzentriert sich auf Werkzeuge .

6.2 Klassifikation von Werkzeugen

Einzelne Entwicklungswerkzeuge sind vergänglich, aber der Typus »Werkzeug« wird überdauern. Aus diesem Grund in diesem Buch nur eine Klassifikation der Werkzeugtypen vorgestellt. Einzelne Werkzeuge werden zur Erläuterung als Beispiele genannt. Die Übersicht kann jedoch keine erschöpfende Aufzählung aller verfügbaren Werkzeuge sein.

Essenzielle Werkzeuge	Softwareentwicklungswerkzeuge gibt es seit Beginn der Softwareentwicklung. Die frühen Entwicklungswerkzeuge waren essenzieller Art. Zu den essenziellen Werkzeugen zählen der Editor , mit dessen Hilfe der Entwickler sein Programm verfasst, der Übersetzer , der das Programm in eine ausführbare Form transformiert, sowie der Binder , der mehrere separat übersetzte Programmeinheiten zu einer ausführbaren Einheit zusammenfasst. Im Falle interpretierter Sprachen übernimmt ein Interpreter die Aufgaben des Übersetzers und Binders zur Laufzeit.
CASE-Werkzeuge	Zur Entwicklung großer Programme sind weitergehende Werkzeuge hilfreich, die Aktivitäten über die reine Programmierung unterstützen. Werkzeuge für das so genannte CASE (<i>Computer Aided Software Engineering</i>) unterstützen neben der Programmierfähigkeit auch die Spezifikation, den Entwurf und den Test von Software. Die Entwicklung von CASE-Werkzeugen hat in den achtziger Jahren des letzten Jahrhunderts seinen Höhepunkt erreicht. Das Interesse kühl-

te dann nach anfänglichem »Hype« wieder ab, nachdem sich die überzogenen Erwartungen an den Zuwachs von Effektivität und Effizienz nicht einstellten, beziehungsweise offensichtlich wurde, dass man die technischen Herausforderungen unterschätzt hatte. Viele der Ideen aus dieser Zeit haben jedoch überlebt, auch wenn der Begriff CASE heute kaum mehr gebräuchlich ist. Die **modellgetriebene Entwicklung** beispielsweise ist keine Erfindung unserer Zeit.

Die Entwicklung von Werkzeugen zur Unterstützung der Softwareentwicklung ist seit dieser Zeit nicht stehen geblieben. Insbesondere die Open-Source-Gemeinde hat einen wesentlichen Beitrag zur Entwicklung einer großen Anzahl von Werkzeugen geleistet. Die Breite und der Spezialisierungsgrad heutiger Werkzeuge, die zu einem großen Teil auch frei verfügbar sind, überragt das Angebot aus den achtziger Jahren bei Weitem.

Entwicklungs-
werkzeuge heute

Weil das Angebot an Werkzeugen heute kaum mehr zu überschauen ist, ist es hilfreich, sie nach ihrem Typus zu klassifizieren. Entwicklungswerkzeuge im Allgemeinen lassen sich gliedern anhand des Gegenstands, den sie bearbeiten, und der Operationen, die sie unterstützen, das heißt, in welcher Weise sie auf den Gegenstand einwirken:

- »Von Werkzeugen behandelte Artefakte«, S. 62
- »Von Werkzeugen unterstützte Operationen«, S. 62
- »Werkzeuge zur Kollaboration und Kommunikation«, S. 73
- »Unterstützung von Prozessmodellen und Methoden«, S. 75

Neben den Aktivitäten im Entwicklungsprozess, die unmittelbare Zwischenergebnisse – meist Dokumente – erstellen, existieren auch Aktivitäten, die phasenübergreifend sind, die also vom Projektbeginn bis zu dessen Ende reichen können. Dazu gehören Planung, Kommunikation und Projektmanagement genauso wie Konfigurations- und Qualitätsmanagement. Auch diese Aktivitäten werden durch Werkzeuge unterstützt. Primär werden hier Werkzeuge betrachtet, die die Erstellung von die Software beschreibenden Dokumenten und Programmen direkt zum Ziel haben. Dazu zählen auch Werkzeuge für den Test. Der Test dient der **dynamischen Prüfung**, inwieweit das Programm seiner Spezifikation genügt. Werkzeuge unterstützen hier die Erstellung, Durchführung, Protokollierung und Auswertung von Testfällen. Zu den Entwicklungswerkzeugen gehören insbesondere aber auch Werkzeuge, die helfen, existierende Programme weiter zu entwickeln. Diese Phase der Entwicklung nennt man Softwarewartung oder Softwareevolution. Auch diese Werkzeuge sind Gegenstand der Betrachtung, da sie die Änderung der Anforderungsspezifikation, des Entwurfes, des Quellcodes oder der Testfälle unterstützen.

II 6 Werkzeuge

6.2.1 Von Werkzeugen behandelte Artefakte

Im Laufe einer Softwareentwicklung entstehen verschiedene **Artefakte** bzw. Dokumente als Zwischen- oder Endergebnisse. Sie beschreiben unterschiedliche Aspekte des Programms. Sie können grob in die folgenden Abstraktionsebenen eingeteilt werden.

Spezifikation Die Spezifikation beschreibt, *was* zu entwickeln ist. Werkzeuge helfen hier zum Beispiel bei der Erfassung, Verwaltung, Pflege, Modellierung und Prüfung der Anforderungen (siehe »Requirements Engineering«, S. 433). Bei einer objektorientierten Modellierung werden heute UML-Werkzeuge eingesetzt, oft integriert in Entwicklungs-umgebungen.

Zur Erhebung und Überprüfung von Anforderungen werden häufig Prototypen entwickelt. Die rasche Entwicklung von Prototypen (*Rapid Prototyping*) wird auch von Werkzeugen unterstützt. Dazu zählen interpretierte Programmiersprachen mit sehr kurzen Zyklen des Schreibens, Übersetzens und Ausführens von Programmen genauso wie etwa Werkzeuge, mit denen man rasch eine vorzeigbare Benutzungsoberfläche interaktiv erstellen kann.

Entwurf Der Entwurf beschreibt, *wie* die Softwarearchitektur aufgebaut sein soll. Dazu werden häufig ebenfalls UML-Werkzeuge verwendet, wobei das erstellte **OOA-Modell** zum **OOD-Modell** verfeinert wird.

Implementierung Für die Implementierung der Softwarearchitektur werden neben den essenziellen Werkzeugen für das Editieren, Übersetzen und Bauen von Programmen zum Beispiel Werkzeuge zur Prüfung, **Restrukturierung** oder **Generierung** von Quellcode eingesetzt. Weitere beobachten die Ausführung des Programms. Dazu gehören **Debugger** genauso wie **Profiler**, mit denen man die Laufzeiten und den Speicherbedarf einer Programmausführung bestimmen kann.

Meta-Information Für alle Ebenen können zwei Arten von Informationen unterschieden werden. Die eine betrifft den Inhalt des Dokuments. Die andere betrifft die Dokumente selbst, ihren Entwicklungsprozess oder die dafür verwendeten Ressourcen. Dazu gehören zum Beispiel der Status eines Dokuments, die Autoren oder die Änderungshistorie. Diese Informationen werden **Meta-Informationen** genannt. Meta-Information ist wichtig, um die Dokumente zu verstehen und weiter zu verarbeiten.

6.2.2 Von Werkzeugen unterstützte Operationen

Im Folgenden wird die Menge abstrakter Operationen auf den jeweiligen Artefakten bzw. Dokumenten beschrieben, die durch Werkzeuge implementiert werden. Die Abb. 6.2-1 enthält hierzu eine Übersicht als Matrix von Operationen und den Abstraktionsebenen der Artefakte, die sie zum Gegenstand haben.

6.2 Klassifikation von Werkzeugen II

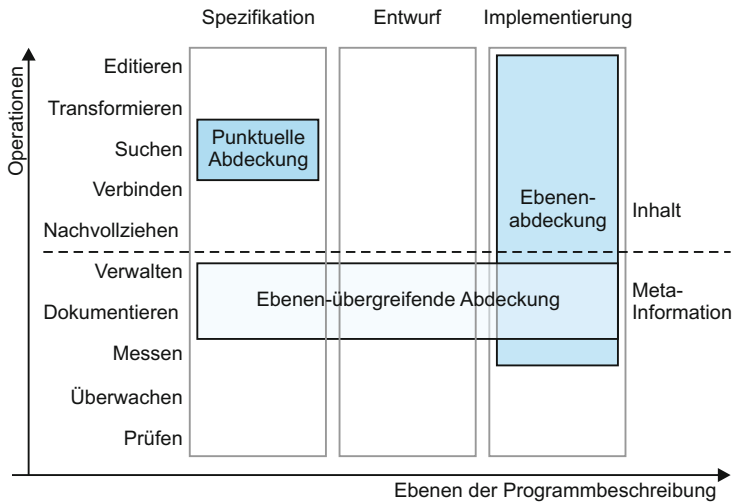


Abb. 6.2-1:
Abdeckungsarten
von Werkzeugen.

Werkzeuge können anhand ihrer Abdeckung dieser Matrix charakterisiert werden. Bei der **punktuellen Abdeckung** unterstützt ein Werkzeug nur eine Operation auf einer Ebene. Bei der **ebenen-übergreifenden Abdeckung** werden dieselben Operationen über verschiedene Ebenen implementiert. Dateibasierte Versionskontrollsysteme erlauben zum Beispiel die unspezifische Verwaltung und Änderungsdokumentation von Artefakten, die in Dateien abgelegt sind, über alle Ebenen hinweg. Andere Werkzeuge decken **mehrere Operationen auf derselben Ebene** ab wie zum Beispiel integrierte Entwicklungsumgebungen.

Bei der Auswahl von Werkzeugen versucht man, alle relevanten Operationen auf allen Ebenen mit möglichst wenig Überlappung zwischen den Werkzeugen abzudecken. Die **Komplementarität** ist deshalb ein wichtiges Kriterium bei der Auswahl von Werkzeugen.

Im Folgenden werden die verschiedenen Operationen näher beschrieben.

Editieren

Editoren unterstützen das Erstellen und Bearbeiten eines Artefakts durch den Menschen. Sie existieren für alle Aktivitäten einer Softwareentwicklung. Für Code gibt es generische Editoren, mit denen man beliebige Texte eingeben kann. Manche dieser generischen Editoren haben aber auch einen unterstützenden Modus für das Editieren von Programmen. Es gibt jedoch auch Editoren, die auf das Erstellen von Programmen spezialisiert sind. Diese heben syntaktische Elemente hervor (Syntax-Highlighting), haben Vorlagen für syntaktische Konstrukte, können gültige Bezeichner automatisch vervollständigen oder registrieren Syntaxfehler während des Tippens.

II 6 Werkzeuge

Syntaktische Editoren stellen sicher, dass man nur syntaktisch korrekte Programme eingibt. Dazu ermitteln sie anhand des Kontextes, welche weiteren Konstrukte an der Einfügestelle überhaupt möglich sind und lassen keine ungültigen Eingaben zu.

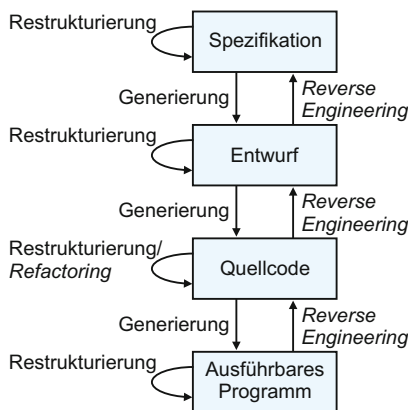
Das Editieren kann sowohl traditionell *textuell* geschehen, wie das Schreiben von Quellcode, als auch *grafisch*, wie beispielsweise bei einem UML-Diagrammeditor. Grafische Editoren werden meist für abstraktere Modelle des Entwurfs verwendet, während Programme meist mit Hilfe textueller Editoren bearbeitet werden. Die visuelle Programmierung erlaubt es aber auch, Programme in grafischer Form zu formulieren.

Anforderungen werden in der Praxis meist informell in natürlicher Sprache und damit textuell formuliert. Es gibt jedoch auch spezielle Werkzeuge für die systematische Erfassung von Anforderungen, die über Editoren verfügen, mit denen Anforderungen strukturiert erfasst werden können. Dazu gehören Werkzeuge wie Caliber-RM, CARE, DOORS, in-Step oder Requisite Pro.

Transformieren

Transformieren unterscheidet sich vom Editieren dadurch, dass das Werkzeug selbst und nicht der Mensch das Dokument bearbeitet. Mögliche **Transformationen** können anhand der Richtung unterschieden werden. Die Abb. 6.2-2 beschreibt diese Richtungen schematisch.

Abb. 6.2-2:
Transformationen.



- Eine **Generierung** ist eine Transformation von einer höheren in eine niedrigere Ebene. Hierzu gehören die traditionellen Werkzeuge Übersetzer, Binder und die beim Programmbau unterstützten Werkzeuge – so genannte Build-Werkzeuge. Außerdem gehören dazu auch codegenerierende Werkzeuge, die bei der modell-

getriebenen Entwicklung zum Einsatz kommen, genauso wie die so genannten GUI-Builder, mit denen grafische Benutzungsoberflächen interaktiv erstellt werden können.

Entwickler haben häufig die Aufgabe, strukturierte Texte einzulesen. Die Struktur der Texte wird mit einer Grammatik beschrieben. Mit Hilfe eines Parser-Generators, wie zum Beispiel Yacc, kann der Entwickler die den Texten zugrunde liegende Grammatik spezifizieren und daraus Quellcode für einen Parser, der die syntaktische Analyse implementiert, generieren lassen. Dieses Vorgehen ist in der Regel weit effizienter und effektiver als die Entwicklung des Parsers von Hand. Bei Änderungen der Grammatik sind auch alle Anpassungen viel einfacher. Beispiel

- Eine **Restrukturierung** ist die Transformation auf derselben relativen Ebene. Diese Werkzeuge überführen Dokumente, die auf derselben Abstraktionsebene anzusiedeln sind. Auf der Code-Ebene ist diese Restrukturierung auch als **Refactoring** bekannt.

Integrierte Entwicklungsumgebungen, wie beispielsweise Eclipse, können dem Programmierer die notwendigen Editieraufgaben beim Refactoring abnehmen. Wenn der Programmierer beispielsweise eine Methode umbenennen möchte, dann muss er lediglich den neuen Namen festlegen. Die Änderung des Namens sowohl bei der Deklaration als auch bei allen Aufrufstellen der Methode übernimmt Eclipse automatisch. Die manuelle Änderung wäre bei großen Programmen sehr mühsam. Beispiel

- Beim **Reverse Engineering** findet die Transformation von einer niederen in eine höhere Ebene statt. Zum *Reverse Engineering* zählen nicht nur Disassembler, die Binärcode in Quellcode überführen, sondern zum Beispiel auch Werkzeuge, die Architektur-sichten aus dem Code extrahieren und visualisieren. Mit Hilfe dieser Werkzeuge kann man verlorenes Wissen wiedergewinnen und sich dann einen Überblick über die Zusammenhänge des Programms verschaffen. Es gibt zum Beispiel Werkzeuge, die Abhängigkeiten zwischen Klassen aus Programmen extrahieren und visualisieren.

Suchen

Wenn keine expliziten Querbezüge existieren, sind Werkzeuge wertvoll, die bei der Suche nach impliziten Bezügen helfen. Meist kann man ein Muster formulieren und das Werkzeug fahndet nach allen Stellen, die diesem Muster genügen.

II 6 Werkzeuge

Beispiel Eines der am häufigsten eingesetzten Werkzeuge für die Suche in der Programmentwicklung ist *grep*. Dieses Werkzeug erlaubt es, große Textmengen nach einem regulären Textmuster abzusuchen. Seine Verwendung ist einfach und kann auf beliebigen Text angewandt werden. Mit `grep mymethod *.java` kann man zum Beispiel alle Java-Dateien eines Verzeichnisses nach dem Methodennamen `mymethod` durchsuchen. Wenn man aber nach allen Verwendungen einer Variablen `i` sucht, dann verwendet man besser Werkzeuge, die Quellcode syntaktisch und semantisch analysieren. In diesem Falle hätte das Muster einen syntaktischen Aspekt (das Vorkommen eines Bezeichners) und einen semantischen (mit einem bestimmten Namen und in einem bestimmten Gültigkeitsbereich).

Visualisierungswerkzeuge gehören auch zu den Werkzeugen, die das Suchen unterstützen. Statt dass das Werkzeug selbst nach einem Muster sucht, bereitet es vielmehr die zu durchsuchende Datenmenge grafisch so auf, dass der Betrachter selbst die Suche vornehmen kann. Visualisierungswerkzeuge eignen sich dann gut, wenn es keinen einfachen Algorithmus gibt, um die Antwort auf eine Frage zu finden, beziehungsweise auch dann, wenn man schon die Frage selbst nicht präzise in Form eines Musters stellen kann.

Verbinden

Entwickler können unterschiedliche Artefakte miteinander verbinden, wenn sie Abhängigkeiten haben. Diese Querbezüge können innerhalb eines Artefakts sein oder auch zwischen Artefakten verlaufen. Dabei können die Artefakte derselben Abstraktionsebene (zum Beispiel zwei Java-Klassen) oder verschiedener Ebenen angehören (zum Beispiel eine Java-Klasse, die eine Funktionalität implementiert, die in einem bestimmten Abschnitt der Spezifikation beschrieben ist). Beim Verständnis und bei der Änderung müssen diese Querverweise beachtet werden. Deshalb gibt es Werkzeuge, mit deren Hilfe man die verschiedenen Artefakte oder Artefakteile durch Querverweise explizit verbinden kann. Diese Querverweise werden dann von dem Werkzeug verwaltet. Das Werkzeug unterstützt den Entwickler bei der Navigation und bei der Analyse von Auswirkungen von Änderungen. Zu diesen Werkzeugen zählen beispielsweise so genannte Traceability-Werkzeuge, die Spezifikation, Entwurf und Code verbinden können.

Beispiel Werkzeuge wie Caliber-RM, CARE, DOORS, in-Step oder Requisite Pro für das *Requirements Engineering* können zum Beispiel Anforderungen und ihre Bezüge zu anderen Anforderungen sowie Code verwalten. Auf diese Weise kann man im Falle der Änderung einer Anforderung

rung die anzupassenden Artefakte finden. Umgekehrt kann der Programmierer schnell herausfinden, für welche Anforderung ein bestimmter Programmteil verantwortlich ist.

Nachvollziehen

Tests liefern im Falle eines Fehlers nur die Aussage, dass das Programm *nicht* korrekt ist. Sie erklären jedoch nicht, wie es zu diesem Fehler kommt. Dazu ist es notwendig, die tatsächlichen und erwarteten Zusammenhänge im Programm herauszufinden und nachzuvollziehen. Hierzu kann man ein Programm **dynamisch** oder statisch untersuchen. Bei der dynamischen Untersuchung wird das Programm ausgeführt, bei der statischen nur der Quelltext betrachtet. Debugger unterstützen eine dynamische Verfolgung der tatsächlichen Programmausführung. Sie erlauben, das Programm an definierten Stellen anzuhalten, Werte von Variablen auszulesen und sogar zu manipulieren. Auf diese Weise kann man das tatsächliche Verhalten für eine bestimmte Eingabemenge untersuchen. Weitere Werkzeuge dieser Kategorie sind solche, die ein *Logging* der Ausführung oder *Capture&Replay* unterstützen. Beim *Logging* kann der Entwickler durch entsprechende Einfügungen in seinen Code das Verhalten des Programmes aufzeichnen. *Capture&Replay*-Werkzeuge helfen, die Interaktion mit einer interaktiven Benutzungsoberfläche aufzuzeichnen und wieder automatisiert abzuspielen.

Statische Werkzeuge können die Zusammenhänge unabhängig von einer konkreten Eingabe allein anhand des Quelltexts ermitteln. Werkzeuge zur Untersuchung statischer Zusammenhänge implementieren beispielsweise das so genannte Program-Slicing. Damit wird ein Programm in »Scheiben« zerlegt, die nur aus den Programmteilen bestehen, die von einander abhängig sind. Das Werkzeug *CodeSurfer* ermittelt beispielsweise alle Anweisungen eines Programms, die von der Änderung einer Variablen an einer bestimmten Stelle im Programmtext beeinflusst werden, beziehungsweise alle Anweisungen, die eine bestimmte Programmstelle beeinflussen. Durch Program-Slicing kann man also ein Programm auf die wesentlichen Teile reduzieren, die für eine bestimmte Fragestellung von Bedeutung sind.

Verwalten & Versionieren von Änderungen

Werkzeuge für das Konfigurationsmanagement helfen beim Verwalten und Versionieren von Artefakten. Sie erlauben es, Änderungen zwischen Versionen und Varianten festzuhalten und zu dokumentieren. Man kann Unterschiede zwischen Versionen und Varianten bestimmen und die Unterschiede wieder zusammenführen. Versions-

II 6 Werkzeuge

kontrollsysteme, die das Konfigurationsmanagement technisch unterstützen, sind nahezu unverzichtbar bei Softwareprojekten – insbesondere bei langlaufenden und solchen mit mehr als einem Entwickler.

Dokumentieren

Bei einem Artefakt interessiert man sich häufig nicht nur für den Inhalt selbst, sondern auch für Aspekte seiner Erstellung oder Änderung oder weitere Information, die über den Inhalt hinaus gehen. Quellcode beispielsweise beschreibt exakt, wie sich das Programm verhalten soll. Er beschreibt jedoch nicht, warum sich das Programm so verhalten soll, welche Annahmen getroffen wurden, welche Alternativen erwogen wurden und wer den Quellcode geschrieben hat. Diese Informationen sind jedoch auch für das Verstehen und die Änderung bedeutsam. Manche dieser Informationen werden von den Entwicklern im Quellcode als Kommentare festgehalten. Wenn diese im Quelltext strukturiert formuliert werden, können Dokumentationswerkzeuge sie leicht auslesen und verarbeiten.

Beispiel In Java gibt es keine klare Trennung zwischen der Signatur einer Klasse und ihrer Implementierung. Wenn man sich nur für die öffentlichen Methoden und Attribute interessiert, kann man mit Hilfe des Werkzeugs Javadoc zum Beispiel HTML-Seiten generieren, die nur diesen Teil der Klasse enthalten. Die Bedeutung der Methoden und ihrer Parameter können durch spezielle Markierungen innerhalb der Kommentare beschrieben werden. Das Werkzeug kann dann zum Beispiel die Dokumentationen aller Parameter einer Methode individuell auflisten.

Meta-Informationen, die nicht selbst im Dokument festgehalten werden, können von Dokumentationswerkzeugen erfasst werden. Das Versionskontrollsystem verwaltet zum Beispiel die Änderungen, ihr Änderungsdatum und den Namen des Autors sowie den Kommentar der Änderung durch den Entwickler. Bekannte Fehler und Änderungswünsche werden in einem Issue-Tracking-System wie Bugzilla oder FlySpray erfasst. Der Autor des Änderungswunsches, das Datum, die Dringlichkeit, der Status und die Zuständigkeit können hier verwaltet werden.

Messen

Um ein Artefakt objektiv zu beurteilen, ist es hilfreich, die Eigenschaften des Artefakts zu quantifizieren. Metriken sind Abbildungen von Eigenschaften eines Artefakts auf eine Zahl oder einen Vektor von Zahlen. Je formaler das Artefakt ist, desto besser lassen sich

Metriken erheben. Bei einer unstrukturierten Spezifikation kann man allenfalls die Anzahl von Wörtern oder Zeilen bestimmen. Ist die Spezifikation halbformal formuliert, das heißt, wenn eine Syntax für die Anforderungen definiert ist, sind die Anforderungen als solche zu erkennen und man kann sie zählen. Da Programme formale Dokumente sind, das heißt sowohl ihre Syntax als auch ihre Semantik definiert ist, lassen sie sich leichter vermessen. Hier lassen sich primär zwei Arten von Werkzeugen unterscheiden, die solche Produktmetriken erheben.

Statische Werkzeuge liefern Metriken, die die Struktur des Programms betreffen. Sie erheben diese Metriken anhand des Quelltextes, ohne dass das Programm dazu ausgeführt werden muss. Typische statische Code-Metriken sind zum Beispiel die Anzahl von Code-Zeilen einer Methode, die Tiefe der Vererbungshierarchie einer Klasse oder die so genannte McCabe-Komplexität, die Bedingungen im Programm zählt.

Dynamische Werkzeuge erfassen Aspekte des Laufzeitverhaltens. Mit Hilfe von **Profilern** kann man die Dauer und Häufigkeit der Ausführung bestimmter Code-Teile vermessen. Mit Code-Coverage-Werkzeugen kann der Grad der Abdeckung des Tests bestimmt werden, also der Anteil der Anweisungen, die durch Tests ausgeführt wurden.

Aspekte der Ausführung eines Programmes lassen sich zum Beispiel mit Valgrind bestimmen. Valgrind ist eine Plattform zur dynamischen Analyse von Programmen. Das bereits übersetzte Programm wird hierzu in einer virtuellen Maschine ausgeführt, die die Maschineninstruktionen des Programms interpretiert und ausführt und dabei dynamische Daten gewinnt. Die Plattform bietet unterschiedliche Werkzeuge, mit denen man Speicherlecks auffinden, Häufigkeit und Dauer von Methodenaufrufen messen sowie den Speicherbedarf bestimmen kann, ohne dass dazu das Programm selbst modifiziert oder in einer bestimmten Weise dazu vorher vorbereitet werden müsste. Beispiel

Es gibt jedoch auch statische Werkzeuge, die Aussagen über das erwartete Laufzeitverhalten eines Programmes machen können. Der Vorteil statischer Werkzeuge ist, dass sie die Eigenschaften eines Programmes im Allgemeinen herleiten können, während dynamische Werkzeuge ihre Ergebnisse allein auf die betrachteten Eingabewerte stützen. Das statische Werkzeug Absint/aiT ist zum Beispiel in der Lage, die maximale Ausführungsdauer eines Programmes sicher abzuschätzen. Dies ist für Echtzeitsysteme wichtig, bei denen es aus Gründen der Sicherheit notwendig ist, sicher vorherzusagen, ob alle Fristen zur Laufzeit eingehalten werden können.

II 6 Werkzeuge

Verfolgen & überwachen

Wenn Merkmale eines Artefakts vermessen sind, dann lässt sich die zeitliche Entwicklung dieser Merkmale leicht **überwachen**. So genannte Software-Armaturenbreter (auch Leitstände, Cockpits und *Dashboards* genannt) führen unterschiedliche Softwaremaße zusammen, verdichten und visualisieren sie und werten sie aus. Dazu werden neben Produktmaßen, wie interne Code-Maße und Fehler- oder Performanzdaten, auch Prozessmaße zusammengefasst, die den Entwicklungsprozess und die dafür notwendigen Ressourcen beschreiben. Eine Zusammenführung dieser Maße mit ihren zeitlichen Verläufen – wie etwa im Axivion Software Cockpit in der Abb. 6.2-3 – bietet ein ganzheitliches Bild der Entwicklung über verschiedene Projekte hinweg.

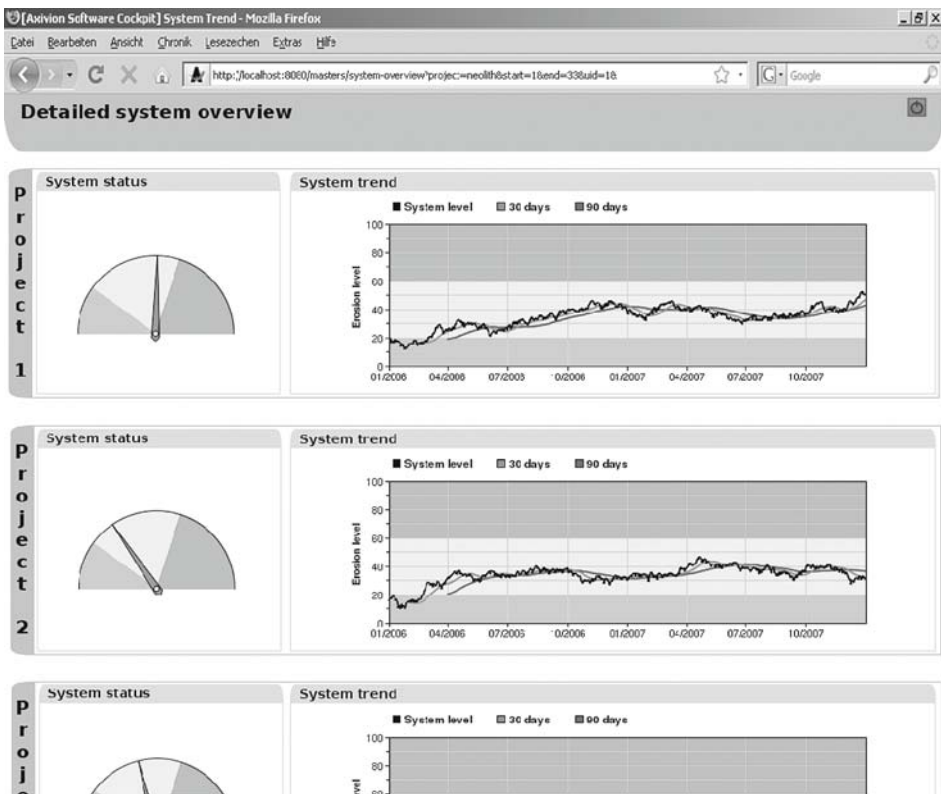


Abb. 6.2-3: Mit statistischen Methoden und Techniken des *Data Minings* lassen sich diese Daten auch auswerten und Trends und Prognosen ermitteln.
Software-Armaturen Brett mit Zeitverläufen von Projektkennzahlen.

Prüfen

Die Prüfung von Artefakten durch den Menschen ist eine teure Angelegenheit. Deshalb liegt es nahe, Prüfungen durch Werkzeuge durchführen zu lassen. Der Test ist ein solcher Versuch. Auch wenn bekannt ist, dass Tests nur die Anwesenheit, aber niemals die Abwesenheit von Fehlern nachweisen können, haben automatisierte Tests den großen Vorteil, dass sie sich leichter wiederholen lassen als eine menschliche Prüfung. Diese Wiederholbarkeit wird zum Beispiel durch Werkzeuge ausgenutzt, die nächtliche *Builds* und Tests unterstützen. Hierzu wird nachts die aktuelle Version des Programms aus dem Versionskontrollsystem übersetzt und automatisierten Tests unterzogen. Testwerkzeuge wie JUnit führen die Tests aus und protokollieren ihre Ergebnisse. Testcoverage-Werkzeuge wie gcov bestimmen, welche Code-Teile durch die Tests ausgeführt wurden, um die Abdeckung der Tests zu bestimmen. Ein Werkzeug wie Cruise-Control steuert diesen ganzen Build- und Testprozess und bereitet die Ergebnisse auf. Am nächsten Morgen erfahren die Entwickler, welche Tests Fehler gefunden haben.

Tests sind dynamische Prüfungen. Sie führen das Programm aus, um es zu prüfen. Statische Werkzeuge prüfen das Programm allein anhand des Quelltexts, ohne das Programm dazu ausführen zu müssen, und können daher Aussagen über das Programmverhalten im Allgemeinen machen. Selbstverständlich können statische Werkzeuge nur einfache Prüfungen übernehmen. Viele interessante Fragen über Programme sind aber prinzipiell nicht von einem Werkzeug entscheidbar. Vielfach kann man aber durch ein Werkzeug zumindest potenzielle Fehler finden lassen, die dann dem menschlichen Experten zur endgültigen Beurteilung vorgelegt werden können.

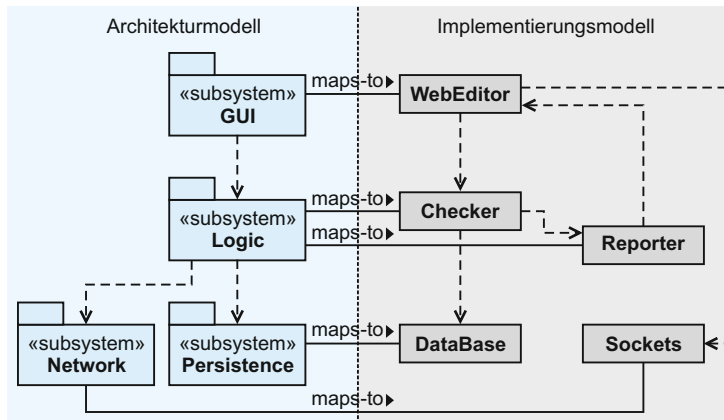
Bei statischen Prüfwerkzeugen ist zu unterscheiden, inwieweit sie vollständige Ergebnisse liefern. Ein Ergebnis ist *vollständig*, wenn alle tatsächlichen Fehler gefunden werden. Ein Werkzeug, das einem solchen Ansatz folgt, ist beispielsweise Polyspace. Wegen der von einem Werkzeug prinzipiell unentscheidbaren Fragen, die man beantworten müsste, ob ein Fehler vorliegt, sind solche Werkzeuge zu einer Überschätzung gezwungen. Es kann dann passieren, dass man sehr viele so genannte **Falschpositive** erhält. Das sind Hinweise eines Werkzeugs auf Fehler, die keine sind. Je weniger Falschpositive ein Werkzeug liefert, desto höher ist seine Präzision. Manche Werkzeuge favorisieren Präzision und geben deshalb keine Garantie ab, dass alle Fehler gefunden werden, versuchen aber, Fehler mit einer hohen Wahrscheinlichkeit zu melden. Zu diesen Werkzeugen gehören kommerzielle Systeme wie Coverity und Systeme aus dem Open-Source-Bereich wie Lint, PMD und Findbugs. Bei allen diesen Werkzeugen muss man sich darüber im Klaren sein, dass sie nur

II 6 Werkzeuge

typische Programmierfehler finden können. Sie prüfen nicht gegen eine Spezifikation der Anforderungen und können deshalb keine Anwendungsfehler finden.

Aber nicht nur der Quellcode lässt sich prüfen. Für die Einhaltung von Vorgaben durch eine Architektur gibt es Werkzeuge wie die »Axivion Bauhaus Suite«, die überprüfen, inwieweit Architekturmodell und Implementierung übereinstimmen. Voraussetzung ist ein statisches Architekturmodell, das die Architekturkomponenten und ihre erwarteten Abhängigkeiten beschreibt. Die Abb. 6.2-4 enthält als Beispiel hierfür drei Architekturkomponenten und ihre Abhängigkeiten im Stile einer typischen Schichtenarchitektur, bei der die grafische Benutzungsoberfläche (GUI) auf die Logik und diese auf die Persistenzschicht zugreifen soll. Auf der rechten Seite befindet sich ein Implementierungsmodell, das solche Werkzeuge aus dem Quelltext durch eine statische Analyse in Form von Implementierungskomponenten – in diesem Beispiel etwa Klassen – und ihre Abhängigkeiten extrahieren. Um die Implementierung gegen die Architekturvorgabe vergleichen zu können, ist es notwendig, Implementierungsmodell und Architekturmodell auf einander abzubilden, was in der Abb. 6.2-4 durch die Beziehung `maps-to` dargestellt ist. Die Abbildung kann eine 1:* sein, d. h. mehrere Implementierungskomponenten können auf dieselbe Architekturkomponente abgebildet werden, aber jede Implementierungskomponente nur auf eine Architekturkomponente.

Abb. 6.2-4: Architekturprüfung.



Mit Hilfe der Tab. 6.2-1 kann das Werkzeug dann automatisiert die folgenden Vergleiche anstellen:

- Eine **Konvergenz** bezeichnet eine Abhängigkeit in der Architektur, die eine Entsprechung in der Implementierung findet. Die Konvergenzen in der Abb. 6.2-4 sind die zwischen den Komponenten GUI und Logic sowie Logic und Persistence.

- Eine **Absenz** ist eine Abhängigkeit in der Architektur, die keine Entsprechung in der Implementierung hat. Die Abhängigkeit von Logic zu Network gehört in diese Kategorie.
- Eine **Divergenz** ergibt sich, wenn die Implementierung eine Abhängigkeit aufweist, die nicht in der Architektur vorgesehen ist. Verletzungen von dieser Art sind in der Abb. 6.2-4 zwischen Reporter und WebEditor sowie zwischen WebEditor und Sockets zu erkennen.

Abhängigkeit	in Implementierung vorhanden	nicht vorhanden
in Architektur gewollt	Konvergenz	Absenz
nicht gewollt	Divergenz	Konvergenz

Tab. 6.2-1:
Mögliche
Vergleiche.

6.2.3 Werkzeuge zur Kollaboration und Kommunikation

Bei der globalen Softwareentwicklung wird Software an geografisch verschiedenen Standorten – oft sogar auf verschiedenen Kontinenten – verteilt entwickelt. Hier ist die gute Unterstützung von Kollaboration¹ und Kommunikation von zentraler Bedeutung für den Erfolg. Kollaboration und Kommunikation ist in der Softwareentwicklung von immenser Bedeutung, weil hier viele verschiedene menschliche Wissensträger zusammenarbeiten müssen. Sie misslingen auch schon im direkten Gespräch von Angesicht zu Angesicht. Sind diese Personen räumlich getrennt, steigt das Risiko des Missverständnisses unmittelbar. Umso mehr ist die Kommunikation gefährdet, wenn Kontinente zwischen Entwicklern liegen.

Mit Hilfe von Werkzeugen zur Unterstützung von Kommunikation und Kollaboration können die Risiken der geografisch und zeitlich getrennten Entwicklung zu einem gewissen Grade gemindert werden. Ganz ausgeschlossen werden können sie damit jedoch nicht. Denn erstens ist ein Werkzeug nur ein Mittel zur Kommunikation und Kollaboration und nicht die Kommunikation und Kollaboration selbst und zweitens sind auch soziokulturelle Unterschiede bei der global verteilten Entwicklung mit einem Werkzeug allein nicht zu überwinden.

Herausforderungen

Eine Reihe der im Kapitel »Von Werkzeugen unterstützte Operationen«, S. 62, erwähnten Werkzeuge unterstützen bereits die Kollaboration. Dazu gehören insbesondere die Werkzeuge des Softwarekonfigurationsmanagements, wie CVS (*Concurrent Version System*) oder Subversion, die eine zentrale Verwaltung einer versionierten Dokumentenablage bieten, auf die Entwickler zeitgleich oder versetzt

Teamfähigkeit

¹Der englische Begriff *Collaboration* bedeutet sinngerecht übersetzt »Zusammenarbeit«, während der deutsche Begriff »Kollaboration« die aktive Unterstützung einer feindlichen Besatzungsmacht gegen die eigenen Landsleute bedeutet. Da die falsche Übersetzung des Begriff Kollaboration jedoch in der Informatik gebräuchlich ist, wird sie hier beibehalten.

II 6 Werkzeuge

über das Internet zugreifen können. Dasselbe gilt für Fehler- und Änderungsmanagementwerkzeuge, wie Bugzilla, FlySpray oder JIRA. Bei Build- und Release-Managementwerkzeugen wie Ant, Maven oder CruiseControl können sich die Entwickler anhand eines Armaturenbretts eine Übersicht über den Verlauf des Build- und Testprozesses verschaffen. Auch Modellierungswerkzeuge wie ArgoUML oder der IBM Rational Software Modeler haben eine Unterstützung für die Arbeit in Teams. Hier können Modelle ausgetauscht werden, die helfen sollen, ein gemeinsames Verständnis aufzubauen. Die Werkzeuge setzen sogar mehrere Modellierer in die Lage, zeitgleich Modelle kollaborativ zu entwickeln.

Soziale Software Zur Kommunikation zwischen Entwicklern eignet sich jede Art von sogenannter sozialer Software. Soziale Software ist Software, die Informations-, Identitäts- und Beziehungsmanagement in sozialen Netzwerken unterstützt [Schm06]:

- Informationsmanagement: Ermöglichung des Findens, Bewertens und Verwaltens von (online verfügbarer) Information.
- Identitätsmanagement: Ermöglichung der Darstellung von Aspekten seiner selbst im Internet.
- Beziehungsmanagement: Ermöglichung Kontakte abzubilden, zu pflegen und neu zu knüpfen.

In der Softwareentwicklung dient soziale Software dem Wissensmanagement und der Expertenverortung im sozialen Netzwerk der Entwickler.

Unidirektionaler Austausch Für das Verbreiten von Wissen werden häufig so genannte Wissenszentren gebildet, über die Informationen zu verschiedenen relevanten Sachverhalten – zum Beispiel über das Web – allen Entwicklern zur Verfügung gestellt werden. Diese Form der Wissensverbreitung findet man sowohl im kommerziellen Bereich, wie etwa beim developersWork von IBM, als auch im Open-Source-Bereich, wie bei Eclipse oder beim Linux Documentation Project. Wissenszentren sind in ihrer Kommunikationsform unidirektional, das heißt, die Information fließt nur in eine Richtung: Entwickler nehmen das Wissen auf, das im Wissenszentrum angeboten wird.

Bidirektionaler Austausch Zur bidirektionalen Kommunikation eignen sich Werkzeuge der folgenden Art:

- Kommunikationssoftware wie E-Mail, Chat, Instant-Messaging oder virtuelle Welten, die synchrone oder asynchrone flüchtige und nicht-öffentliche Kommunikation zwischen Entwicklern erlauben; flüchtig heißt, dass die Information verloren geht, wenn die Empfänger sie nicht selbst aktiv speichern; nicht-öffentlich heißt, dass die Information nur an einen explizit aufgeführten Personenkreis gelangt.

- Web-Foren, Wikis, Web-Logs und Mashups, mit Hilfe derer Entwickler anderen ihre Information asynchron und (teils-)öffentlich zur Verfügung stellen können, ohne die Adressaten zu kennen.
- Soziale Netzwerke wie Facebook, MySpace oder Xing, die es erlauben, andere Entwickler mit einer bestimmten Expertise ausfindig zu machen.

6.2.4 Unterstützung von Prozessmodellen und Methoden

Primär werden Werkzeuge nach den von ihnen bearbeiteten Artefakten sowie nach den Operationen, die sie darauf ausführen, unterschieden. Jedoch können sie darüber hinaus noch weiter im Grad ihrer Unterstützung bestimmter Methoden und Prozessmodelle sowie in ihrer Fähigkeit zur Behandlung mehrerer Projekte charakterisiert werden.

Das **Prozessmodell** beschreibt den geplanten Gesamtprozess zur Entwicklung der Software. Es legt die Abfolge der Aktivitäten fest, die zur Entwicklung durchgeführt werden sollen. Eine Aktivität hat die Erstellung eines Zwischenprodukts zum Ziel. Beispiele für Zwischenprodukte sind Dokumente wie die Anforderungsspezifikation oder der Entwurf. Zwischenprodukte haben aber nicht notwendigerweise immer eine materielle Gestalt in Form eines Dokuments, sondern können auch reines Wissen in den Köpfen der Softwareentwickler sein. Bei agilen Prozessmodellen werden beispielsweise Anforderungen oft nur oral überliefert. Im Rahmen einer Aktivität werden Methoden und Werkzeuge zur Erstellung des Zwischenprodukts eingesetzt. Daraus folgt, dass die Werkzeuge ein bestimmtes Vorgehen unterstützen müssen. Basiert ein Prozessmodell beispielsweise auf der Entwicklung von Prototypen, so müssen Werkzeuge vorhanden sein, die die rasche Entwicklung von Prototypen unterstützen.

Unterstützung für ein Prozessmodell

Als ein Beispiel für **methodenunterstützende** Werkzeuge können UML-Werkzeuge angesehen werden, da sie entsprechende Modellierungskonstrukte wie Klassen und Assoziationen anbieten und die Regeln für gültige UML-Klassendiagramme prüfen. Man könnte Klassendiagramme aber auch mit einem herkömmlichen Zeichenprogramm erstellen. Mit Hilfe des Zeichenprogrammes könnte man genauso gut aber auch die Datenmodellierung mit Hilfe von Entity-Relationship-Diagrammen unterstützen. Ein Zeichenprogramm ist in diesem Sinne **methodenneutral**. Das Zeichenprogramm würde es aber auch nicht verhindern, dass man UML-Klassen- und Entity-Relationship-Diagramme mischt oder sonstige ungültige Diagramme erzeugt.

Methodenunterstützend vs. methodenneutral

Werkzeuge sollten in der Lage sein, mehrere Softwareprojekte einer Organisation zu verwalten. Diese Projekte können sowohl zeitgleich als auch zeitlich versetzt ablaufen. Meist sind Entwickler da-

Multiprojektfähig

II 6 Werkzeuge

bei mehreren Projekten zugeordnet, auf die sie die Werkzeuge anwenden wollen. Werkzeuge müssen dazu **multiprojektfähig** sein, das heißt dem Entwickler die jeweilige Sicht auf die ihnen zugeordneten Projekte gewähren. Zeitgleiche Verwendung erfordert von den Werkzeugen nicht nur, dass sie für die Entwickler parallel verfügbar sein müssen, sondern auch dass sie die Konsistenz ihrer Daten bei zeitgleichem Zugriff gewährleisten müssen, wenn diese Daten zentral verwaltet werden.

Interprojekt-
fähig

Wenn mehrere Projekte verwaltet werden, ist es für viele Werkzeuge auch wünschenswert, dass sie vergleichende Sichten auf die Daten zu den verschiedenen Projekten anbieten. Die Werkzeuge sollten also in diesem Sinne **interprojektfähig** sein. Von einem Softwarearmaturenprojekt, wie im Kapitel »Von Werkzeugen unterstützte Operationen«, S. 62, erwähnt, würde ein Qualitätsmanager für viele Projekte erwarten, dass er alle seine Projekte vergleichend einsehen kann.

6.3 Integrierte Entwicklungsumgebungen

Integrierte Entwicklungsumgebungen sind Plattformen, die eine Reihe von Entwicklungswerkzeugen in einer einheitlichen Benutzungsoberfläche integrieren. Im Englischen wird hierfür der Begriff *Integrated Development Environment* verwendet, dessen Abkürzung **IDE** auch im Deutschen gebräuchlich ist.

Uniforme
Benutzung

Eine IDE integriert mindestens die essenziellen Werkzeuge Editor, Übersetzer, Binder sowie die Build-Werkzeuge und meist auch den Debugger. Der Entwickler kann diese Werkzeuge in uniformer Weise in der Benutzungsoberfläche aufrufen sowie Einstellungen der Werkzeuge in gleicher Weise vornehmen. Ihre Ausgaben werden ebenso uniform von der Benutzungsoberfläche dargestellt. Die **Integration** und **Uniformität** erleichtert es dem Entwickler, mit den Werkzeugen umzugehen. Den Kontroll- und Datenfluss zwischen den Werkzeugen regelt die IDE.

Integration

Für die Integration der Werkzeuge bietet die IDE eine **Infrastruktur** für den Kontroll- und Datenfluss zwischen den Werkzeugen. So kann zum Beispiel die Editorkomponente andere Werkzeuge in Form von Signalen benachrichtigen, dass der Entwickler eine Änderung an einer Datei vorgenommen hat. Der Übersetzer kann dann sofort die Übersetzung der geänderten Datei übernehmen. Die notwendigen Daten für die Übersetzung werden zwischen den Werkzeugen in Form von Dateien oder in bereits vorverarbeiteter Form ausgetauscht.

Beispiel: Eclipse

Offene IDEs erlauben es, weitere Werkzeuge zu integrieren. Eines der bekanntesten Beispiele einer offenen IDE ist **Eclipse**. Nicht nur die **Offenheit** des Quellcodes von Eclipse erlaubt eigene Anpassungen, die Offenheit ist bereits in der Architektur von Eclipse vorgesehen. Eclipse ist eine Plattform, die eine **generische Infrastruktur** für die Integration von Entwicklungswerkzeugen bietet. Die Infrastruktur von Eclipse, die aus den im Folgenden beschriebenen Komponenten besteht, ist in der Abb. 6.3-1 dargestellt.

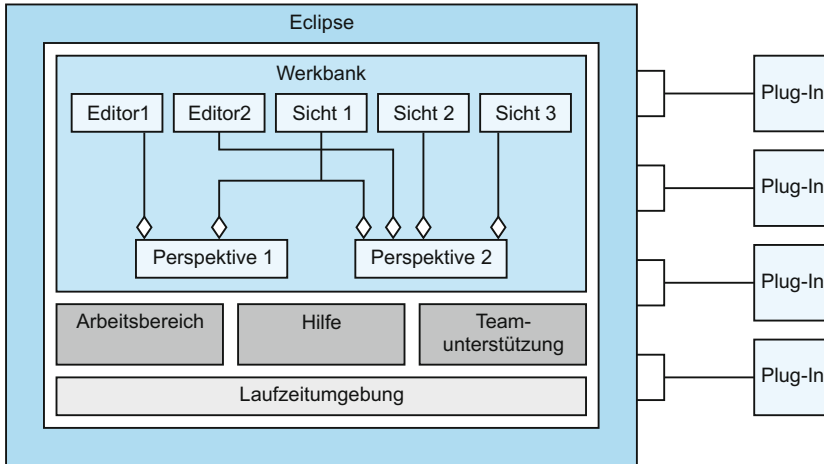


Abb. 6.3-1:
Komponenten von
Eclipse.

Der **Arbeitsbereich** umfasst die Projekte, an denen der Entwickler arbeitet. Ein Projekt definiert ein Unterverzeichnis und die darin enthaltenen Dateien, die der Entwickler editiert. Im Falle von Änderungen an Dateien des Arbeitsbereiches stellt Eclipse allen Komponenten, die sich für solche Änderungen registriert haben, die Liste von Änderungen zur Verfügung. Übersetzer und Build-Werkzeuge können auf Basis dieser Änderungen bestimmen, was neu übersetzt werden muss. Inkrementelle Übersetzer können damit den Übersetzungsaufwand auf das Notwendige reduzieren.

Arbeitsbereich

Die grafische Benutzungsschnittstelle basiert auf einer sogenannten **Werkbank** (*Workbench*), die die allgemeine Struktur für die Interaktion mit dem Benutzer vorgibt. Ihre Implementierung stützt sich auf Programmbibliotheken, die unabhängig vom Betriebssystem sind.

Werkbank

Die Werkbank besteht aus mindestens einem Editor für das Erstellen und Ändern der Dateien sowie verschiedenen **Sichten**, die Informationen über die Objekte im Arbeitsbereich darstellen. Der Fensterinhalt einer Werkbank ist festgelegt durch eine Perspektive. Die **Perspektive** bestimmt, welche Editoren und Sichten dargestellt werden. Verschiedene Perspektiven erlauben es, die notwendigen Editoren

Editor, Sicht,
Perspektive

II 6 Werkzeuge

ren und Sichten auf Knopfdruck aufgabengerecht bereitzustellen. So kann man zum Beispiel eine Sicht für das Editieren, eine zweite für das Testen und eine dritte für die Fehlersuche anbieten.

Team-Unterstützung Die Team-Unterstützung ist eine Komponente, die eine Anbindung an ein Versionskontrollsystem für die Versionierung und Zusammenarbeit in einem Team unterstützt.

Hilfe Eclipse verfügt außerdem über eine Hilfefunktion, die es Werkzeugen erlaubt, Dokumentation für den Benutzer bereitzustellen.

Plug-in Durch Installation von **Plug-ins** kann der Benutzer Eclipse erweitern. Plug-ins sind die kleinsten Softwareeinheiten, die separat entwickelt und ausgeliefert werden, und in die Plattform integriert werden können. Einfache Entwicklungswerkzeuge werden als ein Plug-in implementiert, komplexere Werkzeuge durch mehrere Plug-ins. Die Plug-ins können auf die Infrastruktur von Eclipse sowie auf andere Plug-ins zurückgreifen.

Ausprägungen Eclipse ist eine Infrastruktur für IDEs. Ausprägungen der Infrastruktur gibt es für verschiedene Programmiersprachen. Die beiden bekanntesten sind das *Java Development Tooling* (JDT) für Java und das *C/C++ Development Tooling* (CDT) für C und C++. Eclipse-basierte IDEs gibt es auch für COBOL, PHP und andere Sprachen.

Programmdarstellung Die Ausprägungen für IDEs liefern einen Übersetzer mit, der das Programm in eine Zwischendarstellung überführt. Der Übersetzer von JDT zum Beispiel repräsentiert das Programm als einen Syntaxbaum. Die Plug-ins können auf diese Programmdarstellung zugreifen. Metrikwerkzeuge traversieren zum Beispiel den Syntaxbaum, um das Programm zu vermessen. Refactoring-Werkzeuge traversieren den Syntaxbaum nicht nur, sondern modifizieren ihn sogar. Die Repräsentation des Programms in der einheitlichen Form eines Syntaxbaums erleichtert die Entwicklung von Plug-ins erheblich. Das ist einer der Gründe, warum so viele Open-Source-Entwickler Plug-ins für JDT schreiben.

Eclipse ist aber keineswegs begrenzt auf die Entwicklung von Programmen in einer bestimmten Programmiersprache. Ausprägungen gibt es auch für die Web-Entwicklung oder für spezielle Techniken wie SOA (*Service Oriented Architectures*), die modellgetriebene Entwicklung und vieles mehr.

Trends von IDEs

In den letzten Jahren haben verstärkt die statischen Programmanalysen an Aufwind gewonnen. Reife Werkzeuge wurden sowohl im Open-Source-Bereich als auch von kommerziellen Herstellern entwickelt. Neben dem Fortschritt in der statischen Programmanalyse selbst ist auch insbesondere der Fortschritt in der Hardware dafür verantwortlich. Heutige Computersysteme bieten die notwendige

Rechenleistung für die komplexen Algorithmen der Programmanalyse. Aus diesem Grund ist zu erwarten, dass statische Analysewerkzeuge selbstverständliche Komponenten in IDEs sein werden.

Hatten IDEs früher primär das Ziel, den Programmierer bei der Arbeit zu unterstützen, und boten dazu Editor, Übersetzer, Debugger und dergleichen, so unterstützen IDEs heute alle Aktivitäten der Entwicklung von der Planung und Spezifikation bis hin zur Wartung. Auch die Kommunikation zwischen den am Entwicklungsprozess beteiligten Menschen wird zunehmend von IDEs übernommen. Dieser Trend ist bereits in neuen Werkzeugen zu erkennen. Die Entwicklungsplattform Jazz von IBM beispielsweise stellt die Kollaboration in die Mitte der Softwareentwicklung. Dazu gehört die Unterstützung durch entsprechende Kommunikationsmechanismen, die über reine Entwicklungsaufgaben hinaus führen. Eine höhere Transparenz für alle Beteiligten wird durch holistische Sichten auf das Projekt geschaffen. Auch der Arbeitsfluss der Aktivitäten in einem Team kann selbst festgelegt werden, sodass kein starres Vorgehen mehr vorgegeben wird. Kollaboration, Offenheit und Flexibilität sind die Themen, denen sich zukünftige IDEs verstärkt widmen werden.

Ein Risiko beim Trend, alle möglichen Aktivitäten in einer IDE zu integrieren, ist jedoch die zunehmende Entropie, die leicht dazu führen kann, dass das Zusammenspiel der Werkzeuge fehleranfälliger wird. Auch der Aufwand für die Wartung all der unterschiedlichen Werkzeuge mit ihren diversen Abhängigkeiten nimmt mit der Anzahl der integrierten Werkzeuge drastisch zu. Dies gilt insbesondere, wenn jeder Entwickler die Freiheit hat, seine eigenen Plug-ins zu installieren. Während also einerseits die **Generalisierung** von IDEs weiter voranschreiten wird, so ist auch andererseits zu erwarten, dass IDEs dann für einen bestimmten Zweck oder eine bestimmte Umgebung spezialisiert werden, um die zunehmende Komplexität zu begrenzen.

6.4 Modellgetriebene Entwicklung

In der klassischen Entwicklung werden **Modelle** benutzt, um Anforderungen (siehe auch »Anforderungen modellieren«, S. 547) oder Entwürfe abstrakt zu beschreiben. Anschließend setzen Entwickler das Modell in Code um. Bei der **modellgetriebenen Entwicklung** wird Code direkt aus einem Modell generiert.

Die modellgetriebene Entwicklung wird im Englischen meist mit dem Begriff **MDSD** (*Model-Driven Software Development*) bezeichnet. Da sich auch in der deutschsprachigen Fachliteratur die englischsprachige Abkürzung durchgesetzt hat, wird im Folgenden auch die englische Abkürzung bevorzugt. Gebräuchlich sind auch die Be-

II 6 Werkzeuge

griffe MDD (*Model Driven Development*) und MDA (*Model Driven Architecture*), wobei MDA oft eine spezifische Bedeutung hat (siehe unten).

Modell Modelle spielen in der Softwaretechnik unterschiedliche Rollen. Sie werden benutzt, um Aspekte der Anforderungen, des Entwurfes oder der Implementierung abstrakt zu beschreiben. Anhand des Modells werden Probleme studiert, nach Lösungen gesucht oder Vorhersagen gemacht für das, was wirklich entwickelt wird. Das Modell ist hier ein Stellvertreter, an dem bestimmte Sachverhalte untersucht werden. Es ist ein Abbild des angestrebten technischen Erzeugnisses – also der Software, die entwickelt werden soll (siehe auch »Prinzip der Abstraktion«, S. 26).

In der modellgetriebenen Entwicklung hat das Modell nicht nur einen dokumentarischen Zweck. Dort ist das Modell eine formale Beschreibung, aus der heraus lauffähige Programme automatisiert generiert werden [SVE+07]. Zur modellgetriebenen Entwicklung gehören

- ein Modell,
- die Notation, in der das Modell formal beschrieben ist, und
- Code-Generatoren, die das Modell in ausführbaren Code transformieren.

Vor- und Nachteil eines Modells zugleich ist, dass es eine abstrakte Beschreibung des angestrebten Erzeugnisses ist. Irrelevante Details werden weggelassen. Dies erhöht einerseits die Verständlichkeit für einen Zweck, vermindert aber andererseits die Genauigkeit.

Notation Häufig werden spezielle Notationen für die Modellierung benutzt. Die UML ist eine weit verbreitete Notation für diese Zwecke. Modelle können aber auch textuell beschrieben werden.

Code-Generierung In der traditionellen Entwicklung setzen Entwickler dann das Modell manuell in Code um. Die Alternative zur manuellen Umsetzung ist es, aus dem Modell direkt den Code zu generieren. Diese Idee ist nicht neu, vielmehr ist die automatische Code-Generierung in etablierten Domänen, wie zum Beispiel dem Übersetzerbau, schon lange verbreitet.

Vorteile Die Vorteile der automatischen Code-Generierung für Modelle lauten wie folgt:

- + Es kann abstrakter entwickelt werden, als es auf Code-Ebene möglich ist.
- + Modell und Code sind immer synchron.
- + Die Entwicklung von Code ist automatisiert und damit beliebig wiederholbar.
- + Code-Generierung ist weniger fehlerträchtig als manuelle Umsetzung beziehungsweise können Fehler leichter durch die Korrektur der Code-Generierung wieder beseitigt werden.

6.4 Modellgetriebene Entwicklung II

Wird der generierte Code vom Entwickler geändert, dann entfallen viele der genannten Vorteile. Aus diesem Grunde sollte davon Abstand genommen werden.

Die Nachteile der automatischen Code-Generierung sind:

Nachteile

- Nur praktikabel, wenn der Code für das Modell sehr regulär ist, das heißt, die Code-Generierung muss übertragbar sein.
- Der Aufwand lohnt sich erst, wenn ähnlicher Code mehrfach entwickelt werden muss.
- Die Fehlersuche wird erschwert, weil es mehr Fehlerquellen gibt und heutige Debugger in der Regel nur mit dem generierten Quellcode arbeiten.
- Handgeschriebener Code kann effizienter sein, wenn die Entwickler Spezialfälle bei der manuellen Programmierung ausnützen können.
- Die Lösung ist möglicherweise komplexer durch die unterschiedlichen Techniken, die der Entwickler bei der modellgetriebenen Entwicklung beherrschen muss.
- Modell und Code sind identisch. Fehler im Modell führen unmittelbar zu fehlerhaftem Code. Bei der manuellen Programmierung können Mängel und Fehler möglicherweise noch vom Entwickler während der Umsetzung gefunden werden

Weil es in den seltensten Fällen praktikabel ist, allen Code zu generieren, wird die automatische Code-Generierung die manuelle Programmierung nicht völlig ersetzen, sondern vielmehr ergänzen. Wenn nicht aller Code automatisch generiert wird, ist auf eine Integration mit handgeschriebenem Code zu achten.

Hand-
geschriebener
Code

Die meisten UML-Werkzeuge bieten die Möglichkeit, Codegerüste für ihre Modelle zu generieren. Hier ist man jedoch eingeschränkt durch die Arten von Modellen, die ein UML-Werkzeug unterstützt, und die Code-Generierung, die das UML-Werkzeug vorsieht.

Werkzeuge

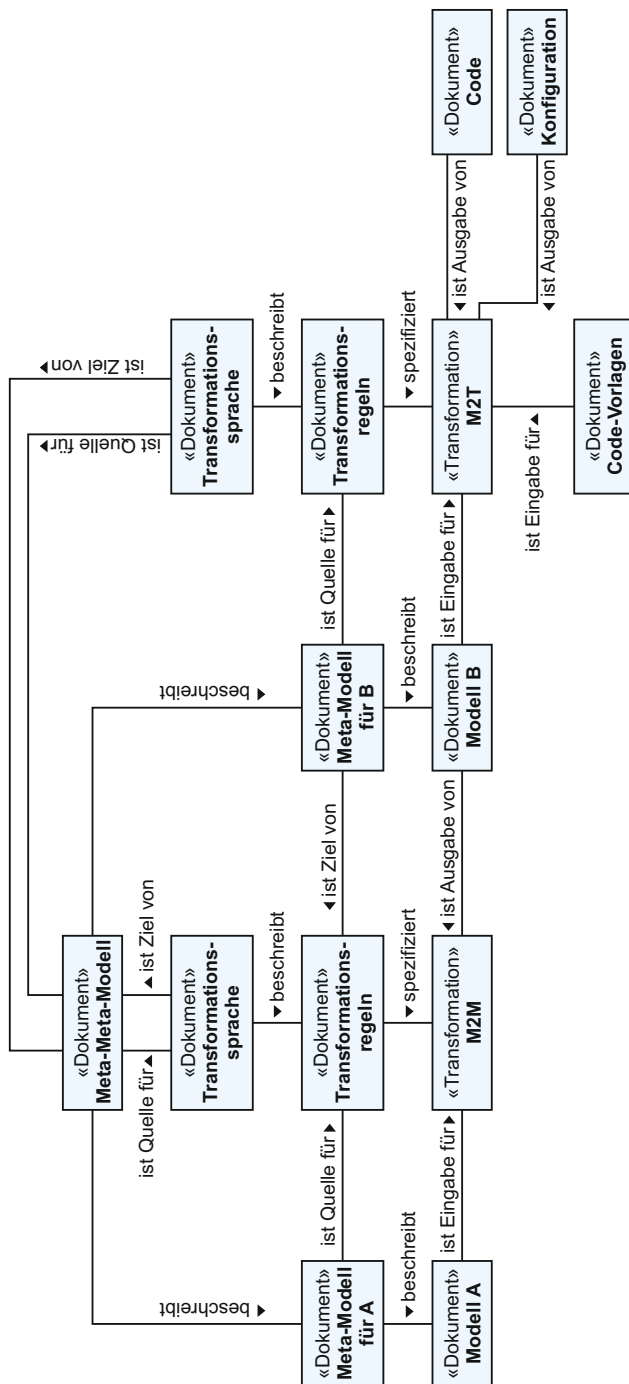
Bei der modellgetriebenen Entwicklung kann man noch einen Schritt weiter gehen. Mit geeigneten Werkzeugen, wie zum Beispiel dem EMF (*Eclipse Modeling Framework*), kann man selbst festlegen, welche Modelle spezifiziert werden können und welcher Code dafür generiert werden soll. Die wesentlichen Komponenten hierfür sind in der Abb. 6.4-1 zusammengefasst.

Zunächst benötigt man eine Sprache, in der Modelle formuliert werden können. Diese Sprache ist im Gegensatz zu einer allgemeinen Programmiersprache stark zugeschnitten auf die Anwendungsdomäne. Deshalb werden diese Sprachen auch domänenspezifische Sprachen genannt.

Domänen-
spezifische
Sprache

II 6 Werkzeuge

*Abb. 6.4-1:
Komponenten der
modellbasierten
Entwicklung.*



6.4 Modellgetriebene Entwicklung II

Ein Modell kann als eine Menge von Sätzen dieser Sprache betrachtet werden. Die Sprache selbst kann vom Entwickler definiert werden. Dazu muss die Sprache selbst durch ein sogenanntes **Meta-Modell** beschrieben werden. Das Meta-Modell ist ein Modell, das die Spezifikation von anderen Modellen darstellt (siehe auch »Prinzip der Abstraktion«, S. 26).

Meta-Modell

Zur Spezifikation des Meta-Modells benötigt man gleichfalls eine Sprache. Diese Sprache wird durch das **Meta-Meta-Modell** beschrieben. Dieses Meta-Meta-Modell ist in das Werkzeug bereits eingebaut und durch sich selbst beschreibbar.

Meta-Meta-Modell

Ein Modell kann mit Hilfe von Transformationsregeln transformiert werden. Sie definieren eine Abbildung von einem Modell auf ein anderes Modell – die Modell-zu-Modell-Transformation (M2M) – oder von einem Modell auf Text (M2T). Letztere wird benutzt, um Code (auch Testfälle) oder andere textuelle Systembeschreibungen – wie zum Beispiel Konfigurationsdateien oder Dokumentation – zu generieren. Bei der Modell-zu-Modell-Transformation kann sowohl zwischen Modellen desselben Meta-Modells als auch zwischen Modellen, die durch verschiedene Meta-Modelle beschrieben sind, transformiert werden.

Transformation

Für die Formulierung der Transformationsregeln zwischen Modellen steht eine Sprache zur Verfügung, die sich auf das Quell- und Zielmetamodell stützt und über die Meta-Meta-Modelle definiert ist.

Modell-zu-Modell
(M2M)

Die Modell-zu-Text-Transformation verwendet Textvorlagen mit Platzhaltern, die durch den Code-Generator ausgefüllt werden. Auch hierfür existiert eine entsprechende Sprache, die man sich als eine Art Skriptsprache vorstellen kann, die ein Modell traversiert und für passende Muster eine Code-Vorlage ausfüllt.

Modell-zu-Text
(M2T)

Zustandsautomaten (siehe auch »Zustandsautomaten«, S. 269) werden häufig für die Modellierung reaktiver Systeme verwendet. Reaktive Systeme verfügen über Sensoren, über die sie äußere Stimuli wahrnehmen können. Außerdem verfügen sie über Aktoren, mit deren Hilfe sie auf den äußeren Stimulus reagieren und damit in ihrer Umwelt etwas bewirken können. Die Reaktion ist meist nicht nur vom Stimulus abhängig, sondern auch von dem inneren Zustand, in dem sich das reaktive System befindet.

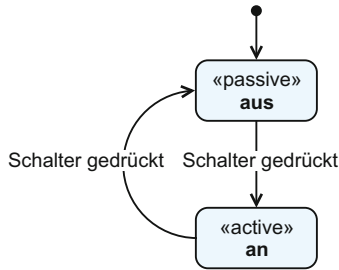
Beispiel

Eine Lampe beispielsweise reagiert auf das Drücken eines Schalters, indem ein Draht zu glühen beginnt. Die Abb. 6.4-2 zeigt eine Zustandsmaschine für eine Lampe mit einem einzelnen Schalter. Vom Initialzustand geht die Ampel direkt in den Zustand aus über. Wird der Schalter in diesem Zustand gedrückt, nimmt die Lampe den Zustand an an. Wird der Schalter noch einmal gedrückt, kehrt die Lampe wieder in den Zustand aus zurück.

II 6 Werkzeuge

Dieses Lampenmodell ist in der Notation für Zustandsdiagramme der UML verfasst. Die beiden Zustände sind mit **Stereotypen** passive beziehungsweise active versehen. Mit Stereotypen lassen sich Modellelemente in der UML mit zusätzlichen benutzerdefinierten Eigenschaften versehen. Indem ein Modellierer eigene Stereotypen einführt, erweitert er die Sprache UML zu einer domänenspezifischen Sprache.

Abb. 6.4-2:
Zustandsautomat
einer Lampe.



Mit active seien hier Zustände versehen, in denen der Draht Strom erhält, damit er zu glühen beginnt. Mit passive seien Zustände markiert, bei denen der Strom zum Draht unterbrochen wird.

In diesem Modell glüht der Draht endlos. Um Strom zu sparen und den Draht vor Überhitzung zu schützen,

könnte man einen Übergang im Modell einführen, der nach einer bestimmten Zeit vom Zustand an automatisch in den Zustand aus überführt. Gäbe es mehrere aktive Zustände, müsste von jedem Zustand aus ein solcher Übergang eingeführt werden. Soll dies generell für alle aktiven Zustände gelten, könnte man dies einfach durch eine Modell-zu-Modell-Transformation beschreiben. Hierzu würde man eine Transformation definieren, die für jeden aktiven Zustand einen zeitlich gesteuerten Übergang zum Zustand aus einführt. Mit Hilfe dieser Transformation würde das Modell übersichtlich bleiben und es hätte dennoch die gewollte Bedeutung, ohne dass die Code-Generierung angepasst werden müsste.

Eine Modell-zu-Text-Transformation kann schließlich für das Zustandsdiagramm automatisch Java-Code generieren. Der generierte Java-Code könnte für das konkrete Beispiel folgendes Aussehen haben:

```
state = aus;
while (true) {
    switch (state) {
        case aus: if (stimulus == Schalter_gedruickt) state = an;
                  break;
        case an:  if (stimulus == Schalter_gedruickt) state = aus;
                  break;
    }
}
```

Die Code-Vorlage, mit Hilfe derer der Code-Generator diesen Code generiert, könnte folgende Struktur haben:

```
state = <<first>>;
while (true) {
    switch (state) {
        <<FOREACH states AS s >>
        case <<s>>:
```

```

    <<FOREACH outgoing(s) AS t >>
        if (stimulus == <<label(t)>>) state = <<target(t)>>;
    <<ENDFOREACH>>
    break;
<<ENDFOREACH>>
}
}

```

Eingebettet in die Vorlage sind Anweisungen, wie der Code generiert werden soll. Eine äußere Schleife iteriert hier über alle Zustände und generiert die entsprechende case-Anweisung für diesen Zustand. Die innere Schleife iteriert über alle ausgehenden Übergänge des aktuellen Zustands und vergleicht den Stimulus auf das *Label* des Übergangs. Im Falle einer Übereinstimmung wird der Zustand dann auf den Zielzustand des Übergangs gesetzt. Die Variable <<first>> sei der Initialzustand des Automaten.

Für die Kommunikation zwischen Softwaresystemen gibt es eine Reihe von Industriestandards wie CORBA, EJB/J2EE, .NET oder XML/SOAP. Systeme sind heute oft gezwungen, in vielen dieser unterschiedlichen Plattformen zu operieren. Zudem ändern sich Standards auch über die Zeit. Um gegen solche Änderungen geschützt zu sein, hat die OMG (*Object Management Group*) das **MDA**-Konzept (*Model Driven Architecture*) vorgeschlagen, das die Ideen der modellgetriebenen Entwicklung umsetzt. Hier findet eine Trennung zwischen eigentlicher Applikationslogik und der Plattform für die Kommunikation statt. Hierzu wird ein Modell für die Applikationslogik erstellt, das unabhängig von der Plattform ist. Dieses Modell wird **PIM** (*Platform Independent Model*) genannt. Die Realisierung des PIM für eine konkrete Plattform wird in einem plattformspezifischen Modell **PSM** (*Platform-Specific Model*) angegeben, das mittels Modell-zu-Modell-Transformationen hergeleitet wird. Für das PSM wird schließlich durch Modell-zu-Text-Transformation Code in der zugrunde liegenden Programmiersprache generiert.

Beispiel

Mohagheghi und Dehlen haben in einer Literaturstudie von Veröffentlichungen zu empirischen Untersuchungen der modellgetriebenen Entwicklung im Zeitraum 2000–2007 lediglich 25 Veröffentlichungen gefunden, von denen gerade einmal sieben von abgeschlossenen Projekten berichteten [MoDe08]. Die Domänen, aus denen diese Studien stammen, sind überwiegend die Telekommunikation und betriebswirtschaftliche Anwendungen und zu einem geringeren Maße die Domänen Militär, Aerodynamik und Flugsysteme sowie Web-Anwendungen.

Studien zu MDSD

In vielen der betrachteten Artikel wurde die Reife der notwendigen Werkzeuge noch bemängelt. Dennoch fanden sie Belege für die Verbesserung der Softwarequalität. Interessanterweise fanden sie darüber hinaus sowohl Belege für die Reduktion genauso wie für

II 6 Werkzeuge

die Erhöhung des Entwicklungsaufwands nach der Einführung der modellgetriebenen Entwicklung. Letztere ist zu einem großen Teil auf mangelhafte Werkzeugunterstützung und den erhöhten Aufwand für das Lernen und die Verwendung dieser komplexeren Technik zurück zu führen. Sie bemängeln jedoch, dass diese Aussagen zumeist von relativ kleinen Fallstudien stammen und es nur wenige Untersuchungen in realistischen, großen Projekten existieren. Gegenwärtig gibt es noch nicht genug aussagekräftige empirische Studien, um fundierte Aussagen zum Effekt von MDSD zu machen.

Eine der wenigen Studien zu großen industriellen Projekten, in denen MDSD eingesetzt wurde, ist die von Baker et al. [BLW05]. Sie wird auch in der Meta-Studie von Mohagheghi und Dehlen aufgeführt. Baker et al. berichten über die fünfzehnjährigen Erfahrungen, die Motorola mit MDSD gesammelt hat:

- + In manchen Geschäftsbereichen werden 65 % – 85 % des Codes automatisch generiert.
- + MDSD hatte eine nachhaltige Erhöhung der Produktivität im Bereich von Faktor 2 bis 8.
- + Der Testaufwand konnte um ein Drittel reduziert werden, dank Testfällen, die aus dem Modell generiert werden können.
- + Die Fehlerbehebungszeit senkte sich um Faktoren von 30 bis 70.
- + Die Anzahl der Fehler konnte um Faktoren von 1,2 bis 4 verringert werden.

Diesen Vorteilen stehen auf der anderen Seite auch einige Herausforderungen gegenüber:

- Die Integration des automatisch generierten Codes mit handgeschriebenem stellte eine solche Schwierigkeit dar.
- Werkzeuge fehlten, waren schlecht integriert oder generierten wenig effizienten Code.
- Entwickler waren unerfahren in der Technik und Prozesse mussten umgestellt werden.

Evolution in MDSD Ein bis dato auch kaum beachtetes Problem ist der Aspekt der Evolution in der MDSD. Software muss stets weiterentwickelt werden, ob sie nun in einer universellen Programmiersprache oder in einer domänenspezifischen Sprache verfasst ist. Im Falle von MDSD kann sich die Evolution auf verschiedenen Ebenen vollziehen: im Modell selbst, im Metamodell, in den Generatoren und in den Werkzeugen, die MDSD unterstützen. Diese verschiedenen Dimensionen führen zu einer weiteren Komplexität in MDSD.

Trends Die Ideen der modellgetriebenen Entwicklung sind nicht neu. Voraussetzung für den erfolgreichen Einsatz generativer Techniken und domänenspezifischer Sprachen ist es, dass die Anwendungsdomäne wohl verstanden ist und eine sehr reguläre Lösungsstruktur auf-

6.5 Auswahlkriterien bei der Anschaffung von Werkzeugen II

weist. Die MDSD wird also in solchen Bereichen Erfolg haben und sich weiter ausbreiten. Sie wird aber kaum universelle Programmiersprachen verdrängen können.

Nach der Etablierung von Hochsprachen, die Assemblersprachen weitgehend verdrängt haben, waren Entwickler in der Lage noch größere Systeme zu bauen. Mit der Anhebung der Abstraktion durch domänenspezifische Sprachen und der Unterstützung durch automatische Code-Generierung wird dieser Trend zukünftig vermutlich fortgesetzt werden. So wenig wie es aber universellen Hochsprachen gelungen ist, Programmier- und insbesondere Spezifikationsfehler zu verbannen, so wenig wird dies auch MDSD gelingen. Softwareentwicklung wird auch mit MDSD weiterhin ein äußerst schwieriges Unterfangen bleiben.

6.5 Auswahlkriterien bei der Anschaffung von Werkzeugen

Bei der Auswahl von Werkzeugen sind neben allgemeinen Kriterien, die an jede Software angelegt werden – wie zum Beispiel Kosten und Nutzen, Gebrauchstauglichkeit, Ressourcenverbrauch, Robustheit und Verfügbarkeit – spezifische Kriterien zu beachten.

Softwareentwicklungswerkzeuge sind Programme. Damit sind ihre Auswahlkriterien und -verfahren eine Spezialisierung der Kriterien und Auswahlverfahren von Programmen im Allgemeinen. Ein Programm wird unter einer Vielzahl von Programmen ausgewählt, wenn es die Bedürfnisse des Benutzers an die Funktionalität und Qualität ausreichend erfüllt und das Verhältnis von Kosten und Nutzen günstiger ist als bei den anderen Programmen der Auswahl.

Kosten und Nutzen

Die Kosten können oder sollen nicht notwendigerweise immer ausschließlich in Geld gemessen werden. Der Begriff Kosten bezeichne deshalb hier allgemeiner jeden Einsatz von Ressourcen, um das Werkzeug einzuführen, zu betreiben und auch wieder abzuschaffen beziehungsweise abzulösen.

Kosten

Die Einführungskosten umfassen die Auswahl und Anschaffung des Werkzeugs, die Schulung der Entwickler in der Benutzung des Werkzeugs, die Installation und Bereitstellung des Werkzeugs, die Anschaffung zusätzlicher Hardware für den Betrieb und gegebenenfalls die Migration oder Aufbereitung von Daten, die das Werkzeug übernehmen soll. Bei großen Organisationen dominieren hier meist die Schulungskosten.

Einführungskosten

II 6 Werkzeuge

Betriebskosten	Zu den Betriebskosten zählen die Aufwände in der Verwendung des Werkzeugs und seiner Administration sowie etwaige laufende Lizenzgebühren für die Miete oder Wartung. Meist dominieren hier die Kosten für die Aufwände der Entwickler über die Kosten für die Hardware- & Softwareplattform, die die Entwicklungsprozesse voraussetzen.
Fehlerkosten	Im ungünstigen Fall gehören zu den Kosten auch die Schäden, die durch die Mängel des Werkzeugs oder seine falsche Benutzung entstehen. Folgekosten durch Schäden müssen bedacht werden, sind jedoch schwer kalkulierbar.
Ablösekosten	Auch die Kosten der Ablösung des Werkzeugs sind schwer zu kalkulieren. Ist ein Werkzeug in einer Organisation erst einmal etabliert und stark in den Entwicklungsprozess integriert, fällt es schwer, es wieder zu entfernen, weil dann Prozesse wieder umstrukturiert und Entwickler umgeschult werden müssen sowie eine Strategie zur Erhaltung der vielen, oft für das Unternehmen essenziellen Daten, die das Werkzeug angesammelt hat, entwickelt werden muss. Man spricht dann von der Werkzeugfalle, aus der man sich nicht mehr leicht befreien kann. Wann die Ablösung eintritt, ist schwer vorherzusagen und hängt nicht selten von äußeren Umständen ab, die nichts mit dem Werkzeug selbst zu tun haben.
Kosten für Open-Source-Werkzeuge	Open-Source-Systeme werden nicht wegen des Zugriffs auf den Quellcode gerne eingesetzt, sondern weil sie häufig als kostengünstige Alternative zu kommerziellen Systemen gelten. Hier kann man sich zwar die Lizenzkosten für die Anschaffung und Wartung sparen, aber dennoch können die anderen Kosten dies wieder aufwiegen. Die Haftung für Fehlerfolgekosten von Open-Source-Werkzeugen ist schwer durchzusetzen. Zur Fehlerbehebung oder Erweiterung der Funktionalität von Open-Source-Werkzeugen muss man möglicherweise doch externe oder interne Projekte finanzieren.
Nutzen	Der Nutzen wird in der Erhöhung der Effektivität und Effizienz bei der Softwareentwicklung gesehen. Auch der Nutzen lässt sich schwer kalkulieren. Nicht selten werben Werkzeughersteller mit Effizienzsteigerungen von 20% und mehr. Falls solche Aussagen überhaupt empirisch belegt sind, ist es fraglich, ob sich die Verhältnisse, in denen die Daten erhoben wurden, wirklich auf die eigene Situation übertragen lassen. Softwareentwicklung ist ein komplexer Prozess mit sehr vielen, kaum zu kontrollierenden Einflussvariablen. Der eigene Nutzen kann höchstens durch Pilotversuche in der eigenen Organisation untersucht werden. Dazu ist jedoch unter Umständen eine hohe Anfangsinvestition notwendig. Und der wirkliche Nutzen stellt sich nicht selten erst langfristig ein.

6.5 Auswahlkriterien bei der Anschaffung von Werkzeugen II

Anforderungen

Der Nutzen stellt sich erst ein, wenn die Anforderungen erfüllt werden. Anwender haben Anforderungen an Funktionalität und an Qualität. Darüber hinaus ist auch der Werkzeughersteller selbst inklusive seines Lizenzmodells und seiner Unterstützung mit entscheidend für die Auswahl eines Werkzeugs.

Zu den funktionalen Anforderungen an ein Softwareentwicklungswerkzeug sind folgende Fragen zu stellen:

Funktionale
Anforderungen

- Wie unterstützt das Werkzeug eine konkrete Methode beziehungsweise ein Vorgehensmodell?
- Welche Operationen implementiert das Werkzeug für welche Dokumente?
- Wie sieht hierbei die Abdeckung des Werkzeugs über die Operationen und Ebenen der Programmbeschreibungen aus?
- Wie komplementär ist das Werkzeug zu bereits eingesetzten Werkzeugen?
- Ist das Werkzeug in der Lage, mit anderen Werkzeugen zusammenzuwirken?
- Welchen Grad der Automatisierung bietet das Werkzeug?
- Wenn es sich um ein Werkzeug handelt, das Programmcode verarbeitet: Welche Programmierparadigmen und -sprachen werden unterstützt?
- Ist das Werkzeug multi- und interprojektfähig?

Weitere Qualitätsaspekte, die in Betracht gezogen werden müssen, lauten wie folgt:

Qualität

- Sind die vorhandenen Entwickler in der Lage, sich mit vertretbarem Aufwand ein Werkzeug anzueignen und es zu beherrschen?
- Welche Anforderungen an Speicher und Rechenleistung hat das Werkzeug?
- Skaliert das Werkzeug mit der Anzahl der Entwickler und dem Umfang der damit zu erstellenden Software?
- Kann das Werkzeug Schritt für Schritt in eine Organisation eingeführt werden?
- Wie robust und hoch verfügbar ist das Werkzeug?

Über die funktionalen und nichtfunktionalen Anforderungen hinaus spielt auch der Werkzeughersteller selbst eine große Rolle bei der Entscheidung, weil es sich beim Werkzeugkauf meist um eine langfristig angelegte Investition handelt, die nicht ohne Weiteres zurückgenommen werden kann.

Hersteller

- Wie sieht das Lizenzmodell des Werkzeugbauers aus?
- Wie sicher ist es, dass der Werkzeugbauer das Werkzeug auch langfristig wartet?
- Wie gut ist die technische und organisatorische Unterstützung des Werkzeugbauers?

II 6 Werkzeuge

- Welche Haftungen übernimmt der Werkzeugbauer und welche Garantien gibt er?
- Wie sehen die langfristigen Perspektiven bei der Weiterentwicklung des Werkzeugs aus?
- Welchen Einfluss kann man als Kunde auf diese Entwicklung nehmen?
- Welche Schulungen und weitere Dienstleistungen übernimmt der Werkzeugbauer oder seine Partner rund um das Werkzeug?

Auch die Gemeinde, die sich möglicherweise zu einem Werkzeug gebildet hat, kann bei der Anschaffungsentscheidung mitbestimmend sein. Wenn man sich mit vielen weiteren Anwendern des Werkzeugs austauschen kann, ist man weniger abhängig vom Werkzeugbauer selbst.

6.6 Evaluationsverfahren für die Anschaffung

Evaluationskriterien für die Einschätzung von Entwicklungswerkzeugen wurden bereits im Kapitel »Auswahlkriterien bei der Anschaffung von Werkzeugen«, S. 87, genannt. Diese Kriterien werden in einem systematischen Evaluationsverfahren herangezogen. Das Verfahren beschreibt die einzelnen Schritte der Evaluation und ist damit mehr als nur die Überprüfung von Kriterien.

Besonderheiten Eine Evaluation von Entwicklungswerkzeugen besitzt folgende zusätzliche Besonderheiten im Vergleich zur Auswahl von Software im Allgemeinen:

- Die Benutzer sind Software-Experten (fest umrissene Zielgruppe); sie bringen damit ein technisches Verständnis für Software mit.
- Die Funktionalität ist gegen die gewählten Vorgehensmodelle und Entwicklungsmethoden zu evaluieren.
- Es ist das Zusammenspiel der Werkzeuge zu prüfen, insbesondere mit den bereits in der Organisation eingesetzten.

Wiese beschreibt ein allgemeines Evaluationsverfahren für die Auswahl beliebiger Software [Wies98]. Dieses Verfahren ist im Folgenden für den engeren Kontext der Entwicklungswerkzeugauswahl angepasst dargestellt. Es besteht aus den folgenden Schritten:

- 1 Rahmenbedingungen klären
- 2 Ziele definieren
- 3 Kriterien und Prüfverfahren festlegen
- 4 Marktübersicht
- 5 Grobauswahl
- 6 Angebotseinholung
- 7 Feinauswahl
- 8 Entscheidung

1 Rahmenbedingungen klären

Von einem groben Ziel des Werkzeugeinsatzes kann zu diesem Zeitpunkt bereits ausgegangen werden, da die Entscheidung für eine Evaluation bereits gefallen ist. Meist ist auch schon ein grober finanzieller Rahmen für die Anschaffung des Werkzeugs selbst abgesteckt.

Nun geht es darum, die Evaluation systematisch zu planen und durchzuführen. Zunächst ist deshalb zu klären, wer zu welchem Zweck in welchem Zeitraum mit welchen Mitteln und Entscheidungskompetenzen an der Evaluation teilnimmt. Die Evaluation wird einem Evaluator oder einer Gruppe von Evaluatoren übertragen. Bei der Evaluation sollten aber auch alle später von der Werkzeugeinführung Betroffenen involviert werden. Insbesondere die Entscheidung erfordert zusätzlich die Entscheider, die nicht selbst Evaluatoren sind. Die Evaluatoren werden in der Regel nur Empfehlungen aussprechen können.

2 Ziele definieren

Die meist anfangs erst vage formulierten Ziele des Werkzeugeinsatzes müssen nun verfeinert werden. Dazu muss zunächst der Bedarf ermittelt werden. Hierzu ist es notwendig, analog zur Softwareentwicklung selbst, eine Ist- und Sollanalyse durchzuführen.

Da es beim Einsatz von Werkzeugen in der Regel darum geht, die Softwareentwicklung effektiver und effizienter zu machen (andere Gründe könnten äußere Umstände sein, wie zum Beispiel Vorgaben durch Kunden oder Standards), ist der Grund für den Werkzeugeinsatz im Softwareentwicklungsprozess zu suchen. Die Ist-Analyse untersucht deshalb den Entwicklungsprozess auf dessen Schwachstellen. Dazu müssen alle Beteiligten involviert werden: Kunden oder deren Vertreter, Projektmanager und Entwickler aller Art (Architekten, Programmierer, Tester). Darüber hinaus muss eine Inventur vorhandener Werkzeuge und praktizierter Vorgehensmodelle und eingesetzter Methoden durchgeführt werden, da die neuen Werkzeuge sich in dieses existierende Umfeld später integrieren lassen müssen.

Istanalyse

In der Sollanalyse werden Anforderungen an die Werkzeuge erhoben, die die identifizierten Schwachstellen in der Entwicklung beseitigen sollen. Diese Ziele werden einerseits Top-down aus den obersten Unternehmenszielen abgeleitet, da alle in der Entwicklung zu ergreifenden Maßnahmen diesen Zielen dienen müssen. Die Ziele werden andererseits aber meist auch Bottom-up durch einen stark inhaltlich-funktionalen Fokus mit Bezug zu den konkreten in der Ist-Analyse identifizierten Schwachstellen und Rahmenbedingungen aufgestellt. Dies gilt insbesondere dann, wenn Ersatz für existierende Werkzeuge gesucht wird oder ein konkretes Entwicklungsproblem gelöst werden muss.

Sollanalyse

II 6 Werkzeuge

Entwickler
mitnehmen Die Entwickler – als spätere Benutzer – müssen bei der Sollanalyse mit einbezogen werden, um schon früh Akzeptanz sicher zu stellen. Zum einen werden sie es später sein, die die Werkzeuge benutzen werden. Zum anderen könnten sie die Werkzeuge gar als Bedrohung empfinden. Die Ängste der Entwickler könnten dann zu offener oder verborgener Ablehnung des Vorhabens führen. Soll beispielsweise ein Bug-Tracking-System eingeführt werden, in dem Entwickler Fehler protokollieren sollen, so könnten Programmierer fürchten, dass damit ihre Fehler öffentlich und gegen sie verwendet werden. Die Erhebung von Softwaremetriken kann in ähnlicher Weise die Angst schüren, dass nach solchen Metriken Leistung bemessen wird (was in der Tat mehr als fragwürdig wäre). Diese Bedenken müssen erkannt und bedacht werden. Sie werden selten offen ausgesprochen.

3 Kriterien und Prüfverfahren festlegen

Aus den Zielen werden die Evaluationskriterien abgeleitet. Grobe Kriterien werden durch Unterkriterien verfeinert. Die Kriterien werden gegeneinander gewichtet. Die Gewichtung ist jedoch nur für Kriterien praktikabel, die auf derselben relativen Ebene der Kriterienzerlegung stehen.

K.O.-Kriterien Insbesondere werden K.O.-Kriterien aufgestellt. Diese müssen jedoch mit einer besonderen Sorgfalt festgelegt und möglicherweise mit einer gewissen Toleranz ausgestattet werden. Sonst kann es passieren, dass ein Werkzeug verworfen wird, das ein K.O.-Kriterium nur knapp verfehlt, aber bei allen anderen Kriterien bestens dasteht.

Ordinalskala Die Kriterien werden meist in einer Ordinalskala abgestuft, wie zum Beispiel »voll erfüllt«, »teilweise erfüllt« und »nicht erfüllt«. Die Differenzierung muss angemessen sein. Eine zu detaillierte Differenzierung ist nur pseudogenau und nicht praktikabel.

Prüfverfahren Für jedes Kriterium muss festgelegt werden, wie die Einschätzung bestimmt wird. Dies ist umso bedeutsamer, wenn mehrere Personen unabhängig voneinander die Evaluation vornehmen sollen. Ansonsten könnten die Kriterien unterschiedlich bewertet werden, wenn die Evaluatoren unterschiedliche Vorstellungen haben.

Aufgaben &
Szenarien Weil abstrakte Kriterien schwer einheitlich einzuschätzen sind, sollten konkrete Handlungsanweisungen und Szenarien für die Evaluation vorgesehen werden. Dazu werden Aufgaben und Szenarien definiert, für die das Werkzeug später eingesetzt werden soll. In einer szenariobasierten Evaluation wird der Evaluator das Werkzeug imaginär oder auch real in einem konkreten Szenario auf die Aufgabe ansetzen, um zu beurteilen, inwieweit das Werkzeug die Anforderung erfüllt. Die Aufgaben und Szenarien müssen realistisch gewählt werden und sollten das spätere Einsatzgebiet ausreichend abdecken.

Wenn die Aufgaben und Szenarien einheitlich für alle Werkzeuge definiert sind, spricht man auch von einem »Benchmark-Test«. Die verschiedenen Werkzeuge werden dann anhand definierter Kriterien in einer relativen Rangfolge angeordnet. Dies erlaubt einen relativen Vergleich.

Benchmark-Test

4 Marktübersicht

Der Evaluator verschafft sich durch Recherche einen Überblick über den entsprechenden Werkzeugmarkt. Suchmaschinen und Online-Lexika werden viele Hinweise liefern, da Firmen, die Entwicklungswerkzeuge für Software anbieten, selbstverständlich im Internet präsent sind. Messen und Fachzeitschriften (möglicherweise sogar mit Testberichten) sowie Bekannte, Experten und Berater können weitere Hinweise bieten.

Ziel dieser Aktivität ist es, eine umfassende Übersicht über Werkzeuganbieter, deren Partner und deren Vertrieb sowie die Werkzeuge und ihre Eigenschaften zu gewinnen. Die Suche geht also in die Breite nicht in die Tiefe. Nicht selten kann aufgrund der gefundenen Information der Anforderungskatalog noch verfeinert und erweitert werden.

5 Grobauswahl

Die in der Marktanalyse gefundenen Werkzeuge werden anhand der über sie bekannten Information eingeschätzt. Die Erfüllung jeder einzelnen Anforderung an ein Werkzeug wird hierbei anhand der festgelegten Ordinalskala bewertet. Sollten Informationen zu einzelnen Punkten nicht vorhanden sein, können Fragen direkt an den Anbieter gestellt werden, sofern das Werkzeug in der unvollständigen Auswertung einen guten Eindruck macht. Ist genug Zeit vorhanden und die Anzahl der Werkzeuge in der Auswahl überschaubar, kann der Evaluator sich das Werkzeug demonstrieren lassen. Hierzu bieten Hersteller Präsentationen vor Ort, Demo-Versionen, Videos oder Webinars (Präsentationen über das Web) an.

Bei der Bewertung anhand einer Ordinalskala – wie »erfüllt«, »teilweise erfüllt«, »nicht erfüllt« – sollte jede Einschätzung ausreichend begründet und dokumentiert werden, damit sie nachvollziehbar ist. Aus der Liste der Werkzeuge wird eine bestimmte Anzahl der besten Kandidaten für eine genauere Evaluation in die engere Auswahl genommen. Die Anzahl richtet sich nach dem Aufwand für die detaillierte Evaluation und der dafür verfügbaren Zeit.

6 Angebotseinholung

Der Evaluator tritt spätestens jetzt in Kontakt mit den Werkzeuganbietern. Bei hochpreisigen Werkzeugen werden Preise selten im Internet oder in Katalogen veröffentlicht. Deshalb bittet der Evaluator neben weiterführenden Informationen auch um ein Angebot.

II 6 Werkzeuge

7 Feinauswahl

In der Feinauswahl werden die vielversprechendsten Werkzeuge genauer unter die Lupe genommen. Sie werden systematisch ausprobiert. Diese Evaluation hat viele Parallelen mit einer wissenschaftlich-empirischen Untersuchung. Aus diesem Grunde kann man Methoden einsetzen, die in der empirischen Wissenschaft für den Erkenntnisgewinn entwickelt wurden. Diese Methoden können für die Evaluation von Werkzeugen entsprechend adaptiert werden. In der empirischen Wissenschaft gibt es grob die folgenden Klassen von Methoden:

- Befragungen
- Fallstudien
- Kontrollierte Experimente

Kontrollierte
Experimente

Kontrollierte Experimente haben eine höhere Objektivität als andere Methoden [Prec01]. Die Ursache-Wirkungskette kann damit besser ergründet werden. Allerdings sind sie sehr aufwendig. Sie kommen daher nur in Fällen eines hohen Risikos bei der Anschaffung von Werkzeugen in Frage, also beispielsweise wenn ein sehr großes Unternehmen alle seine Entwickler mit einem Werkzeug ausstatten möchte. Verschiedene Versuchsgruppen könnten hier mit unterschiedlichen Werkzeugen dasselbe Problem bearbeiten. Dann könnten Dauer und Qualität der Problemlösung mit Hilfe von statistischen Methoden verglichen werden.

Für kontrollierte Experimente ist eine größere Anzahl von Probanden notwendig, die alle dieselbe Aufgabe lösen. Dies ist in industriellen Umgebungen selten machbar. Leider hat die Softwareentwicklung darüber hinaus eine enorme Menge von Einflussvariablen, die kaum alle in einem Experiment kontrolliert werden können – eine Eigenschaft, die Experimente in der Softwaretechnik grundsätzlich erschwert. Experimente sollen deshalb im Folgenden nicht mehr weiter betrachtet werden. Im günstigsten Fall haben jedoch Wissenschaftler bereits diese Arbeit für Praktiker geleistet [EnRo03].

Fallstudien

Fallstudien werden häufig auch In-Vivo-Experiment genannt, sind also Experimente, die im richtigen Leben stattfinden und nicht unter Laborbedingungen wie kontrollierte Experimente [Yin03]. Im Falle einer Werkzeugevaluation würde das Werkzeug in einem realen Projekt unter realen Bedingungen zum Einsatz kommen und dort untersucht werden. Die Einflussfaktoren werden also im Gegensatz zum Experiment nicht kontrolliert. Auch hier ist mehr Zuverlässigkeit in der Aussage zu erwarten als bei einer bloßen Befragung. Allerdings ist auch der Aufwand höher und eine entsprechende Lizenz muss vorhanden sein. Fallstudien kommen deshalb bei einer Werkzeugauswahl nur in Frage, wenn nur noch sehr wenige Werkzeuge in der Auswahl sind und das Risiko einer Fehlentscheidung bei der Werkzeuganschaffung hoch ist. Fallstudien sind jedoch ein probates

Mittel bei der inkrementellen Einführung des Werkzeugs. Hier wird das Werkzeug zunächst in einem Pilotprojekt in der eigenen Organisation ausprobiert. Die dabei gewonnenen Erkenntnisse helfen dann bei der flächendeckenden Einführung.

Bei Befragungen (Interviews oder Fragebogen) werden dem Benutzer eines Werkzeugs Fragen über die Produkteigenschaften gestellt. Er beantwortet die Fragen aufgrund der Erfahrungen, die er selbst mit dem Produkt gemacht hat. Eine systematische Erfahrungsgrundlage kann durch die Bearbeitung einer vorgegebenen Standardaufgabe mit dem Werkzeug geschaffen werden. Damit die Befragungen zu verschiedenen Werkzeugen vergleichbar sind, sollten auch die gleichen Personen befragt werden. Das heißt, dass dieselbe Person verschiedene Werkzeuge ausprobiert. Dies wirft andererseits die Frage der Reihenfolge der Evaluation aus. Lern- und Ermüdungseffekte könnten das Ergebnis beeinflussen. Zu entscheiden ist außerdem, ob der Evaluator die Befragung mit Benutzern durchführt oder ob er die Werkzeuge selbst erprobt und sich dann selbst befragt (Durchführungsprotokoll). In letzterem Fall hätte man nur eine Meinung. Die Evaluation ist damit aber auch weniger aufwendig. Außerdem setzt dieses Vorgehen voraus, dass der Evaluator repräsentativ für die späteren Benutzer ist.

Befragungen

Als Ergebnis von Befragungen erhält man »weiche Daten« wie beispielsweise »das System ist bequem zu bedienen«, aber keine »harten Daten« wie »Antwortzeiten, Fehlerraten«. Befragungen sind weniger aufwendig und universeller einsetzbar als alternative empirische Methoden. Auch unstrukturierte Probleme können eingekreist werden. Mögliche Übertreibungen des Befragten und Manipulationen des Interviewers können nachteilig sein. Hinzu kommt, dass Befragungen von den Befragten nicht immer geschätzt werden.

Die Befragung kann schriftlich oder mündlich erfolgen. Schriftliche Befragungen haben den Nachteil, dass der Benutzer schriftlich unter Umständen seine Eindrücke nicht präzise formulieren kann.

Bei der mündlichen Befragung haben sich drei Verfahren als sinnvoll erwiesen:

- Bei der Methode des lauten Denkens kommentiert der Benutzer seine jeweiligen Arbeitsschritte, Handlungsalternativen und Probleme. Nachteilig ist, dass der Benutzer durch die Kombination von Arbeit mit dem Produkt und gleichzeitiger Kommentierung in Schwierigkeiten gerät und durch Faszination oder Überlastung das laute Denken einstellt.
- Bei der konstruktiven Interaktion bearbeiten zwei Benutzer gemeinsam eine Aufgabe und »erzählen« sich gegenseitig, was sie jeweils tun beziehungsweise zu tun gedenken.

II 6 Werkzeuge

Experten-
einschätzung

- Bei der Videokonfrontation wird die Arbeit des Benutzers mit dem Produkt gefilmt. Anschließend wird ihm die Aufzeichnung vorgespielt, und er soll seine Verhaltensweise erläutern.

Bei der Experteneinschätzung wird ein Werkzeug durch einen Experten geprüft. Er orientiert sich dabei – anders als der Benutzer bei Befragungen – weniger an einer Aufgabe mit dem zu evaluierenden Produkt, sondern an fachspezifischen Fragestellungen. Experten benutzen dazu Prüflisten oder Leitfäden.

- Prüflisten unterscheiden sich hinsichtlich ihrer Operationalisierung, ihrer Präzision und ihrer Kontextbedingungen bezüglich der Prüfung. Prüflisten überlassen es dem Prüfer, wie er mit dem zu prüfenden Produkt umgeht, um zu Antworten auf die Prüffragen zu kommen. Die im Kapitel »Auswahlkriterien bei der Anschaffung von Werkzeugen«, S. 87, genannten Auswahlkriterien können Ausgangspunkt für solche Prüflisten sein.
- Leitfäden enthalten neben den Prüflisten Verfahrensvorschriften für die Durchführung der Evaluation. Oppermann et al. beschreiben zum Beispiel eine genaue Durchführungsvorschrift für die Anwendung von Prüffragen [OMR+92].

Ein allgemein anerkanntes, systematisches Verfahren zur Evaluation von Entwicklungswerkzeugen gibt es nicht. In manchen Situationen kann es auch sinnvoll sein, ein eigenes Evaluationsverfahren zu entwickeln. Englisch beschreibt, wie Evaluationsverfahren methodisch entwickelt werden können [Engl93].

8 Entscheidung

Schließlich gelangt die Evaluation zur Entscheidung. Die Ergebnisse der Evaluation zusammen mit der genauen Vorgehensweise werden den Entscheidern und auch den von der Entscheidung betroffenen Personen vorgestellt. Die Entscheider wählen dann das Werkzeug aus. Damit ist die Evaluation aber in der Regel noch nicht vollständig abgeschlossen. Das Werkzeug sollte kontinuierlich in seinem Einsatz – mindestens aber in seiner Einführungsphase – weiter evaluiert werden, um den tatsächlichen Nutzen nachzuweisen und die Kosten zu bestimmen. Hierfür sind entsprechende messbare Parameter festzulegen, die vor und während sowie nach der Einführung erhoben werden. Diese Parameter müssen Aussagen über die Effektivität und Effizienz in der Entwicklung zulassen. Stellt sich der erhoffte Nutzen im Laufe der Zeit nicht ein, müssen die Gründe dafür gesucht werden. Nicht selten sind Probleme auf unzureichende Ausbildung in der Benutzung oder gar Widerstände gegen das Werkzeug seitens der Entwickler zurückzuführen. Die Ursachen müssen erforscht und durch Schulungen, Prozessänderungen oder Ergänzungen am Werkzeug beseitigt werden. Im schlimmsten Fall muss die Entscheidung für den Werkzeugeinsatz zurückgenommen werden.

6.7 Zusammenfassung

Softwareentwicklungswerkzeuge können helfen, die Entwicklung von Software effektiver und effizienter zu machen. Voraussetzung hierfür ist, dass sie zur Aufgabe, die sich dem Entwickler stellt, passen und der Entwickler mit den Werkzeugen zurecht kommt. Das Werkzeug ist dabei stets Teil eines systematischen Vorgehens, das heißt einer Methode.

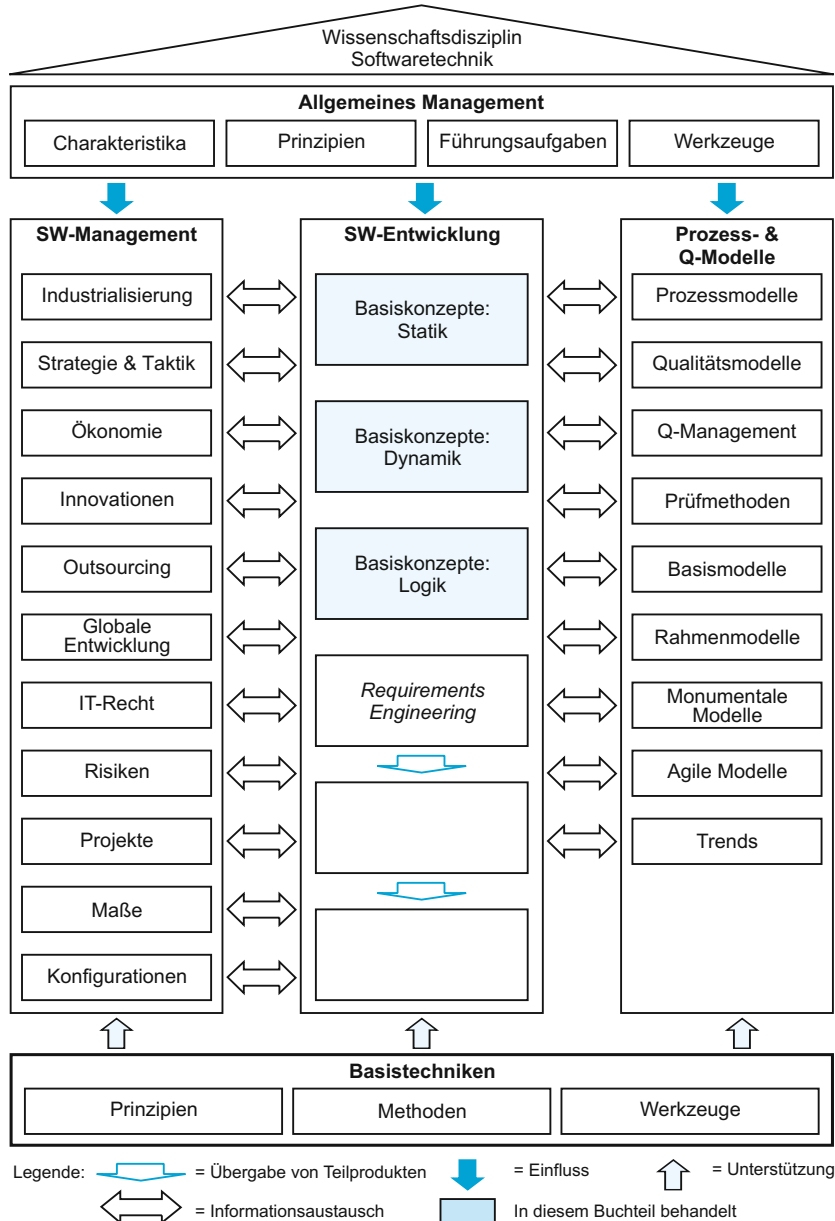
Die Palette heute verfügbarer Werkzeuge ist reichhaltiger denn je. Eine Klassifikation der Werkzeuge hilft, den Überblick zu behalten und Werkzeuge vergleichen zu können. Ein Werkzeug implementiert Operationen für Dokumente, die im Zuge der Entwicklung entstehen. Anhand dieser Operationen und der Abstraktionsebene, der die Dokumente angehören, lassen sich Werkzeuge einordnen.

Da ein Werkzeug allein nicht alle Aktivitäten in der Softwareentwicklung unterstützen kann, ist die Interoperabilität der Werkzeuge von großer Bedeutung. Integrierte Entwicklungsumgebungen sind Plattformen, die eine Infrastruktur für die Integration verschiedener Werkzeuge in einer einheitlichen Benutzungsschnittstelle bieten.

Dabei steht heute nicht nur mehr die Programmierung im Fokus der Werkzeuge. Alle Aufgaben in der Entwicklung von der Anforderungsanalyse über den Entwurf bis hin zur Programmierung und den Test und nicht zuletzt der Evolution und Wartung werden unterstützt. Die modellgetriebene Entwicklung ist ein Trend, der die Modellierung in den Vordergrund stellt, und von der Werkzeugunterstützung lebt. Modelle sind hier formale Systembeschreibungen, aus denen heraus automatisch Code generiert wird. Dieser Prozess ist durchgängig von Werkzeugen unterstützt, die Modelle transformieren und schließlich in Code umwandeln. Weil Softwareentwicklung in großem Maße von der Kommunikation und Zusammenarbeit vieler Entwickler abhängt, unterstützen Werkzeuge auch die Kollaboration und Kommunikation zwischen Teams – auch über Kontinente hinweg.

Bei der Auswahl geeigneter Werkzeuge ist eine Vielzahl von Kriterien zu beachten. Sie sind die Grundlage für systematisches Vorgehen bei der Auswahl von Werkzeugen. Ein systematisches Vorgehen beschreibt die einzelnen Schritte der Evaluation und ist damit mehr als nur die Überprüfung von Kriterien. Diese Schritte gehen von den Rahmenbedingungen und klaren Unternehmenszielen aus, legen die Kriterien und ihre Prüfverfahren fest und engen dann systematisch die Auswahl auf die geeigneten Werkzeuge für die endgültige Entscheidung ein.

III Basiskonzepte



III III Basiskonzepte

In der Wissenschaftsdisziplin Softwaretechnik sind im Laufe der Jahrzehnte viele Konzepte entstanden und auch wieder verschwunden. Eine Reihe von Konzepten hat sich aber als stabil erwiesen. Diese Konzepte – bezogen auf die Softwareentwicklung – werden hier vorgestellt. Dabei wird sich auf Basiskonzepte konzentriert. Interessanterweise sind nicht alle Basiskonzepte neu. Einige Basiskonzepte entstanden bereits in den Anfangszeiten der Softwaretechnik oder wurden aus anderen Disziplinen übernommen. Dazu gehören Zustandsautomaten (1954), Petrinetze (1962) und Entscheidungstabellen (1957).

Basiskonzepte

Konzepte erlauben es, definierte Sachverhalte unter einem oder mehreren Gesichtspunkten zu modellieren. Ein Basiskonzept der Softwareentwicklung lässt sich durch folgende Eigenschaften charakterisieren:

- Atomares Konzept,
- konzeptionell langlebig,
- phasenübergreifend verwendbar,
- in unterschiedlichen Kontexten einsetzbar.

Die erste Eigenschaft besagt, dass ein Konzept *elementar* und *originär* ist. Es ist nicht auf andere Basiskonzepte reduzierbar. Um von einem Basiskonzept zu sprechen, müssen die *erste* Eigenschaft und *mindestens eine* der anderen Eigenschaften erfüllt sein.

Notationen

Eine Notation stellt Informationen durch Symbole dar. Ein Konzept kann durch eine oder mehrere Notationen dargestellt werden. In Abhängigkeit von der Notation ist das betreffende Basiskonzept z. T. semantisch leicht modifiziert. Im Allgemeinen sind die Notationen aber gegenseitig substituierbar. Notationen lassen sich in die Kategorien *textuell* – *grafisch* und *informal* – *formal* klassifizieren. Die Abb. 6.0-2 zeigt die sich daraus ergebenden vier Quadranten.

Textuell	Texte beschreiben Informationen in Form von natürlichsprachlichen (z. B. deutschen) Texten – oft auch als Prosa (von: Schrift in ungebundener Form) oder verbale Beschreibung (von: wörtlich, mit Worten) bezeichnet.
Grafisch	Grafiken beschreiben Informationen durch grafische Symbole und Linien zwischen den Symbolen. Die Symbole und Linien werden fast immer durch Namen oder Texte annotiert.
Formal	Unter formaler Beschreibung versteht man im Allgemeinen die Verwendung formaler, textueller Sprachen. Eine formale Sprache wird durch eine Grammatik definiert, die angibt, welche Zeichen-

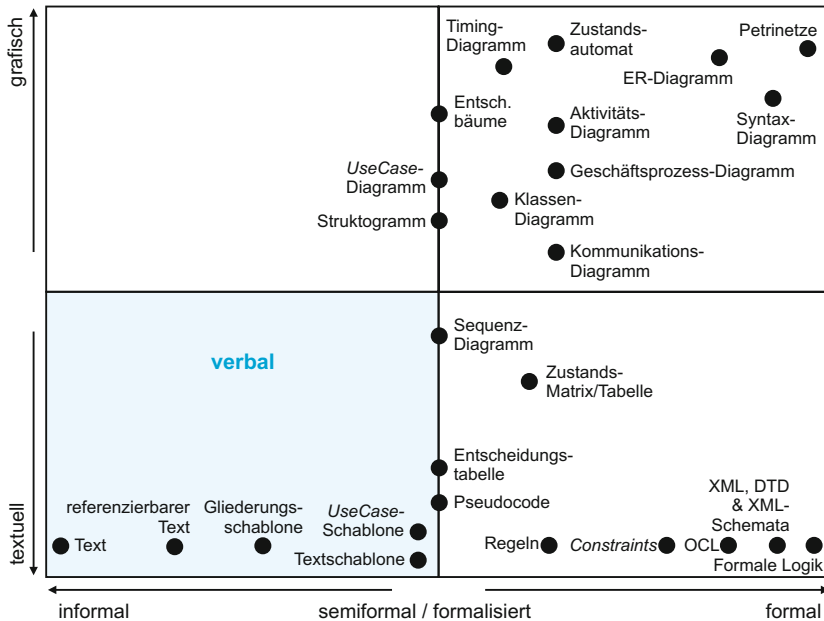


Abb. 6.0-2: Klassifizierung der Basiskonzepte nach der Darstellungsform ihrer Notationen.

ketten erlaubte Sätze der Sprache sind. Die Verbindung »formale Beschreibung« und »textuelle Darstellung« muss aus heutiger Sicht erweitert werden auf »grafische Darstellung«. Der hier verwendete Begriff »formal« meint nur »syntaktisch formal«. Die Semantik ist damit noch nicht formal definiert. Die formale Definition der Semantik gelingt in der Informatik nur in Sonderfällen, selbst Programmiersprachen sind in ihrer Semantik *nicht* formal definiert.

Auch für grafische Darstellungen kann eine Grammatik festgelegt werden. Eine grafische Syntax bestimmt, wie die Grafiken aufgebaut sind.

Hinweis: Visuelles Programmieren

Zwischen einer formalen und einer informalen, d. h. an keine Regeln gebundenen Beschreibung, liegt ein Bereich, den man als semi-formale oder formalisierte Beschreibung bezeichnet. Diese Beschreibungen sind stärker formalisiert als die Umgangssprache, aber nicht streng mathematisch wie eine formale Sprache durch eine Grammatik. Viele Basiskonzepte erlauben mehr als eine Notation oder mischen mehrere Notationen, z. B. semiformalen Text und Grafiken.

Semiformal, formalisiert

Als Standard für die Beschreibung vieler Basiskonzepte hat sich die **UML** (*Unified Modeling Language*) durchgesetzt – standardisiert als ISO/IEC 19501, siehe OMG UML-Website (<http://www.uml.org/>). Die UML 2 unterscheidet 11 Diagrammarten, wobei viele miteinander in Bezug stehen. Während die UML für die Modellierung von Softwaresystemen konzipiert wurde, gibt es für die Modellierung von Systemen **SysML** (*Systems Modeling Language*), siehe OMG SysML Website

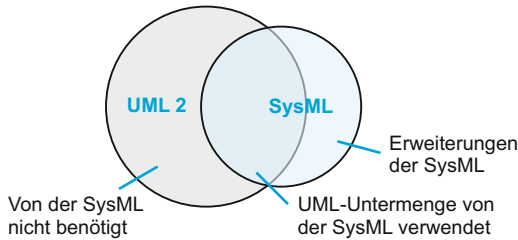
UML & SysML



III III Basiskonzepte

(<http://www.omg.sysml.org/>). Sie basiert auf der UML 2 und dient der ganzheitlichen Modellierung von Systemen – unabhängig von Software oder Hardware (Abb. 6.0-3) (siehe auch [FMS08]).

Abb. 6.0-3: SysML umfasst eine Untermenge der UML 2 und wird durch zusätzliche Modellelemente erweitert.



Hinweis

Die UML ist für die Modellierung von Echtzeit- und von Kommunikationssystemen nur bedingt geeignet. Es gibt aber eine Reihe von Büchern, in denen gezeigt wird, wie die UML auch für diese Gebiete eingesetzt werden kann, z. B. [Doug04] und [Dold03].

Sichten

Ein Softwaresystem kann unter folgenden drei Gesichtspunkten, Sichten oder Perspektiven beschrieben, spezifiziert, modelliert, analysiert, entworfen und programmiert werden:

■ **Statik:** Stabiler Aufbau des Systems untergliedert in:

- ☐ Funktionalität
- ☐ Daten

■ **Dynamik:** Dynamisches Verhalten des Systems untergliedert in:

- ☐ Fachaspekte
- ☐ Kontrollstrukturen
- ☐ Geschäftsprozesse und *Use Cases*
- ☐ Zustandsautomaten
- ☐ Petrinetze
- ☐ Zeitaspekte
- ☐ Szenarien

■ **Logik:** Logische Abhängigkeiten innerhalb des Systems untergliedert in:

- ☐ Formale Logik
- ☐ *Constraints* und die OCL in der UML
- ☐ Entscheidungstabellen
- ☐ Regeln

Diese verschiedenen Sichten sind *nicht* unabhängig voneinander. In der Regel werden mehrere der Basiskonzepte in Bezug gesetzt. Die UML ermöglicht es, verschiedene Konzepte zu integrieren.



In der UML wird zwischen **Struktur** (vergleichbar mit der hier verwendeten Bezeichnung »Statik«) und **Verhalten** (vergleichbar mit der hier verwendeten Bezeichnung »Dynamik«) unterschieden. Die Logik kann in der UML an verschiedenen Modellelementen annotiert oder es kann die OCL (*Object Constraint Language*) verwendet werden (siehe »Constraints und die OCL in der UML«, S. 377).

In der UML 2 werden sechs verschiedene Strukturdiagramme unterschieden¹:

- **Klassendiagramm**
- **Objektdiagramm**
- **Paketdiagramm**
- **Komponentendiagramm**
- Verteilungsdiagramm
- Kompositionsstrukturdiagramm

Sieben Diagramme erlauben es, das Verhalten zu modellieren:

- **Aktivitätsdiagramm**
- **Use Case-Diagramm** (auch Anwendungsfalldiagramm genannt)
- **Zustandsdiagramm**
- **Timing-Diagramm** (auch Zeitverlaufdiagramm genannt)
- **Sequenzdiagramm**
- **Kommunikationsdiagramm**
- Interaktionsübersichtsdiagramm

Nicht alle diese Diagramme beschreiben Basiskonzepte. Daher werden hier nur die fett dargestellten Diagrammartentypen im Rahmen der Basiskonzepte behandelt.

In der UML können mehrere Diagrammartentypen miteinander kombiniert werden, wenn es für die Modellierung nützlich ist. Viele Diagramme erlauben eine hierarchische Verfeinerung.

UML-Profile

Die UML ist als Beschreibungs- und Modellierungssprache *nicht* für alle Basiskonzepte geeignet. Es gibt in der UML jedoch die Möglichkeit, **UML-Profile** für spezielle Zwecke zu erstellen [OMG09a, S. 653 ff.]. Das Sprachkonzept Profil (*profile*) ist ein leichtgewichtiger Erweiterungsmechanismus der UML – leichtgewichtig, weil er das Metamodell der UML *nicht* verändert (siehe auch »Prinzip der Abstraktion«, S. 26).

Ein Profil ist ein Paket (*package*), das es erlaubt, Metaklassen des UML-Metamodells zu erweitern, um sie an spezielle Zwecke anzupassen. Ein Profil übernimmt die Notation und die Semantik eines

¹Einen kompakten Überblick über alle wichtigen Diagramme der UML bietet die *Quick Reference Map »UML 2«* (2. Auflage) von Heide Balzert auf 8 laminierten und faltbaren Seiten (W3L-Verlag, ISBN 978-3-937137-75-9).

III III Basiskonzepte

Pakets (siehe auch »Pakete«, S. 145). Vor den Paketnamen wird »profil« geschrieben (Abb. 6.0-4). Ein Profil-Paket kann andere Pakete importieren (»import«).

Beispiel 1a In der Abb. 6.0-4 wird das Paket `Types` von dem Profil `euroNorm` importiert. Der Datentyp `Typpruefung` wird als ein Typ in dem Stereotyp `Euro5Diesel` verwendet. Die Klasse `JavaInteger` wird ebenfalls als ein Attributtyp eingesetzt.

Stereotyp Die Erweiterungen erfolgen im Wesentlichen durch die Definition von Stereotypen. Ein **Stereotyp** legt fest, wie eine im Metamodell der UML vorgegebene Metaklasse für spezifische Zwecke angepasst werden kann. Um einen Stereotyp von anderen Klassen zu unterscheiden, wird ein Schlüsselwort in französischen Anführungszeichen (Guillemets) über der Klasse angegeben. Ein Stereotyp kann über Attribute verfügen, die auch als Eigenschaftsdefinitionen (*tag definition*) bezeichnet werden (Abb. 6.0-4). Außerdem können *Constraints* angegeben werden.

Anwendung Um die Elemente eines Profils nutzen zu können, muss das Profil angewendet werden (»applied«) (Abb. 6.0-4). Die Anwendungsbeziehung ist eine spezielle Importbeziehung. Das Paket importiert alle Elemente aus dem Profil. Die Erweiterung, die das Profil definiert, gilt für das entsprechende Paket. Für jede Eigenschaftsdefinition des Stereotypen kann bei der Anwendung ein Eigenschaftswert (*tagged value*) ergänzt werden. Die Eigenschaftswerte werden mit vorangestelltem Stereotypbezeichner als Attribute in das Element oder alternativ in ein Kommentarsymbol eingetragen und mit dem Element verbunden, auf den der Stereotyp angewendet wird.

Beispiel Die Abb. 6.0-4 zeigt, wie der Stereotyp `Euro5Diesel` in der Klasse `PKW` angewandt wird. Die Typen aus dem Paket `Types` sind ebenfalls in dem Paket `fahrzeuge` anwendbar.

Klassifikationskriterien

Basiskonzepte lassen sich nach folgenden Kriterien klassifizieren:

- Notation: textuell – grafisch, informal – formal (siehe Abb. 6.0-2)
- Zeitlicher Einsatz: In welchen Entwicklungsphasen einsetzbar
- Anwendungsbereiche: Software, softwareintensive Systeme

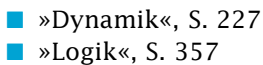
i Im Folgenden werden eine kaufmännisch/administrative Fallstudie und eine softwareintensive Fallstudie in Form verbal beschriebener Lasten- und Pflichtenhefte vorgestellt:

- »Fallstudie: SemOrg – Die Spezifikation«, S. 107
- »Fallstudie: Fensterheber – Die Spezifikation«, S. 117

Anschließend werden die einzelnen Basiskonzepte vorgestellt und ihr Einsatz anhand der Fallstudien demonstriert:

- »Statik«, S. 127

Abb. 6.0-4: Beispiel für die Definition und die Anwendung eines UML-Profiles.



III III Basiskonzepte

Syntaxnotation

Für die Beschreibung einer Syntax gibt es verschiedene Möglichkeiten. Hier wird folgender Beschreibungsformalismus für die Syntax verwendet²:

- **Syntaxsymbole**: Symbole, die zur Beschreibung der Syntax verwendet werden, sind kursiv dargestellt, z.B. { }, sonst handelt es sich um terminale Symbole.
- **Schlüsselwörter**: Schlüsselwörter sind fett dargestellt.
- **/ = oder**, z.B. + / -, d.h. + oder - muss gewählt werden.
- **[] = optional**, z.B. [*Klasse* / *Datentyp* / *Aufzählungstyp* / *Primitiver Datentyp*], d.h. die Typangabe Klasse usw. ist optional und kann weggelassen werden.
- **{ } = Auswahl** notwendig, z.B. [{ **private** / **public** / **protected** / **final** / **static** }], d.h. in der Regel sind oder-Alternativen in eine geschweifte Klammer eingeschlossen. Eine der Alternativen muss gewählt werden.
- **+ = kann wiederholt** werden, z.B. [{ **private** / **public** / **protected** / **final** / **static** }]+ *Type*, d.h. die Angaben in geschweiften Klammern können mehrfach angegeben werden.
- **... = Liste** (Elemente durch "," getrennt), z.B. [**readOnly** / **union** / **subsets**<attr> / **redefines**<attr> / **ordered** / **unique** / *Einschränkung -Attr*] ..., d.h. mehrere Angaben können aufgeführt werden, jeweils durch ein Komma getrennt.
- **A ::= B** (A wird **definiert** durch B), z.B. *Sichtbarkeit* ::= [- / # / + / ~].
- **Platzhalter**, z.B. *Type* im Beispiel »wiederholt«, d.h. *Type* wird *nicht* hingeschrieben, sondern es wird an anderer Stelle definiert und steht hier nur als **Platzhalter**.
- **Voreinstellungen** sind unterstrichen, z.B.
Generalisierungseigenschaften ::= [{ incomplete, disjoint } / { complete, disjoint } / { incomplete, overlapping } / { complete, overlapping }]

Ausbildung



Für jeden Softwareentwickler und insbesondere jeden *Requirements Engineer* bzw. Systemanalytiker gehört die Beherrschung der UML zum täglichen »Handwerkszeug«. Die **OMG** als Entwickler der UML hat ein Ausbildungsprogramm unter der Bezeichnung »OMG Certified UML Professional™« veröffentlicht, siehe Website UML Professional (<http://www.omg.org/uml-certification/index.htm>). Es werden drei Kenntnisstufen unterschieden: *Fundamental*, *Intermediate* und *Advanced*. Die hier behandelten UML-Konzepte decken weitgehend die Kenntnisstufe *Fundamental* ab.

²Es handelt sich um eine Pseudo-EBNF, angelehnt an die EBNF. Bei der EBNF (Extended Backus-Naur-Form) handelt es sich um eine Notation für die Beschreibung der Syntax von Programmiersprachen.

7 Fallstudie: SemOrg – Die Spezifikation

Um die verschiedenen Techniken und Konzepte zu veranschaulichen, wird eine **kaufmännische Fallstudie SemOrg** (für Seminarorganisation) durchgängig verwendet und referenziert. Das folgende **Lastenheft** und **Pflichtenheft** verwendet nur natürlichsprachliche Anforderungen zur Festlegung der Anforderungen. Der Aufbau von Lasten- und Pflichtenheft ist im Kapitel »Schablonen für Lastenheft, Pflichtenheft und Glossar«, S. 492, erklärt. Fachbegriffe werden in einem **Glossar** definiert (siehe unten). TBD steht für *To be defined* – muss noch definiert werden bzw. *To be determined* – muss noch festgelegt werden.

Lastenheft SemOrg

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Manfred Muster- mann	Geschäfts- führer Teachware	in Bear- beitung	10/09	

Voreinstellungen (*Kursiv dargestellt*):

Priorität aus Auftraggebersicht = {hoch, *mittel*, niedrig}

Priorität aus Auftragnehmersicht = {hoch, *mittel*, niedrig}

Stabilität der Anforderung = {fest, *gefestigt*, volatil}

Kritikalität der Anforderung = {hoch, mittel, *niedrig*, keine}

Entwicklungsrisiko der Anforderung = {hoch, mittel, *niedrig*}

1 Visionen und Ziele

/LV10/ Die Firma Teachware soll durch das System in die Lage versetzt werden, die von ihr veranstalteten Seminare sowie Kunden und Dozenten effizient rechnerunterstützt zu verwalten.

/LV20/ Die Kunden der Firma Teachware sollen über das Web möglichst viele Vorgänge selbst durchführen können.

/LZ10/ Ein Interessent oder ein Kunde kann mindestens 20 Stunden jeden Tag Seminare und Veranstaltungen über das Web selektieren und eine Veranstaltung online buchen, damit die Mitarbeiter der Fa. Teachware von solchen Tätigkeiten entlastet werden.

III 7 Fallstudie: SemOrg – Die Spezifikation

2 Rahmenbedingungen

/LR10/ SemOrg ist eine kaufmännisch/administrative Web-Anwendung.

/LR20/ Zielgruppe sind die Mitarbeiter der Fa. Teachware (Kundensachbearbeiter, Seminarsachbearbeiter, Veranstaltungsbetreuer) sowie Interessenten und Kunden.

3 Kontext und Überblick

/LK10/ Das System besitzt eine Softwareschnittstelle zu einem Buchhaltungssystem.

/LK20/ Das System ist mindestens 20 Stunden pro Tag im Intranet der Fa. Teachware und im Internet verfügbar.

4 Funktionale Anforderungen

/LF10/ Das System *soll* Interessenten und Kunden die Möglichkeit bieten, sich über Seminare und Veranstaltungen zu informieren, Veranstaltungen zu buchen und einen Seminarkatalog anzufordern.

/LF20/ Das System *muss* dem Kundensachbearbeiter die Möglichkeit bieten, neue Kunden/Firmen zu erfassen und vorhandene Kunden-/Firmendaten zu aktualisieren und Kunden/Firmen zu löschen.

/LF30/ Das System *muss* dem Kundensachbearbeiter die Möglichkeit bieten, Seminare und Veranstaltungen zu selektieren, Veranstaltungen für Interessenten und Kunden zu buchen und für angeforderte Seminarkataloge Versandpapiere zu erstellen.

/LF40/ Das System *muss* dem Seminarsachbearbeiter die Möglichkeit bieten, neue Dozenten zu erfassen und vorhandene Dozentendaten zu aktualisieren und Dozenten zu löschen.

/LF50/ Das System *muss* dem Seminarsachbearbeiter die Möglichkeit bieten, neue Seminare und Veranstaltungen zu erfassen, vorhandene zu modifizieren oder zu löschen.

/LF60/ Das System *soll* dem Seminarsachbearbeiter die Möglichkeit bieten, für alle Veranstaltungen Hotels auszuwählen und Räume zu reservieren.

/LF70/ Das System *muss* Kunden-, Firmen-, Seminar-, Veranstaltungs- und Dozentendaten permanent speichern.

/LF80/ Das System *muss* fähig sein, dem Buchhaltungssystem Rechnungsdatensätze mindestens einmal am Tag zur Verfügung zu stellen.

5 Qualitätsanforderungen

/LQE10/ Alle Reaktionszeiten auf Benutzeraktionen müssen unter 5 Sekunden liegen.

Systemqualität	sehr gut	gut	normal	nicht relevant
Funktionalität		X		
Zuverlässigkeit			X	
Benutzbarkeit		X		
Effizienz			X	
Wartbarkeit			X	
Portabilität				X

Tab. 7.0-1: Qualitätsanforderungen an SemOrg.

Glossar SemOrg

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Manfred Muster- mann	Geschäfts- führer Teachware	in Bear- beitung	10/09	

Dozent: Führt als freier Mitarbeiter eine oder mehrere angebotene →Veranstaltungen durch. Ist fachlich in der Lage, ein oder mehrere →Seminare abzuhalten.

Firma: Mitarbeiter einer Firma (Ansprechpartner), der für die Aus- und Weiterbildung von Mitarbeitern zuständig ist und sich über Dienstleistungen informiert oder Mitarbeiter zu öffentlichen →Veranstaltungen schickt oder firmeninterne Veranstaltungen bucht.

Interessent: →Kunde, der sich für Dienstleistungen, z.B. Seminar-katalog, interessiert, aber noch an keiner →Veranstaltung teilgenommen hat.

Kunde: Mitarbeiter einer Firma oder Privatperson, der bzw. die an Dienstleistungen interessiert ist, oder ein Seminar bucht und besucht (→Teilnehmer, →Interessent).

Kundensachbearbeiter: Verantwortlich für die Kommunikation mit →Kunden und →Firmen einschließlich der Auskunftserteilung und Buchung.

Seminar: →Seminartyp.

Seminarsachbearbeiter: Verantwortlich für die Planung und Terminierung von →Seminaren und →Veranstaltungen. Zuständig für die Kommunikation und Akquirierung von →Dozenten.

Seminartyp: Beschreibt die Gemeinsamkeiten, die eine Menge von →Veranstaltungen besitzen wie Titel, Zielsetzung, Inhalt, Voraussetzungen.

Seminarveranstaltung: →Veranstaltung.

Teilnehmer: →Kunde, der an einer →Veranstaltung teilnimmt bzw. teilgenommen hat.

Veranstaltung: →Seminar, das zu einem festgelegten Zeitpunkt, an einem festgelegten Ort von einem oder mehreren →Dozenten durchgeführt wird.

III 7 Fallstudie: SemOrg – Die Spezifikation

Veranstaltungsbetreuer: Betreut die →Teilnehmer und
→Dozenten einer →Veranstaltung.

Pflichtenheft SemOrg

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Manfred Muster- mann	Geschäfts- führer Teachware	in Bear- beitung	11/09	Verfeine- rung des Lastenhefts V0.1

Voreinstellungen (*Kursiv dargestellt*):

Priorität aus Auftraggebersicht = {hoch, *mittel*, niedrig}

Priorität aus Auftragnehmersicht = {hoch, *mittel*, niedrig}

Stabilität der Anforderung = {fest, *gefestigt*, volatil}

Kritikalität der Anforderung = {hoch, mittel, *niedrig*, keine}

Entwicklungsrisiko der Anforderung = {hoch, mittel, *niedrig*}

Hinweis

Alle Anforderungen im Pflichtenheft, die einen Bezug zu einer Anforderung im Lastenheft haben, müssen eine entsprechende Referenz auf das Lastenheft haben. Da das Pflichtenheft eine Verfeinerung und Erweiterung des Lastenhefts darstellt, gibt es neue Anforderungen, von denen kein Bezug zum Lastenheft hergestellt werden kann. Es gibt dann keine entsprechende Referenz.

1 Visionen und Ziele

/V10/ (/LV10/) Die Firma Teachware soll durch das System in die Lage versetzt werden, die von ihr veranstalteten Seminare sowie Kunden und Dozenten effizient rechnerunterstützt zu verwalten.

/V20/ (/LV20/) Die Kunden der Firma Teachware sollen über das Web möglichst viele Vorgänge selbst durchführen können.

/Z10/ (/LZ10/) Ein Interessent oder ein Kunde kann mindestens 20 Stunden jeden Tag Seminare und Veranstaltungen über das Web selektieren und eine Veranstaltung online buchen, damit die Mitarbeiter der Fa. Teachware von solchen Tätigkeiten entlastet werden.

2 Rahmenbedingungen

/R10/ (/LR10/) SemOrg ist eine kaufmännisch/administrative Web-Anwendung.

/R20/ (/LR20/) Zielgruppe sind die Mitarbeiter der Fa. Teachware (Kundensachbearbeiter, Seminarsachbearbeiter, Veranstaltungsbetreuer) sowie Interessenten und Kunden.

/R30/ Das System wird in einer Büroumgebung eingesetzt.

/R40/ (/LZ10/) Die tägliche Betriebszeit des Systems muss mindestens 20 Stunden jeden Tag betragen.

/R50/ Der Betrieb des Systems muss unbeaufsichtigt ablaufen.

/R60/ Eingesetzte Software auf der Zielmaschine: Client: Webbrowser (Die marktführenden 3 Webbrowser müssen unterstützt werden), Server: Betriebssystem Windows.

/R70/ Hardwarevoraussetzungen: Client: PC, Bildschirm mit mindestens XGA-Auflösung (1024 x 768), Server: TBD.

/R80/ (/LK10/) Netzwerkverbindung des Servers zum Buchhaltungssystem.

/R90/ (/LK20/) Alle Clients sind über ein Intranet mit dem Server verbunden, der Server hat einen Internetanschluss.

/R100/ Die Entwicklungsumgebung kann identisch mit der Zielumgebung sein.

3 Kontext und Überblick

/K10/ (/LK10/) Das System besitzt eine Softwareschnittstelle zu einem Buchhaltungssystem: TBD.

4 Funktionale Anforderungen

/F10/ (/LF10/) Das System *soll* Interessenten und Kunden die Möglichkeit bieten, sich über Seminare und Veranstaltungen zu informieren, Veranstaltungen zu buchen und einen Seminarkatalog anzufordern.

/F11/ Wenn ein Kunde oder eine Firma sich von einer bereits gebuchten Veranstaltung mehr als X Wochen vor der Veranstaltung abmeldet, dann *muss* das System Stornogebühren in Höhe von Y Euro berechnen oder nach einem Ersatzteilnehmer fragen.

/F12/ Wenn ein Kunde oder eine Firma sich von einer bereits gebuchten Veranstaltung später als X Wochen vor der Veranstaltung abmeldet, dann *muss* das System Stornogebühren in Höhe der Veranstaltungsgebühr berechnen oder nach einem Ersatzteilnehmer fragen.

/F20/ (/LF20/) Das System *muss* dem Kundensachbearbeiter die Möglichkeit bieten, neue Kunden/Firmen zu erfassen und vorhandene Kunden-/Firmendaten zu aktualisieren und Kunden/Firmen zu löschen.

/F30/ (/LF30/) Das System *muss* dem Kundensachbearbeiter die Möglichkeit bieten, Seminare und Veranstaltungen zu selektieren, Veranstaltungen für Interessenten und Kunden zu buchen und zu stornieren sowie für angeforderte Seminarkataloge Versandpapiere zu erstellen.

/F40/ (/LF40/) Das System *muss* dem Seminarsachbearbeiter die Möglichkeit bieten, neue Dozenten zu erfassen, vorhandene Dozentendaten zu aktualisieren, Dozenten zu löschen und Dozenten Seminaren und Veranstaltungen zuzuordnen.

III 7 Fallstudie: SemOrg – Die Spezifikation

/F50/ (/LF50/) Das System *muss* dem Seminarsachbearbeiter die Möglichkeit bieten, neue Seminare und Veranstaltungen zu erfassen, vorhandene zu modifizieren oder zu löschen und Veranstaltungen Seminaren zuzuordnen.

/F60/ (/LF60/) Das System *soll* dem Seminarsachbearbeiter die Möglichkeit bieten, für alle Veranstaltungen Hotels auszuwählen und Räume zu reservieren.

/F61/ Das System *muss* dem Seminarsachbearbeiter und dem Kundensachbearbeiter die Möglichkeit bieten, Veranstaltungen zu stornieren, die angemeldeten Teilnehmer zu informieren, ihnen alternative Veranstaltungen anzubieten oder bei einem Dozentenausfall nach alternativen Dozenten zu suchen und dem Seminarsachbearbeiter vorzuschlagen.

/F70/ (/LF70/) Das System *muss* folgende Kundendaten (maximal 50.000) permanent speichern: Kunden-Nr., Name, Adresse, Kommunikationsdaten, Geburtsdatum, Funktion, Umsatz, Kurzmitteilung, Notizen, Info-Material, Kunde seit.

/F80/ (/LF70/) Das System *muss* folgende Firmendaten (maximal 10.000) permanent speichern, wenn ein Kunde zu einer Firma gehört: Firmenkurzname, Firmenname, Adresse, Kommunikationsdaten, Ansprechpartner, Abteilung, Geburtsdatum, Funktion des Ansprechpartners, Kurzmitteilung, Notizen, Umsatz, Kunde seit.

/F90/ (/LF70/) Das System *muss* folgende Seminartypdaten (maximal 10.000) permanent speichern: Seminarkurztitel, Seminartitel, Zielsetzung, Methodik, Inhaltsübersicht, Tagesablauf, Dauer, Unterlagen, Zielgruppe, Voraussetzungen, Gebühr ohne MWST, max. Teilnehmerzahl, min. Teilnehmerzahl.

/F100/ (/LF70/) Das System *muss* folgende Veranstaltungsdaten (maximal 100.000) permanent speichern: Veranstaltungs-Nr., Dauer (in Tagen), Vom, Bis, Tagesraster-Anfang, Tagesraster-Ende, Anfang erster Tag, Ende letzter Tag, Veranstaltungsort (Hotel/Firma, Adresse, Raum), Kooperationspartner, Öffentlich (Ja/Nein), Netto-Preis, Stornogebühr, min. Teilnehmerzahl, max. Teilnehmerzahl, Teilnehmer aktuell, Durchgeführt (Ja/Nein).

/F110/ (/LF70/) Das System *muss* folgende Dozentendaten (maximal 5.000) permanent speichern: Dozenten-Nr., Name, Adresse, Kommunikationsdaten, Geburtsdatum, Biografie, Honorar pro Tag, Kurzmitteilung, Notizen, Dozent seit.

/F120/ Wenn ein Kunde oder eine Firma eine Seminarveranstaltung bucht, dann *muss* das System folgende Buchungsdaten (maximal 500.000) permanent speichern: Angemeldet am, Bestätigung am, Rechnung am, Abgemeldet am, Mitteilung am.

/F130/ Das System *muss* fähig sein, dem Buchhaltungssystem Rechnungsdatensätze mindestens einmal am Tag zur Verfügung zu stellen.

/F140/ Wenn ein Dozent eine Seminarveranstaltung leitet, dann *muss* das System dies speichern.

/F150/ Wenn ein Kunde oder eine Firma im Zahlungsverzug ist, dann *muss* das System folgende Daten dazu speichern: Datum der Rechnung, die noch nicht bezahlt ist, sowie Betrag der Rechnung.

/F160/ Minimal 3 Tage vor einer Veranstaltung *muss* das System dem Kundensachbearbeiter und dem betreffenden Dozenten die Möglichkeit bieten, eine Teilnehmerliste für die Veranstaltung mit folgenden Daten zu erstellen: Seminartitel, Datum von, Datum bis, Veranstaltungsort, Dozent(en) Pro Teilnehmer: Name, Vorname, Firma, Ort.

/F170/ Nach dem Ende einer Veranstaltung *muss* das System dem Kundensachbearbeiter und dem betreffenden Dozenten die Möglichkeit bieten, eine Teilnehmerurkunde für jeden Veranstaltungsteilnehmer mit folgenden Daten zu erstellen: Anrede, Titel, Vorname, Nachname, von Datum, bis Datum, Seminartitel, Veranstaltungsort, Inhaltsübersicht, Veranstaltungsleiter.

/F180/ Das System *muss* dem Dozenten und dem Veranstaltungsbetreuer die Möglichkeit bieten, für eine Veranstaltung Beurteilungsbögen auszudrucken.

/F190/ Wurde eine Veranstaltung durchgeführt, dann *muss* das System fähig sein, eine Honorarmitteilung an die Buchhaltung zu senden.

5 Qualitätsanforderungen

Die Qualitätsanforderungen sind in der Tab. 7.0-2 aufgeführt.

/QF10/ Beim Zugriff über das Internet muss das System eine sichere Übertragung (z. B. https) ermöglichen.

/QF20/ Das System muss die Rollen entsprechend der Tab. 7.0-3 unterscheiden und die dazugehörigen Zugriffsrechte sicherstellen können.

/QF30/ Wenn ein Benutzer SemOrg nutzen will, dann muss das System eine Autorisierung vom Benutzer verlangen.

/QB10/ Die Grundsätze der DIN EN ISO 9241-110 von 2006 mit dem Titel »Ergonomie der Mensch-System-Interaktion – Teil 110: Grundsätze der Dialoggestaltung« sind einzuhalten.

/QE10/ (/LQE10/) Alle Reaktionszeiten auf Benutzeraktionen müssen unter 5 Sekunden liegen.

6 Abnahmekriterien

/A10/ Gültiges Abnahmeszenario: Ein Seminar neu erfassen, eine Veranstaltung neu erfassen, die Veranstaltung dem Seminar zuordnen, einen Dozenten erfassen und dem Seminar und der Veranstaltung zuordnen.

/A20/ TBD usw.

III 7 Fallstudie: SemOrg – Die Spezifikation

Tab. 7.0-2: Qualitätsanforderungen an SemOrg.

Systemqualität	sehr gut	gut	normal	nicht relevant
Funktionalität				
Angemessenheit		X		
Genauigkeit			X	
Interoperabilität			X	
Sicherheit		X		
Konformität			X	
Zuverlässigkeit				
Reife		X		
Fehlertoleranz			X	
Wiederherstellbarkeit		X		
Konformität			X	
Benutzbarkeit				
Verständlichkeit	X			
Erlernbarkeit			X	
Bedienbarkeit		X		
Attraktivität		X		
Konformität			X	
Effizienz				
Zeitverhalten	X			
Verbrauchsverhalten			X	
Konformität			X	
Wartbarkeit				
Analysierbarkeit	X			
Änderbarkeit			X	
Stabilität	X			
Testbarkeit			X	
Konformität			X	
Portabilität				
Anpassbarkeit	X			
Installierbarkeit			X	
Koexistenz	X			
Austauschbarkeit			X	
Konformität			X	

Tab. 7.0-3: Rollen und Zugriffsrechte in SemOrg.

Rolle	Rechte
Kundensachbearbeiter	/F11/, /F12/, /F20/, /F30/, /F61/, /F70/, /F80/, /F90/, /F100/, /F120/, /F130/, /F150/, /F160/, /F170/
Seminarsachbearbeiter	/F40/, /F50/, /F60/, /F61/, /F90/, /F100/, /F110/, /F140/, /F170/
Dozenten	/F110/, /F140/, /F160/, /F170/, /F180/
Veranstaltungsbetreuer	/F160/, /F170/, /F180/, /F190/
Interessent, Kunde	/F10/

7 Subsystemstruktur (optional)

/K10/ Das System kann in 3 Stufen erstellt werden. Die Funktionalität muss entsprechend der Tab. 7.0-4 realisiert werden. In der Stufe 1 wird die Kernfunktionalität erstellt, in der Stufe 2 wünschenswerte Erweiterungen für den Veranstaltungsbetreuer. In Stufe 3 wird das System für Interessenten und Kunden im Internet geöffnet.

Funktionalität	SemOrg V1.0 (Kern)	SemOrg V2.0	SemOrg V3.0
/F10/ Internet			X
/F11/	X		
/F12/	X		
/F20/	X		
/F30/	X		
/F40/	X		
/F50/	X		
/F60/ Hotels & Räume		X	
/F61/	X		
/F70/	X		
/F80/	X		
/F90/	X		
/F100/	X		
/F110/	X		
/F120/	X		
/F130/	X		
/F140/	X		
/F150/	X		
/F160/ Teilnehmerliste		X	
/F170/ Teilnehmerurkunde		X	
/F180/ Beurteilungsbögen		X	
/F190/	X		

Tab. 7.0-4:
Funktionalität pro
Version.

8 Fallstudie: Fensterheber – Die Spezifikation

Um die verschiedenen Techniken und Konzepte beim Einsatz für **softwareintensive Systeme** zu veranschaulichen, wird eine Fallstudie **Fensterheber** (Türsteuergerät für einen Fensterheber) durchgängig verwendet und referenziert. Das folgende **Lastenheft** und **Pflichtenheft** verwendet nur natürlichsprachliche Anforderungen zur Festlegung der Anforderungen. Der Aufbau von Lasten- und Pflichtenheft ist im Kapitel »Schablonen für Lastenheft, Pflichtenheft und Glossar«, S. 492 erklärt. Fachbegriffe werden in einem **Glossar** definiert (siehe unten).

Lastenheft Fensterheber

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Hans Echt	Produkt- manager Türsteuer- gerät [HoPa02]	in Bear- beitung	5/09	

Voreinstellungen (Kursiv dargestellt):

Priorität aus Auftraggebersicht = {hoch, *mittel*, niedrig}

Priorität aus Auftragnehmersicht = {hoch, *mittel*, niedrig}

Stabilität der Anforderung = {fest, gefestigt, *volatil*}

Kritikalität der Anforderung = {hoch, *mittel*, niedrig, keine}

Entwicklungsrisiko der Anforderung = {hoch, *mittel*, niedrig}

1 Visionen und Ziele

/LV10/ Die Komponente »Fensterheber« ist Teil eines neuen universellen Türsteuergeräts (TSG) in einem eingebetteten System. Das universelle TSG soll dabei verschiedene Funktionalitäten im Fahrzeug dem Fahrer an einer zentralen Position mit Hilfe eines einheitlichen Bedienfeldes zur Verfügung stellen.

/LV20/ Die Fensterheber-Komponente soll das stufenlose Heben und Senken der Seitenfenster des Fahrzeugs ermöglichen.

/LZ10/ Der Fensterheber ist für die Baureihen STAR 390 (Limousine, 4 Türen), STAX 390 (Cabriolet, 2 Türen) und STAL 390 (Coupe, 2 Türen) geplant.

III 8 Fallstudie: Fensterheber – Die Spezifikation

/LZ20/ Die erwarteten Stückzahlen (für alle drei Baureihen) betragen ca. 20.000 Einheiten pro Jahr.

2 Rahmenbedingungen

/LR10/ Die Fahrzeuge sollen weltweit vertrieben werden. Dazu muss der Fensterheber für die Varianten USA, Kanada, Großbritannien, Golfstaaten, Europa und Japan konfigurierbar sein.

3 Kontext und Überblick

/LK10/ Die Fensterheber-Komponente wird in das universelle Türsteuergerät integriert. Das Türsteuergerät integriert die folgenden Komponenten (/LK20/ bis /LK70/):

/LK20/ Fensterheber: Aufgabe dieser Komponente ist das komfortable Heben und Senken der Seitenfenster des Fahrzeugs.

/LK30/ Sitzeinstellung: Aufgabe dieser Komponente ist das Verstellen des Lehnenwinkels, der horizontalen Sitzposition, der Höhe des vorderen Sitzbereichs, der Höhe des hinteren Sitzbereichs und der Schalung des Sitzes.

/LK40/ Benutzermanagement: Aufgabe dieser Komponente ist das benutzerspezifische Abspeichern von Sitz- und Außenspiegelposition.

/LK50/ Türschloss: Aufgabe dieser Komponente ist das Auf- und Zuschließen des Fahrzeugs über Schlüssel, Funksender oder CAN.

/LK60/ Innenraumbeleuchtung: Aufgabe dieser Komponente ist Beleuchtung des Fahrzeuginnen als Hilfe beim Ein- und Aussteigen.

/LK70/ Außenspiegeleinstellung: Aufgabe dieser Komponente ist das Verstellen der Außenspiegel entlang einer horizontalen und einer vertikalen Achse.

/LK80/ Die schematische Darstellung der Abb. 8.0-1 zeigt die Anordnung der einzelnen Bedienelemente.

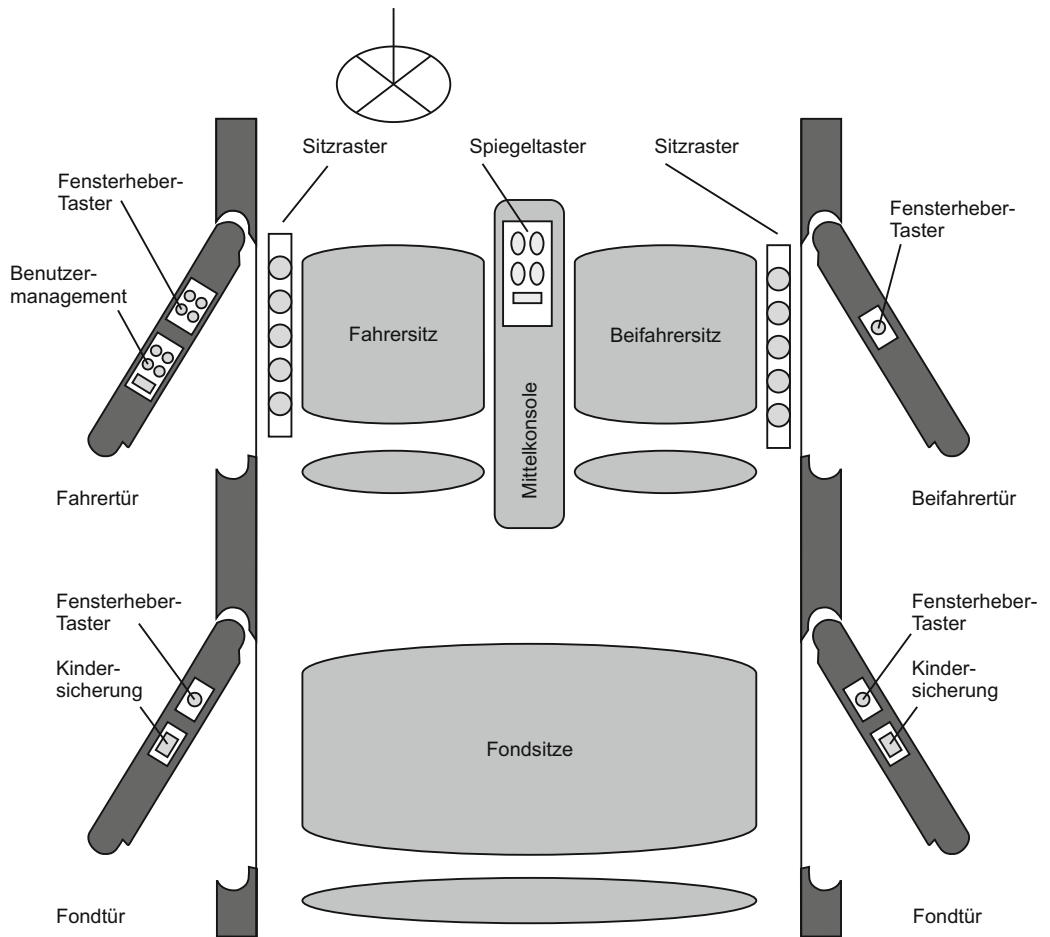
4 Funktionale Anforderungen

/LF10/ Das System Fensterheber *muss* das komfortable Heben und Senken der Seitenfenster des Fahrzeugs erlauben.

/LF20/ Das System *muss* dem Benutzer die Möglichkeit bieten, das Bewegen der Seitenfenster über die vom Türsteuergerät zur Verfügung gestellten Taster zu ermöglichen. Solange ein Taster gedrückt ist, bewegt sich die entsprechende Scheibe in die gewünschte Richtung.

/LF30/ Das System *muss* beim Schließen einer Scheibe immer prüfen, ob ein Hindernis die Scheibenbewegung stört. Wenn ja, wird der Schließvorgang unterbrochen und die Scheibe ganz geöffnet (Einklemmschutz).

/LF40/ Je nach Fahrzeugtyp und Position des TSG im Fahrzeug (Fahrer-, Beifahrerseite) *muss* das System unterschiedliche Eingangsarten beachten.



5 Qualitätsanforderungen

/LQB10/ Das Bewegen der Seitenfenster erfolgt über Taster. Solange ein Taster gedrückt ist, bewegt sich die entsprechende Scheibe in die gewünschte Richtung.

/LQB30/ Wird der Taster ganz durchgedrückt, bewegt sich die Scheibe selbstständig bis in die untere bzw. obere Endposition.

/LQB40/ In der Fahrertür sind Taster für alle vier Seitenfenster untergebracht, in den übrigen Türen finden sich nur Taster für das jeweilige Seitenfenster.

/LQB50/ Ist die Kindersicherung aktiviert, sind die Taster in der Fondtür deaktiviert.

Abb. 8.0-1:
Anordnung der
einzelnen
Bedienelemente.

III 8 Fallstudie: Fensterheber – Die Spezifikation

Tab. 8.0-1: Qualitätsanforderungen an den Fensterheber.

Systemqualität	sehr gut	gut	normal	nicht relevant
Funktionalität		X		
Zuverlässigkeit	X			
Benutzbarkeit		X		
Effizienz			X	
Wartbarkeit				X
Portabilität		X		

Glossar Fensterheber

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Hans Echt	Produkt- manager Türsteuer- gerät	in Bear- beitung	5/09	

CAN-Bus: Steht für *Controller Area Network*. Es handelt sich dabei um ein asynchrones, serielles Bussystem, das für die Vernetzung der Steuergeräte im Fahrzeug verwendet wird.

Eingangssignale: Die Eingangssignale beschreiben die auf dem CAN-Bus für das Türsteuergerät relevanten Nachrichtensignale.

Einklemmschutz: Der Einklemmschutz verhindert des unabsichtliche Schließen des Fensters bei vorhandenem Hindernis.

Fensterheber: Fensterheber sind technische Einrichtungen, die zum Heben und Senken von Fenstern oder ähnlichem dienen.

TSG: Türsteuergerät. Das Türsteuergerät stellt verschiedene Funktionalitäten im Fahrzeug dem Fahrer an einer zentralen Position mit Hilfe eines einheitlichen Bedienfeldes zur Verfügung. Die Fensterheber sind eine Komponente des Türsteuergeräts.

Pflichtenheft Fensterheber

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Hans Echt	Technischer Entwickler Türsteuer- gerät [HoPa02]	in Bear- beitung	6/09	

Voreinstellungen (Kursiv dargestellt):

Priorität aus Auftraggebersicht = {hoch, *mittel*, niedrig}

Priorität aus Auftragnehmersicht = {hoch, *mittel*, niedrig}

Stabilität = {fest, gefestigt, *volatil*}

Kritikalität = {hoch, *mittel*, niedrig, keine}

Entwicklungsrisiko = {hoch, *mittel*, niedrig}

1 Visionen und Ziele

/V10/ (/LV10/) Die Komponente »Fensterheber« ist Teil eines neuen universellen Türsteuergeräts (TSG) in einem eingebetteten System. Das universelle TSG soll dabei verschiedene Funktionalitäten im Fahrzeug dem Fahrer an einer zentralen Position mit Hilfe eines einheitlichen Bedienfeldes zur Verfügung stellen.

/V20/ (/LV20/) Die Fensterheber-Komponente soll dabei das stufenlose Heben und Senken der Seitenfenster des Fahrzeugs ermöglichen.

/Z10/ (/LZ10/) Der Fensterheber ist für die Baureihen STAR 390 (Limousine, 4 Türen), STAX 390 (Cabriolet, 2 Türen) und STAL 390 (Coupe, 2 Türen) geplant.

/Z20/ (/LZ20/) Die erwarteten Stückzahlen (für alle drei Baureihen) betragen ca. 20.000 Einheiten pro Jahr.

2 Rahmenbedingungen

/R10/ (/LR10/) Die Fahrzeuge sollen weltweit vertrieben werden. Dazu muss die Komponente für die Varianten USA, Kanada, Großbritannien, Golfstaaten, Europa und Japan konfigurierbar sein.

3 Kontext und Überblick

/K10/ (/LK20/) Die Fensterheber sollen das stufenlose Heben und Senken der Seitenfenster des Fahrzeugs erlauben.

/K20/ Die Steuerung des Fensterhebers muss anhand der auf dem CAN-Bus angelegten Steuerbotschaften erfolgen.

4 Funktionale Anforderungen

/F10/ (/LF20/) Die Bewegung der Scheiben *muss* durch die Tasten in den Fahrzeugtüren oder durch CAN-Botschaften gesteuert werden.

/F20/ (/LF20/) Die Abb. 8.0-2 beschreibt die Reaktionen auf die unterschiedlichen Steuer-Botschaften auf dem CAN-Bus.

/F30/ (/LF20/) Eine Fensterbewegung *soll* entweder solange andauern, wie der entsprechende Stimulus anliegt, oder aber bis die Scheibe vollständig offen bzw. geschlossen ist.

/F40/ (/LF20/) Wenn ein Signal anliegt, das das Öffnen einer Scheibe zur Folge hat, dann *muss* auf dem entsprechenden Motor die Spannung -12 V angelegt werden. Liegt die Batteriespannung zum Beginn der Scheibenbewegung unterhalb von 10V, so *darf* die Scheibenbewegung nicht durchgeführt werden. Statt dessen *muss* die CAN-Botschaft B_LOW_WIN = 1 gesendet werden. Für die Öffnen-Bewegung sind folgende Fälle zu beachten:

III 8 Fallstudie: Fensterheber – Die Spezifikation

Signal	Fahrzeugtyp	TSG	Reaktion
WIN_VL_OP WIN_VL_CL	Limousine, Coupé, Cabriolet Linkslenker, Rechtslenker	Links	Bewegen der Scheibe vorne links mittels S1.F_MOTOR1 und S1.F_MOTOR2
		Rechts	— ignorieren —
WIN_VR_OP WIN_VR_CL	Limousine, Coupé, Cabriolet Linkslenker, Rechtslenker	Links	— ignorieren —
		Rechts	Bewegen der Scheibe vorne rechts mittels S1.F_MOTOR1 und S1.F_MOTOR2
WIN_HL_OP WIN_HL_CL	Limousine Linkslenker, Rechtslenker	Links	Bewegen der Scheibe hinten links mittels S2.FF_MOTOR1 und S2.FF_MOTOR2
		Rechts	— ignorieren —
	Coupé, Cabriolet Linkslenker, Rechtslenker	Beide	— ignorieren —
WIN_HR_OP WIN_HR_CL	Limousine Linkslenker, Rechtslenker	Links	— ignorieren —
		Rechts	Bewegen der Scheibe hinten rechts mittels S2.FF_MOTOR1 und S2.FF_MOTOR2
	Coupé, Cabriolet Linkslenker, Rechtslenker	Beide	— ignorieren —
S1.FHB_VL	Limousine, Coupé, Cabriolet Linkslenker	Links	Bewegen der Scheibe vorne links mittels S1.F_MOTOR1 und S1.F_MOTOR2
		Rechts	— ignorieren —
	Limousine, Coupé, Cabriolet Rechtslenker	Links	Bewegen der Scheibe vorne links mittels S1.F_MOTOR1 und S1.F_MOTOR2
		Rechts	Signal CAN.WIN_VL_x senden
S1.FHB_VR	Limousine, Coupé, Cabriolet Linkslenker	Links	Signal CAN.WIN_VR_x senden
		Rechts	Bewegen der Scheibe vorne rechts mittels S1.F_MOTOR1 und S1.F_MOTOR2
	Limousine, Coupé, Cabriolet Rechtslenker	Links	— ignorieren —
		Rechts	Bewegen der Scheibe vorne rechts mittels S1.F_MOTOR1 und S1.F_MOTOR2
S1.FHB_HL	Limousine Linkslenker	Links	Bewegen der Scheibe hinten links mittels S2.FF_MOTOR1 und S2.FF_MOTOR2
		Rechts	— ignorieren —
	Coupé, Cabriolet Linkslenker	Beide	— ignorieren —
		Links	— ignorieren —
	Coupé, Cabriolet Rechtslenker	Beide	— ignorieren —
		Rechts	Signal CAN.WIN_HL_x senden
S1.FHB_HR	Limousine Linkslenker	Links	Signal CAN.WIN_HR_x senden
		Rechts	— ignorieren —
	Coupé, Cabriolet Linkslenker	Beide	— ignorieren —

Abb. 8.0-2:
Reaktionen auf die
unterschiedlichen
Steuer-Botschaften
auf dem CAN-Bus.

/F50/ (/LF20/) Ist die relevante Schalterstellung gleich Fenster runter man. oder ist die relevante CAN-Botschaft WIN_x_OP = 01, so *muss* die Scheibe nach unten bewegt werden. Die Bewegung endet, wenn

/F51/ (/LF20/) das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird),

/F52/ (/LF20/) oder sich die Scheibe in der unteren Position befindet (d. h. F_UNTEN bzw. FF_UNTEN),

/F53/ (/LF20/) oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet,

/F54/ (/LF20/) oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der unteren Position befindet; in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen

/F55/ (/LF20/) oder die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der unteren Position befindet; in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen.

/F60/ (/LF20/) Wurde die Schalterstellung Fenster runter auto. oder die CAN-Botschaft WIN_x_OP = 10 erkannt, so *muss* die Scheibe solange nach unten bewegt werden, bis

/F61/ (/LF20/) sich die Scheibe in der unteren Position befindet (d. h. F_UNTEN bzw. FF_UNTEN),

/F62/ (/LF20/) oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet,

/F63/ (/LF20/) oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der unteren Position befindet; in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen

/F64/ (/LF20/) oder die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der unteren Position befindet; in diesem Fall wird Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen.

/F70/ (/LF20/) Falls ein Signal anliegt, das das Schließen einer Scheibe zur Folge hat, *muss* auf dem entsprechenden Motor die Spannung +12 V angelegt werden. Liegt die Batteriespannung zum Beginn der Scheibenbewegung unterhalb von 10V, so wird die Scheibenbewegung nicht durchgeführt. Statt dessen wird die CAN-Botschaft B_LOW_WIN = 1 gesendet. Für die Schließen-Bewegung sind folgende Fälle zu beachten:

III 8 Fallstudie: Fensterheber – Die Spezifikation

/F80/ (/LF20/) Ist die relevante Schalterstellung gleich Fenster hoch man. oder ist die relevante CAN-Botschaft WIN_x_CL = 01, so *muss* die Scheibe nach oben bewegt werden. Die Bewegung endet, wenn

/F81/ (/LF20/) das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird),

/F82/ (/LF20/) oder sich die Scheibe in der oberen Position befindet (d. h. F_OBEN bzw. FF_OBEN),

/F83/ (/LF20/) oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet,

/F84/ (/LF20/) oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der oberen Position befindet; in diesem Fall wird der Einklemmschutz aktiviert (s.u.),

/F85/ (/LF20/) oder die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der oberen Position befindet; in diesem Fall wird Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen.

/F90/ (/LF20/) Wurde die Schalterstellung Fenster hoch auto. oder die CAN-Botschaft WIN_x_CL = 10 erkannt, so *muss* die Scheibe solange nach oben bewegt werden, bis

/F91/ (/LF20/) sich die Scheibe in der oberen Position befindet (d. h. F_OBEN bzw. FF_OBEN),

/F92/ (/LF20/) oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet,

/F93/ (/LF20/) oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der oberen Position befindet; in diesem Fall wird der Einklemmschutz aktiviert (siehe /F110/),

/F94/ (/LF20/) oder die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der oberen Position befindet; in diesem Fall wird Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen.

/F100/ (/LF20/) Für die automatische Bewegung ist es nicht erforderlich, dass das Signal über den ganzen Bewegungszeitraum anliegt.

/F110/ (/LF30/) Beim Schließen einer Scheibe *muss* immer geprüft werden, ob ein Hindernis die Scheibenbewegung stört. Wenn ja, wird der Schließvorgang unterbrochen und die Scheibe ganz geöffnet (Einklemmschutz).

/F120/ (/LF40/) Je nach Fahrzeugtyp und Position des TSG im Fahrzeug (Fahrer-, Beifahrerseite) *muss* das System unterschiedliche Eingänge beachten.

8 Fallstudie: Fensterheber – Die Spezifikation III

/F130/ (/LF40/) Die Eingangssignale zeigt die Tab. 8.0-2.

/F140/ (/LF30/) Einklemmschutz: Wird das Einklemmen eines Gegenstandes erkannt, so *muss* die aktuelle Scheibenbewegung nach oben sofort beendet und die Scheibe in die untere Position bewegt werden. Ein Unterbrechen der Abwärtsbewegung durch Taste oder CAN-Botschaft ist nicht möglich. Die Fehlerbehandlungsregeln für die Abwärtsbewegung gelten entsprechend.

Eingänge	Signale
Fensterhebertasten	S1.FHB_VL, S1.FHB_VR, S1.FHB_HL, S1.FHB_HR
Position vorderes Fenster	S1.F_OBEN, S1.F_BEWEG, S1.F_UNTEN
Kindersicherung und Fensterhebertaste Fondtür	S2.FKIND_SICH, S2.FFHB
Position Fond-Fenster	S2.FF_OBEN, S2.FF_BEWEG, S2.FF_UNTEN
CAN-Signale zum Bewegen der Fenster (Fahrer-TSG: nur Botschaft 0x234, Beifahrer-TSG: Botschaften 0x234 und 0x235)	CAN.WIN_VL_OP, CAN.WIN_VR_OP, CAN.WIN_HL_OP, CAN.WIN_HR_OP, CAN.WIN_VL_CL, CAN.WIN_VR_CL, CAN.WIN_HL_CL, CAN.WIN_HR_CL
Fahrzeugtyp	CAN.FCODE T0, CAN.FCODE T1

Tab. 8.0-2:
Eingangssignale.

5 Qualitätsanforderungen

Die detaillierten Qualitätsanforderungen sind in der Tab. 8.0-3 festgelegt.

/QF10/ Im Ruhezustand darf das TSG nicht mehr als 1,5 mA Strom verbrauchen.

/QF20/ Im aktiven Zustand darf der Stromverbrauch ohne aktive Aktoren 500mA nicht überschreiten. Bei aktivierten Aktoren darf der Stromverbrauch entsprechend höher sein.

6 Abnahmekriterien

Die folgenden Szenarien sind für eine Abnahme des Systems erfolgreich gemäß der Spezifikation durchzuführen:

/A 10/ Abnahmeszenario 1: Öffnen eines Fensters.

/A 20/ Abnahmeszenario 2: Schließen eines Fensters.

/A 30/ Abnahmeszenario 3: Einbringen eines Objektes in den Klemmbereich während des Schließens.

III 8 Fallstudie: Fensterheber – Die Spezifikation

Tab. 8.0-3: Qualitätsanforderungen an die Fensterheber.

Systemqualität	sehr gut	gut	normal	nicht relevant
Funktionalität				
Angemessenheit		X		
Genauigkeit			X	
Interoperabilität			X	
Sicherheit	X			
Konformität			X	
Zuverlässigkeit				
Reife			X	
Fehlertoleranz		X		
Wiederherstellbarkeit		X		
Konformität			X	
Benutzbarkeit				
Verständlichkeit	X			
Erlernbarkeit	X			
Bedienbarkeit	X			
Attraktivität			X	
Konformität			X	
Effizienz				
Zeitverhalten	X			
Verbrauchsverhalten			X	
Konformität			X	
Wartbarkeit				
Analysierbarkeit				X
Änderbarkeit				X
Stabilität				X
Testbarkeit				X
Konformität				X
Portabilität				
Anpassbarkeit	X			
Installierbarkeit	X			
Koexistenz		X		
Austauschbarkeit		X		
Konformität				X

9 Statik

Unter der Statik eines Systems wird die **stabile Struktur** eines Systems verstanden. Eine Struktur gibt die Anordnung der Teile eines Ganzen zueinander an (siehe »Prinzip der Strukturierung«, S. 34). Eine statische Struktur beschreibt dementsprechend die Anordnung der Teile eines Ganzen, die sich über die Zeit hinweg, z. B. während der Laufzeit des Softwaresystems, *nicht* ändert.

In der Softwaretechnik bestehen die Teile eines Systems aus Funktionen und Daten, die in stabilen Strukturen angeordnet sind.

Die Funktionalität wird in der Softwaretechnik durch einzelne Funktionen (Funktionen, Prozeduren, Methoden) oder durch semantisch zusammengehörige Funktionsgruppen (Klassen, Schnittstellen, Komponenten) beschrieben:

■ »Funktionalität«, S. 127

Der Zusammenhang der Funktionen oder Funktionsgruppen wird mit Hilfe von statischen Relationen dargestellt:

■ »Funktions-Strukturen«, S. 142

Daten können in Form von Datenelementen spezifiziert werden:

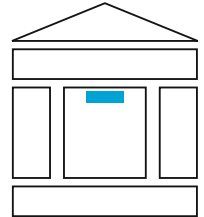
■ »Daten«, S. 181

Der Zusammenhang zwischen Datenelementen kann mit Hilfe von statischen Relationen beschrieben werden:

■ »Daten-Strukturen«, S. 190

Für die Ermittlung und Überprüfung von Attributen gibt es konstruktive und analytische Schritte:

■ »Box: Attribute – Methode und Checkliste«, S. 187



9.1 Funktionalität

Jedes Softwaresystem stellt durch seine Systemkomponenten Funktionalität für andere Systemkomponenten und/oder den Benutzer zur Verfügung. Der Begriff Funktionalität steht für alle Funktionen eines Systems oder einer Systemkomponente.

Eine **Funktion** beschreibt eine Tätigkeit oder eine klar umrissene Aufgabe innerhalb eines größeren Zusammenhangs.

Definition

In der Softwaretechnik ermittelt eine Funktion aus Eingabedaten Ausgabedaten und/oder bewirkt eine Veränderung des Inhalts oder der Struktur von Informationen. Bei Funktionen kann es sich um Benutzerfunktionen handeln (für den Benutzer sichtbar und nutzbar)

III 9 Statik

oder um Systemfunktionen für andere Systemkomponenten, die entsprechend den Rechte- bzw. Zugriffskonzepten nutzbar und/oder sichtbar sind.

i Funktionen können einzeln oder zu fachlichen Gruppen zusammengefasst beschrieben werden:

- »Einzelne Funktionen«, S. 128

- »Zusammenfassung von Funktionen«, S. 131

Für die Ermittlung und Überprüfung von Klassen gibt es konstruktive und analytische Schritte:

- »Box: Klassen – Methode und Checkliste«, S. 137

9.1.1 Einzelne Funktionen

Notation Einzelne Funktionen können informal, semiformal oder formal sowie textuell oder grafisch beschrieben werden. Dabei kann der Beschreibung eine Syntax, eine Semantik und/oder Benennungskonventionen zugrunde liegen.

Das Abstraktionsniveau kann unterschiedlich sein.

Beispiel 1a: Natürlichsprachliche Formulierung ohne festgelegte Syntax und Benennungskonventionen (hohes Abstraktionsniveau):
SemOrg »/LF41/ Neue Dozenten müssen erfasst werden können.«

Beispiel 1b Natürlichsprachliche Formulierung mit Benennungskonvention (Funktion soll wie folgt beschrieben sein: Verb und Objekt oder Substantiv und Verb):
»/LF41/ erfasse neue Dozenten« oder alternativ
»/LF41/ Neue Dozenten erfassen«

Beispiel 1c Natürlichsprachliche Formulierung entsprechend einer vorgegebenen Schablone (Abb. 9.1-1):
»/LF41/ Das System muss dem Seminarsachbearbeiter die Möglichkeit bieten, neue Dozenten zu erfassen.«

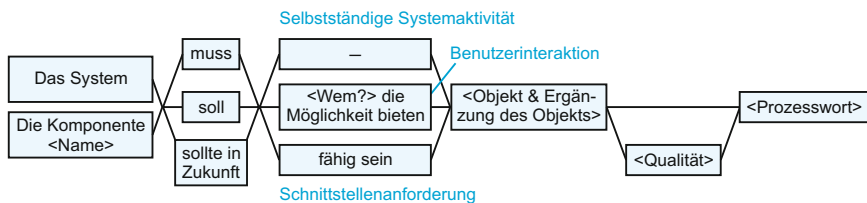


Abb. 9.1-1: Aufbau einer Anforderungsschablone ohne Bedingungen. Oft werden Funktionen auch als beschriftetes Rechteck (»Kästchen«) dargestellt.

Beispiel 1d Die Abb. 9.1-2 zeigt eine Funktion in grafischer Darstellung.

Im Rahmen der Methode *Structured Analysis* (SA), die früher eine große Bedeutung hatte, gibt es Datenflussdiagramme, die Funktionen, Speicher, Schnittstellen sowie Datenflüsse dazwischen enthalten. Die Funktionen werden als Kreise dargestellt ([Marc78], [Your89]).

erfasse neue
Dozenten
/LF41/

Abb. 9.1-2: Beispiel für die grafische Darstellung einer Funktion.

In prozeduralen Programmiersprachen werden Funktionen als Prozeduren oder Funktionen entsprechend der Sprachsyntax mit oder ohne Parameter beschrieben. In der objektorientierten Programmierung werden Funktionen als Methoden spezifiziert:

Beispiel 1e

»void erfasseNeueDozenten(String nachname) {}; // LF41«

Prüfen Sie, ob ein aussagefähiger Funktionsname vorliegt (siehe »Prinzip der Verbalisierung«, S. 46). Ein **Funktionsname** soll

QS-Checkliste
Funktion

- ein starkes Verb enthalten,
- in der Regel mit einem Verb beginnen, evtl. gefolgt von einem Substantiv, z.B. drucke, aendere, zeigeFigur, leseAdresse, verschiebeRechteck,
- beschreiben, was die Funktion »tut«. Mit einem Kleinbuchstaben beginnen.



In Java gelten zusätzlich folgende Konventionen für Methodenbezeichner:

Ein **Methodenname** heißt

- getAttributname, wenn ein Attributwert eines Objekts gelesen wird,
- setAttributname, wenn ein Attributwert eines Objekts gespeichert wird,
- isAttributname, wenn das Ergebnis nur wahr (*true*) oder falsch (*false*) sein kann, z.B. isVerheiratet, isVerschlossen.

Für Funktionen ist eine funktionale Bindung anzustreben (siehe »Prinzip der Bindung und Kopplung«, S. 37), bei der alle Elemente an der Verwirklichung einer einzigen, abgeschlossenen Aufgabe beteiligt sind.

Bindung

Eine **funktionale Bindung** liegt vor, wenn alle Elemente an der Verwirklichung einer einzigen, abgeschlossenen Funktion beteiligt sind.

Definition

Prüfen Sie die **funktionale Bindung** einer Funktion anhand folgender Kriterien:

QS-Checkliste
Funktionale
Bindung

- Alle Elemente tragen dazu bei, ein einzelnes, spezifisches Ziel zu erreichen.
- Es gibt keine überflüssigen Elemente.
- Die Aufgabe lässt sich mit genau einem Verb und genau einem Objekt vollständig beschreiben.



III 9 Statik

- Leichter Austausch gegen eine andere Funktion, die denselben Zweck erfüllt.
- Hohe Kontextunabhängigkeit, d. h. einfache Beziehungen zur Umwelt.
- Die Funktion ist »kurz« (siehe auch [Fowl05, S. 76 ff.]).

Beispiel Die folgende Java-Methode berechnet die Quadratwurzel:

Java

```
public double berechneQuadratwurzel(double x)
{
    double y0, y1;
    int i;
    assert (x >= 0.0);
    y1 = 1.0;
    i = 1;
    for (;;)
    {
        y0 = y1;
        y1 = 0.5 * (y0 + x / y0);
        i++; assert (i <= 50);
        if (Math.abs(y1 - y0) < 1e-6) return y1;
    }
}
```

Eine Analyse dieser Methode zeigt, dass alle Anweisungen nur dazu beitragen, aus dem Eingabewert die Quadratwurzel zu ermitteln. Sie ist daher funktional gebunden. Würde diese Methode jetzt so erweitert, dass sie noch eine Statistik über die Häufigkeit gleicher Quadratwurzelberechnungen erstellt, dann wäre sie *nicht* mehr funktional gebunden, da sie noch eine weitere Funktion ausführen würde.

Eine funktionale Bindung führt zu einer wesentlichen Verfestigung der internen Funktionsstruktur. Eine funktionale Bindung bringt folgende Vorteile mit sich:

- + Der Umfang der Funktion ist überschaubar und besteht aus relativ wenigen Zeilen Code.
- + Hohe Kontextunabhängigkeit der Funktion (die Bindungen befinden sich innerhalb einer Funktion, nicht zwischen Funktionen).
- + Geringe Fehleranfälligkeit bei Änderungen.
- + Hoher Grad der Wiederverwendbarkeit, da weniger spezialisiert.
- + Leichte Erweiterbarkeit und Wartbarkeit, da sich Änderungen auf isolierte, kleine Teile beschränken.

Der Bindungsgrad einer Funktion lässt sich *nicht* automatisch ermitteln. Er kann durch manuelle Prüfmethode bestimmt werden.

Zur Klassifikation

- Funktionen werden in der Regel textuell beschrieben.
- Der Grad der Formalität reicht von informal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

9.1.2 Zusammenfassung von Funktionen

Oft werden Funktionen von vornherein oder erst später zu Gruppen zusammengefasst. In der objektorientierten Softwareentwicklung werden mehrere Funktionen nach fachlichen Gesichtspunkten zu Klassen, Schnittstellen (*interfaces*) oder Komponenten zusammengefasst.

Klassen

Eine Klasse ist allgemein eine Gruppe von Dingen, Lebewesen oder Begriffen mit gemeinsamen Merkmalen.

In der objektorientierten Welt spezifiziert eine **Klasse** die Gemeinsamkeiten einer Menge von Objekten mit denselben Eigenschaften (Attributen), demselben Verhalten (Operationen, Methoden) und denselben Beziehungen. Eine Klasse besitzt einen Mechanismus, um Objekte zu erzeugen (*object factory*). Jedes erzeugte Objekt gehört zu genau einer Klasse.

Definition

Unter dem Gesichtspunkt der Funktionalität stellt eine Klasse seiner Umwelt Operationen bzw. Methoden als Dienstleistung zur Verfügung. Eine Operation wandelt dabei Eingabedaten in Ausgabedaten um und/oder verändert die internen Attribute (wenn Attribute auf Klassenebene vorhanden sind).

Eine Klasse realisiert eine Systemkomponente. Mehrere Klassen zusammen können zu einem Paket gruppiert werden und realisieren dann ein Subsystem (siehe »Pakete«, S. 145).

Klassen werden grafisch – heute meist in der UML-Notation – oder textuell in der Syntax der jeweiligen objektorientierten Programmiersprache dargestellt. Die UML-Notation befindet sich auf einem höheren Abstraktionsniveau als die Darstellung in einer Programmiersprache.

Notationen

In der UML-Notation wird eine Klasse durch ein dreigeteiltes Rechteck dargestellt (Abb. 9.1-3). Im oberen Teil steht zentriert und fett der Klassenname. Im mittleren Teil sind – jeweils linksbündig – die Attribute angegeben, im unteren Teil die Operationen. Dem Operationsnamen folgt ein Klammerpaar (). Sind Attribute und Operationen noch nicht festgelegt oder im Moment nicht relevant, können sie weggelassen werden. Attribute und Operationen können näher spezifiziert werden.



Der Klassenname ist stets ein Substantiv im Singular. Zusätzlich kann ein Adjektiv angegeben werden. Er beschreibt also ein einzelnes Objekt der Klasse.

Benennung

III 9 Statik

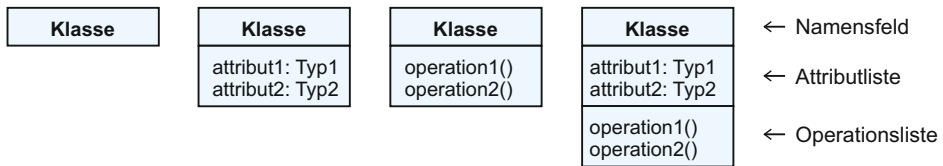
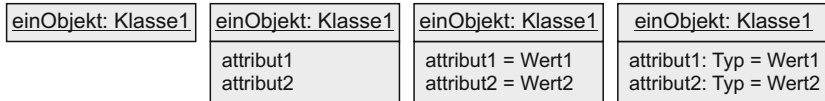


Abb. 9.1-3: Klassen beschreiben Funktionalität auf der Typebene (siehe »Prinzip der Abstraktion«, S. 26). Für die Exemplarebene gibt es **Objekte**. Die UML-Notation für Objekte zeigt die Abb. 9.1-4.

Abb. 9.1-4:
Mögliche UML-
Objektnotationen.



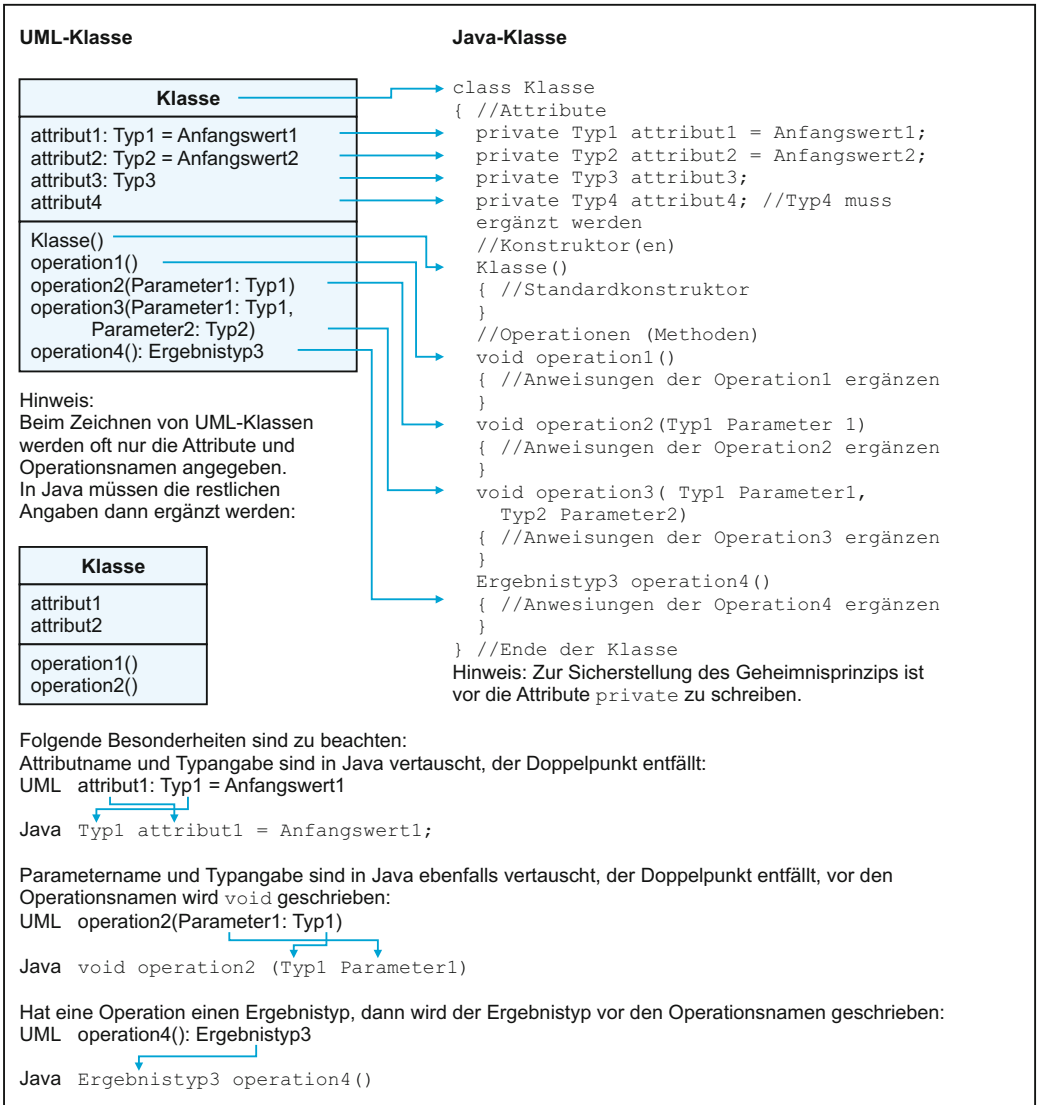
Wie Sie systematisch von einer in der UML modellierten Klasse zu einem Java-Klassengerüst gelangen, zeigt die Abb. 9.1-5.

Hinweis

Bei Klassen, die keine klasseneigenen Attribute besitzen, liefern aufgerufene Methoden bei gleichen Eingabedaten immer die gleichen Ergebnisse, wenn sie nicht auf andere Methoden oder gespeicherte Daten zugreifen. Besitzen Klassen dagegen klasseneigene Attribute (Objekt- oder Klassenattribute), dann hängt das Ergebnis beim Aufruf von Methoden mit identischen Eingabedaten u. U. vom Zustand der klasseneigenen Attribute ab. Die Klasse kann sich in einem solchen Fall wie ein Zustandsautomat verhalten (siehe »Zustandsautomaten«, S. 269).

Schnittstellen

In der objektorientierten Software-Entwicklung gibt es neben Klassen noch **Schnittstellen** (*interfaces*). Der Begriff wird *nicht* einheitlich verwendet. In der Regel definieren Schnittstellen Dienstleistungen für aufrufende Klassen, ohne etwas über die Implementierung der Dienstleistungen festzulegen. Es werden **funktionale Abstraktionen** in Form von Operationssignaturen bereitgestellt, die das »Was«, aber nicht das »Wie« festlegen. Eine Schnittstelle besteht also im Allgemeinen nur aus Operationssignaturen, d. h. sie besitzt *keine* Operationsrümpfe, *keine* Attribute oder Assoziationen (siehe »Assoziation«, S. 158). Schnittstellen können jedoch in Vererbungsstrukturen verwendet werden. Eine Schnittstelle ist äquivalent zu einer Klasse, die *keine* Attribute und ausschließlich **abstrakte Operationen** besitzt. Für manche Situationen ist es auch nützlich, öffentliche Attribute in einer Schnittstelle bereitzustellen.



In der UML-Notation gibt es zwei alternative Notationen für eine Schnittstelle:

- Eine Schnittstelle kann wie eine Klasse, allerdings mit dem Zusatz `<<interface>>` vor dem Schnittstellennamen, dargestellt werden. Implementiert eine Klasse eine Schnittstelle, dann wird zwischen der Klasse und der Schnittstelle ein gestrichelter Vererbungspfeil gezeichnet (Abb. 9.1-6, linke Seite).

Abb. 9.1-5: Von UML-Klassen zu Java-Klassen.

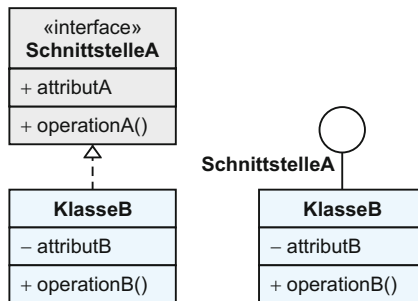


III 9 Statik

- Ebenso kann eine Schnittstelle als *nicht* ausgefüllter Kreis, beschriftet mit dem Schnittstellennamen, gezeichnet werden (in der UML als *ball* bezeichnet). Die implementierende Klasse wird durch eine Linie mit dem Kreissymbol verbunden (Abb. 9.1-6, rechte Seite).

Ein Attribut in einer Schnittstelle bedeutet, dass die Klasse, die diese Schnittstelle realisiert, sich so verhalten muss, als ob sie das Attribut selbst besitzt.

Abb. 9.1-6:
Alternative UML-
Notationen für
eine Schnittstelle.



Java

Java-Schnittstellen erlauben die Deklaration abstrakter Operationen und öffentlicher konstanter Klassenattribute. Die Implementierung der abstrakten Operationen erfolgt durch Java-Klassen.

Dabei ist es möglich, dass verschiedene Java-Klassen dieselbe Schnittstelle auf unterschiedliche Weise implementieren. Für die aufrufende Klasse ergibt sich dadurch keine Änderung, da die Schnittstelle unverändert bleibt.



Benutzt eine Klasse eine Schnittstelle, dann wird dies in der UML durch einen gestrichelten Pfeil mit der Beschriftung <<use>> angegeben (Abb. 9.1-7 oben). Wird die Schnittstelle durch einen Kreis dargestellt, dann wird durch ein Halbkreissymbol (in der UML als *socket* bezeichnet) angedeutet, dass die damit verbundene Klasse die angegebene Schnittstelle benötigt (Abb. 9.1-7 unten). Durch die Kombination von Kreis- und Halbkreissymbol wird verdeutlicht, wie Schnittstellenbereitsteller und Schnittstellenbenutzer ineinandergreifen.

Wegen ihres Aussehens wird diese Darstellung umgangssprachlich auch als *Lollipop*-Darstellung bezeichnet.

Ein **Schnittstellename** soll folgenden Konventionen gehorchen:

- Es gelten die Kriterien, die bei der Klasse (siehe oben) aufgeführt sind.
- Es wird oft die Endung *-able* (im Englischen) verwendet, z.B. *Cloneable*, *Serializable*.
- Eine andere Variante ist, das *I* als Prefix zu verwenden (z.B. *IDialog*) – so wird es beispielsweise in der Eclipse-Programmierungsumgebung gehandhabt.

Lollipop =
Lutscher

QS-Checkliste
Schnittstelle



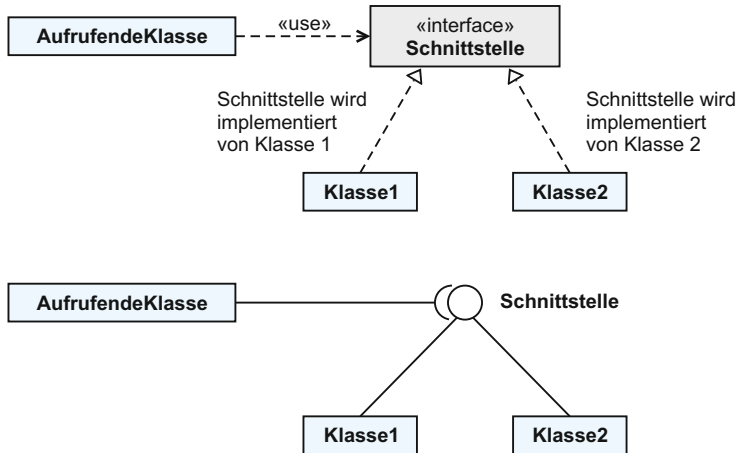


Abb. 9.1-7: Das Schnittstellenkonzept.

Komponenten

Eine **Komponente** (*component*) ist ein binärer Softwarebaustein, der Dritten Funktionalität über Schnittstellen zur Verfügung stellt und unabhängig ausgeliefert werden kann. Eine Komponente benötigt eine Komponentenplattform und besitzt die Fähigkeit zur Selbstbeschreibung. Sie weist nur explizite Kontextabhängigkeiten auf und verfügt über die Fähigkeit zur Anpassung, um eine möglichst hohe Kompositionsfähigkeit zu erreichen.

Definition

Eine Komponente ist also ein Softwarebaustein, der über klar definierte Schnittstellen Verhalten (Funktionalität) bereitstellt. Außer den Schnittstellen, die eine Komponente realisiert und zur Verfügung stellt, können diejenigen Schnittstellen spezifiziert werden, die sie selbst benötigt. Das Innenleben der Komponente, d.h. die technische Realisierung, bleibt nach außen verborgen, mit anderen Worten: Die Komponente realisiert das Geheimnisprinzip. Eine Komponente kann als eine spezifische Ausprägung eines **Moduls** angesehen werden (siehe »Prinzip der Modularisierung«, S. 40).

Die komponentenbasierte Entwicklung basiert darauf, dass Schnittstellen von ausgelieferten Komponenten nicht mehr modifiziert werden, während die interne Realisierung beliebig geändert werden darf. Es gibt verschiedene Komponentenmodelle: JavaBeans, EJBs (*Enterprise JavaBeans*), CORBA Components, COM (*Components Object Model*) und .NET components.

Der Vorteil von Komponenten ist, dass sie nicht nur zur Entwurfszeit, sondern auch zur Implementierungszeit gegen Komponenten mit äquivalenter Funktionalität und kompatiblen Schnittstellen ausgetauscht werden können. Ebenso können in ein laufendes System neue Komponenten leicht eingefügt werden.

Austauschbarkeit

III 9 Statik

Komponenten in der UML

In der UML wird der Komponentenbegriff weiter gefasst:

Definition

»A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided interfaces« [OMG09a, S. 146].



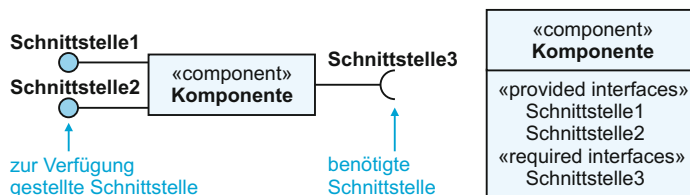
Die Komponente ist in der UML eine Spezialisierung der Klasse und besitzt somit alle Eigenschaften einer Klasse. Sie wird durch das Klassensymbol (Rechteck) mit dem Stereotyp «component» dargestellt. Zusätzlich oder alternativ kann in der rechten oberen Ecke ein Komponenten-Piktogramm eingetragen werden (Abb. 9.1-8).

Abb. 9.1-8:
Notation einer
Komponente.



Aus Sicht des Softwareentwicklers, der eine Komponente benutzt, müssen deren Schnittstellen spezifiziert werden. Man spricht hier auch von der externen oder Black-Box-Sicht. Dafür bietet die UML mehrere Notationen an. Bei der »Lollipop-Notation« werden zur Verfügung gestellte Schnittstellen durch einen kleinen Kreis dargestellt, benötigte Schnittstellen durch einen Kreisbogen (siehe oben). Alternativ kann eine Notation mit den Schlüsselwörtern «provided interfaces» und «required interfaces» verwendet werden (Abb. 9.1-9).

Abb. 9.1-9:
Notation für die
Schnittstelle einer
Komponente.



Es ist auch möglich, dass die zur Verfügung gestellten und benutzten Schnittstellen in der Klassennotation dargestellt werden, d. h. durch ein Rechteck, in das alle Operationen und ggf. Attribute eingetragen werden. Von der Komponente führen gestrichelte Pfeile auf die Schnittstellen. Die transparente Dreiecks-Pfeilspitze bedeutet, dass die Komponente diese Schnittstelle anderen Softwarebausteinen zur Verfügung stellt, d. h., die Komponente realisiert diese

Schnittstellen. Die offene Pfeilspitze mit dem Schlüsselwort «use» bedeutet, dass die Komponente diese Schnittstelle selbst benötigt (Abb. 9.1-10).

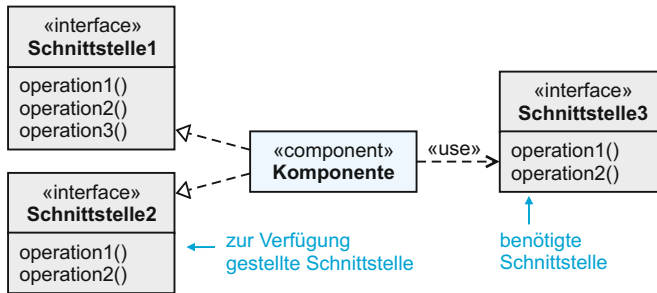


Abb. 9.1-10:
Alternative
Notation für die
Schnittstellen
einer Komponente.

Um die Schnittstellen einer Komponente oder die Komponente selbst genauer zu beschreiben, kann ein Protokollzustandsautomat verwendet werden (siehe »Verhaltens- vs. Protokollzustandsautomaten«, S. 287). Er gibt vor, in welcher Reihenfolge Operationen aufgerufen werden dürfen.

Zustands-
automat

Durch die zur Verfügung gestellten und benutzten Schnittstellen sind die Komponenten miteinander verbunden. Diese Abhängigkeiten werden im **Komponentendiagramm** (*component diagram*) dokumentiert (siehe »Weitere Strukturen«, S. 177).

Komponenten-
diagramm

Zur Klassifikation

- Klassen, Schnittstellen und Komponenten werden sowohl textuell als auch grafisch (mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von informal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

9.1.3 Box: Klassen – Methode und Checkliste

Zum Identifizieren und Überprüfen von Klassen – insbesondere um sie in ein UML-Klassendiagramm einzutragen – helfen die Boxen »Checkliste Klasse«. Auf einige Aspekte zu den Boxen wird im Folgenden eingegangen.

Konstruktive Schritte

Liegt bereits eine **Anforderungsspezifikation** (z.B. Lasten- oder Pflichtenheft) vor, dann können aus den dort angegebenen Angaben Klassen identifiziert werden.

1 Anforderungs-
spezifikation

Box 1a:
Checkliste
Klassen

Ergebnisse

■ **Klassendiagramm**

Tragen Sie jede Klasse – entweder nur mit Namen oder mit wenigen wichtigen Attributen/Operationen – in das Klassendiagramm ein.

■ **Kurzbeschreibung der Klassen**

Erstellen Sie für jede Klasse, deren Name nicht selbsterklärend ist, eine Kurzbeschreibung von 25 oder weniger Wörtern.

Konstruktive Schritte

1 Welche Klassen lassen sich aus der Anforderungsspezifikation (Lasten-/Pflichtenheft) identifizieren?

- Die Anforderungsspezifikation insbesondere bezogen auf Funktions- und Datenbeschreibungen durchsehen. Funktionen und Daten, die semantisch zusammengehören, daraufhin prüfen, ob sie zu einer Klasse zusammengefasst werden können.

2 Welche Klassen lassen sich mittels Dokumentanalyse identifizieren (Bottom-up)?

- Aus Formularen und Listen Attribute entnehmen und zu Klassen zusammenfassen.
- Sanierung von Software-Altsystemen: Arbeitsabläufe anhand von Benutzerhandbüchern, Bildschirmmasken, Dateibeschreibungen ermitteln. Anhand des laufenden Systems die Funktionen ausführen. Klassen mit Hilfe der Bildschirmmasken ermitteln.
- Bei technischen Systemen bieten sich die realen Objekte als Ausgangsbasis an, z. B. Lagermodul.

3 Welche Klassen lassen sich aus der Beschreibung der Use Cases identifizieren (Top-down)?

- Beschreibung nach Klassen durchsuchen. Oft sind Substantive potenzielle Klassen.
- Potenzielle Klassen auf Attribute überprüfen.
- Akteure, über die man sich etwas »merken« muss, sind potenzielle Klassen.

4 Zu welchen Kategorien gehören die Klassen?

- a** Konkrete Objekte bzw. Dinge, z. B. PKW.
- b** Personen und deren Rollen, z. B. Kunde, Mitarbeiter, Dozent.
- c** Informationen über Aktionen, z. B. Banküberweisung durchführen.
- d** Orte, z. B. Wartezimmer.
- e** Organisationen, z. B. Filiale.
- f** Behälter, z. B. Lagerplatz.
- g** Dinge in einem Behälter, z. B. Reifen in einem Lagerplatz.
- h** Ereignisse, z. B. Eheschließung.
- i** Kataloge, z. B. Produktkatalog.
- j** Verträge, z. B. Autokaufvertrag.

Beispiel 1: Die Funktionen, die sich im Lastenheft auf einen Dozenten beziehen, werden zu einer Klasse zusammengefasst (Abb. 9.1-12, links). Die zusätzlichen Informationen im Pflichtenheft ermöglichen die Festlegung von Attributen und die Modellierung von Assoziationen (Abb. 9.1-12, rechts).

2 Dokumentanalyse Besonders einfach lassen sich Klassen mittels der sogenannten **Dokumentanalyse** identifizieren. Dokumente enthalten Attribute, die mittels Bottom-up-Vorgehen zu Klassen zusammengefasst wer-

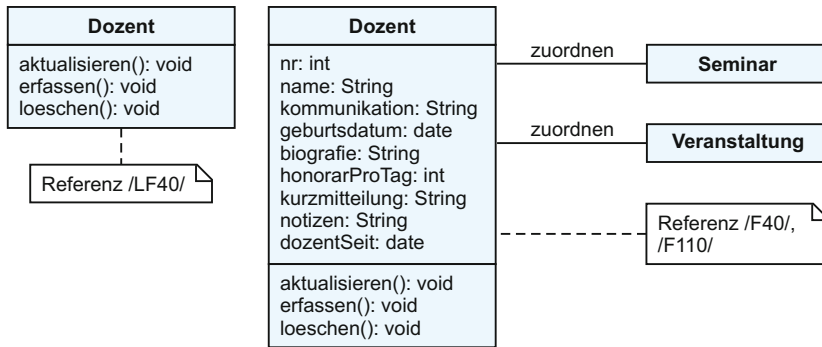


Abb. 9.1-12: UML-Modellierung der Anforderungen an Dozenten.

den. Wählen Sie den Klassennamen nach der Gesamtheit der Attribute. Da die Dokumentanalyse auch zum Identifizieren von Assoziationen dient, werden meist gleichzeitig mit den Klassen Assoziationen ermittelt (siehe »Assoziation«, S. 158).

Aus dem Formular zur Seminaranmeldung lassen sich mittels Dokumentanalyse die Klassen Teilnehmer, Seminar und Rechnungsempfänger ableiten (Abb. 9.1-14).

Beispiel 2:
SemOrg

Bei der Dokumentanalyse muss besonders darauf geachtet werden, dass

- aus einem Dokument im Allgemeinen mehrere Objekte derselben oder verschiedener Klassen abgeleitet werden können,
- die Objekte in einem Dokument im Allgemeinen nicht vollständig dargestellt sind,
- ein gegebenes Dokument nicht repräsentativ sein muss.

Aus der Beschreibung der *Use Cases* (siehe »Use Case-Diagramme und -Schablonen«, S. 255) können Sie mittels der Top-down-Vorgehensweise Klassen ableiten. Gehen Sie den Text durch und durchsuchen Sie ihn nach Klassen. Oft sind die Substantive potenzielle Klassen. Ebenso kann sich eine Klasse hinter Verben verbergen. Eine Klasse lässt sich relativ leicht durch ihre Attribute identifizieren. Der Erfolg dieser Methode wird entscheidend durch die Sicherheit des *Requirements Engineers* im Erkennen der potenziellen Klassen bestimmt.

3 Use Cases

4 Kategorien

Die oben identifizierten Klassen lassen sich folgenden Kategorien zuordnen:

Beispiel 3:
SemOrg

- Personen und deren Rollen: Teilnehmer, Rechnungsempfänger, Dozent.
- Kataloge: Seminar, Veranstaltung.

Box 1b:
Checkliste
Klassen

Analytische Schritte

5 Liegt ein aussagefähiger Klassenname vor?

Der **Klassenname** soll

- der Fachterminologie entsprechen,
- ein Substantiv im Singular sein, zusätzlich kann ein Adjektiv angegeben werden, z. B. Seminar, ÖffentlicheAusschreibung,
- so konkret wie möglich gewählt werden,
- dasselbe ausdrücken wie die Gesamtheit der Attribute,
- nicht die Rolle dieser Klasse in einer Beziehung zu einer anderen Klasse beschreiben,
- eindeutig im Paket bzw. im System sein und
- nicht dasselbe ausdrücken wie der Name einer anderen Klasse.

6 Besitzt die Klasse eine starke Bindung?

- Jede Operation/Methode der Klasse ist für sich funktional gebunden.
- Alle Operationen/Methoden der Klasse arbeiten auf einer einzigen Datenstruktur.
- Die Klasse enthält *keine* Operationen/Methoden, die an andere Klassen delegiert werden können.

7 Keine Code-Vervielfältigung

- Weder innerhalb einer Klasse noch zwischen Klassen dürfen Code-Stücke in identischer Form mehrfach vorkommen.

8 Ist das gewählte Abstraktionsniveau richtig?

- Die Ziele sind *nicht* möglichst viele Klassen oder Klassen möglichst geringer Komplexität zu identifizieren.

9 Wann liegt keine Klasse vor?

- *Keine* Klassen bilden, um Mengen von Objekten zu verwalten.

10 Fehlerquellen

- Zu kleine Klassen.
- Für jeden Bericht/Report eine Klasse modellieren.
- Bei der Anforderungsspezifikation/fachlichen Lösung/OOA-Modell:
 - ☐ Klasse modelliert Benutzungsoberfläche.
 - ☐ Klasse verwaltet Menge von Objekten (Container-Klasse).
 - ☐ Klasse modelliert Entwurfs- oder Implementierungsdetails.

Analytische Schritte

6 Starke Bindung

Um eine starke Bindung (siehe »Prinzip der Bindung und Kopplung«, S. 37) zu erreichen, sollte der Entwurf einer Klasse »verantwortungsgetrieben« sein – im Englischen bezeichnet als *responsibility-driven design* (RDD) [WiMc02]. Jeder Klasse sollten gut definierte Verantwortlichkeiten zugeordnet werden, die einen Teil der Anwendungsfunktionalität abdecken. Jede Klasse sollte für ihre eigenen Daten verantwortlich sein.

Beispiel 4a

In einem E-Learning-System werden die Lernenden in einem LernendenContainer verwaltet. In einer Liste werden dort die Referenzen auf die einzelnen Lernenden-Objekte aufbewahrt. Zusätzlich soll noch eine Statistik über das Verhalten der Lernenden erstellt werden wie durchschnittliche Lerndauer aller Lernenden, Lernzeiten im Laufe eines Tages, einer Woche, eines Monats usw.

Wo würden Sie diese Informationen berechnen und ablegen?



Es bieten sich zwei Alternativen an: Die Klasse `LernendenContainer` kann um entsprechende Klassenattribute und Klassenmethoden erweitert werden. Oder es wird eine neue Klasse `LernendenStatistik` angelegt, die nur die für die Statistik notwendigen Attribute und Methoden enthält. Betrachtet man die Verantwortlichkeiten, dann ist nur die zweite Lösung geeignet, um eine starke Bindung der Klassen sicherzustellen.

Beispiel 4b

Weder innerhalb einer Klasse noch zwischen Klassen dürfen Code-Stücke in identischer Form mehrfach vorkommen. Bei Änderungen findet ein Wartungs-Programmierer oft nur eines dieser identischen Code-Stücke und ändert dann nur dieses. Schwer zu findende Folgefehler treten dann auf.

7 code duplication

Anmeldung zu TEACHWARE-Seminaren

Als Teilnehmer zu nachfolgenden TEACHWARE-Seminaren wird angemeldet:

_____	_____	_____
Titel	Vorname	Nachname
_____	_____	_____
Seminar-Nr.	Seminarbez.	vom - bis
_____	_____	_____
_____	_____	_____

Anmeldebestätigung und Rechnung erbeten an:

_____	_____	_____
Titel	Vorname	Name
_____	_____	_____
Firma	Str./Postfach	
_____	_____	
LKZ	PLZ	Ort
_____	_____	_____
Telefon		

Arrows point from the following fields to labels:

- From the first row of the first table to **Teilnehmer**
- From the first row of the second table to **Seminar**
- From the first row of the third table to **Rechnungsempfänger**

Abb. 9.1-14:
Beispiel zur
Dokumentanalyse.

9.2 Funktions-Strukturen

Funktionen werden in der Regel zu größeren Einheiten zusammengefasst.

Einzelne Funktionen werden oft durch Funktionsbäume hierarchisch strukturiert:

- »Funktionsbaum«, S. 143

In der objektorientierten Softwareentwicklung werden Klassen (siehe »Funktionalität«, S. 127) und Schnittstellen mit Hilfe verschiedener Basiskonzepte zum Aufbau von statischen Strukturen verwendet. Pakete dienen im Wesentlichen dazu, Klassen zu gruppieren:

- »Pakete«, S. 145

Bei der einfachen Vererbung entstehen Baumstrukturen, bei der Mehrfachvererbung Netzstrukturen zwischen Klassen, die für alle Objekte aller Klassen wirksam sind. Ergänzt werden können Vererbungsstrukturen um Schnittstellen-Strukturen:

- »Vererbung«, S. 150

Mit Hilfe von Assoziationen werden mögliche Verbindungen zwischen Objekten von Klassen modelliert. Die so modellierten Verbindungen können während der Laufzeit hergestellt werden, müssen es aber nicht. Die technischen Voraussetzungen für einen Verbindungsaufbau müssen in der Programmierung bereitgestellt werden – sie müssen aber während der Laufzeit nicht in Anspruch genommen werden. Dies ist ein wesentlicher Unterschied zur Vererbung, die immer gilt. Insofern haben Assoziationen nicht nur einen statischen, sondern auch einen dynamischen Aspekt:

- »Assoziation«, S. 158

Assoziationen können auch hierarchisch strukturiert werden:

- »Aggregation und Komposition«, S. 171

Für spezielle Zwecke können weitere Strukturen modelliert werden:

- »Weitere Strukturen«, S. 177

Für die Ermittlung und Überprüfung von wichtigen Basiskonzepten gibt es konstruktive und analytische Schritte:

- »Box: Pakete – Methode und Checkliste«, S. 148

- »Box: Vererbung – Methode und Checkliste«, S. 155

- »Box: Assoziationen – Methode und Checkliste«, S. 166

- »Box: Multiplizitäten – Methode und Checkliste«, S. 169

- »Box: Komposition und Aggregation – Methode und Checkliste«, S. 175

9.2.1 Funktionsbaum

Funktionen stehen oft in einem hierarchischen Abhängigkeitsverhältnis. Ein **Funktionsbaum** ermöglicht es, eine funktionale Hierarchie in Form eines Baumdiagramms darzustellen.

Ein Funktionsbaum wird grafisch dargestellt. Funktionen werden i. Allg. als beschriftete Rechtecke (»Kästchen«) gezeichnet. Die hierarchische Beziehung zwischen den Funktionen wird durch unbeschriftete Linien dargestellt. Eine Funktion A wird in einer Hierarchieebene über den Funktionen B und C angeordnet, wenn A übergeordnet zu B und C ist (Abb. 9.2-1). Übergeordnet kann Verschiedenes bedeuten:

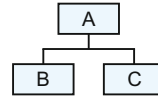


Abb. 9.2-1:
Darstellung eines
Funktionsbaums.

- Besteht-aus (meist in der Spezifikationsphase) (Funktion A besteht aus den Funktionen B und C),
- Ruft-auf (meist in der Entwurfsphase) (Funktion A ruft die Funktionen B und C auf).

Man nennt A auch die Vaterfunktion von B und C. B und C werden entsprechend als Kindfunktionen von A bezeichnet.

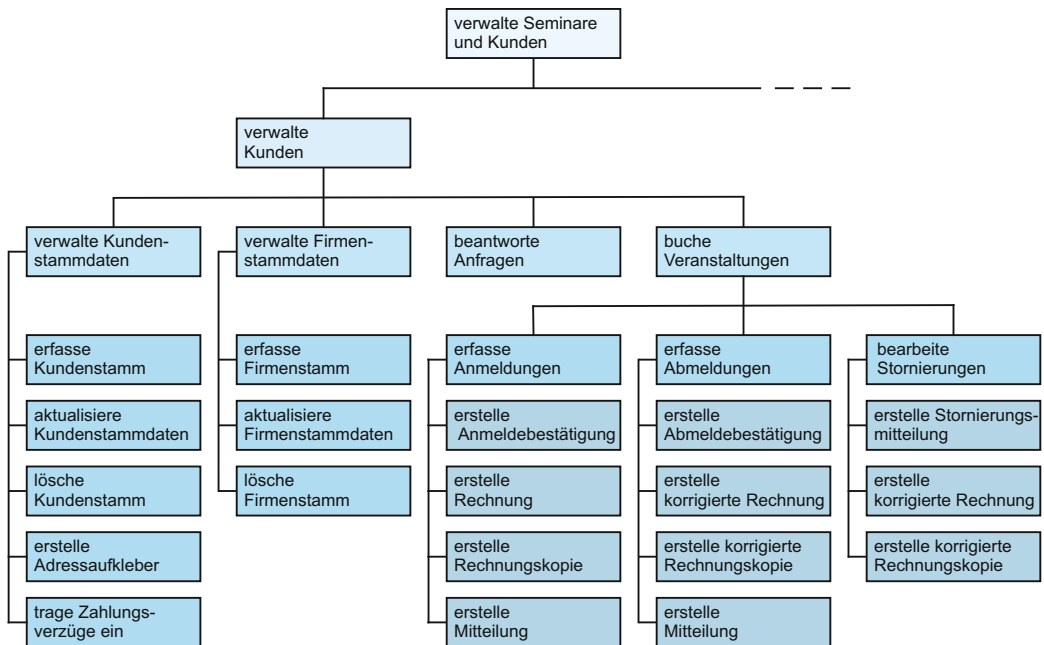


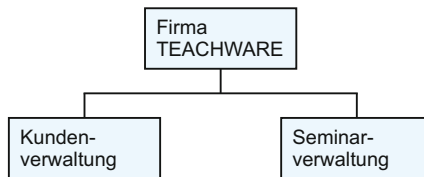
Abb. 9.2-2:
Funktionsbaum
der Kundenverwaltung.

III 9 Statik

Beispiel: Die Kundenverwaltung lässt sich in den Funktionsbaum der Abb. 9.2-2 gliedern. Sie zeigt eine sachlogische, tätigkeitsorientierte Zergliederung der Funktion »verwalte Kunden«. Sachlogisch deshalb, weil nichts darüber ausgesagt wird, wer welche Funktion ausführt. Die Funktion »trage Zahlungsverzüge ein« wird z. B. von dem Buchungssystem ausgeführt, während die anderen Funktionen der Kundenverwaltung vom Kundensachbearbeiter getätigt werden.

Funktionsbäume werden auch in anderen Kontexten verwendet, z. B. zur Gliederung einer Firma in organisatorische Einheiten (Abb. 9.2-3).

Abb. 9.2-3:
Organigramm
einer
Organisation.



Funktionsbäume lassen sich folgendermaßen bewerten:

- + Bewährtes Konzept zur systematischen Gliederung von Funktionen.
- + Gibt erste Hinweise für eine mögliche Dialoggestaltung.
- Berücksichtigen nur die funktionale Sicht.

QS-Checkliste
Funktionsbaum



Prüfen Sie, ob folgende Aussagen zutreffen:

- Die Relation des Funktionsbaums ist angegeben, z. B. »besteht aus«.
- Unter einer gemeinsamen Vaterfunktion sind nur Funktionen angeordnet, die fachlich eng zusammengehörende Tätigkeiten beschreiben. Was eng zusammengehört, kann nur mit Fachwissen entschieden werden.
- Auf einer Hierarchieebene sind Funktionen angeordnet, die sich auf gleichem Abstraktionsniveau befinden.



Der Funktionsbaum lässt sich auf keinen bestimmten Urheber zurückführen. Auch lässt sich kein Zeitpunkt angeben, zu dem zum ersten Mal ein Funktionsbaum verwendet wurde. In dem 1974 entwickelten Konzept HIPO war er bereits ein Bestandteil des Konzepts. Es handelt sich jedoch um ein so allgemeines Basiskonzept, dass es sicher bereits viel früher eingesetzt wurde.

Zur Klassifikation

- Funktionsbäume werden grafisch mit textuellen Annotationen dargestellt.
- Der Grad der Formalität ist informal bis semiformal.
- Sie werden in der Spezifikations- und Entwurfsphase eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

9.2.2 Pakete

Pakete in der Softwareentwicklung

Bei umfangreichen Softwareentwicklungen entstehen viele Klassen und Diagramme. Um einen Überblick über diese Vielfalt zu bewahren, wird ein Strukturierungskonzept benötigt, das von den Details abstrahiert und die übergeordnete Struktur verdeutlicht.

Pakete (*packages*) sind ein solcher Strukturierungsmechanismus. Sie erlauben es, Komponenten zu einer größeren Einheit zusammenzufassen. Der Paketbegriff ist allerdings nicht einheitlich definiert. Üblich sind auch die Begriffe Subsystem, *subject* und *category*.

Pakete in der UML

In der UML gruppiert ein Paket Modellelemente (z. B. Klassen) und Diagramme. Ein Paket kann selbst Pakete enthalten. Ein Paket wird in der UML als Rechteck mit einem Reiter dargestellt (siehe Marginalspalte). Wird der Inhalt des Pakets nicht gezeigt, dann wird der Paketname in das Rechteck geschrieben, andernfalls in den Reiter. Der Paketname muss im beschriebenen Softwaresystem eindeutig sein.

Elemente können direkt in den Paketrumpf gezeichnet werden. Eine alternative Darstellung erlaubt es auch, über eine Liniendarstellung und ein Pluszeichen die Zugehörigkeit zu einem Paket zu zeigen (Abb. 9.2-4).



Abb. 9.2-4:
Notation von
Paketen.

Für die in einem Paket enthaltenen Elemente können Sichtbarkeiten definiert werden (Abb. 9.2-5):

- **+** : public-Element, ist für alle Pakete sichtbar, die das betreffende Paket importieren.
 - **-** : private-Element, ist nur in dem betreffenden Paket sichtbar.
- Auf public-Elemente kann außerhalb des Pakets jederzeit durch die qualifizierten Namen zugegriffen werden, z. B. `Paket::Klasse`. Wird bei einem Element keine Sichtbarkeit angegeben, dann gilt implizit die Sichtbarkeit `public`. Die Sichtbarkeiten `protected` und `package` können in Zusammenhang mit Paketelementen *nicht* verwendet werden.

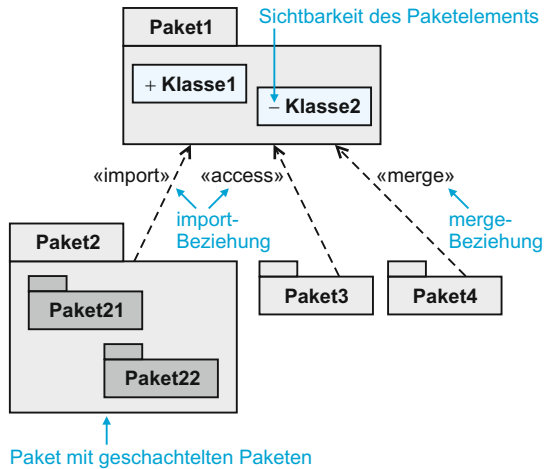
Die Abb. 9.2-5 zeigt die Notation für Pakete im Überblick.

Sichtbarkeit

Notation Pakete

III 9 Statik

Abb. 9.2-5:
Notation für
Pakete.



Paket-Import

Zwischen zwei Paketen kann eine *import*-Beziehung existieren, die durch einen gestrichelten Pfeil und das Schlüsselwort *«import»* oder *«access»* dargestellt wird. Der Pfeil zeigt auf dasjenige Paket, das importiert werden soll. Ist kein Schlüsselwort angegeben, dann gilt implizit *«import»*. Wenn ein **Paket-Import** spezifiziert ist, dann können die Elemente des importierten Pakets im importierenden Paket ohne qualifizierenden Namen verwendet werden.

Ein Paket-Import kann auch zwischen einem Paket und einem Element eines anderen Pakets existieren. Dann zeigt die Pfeilspitze direkt auf dieses Element. Das Importieren eines Pakets ist semantisch äquivalent zum Importieren jedes einzelnen Elements dieses Pakets.

«import»

Das Schlüsselwort *«import»* beschreibt in der Abb. 9.2-6 einen *public*-Import zwischen den Paketen P3 und P1. Die Sichtbarkeit der Klassen bleibt durch den Import voll erhalten. In P3 kann auf die Klassen K1 und K2 zugegriffen werden, ohne dass diese Klassen durch den Paketnamen P1 qualifiziert werden müssen. P3 wird selbst von einem weiteren Paket P4 importiert. In P4 kann direkt auf die Klassen K1, K2, K5 und K6 zugegriffen werden.

«access»

Das Schlüsselwort *«access»* kennzeichnet einen *privaten* Import zwischen den Paketen P3 und P2 (Abb. 9.2-6). In P3 kann auf die Klassen K3 und K4 ohne qualifizierenden Namen zugegriffen werden. Die Sichtbarkeit dieser Klassen wird in P3 *privat*. Daher können diese Zugriffe *nicht* über die *import*-Beziehung an P4 weitergegeben werden, sondern P4 kann nur auf die Klassen von P3 (K5, K6) und P1 (K1, K2) zugreifen.

Paketdiagramme

Die Pakete und ihre Abhängigkeiten (*import*, *merge*) werden im **Paketdiagramm** dargestellt. Dabei dienen Paketdiagramme im objektorientierten Entwurf auch dazu, die verschiedenen Architekturschichten (Benutzungsoberfläche, Fachkonzept, Datenbankzugriffe etc.) präzise voneinander zu trennen.

9.2 Funktions-Strukturen III

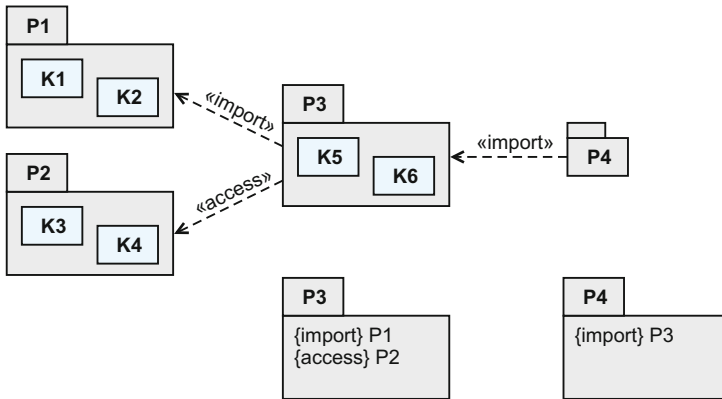


Abb. 9.2-6: Import-Beziehung zwischen Paketen.

Die Abb. 9.2-7 zeigt, wie mit Hilfe von Paketen SemOrg strukturiert werden kann.

Beispiel: SemOrg

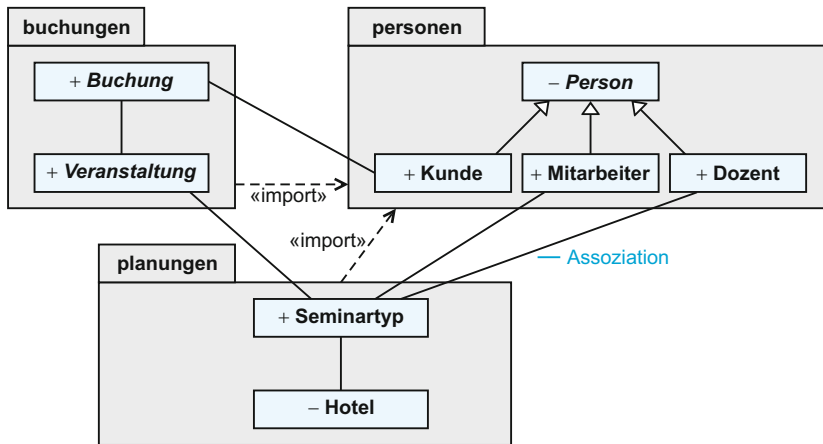


Abb. 9.2-7: Paketdiagramm zu SemOrg.

- Ein Paket fasst mehrere Elemente (z. B. Klassen) zusammen. Es kann auch weitere Pakete enthalten.
- Für Paketelemente kann die Sichtbarkeit public (+) oder private (–) spezifiziert werden. Als Voreinstellung gilt public. Außerhalb des Pakets kann nur auf public-Elemente zugegriffen werden.
- Der Paket-Import ermöglicht einen einfachen Zugriff auf Paketelemente ohne qualifizierenden Namen, z. B. Paket1::Klasse1.
- Paketelemente, die mittels «import» importiert werden, bleiben public.
- Paketelemente, die mittels «access» importiert werden, werden private.
- Der Paket-Merge kopiert Elemente aus dem Zielpaket in das Quellpaket, wo sie verändert werden können.

III 9 Statik

Pakete in Java

Aufgaben **Java** In Java können Klassen zu Paketen zusammengefasst werden. Pakete dienen in Java dazu,

- große Gruppen von Klassen, die zu einem gemeinsamen Aufgabenbereich gehören, zu bündeln und zu verwalten,
- potenzielle Namenskonflikte zu vermeiden,
- Zugriffsrechte und Sichtbarkeiten zu definieren und zu kontrollieren,
- eine Hierarchie von verfügbaren Subsystemen aufzubauen.

Jede Klasse in Java ist Bestandteil genau eines Pakets.

Schreibweise Paketnamen enthalten ausschließlich Kleinbuchstaben, z. B. `inout`, und beginnen immer mit der URL des Herstellers in umgekehrter Reihenfolge, z. B. `de.w3l.inout`. Oft wird die Länderkennung – hier `de` – weggelassen: `w3l.inout`.



Die Namenskonventionen werden ausführlich in [Bloc05, S. 165 (Item 38)] erläutert.

Paket-Hierarchie

Pakete sind in Java hierarchisch gegliedert, d. h. ein Paket kann Unterpakete besitzen, die selbst wieder in Unterpakete aufgeteilt sind, usw. Die Paket-Hierarchie wird durch eine Punktnotation ausgedrückt: `paket.unterpaket1.unterpaket11.Klasse`.

C++

In C++ gibt es das Konzept des Pakets nicht.

Zur Klassifikation

- Pakete werden sowohl textuell (in Programmiersprachen) als auch grafisch (in der UML mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

9.2.3 Box: Pakete – Methode und Checkliste

Zum Identifizieren und Überprüfen von Paketen hilft die Box »Checkliste Pakete«. Auf einige Aspekte zu der Box wird im Folgenden eingegangen.

Konstruktive Schritte

- 1 Top-down Bei großen Systemen sollte das Gesamtsystem in Pakete unterteilt werden. Ein Paket entspricht einem in sich abgeschlossenen Teilsystem. Umfangreiche Pakete sind in weitere Pakete zu zerlegen. In der Spezifikationsphase ist die Paketbildung vor allem für *Use Cases* und für Klassen sinnvoll.

Ergebnis

■ Paketdiagramm

Erstellen Sie ein oder mehrere Paketdiagramme. Ordnen Sie jedem Paket Modellelemente zu. Spezifizieren Sie die Abhängigkeiten zwischen den Paketen.

Konstruktive Schritte

1 Welche Pakete ergeben sich durch ein Top-down-Vorgehen?

Bei großen Anwendungen:

- Unterteilen Sie das Gesamtsystem in Teilsysteme (Pakete).
- Zerlegen Sie umfangreiche Pakete in weitere Pakete.

2 Welche Pakete ergeben sich durch ein Bottom-up-Vorgehen?

Bei kleinen und mittleren Anwendungen:

- Fassen Sie Klassen unter einem Oberbegriff zusammen.

3 Use Cases paketieren

- Fassen Sie *Use Cases* durch Bildung von Funktionsgruppen zu Paketen zusammen.

4 Klassen paketieren

- Fassen Sie Klassen durch Bildung von »höheren« Datenstrukturen zusammen.

Analytische Schritte

5 Besitzt das Paket eine starke Bindung?

(siehe »Prinzip der Bindung und Kopplung«, S. 37)

- Es umfasst einen Themenbereich, der für sich allein betrachtet und verstanden werden kann.
- Es enthält Klassen, die logisch zusammengehören, z. B. Artikel, Lieferant und Lager.
- Es erlaubt eine System-Betrachtung auf einer höheren Abstraktionsebene.

6 Besitzt das Paket eine schwache Kopplung?

(siehe »Prinzip der Bindung und Kopplung«, S. 37)

- Vererbungsstrukturen liegen möglichst innerhalb eines Pakets. Wenn nötig, dann nur in vertikaler Richtung durchschneiden, d. h. zu jeder Unterklasse sollen alle Oberklassen in dem Paket enthalten sein.
- Aggregationen sind nicht durchtrennt.
- Möglichst wenig Assoziationen sind durchtrennt.

7 Ist der Paketname geeignet?

- Der Inhalt eines Pakets muss mit 25 Worten oder weniger beschreibbar sein.
- Aus der Beschreibung den Namen ableiten.
- Paketnamen dürfen keine Verben enthalten.

8 Umfang?

- Die Größe der Pakete soll mit der Größe des modellierten Systems korrelieren.

9 Fehlerquellen

- Zu kleine Pakete.
- Keine in sich geschlossene Einheit.

Box: Checkliste Pakete

Während bei kleineren Systemen ganz auf Pakete verzichtet werden kann, empfiehlt es sich bei Systemen mittlerer Größenordnung, nach der Erstellung der *Use Cases* oder spätestens nach der statischen Modellierung Pakete bzw. Teilsysteme zu bilden.

Fassen Sie *Use Cases* zu Paketen zusammen, indem Sie entsprechende Funktionsgruppen bilden, z. B. Einkauf, Auftragsbearbeitung. *Use Cases*, die durch eine extend-Beziehung, eine include-Beziehung oder eine Generalisierung verbunden sind, gehören in dasselbe Paket wie der Basis-*Use Case*.

2 Bottom-up

3 Use Cases paketieren

III 9 Statik

- 4 Klassen paketieren** Bilden Sie Pakete von Klassen, indem Sie »höhere« Datengruppen bilden. Beispielsweise können alle Klassen, die sich mit dem Artikelstamm befassen (Artikel, Lagerartikel, Lieferant, Lager) in einem Paket Artikelstamm zusammengefasst werden.

Analytische Schritte

- 5 Starke Bindung** In sich abgeschlossene Pakete unterstützen die Arbeitsteilung, denn jedes Paket bildet eine Arbeitseinheit für ein Team. Ein in sich abgeschlossenes Paket besitzt folgende Eigenschaften:
- Es führt den Leser durch das System.
 - Es enthält einen Themenbereich, der für sich allein betrachtet und verstanden werden kann, oder es ist mit minimalen Bezügen zu anderen Paketen verständlich.
 - Es besteht nicht einfach aus einer Menge von Modellelementen, sondern Pakete sollen eine Betrachtung des Systems auf höherer Abstraktionsebene ermöglichen. Es muss daher *Use Cases* und Klassen zusammenfassen, die dem gleichen Themenbereich angehören.
 - Klassen innerhalb einer Generalisierungsstruktur sollten immer vollständig in einem Paket liegen.
 - Kompositionen & Aggregationen liegen vollständig in einem Paket.
 - Klassen, zwischen deren Objekten eine intensive Kommunikation stattfindet, gehören zum selben Paket.

9.2.4 Vererbung

In der Regel besteht ein Softwaresystem aus einer Vielzahl von Klassen. Manche Klassen besitzen dabei gemeinsame oder ähnliche klingende Attribute und Operationen.

Die Einfachvererbung

- Generalisieren** In der Objektorientierung spricht man von **Generalisieren**, wenn man in Klassen **semantische Gemeinsamkeiten** feststellt. Diese Gemeinsamkeiten, die sich auf Attribute, Operationen und Assoziationen beziehen, werden in einer neuen sogenannten **Oberklasse** (*superclass*) – auch **Basisklasse** genannt – zusammengefasst. Aus den betrachteten Klassen werden die Gemeinsamkeiten entfernt. Diese Klassen heißen dann **Unterklassen** (*subclass*) – auch **abgeleitete Klassen** genannt – und werden in der UML durch eine Linie mit einem weißen bzw. transparenten Dreieckspfeil mit der Ober-

klasse verbunden, wobei der Pfeil zur Oberklasse zeigt (Abb. 9.2-9). In der UML spricht man von Generalisierungs-/Spezialisierungshierarchien, in der objektorientierten Programmierung von Vererbung.

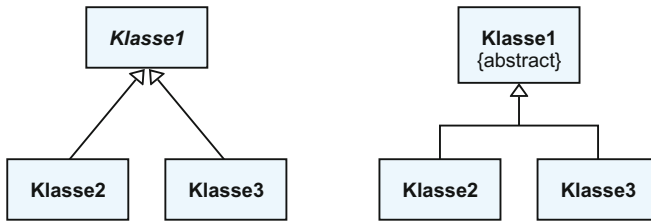


Abb. 9.2-9: UML-Notation für die Generalisierung/Spezialisierung.

Wie die Abb. 9.2-9 zeigt, gibt es zwei alternative Notationen. Die Linien können von jeder Unterklasse direkt zur Oberklasse geführt werden (linke Notation) oder die Linien werden zuerst zusammengeführt und dann ein Pfeil zur Oberklasse gezeichnet (rechte Notation).

Wenn ein Objekt der Oberklasse nur mit den eigenen Attributen und Operationen Sinn macht, dann sollte die Objekterzeugung möglich gemacht werden. Wenn eine Objekterzeugung *nicht* sinnvoll ist, dann sollte sie verboten werden. Dies erreicht man dadurch, dass man die Oberklasse zu einer **abstrakten Klasse** macht. Man unterscheidet daher konkrete Klassen, kurz Klassen genannt, von denen Objekte erzeugt werden können, und abstrakte Klassen, von denen *keine* Objekte erzeugt werden können.

In der UML kennzeichnet ein kursiv geschriebener Klassenname eine abstrakte Klasse. Alternativ oder zusätzlich kann in geschweiften Klammern auch das Wort **abstract** rechts unter den Klassennamen geschrieben werden (siehe Abb. 9.2-9).



Beim Generalisieren, d. h. beim Suchen nach Gemeinsamkeiten in Klassen, darf man sich nicht durch gleiche Bezeichnungen in verschiedenen Klassen dazu verleiten lassen, sofort auf gleiche Semantik zu schließen. Es ist daher immer sorgfältig zu prüfen, ob mit gleichen Bezeichnungen auch die gleiche Semantik verbunden ist.



Eine Unterklasse ist vollständig konsistent mit ihrer Oberklasse, enthält aber in der Regel zusätzliche Informationen (Attribute, Operationen, Assoziationen). Durch die Vererbung entsteht eine **Klassenhierarchie** bzw. eine Vererbungsstruktur.

Hierarchie

Bei der **Vererbung** geht es *nicht* nur darum, gemeinsame Eigenschaften und Verhaltensweisen zusammenzufassen, sondern eine Vererbungshierarchie muss immer auch eine Generalisierung bzw. Spezialisierung darstellen. Man muss sagen können:

»ist ein«

Jedes Objekt der Unterklasse »ist ein« (*is a*) Objekt der Oberklasse.

Beim Vorgang der Generalisierung geht es darum, aus vorhandenen Klassen Gemeinsamkeiten zu isolieren und in einer Oberklasse zusammenzufassen. In der Praxis kommt aber auch der umgekehrte Vorgang vor:

Spezialisieren

III 9 Statik

Eine vorhandene Klasse um Spezialitäten zu erweitern, d.h. an eine vorhandene Klasse eine Unterklasse hängen. Dieser Vorgang wird **Spezialisieren** genannt.

Beispiel: Die Abb. 9.2-10 zeigt die Vererbungsstrukturen in der Fallstudie »Seminarorganisation«.

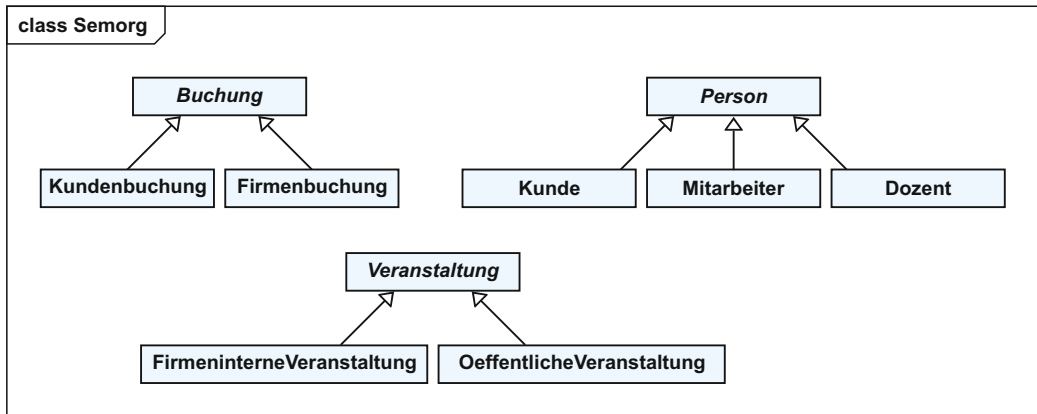


Abb. 9.2-10: Vererbungsstrukturen in der Fallstudie »Seminarorganisation«.

Durch das Konzept der Vererbung ist es in einer Unterklasse möglich, geerbte Operationen zu überschreiben und geerbte Attribute zu verbergen. Eine Unterklasse **überschreibt** (*override*) bzw. redefiniert (*redefine*) eine Operation einer Oberklasse, wenn sie eine Operation gleichen Namens enthält. Bei der Redefinition ist folgende Einschränkung zu beachten: In einer redefinierten Operation müssen die Anzahl und die Typen der Ein-/Ausgabeparameter gleich bleiben. Mit anderen Worten: Die Schnittstelle der neuen Operation in der Unterklasse muss konform zur Operation der Oberklasse sein. Überschreiben erleichtert es also den Unterklassen, das Verhalten einer bestehenden Klasse zu erweitern.

Vererben von Assoziationen

Nicht nur Attribute und Operationen werden vererbt, sondern auch **Assoziationen**.

Unterklassen-Objekte wie Oberklassen-Objekte

Klassen können wie Typen verwendet werden. Spezifiziert man eine Operation, die einen Parameter vom Typ der Oberklasse enthält, dann kann während der Laufzeit dort auch ein Objekt einer Unterklasse stehen.

Beispiel Steht auf der Parameterliste ein Referenz-Attribut vom Typ der Klasse Konto, dann kann beim Aufruf der Operation, d.h. zur Laufzeit, eine Referenz auf ein Objekt der Klasse Konto oder eine Referenz auf ein Objekt der Unterklasse Sparkonto angegeben werden (wenn Sparkonto = Unterklasse von Konto).



Generell gilt: Ein Objekt einer Unterklasse kann überall dort verwendet werden, wo ein Objekt der Oberklasse erlaubt ist.

Keine völlig andere Funktionalität

Die überschriebene Operation sollte *nicht* eine völlig andere Funktionalität realisieren, sondern die ursprüngliche Operation mit `super.XXX()` nutzen und nur um zusätzliche Funktionalität erweitern.

Hinweis

Durch die Einfachvererbung entsteht eine statische Struktur in Form eines **gerichteten Baums**, die für alle Objekte aller Klassen gültig ist.

Struktur

Die Mehrfachvererbung

Die **Mehrfachvererbung** ist eine Vererbungsstruktur, in der jede Klasse *mehrere direkte* Oberklassen besitzen kann. Sie kann als azyklisches Netz dargestellt werden. Bei der Mehrfachvererbung kann der Fall auftreten, dass eine Klasse von ihren Oberklassen zwei Attribute oder Operationen gleichen Namens erbt. Hier muss festgelegt werden, wie diese Konflikte zu lösen sind.

Die Abb. 9.2-11 zeigt ein Beispiel für die Mehrfachvererbung. Die Klasse `UhrAnzeige` ist hier als abstrakte Klasse modelliert, weil es – in diesem Modell – außer der Digital-, der Analog- und der Analog-Digital-Anzeige keine andere Anzeige gibt.

Beispiel

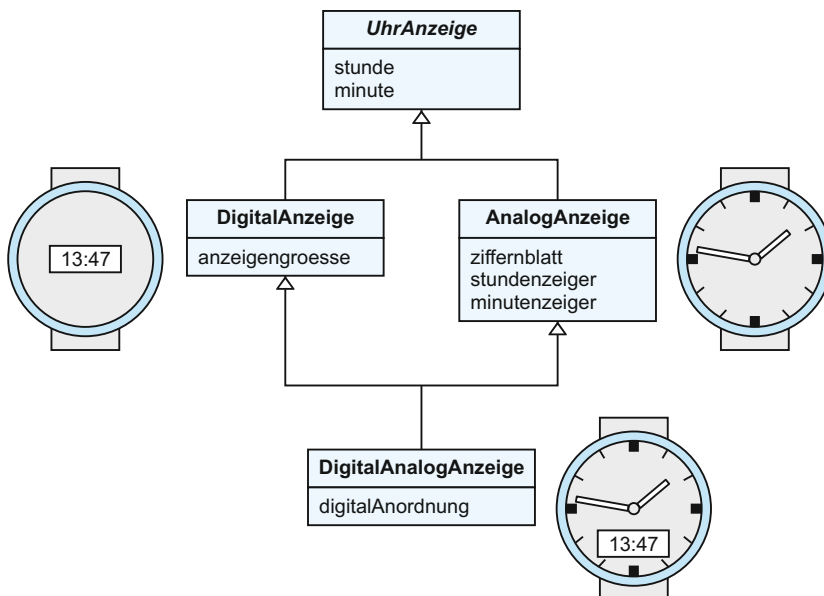
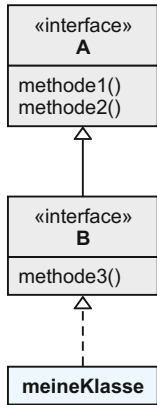


Abb. 9.2-11:
Beispiel für die
Mehrfach-
vererbung.

III 9 Statik

Abb. 9.2-12:
Beispiel für die
Vererbung von
Schnittstellen.



In Java ist eine echte Mehrfachvererbung *nicht* möglich. Durch die Verwendung von Schnittstellen ist aber eine Art Mehrfachvererbung möglich. In Java können Schnittstellen andere Schnittstellen – auch mehrere – erben (extends).

Wenn eine Klasse eine Schnittstelle B implementiert, die eine Schnittstelle A erbt, dann muss die Klasse alle Methoden implementieren, die in der Vererbungskette definiert sind (Abb. 9.2-12).

In C++ ist eine echte Mehrfachvererbung möglich.

In der Praxis wird die Mehrfachvererbung in der Spezifikationsphase, in der das Fachkonzept modelliert wird, kaum benötigt. Im Entwurf, in dem das technische Konzept modelliert wird, wird die Mehrfachvererbung aber häufig benötigt und eingesetzt.

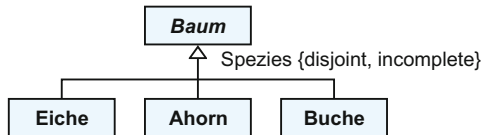
Generalisierungseigenschaften

Für Vererbungsstrukturen können Generalisierungseigenschaften angegeben werden. Die UML kennt folgende vordefinierte Generalisierungseigenschaften:

■ *incomplete*

Die betreffende Vererbungsstruktur enthält einen Teil der Unterklassen. Es gibt weitere Unterklassen, die das Modell noch nicht enthält. Beispielsweise modelliert die Vererbungsstruktur der Abb. 9.2-13 nur einige Spezies von Bäumen.

Abb. 9.2-13:
Generalisierungseigenschaften
disjoint und
incomplete.



■ *disjoint*

Die Eigenschaften der Unterklassen überschneiden sich nicht. In der Abb. 9.2-13 werden unterschiedliche Spezies von Bäumen modelliert.

■ *overlapping*

In der Abb. 9.2-14 besitzt ein Segelboot sowohl Eigenschaften von Fahrzeugen mit Windantrieb als auch von Wasserfahrzeugen. Die Klasse Segelboot wird daher von beiden Klassen abgeleitet, d.h. als Mehrfachvererbung modelliert.

■ *complete*

Die Menge der Unterklassen ist vollständig. Weitere Unterklassen werden aufgrund der Problemstellung nicht erwartet.

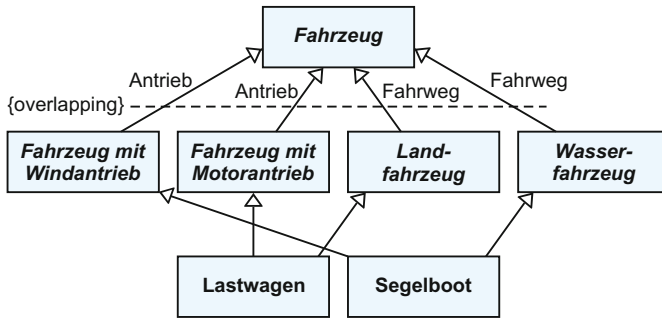


Abb. 9.2-14:
Generalisierungseigenschaft
overlapping.

Folgende Kombinationen sind möglich:

Generalisierungseigenschaften ::= [{incomplete, disjoint} / {complete, disjoint} / {incomplete, overlapping} / {complete, overlapping}]

Durch die Mehrfachvererbung entsteht eine statische Struktur in Form eines **gerichteten azyklischen Graphen mit partieller Ordnung**, die für alle Objekte aller Klassen gültig ist.

Struktur

Vererbungsstrukturen werden in der UML in einem **Klassendiagramm** modelliert.



Zur Klassifikation

- Vererbungsstrukturen werden sowohl textuell (in Programmiersprachen) als auch grafisch (in der UML mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

9.2.5 Box: Vererbung – Methode und Checkliste

Zum Identifizieren und Überprüfen von Vererbungsstrukturen hilft die Box »Checkliste Vererbung«. Auf einige Aspekte zu der Box wird im Folgenden eingegangen.

Konstruktive Schritte

Vererbungsstrukturen können prinzipiell durch ein Top-down- oder Bottom-up-Vorgehen ermittelt werden.

Beim Bottom-up-Vorgehen prüfen Sie für zwei oder mehrere Klassen, ob sie genügend Gemeinsamkeiten besitzen, damit sich eine neue Oberklasse bilden lässt. Verlagern Sie gemeinsame Attribute, Operationen und Assoziationen in die neu gebildete Oberklasse.

1 Bottom-up

**Box: Checkliste
Vererbung**

Ergebnis

■ **Klassendiagramm**

Alle Vererbungsstrukturen in das Klassendiagramm eintragen. Abstrakte Klassen bilden.

Konstruktive Schritte

1 Ergibt sich durch eine Generalisierung eine Einfachvererbung (Bottom-up)?

- Gibt es gleichartige Klassen, aus denen sich eine Oberklasse bilden lässt?
- Ist eine vorhandene Klasse als Oberklasse geeignet?

2 Ergibt sich durch Spezialisierung eine Einfach- oder Mehrfachvererbung (Top-down)?

- Kann jedes Objekt der Klasse für jedes Attribut einen Wert annehmen?
- Kann jede Operation/Methode auf jedes Objekt der Klasse angewendet werden?
- Lassen sich für ursprüngliche Klassen Objekte erzeugen oder erfolgt dies nur noch für die Unterklassen?

Analytische Schritte

3 Liegt eine »gute« Vererbungsstruktur vor?

- Verbessert die Vererbungsstruktur das Verständnis des Modells?
- Benötigt jede Unterklasse alle geerbten Attribute, Operationen/Methoden und Assoziationen?
- Liegt eine Ist-ein-Beziehung vor?
- Entspricht die Vererbungsstruktur den »natürlichen« Strukturen des Problem-bereichs?
- Besitzt sie maximal drei bis fünf Hierarchiestufen?

4 Wann liegt keine Vererbung vor?

- Die Unterklassen bezeichnen nur verschiedene Arten, unterscheiden sich aber weder in ihren Eigenschaften noch in ihrem Verhalten.

Sollen für diese Oberklasse Objekte erzeugt werden oder dient sie nur dazu, Eigenschaften ihrer Unterklassen zusammenzufassen? Im zweiten Fall ist sie als abstrakte Klasse zu kennzeichnen.

2 Top-down

Bei der Top-down-Vorgehensweise gehen Sie von den allgemeineren Klassen aus und suchen nach spezialisierten Klassen. Betrachten Sie eine Klasse und prüfen Sie für jedes ihrer Objekte, ob dieses Objekt alle Attribute mit Werten besetzt und ob jede Operation angewendet werden kann. Müssen von der ursprünglichen Klasse nach dem Erstellen der Generalisierungsstruktur noch Objekte erzeugt werden? Wenn nein, dann ist diese Klasse als abstrakt zu kennzeichnen.

Beispiel:
SemOrg

In der Abb. 9.2-16 (links) besitzen die Attribute *kooperationspartner*, *teilnehmerzahlMin*, *teilnehmerzahlAktuell* und *stornogebuehr* nur dann einen Wert, wenn es sich um eine öffentliche Veranstaltung handelt, während *pauschalpreis* und *teinnehmerzahlMax* nur im Fall einer firmeninternen Veranstaltung Werte annehmen. Die Klasse *Veranstaltung* wird daher um die Unterklassen *Oeffentliche*

Veranstaltung und Firmeninterne Veranstaltung ergänzt. Da jede Veranstaltung entweder öffentlich oder firmenintern angeboten wird, ist die Klasse Veranstaltung als abstrakt zu kennzeichnen.

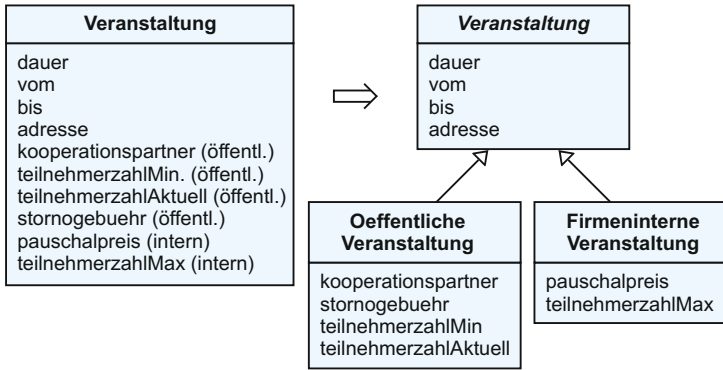


Abb. 9.2-16:
Spezialisierung
durch Top-down-
Vorgehen.

Analytische Schritte

Nur »gute« Generalisierungsstrukturen können ein Klassendiagramm verbessern. Das bedeutet:

- Jede Unterklasse soll die geerbten Attribute und Assoziationen der Oberklasse auch benötigen, d. h., jedes Objekt der Unterklasse belegt die geerbten Attribute mit Werten und kann entsprechende Objektbeziehungen besitzen. Diese Art der Modellierung führt zu tiefen Generalisierungsstrukturen. Hier ist kritisch abzuwägen, ob nicht Unterklassen Attribute und/oder Assoziationen besitzen können, die nicht von allen Objekten benötigt werden, und dadurch eine flachere Hierarchie erreicht wird.
- Eine »is a«-Beziehung liegt vor, d. h., ein Objekt der Unterklasse »ist ein« Objekt der Oberklasse. In der Abb. 9.2-16 gilt beispielsweise: Eine Oeffentliche Veranstaltung ist eine Veranstaltung. Würde sich für eine Generalisierungsstruktur dagegen ein Satz »Ein Mitarbeiter ist ein Kunde« formulieren lassen, dann liegt hier keine sinnvolle Generalisierung vor, auch wenn Mitarbeiter die gleichen Eigenschaften wie der Kunde besitzt. Die »is a«-Beziehung ist transitiv. Sie macht deutlich, dass es für eine gute Generalisierungsstruktur nicht ausreicht, wenn die Unterklasse zu den geerbten Attributen und Operationen eigene Attribute und Operationen hinzufügt.
- Die Vererbungshierarchie sollte nicht zu tief sein, denn um eine Unterklasse zu verstehen, müssen alle ihre Oberklassen betrachtet werden. Bis zu einer Tiefe von drei Ebenen gibt es normalerweise keine Verständnisprobleme, bei der Tiefe von fünf oder sechs Ebenen können bereits Schwierigkeiten auftreten.

3 »gute«
Vererbungs-
struktur

9.2.6 Assoziation

Während der Laufzeit eines Softwaresystems entstehen eine Vielzahl von Objekten mit einer Vielzahl von Beziehungen zwischen einzelnen Objekten, die auf- und auch wieder abgebaut werden. Während die UML es erlaubt, Assoziationen zu beschreiben, enthalten Programmiersprachen dafür keine originären Konstrukte. Assoziationen müssen auf Programmiersprachenebene in der Regel ausprogrammiert werden, wofür es mehrere Möglichkeiten gibt.



Klassen und **Assoziationen** werden in der UML im **Klassendiagramm** modelliert. Bestehen zwischen Objekten von Klassen Beziehungen, dann wird dies durch **Linien zwischen diesen Klassen** angegeben. Das Klassendiagramm beschreibt im Gegensatz zum Objektdiagramm *nicht* einen Schnappschuss, sondern die **grundsätzliche Struktur des Systems**.

Multiplizitäten

Ein Objekt kann zu *keinem* anderen Objekt, zu *genau einem* anderen Objekt oder zu *vielen anderen* Objekten eine Beziehung haben. Dieser Sachverhalt wird durch die sogenannten **Multiplizitäten** (*multiplicities*) der Assoziation beschrieben. Während die Assoziationslinie zwischen zwei Klassen zunächst nur aussagt, dass sich Objekte der beteiligten Klassen kennen können, spezifiziert die Multiplizität – auch Wertigkeit genannt – *wie viele* andere Objekte ein bestimmtes Objekt kennen kann oder muss.

Notation

Eine Assoziation wird durch eine Linie zwischen zwei Klassen beschrieben. An jedem Ende der Linie *muss* zusätzlich die Multiplizität angegeben werden.

Die Abb. 9.2-17 zeigt ein Beispiel für mögliche Multiplizitäten zwischen zwei Klassen, dargestellt in der UML-Notation.



Die Multiplizität, die zu einer Klasse A gehört, wird in der UML-Notation am Assoziationsende bei der Klasse B eingetragen und umgekehrt! Ein weiteres Beispiel zeigt die Abb. 9.2-18.

Wie die Beispiele zeigen, lassen sich Kann- und Muss-Assoziationen unterscheiden. Eine **Kann-Assoziation** hat als Untergrenze die Multiplizität 0, eine **Muss-Assoziation** die Multiplizität 1 oder größer. Die Notation * stellt dabei eine Abkürzung für 0..* dar. Die Abb. 9.2-19 zeigt weitere mögliche Multiplizitäten. Die Notation 1,3,5 besagt beispielsweise, dass eine, drei oder fünf Objektbeziehungen vorliegen müssen.

Tipp

Nur wenige Muss-Assoziationen

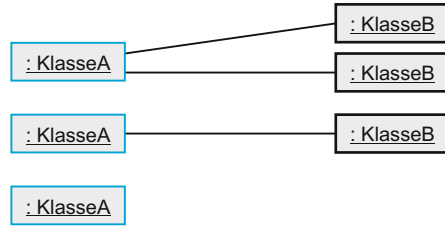
Die meisten Klassendiagramme enthalten *zu viele* Muss-Assoziationen. Im Zweifelsfalle sollten Sie daher immer mit einer Kann-Assoziation arbeiten.

9.2 Funktions-Strukturen III

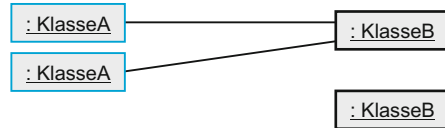


Objekte der Klasse **A** können Beziehungen zu 0, 1 oder mehreren Objekten der Klasse **B** haben (Notation: *).
Objekte der Klasse **B** müssen eine Beziehung zu genau einem Objekt der Klasse **A** haben (Notation: 1).

Mögliche Objektkonstellationen:



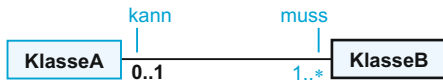
Unmögliche Objektkonstellationen:



Auf der Exemplarebene können in der UML Objekte in **Objektdiagrammen** dargestellt werden. Dabei können Assoziationsbeziehungen zwischen Objekten – man spricht dann oft von *Links* – analog wie im Klassendiagramm dargestellt werden. Multiplizitätsangaben entfallen allerdings.

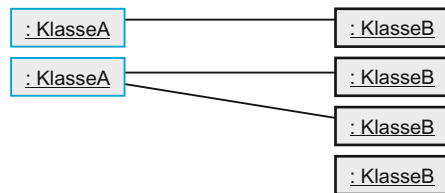
Assoziationen können benannt werden. Der **Assoziationsname** beschreibt im Allgemeinen nur eine Richtung der Assoziation, wobei ein schwarzes Dreieck die Leserichtung angibt (Abb. 9.2-20). Oft handelt es sich bei Assoziationsnamen um ein Verb. Der Assoziationsname kann fehlen, wenn die Bedeutung der Assoziation offensichtlich ist.

Abb. 9.2-17:
Beispiel für
mögliche
Multiplizitäten
zwischen zwei
Klassen.
Assoziations-
name



Objekte der Klasse **A** müssen zu einem oder mehreren Objekten der Klasse **B** Beziehungen haben (Notation: 1..*).*
Objekte der Klasse **B** können zu keinem oder zu genau einem Objekt der Klasse **A** eine Beziehung haben (Notation: 0..1).

Mögliche Objektkonstellationen:



Unmögliche Objektkonstellationen:



Während der Assoziationsname die Semantik der Assoziation beschreibt, enthält der sogenannte Rollename – kurz **Rolle** genannt – Informationen über die Bedeutung einer Klasse – bzw. ihrer Objekte – in der Assoziation. Der Rollename (Anfangsbuchstabe klein

Abb. 9.2-18:
Beispiel für
mögliche
Multiplizitäten.

III 9 Statik

geschrieben) wird jeweils an ein Ende der Assoziation geschrieben, und zwar bei der Klasse, deren Bedeutung in der Assoziation sie näher beschreibt. Die geschickte Wahl der Rollennamen kann zur Verständlichkeit des Klassenmodells mehr beitragen als der Name der Assoziation.

Von einer Klasse können natürlich Assoziationen zu mehr als einer anderen Klasse bestehen. In der Regel haben Klassen zu mehreren anderen Klassen Assoziationen.

Abb. 9.2-19:
Beispiele für
Multiplizitäten in
der UML-Notation.

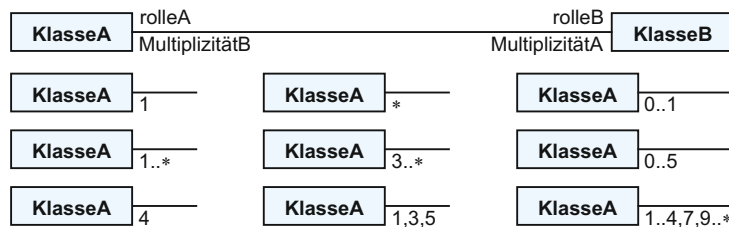
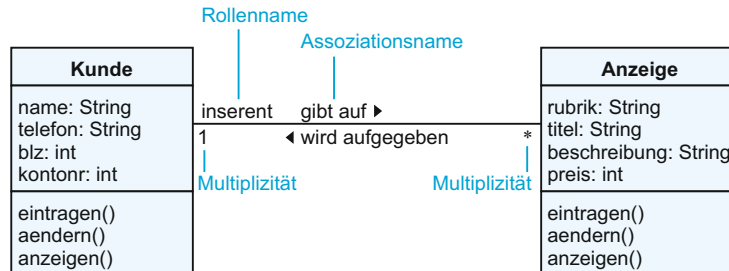


Abb. 9.2-20: Das
UML-
Klassendiagramm
zeigt die
Assoziation
zwischen den
Klassen Kunde und
Anzeige.



Beispiel Die Abb. 9.2-21 zeigt die Klasse **Mitarbeiter**, die zu den Klassen **Firma** und **PKW** eine Assoziation besitzt. Die Assoziationen sind wie folgt zu lesen:

- Eine Firma hat in ihrer Rolle als Arbeitgeber null, einen oder mehrere Mitarbeiter.
- Ein Mitarbeiter ist in seiner Rolle als Arbeitnehmer Mitglied genau einer Firma.
- Ein Mitarbeiter kann in seiner Rolle als Fahrer einen PKW fahren.
- Ein PKW kann in seiner Rolle als Dienstwagen von einem Mitarbeiter gefahren werden (kann Dienstwagen eines Mitarbeiters sein).

Abb. 9.2-21:
Beispiel für die
sinnvolle Wahl von
Rollennamen.



Reflexive Assoziationen

Es ist auch möglich, dass zwischen Objekten der gleichen Klasse eine Assoziation besteht. Man spricht dann von einer **reflexiven Assoziation**. Rollennamen sind bei reflexiven Assoziationen immer anzugeben, damit die Verständlichkeit gewährleistet ist.

Die Abb. 9.2-22 zeigt eine Klasse *Angestellter*, bei der es zwischen den Objekten eine Beziehung Chef – Mitarbeiter geben kann. Beispiel

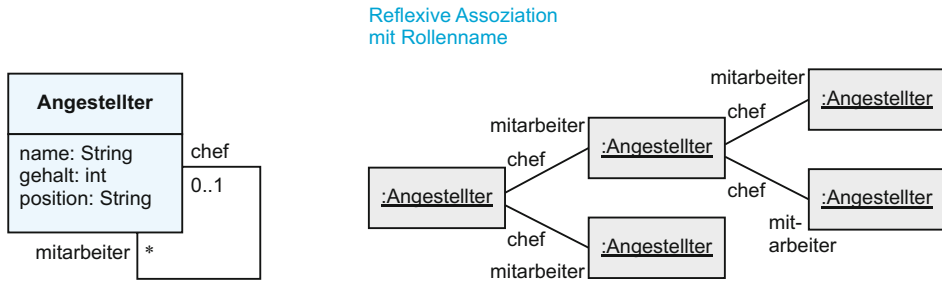


Abb. 9.2-22:
Beispiel für eine
reflexive
Assoziation und
eine mögliche Ob-
jekt-konstellation.

Mehrere Assoziationen zwischen zwei Klassen

In vielen Fällen gibt es zwischen zwei Klassen auch mehr als eine Assoziation. In diesen Fällen müssen Rollen- oder Assoziationsnamen angegeben werden.

Zwischen den Klassen *Sportler* und *Wettbewerb* (Abb. 9.2-23) gibt es zwei Assoziationen. Zwischen den Assoziationen gibt es noch die Randbedingung, dass die Sieger eine Teilmenge der Teilnehmer bilden (siehe auch »Constraints und die OCL in der UML«, S. 377). Beispiel

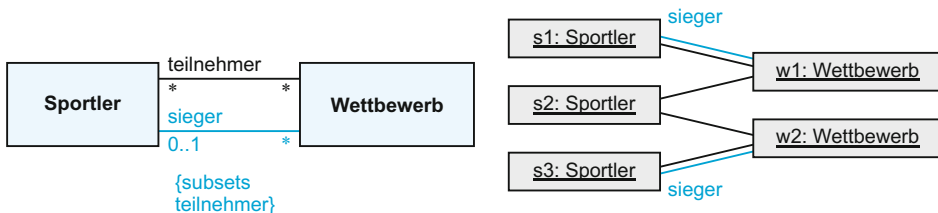


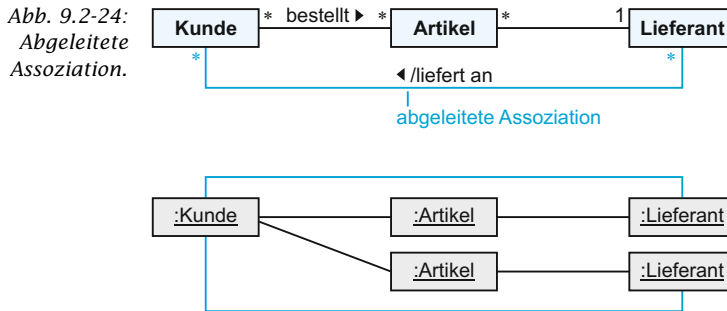
Abb. 9.2-23:
Darstellung von
zwei Assoziationen
zwischen zwei
Klassen, ergänzt
um eine mögliche
Objekt-
Konstellation.

Abgeleitete Assoziation

Eine Assoziation heißt abgeleitet (*derived association*), wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sie fügt keine neue Information zum Modell hinzu und ist daher redundant. Eine abgeleitete Assoziation wird durch das Präfix »/« vor dem Assoziationsnamen oder einen Rollennamen gekennzeichnet.

III 9 Statik

Beispiel Wie das Objektdiagramm der Abb. 9.2-24 zeigt, gibt es einen »direkten Weg« vom Kunden zum Lieferanten und einen »Umweg« über den Artikel, d. h. die Assoziation »/liefert an« wird durch die beiden anderen Assoziationen beschrieben.



Assoziationsklassen

Klasse &
Assoziation

Eine Assoziation kann zusätzlich die Eigenschaften einer Klasse besitzen, d. h. sie hat Attribute und Operationen sowie Assoziationen zu anderen Klassen. Zur Darstellung wird ein Klassensymbol verwendet, das über eine gestrichelte Linie mit der Assoziation verbunden wird (Abb. 9.2-25). Sie heißt **Assoziationsklasse** (*association class*).

Von der
Assoziations- zur
»normalen« Klasse

Durch diese Modellbildung mit einer Assoziationsklasse bleibt die ursprüngliche Assoziation zwischen den beteiligten Klassen bestehen und damit im Modell deutlich sichtbar. Beim Übergang zur objektorientierten Programmierung ist es notwendig, eine Assoziationsklasse in eine eigenständige Klasse und zwei Assoziationen aufzulösen. Diese Transformation erfolgt nach dem Schema der Abb. 9.2-25.

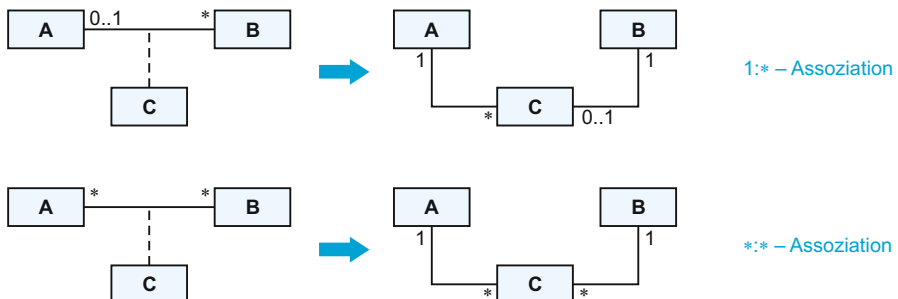


Abb. 9.2-25:
Auflösen einer
Assoziationsklasse.

Höherwertige Assoziationen

In den meisten Anwendungsfällen treten sogenannte binäre Assoziationen auf, d.h. Assoziationen zwischen den Objekten von jeweils zwei Klassen. Prinzipiell ist auch eine Assoziation möglich, die Objekte von drei oder mehr Klassen miteinander in Beziehung setzt. Liegt eine solche Assoziation vor, dann spricht man von **höherwertigen Assoziationen** oder **n-ären Assoziationen**. Höherwertige Assoziationen werden mit Hilfe eines Diamanten-Symbols modelliert, von dem die Linien zu den beteiligten Klassen ausgehen.

Die Abb. 9.2-26 modelliert, welcher Programmierer in welchem Projekt welche Programmiersprache verwendet. Konkrete Beziehungen zeigt die Tab. 9.2-1.

Beispiel

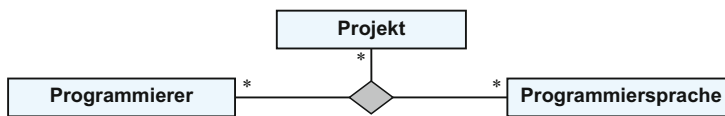


Abb. 9.2-26:
Beispiel für eine
ternäre
Assoziation.

Programmierer	Projekt	Programmierersprache
Meier	A	C++
Schröder	B	Java
Ludwig	A	Java

Tab. 9.2-1: Beispiel
für eine ternäre
Assoziation.

Die Abb. 9.2-27 modelliert, dass ein Fußballspieler innerhalb eines Jahres in verschiedenen Vereinen aktiv sein kann. Hier ist die ternäre Assoziation zusätzlich mit einer assoziativen Klasse verbunden. Beispielsweise kann für den Fußballer »Müller« festgehalten werden, welches Ergebnis er im Jahr 2006 für den Verein »FC 06« erzielt hat.

Beispiel

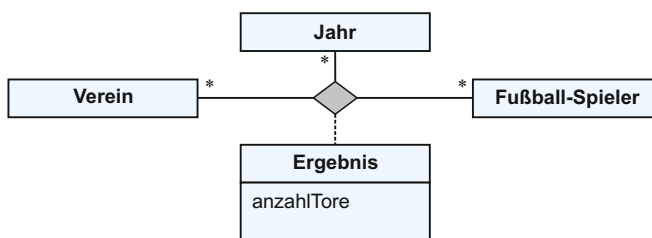


Abb. 9.2-27:
Beispiel für eine
ternäre
Assoziation mit
einer assoziativen
Klasse.

Navigierbarkeit

Assoziationen sind zunächst unspezifiziert. Die UML ermöglicht es, zusätzlich die **Navigierbarkeit** (*navigability*) einer Assoziation zu definieren. Besteht zwischen zwei Klassen A und B eine Assoziation und ist diese Assoziation von A nach B navigierbar, dann bedeu-

Assoziation mit
Richtung

III 9 Statik

tet dies, dass Objekte von A auf Objekte von B zugreifen können, aber *nicht* umgekehrt. In welchen Richtungen eine Assoziation navigierbar ist, wird im UML-Klassendiagramm durch Pfeile angegeben. Binäre Assoziationen können in eine oder in beide Richtungen navigierbar sein.

Unidirektional – nur ein Objekt kennt das andere

Beispiel 1a Ein Shop nimmt Bestellungen über Fax entgegen. Die Abb. 9.2-28 zeigt anhand einer Faxbestellung die Modellierung einer Bestellung.

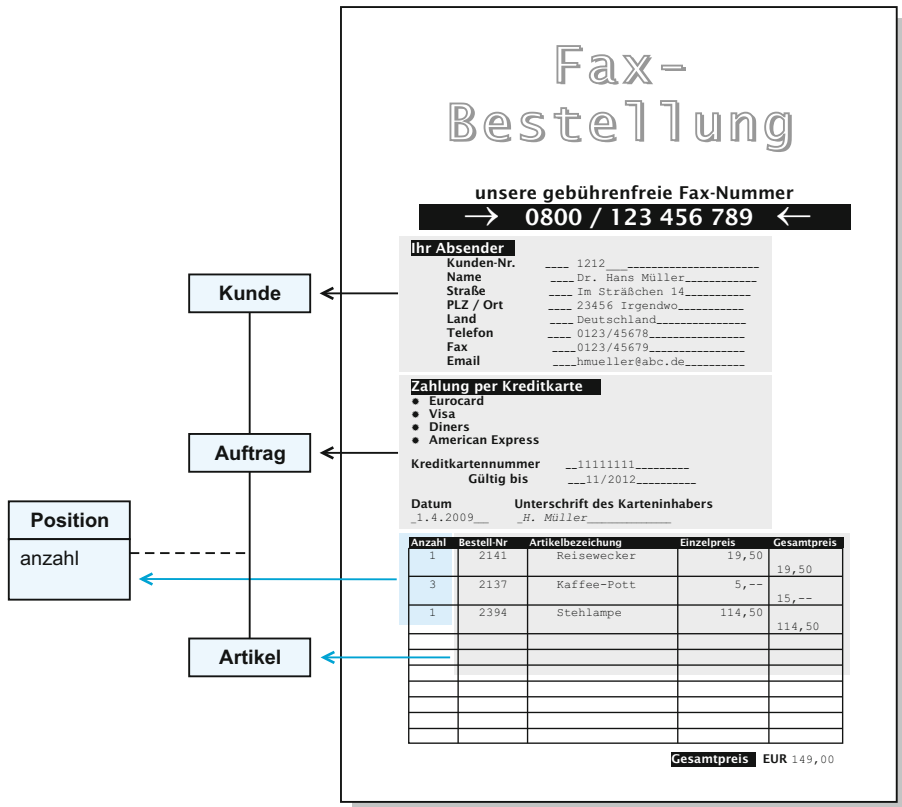


Abb. 9.2-28: Nach der Transformation der Assoziationsklasse, der Festlegung der Multiplizitäten und der Rollen ergibt sich das Modell der Abb. 9.2-29. Fachlich betrachtet muss von Position auf Artikel zugegriffen werden, aber *nicht* umgekehrt. Analog muss vom Auftrag auf die Positionen zugegriffen werden, jedoch *nicht* von den Positionen auf den Auftrag.

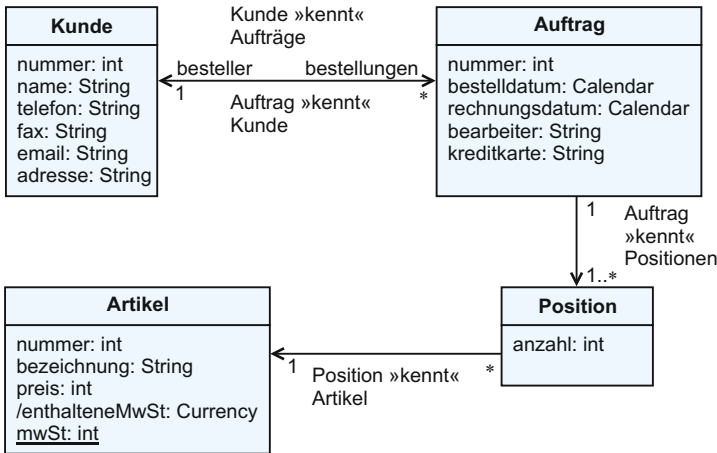


Abb. 9.2-29: Navigationsrichtung der Assoziationen im Shop.

Beziehungen, bei denen nur eine Navigationsrichtung bekannt sein muss, werden als **unidirektionale Assoziationen** bezeichnet und modelliert. Sie werden auch einseitige oder gerichtete Assoziationen genannt.

Bidirektional – beide Objekte kennen einander

Zwischen **Kunde** und **Auftrag** existiert eine bidirektionale Assoziation, da vom **Auftrag** ausgehend ein neuer **Kunde** (Besteller) erfasst oder ein vorhandener **Kunde** zugeordnet und umgekehrt zu jedem **Kunden** die Liste aller erteilten **Aufträge** (Bestellungen) angezeigt werden soll. Ein **Kunden**-Objekt kennt also seine **Auftrags**-Objekte und jedes **Auftrags**-Objekt kennt sein **Kunden**-Objekt.

Beispiel 1c

Bei einer **bidirektionalen Assoziation** können die entsprechenden Objektbeziehungen in beiden Richtungen durchlaufen werden. Sie ist gleichwertig zu zwei gerichteten Assoziationen (Abb. 9.2-30) (beide Assoziationen im unteren Diagramm tragen denselben Assoziationsnamen). Sie ist komplexer zu realisieren als eine unidirektionale Assoziation (wegen der Referenzverwaltung auf beiden Seiten und der Konsistenzhaltung) und sollte nur dort eingesetzt werden, wo sie wirklich benötigt wird.



Abb. 9.2-30: Bidirektionale Assoziation.

Notation Navigierbarkeit

Die Richtung, in der eine Assoziation realisiert werden muss, wird im UML-Klassendiagramm mit einer Pfeilspitze gekennzeichnet. Soll eine Assoziation in beiden Richtungen durchlaufen werden, werden beide Pfeilspitzen eingetragen. In der UML ist es auch möglich, Navigationsrichtungen explizit auszuschließen.

III 9 Statik

Abb. 9.2-31:
Navigierbarkeit
von Assoziationen.



Dies wird durch ein »x« auf der Linie gekennzeichnet. Ist nur eine Linie gezeichnet, dann ist die Assoziation unspezifiziert, d.h. über die Navigation wird noch nichts ausgesagt. Die Abb. 9.2-31 zeigt die verschiedenen Möglichkeiten zur Navigierbarkeit im Überblick.

Hinweis

Viele Notationselemente der Assoziation können auch in **Objektdiagrammen** verwendet werden, um dessen Aussagegehalt zu steigern. Wird der Assoziationsname an die Objektverbindung angetragen, dann muss er unterstrichen werden.

Struktur

Durch Assoziationen entstehen **gerichtete Graphen**.

Zur Klassifikation

- Assoziationen werden in der Regel grafisch (mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität ist semiformal bis formal.
- Sie werden in der Spezifikations- und der Entwurfsphase eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet – schwerpunktmäßig aber in kaufmännisch/administrativen Anwendungen.

9.2.7 Box: Assoziationen – Methode und Checkliste

Zum Identifizieren und Überprüfen von Assoziationen – insbesondere um sie in ein UML-Klassendiagramm einzutragen – helfen die Boxen »Checkliste Assoziationen«. Auf einige Aspekte zu den Boxen wird im Folgenden eingegangen.

Konstruktive Schritte

- 1 Anforderungsspezifikation Liegt bereits eine **Anforderungsspezifikation** (z.B. Lasten- oder Pflichtenheft) vor, dann kann aus natürlichsprachlichen Anforderungen auf Assoziationen geschlossen werden.

Beispiel:
SemOrg

Im Pflichtenheft befindet sich folgende Anforderung:
/F50/ (/LF50/) Das System *muss* dem Seminarsachbearbeiter die Möglichkeit bieten, neue Seminare und Veranstaltungen zu erfassen, vorhandene zu modifizieren oder zu löschen und Veranstaltungen Seminaren *zuzuordnen*.

Ergebnis

■ Klassendiagramm

Assoziationen im ersten Schritt nur als Linie eintragen. Noch *keine* Multiplizitäten, Aggregationen, Kompositionen, Rollen, Namen, Einschränkungen (*constraints*).

Konstruktive Schritte

1 Welche Assoziationen lassen sich aus der Anforderungsspezifikation (Lasten-/Pflichtenheft) und aus den Use Cases ermitteln?

In den Beschreibungen nach Verben suchen, insbesondere:

- a Räumliche Nähe (in der Nähe von),
- b Aktionen (fährt),
- c Kommunikation (redet mit, verheiratet mit),
- d Besitz (hat),
- e allgemeine Beziehungen (zuordnen).

2 Welche Assoziationen lassen sich mittels Dokumentanalyse ableiten?

- Aus Primär- und Fremdschlüsseln ermitteln.

3 Liegt eine Assoziation der folgenden Kategorien vor?

- A ist physische Komponente von B.
- A ist logische Komponente von B.
- A ist eine Beschreibung für B.
- A ist eine Zeile einer Liste B.
- A ist ein Mitglied von B.
- A ist eine organisatorische Einheit von B.
- A benutzt B.
- A kommuniziert mit B.
- A besitzt B.

Diese Kategorien können beim Modellieren dazu beitragen, konkretere Fragen zu stellen und mehr Informationen über das Analysemodell zu gewinnen.

4 Welche Einschränkungen muss die Assoziation erfüllen?

- Eine Assoziation: {ordered}.
- Mehrere Assoziationen: {xor}, {subsets Eigenschaft}.

Zur Überprüfung und Veranschaulichung Objektdiagramme erstellen.

5 Welche Rollen spielen die beteiligten Klassen?

Je allgemeiner der Klassenname, desto wichtiger der Rollename! Rollennamen angeben, wenn

- 1 die Assoziation zwischen Objekten derselben Klasse existiert,
- 2 eine Klasse in verschiedenen Assoziationen auch verschiedene Rollen spielt,
- 3 durch den Rollennamen die Bedeutung der Klasse in der Assoziation genauer spezifiziert werden kann.

Box 1a:

Checkliste

Assoziationen

Das Verb *zuordnen* gibt einen Hinweis auf eine Assoziation zwischen Veranstaltung und Seminar.

Das folgende Beispiel zeigt eine Zuordnung zu Kategorien.

3 Kategorien

Assoziationen der Seminarorganisation lassen sich folgenden Kategorien zuordnen:

Beispiel:
SemOrg

- A ist logische Komponente von B: Veranstaltung zu Seminartyp.
- A ist ein Mitglied von B: Kunde zu Firma.

III 9 Statik

4 Einschränkungen Um Einschränkungen zu ermitteln, die zwei oder mehrere Assoziationen betreffen, ist die Erstellung von Objektdiagrammen nützlich.

Beispiel: Aus dem Pflichtenheft ist nicht ersichtlich, ob ein Seminarleiter gleichzeitig auch Referent auf derselben Veranstaltung sein muss. Eine Rückfrage beim Auftraggeber ergibt, dass diese Restriktion gilt: Der Seminarleiter muss Referent sein, aber außer ihm kann es noch andere Referenten geben.

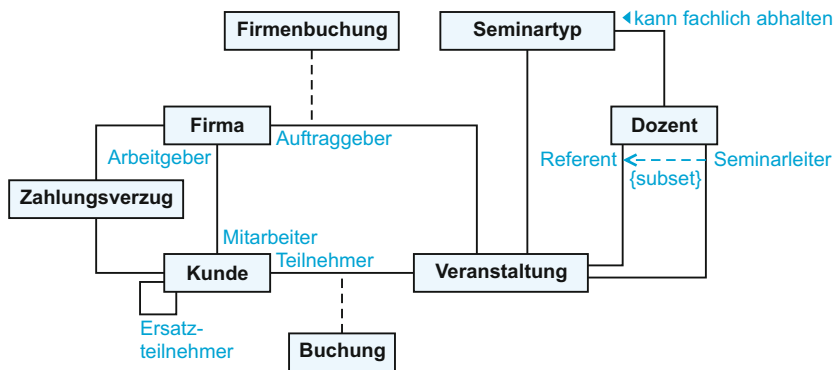
5 Rollen

Beispiel: Folgende Rollen lassen sich ermitteln:

- SemOrg
- Die Klasse Dozent spielt im Bezug auf Seminarveranstaltungen verschiedene Rollen: Referent und Seminarleiter.
 - In der Assoziation Kunde – Firma verdeutlicht die Rolle Mitarbeiter die Rolle der Klasse Kunde und Arbeitgeber die Rolle der Klasse Firma.
 - In der Assoziation Kunde – Veranstaltung verdeutlicht die Rolle Teilnehmer die Rolle der Klasse Kunde.
 - In der Assoziation Firma – Veranstaltung verdeutlicht die Rolle Auftraggeber die Rolle der Klasse Firma.

Der Zusammenhang zwischen Dozent und Seminartyp wird durch den Assoziationsnamen kann fachlich abhalten verständlicher. Die Rollen und Assoziationsnamen sind in der Abb. 9.2-33 blau eingetragen.

Abb. 9.2-33:
Klassen,
Assoziationen und
Rollen der »Semi-
narorganisation«.



Analytische Schritte

6 Assoziations- & Rollennamen

Assoziationen sollen benannt werden, wenn ihre Bedeutung nicht offensichtlich ist. Assoziationsnamen werden zentriert an die Linie geschrieben, Rollennamen in die Nähe der Klasse, die sie genauer spezifizieren. Mit einem einfachen Test können Sie prüfen, ob ein guter Assoziationsname gewählt wurde. Bilden Sie mit den beteiligten Klassen und dem Assoziationsnamen einen Satz.

Analytische Schritte

6 Ist ein Assoziations- oder Rollename notwendig oder sinnvoll?

- Namen sind notwendig, wenn zwischen zwei Klassen mehrere Assoziationen bestehen.
- Rollennamen sind gegenüber Assoziationsnamen zu bevorzugen.
- Rollennamen sind bei reflexiven Assoziationen immer notwendig.
- Rollennamen sind Substantive, Assoziationsnamen enthalten Verben.
- Test für Assoziationsname: »Klasse1 Assoziationsname Klasse2« ergibt einen sinnvollen Satz.
- Rollename: Welche Rolle spielt die Klasse X gegenüber einer Klasse Y?

7 Liegt eine 1:1-Assoziation vor?

Zwei Klassen sind zu modellieren, wenn

- die Objektbeziehung in einer oder beiden Richtungen optional ist und sich die Objektbeziehung zwischen beiden Objekten ändern kann,
- es sich um zwei umfangreiche Klassen handelt,
- die beiden Klassen eine unterschiedliche Semantik besitzen.

8 Existieren zwischen zwei Klassen mehrere Assoziationen?

Prüfen Sie, ob die Assoziationen

- eine unterschiedliche Bedeutung besitzen oder/und
- unterschiedliche Multiplizitäten haben.

9 Sind abgeleitete Assoziationen korrekt verwendet?

- Abgeleitete Assoziationen fügen keine neue Information zum Modell hinzu.
- Sie lassen sich leicht mittels Objektdiagrammen erkennen.

10 Soll eine assoziative Klasse oder eine eigenständige Klasse modelliert werden?

- Assoziative Klasse betont die Assoziation zwischen den beteiligten Klassen.
- Assoziative Klassen lassen sich in eigenständige Klassen wandeln.

11 Fehlerquellen

- Verwechseln von Assoziation mit Vererbung.

Box 1b:

Checkliste

Assoziationen

Besonders gut gewählte Rollennamen tragen viel zur Lesbarkeit eines Modells bei. Um Rollennamen zu finden, wählen Sie eine Klasse, z. B. Firma, und prüfen Sie für jede Assoziation, die von ihr ausgeht: Welche Rolle spielt die Klasse? Fassen Sie Rollen einer Klasse analog zu den Rollen einer Person im täglichen Leben auf, z. B. Person spielt die Rollen Mitarbeiter (gegenüber Chef), Vorgesetzter (gegenüber eigenen Mitarbeitern) und Vater (gegenüber Kindern).

Besteht zwischen zwei Klassen eine 1:1-Assoziation, dann ist zu prüfen, ob eine Zusammenfassung sinnvoll ist oder nicht.

7 1:1-Assoziation

9.2.8 Box: Multiplizitäten – Methode und Checkliste

Zum Identifizieren und Überprüfen von Multiplizitäten hilft die Box »Checkliste Multiplizitäten«. Auf einige Aspekte zu der Box wird im Folgenden eingegangen.

Box: Checkliste Multiplizitäten

Ergebnis

■ Klassendiagramm

Alle Multiplizitäten in das Klassendiagramm eintragen.

Konstruktive/analytische Schritte

1 Ist ein Schnappschuss oder ist die Historie zu modellieren?

Aus den Anfragen an das System ergibt sich, ob

- ein Schnappschuss (1- bzw. 0..1-Multiplizität) oder
- die Historie (*many*-Multiplizität) zu modellieren ist.

2 Liegt eine Muss- oder Kann-Assoziation vor?

- Bei einer einseitigen Muss-Assoziation (Untergrenze ≥ 1 auf einer Seite) gilt: Sobald das Objekt A erzeugt ist, muss auch die Beziehung zu dem Objekt B aufgebaut und B vorhanden sein bzw. erzeugt werden.

- Bei einer wechselseitigen Muss-Assoziation (Untergrenze ≥ 1 auf beiden Seiten) gilt:

Sobald das Objekt A erzeugt ist, muss auch die Beziehung zu dem Objekt B aufgebaut und ggf. das Objekt B erzeugt werden. Wenn das letzte Objekt A einer Beziehung gelöscht wird, dann muss auch Objekt B gelöscht werden.

- Bei einer Kann-Beziehung (Untergrenze = 0) kann die Beziehung zu einem beliebigen Zeitpunkt nach dem Erzeugen des Objekts aufgebaut werden.

3 Enthält die Multiplizität feste Werte?

- Ist eine Obergrenze vom Problembereich her zwingend vorgegeben (z. B. maximal 6 Spieler)? Im Zweifelsfall mit variablen Obergrenzen arbeiten.
- Ist die Untergrenze vom Problembereich her zwingend vorgegeben (z. B. mindestens 2 Spieler)? Im Zweifelsfall mit »0« arbeiten.
- Gelten besondere Einschränkungen für die Multiplizitäten (z. B. eine gerade Anzahl von Spielern)?

4 Fehlerquelle

- Oft werden Muss-Assoziationen verwendet, wo sie nicht benötigt werden.

Konstruktive/analytische Schritte

- | | |
|-----------------------------|---|
| 1 Schnappschuss? | Die Frage, ob ein Schnappschuss oder die Historie zu modellieren ist, lässt sich am einfachsten beantworten, indem Sie Anfragen formulieren. Die Fragestellung »Wer hat sich zurzeit den roten Mercedes ausgeliehen?« weist auf einen Schnappschuss hin und die Frage »Welche Personen haben den roten Mercedes im letzten Jahr ausgeliehen? « zeigt, dass die Historie modelliert werden muss. Während bei einem Schnappschuss eine alte Objektbeziehung gelöst wird, bevor eine neue aufgebaut wird, wird bei der Historie eine neue Objektbeziehung zwischen den jeweiligen Objekten hinzugefügt. Auch hier können Sie Objektdiagramme nutzbringend einsetzen. |
| 2 Muss- oder Kann-Beziehung | Bei der Überlegung, ob eine Muss- oder Kann-Assoziation vorliegt, sollten Sie folgende Fragen stellen: <ul style="list-style-type: none"> ■ Wie und wann werden Objekte der Klassen erzeugt? ■ Können beteiligte Objekte gelöscht werden und welche Konsequenzen hat dies? |
| 3 Feste Grenzen | Bei technischen Systemen liegt oft eine vom Problem her fest vorgegebene Anzahl von Objekten vor. Ist vom Problem her keine zwingende Obergrenze vorgegeben, so sollte die variable Obergrenze |

many gewählt werden. Bei Informationssystemen liegt meistens eine unbestimmte Anzahl von Objekten vor. Hier sollten Sie eine variable Anzahl auch dann wählen, wenn der Auftraggeber eine willkürliche Obergrenze angibt.

9.2.9 Aggregation und Komposition

Eine Assoziation besteht zwischen gleichrangigen Klassen. Neben der Assoziation können aber noch zwei Spezialfälle auftreten, die zu einer »hierarchischen« Assoziation führen:

- Aggregation und
- Komposition.

Eine **Aggregation** (*aggregation*) liegt vor, wenn zwischen den Objekten der beteiligten Klassen (kurz: den beteiligten Klassen) eine Rangordnung gilt, die sich durch »ist Teil von« bzw. »besteht aus« beschreiben lässt. Man spricht auch vom Ganzen und seinen Teilen (*whole-part*).

Aggregation =
Anhäufung,
Zusammenhang

Die Objekte der Aggregation bilden einen **gerichteten azyklischen Graphen**. Das bedeutet: Wenn B Teil von A ist, dann darf A nicht Teil von B sein. *Shared aggregation (weak ownership)* bedeutet, dass ein Teilobjekt mehreren Aggregatobjekten zugeordnet werden kann. Das entsprechende Objektdiagramm bildet eine Netzstruktur.

Struktur

Eine **Komposition** (*composition, composite aggregation*) ist eine starke Form der Aggregation. Auch hier muss eine »ist Teil von«-Beziehung vorliegen und die Objekte formen einen **gerichteten azyklischen Graphen**. Darüber hinaus muss gelten:

Komposition

- Jedes Objekt der Teilklassse kann – zu einem Zeitpunkt – nur Komponente eines einzigen Objekts der Aggregatklasse sein, d. h. die bei der Aggregatklasse angetragene Multiplizität darf nicht größer als eins sein (*unshared aggregation, strong ownership*). Ein Teil darf jedoch – zu einem anderen Zeitpunkt – auch einem anderen Ganzen zugeordnet werden.
- Die dynamische Semantik des Ganzen gilt auch für seine Teile (*propagation semantics*). Wird beispielsweise das Ganze kopiert, so werden auch seine Teile kopiert.
- Wird das Ganze gelöscht, dann werden automatisch seine Teile gelöscht (*they live and die with it*). Ein Teil darf jedoch zuvor explizit entfernt werden.

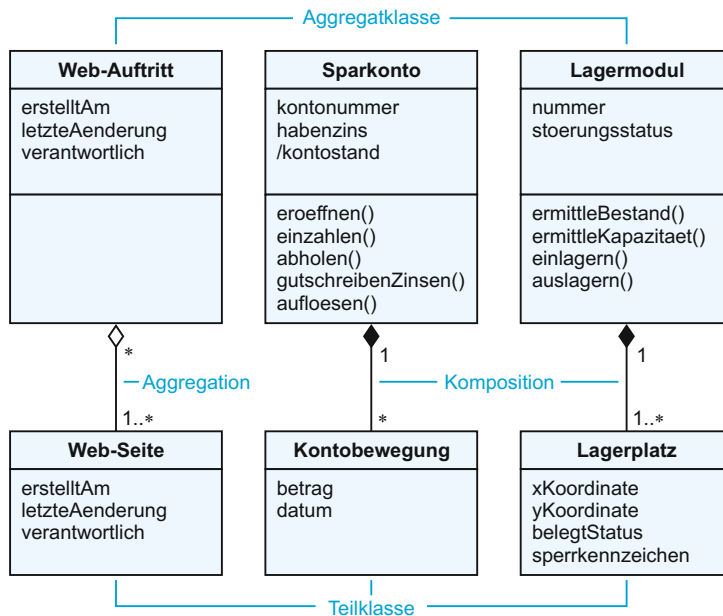
In beiden Fällen kennzeichnet in der UML eine Raute das Ganze. Bei einer Aggregation ist es eine weiße bzw. transparente, bei der Komposition eine schwarze bzw. gefüllte Raute. Alle anderen Angaben (Multiplizitäten, Namen, Rollen, Restriktionen usw.) werden analog zur Assoziation angegeben.

Notation

III 9 Statik

Beispiel In der Abb. 9.2-36 können einem Web-Auftritt mehrere Webseiten zugeordnet sein. Jede Webseite kann in mehreren Web-Auftritten referenziert werden. Es liegt daher eine *shared aggregation* vor. Die Mitte der Abbildung modelliert ein Sparkonto, zu denen es jeweils mehrere Kontobewegungen gibt, wobei jede Kontobewegung nur einem Sparkonto zugeordnet sein kann. Wird das Sparkonto-Objekt kopiert, dann werden auch alle ihm zugeordneten Kontobewegungen kopiert. Auf der rechten Seite der Abbildung ist ein Lagermodul eines Regallagers modelliert. Jedem Lagermodul sind mehrere Lagerplätze zugeordnet. Jeder Lagerplatz ist umgekehrt genau Teil eines Lagermoduls. In den beiden letzten Fällen liegt jeweils eine *Komposition* vor.

Abb. 9.2-36:
Aggregation vs.
Komposition.



Abgrenzung Assoziation vs. Aggregation vs. Komposition

Die Abgrenzung zwischen der »einfachen« Assoziation, der Aggregation und der Komposition ist in der Praxis oft schwierig. Viele Methoden unterscheiden nur zwischen »einfacher« Assoziation und Aggregation. Einige Autoren, z.B. [Fowl97a], verwenden nur die »einfache« Assoziation, um die Abgrenzungsproblematik zu vermeiden. [BeMa93] diskutieren verschiedene Definitionsmöglichkeiten für eine Aggregation. Die Abb. 9.2-37 zeigt, wie diese verschiedenen Definitionen auf die UML abgebildet werden.

- Ein Teil-Objekt darf nur zu einem Aggregat-Objekt gehören (*exclusive*). Kommentar: Diese Einschränkung ist zu eng, da sie die *shared aggregation* ausschließt.
- Das Teil-Objekt darf nicht vor dem Aggregat-Objekt erzeugt werden. Kommentar: Diese Einschränkung führt zu Problemen, wenn ein Teil-Objekt bereits existiert und einem Ganzen zugeordnet werden soll.
- Wenn das Aggregat-Objekt gelöscht wird, dann müssen alle Teil-Objekte ebenfalls gelöscht werden. Kommentar: Diese Einschränkung ermöglicht es *nicht*, Teil-Objekte für ein neues Aggregat-Objekt zu verwenden. Es ist daher sinnvoll, zwischen abhängigen (*dependent*) und nicht-abhängigen (*independent*) Teil-Objekten zu unterscheiden. Wird das Aggregat-Objekt gelöscht, dann werden nur die abhängigen Teil-Objekte gelöscht.

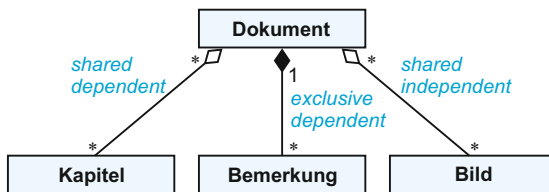


Abb. 9.2-37:
Verschiedene
Möglichkeiten zur
Definition einer
Aggregation/
Komposition.

[CoYo91] unterscheiden die folgenden Aggregationsstrukturen (*whole part*):

- Das Ganze und seine Teile. Beispiel: Der PKW (Ganzes) und sein Motor (Teil).
- Der Behälter und sein Inhalt. Beispiel: Das Flugzeug (Behälter) und sein Pilot (Inhalt).
- Die Kollektion und ihre Mitglieder. Beispiel: Firma (Kollektion) und Angestellte (Mitglieder).

[Odel94] unterscheidet sechs verschiedene Arten der Aggregation (*composition*):

- Konfiguration von Teilen in einem Ganzen (*component-integral object composition*). Sie definiert, aus welchen Teilen ein Objekt besteht und ist die häufigste Art der Aggregation. Teil-Objekte dürfen entfernt werden. Beispiele: Szenen sind Teile eines Films. Räder sind Teile eines Autos.
- Invariante Konfiguration von Teilen in einem Ganzen (*materialobject composition*). Diese Aggregation definiert, »aus was ein Objekt gemacht ist«. Hier dürfen Teil-Objekte nicht entfernt werden. Beispiele: Ein Baum besteht teilweise aus Holz. Ein Auto besteht teilweise aus Blech.
- Bei der Gleichartigkeit von Teilen und Ganzem (*portion-object composition*) sind die Teile im Prinzip dasselbe wie das Ganze. Beispiele: Ein Meter ist ein Teil eines Kilometers. Eine Brotscheibe ist Teil eines Brotlaibs.

III 9 Statik

- Invariante und gleichartige Konfiguration von Teilen in einem Ganzen (*place-area composition*). Die Teile können nicht von dem Ganzen getrennt werden. Beispiele: München ist ein Teil von Bayern. Ein Gipfel ist Teil eines Berges.
- Kollektion von Teilen in einem Ganzen (*member-bunch composition*). Beispiele: Ein Student ist Teil einer Universität. Ein Schiff ist Teil einer Flotte.
- Invariante Kollektion von Teilen in einem Ganzen (*memberpartnership composition*). Wird ein Mitglied entfernt, so wird auch das Ganze zerstört. Beispiel: Stan Laurel ist Teil von »Laurel und Hardy«.

Hinweis Auf der Programmierenebene unterscheiden sich Assoziation, Aggregation und Komposition nicht. Auf der Modellierungsebene erhöht die Unterscheidung aber die Verständlichkeit des Modells.

Exemplarebene Aggregationen und Kompositionen können auch in UML-Objektdiagramme eingetragen werden.

Hinweis Ternäre und höhere Assoziationen können keine Aggregation oder Komposition bilden.

CRC-Karten

CRC-Karten (*Class/Responsibility/Collaboration*-Karten) wurden erstmalig von Beck und Cunningham als Hilfsmittel für die Ausbildung in der objektorientierten Programmierung eingeführt. Sie sind ein wesentlicher Bestandteil der Methode von [WWW90]. Inzwischen sind CRC-Karten eine weit verbreitete Technik, die in zahlreiche objektorientierte Methoden integriert wurde.

Eine **CRC-Karte** ist eine Karteikarte. Oben auf der Karte wird der Name der Klasse (*class*) eingetragen. Die restliche Karte wird in zwei Hälften geteilt. Auf der einen Hälfte werden die Verantwortlichkeiten (*responsibilities*) der Klasse notiert. Darunter sind sowohl das Wissen der Klasse als auch die zur Verfügung gestellten Operationen zu verstehen. Ein Objekt der beschriebenen Klasse kann seine Aufgabe selbst erfüllen oder es kann hierzu die Hilfe anderer Objekte in Anspruch nehmen. Die dafür notwendigen Klassen (*collaborations*) werden auf der anderen Kartenseite eingetragen.

CRC-Karten sind nicht als Alternative, sondern als Ergänzung zum OOA-Modell zu verstehen (siehe »Beispiel: Objektorientierte Analyse«, S. 548). Wie die Abb. 9.2-38 zeigt, werden die Informationen auf einer CRC-Karte auf einer höheren Abstraktionsebene dargestellt als im Klassendiagramm. Die ermittelten Klassen bilden immer einen Stapel von Karteikarten und können je nach Verwendungszweck entsprechend angeordnet werden. Zur Modellierung der dynamischen

Aspekte werden die Karten so angeordnet, dass sie den Botschaftenfluss aufzeigen. Bei der Darstellung des statischen Modells werden die Karten entsprechend den Vererbungsstrukturen und Aggregat-Hierarchien angeordnet [Booc94].

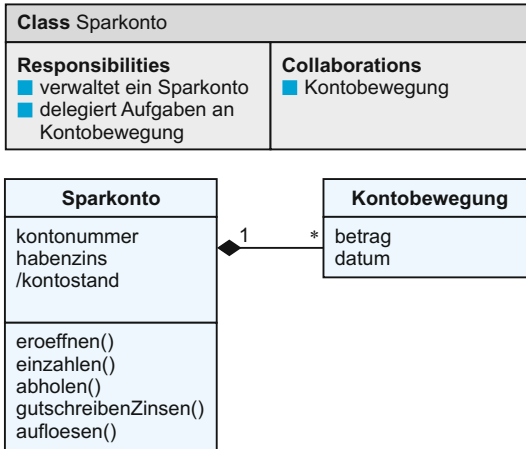


Abb. 9.2-38: CRC-Karte vs. UML-Klassendiagramm.

Zur Klassifikation

- Aggregation und Komposition werden grafisch (mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden vor allem in der Spezifikationsphase eingesetzt.
- Sie werden vor allem in kaufmännisch/administrativen Anwendungsbereichen verwendet.

9.2.10 Box: Komposition und Aggregation – Methode und Checkliste

Zum Identifizieren und Überprüfen von Kompositionen und Aggregationen hilft die Box »Checkliste Kompositionen & Aggregationen«. Auf einige Aspekte zu der Box wird im Folgenden eingegangen.

Konstruktive/analytische Schritte

Wählen Sie eine **Komposition**, wenn folgende Bedingungen eindeutig erfüllt sind: **1** Komposition

- Die Beziehung kann durch »besteht aus« oder »ist enthalten in« beschrieben werden (*whole part*), z. B.: Ein Auftrag besteht aus Auftragspositionen.
- Die Multiplizität der Aggregatklasse darf nicht größer als eins sein (*unshared aggregation, strong ownership*).

**Box: Checkliste
Kompositionen &
Aggregationen**

Ergebnis

■ **Klassendiagramm**

Alle Aggregationen und Kompositionen in das Klassendiagramm eintragen.

Konstruktive/analytische Schritte

1 Für eine Komposition gilt:

- Es liegt eine Ist-Teil-von-Beziehung vor.
- Die Multiplizität bei der Aggregatklasse beträgt 0..1 oder 1 (*unshared aggregation, strong ownership*).
- Die Lebensdauer der Teile ist an die des Ganzen gebunden. Ein Teil darf jedoch zuvor explizit entfernt werden.
- Die Funktionen des Ganzen werden automatisch auf seine Teile angewendet.
- Muster bilden eine gute Orientierungshilfe (siehe »OOA-Muster«, S. 550).
Dazu gehören
 - ☐ Liste (Bestellung – Bestellposition),
 - ☐ Baugruppe (Auto – Motor) und
 - ☐ Stückliste mit physikalischem Enthaltensein (Verzeichnis – Verzeichnis).

2 Für eine Aggregation gilt:

- Es liegt eine Ist-Teil-von-Beziehung mit *shared aggregation* (ein Teilobjekt kann mehreren Aggregatobjekten zugeordnet werden) vor.
- Sie ist selten.

3 Im Zweifelsfall immer eine einfache Assoziation verwenden.

4 Fehlerquelle

- Prinzipiell ist es möglich, jedes Attribut als Klasse zu modellieren und mittels einer Komposition mit der ursprünglichen Klasse zu verbinden. Dies führt jedoch zu schlechten Modellen.

- Wird das Ganze gelöscht, dann werden automatisch seine Teile gelöscht (*they live and die with it*). Ein Teil darf jedoch zuvor explizit entfernt werden.
- Das Ganze ist verantwortlich für das Erzeugen seiner Teil-Objekte.

2 Aggregation Die Aggregation kommt relativ selten vor. Da ihre Anwendung in der Praxis wegen der unpräzisen Definition problematisch ist und die Modellierung genauso gut mit einer »einfachen« Assoziation erfolgen kann, sollte auf dieses Konstrukt lieber verzichtet werden. Die folgenden Beispiele zeigen, in welchen Fällen die Aggregation angewendet werden kann.

Beispiel Die Abb. 9.2-40 modelliert die Stücklistenproblematik, bei der physisches und logisches Enthaltensein zu unterschiedlichen Modellen führen.

Beispiel Die Abb. 9.2-41 zeigt ein Beispiel, wo Komposition und Aggregation nebeneinander verwendet werden. Jede Klasse kann in höchstens einem Paket enthalten sein. Sie kann jedoch in mehreren – anderen – Paketen referenziert werden.

9.2 Funktions-Strukturen III

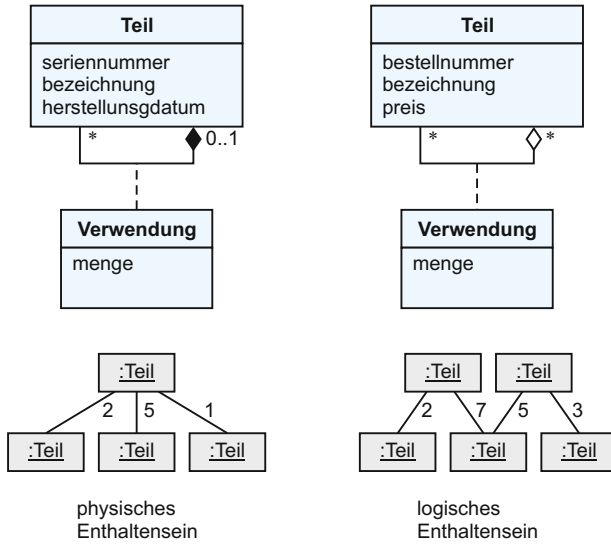


Abb. 9.2-40:
Physisches und
logisches
Enthaltensein.

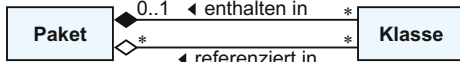


Abb. 9.2-41:
Komposition und
Aggregation.

9.2.11 Weitere Strukturen

Komponentendiagramm

Durch die zur Verfügung gestellten und benutzten Schnittstellen sind die UML-Komponenten miteinander verbunden (siehe »Funktionalität«, S. 127). Diese Abhängigkeiten werden im **Komponentendiagramm** (*component diagram*) dokumentiert.



Ein Browser, der mit Plug-ins arbeiten soll, definiert die benötigte Schnittstelle, die in der Abb. 9.2-42 durch einen Halbkreis dargestellt wird. Damit eine Komponente, z.B. QuickTime, mit dem Browser konnektiert werden kann, muss sie diese Plug-in-Schnittstelle realisieren.

Beispiel

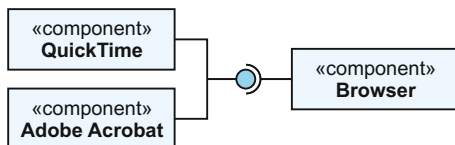


Abb. 9.2-42:
Einfaches Kompo-
nentendiagramm.

Ein weiteres Element im Komponentendiagramm ist das **Artefakt** (*artefact*), das eine physische Informationseinheit auf Exemplartyp-ebene darstellt, z. B. ein Modell, eine Quellcode-Datei, eine ausführbare Binärdatei oder eine Archivdatei.

Artefakt

III 9 Statik

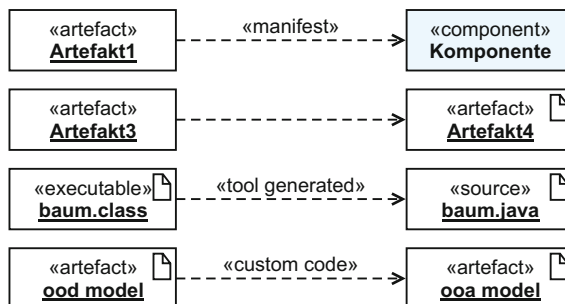
Notation Artefakte werden in der Rechteck-Notation dargestellt und mit dem Stereotypen «artifact» gekennzeichnet. Alternativ oder zusätzlich kann das Artefakt-Symbol eingetragen werden. Außerdem muss der Name des Artefakts, z. B. ein Dateiname, angegeben werden. Diese Namen werden stets unterstrichen.

Durch einen weiteren Stereotypen kann die Art des Artefakts genauer spezifiziert werden. Die UML enthält dafür einige vordefinierte Stereotypen:

- «file»: Datei.
- «document»: Datei, die weder Quellcode noch ausführbaren Programmcode enthält.
- «source»: Datei, die Quellcode (für einen Compiler oder Interpreter) enthält.
- «executable»: Datei, die direkt ausführbaren Programmcode enthält (Maschinencode, Bytecode).
- «library»: Datei, die eine Bibliothek enthält (z. B. dll-Datei).

Zwischen einem Artefakt und einer Komponente und auch zwischen zwei Artefakten können Abhängigkeiten existieren, die durch gestrichelte Pfeile modelliert werden. Die Abb. 9.2-43 zeigt die Notation für verschiedene Artefakte und einige Abhängigkeiten.

Abb. 9.2-43:
Notation für
Artefakte und
deren
Abhängigkeiten.



Der Stereotyp «manifest» verbindet ein Artefakt und eine Komponente. Diese Abhängigkeit sagt aus, dass Artefakt1 durch die Komponente realisiert bzw. manifestiert wird. Zwischen Artefakt3 und Artefakt4 besteht eine nicht näher spezifizierte Abhängigkeit. Die Pfeilrichtung besagt, dass Artefakt3 zur Erfüllung seiner Aufgaben Artefakt4 benötigt. Bei baum.class und baum.java handelt es sich um Artefakte einer bestimmten Art (siehe oben). Der Stereotyp «tool generated» besagt, dass die class-Datei automatisiert (mittels Werkzeugunterstützung) aus der Java-Datei erstellt wird. Der Stereotyp «custom code» besagt, dass das OOD-Modell durch eine manuelle Operation (hier UML-Modellierung) aus dem OOA-Modell erstellt wird.

Komponenten können intern aus mehreren Klassen (allgemein: aus *Classifiern*) bestehen. Diese interne Sicht oder White-Box-Sicht zeigt, wie das extern zur Verfügung gestellte Verhalten intern realisiert wird und betrifft nur den Entwickler der Komponente.

Komponenten-Struktur

Das Innenleben einer Komponente kann durch einen zusätzlichen Bereich im Rechteksymbol modelliert werden, der mit dem Schlüsselwort «realizations» gekennzeichnet ist. Anschließend werden die *Classifier* aufgeführt, die die Komponente realisieren (Abb. 9.2-44). *Classifier* bilden im Metamodell die Oberklasse zu Klassen, Schnittstellen etc. In der Abb. 9.2-44 werden als *Classifier* Klasse1 und Klasse2 verwendet. Alternativ für die interne Sicht ist eine Schachtelung der grafischen Symbole möglich. Hier wird zusätzlich eine Komposition zwischen den beteiligten Klassen gezeigt. Als weitere Alternative können diese Klassen durch eigene Symbole dargestellt und mit dem gestrichelten Pfeil mit der Komponente verbunden werden.

Notation

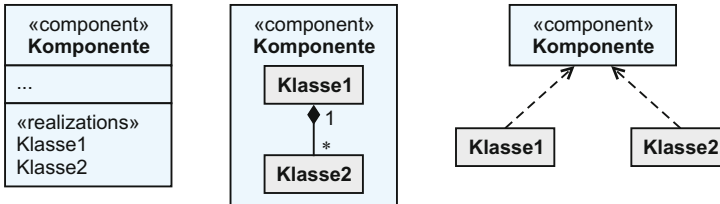


Abb. 9.2-44:
Notation für das
Innenleben einer
Komponente.

Abhängigkeiten zwischen UML-Elementen

Zwischen Elementen eines UML-Modells können Abhängigkeiten (*dependencies*) existieren. Sie werden durch einen gestrichelten Pfeil modelliert (Abb. 9.2-45) und bedeuten, dass das abhängige Element (*client*) entweder semantisch oder strukturell vom Basis-Element (*supplier*) abhängig ist. Die Art der Abhängigkeit kann durch einen Stereotypen spezifiziert werden (siehe »Basiskonzepte«, S. 99). Zusätzlich kann ein Name angegeben werden. Die UML stellt eine Reihe verschiedener Abhängigkeiten zur Verfügung und es können weitere definiert werden.

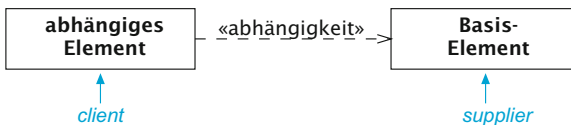


Abb. 9.2-45:
Abhängigkeiten
zwischen
Modellelementen.

Die Abb. 9.2-46 modelliert folgende Arten von Abhängigkeiten:

- «instantiate»: Die Klasse Fahrzeug kann als Exemplar (*instance*) der Klasse Fahrzeugtyp aufgefasst werden.

III 9 Statik

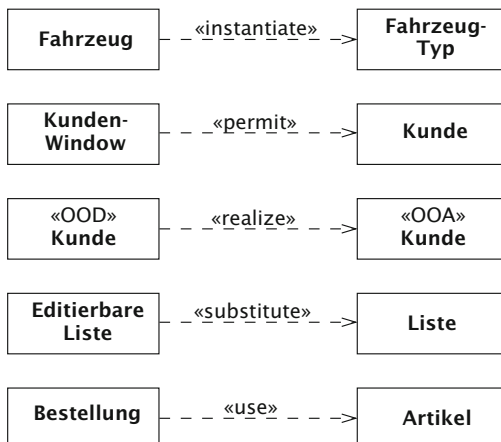
- «permit»: Die Klasse Kunden-Window besitzt Zugriffsrechte auf die Klasse Kunde, d.h. Kunden-Window darf auf die nicht-öffentlichen Elemente von Kunde direkt zugreifen.
- «realize»: Die OOD-Klasse Kunde realisiert die OOA-Klasse Kunde. Diese Beziehung beschreibt eine Abstraktion zwischen zwei Elementen. Sie kann beispielsweise für die Spezifikation von Verfeinerungen und Optimierungen verwendet werden.
- «substitute»: Die Klasse EditierbareListe kann die Klasse Liste ersetzen. Das bedeutet, dass die Klasse EditierbareListe hundertprozentig kompatibel zu der Klasse Liste ist, aber zusätzliche Funktionalität bereitstellt.
- «use»: Die Klasse Bestellung benötigt für ihre volle Funktionalität die Klasse Artikel.

Es wird empfohlen, Abhängigkeiten nur dann zu verwenden, wenn zwischen zwei Klassen – oder allgemeiner zwischen zwei Modellelementen – ein Zusammenhang besteht, der *nicht* durch eine Assoziation abgebildet werden soll [JRH+04].

Zur Klassifikation

- Abhängigkeiten zwischen Komponenten, Artefakten und weiteren UML-Modellelementen werden in grafischer Form mit textuellen Annotationen dargestellt.
- Der Grad der Formalität ist semiformal.
- Die Abhängigkeiten werden in der Spezifikations- und der Entwurfsphase angegeben.
- Sie werden in allen Anwendungsbereichen verwendet.

Abb. 9.2-46:
Verschiedene
Abhängigkeiten
zwischen
Modellelementen.



9.3 Daten

Im einfachsten Fall wird ein Datenelement durch einen Namen beschrieben. In der Regel besitzen Datenelemente aber noch eine Reihe weiterer Eigenschaften, die je nach Einsatzbereich ergänzt werden. Beispiele für solche Eigenschaften sind:

- Typ des Datenelements, z. B. String.
- Initialisierungswert des Datenelements, z. B. `int anzahl = 0;`.
- Muss- oder Kann-Element, d. h. muss es einen Wert haben oder kann es einen Wert besitzen.
- Struktur des Datenelements, z. B. mehrdimensionales Feld.
- Darstellung der Literale für dieses Datenelement.

Datenelemente können informal, semiformal oder formal spezifiziert werden. Die Beschreibung kann textuell oder grafisch erfolgen. In Anforderungsspezifikationen, z. B. Lasten- und Pflichtenheften, werden Datenelemente oft informal und textuell spezifiziert. In Programmiersprachen wird die Syntax von Sprachelementen oft in Form einer **EBNF**-Notation und grafisch in Form von **Syntaxdiagrammen** angegeben.

Syntax

In einem Pflichtenheft können Datenelemente wie folgt beschrieben werden (siehe »Fallstudie: SemOrg – Die Spezifikation«, S. 107):

Beispiel:
SemOrg

/F100/ (/LF70/) Das System *muss* folgende Veranstaltungsdaten (maximal 100.000) permanent speichern:

Veranstaltungs-Nr., Dauer (in Tagen), Vom, Bis, Tagesraster-Anfang, Tagesraster-Ende, Anfang erster Tag, Ende letzter Tag, Veranstaltungsort (Hotel/Firma, Adresse, Raum), Kooperationspartner, Öffentlich (Ja/Nein), Netto-Preis, Stornogebühr, min. Teilnehmerzahl, max. Teilnehmerzahl, Teilnehmer aktuell, Durchgeführt (Ja/Nein).

In diesem Beispiel werden außer den Datenelement-Namen zum Teil noch Hinweise auf den Wertebereich gegeben. Weitere Eigenschaften sind nicht aufgeführt.

Daten werden häufig in der **UML** modelliert (Attribute genannt), in **Programmiersprachen** implementiert (Variablen genannt) oder in **XML** spezifiziert (Elemente genannt).

Attribute,
Variablen,
Elemente

UML

Die Abb. 9.3-1 zeigt, wie ein **Attribut** in der UML spezifiziert werden kann, wobei alle Angaben mit Ausnahme des Attributnamens optional sind.

Die einzelnen Angaben werden im Folgenden näher erläutert und beschrieben.

Klasse
attribut1
attribut2: Typ
attribut3: Typ {Eigenschaftswert}
attribut4: Typ = Anfangswert
attribut5: Typ [0..10]
<u>klassenattribut</u>
/abgeleitetes Attribut

Abb. 9.3-1:
Notation für
Attribute.

III 9 Statik

Attributname Der Attributname muss im Kontext der Klasse eindeutig sein. Im Allgemeinen wird ein Substantiv dafür verwendet. Attributnamen beginnen laut UML-Spezifikation mit einem kleinen Anfangsbuchstaben und dürfen beliebige Zeichen (z. B. Umlaute, Sonderzeichen, Leerzeichen) enthalten. Bei der Erstellung von Analysemodellen ist dies äußerst praktisch. Bei Entwurfsmodellen ist es allerdings empfehlenswert, sich an die Syntaxvorschriften der verwendeten Programmiersprache zu halten. Attributnamen, die aus mehreren Wörtern bestehen, können mit einem Leerzeichen oder einem Unterstrich getrennt werden. Häufig beginnt jedes neue Wort im Attributnamen mit einem Großbuchstaben (Kamelhöckernotation). Da ein Attributname nur innerhalb der Klasse eindeutig ist, verwendet man außerhalb des Klassenkontextes die Bezeichnung `Klasse.attribut`.

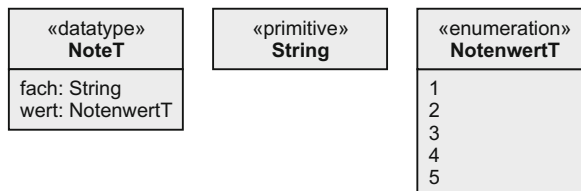
Attributtyp Der Typ von Attributen kann in der UML modelliert werden durch

- Datentypen (Stereotyp «datatype»),
- primitive Datentypen (Stereotyp «primitive») als Sonderfall der Datentypen,
- Aufzählungstypen (Stereotyp «enumeration») als Sonderfall der Datentypen und
- Klassen.

Stereotyp Bei der Definition der Datentypen wird das Konzept der **Stereotypen** verwendet. Sie ermöglichen es, existierende Modellelemente mit einer geänderten Semantik zu versehen. Die Stereotypen «datatype», «primitive» und «enumeration» sind in der UML vordefiniert (siehe auch »Basiskonzepte«, S. 99).

Datentypen **Datentypen** sind Typen, deren Werte keine Identität besitzen. Das ist der wesentliche Unterschied zu Klassen, deren Objekte stets eine Objektidentität haben. Die nicht vorhandene Objektidentität hat zur Folge, dass Werte eines Datentyps nur innerhalb des Objekts existieren können, zu dem sie gehören. Außerdem gibt es jeden Wert nur einmal. Kommt beispielsweise die Zahl 7 mehrmals vor, so handelt es sich immer um ein und dieselbe Zahl. Datentypen dürfen wie Klassen Attribute und Operationen besitzen. Datentypen werden im Allgemeinen dafür verwendet, Strukturen zu beschreiben. Beispielsweise spezifiziert in der Abb. 9.3-2 ein Datentyp die NoteT eine Note bestehend aus dem Fach und dem Notenwert.

Abb. 9.3-2:
Datentypen zur
Spezifikation von
Attributtypen.



Primitive Datentypen sind in der UML spezielle Datentypen, die *keine* Struktur besitzen. Sie werden ebenfalls in der Klassennotation modelliert und mit dem Schlüsselwort «primitive» gekennzeichnet. In der UML sind die folgenden vier primitiven Datentypen vordefiniert:

- Boolean (kann die Werte true und false annehmen)
- String
- Integer (ganze Zahlen)
- UnlimitedNatural (natürliche Zahlen)

Es ist zulässig, weitere primitive Datentypen zu definieren.

Ein **Aufzählungstyp** (*enumeration datatype*) definiert eine endliche Menge von Werten. Er wird mit dem Schlüsselwort «enumeration» gekennzeichnet. Die Werte des Aufzählungstyps können frei gewählt werden. Die Abb. 9.3-2 zeigt wie der Aufzählungstyp NotenwertT mit den einzelnen Notenwerten modelliert wird. In der Analyse dient die Typdefinition dem Zweck, das Attribut aus fachlicher Sicht möglichst präzise zu beschreiben. In Entwurf und Implementierung wird in Abhängigkeit von der gewählten Programmiersprache der Typ neu definiert. Es empfiehlt sich, Datentypen – in Abhängigkeit von der jeweiligen Anwendung – einmal zu definieren und bei jedem Projekt wieder zu verwenden. Um sie deutlich von den Klassen zu unterscheiden, wird in diesem Buch für Datentypen das Postfix »T« verwendet, z. B. NoteT. Die Attribut- und Operationsliste wird bei der Spezifikation von Datentypen oft unterdrückt, insbesondere, wenn es sich um einen primitiven Datentyp handelt.

Eigenschaftswerte (*property values*) spezifizieren, ob die Attribute bestimmte Eigenschaften oder Merkmale besitzen. Sie werden in geschweiften Klammern angegeben. Mehrere Eigenschaftswerte werden durch Kommas getrennt. Für Attribute bietet die UML beispielsweise folgende Eigenschaftswerte:

- {readOnly}: Attribut darf nicht verändert werden.
- {ordered}: Wenn ein Attribut aus mehreren Werten besteht, dann wird dadurch festgelegt, dass sie geordnet sind. Besteht das Attribut nur aus einem Element, dann hat dieser Eigenschaftswert keine Wirkung.

Der **Anfangswert** (*initial value*) legt fest, welchen Wert ein neu erzeugtes Objekt für dieses Attribut annimmt. Dieser Wert kann später beliebig geändert werden.

Für ein Attribut kann die **Multiplizität** (*multiplicity*) definiert werden. Diese Angabe erfolgt in eckigen Klammern.

Können für einen Studierenden (vgl. Abb. 9.3-2) bis zu 11 Noten eingetragen werden, dann wird dies wie folgt spezifiziert:

note: NoteT [0..10] bzw. note [0..10].

Primitive
Datentypen

Aufzählungstyp

Eigenschafts-
werte

Anfangswert

Multiplizität

Beispiel

III 9 Statik

Sind bei der Multiplizität die untere und obere Angabe identisch, dann reicht ein einziger Wert, d. h. $[5..5] = [5]$. Ist die untere Grenze gleich Null und die obere Grenze unspezifiziert, dann gilt $[0..*] = [*]$. Die Multiplizität $[1] = [1..1]$ bedeutet, dass das Attribut genau einen Wert besitzt. Das heißt, dass dieser Wert beim Erzeugen eines Objekts der Klasse eingetragen werden muss. Falls keine Multiplizität angegeben wird, gilt $[1]$ als Voreinstellung. Soll ausgedrückt werden, dass es sich um ein optionales Attribut handelt, das irgendwann einmal einen Wert erhalten kann, so muss die Multiplizität $[0..1]$ angegeben werden. Ist die Obergrenze der Multiplizität größer als 1, dann handelt es sich um ein Attribut, das aus mehreren Werten bestehen kann.

Beispiel Für eine Klasse *Student* ergeben sich die Attributtypen der Abb. 9.3-3.

Abb. 9.3-3:
Attributtypen für
Klasse *Student*.

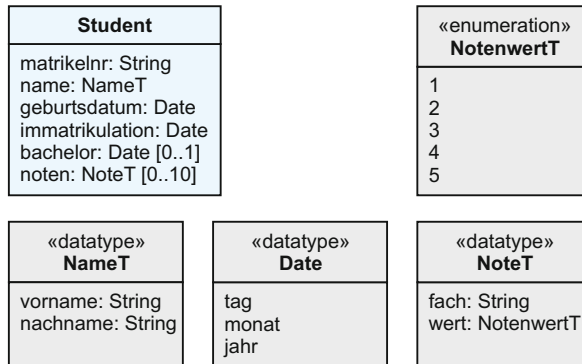
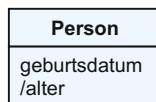


Abb. 9.3-4:
Abgeleitetes
Attribut.



Das Symbol »/« kennzeichnet ein abgeleitetes Attribut. Der Wert eines **abgeleiteten Attributs** (*derived attribute*) kann jederzeit aus anderen Attributwerten berechnet werden (Abb. 9.3-4). Ein abgeleitetes Attribut darf *nicht* geändert werden.

Außer den oben beschriebenen (Objekt-)Attributen sind manchmal Klassenattribute notwendig. Ein **Klassenattribut** (*class scope attribute*) liegt vor, wenn nur ein Attributwert für alle Objekte einer Klasse existiert. Klassenattribute existieren auch dann, wenn es zu einer Klasse – noch – keine Objekte gibt. Um die Klassenattribute von den (Objekt-)Attributen zu unterscheiden, werden sie in der UML unterstrichen (z. B. klassenattribut).

Geheimnis-
prinzip

Das Geheimnisprinzip der Objektorientierung besagt, dass Attribute nur über die Operationen der zugehörigen Klasse geändert und gelesen werden dürfen. Trotzdem ist es kein Widerspruch, wenn die

Attribute im Klassendiagramm modelliert werden. Die Attribute sind nur sichtbar für den Modellierer bzw. den *Requirements Engineer*, aber nicht sichtbar für andere Klassen bzw. deren Objekte.

Für Attribute können **Einschränkungen** (*constraints*) in umgangssprachlicher oder in maschinenlesbarer Form definiert werden. Eine Einschränkung ist eine Invariante bzw. eine Zusicherung, die immer wahr sein muss. Eine Reihe von Einschränkungen sind in der UML bereits vordefiniert, weitere können durch die Modellierer beliebig spezifiziert werden (siehe »Constraints und die OCL in der UML«, S. 377). Einschränkungen werden als Text in geschweiften Klammern angegeben. Sie können sich auf ein einzelnes Attribut oder mehrere Attribute beziehen.

Einschränkung

Für das Attribut `geburtsdatum` einer Klasse `Student` gilt die Einschränkung: `{geburtsdatum <= aktuelles Datum}`

Beispiele

Für die Attribute der Klasse `Student` in der Abb. 9.3-3 gilt:

`{bachelor > immatrikulation > geburtsdatum}`

Für eine Klasse `Artikel` mit den Attributen `einkaufspreis` und `verkaufspreis` gilt, dass der Verkaufspreis mindestens 150 Prozent des Einkaufspreises betragen soll. Dann muss durch die Implementierung sichergestellt werden, dass beim Ändern des einen Preises auch der andere geändert wird:

`{verkaufspreis >= 1.5 * einkaufspreis}`

Die Syntax für Attribute in der UML sieht wie folgt aus:

Syntax

```
Attribut ::= [ Sichtbarkeit ] [ / ] name [ : Typ ]
[ [ Multiplizität ] [ =Anfangswert ] [ { Eigenschaftswerte-Attr } ]
Sichtbarkeit ::= [ - | # | + | ~ ]
Multiplizität ::= [ * | 0..* | 1..* | 0..1 | 1 | n..m (mit 0<=n<m) |
n (wenn n=m) ] (Hinweis: * ist gleichbedeutend mit 0..*)
Typ ::= [ Klasse | Datentyp | Aufzählungstyp | Primitiver
Datentyp ]
Primitiver Datentyp ::= [ String | Boolean | Integer | Unlimited
Natural ]
Eigenschaftswerte-Attr ::= [ readOnly | union | subsets<attr> | re-
defines<attr> | ordered | unique | Einschränkung-Attr ] ...
```

Programmiersprachen

Interessant ist, dass es in Programmiersprachen z. B. gegenüber der UML nur eingeschränkte Möglichkeiten für die Festlegung von Attributeigenschaften gibt. Viele Eigenschaften müssen explizit »ausprogrammiert« werden.

Die Syntax einer Variablen- bzw. Konstantendeklaration in Java sieht in einer Pseudo-EBNF wie folgt aus:

Beispiel

`FieldDeclaration ::=`

III 9 Statik

```
{ private / public / protected / final / static / }+ Type
{ Identifier / VariableDeclaratorId [ ]
[ = { Expression / ArrayInitializer } ] }...;
```

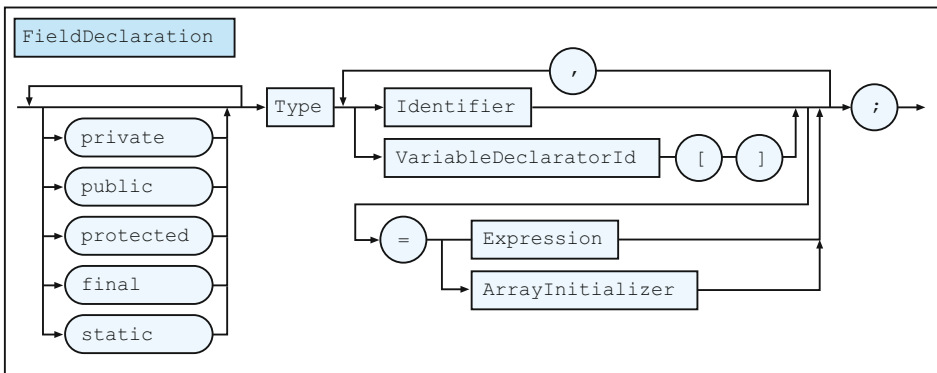


Abb. 9.3-5: So sieht eine Variablendeklaration in Java aus (Ausschnitt).

Das entsprechende Syntaxdiagramm zeigt die Abb. 9.3-5. Zu beachten ist, dass von den Schlüsselwörtern **private**, **public** und **protected** nur jeweils ein Schlüsselwort verwendet werden darf. Außerdem dürfen die Schlüsselwörter **final** und **static** nur jeweils einmal auftreten. Dieses Beispiel zeigt, dass die Bedeutung einer Sprachkonstruktion, d.h. die Semantik, durch die Syntax oft nicht oder nur unvollständig beschrieben werden kann.

XML

In der Markup-Sprache **XML** bestehen XML-Dokumente aus Elementen. Ein **Element** besteht aus einer Anfangsmarkierung, dem eigentlichen Elementinhalt und einer Endmarkierung.

Beispiel Eine PLZ wird beispielsweise durch das Element `<plz>12340</plz>` spezifiziert. Der Inhalt dieses Elements ist die Zeichenfolge 12340, die durch eine Anfangs- und Endmarkierung geklammert wird.

- Elementname Für die Elementnamen gibt es einige Vorschriften:
- Der Name muss mit einem Buchstaben, einem Unterstrich oder einem Doppelpunkt beginnen.
 - Danach dürfen als zusätzliche Zeichen Ziffern, Bindestrich und Punkt verwendet werden.
 - Die Zeichenfolge »xml« ist reserviert und darf nicht am Anfang eines Namens verwendet werden.
 - Die Länge eines Namens ist nicht begrenzt.
 - Groß- und Kleinschreibung werden unterschieden.

Einfache Elemente Bei einfachen Elementen besteht der Inhalt aus einer Zeichenkette.

Leere Elemente Neben Elementen mit Inhalt gibt es auch leere Elemente.

9.4 Box: Attribute – Methode und Checkliste III

Eine Adresse wird um die Angabe `<beruflich></beruflich>` erweitert. Zwischen Anfangs- und Endemarkierung steht *keine* Information. Dieses Element sagt aus, dass es sich bei einer Adresse um einen beruflichen Kontakt handelt. Ein Inhalt ist hier *nicht* notwendig.

Beispiel

Für inhaltslose Elemente wird im Allgemeinen eine Kurzform verwendet, z. B. `<beruflich/>`.

Jedes Element in einem XML-Dokument – auch ein leeres Element – kann beliebig viele **Attribute** enthalten. Sie werden in der Anfangsmarkierung durch Name-Wert-Paare angegeben. Beispielsweise könnte zu einer Adresse angegeben werden, wann sie ins Adressbuch eingetragen wurde. Attributwerte sind Zeichenketten, die in doppelten oder einfachen Anführungszeichen stehen.

Attribute

Für die Bildung von Attributnamen gelten die gleichen Vorschriften wie für Elementnamen. Innerhalb einer Markierung muss jeder Attributname eindeutig sein. Es ist jedoch erlaubt, den gleichen Attributnamen in verschiedenen Elementen zu verwenden.

Attributname

Werden in einer Anfangsmarkierung mehrere Attribute (Name-Wert-Paare) aufgeführt, dann müssen sie durch ein Leerzeichen getrennt werden. Die Reihenfolge ist beliebig.

Mehrere Attribute

Elemente mit Attributen:

Beispiele

```
<adresse erfasst_am="2009-07-17"
        aktualisiert_am="2009-10-11">
    ...
</adresse>
<beruflich kunde="ja"/>
```

XML definiert im Sprachstandard *keine* einzige Markierung. Es liegt im Ermessen des Entwicklers, sich selbst passende Markierungen zu überlegen. Durch Dokumenttyp-Definitionen und durch XML-Schemata können Elemente weiter spezifiziert werden:

Keine
Markierungs-
vorgaben

■ »XML, DTD und XML-Schemata«, S. 190

9.4 Box: Attribute – Methode und Checkliste

Zum Identifizieren und Überprüfen von Attributen helfen die Boxen »Checkliste Attribute«. Auf einige Aspekte zu den Boxen wird im Folgenden eingegangen.

Konstruktive Schritte

Liegt bereits eine **Anforderungsspezifikation** (z. B. Lasten- oder Pflichtenheft) vor, dann können aus den dort angegebenen Angaben Attribute identifiziert werden.

1 Anforderungs-
spezifikation

**Box 1a:
Checkliste
Attribute**

Ergebnisse

■ **Klassendiagramm**

Tragen Sie für jedes Attribut den Namen in das Klassendiagramm ein. Kennzeichnen Sie Klassenattribute und abgeleitete Attribute.

Spezifizieren Sie Attribute vollständig durch Angabe des Typs, der Multiplizität und der Eigenschaftswerte. Für komplexe Attribute sind ggf. entsprechende Typen zu definieren.

■ **XML-Dokument**

Attribute können auch in XML spezifiziert werden.

Konstruktive Schritte

1 Welche Attribute lassen sich aus der Anforderungsspezifikation (Lasten-/Pflichtenheft) oder mittels Dokumentanalyse identifizieren?

- Die Anforderungsspezifikation bezogen auf Datenbeschreibungen durchsehen.
- Einfache Attribute sind ggf. zu Datenstrukturen zusammenzufassen.
- Prüfen, ob alle Attribute wirklich notwendig sind.
- Für jedes Attribut prüfen, ob es »im Laufe seines Lebens« einen Wert annehmen kann.

2 Welche Attribute lassen sich anhand der Use Cases identifizieren?

- Benötigte Daten zur Ausführung der Aufgaben eines Use Case.
- Benötigte Daten für Listenfunktionalität.

3 Wurden geeignete Attributtypen gewählt und u. U. als Datentypen (Stereotyp «datatype») oder Aufzählungstypen (Stereotyp «enumeration») beschrieben?

- Vorgegebene Typen *nur* verwenden, wenn problemadäquat.
- Attribute beliebigen Typs definieren, um problemadäquate Modellierung auf ausreichendem Abstraktionsniveau zu erreichen.

Analytische Schritte

5 Klasse oder komplexes Attribut?

Das folgende Beispiel zeigt, dass die Alternativen »Klasse« oder »Attribut« sorgfältig überlegt werden müssen.

Zu einem Artikel werden folgende Informationen benötigt: *nr*, *bezeichnung*, *preisHaendler*, *preisGrosskunde*, *preisEinzelkunde*, *waehrung*, *lagerbezeichnung*, *lagerort*. Pflichtattribute sind *nr* und *bezeichnung*.

Es gibt drei Alternativen:

- Es werden alle Attribute einer Klasse *Artikel* zugeordnet.
- Es werden die drei Klassen *Artikel*, *Preis* und *Lager* modelliert (Abb. 9.4-3, oben).
- Es werden die zwei Klassen *Artikel* und *Lager* modelliert (Abb. 9.4-3, unten).

Welche Alternative ist die beste? Die Angaben zum Preis sind offensichtlich Attribute, da sie ohne ein entsprechendes Artikelobjekt *keinen* Sinn ergeben. Eine eigene Klasse mit der notwendigen Assoziation erhöht unnötig die Anzahl der Klassen. Außerdem besteht zwischen den Klassen *Artikel* und *Preis* dann eine starke Kopplung.

Analytische Schritte**4 Ist der Attributname geeignet?**

Der **Attributname** soll

- kurz, eindeutig und verständlich im Kontext der Klasse sein,
- ein Substantiv oder Adjektiv-Substantiv sein (kein Verbl!),
- den Namen der Klasse nicht wiederholen (Ausnahme: feststehende Begriffe),
- bei komplexen (strukturierten) Attributen der Gesamtheit der Komponenten entsprechen,
- nur fachspezifische oder allgemein übliche Abkürzungen enthalten.

5 Klasse oder komplexes Attribut?

- Klasse: Objektidentität, gleichgewichtige Bedeutung im System, Existenz unabhängig von der Existenz anderer Objekte, Zugriff in beiden Richtungen grundsätzlich möglich.
- Attribut: keine Objektidentität, Existenz abhängig von Existenz anderer Objekte, Zugriff immer über das Objekt, untergeordnete Bedeutung.

6 Wurde das richtige Abstraktionsniveau gewählt?

- Wurden komplexe Attribute gebildet?
- Bilden komplexe Attribute geeignete Datenstrukturen?
- Ist die Anzahl der Attribute pro Klasse angemessen?

7 Gehört das Attribut zu einer Klasse oder einer Assoziation?

- Test: Muss das Attribut auch dann zu jedem Objekt der Klasse gehören, wenn die betreffende Klasse isoliert von allen anderen Klassen betrachtet wird?
- ☐ Wenn ja, dann gehört das Attribut zu dieser Klasse.
- ☐ Wenn nein, dann ist zu prüfen, ob es sich einer Assoziation zuordnen lässt.
- ☐ Ist keine Zuordnung möglich, dann spricht viel für eine vergessene Klasse oder Assoziation.

8 Liegen Klassenattribute vor?

- Ein Klassenattribut liegt vor, wenn gilt:
- ☐ Alle Objekte der Klasse besitzen für dieses Attribut denselben Attributwert.
- ☐ Es sollen Informationen über die Gesamtheit der Objekte modelliert werden.

9 Sind Schlüsselattribute fachlich notwendig? Schlüsselattribute werden nur dann eingetragen, wenn sie – unabhängig von ihrer identifizierenden Eigenschaft – Bestandteil des Fachkonzepts sind.

10 Werden abgeleitete Attribute korrekt verwendet?

- Information ist für den Benutzer sichtbar.
- Lesbarkeit wird verbessert.

11 Wann wird ein Attribut nicht eingetragen?

- Es handelt sich um ein Attribut, das den internen Zustand eines Lebenszyklus beschreibt und außerhalb des Objekts nicht sichtbar ist.
- In der Spezifikationsphase:
Es beschreibt Entwurfs- oder Implementierungsdetails.
- Es handelt sich um ein abgeleitetes Attribut, das nur aus Performance-Gründen eingefügt wurde.

12 Fehlerquellen

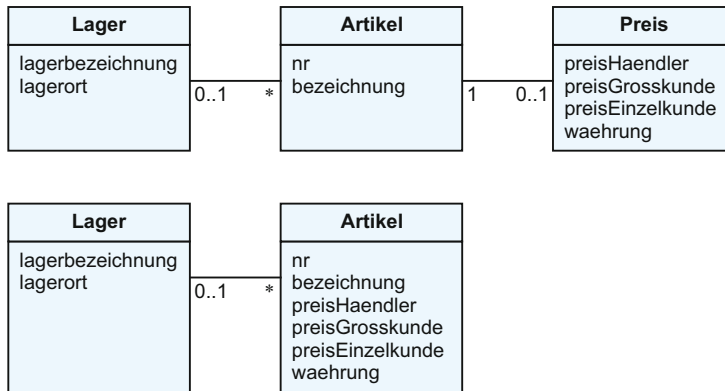
- Verwenden atomarer Attribute anstelle von komplexen Datenstrukturen.
- Formulieren von Assoziationen als Attribute (Fremdschlüssel!).

**Box 1b:
Checkliste
Attribute**

Ein Lager wird sinnvollerweise durch eine eigene Klasse modelliert. Ein konkretes Lager soll unabhängig davon, ob es gerade Artikel enthält, im System existieren. Es sollen Anfragen der Art »Welche Artikel befinden sich im Lager West in der Nordstadt« möglich sein. Daher stellt die dritte Alternative die beste Modellierung dar.

III 9 Statik

Abb. 9.4-3: Zwei Modellierungsalternativen für Artikeldaten.



9.5 Daten-Strukturen

Datenelemente können durch statische Beziehungen zu Datenstrukturen verknüpft werden. In Programmiersprachen legt die Syntax fest, wie die einzelnen Sprachelemente miteinander kombiniert werden dürfen.

i Baumartige, statische Strukturen zwischen Datenelementen lassen sich mit XML und XML-Schemata beschreiben:

■ »XML, DTD und XML-Schemata«, S. 190

Ein wichtiges Modell zur Verknüpfung von Datenelementen ist das ER-Modell:

■ »Entity-Relationship-Modell«, S. 199

Für die Anwendungsbereiche *Data Warehouses* und *Data Marts* müssen zusätzliche Beziehungen und Eigenschaften zwischen Datenelementen festlegbar sein:

■ »Multidimensionale Datenstrukturen«, S. 214

9.5.1 XML, DTD und XML-Schemata

XML¹

Strukturierte Elemente Außer einfachen Elementen (siehe »Daten«, S. 181) können in **XML** auch strukturierte Elemente definiert werden, die selbst wieder Elemente enthalten, die sowohl einfache als auch strukturierte Elemente sein können. Diese Elemente müssen korrekt ineinander geschachtelt sein.

Beispiel Strukturiertes Element mit zwei einfachen Elementen:

```

<name>
  <vorname>Marie</vorname>

```

¹Ein Teil der Inhalte wurde mit freundlicher Genehmigung der W3L GmbH dem Buch »Basiswissen Web-Programmierung« von Heide Balzert entnommen.

```
<name><nachname>Risser</nachname>
</name>
```

Jedes XML-Dokument besitzt ein **Wurzelement**. Das ist dasjenige Element, das alle anderen Elemente enthält. Jedes XML-Dokument bildet daher eine **Baumstruktur von Elementen**.

Wurzelement

Ein Brief kann als XML-Dokument wie folgt formuliert werden:

Beispiel 1a

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<brief>
  <adresse>
    <name>Meier</name>
    <strasse>Elisenstraße</strasse>
    <plz>80335</plz>
    <ort>München</ort>
  </adresse>
  <betreff>Ihre Anfrage vom 26.07.2009 bezüglich einer
    Schulung
  </betreff>
  <anrede>Sehr geehrte Damen und Herren</anrede>
  <text>der Text des Briefes, der sich jetzt strukturell
    eindeutig von Betreff und Anrede abhebt.</text>
</brief>
```

Das Wurzelement ist in diesem Beispiel `<brief>`. Die »encoding«-Anweisung in der ersten Zeile erlaubt die Verwendung von deutschen Umlauten im Text.

Ein wichtiger Vorteil von XML liegt darin, dass z.B. mit einem Suchprogramm nach allen Briefen, die im Betreff das Wort »Schulung« enthalten, gesucht werden kann. Mit XML können erstens Memos von Briefen unterschieden werden, zweitens kann gezielt nach Wörtern im Betreff gefahndet werden. Voraussetzung ist natürlich, dass in allen Briefen die gleichen Namen für Markierungen verwendet werden.

XML definiert im Sprachstandard *keine* einzige Markierung. Es liegt im Ermessen des Entwicklers, sich selbst passende Markierungen zu überlegen.

Dokumenttyp-Definitionen

Wenn in einem großen Unternehmen viele Mitarbeiter Briefe im XML-Format schreiben, jedoch jeder seine eigenen Markierungen entwickelt, geht der Vorteil der automatisierten Verarbeitung (z.B. bei Suchanfragen) schnell verloren. Die Einhaltung einer Namenskonvention für Markierungen bzw. einer einheitlichen Struktur für gleichartige Dokumente lässt sich in XML aber optional automatisch sicherstellen.

III 9 Statik

DTDs = Struktur
von XML-
Dokumenten

Dokumenttyp-Definitionen (**DTDs**) beschreiben die Struktur von XML-Dokumenten. In einer DTD wird angegeben, welche Elemente ein Dokument enthalten muss und wie sie ineinander geschachtelt sein müssen. Eine DTD ist entweder in das Dokument integriert oder das Dokument enthält die **URL**, über die auf die DTD zugegriffen werden kann.

Beispiel 1b Eine DTD für einen Brief kann folgendermaßen aussehen:

```
<!ELEMENT brief(adresse, betreff, anrede+, text)>
<!ELEMENT adresse(firma?, name, strasse, plz, ort)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT strasse(#PCDATA)>
<!ELEMENT plz(#PCDATA)>
<!ELEMENT ort(#PCDATA)>
<!ELEMENT betreff(#PCDATA)>
<!ELEMENT anrede(#PCDATA)>
<!ELEMENT text(#PCDATA)>
```

Nach dieser DTD besteht ein Brief aus einer Adresse, einem Betreff, einer oder mehreren Anreden (spezifiziert durch das »+«-Symbol, z. B. »Sehr geehrter Herr Müller,« »Sehr geehrte Damen und Herren«) und dem Text. Die Adresse ist wieder in einzelne Elemente unterteilt, wobei die Angabe einer Firma optional ist (spezifiziert durch das »?«-Symbol). »#PCDATA« erlaubt beliebigen Text, jedoch keine weiteren Unterelemente. Um in das Element Text sowohl beliebigen Text als auch Unterelemente schreiben zu können, muss es als »<!ELEMENT Text(ANY)>« definiert werden.

Mit Hilfe von DTDs lassen sich Dokumente nur oberflächlich spezifizieren. Aus der Brief-DTD von Beispiel 1b geht z. B. nicht hervor, dass eine Postleitzahl nur aus Ziffern bestehen darf. Für den Austausch von Adressen zwischen unterschiedlichen Systemen ist dies jedoch ein wichtiges Kriterium.

XML-Schemata

Diesen Nachteil besitzen XML-Schemata nicht. Es handelt sich dabei um eine XML-basierte Sprache (XML-Schemata sind also wieder XML-Dokumente), die wesentlich mächtigere Konstrukte zur Spezifikation von Struktur, Inhalt und Semantik von XML-Dokumenten enthält als DTDs es ermöglichen. Ein **XML-Schema** legt fest, wie ein XML-Dokument aufgebaut sein muss. Im Unterschied zur DTD ist ein Schema selbst ein XML-Dokument, das auf Wohlgeformtheit geprüft werden kann. Es wird in eine separate Datei mit der Dateiendung .xsd geschrieben.

Aufbau

Da ein XML-Schema ein XML-Dokument ist, beginnt es mit der XML-Deklaration, die die XML-Version und den Zeichensatz definiert. Das Wurzelement eines Schemas ist immer <xs:schema>. An dieser Aus-

zeichnung erkennt der Parser, dass es sich *nicht* um ein »normales« XML-Dokument, sondern um ein Schema handelt. Ein XML-Schema besitzt somit folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">...
</xs:schema>
```

Das Wurzelement eines XML-Schemas ist das Element `schema`. Zur Kennzeichnung, dass es sich bei dem jeweiligen Dokument um eine XML-Schema-Spezifikation handelt, muss im Wurzelement der Namensraum `<http://www.w3.org/2001/XMLSchema>` deklariert werden. In diesem Namensraum sind alle Elemente, Attribute und die vordefinierten Datentypen enthalten, die beim Entwurf eines Schemas verwendet werden können. Über das `xmlns`-Attribut wird ein Präfix an den Namensraum gebunden. Als Namensraum-Präfix wird meist `xs` oder `xsd` benutzt.

Wurzelement

Im XML-Dokument teilt die folgende Angabe dem XML-Prozessor mit, wo sich das Schema befindet, das für die Validierung verwendet werden soll:

Schema referenzieren

```
<adressbuch xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="adressen.xsd"> ...
</adressbuch>
```

Die Angabe `adressbuch` kennzeichnet das Wurzelement des XML-Dokuments und `"adressen.xsd"` gibt den aktuellen Dateinamen einschließlich des kompletten Pfades für das XML-Schema an. In diesem Fall liegt die XSD-Datei im gleichen Verzeichnis wie das XML-Dokument, das sie referenziert.

Jedes Element, das im XML-Dokument verwendet wird, muss im XML-Schema deklariert werden. Es werden einfache und komplexe Elemente unterschieden. Jedes Element wird in der folgenden Syntax beschrieben:

Elemente

```
<xs:element name="elementname" ...>
```

Einfache Elemente dürfen weder andere Elemente noch Attribute enthalten.

Einfache Elemente

Hier wird das Element `vorname` vom Typ `String` deklariert:

Beispiel

```
<xs:element name="vorname" type="xs:string" />
```

XML-Schemata geben eine breite Palette von Standardtypen vor, die ebenfalls mit dem Präfix `xs` beginnen. Dazu gehören:

Standardtypen

- `xs:string` für Zeichenketten.
- `xs:decimal` für dezimale Zahlen beliebiger Genauigkeit, wobei die Ziffern nach dem Dezimalzeichen einschließlich dieses Zeichens entfallen können.
- `xs:integer` für ganze Zahlen beliebiger Größe.
- `xs:float` für Gleitkommazahlen (einfacher Genauigkeit).

III 9 Statik

- `xs:boolean` für logische Werte `true` und `false`.
- `xs:date` für ein Kalenderdatum in dem Format `jjjj-mm-dd`.

Komplexe Elemente Komplexe Elemente bestehen im Allgemeinen aus mehreren Elementen. Auch wenn ein Element ein Attribut enthält, muss es im XML-Schema als komplexes Element deklariert werden. Beispielsweise bildet eine Adresse ein komplexes Element. Dies wird durch das Element `<xs:complexType>` ausgedrückt. Die darin geschachtelte Markierung `<xs:sequence>` fordert, dass die enthaltenen Elemente genau in der vorgegebenen Reihenfolge vorliegen müssen. Verwenden Sie `<xs:sequence>` zunächst wie hier angegeben.

Beispiel

```
<xs:element name="adresse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>    ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Komplexer Datentyp Für komplexe Elemente kann auch ein neuer Typ definiert und dem Element zugewiesen werden. Diese Form besitzt den Vorteil, dass ein komplexer Datentyp einmal definiert und dann beliebig oft referenziert werden kann. Man spricht auch von **benannten Datentypen**, während Typen ohne Namen als **anonyme Typen** bezeichnet werden. XML-Schemata mit benannten komplexen Datentypen führen im Allgemeinen zu einer übersichtlicheren Darstellung. Daher sollten Sie diese Form der Schema-Definition bevorzugen.

Beispiel Für den Datentyp der Adresse ergibt sich folgende Deklaration:

```
<xs:complexType name="adresseType">
  <xs:sequence>
    <xs:element name="vorname" type="xs:string"/>
    <xs:element name="nachname" type="xs:string"/>    ...
  </xs:sequence>
</xs:complexType>
```

Dieser Typ wird wie folgt verwendet:

```
<xs:element name="adresse" type="adresseType"/>
```

Häufigkeit von Elementen Mit `minOccurs` kann man in einem XML-Schema festlegen, wie oft ein Element mindestens vorkommen muss. Als Voreinstellung gilt der Wert 1. Es bildet das Gegenstück zu `maxOccurs`, das die maximale Anzahl von Elementen festlegt. Auch hier gilt als Voreinstellung der Wert 1.

Ein Adressbuch soll mindestens eine und kann beliebig viele Adressen enthalten. Dies wird im XML-Schema durch folgende Angabe definiert:

Beispiel

```
<xs:complexType name="adressbuchType">
  <xs:sequence>
    <xs:element name="adresse" type="adresseType"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Auch leere Elemente werden in einem XML-Schema als komplexe Elemente definiert. Analog zu den Elementen mit Inhalt kann deren Häufigkeit angegeben werden.

Leere Elemente

Ein optionales leeres Element `<beruflich/>` wird in einem Element vom Typ `adresseType` wie folgt definiert:

Beispiel

```
<xs:complexType name="adresseType">
  <xs:sequence> ...
    <xs:element name="beruflich" minOccurs="0">
      <xs:complexType />
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Mit dem Kompositor `<xs:sequence>` können Sie definieren, dass die Elemente in einer komplexen Struktur nur in der festgelegten Reihenfolge stehen dürfen.

Festgelegte
Reihenfolge

Vorname und Nachname sollen im XML-Dokument in dieser Reihenfolge stehen:

Beispiel

```
<xs:complexType name="adresseType">
  <xs:sequence>
    <xs:element name="vorname" type="xs:string"/>
    <xs:element name="nachname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Der Kompositor `<xs:choice>` ist zu verwenden, wenn mehrere Alternativen zur Wahl stehen.

Alternativen

Ein Element vom Typ `telefonType` besteht entweder aus dem Element `mobil` oder aus dem Element `festnetz`:

Beispiel

```
<xs:complexType name="telefonType">
  <xs:choice>
    <xs:element name="mobil" type="xs:string"/>
    <xs:element name="festnetz" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```


III 9 Statik

Beliebige Reihenfolge Mit dem Kompositor `<xs:all>` können Sie definieren, dass die Elemente in einer komplexen Struktur in beliebiger Reihenfolge stehen dürfen. Jedes enthaltene Element darf nur einmal vorkommen.

Beispiel

```
<xs:complexType name="personType">
  <xs:all>
    <xs:element name="vorname" type="xs:string"/>
    <xs:element name="nachname" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

Voreinstellungen für Elemente Für einfache Elemente kann zusätzlich zum Elementtyp eine Voreinstellung definiert werden. Dieser Wert wird verwendet, wenn im XML-Dokument kein Wert angegeben wird.

Beispiel Die Adresse kann z.B. um das Element `land` erweitert werden, das mit Deutschland vorbelegt wird, weil dies für die meisten Adressen zutrifft:

```
<xs:element name="land" type="xs:string" default="Deutschland"/>
```

Einfache Datentypen

Außer den vorgegebenen Standardtypen und den komplexen Typen können einfache Typen (*simpleTypes*) definiert werden. Elemente vom einfachen Datentyp dürfen weder Kindelemente noch Attribute besitzen. Einfache Typen werden wie komplexe Typen bei der Deklaration der Elemente referenziert.

Einschränkungen Einfache Typen werden mithilfe von Einschränkungen (*restrictions*) aus den Standardtypen abgeleitet. Auch der Standardtyp `integer` wird aus dem Standardtyp `decimal` durch eine Einschränkung abgeleitet, indem die Anzahl der zulässigen Nachkommastellen auf Null gesetzt wird.

Beispiel Ein Typ `plzType` kann auf der Basis ganzer Zahlen definiert werden:

```
<xs:simpleType name="plzType">
  <xs:restriction base="xs:integer">
  </xs:restriction>
</xs:simpleType>
```

Und so wird der Typ im Element referenziert:

```
<xs:element name="plz" type="plzType"/>
```

Wertebereich festlegen Einschränkungen bieten noch mehr Möglichkeiten. Man kann damit beispielsweise den Wertebereich für numerische Elemente festlegen.

Beispiel Hier wird für den Typ `alterType` der Wertebereich auf 1 bis 99 eingeschränkt:

```
<xs:simpleType name="alterType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:maxInclusive value="99"/>
</xs:restriction>
</xs:simpleType>
```

Für Elemente vom Typ `<xs:string>` kann man die minimale und maximale Länge definieren. Länge festlegen

Ein Element vom Typ `passwortType` muss mindestens sechs und darf höchstens 20 Zeichen enthalten: Beispiel

```
<xs:simpleType name="passwortType">
  <xs:restriction base="xs:string">
    <xs:minLength value="6"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
```

Mithilfe der Angabe `<xs:enumeration>` kann man festlegen, dass ein Element nur konkrete Werte annehmen darf. Aufzählung

Das folgende Schema definiert einen Typ `kontaktType` mit den aufgeführten Werten: Beispiel

```
<xs:simpleType name="kontaktType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="freund"/>
    <xs:enumeration value="verein"/>
    <xs:enumeration value="beruf"/>
  </xs:restriction>
</xs:simpleType>
```

Diese Eigenschaften werden auch als einschränkende **Facetten** bezeichnet. Sie können jeweils für die angegebenen Datentypen verwendet werden: Facetten

- `xs:string` - `minLength`, `maxLength`, `enumeration`
- `xs:decimal` - `minInclusive`, `maxInclusive`, `enumeration`
- `xs:integer` - `minInclusive`, `maxInclusive`, `enumeration`
- `xs:float` - `minInclusive`, `maxInclusive`, `enumeration`
- `xs:date` - `minInclusive`, `maxInclusive`, `enumeration`

Attribute definieren

In einem XML-Schema können wie in einer DTD Attribute deklariert werden. Besitzt ein Element Attribute, muss es im XML-Schema als komplexes Element spezifiziert werden. Das gilt auch dann, wenn das Element selbst keine Kindelemente, sondern nur Attribute enthält. Für ein Element können beliebig viele Attribute definiert werden. Sie werden nacheinander aufgeführt, wobei die Reihenfolge keine Rolle spielt. Das bedeutet, dass die Attribute im XML-Dokument in anderer Reihenfolge vorkommen dürfen als im XML-Schema. Sie werden nach folgendem Schema definiert: Attribute

III 9 Statik

```
<xs:complexType name="...">
  <xs:sequence>    ...
</xs:sequence>
  <xs:attribute name="name1" type="..." use="..."/>
  <xs:attribute name="name2" type="..." use="..."/>
</xs:complexType>
```

Attributtypen Als Attributwerte sind nur einfache Datentypen zulässig, d.h. die Standardtypen oder die `simpleTypes`, die oben eingeführt wurden. Attribute können keine komplexen Strukturen bilden.

use Man kann mit `use="optional"` angeben, dass ein Attribut optional ist. Pflichtattribute müssen mit `use="required"` definiert werden. Erfolgt keine dieser Angaben, dann handelt es sich um ein optionales Attribut.

Beispiel Bei folgendem Ausschnitt aus einem XML-Dokument besitzt das strukturierte Element `adresse` die optionalen Attribute `erfasst_am` und `aktualisiert_am`:

```
<adresse erfasst_am="2005-07-17"
          aktualisiert_am="2006-09-17">
  <vorname>Marie</vorname>
  ...
</adresse>
```

Dieser XML-Ausschnitt wird durch folgendes Schema definiert:

```
<xs:element name="adresse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>    ...
    </xs:sequence>
    <xs:attribute name="erfasst_am" type="xs:date"/>
    <xs:attribute name="aktualisiert_am" type="xs:date"/>
  </xs:complexType>
</xs:element>
```

Voreinstellungen für Attribute Analog zu Elementen können auch für Attribute Voreinstellungen definiert werden. Wird dem Attribut im XML-Dokument kein Wert zugewiesen, dann gilt die Voreinstellung.

Beispiel Für das optionale Attribut `sprache` wird die Voreinstellung `deutsch` festgelegt:

```
<xs:complexType name="adresseType">
  <xs:sequence>
    <xs:element name="vorname" type="xs:string"
      minOccurs="0"/>    ...
  </xs:sequence>
  <xs:attribute name="sprache" type="xs:string"
    default="deutsch"/>
</xs:complexType>
```

Ist im XML-Dokument kein Wert angegeben, gilt diese Voreinstellung. Ebenso kann der voreingestellte Wert überschrieben werden. Gültige XML-Elemente sind:

- `<adresse sprache="englisch">... </adresse>`
- `<adresse>... </adresse>` : hier gilt per Voreinstellung `sprache="deutsch"`

XML-Schemata dienen einem ganz ähnlichen Zweck wie DTDs. Sie besitzen aber eine Reihe von Vorteilen und werden wohl die DTDs zukünftig ersetzen. Der größte Vorteil besteht darin, dass XML-Schemata selbst XML-Dokumente sind, die entsprechend auf Wohlgeformtheit geprüft werden können. Sie können genauso verarbeitet werden wie alle XML-Dokumente. Außerdem bieten XML-Schemata die Möglichkeit, Datentypen für Elemente und Attribute zu definieren sowie Wertebereiche und Muster für Zeichenfolgen zu spezifizieren. Diese Eigenschaften sind besonders wichtig, wenn es um die Kommunikation zwischen Anwendungen geht.

Schema vs. DTD

Zur Klassifikation

- XML-Dokumente kombiniert mit DTDs oder XML-Schemata werden textuell und formal beschrieben.
- Sie werden in allen Entwicklungsphasen für verschiedene Zwecke eingesetzt. Ein Einsatzbereich ist die Spezifikation von grafischen Benutzungsoberflächen – die Spezifikation geschieht in der Regel durch Grafikeditoren, die die Daten im XML-Format ablegen. Analog legen viele UML-Editoren ihre Daten ebenfalls im XML-Format ab. Mit Hilfe des Datenaustauschformats **XMI** ist es möglich, UML-Daten zwischen verschiedenen UML-Werkzeugen auszutauschen. Für die Spezifikation von Geschäftsprozessen (siehe »Geschäftsprozesse und *Use Cases*«, S. 250) wird ebenfalls oft XML als Spezifikationssprache verwendet, z. B. bei der Sprache **BPEL**. Zur Konfigurationsbeschreibung, z. B. bei Web-Anwendungen, wird ebenfalls XML eingesetzt.

9.5.2 Entity-Relationship-Modell

Mit dem **Entity-Relationship-Modell** (kurz: ER-Modell oder ERM) werden Datenmodelle spezifiziert, die dazu dienen, operative Datenbanken zu betreiben. Es wurde 1976 von P. Chen entwickelt [Chen76]. Im Deutschen wird manchmal auch die Bezeichnung Gegenstands-Beziehungs-Modell verwendet. Im Angelsächsischen ist auch der Begriff *Information Modeling* gebräuchlich. Ziel des ER-Modells ist es, die permanent gespeicherten Daten und ihre Beziehungen untereinander zu beschreiben. Die Analyse der Information erfolgt aus fachlogischer Sicht.

ER-Modell

III 9 Statik

Konzeptionelles
Modell

Es entsteht ein **konzeptionelles Modell**, das gegen Veränderungen der Funktionalität weitgehend stabil ist.

Historisch ist das ER-Modell vor den objektorientierten Konzepten entstanden. Da man heute das ER-Modell als einen Spezialfall der objektorientierten Konzepte ansehen kann, geht die folgende Darstellung des ER-Modells davon aus, dass die OO-Konzepte bekannt sind. Daher beschränkt sich die folgende Beschreibung auf die Besonderheiten des ER-Modells.

i Im Gegensatz zu objektorientierten Konzepten handelt es sich beim ER-Modell um ein Datenmodell, auf dem mit Standardoperationen gearbeitet wird:

■ »ER-Konzepte und OO-Konzepte im Vergleich«, S. 200

Im ER-Modell werden Entitäten in Tabellen gespeichert:

■ »Schlüssel, Tabellen und Dateien«, S. 204

ER-Modelle können um die Konzepte Vererbung und Aggregation erweitert werden:

■ »Beispiele für semantische Datenmodelle«, S. 207

Mit Hilfe von ER-Modellen können ganze Unternehmen und sogar »Welten« bezogen auf definierte Kontexte modelliert werden:

■ »Unternehmensdatenmodelle und Weltmodelle«, S. 209

Ein Quervergleich erleichtert die Übersicht:

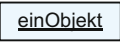


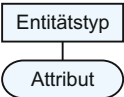

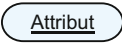
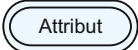
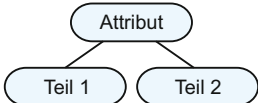



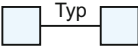
















■ »Zusammenfassung«, S. 213

9.5.2.1 ER-Konzepte und OO-Konzepte im Vergleich

Vereinfachend ausgedrückt, unterscheiden sich die ER-Konzepte von den OO-Konzepten dadurch, dass in den ER-Konzepten *kein* Verhalten modelliert wird, d. h. Operationen und Botschaften fehlen. Außerdem unterscheiden sich die Terminologie und die Notation von den OO-Konventionen. Die Abb. 9.5-1 und die Abb. 9.5-2 zeigen im Vergleich die wichtigsten Begriffe und Symbole. Anstelle von Multiplizitäten wird im ER-Modell von **Kardinalitäten** gesprochen. Im Folgenden werden beide Begriffe synonym verwendet.

Wie die Abbildungen zeigen, wurden die ursprünglichen Konzepte und die Notation von Chen inzwischen wesentlich erweitert. Man bezeichnet das ER-Modell ergänzt um die Konzepte Aggregation und Vererbung auch als **semantische Datenmodellierung**. In der Chen-Notation wird jede Assoziation durch eine Raute dargestellt. Um auch große Modelle darstellen zu können, entfällt bei den meisten heute verwendeten Notationen die Raute. Da in der Chen-Notation *keine* Unterscheidung zwischen Muss- und Kann-Beziehungen möglich ist, wurde die Notation ergänzt und modifiziert. Die Abb. 9.5-3 zeigt vergleichend weitere in der Praxis übliche Notationen.

9.5 Daten-Strukturen III

UML	ERM	Notation [Chen76]	Notationserweiterungen
Objekt 	Entität (entity)	kein Symbol	
Klasse 	Entitätstyp (entity type)		
Klassenextension (extent)	Entitätsmenge (entity set)		
Attribut	Attribut		
Attributtyp	Wertebereich, Domäne, Wertetypen (domain, value-set)		
—	Schlüsselattribut		
—	Mehrwertiges Attribut (kann ein oder mehrere Werte aus dem Wertebereich annehmen)		
Stereotyp «Structure»	Zusammengesetztes Attribut		
Verbindung (link) zwischen Objekten 	Beziehung (relationship)		
Assoziation 	Beziehungstyp, Assoziation (relationship type)		
Multiplizität    	Kardinalität, Komplexität, Stelligkeit     keine Unterscheidung zwischen Muss und Kann	MC-Notation    	Krähenfuß-Notation    

Bei der **MC-Notation** kann man das C als **choice** (wahlfrei) und das M als **multiple** (mehrfach) interpretieren.

Bei der **numerischen Notation** erfolgt die Multiplizitätsangabe in einer (min, max)-Notation. Sie besagt, in wie vielen Beziehungen eine Entität einer Entitätsmenge mindestens (min) und höchstens

Abb. 9.5-1: OO-Konzepte und OO-Notation im Vergleich zu ER-Konzepten und ER-Notationen.

III 9 Statik

UML	ERM	Notation [Chen76]	Notationserweiterungen
Reflexive Assoziation 	Rekursive Beziehung		
Assoziative Klassen 	Beziehungstyp mit eigenen Attributen		
Aggregation 	Aggregation ist-Teil-von-Beziehung		
Vererbung 	Vererbung ist-ein-Beziehung		

Abb. 9.5-2: OO-Konzepte und OO-Notation im Vergleich zu ER-Konzepten und ER-Notationen.

(max) vorkommt. In den ER-Grafiken wird auf die Klammerung der Zahlenpaare meistens verzichtet. Ist min=max, dann wird nur ein Wert angegeben.

In der **Krähfuß-Notation** (crow's foot notation) – auch Martin-Notation oder IE-Notation genannt – werden die Attribute – wie in der UML – in das Rechteck des Entitätstyps eingetragen – getrennt durch eine waagrechte Linie vom Namen des Entitätstyps.

Abb. 9.5-3: Alternative Darstellungsmöglichkeiten von Kardinalitätsangaben.

Chen-notation	MC-Notation	Krähfuß-notation	Pfeil-notation	Bachman-notation	Numerische Notation

Bei den meisten Notationen wird die Multiplizität einer Assoziation jeweils an genau der *gegenüberliegenden* Entitätsmenge angetragen – wie in der UML-Notation. Bei einigen Notationen wird die Multiplizität einer Assoziation aber umgekehrt, d. h. an derselben Seite angetragen, z. B. bei der numerischen Notation (Abb. 9.5-4).

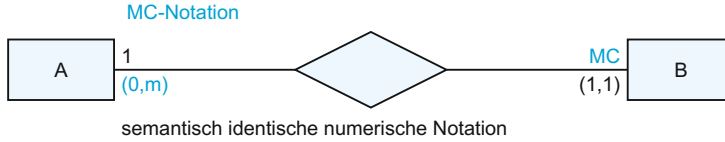


Abb. 9.5-4: MC-Notation vs. numerische Notation.

Beim Betrachten von ER-Diagrammen ist daher jeweils zu klären, welche Notation verwendet wird.

Bei Notationen, die die Buchstaben M, N (oder m, n) für die Obergrenze verwenden, bedeutet die Angabe eine unbestimmte Anzahl. In der Regel kennt man die konkrete Obergrenze auch nicht. Ist dies jedoch der Fall – oft bei technischen Systemen – dann lassen einige Notationen auch die Angabe konkreter Obergrenzen zu.

Die Abb. 9.5-5 zeigt einen Ausschnitt aus der Fallstudie »Seminarorganisation« in Chen-Notation mit Attributdarstellung. In der Regel lässt man die Attribute in ER-Diagrammen weg. Beim Einsatz von Werkzeugen können die Attribute wahlweise angezeigt oder wegblendet werden.

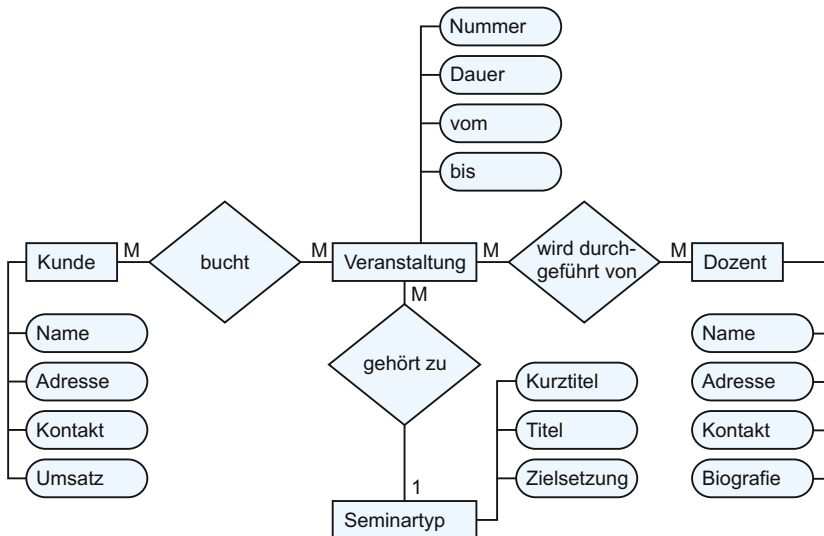


Abb. 9.5-5: Modellierung der »Seminarorganisation« (Ausschnitt) in Chen-Notation.

III 9 Statik

9.5.2.2 Schlüssel, Tabellen und Dateien

In der objektorientierten Welt besitzt jedes Objekt eine eigene **Objektidentität**, die unabhängig von den Attributwerten des Objekts ist.

Im ER-Modell verwendet man zur Identifikation von Entitäten **Schlüssel**. Man unterscheidet daher *beschreibende* Attribute, um die anwendungsrelevanten Eigenschaften der Entitäten festzuhalten, und *identifizierende* Attribute, die Schlüssel zur eindeutigen Identifikation einer Entität innerhalb ihrer Entitätsmenge bilden. Ein Schlüssel kann aus einem oder mehreren identifizierenden Attributen zusammengesetzt sein.

Beispiel Für eine Entitätsmenge lässt sich oft auch mehr als ein Schlüssel angeben, z. B.

Entitätsmenge: Stadt

Attribute: PLZ, Staat, Einwohnerzahl, Vorwahl

Schlüssel: PLZ und Staat oder Vorwahl

In einem solchen Fall wird stets ein Schlüssel als **Primärschlüssel** ausgezeichnet. Von einem Schlüssel wird Minimalität verlangt:

Falls die Attributkombinationen K1 und K2 beide eine Entität identifizieren können, aber K1 in K2 enthalten ist, dann ist nur K1 Schlüssel, K2 hingegen ist ein Schlüsselkandidat.

Ein **Schlüssel** K ist stets eine *minimale*, identifizierende Attributkombination. Jede echte Obermenge von K ist ein Schlüsselkandidat. Schlüssel werden unterstrichen dargestellt.

Beispiel Für Stadt ist {Staat, Vorwahl} Schlüsselkandidat, Vorwahl ist Schlüssel, {PLZ, Staat} ist ebenfalls Schlüssel.

Minimal bedeutet, dass die eindeutige Identifizierbarkeit verloren geht, wenn ein Attribut entfernt wird. Es ist möglich, dass ein fachlich notwendiges Attribut gleichzeitig als Schlüssel fungiert (z. B. Matrikelnummer). Andernfalls muss beim ER-Modell ein *künstliches* Schlüsselattribut hinzugefügt werden.

Tabellen Entitätstypen mit ihren Entitäten lassen sich durch **Tabellen** darstellen.

Beispiel 1a: Jeder Kunde kann Mitarbeiter einer Firma sein. Sonst ist er ein Privatkunde. Zu jeder Firma sind u. a. der Firmenkurzname, der Firmenname und der Umsatz zu speichern. Der Firmenkurzname ist das Schlüsselattribut.

SemOrg

Der Entitätstyp Firma mit zwei Entitäten sieht in Tabellenform wie in der Abb. 9.5-6 aus, wobei der Schlüssel unterstrichen dargestellt ist.

9.5 Daten-Strukturen III

Name des Entitätstyps		Attribute		
Schlüssel				
Firma	<u>Kurzname</u>	Name	Adresse	Umsatz
Innosoft	Innovative Software GmbH	20.800,-	Entitäten
Hardsoft	Hard & Soft KG	33.200,-	

Abb. 9.5-6: Aufbau der Tabelle Firma.

Solche Tabellen können in Dateien gespeichert werden. Es gilt folgende Regel:

- Für jeden Entitätstyp wird ein Speicher bzw. eine Datei benötigt. Jede Entität des entsprechenden Entitätstyps stellt einen Eintrag in diese Datei dar.

Regel 1

Das ER-Modell der Abb. 9.5-7 soll in Tabellen umgewandelt werden.

Beispiel 1b: SemOrg



Abb. 9.5-7: Assoziation zwischen Kunde und Firma.

Es werden zwei Tabellen benötigt. Die in Beispiel 1a angegebene Tabelle bleibt unverändert. Die Abb. 9.5-8 zeigt die Kundentabelle.

		zusätzliches Attribut (Fremdschlüssel)		
Kunde	<u>Personal-Nr.</u>	Funktion	Umsatz	Kurzname
10	Berater	5.200,-	InnoSoft	HardSoft
20	Systemanalytiker	10.300,-	HardSoft	

Abb. 9.5-8: Aufbau der Tabelle Kunde.

Es gilt folgende Regel:

- Sind zwei Entitätstypen A und B durch einen 1:1- oder M:1-Beziehungstyp verbunden, dann wird der Primärschlüssel von B als so genannter **Fremdschlüssel** in A eingetragen, d.h. als zusätzliches Attribut.

Regel 2

Zwischen Kunde und Veranstaltung besteht der M:M-Beziehungstyp bucht (Abb. 9.5-9).

Beispiel 1c: SemOrg

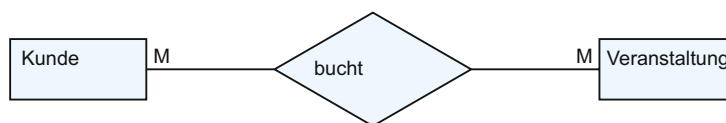


Abb. 9.5-9: Assoziation zwischen Kunde und Veranstaltung.

Eine Transformation in Tabellen erfordert drei Tabellen. Die Tabelle Kunde (Beispiel 1b) bleibt unverändert. Die Tabelle Veranstaltung und die Tabelle bucht zeigt die Abb. 9.5-10.

III 9 Statik

Abb. 9.5-10:
Aufbau der
Tabellen

Veranstaltung und
bucht.

Veranstaltung	<u>Veranstaltungs-Nr.</u>	Dauer	Vom	Teiln. aktuell
	22	3	01.03.01	15
	94	2	04.07.01	8
	37	1	10.10.01	128

Für den Beziehungstyp *bucht* wird eine eigene Tabelle benötigt:

	Schlüssel, bestehend aus zwei Schlüsselattributen	
bucht	<u>Personal-Nr.</u>	<u>Veranstaltungs-Nr.</u>
	10	94
	10	22
	27	37

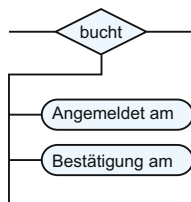
Der erste Eintrag bedeutet, dass der Kunde mit der Personal-Nr. 10 die Veranstaltung mit der Nummer 94 gebucht hat. Beide Attribute bilden einen gemeinsamen Schlüssel für die Einträge in dieser Tabelle.

Es gilt folgende Regel:

- Regel 3 ■ Sind zwei Entitätstypen A und B durch einen M:N-Beziehungstyp verbunden, dann wird für den Beziehungstyp eine eigene Tabelle angelegt. Als Attribute werden die Schlüssel der Entitätstypen verwendet, die der Beziehungstyp verbindet. Beide Schlüsselattribute zusammen bilden den Primärschlüssel dieser Tabelle.

Attributen einer Beziehung kann man ebenfalls Attribute zuordnen, wenn dies fachlich notwendig ist. Diese Attribute treten nur auf, wenn zwischen zwei Entitäten eine entsprechende Beziehung besteht.

Abb. 9.5-11:
Attribute zu bucht.

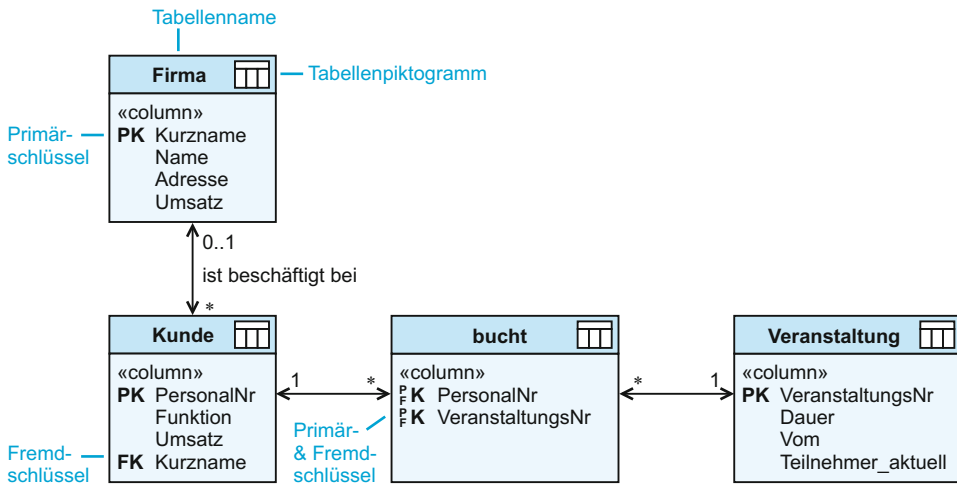


Für die Seminarorganisation ist es wichtig zu wissen, wann ein Kunde eine Veranstaltung bucht (Angemeldet am), wann der Kunde die Buchungsbestätigung erhalten hat (Bestätigung am), ob er sich wieder abgemeldet hat, ob er eine Änderungsmitteilung erhalten und wann er die Rechnung zugeschießt bekommen hat. Diese Attribute werden in die Tabelle bucht (siehe Beispiel 1c) als zusätzliche Attribute aufgenommen (Abb. 9.5-11).

Erweiterte UML-Notation

In der UML kann man Tabellen, Primär- und Fremdschlüssel usw. nicht darstellen. In [NaMa01, S. 119–147] wird ein **UML-Profil** für den Datenbank-Entwurf vorgestellt. Mit den UML-Erweiterungsmechanismen **Stereotypen** und **Tagged Values** werden erweiterte Dia-

grammelemente definiert (siehe »Basiskonzepte«, S. 99). Die Abb. 9.5-12 gibt einen Überblick über einige dieser neuen Elemente am Beispiel der Fallstudie SemOrg (Beispiele 1a bis 1c).



Viele UML-Werkzeuge unterstützen dieses Profil.

Abb. 9.5-12:
Tabellennotation
in einer
erweiterten UML.

9.5.2.3 Beispiele für semantische Datenmodelle

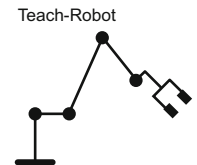
Verwendet man die semantische Datenmodellierung als Erweiterung der ER-Modellierung für die fachliche Lösung, dann erhält man als Beispiele die Modellierungen der Abb. 9.5-13 und der Abb. 9.5-14.

Im Gegensatz zu kaufmännisch/administrativen Anwendungen, wie der Fallstudie »Seminarorganisation«, besitzt bei technischen Systemen die Multiplizität oft eine feste Obergrenze.

Die in den Funktionsbäumen (siehe »Funktionsbaum«, S. 143) beschriebenen Funktionen benutzen die in ER-Diagrammen modellierten Datenstrukturen, um ihre Aufgabe zu erfüllen. Eine Möglichkeit, den Zusammenhang zwischen beiden Sichten herzustellen, ist die Erstellung einer Assoziationsmatrix. In der Abb. 9.5-15 werden die Funktionen den Dateien gegenübergestellt.

Ein semantisches Datenmodell soll für das zu entwickelnde Softwaresystem die relevanten Entitätsmengen und die Beziehungen zwischen den Entitätsmengen mit ihrer fachlichen Bedeutung darstellen. Folgende Überprüfungen sind durchzuführen:

- Besitzt jeder Entitätstyp mindestens ein Attribut? Ist dies nicht der Fall, dann liegt kein Entitätstyp vor.
- Sind die Entitätstypen durch Substantive, die Beziehungstypen durch Verben beschrieben? Ist dies nicht der Fall, dann sind die Beziehungstypen zu überprüfen.



Zusammenhang
mit Funktions-
bäumen

QS-Checkliste
Semantisches
Datenmodell



QS-Checkliste
Semantisches
Datenmodell



III 9 Statik

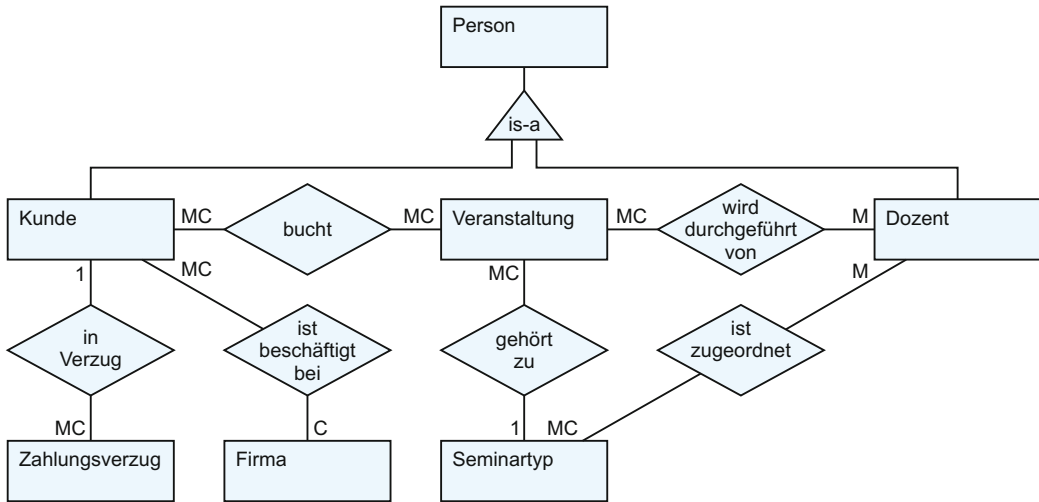


Abb. 9.5-13:
Semantisches
Datenmodell der
Fallstudie »Semi-
narorganisation«
(MC-Notation).

■ Verwenden in einer *is-a*-Beziehung alle Spezialisierungstypen alle Attribute des Generalisierungstyps? Ist dies nicht der Fall, dann ist die *is-a*-Beziehung zu überprüfen.

■ Sind zwei Entitätstypen identisch? Identität kann vorliegen, wenn eine oder mehrere der folgenden Bedingungen erfüllt sind:

- ☐ Die Entitätstypen stehen in einer 1:1-Beziehung.
- ☐ Sie sind durch dieselben Beziehungstypen mit der Umgebung verbunden.

☐ Sie besitzen dieselben Schlüsselattribute.

☐ Sie besitzen dieselben Attribute.

■ Jeder Beziehungstyp ist zu überprüfen auf

☐ seine Notwendigkeit, d. h. bringt er neue Informationen in das Modell?

☐ die korrekte Darstellung des Sachverhalts.

■ Liegt ein Entitätstyp oder ein Attribut vor?

Eine Entität muss eindeutig identifizierbar sein und durch Attribute beschrieben werden. Farbe ist ein Attribut vom Auto. Würde aber die Zusammensetzung von Farbe interessieren, so wäre sie ein Entitätstyp. Abhängig vom Blickwinkel können Attribute zu Entitätstypen werden und umgekehrt.

Bewertung + Im kaufmännischen Anwendungsbereich ist eine semantische Datenmodellierung ein absolutes Muss!

+ Auch in vielen technischen Bereichen ist die Komplexität der Daten so groß, dass ein semantisches Datenmodell erforderlich ist (siehe Roboter-Modellierung).

+ Voraussetzung für einen relationalen Datenbankentwurf.

– Dynamische Aspekte wie in der OO-Welt können nicht dargestellt werden.

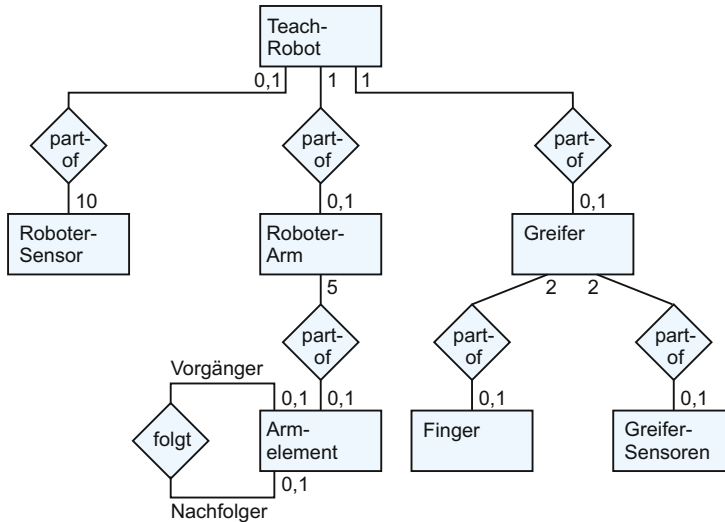


Abb. 9.5-14:
Semantisches
Datenmodell eines
Roboters
(numerische
Notation).

- Semantische Datenmodelle können sehr umfangreich werden und sind dann schwer zu überblicken. Es fehlt ein Verfeinerungsmechanismus, um mehrere Abstraktionsebenen bilden zu können.

Heute sollten zunächst die OO-Konzepte zur Modellierung verwendet werden, da die OO-Konzepte allgemeiner sind. Falls erforderlich, kann dann durch Weglassen der dynamischen Anteile eine Reduktion auf ein semantisches Datenmodell vorgenommen werden.

Empfehlung

9.5.2.4 Unternehmensdatenmodelle und Weltmodelle

In der Fallstudie »Seminarorganisation« wurde ein Teilausschnitt der Firma Teachware modelliert. Beispielsweise können auch die Mitarbeiter von Teachware durch ein separates Anwendungssystem verwaltet werden (Lohn- und Gehaltsabrechnung).

Bei größeren Unternehmen gibt es eine Vielzahl von Anwendungssystemen, die Teilbereiche verwalten. Betrachtet man jeweils nur den Bereich eines Unternehmens, der durch ein entsprechendes Anwendungssystem bearbeitet wird, dann führt dies zu folgenden Problemen:

- Mehrfachverwaltung der gleichen Datenbestände,
- Brüche bei der Abwicklung von übergreifenden Geschäftsprozessen,
- inkompatible Informationsflüsse.

Um diese Probleme zu vermeiden, ist es das Ziel, ein umfassendes **Unternehmensdatenmodell** aufzustellen. Es soll die Informationsstrukturen und Geschäftsprozesse aller Bereiche eines Unternehmens unter Berücksichtigung der Schnittstellen zueinander in

Unternehmens-
datenmodell

III 9 Statik

Abb. 9.5-15:
Assoziationsmatrix
zwischen
Funktionen und
Daten (Ausschnitt).

Funktionen	Daten	Kunde	Firma	Zahlungsverzug	bucht	Veranstaltung	wird durchgeführt von	Dozent	ist zugeordnet	Seminartyp
erfasse Kundenstammdaten	c									
aktualisiere Kundenstammdaten	u									
lösche Kundenstammdaten	d									
erstelle Adressaufkleber	r									
trage Zahlungsverzüge ein			c							
erfasse Firmenstamm		c								
aktualisiere Firmenstamm		u								
lösche Firmenstamm		d								
beantworte Fragen	r	r								
erstelle Anmeldebestätigung	r	r	r	r	r				r	
erstelle Rechnung	r	r	r	r	r				r	
erstelle Rechnungskopie	r	r	r	r	r				r	
erstelle Mitteilung	r	r	r	r	r	r	r	r	r	
erstelle Abmeldebestätigung	r	r	r	r	r				r	
erstelle korrigierte Rechnung	r	r	r	r	r				r	
erstelle korrigierte Rechnungskopie	r	r	r	r	r				r	
erstelle Mitteilung	r	r	r	r	r				r	
erstelle Stornierungsmittelung	r	r	r	r	r	r	r		r	
erstelle korrigierte Rechnung	r	r	r	r	r				r	
erstelle korrigierte Rechnungskopie	r	r	r	r	r				r	

Legende: c = Create; u = Update; r = Read; d = Delete

einheitlicher und ganzheitlicher Form darstellen. Zur Beschreibung von Unternehmensdatenmodellen werden heute noch semantische Datenmodelle verwendet.

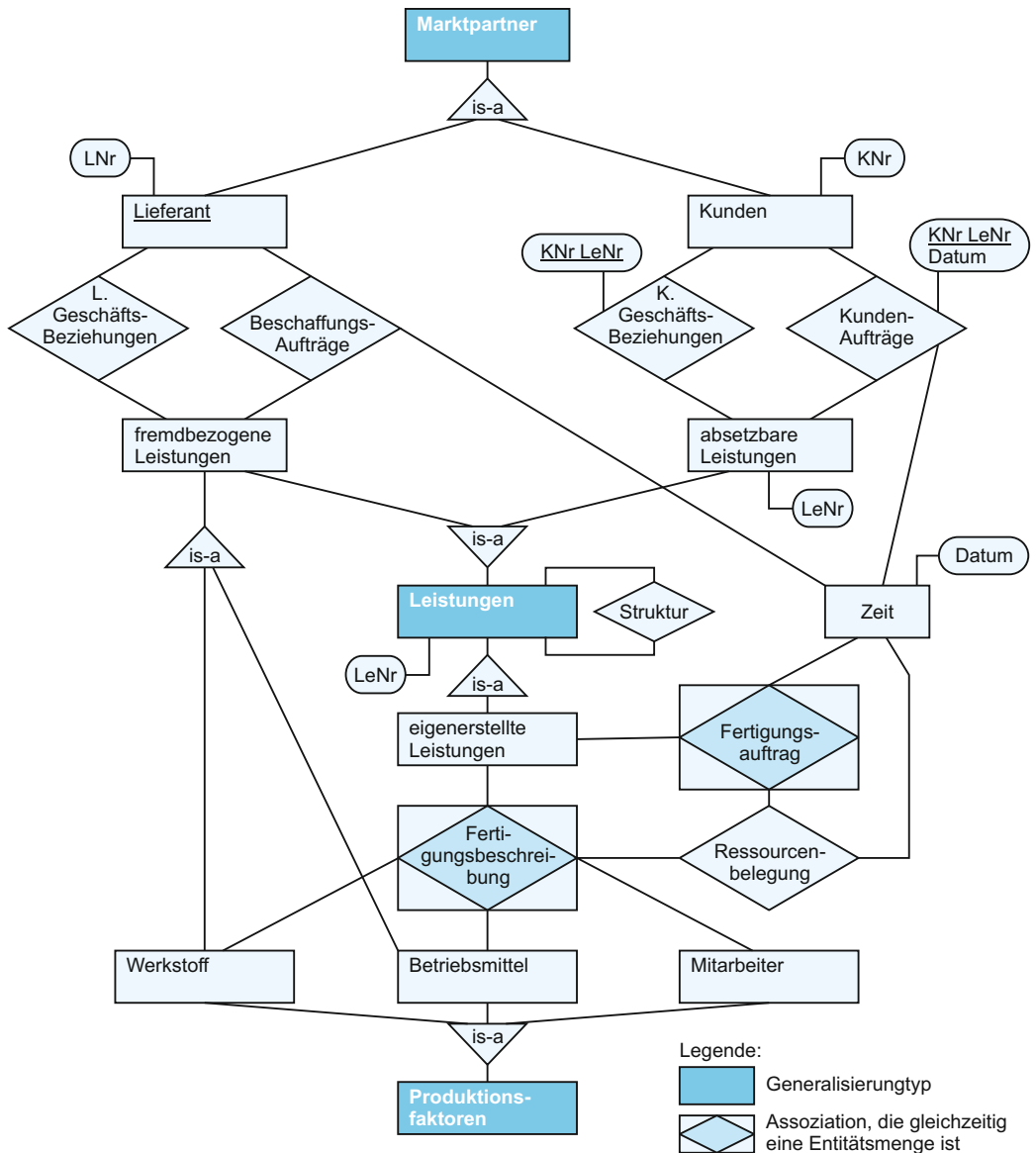
Je nach Abstraktionsebene und Verdichtungsgrad können Datenmodelle unterschiedlicher Ebenen definiert werden.

Ein **strategisches Datenmodell** enthält nur die wichtigsten Entitätstypen, Assoziationen, Aggregationen und Generalisierungen. Strategische Datenmodelle enthalten nur grundsätzliche Branchenunterschiede, keine detaillierten Unterschiede. Ein solches Datenmodell umfasst in der Regel rund 20 bis 30 Entitätstypen, Assoziationen, Aggregationen und Generalisierungen.

Die nächste Abstraktionsebene zeigt bereits unterschiedliche Strukturen innerhalb einer Branche. Ein solches Modell umfasst ca. 200 bis 500 Entitätstypen und Verknüpfungen. A.-W. Scheer hat in dem Buch »Wirtschaftsinformatik« ein Unternehmensmodell für Industriebetriebe auf dieser Ebene entwickelt [Sche90a]. Es enthält rund 300 Entitätstypen und wurde bereits mehrfach bei der praktischen Aufstellung von unternehmensweiten Datenmodellen als Referenzmodell eingesetzt. Das Modell benötigt ein Blatt der Größe DIN A1.

In [BiRo90] ist das Datenmodell für ein Versicherungsunternehmen dargestellt. In [Sche90b] wird die Grobstruktur eines Unternehmensdatenmodells gezeigt (Abb. 9.5-16).

Literatur



In [Sche90a] wird beschrieben, in welchen Schritten ein Unternehmensdatenmodell entwickelt werden kann.

Abb. 9.5-16: Grobstruktur eines Unternehmensdatenmodells [Sche90b].

III 9 Statik

Eine weitere Verfeinerung eines solchen Modells führt zu einem unternehmensindividuellen Datenmodell mit einem Umfang von 300 bis 1500 Entitätstypen und Assoziationen.

Bei der Modellierung von Unternehmensdatenmodellen wird intensiv die Generalisierungshierarchie verwendet.

Betrachtet man die Robotermodellierung (siehe »Beispiele für semantische Datenmodelle«, S. 207), dann zeigt dieses ER-Modell sicher nur einen Teilaspekt, wenn es um die Steuerung eines Roboters in einer Fertigungsumgebung geht. Folgende Sichten müssen modelliert werden [DiHu91, S. 70]:

- Die Umweltsicht (oberste Beschreibung der Roboterzelle),
- die Robotersicht (Modellierung des Betriebsmittels Roboter),
- die Montagesicht (Beschreibung von Montagefolge und Bewegungssegment),
- die Sicht der Handlungsobjekte (Beschreibung von Produkten) und
- die Konstruktionssicht (Konstruktionsmodell physikalischer Objekte).

Weltmodell Die Zusammenfassung dieser verschiedenen Einzelsichten ergibt ein **Weltmodell** der Fertigungsumgebung, der Betriebsmittel und Werkstücke und der Produktionsdaten. In dem Buch »Informationsverarbeitung in der Robotik« beschreiben die Autoren R. Dillmann und M. Huck die einzelnen Sichten eines solchen Weltmodells. Es besteht aus ungefähr 100 Entitätstypen und ihren Assoziationen.

Bei der Zusammenfassung von Teilmodellen zu einem Gesamtmodell muss Folgendes beachtet werden:

- Erkennung und Auflösung von Synonymen (verschiedene Benennungen für den gleichen Sachverhalt),
- Erkennung und Auflösung von Homonymen (Gleichbenennung unterschiedlicher Sachverhalte).

Diese beiden Punkte beziehen sich sowohl auf Entitätstypen und Assoziationen als auch auf Attribute.

- Ermittlung nicht erfasster Beziehungen zwischen den Einzelsichten,
- Zusammenfassen unterschiedlicher Entitätstypen für identische Informationseinheiten.

Top-down vs. Bottom-up Bei der Entwicklung eines Unternehmensdatenmodells oder eines Weltmodells kann man bei den generellen Begriffen beginnen und durch eine zunehmende Spezialisierung die Begriffe in feinere Begriffe aufspalten (Top-down-Methode). Bei der Generalisierung werden zunächst die auf einer detaillierten Ebene eingeführten Begriffe verallgemeinert (Bottom-up-Methode). In der Praxis wird oft eine Mischung beider Methoden verwendet (siehe auch »Methoden«, S. 53).

9.5.2.5 Zusammenfassung

Das ER-Modell (ERM, Entity-Relationship-Modell) erlaubt die Modellierung von permanent zu speichernden Datenstrukturen und ihren Beziehungen zueinander. Entitäten (*entities*) werden zu Entitätstypen (*entity types*) zusammengefasst. Gleichrangige, fachliche Zusammenhänge zwischen Entitäten werden durch Beziehungen (Assoziationen, *relationships*) beschrieben, die zu Beziehungstypen (*relationship types*) zusammengefasst werden. Durch Rollen wird angegeben, welche Funktion eine Entität in einer Beziehung spielt. Die Eigenschaften von Entitäten und Beziehungen werden durch Attribute beschrieben. Jede Entität muss durch einen eindeutigen Schlüssel identifizierbar sein. Der Komplexitätsgrad einer Assoziation wird durch die Kardinalität (Multiplizität in der UML) angegeben (Muss- oder Kann-Beziehung, variable oder feste Obergrenze). Eine rekursive bzw. reflexive Beziehung liegt vor, wenn ein Entitätstyp mit sich selbst in Beziehung steht. Das Ergebnis einer ER-Modellierung bezeichnet man als konzeptionelles Modell.

ER-Modell

Die semantische Datenmodellierung erweitert das ER-Modell um die Aggregation und die Vererbung. Bei der Aggregation handelt es sich um einen Spezialfall der Assoziation. Es wird eine Ist-Teil-von-Hierarchie (*is-part-of*) beschrieben. Eine Vererbung erlaubt es, gemeinsame Attribute verschiedener Entitätstypen (*subtypes*) in neuen, übergeordneten Entitätstypen (*supertypes*) zusammenzufassen (is-a-Hierarchie). Diese gemeinsamen Attribute werden an die untergeordneten Entitätstypen vererbt. Die beiden Konzepte Aggregation und Generalisierung werden insbesondere für Unternehmensdatenmodelle benötigt.

Semantische Datenmodellierung

Entitätstypen und Beziehungstypen können auf Tabellen abgebildet werden. Die Entitäten werden als Zeilen in die Tabellen eingetragen. Alle Entitäten einer Tabelle bilden eine Entitätsmenge (*entity set*). Einen Überblick über die Notationen und die Elemente gibt die Abb. 9.5-17.

Abbildung auf Tabellen

Vereinfacht lässt sich sagen, dass ein Klassendiagramm ohne Operationen und Botschaften einem ER-Modell entspricht. Allerdings müssen in einem ER-Modell Schlüssel-Attribute für jeden Entitätstyp ergänzt werden. Ein UML-Profil für den Datenbank-Entwurf wird in [NaMa01, S. 119–147] beschrieben.

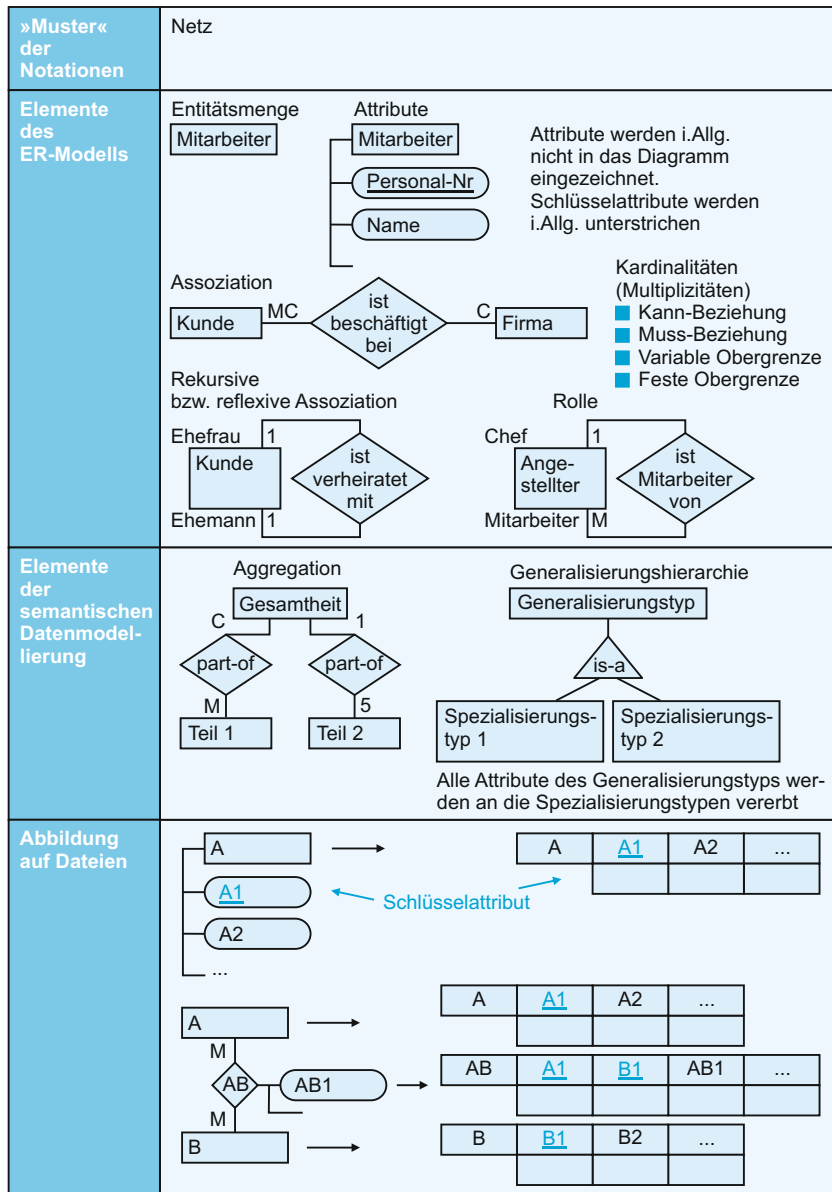
Zusammenhang mit OO-Konzepten

Zur Klassifikation

- Das ER-Modell und die semantische Datenmodellierung werden grafisch mit semiformalen Annotationen dargestellt.
- Einsatz in der Anforderungsspezifikation und im Entwurf.
- Sie werden vor allem in kaufmännisch/administrativen Anwendungen verwendet, z. T. aber durch OO-Konzepte abgelöst.

III 9 Statik

Abb. 9.5-17:
Notationen und
Elemente des ER-
Modells und der
semantischen Da-
tenmodellierung.



9.5.3 Multidimensionale Datenstrukturen

Mit dem ER-Ansatz (»Entity-Relationship-Modell«, S. 199) werden Datenmodelle für operative Datenbanken spezifiziert. In einer operativen Datenbank werden Daten und ihre Beziehungen so gespeichert, dass sie für den täglichen Einsatz in einem Unternehmen effizient verwendet werden können. Täglicher Einsatz bedeutet, dass Daten erfasst, geändert, gelesen und gelöscht werden und den aktuellen

Stand des Geschäftsverlaufs wiedergeben. Ein Beispiel dafür ist die Verwaltung der Firma Teachware mit dem Anwendungssystem »Seminarorganisation«.

In der Praxis hat sich gezeigt, dass in Unternehmen ein zunehmender Bedarf besteht, nicht nur auf die aktuellen Daten zuzugreifen, sondern Auswertungen über einen längeren Zeitraum hinweg vorzunehmen, um z. B. Trends festzustellen. Für die Firma Teachware ist es z. B. wichtig zu wissen, welche Seminartypen an welchen Veranstaltungsorten in den letzten drei Jahren am erfolgreichsten waren. Es hat sich herausgestellt, dass solche Datenanalysen nicht effizient durchgeführt werden können, wenn ein normales ER-Modell zugrunde gelegt wird, d. h. die Daten müssen für eine »optimale« Datenanalyse anders organisiert werden. Ziel ist es, die verfügbaren quantitativen Daten für die Analyse so anzuordnen, dass sie dem intuitiven Geschäftsverständnis des Managers möglichst weitgehend entsprechen. Hinzu kommt, dass intensive Datenanalysen auf operativen Datenbanken nicht zeitoptimal ausgeführt werden können und durch solche Analysen die Leistungsfähigkeit operativer Datenbanken herabgesetzt wird. Außerdem fehlen in operativen Datenbanken oft die historischen Datenbestände. In großen Unternehmen gibt es zudem meist mehrere operative Datenbanken unterschiedlicher Struktur, sodass eine Datenanalyse über alle Datenbanken hinweg schwierig ist.

Datenanalyse

Die Basis für Datenanalysen bilden spezielle Datenbanken:

■ »Data Warehouse und Data Marts«, S. 215

Damit die Daten analytisch ausgewertet werden können, müssen sie multidimensional strukturiert werden:

■ »OLAP und Hyperwürfel«, S. 217

Es gibt verschiedene Ansätze, um Daten für Analysen geeignet zu modellieren und auf Tabellen abzubilden:

■ »Modellierungsansätze«, S. 222

Zusammenfassend wird ein Überblick gegeben:

■ »Zusammenfassung«, S. 226

Ein Beispiel für weiterführende Literatur ist [GGP09], zu dem auch ein E-Learning-Kurs verfügbar ist.

i

Literatur

9.5.3.1 Data Warehouse und Data Marts

Ein *Data Warehouse* ist eine Datenbank, die Daten aus einer oder mehreren operativen Datenbanken sowie externen Datenbanken erhält, integriert und aggregiert und damit eine effiziente Datenanalyse ermöglicht. Der »Vater des *Data Warehouse*«, W. Inmon [Inmo96], gibt folgende Definition:

warehouse =
Lagerhaus,
(Waren-) Lager,
Speicher,
Bewahranstalt,
Wohnsilo

III 9 Statik

Definition

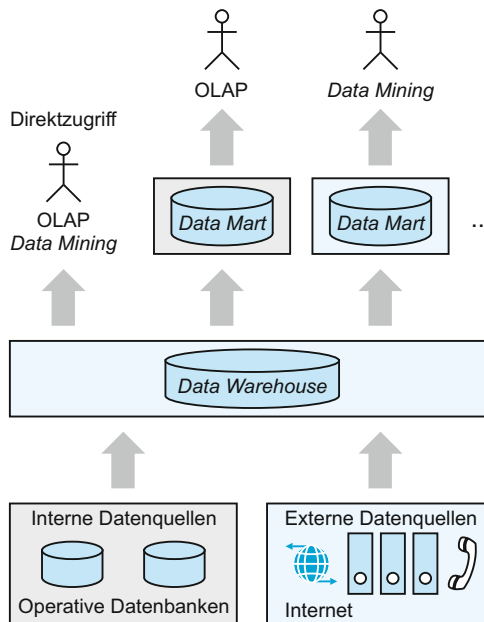
Ein **Data Warehouse** ist eine themenorientierte, integrierte, zeitabhängige, nichtflüchtige Datensammlung zur Unterstützung von Managemententscheidungen.

Im *Data Warehouse* sind alle für Analysen notwendigen Daten eines Unternehmens gespeichert. Um einzelnen Anwendergruppen, Bereichen oder Abteilungen nur die Daten zur Verfügung zu stellen, die sie für ihre Aufgaben benötigen, sind die *Data Marts* entstanden.

mart = Markt,
Handelszentrum,
Marktplatz

Data Marts stellen funktionsbereichs- oder personengruppenspezifische Extrakte aus der *Data Warehouse*-Datenbasis zur Verfügung. Sie erlauben dadurch dezentrale fachliche Sichten auf das zentrale *Data Warehouse*. Die Abb. 9.5-18 zeigt die grundsätzliche Architektur von *Data Warehouse*-Umgebungen.

Abb. 9.5-18:
Grundsätzliche
Architektur von
Data Warehouse-
Umgebungen.



Zwei wichtige Ansätze für die Auswertung der Daten sind OLAP und *Data Mining*.

OLAP (**Online Analytical Processing**) erlaubt schnelle und interaktive Zugriffe auf relevante Informationen in multidimensionale Strukturen. Bei multidimensionaler Aufbereitung (siehe »OLAP und Hyperwürfel«, S. 217) können die Daten vom Benutzer schneller und aussagekräftiger zur Entscheidungsfindung analysiert werden.

Data Mining bezeichnet die automatische Suche nach verborgenen Zusammenhängen in großen Datenbeständen. In einem möglichst großen und umfassenden Datenbestand wird automatisch eine Vielfalt von Zusammenhängen entdeckt, die dann noch bezüglich ihrer Aussagekraft und Relevanz interpretiert werden müssen.

mining = Bergbau

Die Abb. 9.5-19 zeigt, welche Daten über den Kunden Schulz in den verschiedenen Datenspeichern aufbewahrt werden. In der operativen Datenbank steht der aktuelle Umsatz, im *Data Warehouse* stehen die jeweiligen Quartalsumsätze für einen festgelegten Zeitraum. Im *Data Mart* werden die aufsummierten Quartalsumsätze zu Verfügung gestellt. Durch Ad-hoc-Abfragen bekommt der Kundensachbearbeiter spezifische Informationen temporär zur Verfügung gestellt.

Beispiel:
SemOrg

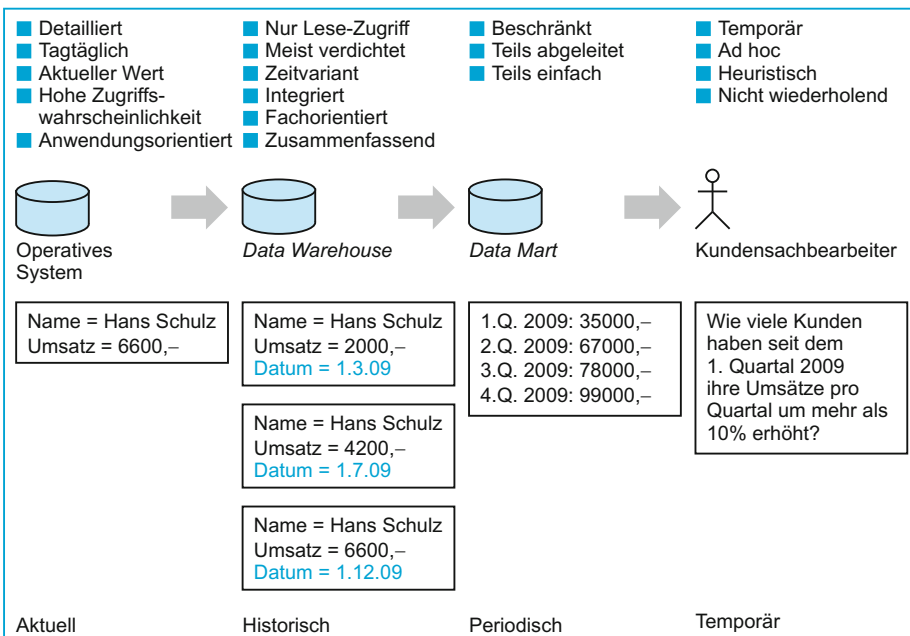


Abb. 9.5-19: Datencharakteristika von operativen Datenbanken, Data Warehouses und Data Marts.

9.5.3.2 OLAP und Hyperwürfel

OLAP-Systeme sollen

- Managern und qualifizierten Mitarbeitern aus den Fachabteilungen die Möglichkeit geben, schnelle, interaktive und komplexe Analysen durchzuführen,
- die gesammelten Daten aus möglichst vielen verschiedenen Perspektiven zeigen und
- die Analyseergebnisse in akzeptabler Zeit auch auf einer sehr großen Datenbasis bereitstellen.

III 9 Statik

Multi-
dimensionalität

Diese Ziele können durch eine Anordnung der Daten in multidimensionalen Datenstrukturen gut erreicht werden. Multidimensionalität bedeutet hierbei, dass quantitative **Kennzahlen** (*facts*) bzw. betriebswirtschaftliche Variablen, z. B. Umsatz- oder Kostengrößen, die durch mehrere sachliche Kriterien, wie z. B. Perioden, Kunden, Artikel, Niederlassungen oder Regionen, beschreibbar sind, in mehrdimensionalen Datencontainern angeordnet sind. Diese Größen lassen sich dann als Sammlung von Würfeln veranschaulichen, wobei die einzelnen **Dimensionen** durch die zugehörigen textindizierten Würfelkanten repräsentiert werden.

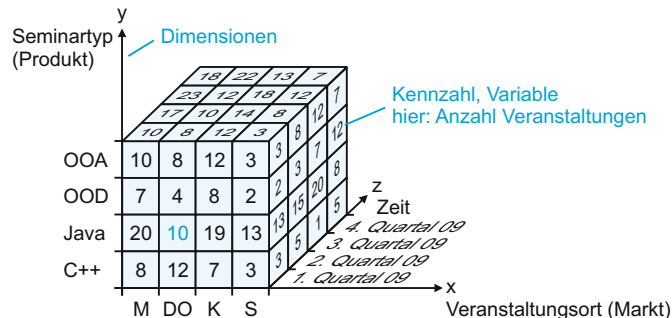
Hyper = über das
normale, übliche
Maß weit
hinausgehend

Der Begriff des Datenwürfels (*data cube*) hat sich im allgemeinen Sprachgebrauch durchgesetzt. Allerdings sind die üblichen Datenstrukturen nicht auf drei Dimensionen begrenzt und auch die Würfelkanten müssen nicht gleich lang sein. Wegen dieser Besonderheiten spricht man daher auch von einem **Hyperwürfel** (*hyper cube*).

Beispiel:
SemOrg

Der Seminarsachbearbeiter möchte feststellen, welche Seminartypen an welchen Veranstaltungsorten am erfolgreichsten sind. Die Abb. 9.5-20 zeigt einen 3D-Datenwürfel, der die relevanten Dimensionen modelliert.

Abb. 9.5-20:
Beispiel für einen
dreidimensionalen
Hyperwürfel.



In diesem *Data Warehouse* sind die Daten über erfolgte Veranstaltungen abgelegt. Die Dimensionen sind Seminartyp, Veranstaltungsort und Zeit. Die Kennzahl gibt die Anzahl der Veranstaltungen an.

Die Dimensionen bilden die Achsen des Koordinatensystems. Durch die betrachteten Attributwerte (*dimension elements*) bzw. Kombinationen hiervon wird der Würfel in Zellen zerlegt. Jeder Zelle wird eine Kennzahl als Funktionswert in Abhängigkeit der drei Dimensionen zugeordnet.

In Dortmund fanden 2009 beispielsweise 10 Veranstaltungen zu dem Seminartyp Java statt.

Jede Kennzahl ist charakterisiert durch ein (quantitatives) Maß (*measure*), hier Anzahl Veranstaltungen.

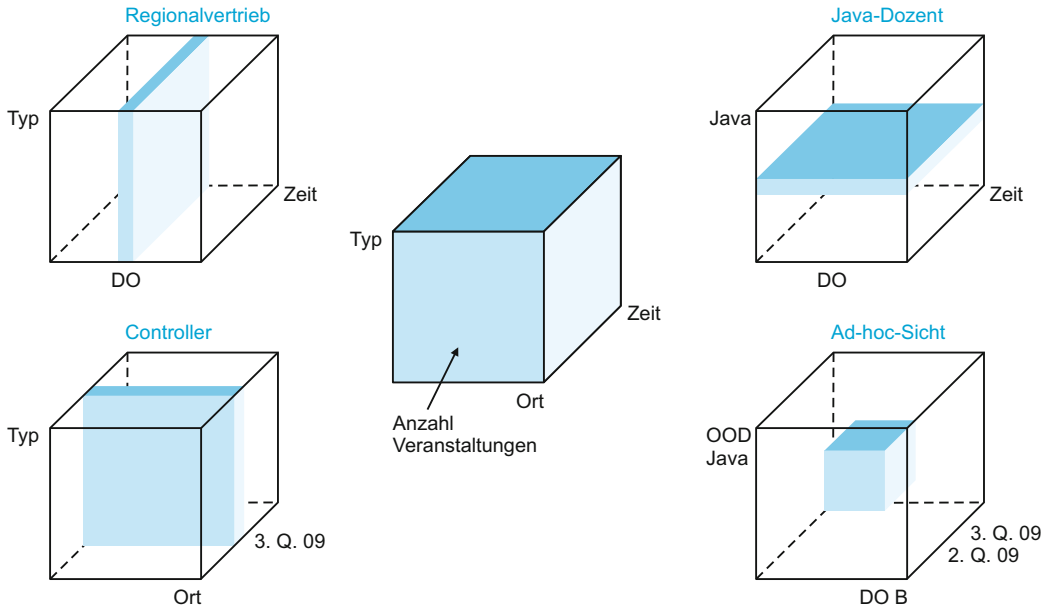
Für detaillierte Fragestellungen des Benutzers stehen verschiedenartige Operationen zur Manipulation des Hyperwürfels zur Verfügung. Die Operationen ermöglichen im Wesentlichen einen Wechsel von Dimensionen und Verdichtungsstufen. Die wichtigsten Operationen sind:

- *slice* (in vertikale oder horizontale Schichten schneiden)
- *dice* (verschiedene Sichten auf die Daten bzw. Würfel oder Teilwürfel)
- *drill down* bzw. *roll up* (Wechsel in niedere bzw. höhere Verdichtungsstufe)

Operationen auf dem Hyperwürfel

Die Abb. 9.5-21 zeigt die Ergebnisse von *slice* & *dice*-Operationen. Für einen Java-Dozenten ist es z.B. interessant zu wissen, zu welcher Zeit an welchem Ort die meisten Java-Veranstaltungen stattfanden. Der Regionalvertrieb möchte dagegen wissen, welche Seminar-typen in welchem Jahr in Dortmund am erfolgreichsten waren.

Beispiel:
SemOrg



Ein Hyperwürfel hat den Vorteil, dass sich aus ihm eine Reihe weiterer Aggregationen² leicht berechnen lassen:

- 1 Eine Projektion auf die xy-Ebene (Veranstaltungsort x Seminartyp) liefert die Anzahl der Veranstaltungen jeweils über alle Quartale summiert (Abb. 9.5-22).

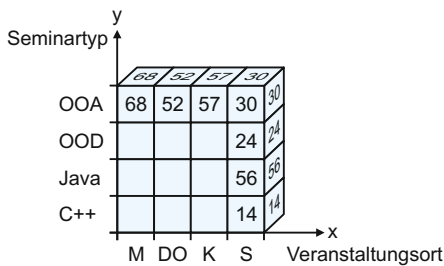
Abb. 9.5-21:
Selektion
unterschiedlicher
Datensichten
mittels der *slice* &
dice-Operation.

²Aggregation: hier: Verdichtung von Werten zu einem übergeordneten Wert, z.B. durch Summenbildung.

III 9 Statik

- 2 Eine Projektion auf die xz-Ebene (Veranstaltungsort x Zeit) liefert die Anzahl Veranstaltungen jeweils über alle Seminartypen summiert.
- 3 Eine Projektion auf die yz-Ebene (Seminartyp x Zeit) liefert die Anzahl Veranstaltungen jeweils über alle vier Orte summiert.
- 4 Eine Projektion auf die x-Achse (Veranstaltungsort) liefert die Anzahl Veranstaltungen summiert über Seminartypen und Quartale.
- 5 Eine Projektion auf die y-Achse (Seminartyp) liefert die Anzahl Veranstaltungen summiert über Orte und die Zeit.
- 6 Eine Projektion auf die z-Achse (Zeit) liefert die Anzahl Veranstaltungen summiert über die Seminartypen und die Orte.
- 7 Die Gesamtzahl aller Veranstaltungen über alle Quartale, Seminartypen und Orte ergibt sich durch Kollabieren des Würfels in die dem Ursprung am nächsten liegende Zelle, d. h. durch Summieren aller drei Dimensionen.

Abb. 9.5-22:
Würfel-Projektion
auf die xy-Ebene.



Jede Projektion bildet eine weitere Aggregation (hier: Summe) bzw. einen Unterwürfel. Eine solche Operation heißt *roll up* (zusammenrollen). Die umgekehrte Operation *drill down* navigiert von Summen zu detaillierteren Daten, sofern diese im *Data Warehouse* zur Verfügung stehen.

Beispiel: Der Kundensachbearbeiter möchte wissen, wie die Anzahl der Veranstaltungen sich auf die Bundesländer pro Jahr für die Themengruppe Programmierung verteilt. Um diese Frage zu beantworten, ist es notwendig, die Dimensionen um Dimensionsebenen, auch Elementhierarchien, Konsolidierungsebenen oder Aggregationsstufen genannt, zu ergänzen. Die Abb. 9.5-23 zeigt eine entsprechende Erweiterung in ER-Darstellung.

Sind Dimensionshierarchien modelliert, dann kann mit *roll up* zusätzlich in eine höhere Hierarchiestufe und mit *drill down* in eine niedrigere Hierarchiestufe navigiert werden. Mit der Operation *drill across* kann innerhalb einer Hierarchieebene von einem Datensatz zum anderen gewechselt werden.

Granularität Die Struktur einer Dimension lässt sich also nach »oben« und »unten« erweitern. Innerhalb einer Hierarchieebene sollte eine ähnliche Granularität herrschen. Die niedrigste Ebene einer Dimension ent-

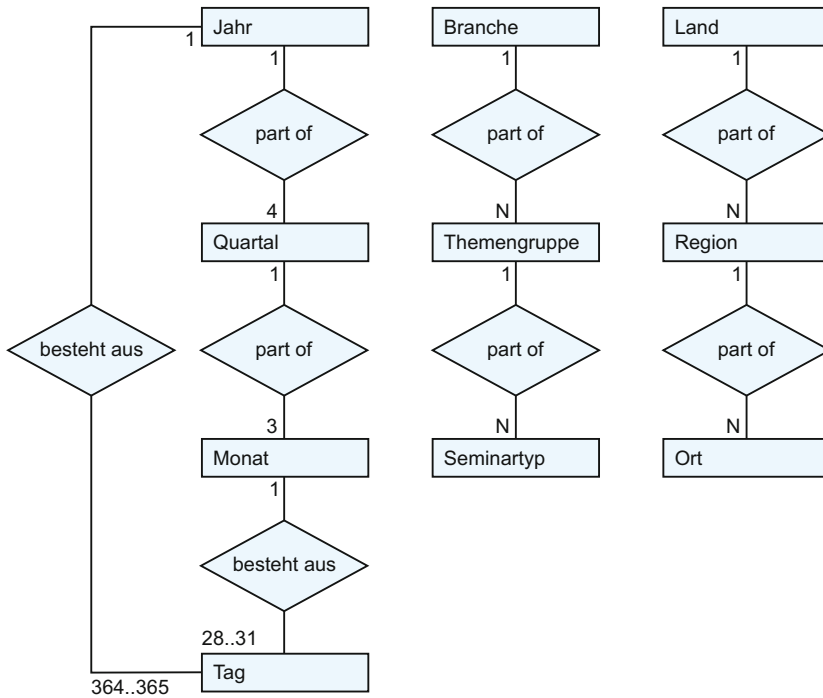


Abb. 9.5-23:
Dimensions-
hierarchien der
»Seminar-
organisation«.

spricht der feinsten verfügbaren Granularität von Informationen bezogen auf die Analyse von Kennzahlen in dieser Dimension. Ist Quartal z. B. die tiefste Zeitebene bei der Seminarorganisation, dann können keine Abfragen auf Monats- oder Tagesebene durchgeführt werden. Je feiner die Granularität ist, desto genauer können Ursachen analysiert werden. Auf der anderen Seite bedeutet dies aber auch, dass mehr Daten beschafft, transformiert und gespeichert werden müssen. Die OLAP-Würfel werden größer, und es müssen mehr Aggregationen vorgenommen werden.

Im Beispiel der Seminarorganisation wird nur eine Kennzahl gespeichert: die Anzahl der Veranstaltungen. Wünschenswert sind aber auch noch die Angaben zu Anzahl der Teilnehmer und zum Umsatz. Mehrere Kennzahlen können entweder in jeder Zelle des Würfels gespeichert oder in einer Kennzahlen-Dimension dargestellt werden.

Mehrere
Kennzahlen

Die Mindestanzahl der möglichen Fragestellungen, die mit einem OLAP-Würfel beantwortet werden können, ergibt sich zu:

Anzahl
Fragestellungen

$$N = n_1 \cdot n_2 \cdot \dots \cdot n_m,$$

wobei m die Anzahl der Dimensionen und n_1 die Anzahl Elemente in den Hierarchieebenen der Dimension 1 angibt usw. Die Anzahl der Elemente ergibt sich aus der Summe der Elemente über alle Hierarchieebenen hinweg plus ein Element für die Zusammenfassung (alle

III 9 Statik

Seminartypen usw.). Ein Element entspricht dabei einer Beschriftung einer Zeile oder Spalte in der OLAP-Analyse, z. B. OOA, OOD, Java, C++.

Richtlinien für
OLAP-Würfel

Bei der Gestaltung multidimensionaler Datenstrukturen sind folgende Richtlinien zu berücksichtigen [GaI98]:

- **Anzahl der Dimensionen:** 4 bis max. 10, möglichst Beschränkung auf maximal 6 bis 8, sonst leidet die Verständlichkeit.
- **Anzahl der Hierarchiestufen innerhalb einer Dimension:** max. 7 Stufen, sonst ist die Navigation von der höchsten Stufe bis auf die Basiselemente zeitraubend.
- **Anzahl der in einem Aggregationselement gebündelten Elemente:** max. 15 bis 20 Elemente, sonst geht der Überblick über die Struktur einer Dimension am Bildschirm rasch verloren.
- **Die Dimensionen sollen orthogonal zueinander stehen**, d. h. unabhängig voneinander sein:
 - Überprüfung durch paarweise Analyse der Multiplizitäten zwischen Basiselementen:
 - 1:1-Beziehung: keine Unabhängigkeit!
 - 1:N-Beziehung: in der Regel keine Unabhängigkeit!
 - M:N-Beziehung: in der Regel unabhängig.

9.5.3.3 Modellierungsansätze

Die Modellierung multidimensionaler Datenstrukturen weist eine Reihe von Besonderheiten auf, die im Modell berücksichtigt werden müssen ([GaI98], [Holt99, S. 149 ff.]):

- **Abbildung der Zeit:**

Die zeitliche Komponente ist von großer Bedeutung, da nahezu alle Elemente in ihrer Semantik direkt durch die Zeit charakterisiert sind. Die einzelnen Elemente müssen über einen längeren Zeitraum mit unterschiedlichen Verdichtungsgraden dargestellt werden können. Neben einer historischen Betrachtung sollen auch Zukunftsprognosen möglich sein.
- **Verdichtete Daten:**

Verdichtete Daten werden aus bereits vorhandenen Daten extrahiert.
- **Kennzahlverknüpfungen:**

Zwischen Kennzahlen müssen Verknüpfungen beschreibbar sein.

ER-Modellierung

Im Rahmen der ER-Modellierung lässt sich ein Hyperwürfel der Dimension n als Beziehungstyp ansehen, der eine n -äre Beziehung zu den Entitätstypen hat, die die Dimensionen repräsentieren [GaI98]. Die Kennzahlen werden dann als Attribute dem Beziehungstyp zugeordnet. Die Abb. 9.5-24 zeigt die ER-Modellierung des Hyperwürfels der »Seminarorganisation«, wobei die Dimensionshierarchien in Grau ergänzt sind.

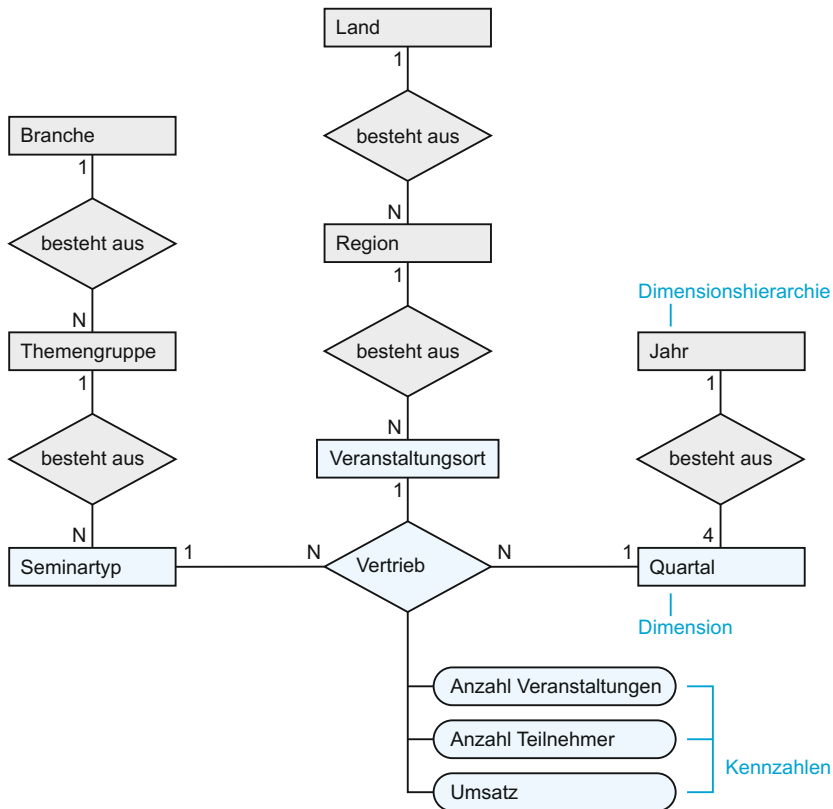


Abb. 9.5-24: ER-Modellierung eines Hyperwürfels mit Dimensionshierarchien (in Grau).

Eine genauere Untersuchung zeigt, dass die Konzepte des ER-Modells nicht ausreichen, um die Modellierungsgesichtspunkte für multidimensionale Datenstrukturen geeignet abzudecken. In [Holt99] werden verschiedene ER-Erweiterungen vorgestellt und bewertet.

Neuere Ansätze verwenden OO-Konzepte zur multidimensionalen Modellierung. Als grafische Notation wird vielfach die UML verwendet. Ein Vergleich bestehender Ansätze ist in [DoLe05] zu finden. Stellvertretend für diese Ansätze sei hier aufgrund ihres Bekanntheitsgrades die **Multidimensional UML** ($_m$ UML) genannt. Die in [HaHe99] und [Herd01] vorgestellte $_m$ UML ist eine multidimensionale Erweiterung der UML, in der durch **Stereotypen**, **Constraints**, **Kommentare** und **Tagged Values** die Standardnotation ergänzt wird (siehe »Basiskonzepte«, S. 99). Ausgangspunkt ist dazu die **Multidimensional Modeling Language** (MML), aus der die multidimensionalen Konstrukte übernommen sind.

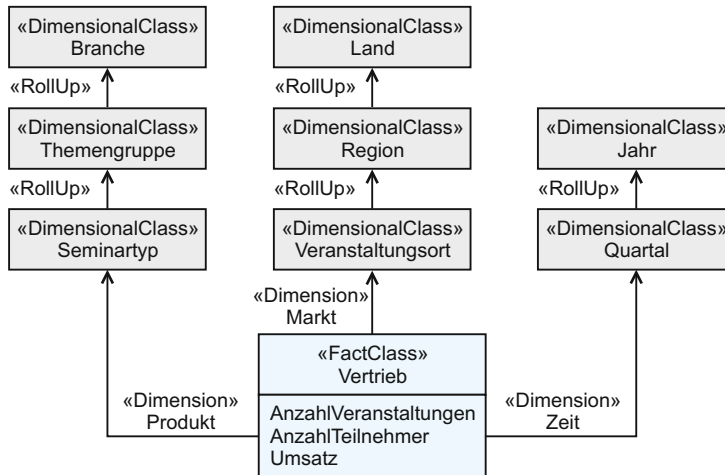
OO-Modellierung

Zur Darstellung der multidimensionalen Modelle wird das Klassendiagramm der UML verwendet. Die Kennzahlen werden durch Attribute in Faktenklassen dargestellt, die mit dem Stereotyp FactClass

III 9 Statik

gekennzeichnet sind. Dimensionsebenen werden durch Klassen mit dem Stereotyp `DimensionalClass` gebildet. Hierarchien in den Dimensionen werden durch gerichtete Assoziationen beschrieben (Stereotyp `RollUp`). Als Beispiel für die Verwendung der m UML wird in der Abb. 9.5-25 das multidimensionale Modell zur Seminarveranstaltung dargestellt.

Abb. 9.5-25:
Modellierung des
Hyperwürfels für
die Seminarorga-
nisation in m UML-
Notation.



ADAPT-Modellierung Dan Bulos hat für die multidimensionale Datenstrukturierung eine eigene grafische Modellierungsnotation entwickelt, die ihren Ursprung in der Unternehmensberatungspraxis hat [BuFo98] (siehe auch [GGP09, S. 90 ff.]).

Methodik Bei der Erstellung eines *Data Warehouse* kann man folgendermaßen vorgehen [Wiek99]:

1 Basismodell

Entsteht aus dem zugrunde gelegten operativen Datenmodell, z. B. einem unternehmensweiten Datenmodell. Gruppierung der vorhandenen Daten nach: relevant, irrelevant, potenziell relevant für das *Data Warehouse*. Festlegung der Granularität. Der Umstieg von Monats- auf Tagesebene kann das Volumen um den Faktor 30 erhöhen.

2 Historisierungsmodell

Einführung des Faktors Zeit und historischer Daten. Jede Entität des *Data Warehouse*-Modells erhält einen zeitlichen Zusatz in ihrem Primärschlüssel, soweit dieser noch nicht vorhanden ist. Die Historisierung weist jedem Datensatz einen Gültigkeitszeitraum zu, z. B. einen Zeitstempel oder eine Gültig-von-/Gültig-bis-Logik.

3 Dimensionsmodell

Festlegung des Aufbaus der Strukturinformationen, d. h. der Dimensionsdaten.

4 Aktualisierungsmodell

Anpassung an Aktualisierungszeitpunkte. Beispielsweise Aufteilung einer Tabelle in drei verschiedene Tabellen (Tabelle mit Daten, die sich nie ändern; Tabelle mit Daten, die sich selten ändern; Tabelle mit Daten, die sich häufig ändern).

5 Qualitätsmodell

Festlegung von Regeln für Konsistenz- und Plausibilitätsüberprüfungen.

6 Zugriffsmodell

Aufbau von Zugriffsstrukturen (Umgruppierungen, Verdichtungen).

ER-Modelle lassen sich auf Tabellen und damit auch auf Dateien abbilden (»Schlüssel, Tabellen und Dateien«, S. 204). Damit ist es auch möglich, das ER-Modell der Abb. 9.5-25 auf eine Tabellenstruktur abzubilden (Abb. 9.5-26).

Abbildung auf Tabellen

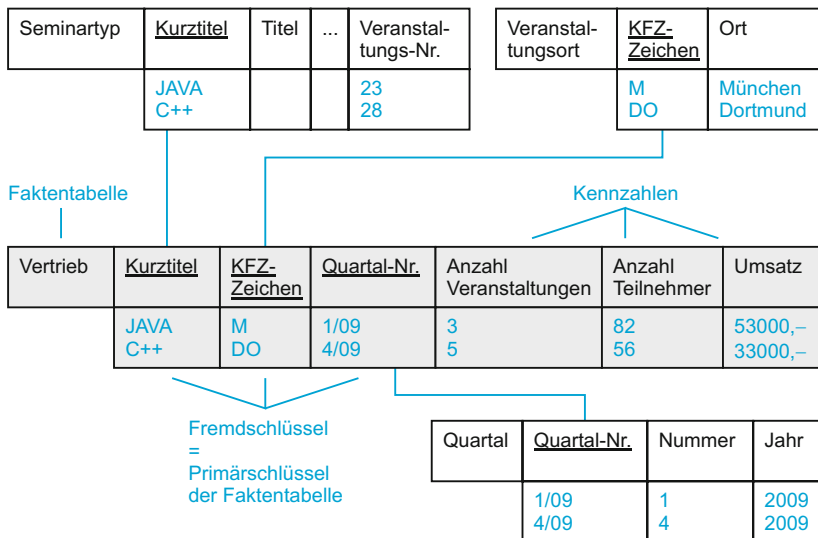


Abb. 9.5-26: Sternschema als Tabellenstruktur.

Als Tabellenstruktur ergibt sich ein so genanntes **Sternschema** (*starjoin*), das in der Mitte die sogenannte Faktentabelle mit den Kennzahlen enthält. Diese ist sternförmig mit den Dimensionstabellen verbunden. Dabei sind die Primärschlüssel der Dimensionstabellen als Fremdschlüssel in der Faktentabelle enthalten. Alle Fremdschlüssel in der Faktentabelle bilden gleichzeitig zusammen den Primärschlüssel der Faktentabelle. Die Dimensionshierarchien sind in der Abb. 9.5-26 nicht dargestellt.

Sternschema

III 9 Statik

9.5.3.4 Zusammenfassung

Während sich die ER-Konzepte gut dazu eignen, operativ genutzte Datenbestände zu modellieren, werden für Analysen auf Datenbeständen multidimensionale Datenstrukturen in Form von Hyperwürfeln modelliert. Mit OLAP-Operationen kann der Benutzer in den Hyperwürfeln navigieren. *Data Mining* weist den Benutzer auf verborgene Zusammenhänge hin. Die Datenbestände werden aus operativen Datenbeständen in *Data Warehouses* importiert. *Data Marts* stellen Ausschnitte aus *Data Warehouses* für spezielle Themenbereiche zur Verfügung.

Zusammenhang
mit ER-Modellen
und OO-Modellen

Aufgrund der besonderen Anforderungen an *Data Warehouses* eignet sich die ER-Modellierung nur bedingt für Hyperwürfel. OO-Ansätze befinden sich im Forschungsstadium.

Zur Klassifikation

- Die Modellierung von *Data Warehouses* erfolgt i.Allg. grafisch durch die Anwendung der ER-, der UML- oder einer verwandten Notation.
- Der Grad der Formalität ist in der Regel semiformal.
- Der Einsatz erfolgt in der Spezifikations- und Entwurfsphase.
- Der Anwendungsbereich ist auf *Data Warehouses* beschränkt.

10 Dynamik

Unter der Dynamik eines Softwaresystems werden sein Verhalten und seine Veränderungen während der Laufzeit des Systems verstanden. Während der Laufzeit kann eine dynamische Struktur entstehen oder sich verändern. Die Basiskonzepte zur Dynamik lassen sich in fachliche Aspekte und in zeitliche Aspekte gliedern.

Unter fachlichen Aspekten wird verstanden, dass das Basiskonzept den dynamischen Ablauf für das Softwaresystem festlegt. Dies ist mit folgenden Basiskonzepten möglich:

Kontrollstrukturen legen fest, unter welchen Bedingungen welche Teile eines Programms durchlaufen werden:

■ »Kontrollstrukturen«, S. 227

Geschäftsprozesse legen Arbeitsabläufe und deren Interaktion mit Akteuren fest, *Use Cases* erlauben es, die Benutzung eines Systems durch einen Akteur aus der Außensicht zu beschreiben:

■ »Geschäftsprozesse und *Use Cases*«, S. 250

Zustandsautomaten beschreiben das Verhalten eines Systems in Abhängigkeit von seiner Historie:

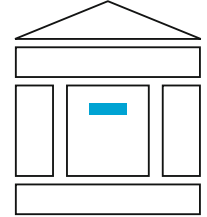
■ »Zustandsautomaten«, S. 269

Petrinetze ermöglichen es, nichtdeterministische Abläufe zu spezifizieren:

■ »Petrinetze«, S. 303

Unter zeitlichen Aspekten wird verstanden, dass bestimmte Zeitpunkte oder Zeitabläufe beschrieben, analysiert oder spezifiziert werden. Dies ist mit zeitlichen Schnappschüssen – in der Regel auf Exemplarebene – möglich:

■ »Szenarien«, S. 332



i
Fachaspekte

Zeitaspekte

10.1 Kontrollstrukturen

Kontrollstrukturen (*control structures*) steuern die Ausführung von Anweisungen, d. h. sie geben an, in welcher Reihenfolge, ob bzw. wie oft Anweisungen ausgeführt werden sollen. Es werden fünf semantisch unterschiedliche Kontrollstrukturen unterschieden (Abb. 10.1-1).

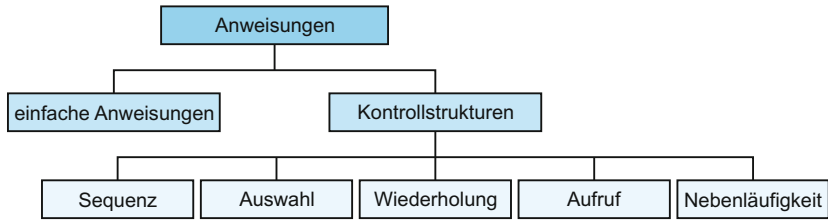
Kontrollstrukturen

Seit der Entwicklung der **strukturierten Programmierung** Anfang der 70er Jahre gilt es als »Stand der Technik«, nur die Kontrollstrukturen Sequenz, Auswahl, Wiederholung und Aufruf zu verwenden. Nebenläufigkeit spielte damals noch keine Rolle.

Strukturierte Programmierung

III 10 Dynamik

Abb. 10.1-1:
Gliederung von
Anweisungen.



Die Semantik dieser fünf Kontrollstrukturen wird in den folgenden Kapiteln skizziert:

- »Die Sequenz«, S. 229
- »Die Auswahl«, S. 229
- »Die Wiederholung«, S. 231
- »Der Aufruf«, S. 235
- »Die Nebenläufigkeit«, S. 235

3 Notationen Gleichzeitig werden sie in drei Notationen angegeben:

- Struktogramm-Notation,
- Java-Syntax und
- Aktivitätsdiagramm in der UML-Notation.



Ben Shneiderman

Aktivitäts-
diagramm



Die **Struktogramm-Notation** beruht auf einem Vorschlag von [NaSh73], daher auch **Nassi-Shneiderman-Diagramm** genannt, und ermöglicht eine grafische Darstellung von Kontrollstrukturen. Die Notation ist in [DIN66261] genormt.

Eine moderne grafische Darstellungsform von Programmen und Modellen ermöglicht die **UML**. Das **Aktivitätsdiagramm** der UML ermöglicht es, Kontrollstrukturen in Fluss-Notation zu beschreiben. Es benutzt Rechtecke mit abgerundeten Ecken zur grafischen Darstellung von Aktionen (*actions*) sowie kleine Rauten (*diamonds*) zur Darstellung von Verzweigungen (*decision nodes*) und Zusammenführungen (*merge nodes*). Die Symbole werden durch Pfeile miteinander verbunden, die den möglichen Kontrollfluss angeben. Alternativ gibt es noch eine kompakte Knoten-Notation.

PAP Die Aktivitätsdiagramme können als Nachfolger der **Programmablaufpläne** (PAPs) angesehen werden, bei denen ebenfalls grafische Symbole verwendet werden, die durch Linien miteinander verbunden sind. PAPs – auch **Flussdiagramme** genannt – sind bereits seit 1969 in Gebrauch und genormt [DIN66001]. Da die UML-Notation heute Industrie-Standard ist und durch entsprechende Werkzeuge unterstützt wird, verlieren PAPs immer mehr an Bedeutung. Sowohl Aktivitätsdiagramme in Fluss-Notation als auch PAPs sind aus Sicht der strukturierten Programmierung für die Konzeption von Programmen *nicht* gut geeignet. Die Aktivitätsdiagramme in Knoten-Notation unterstützen dagegen die strukturierte Programmierung. Einen Quervergleich grafischer Notationen für Kontrollstrukturen enthält [DIN EN28631].

Kontrollstrukturen können beliebig ineinander geschachtelt werden. Dadurch ist es möglich, komplexe Problemstellungen zu spezifizieren und zu programmieren. Schachtelung

Für die Ermittlung und Überprüfung von Aktivitäten gibt es konstruktive und analytische Schritte:

- »Box: Aktivität – Methode und Checkliste«, S. 245
- Zum Abschluss wird das Wichtigste zusammengefasst:
- »Zusammenfassung«, S. 249

10.1.1 Die Sequenz

Erfordert eine Problemlösung, dass mehrere Anweisungen hintereinander auszuführen sind, dann formuliert man eine **Sequenz** bzw. eine Aneinanderreihung von Anweisungen. Einen Überblick über die Darstellung der Sequenz in den verschiedenen Notationen gibt die Abb. 10.1-2.

Sequenz	allgemein	Erläuterung
Struktogramm	<div><div>Anweisung1</div><div>Anweisung2</div><div>Anweisung3</div></div>	Ein beliebig groß gewähltes Viereck wird nach jeder Anweisung mit einer horizontalen Linie abgeschlossen.
Java	Anweisung1; Anweisung2; Anweisung3;	Die einzelnen Anweisungen werden jeweils durch ein Semikolon (;) voneinander getrennt.
UML	<div><div>↓</div><div>Anweisung1</div><div>↓</div><div>Anweisung2</div><div>↓</div><div>Anweisung3</div><div>↓</div></div>	Einfache Anweisungen werden durch Rechtecke mit abgerundeten Ecken (Aktionen) dargestellt, die wiederum durch Pfeile verbunden werden, die den Kontrollfluss angeben.

Abb. 10.1-2:
Notationen für die
Darstellung einer
Sequenz.

10.1.2 Die Auswahl

Sollen Anweisungen nur in Abhängigkeit von bestimmten Bedingungen ausgeführt werden, dann verwendet man das Konzept der **Auswahl** – auch **Verzweigung** oder **Fallunterscheidung** genannt.

III 10 Dynamik

- 3 Konzepte Es gibt drei verschiedene Auswahl-Konzepte, die jeweils für bestimmte Problemlösungen geeignet sind:
- einseitige Auswahl (Abb. 10.1-3),
 - zweiseitige Auswahl (Abb. 10.1-3),
 - Mehrfachauswahl (Abb. 10.1-4).

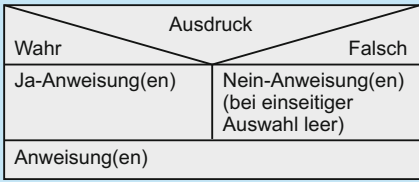
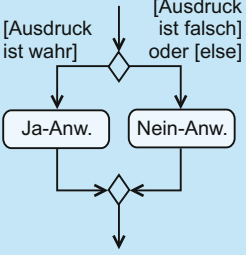
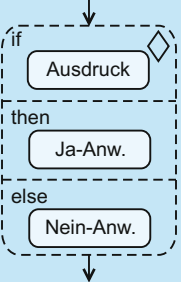
Auswahl (ein- und zweiseitig)	allgemein	Erläuterung
Strukto- gramm		Ist der Ausdruck wahr, dann werden die Ja-Anweisungen, sonst die Nein-Anweisungen ausgeführt.
Java	<pre> if (Ausdruck) Ja-Anweisung; else Nein-Anweisung; Anweisung(en); </pre>	Semantik analog zum Struktogramm. Bei der einseitigen Auswahl fehlt » else Anweisung«. Das Ergebnis des Ausdrucks muss vom Typ boolean sein. Anstelle von Anweisung ; kann auch ein Block stehen: {Anweisungen ; }
UML	<div style="display: flex; justify-content: space-around;"> <div> <p>Fluss-Notation</p>  </div> <div> <p>Knoten-Notation</p>  </div> </div>	Semantik analog zu Java. Bei der einseitigen Auswahl entfällt das abgerundete Rechteck mit Nein-Anweisungen. In eckigen Klammern steht an den Pfeilen ein Wächter (<i>guard</i>). Der Ausdruck muss wahr sein, damit der Kontrollfluss entlang des jeweiligen Pfeiles verlaufen kann.

Abb. 10.1-3: Darstellungsformen für die ein- und zweiseitige Auswahl.

In der UML gibt es zwei Möglichkeiten, eine ein- und zweiseitige Auswahl darzustellen. Die linke Darstellung (siehe Abb. 10.1-3) zeigt durch die Pfeile detailliert den Kontrollfluss (Fluss-Notation). In der kompakten rechten Darstellung wird die Auswahl durch einen sogenannten **Entscheidungsknoten** (*conditional node*) repräsentiert (Knoten-Notation). Der Entscheidungsknoten ist ein Sonderfall des strukturierten Knotens der UML (*structured activity node*). Der Entscheidungsknoten wird durch ein gestricheltes Rechteck mit abgerundeten Ecken dargestellt, das aus mehreren Bereichen besteht. Der **if**-Bereich enthält den Ausdruck, der über den weiteren Kontrollfluss entscheidet. Ist der Ausdruck wahr, dann werden die Anwei-

sungen im `then`-Bereich ausgeführt, sonst die Anweisungen im `else`-Bereich. Der `else`-Bereich kann auch fehlen. In diesem Fall liegt eine einfache Auswahl vor.

Durch die Bedingung – genauer gesagt durch das Ergebnis der Auswertung eines Ausdrucks (*expression*) – wird eine Auswahl der auszuführenden Anweisungen vorgenommen. Ist die Bedingung erfüllt bzw. wahr (`true`), dann werden die Ja-Anweisungen ausgeführt, sonst die Nein-Anweisungen (`false`).

Zweiseitige
Auswahl

Bei der einseitigen Auswahl handelt es sich um einen Sonderfall der zweiseitigen Auswahl. Im `else`-Zweig steht *keine* Anweisung. Bei Programmiersprachen fehlt bei der einseitigen Auswahl der `else`-Zweig, ebenso in der Knoten-Notation der UML.

Einseitige Auswahl

Muss zwischen mehr als zwei Möglichkeiten gewählt werden, dann wird die Mehrfachauswahl verwendet (Abb. 10.1-4).

Mehrfach-
auswahl

Die UML-Notationen kennen keine spezielle Darstellung für die Mehrfachauswahl. In der Knoten-Notation werden in den Entscheidungsknoten mehrere `if`-Bereiche eingetragen, für jeden Fall ein `if`-Bereich.

Innerhalb einer Auswahl kann wieder eine Auswahl stehen usw. Es liegen dann geschachtelte Auswahlanweisungen vor.

Schachtelung

10.1.3 Die Wiederholung

Sollen eine oder mehrere Anweisungen in Abhängigkeit von einer Bedingung wiederholt oder für eine gegebene Anzahl von Wiederholungen durchlaufen werden, so ist das Konzept der **Wiederholung** bzw. Schleife zu verwenden. Drei Wiederholungskonstrukte werden unterschieden:

- Wiederholung mit Abfrage vor jedem Wiederholungsdurchlauf (abweisende Wiederholung),
- Wiederholung mit Abfrage nach jedem Wiederholungsdurchlauf (akzeptierende Wiederholung),
- Wiederholung mit fester Wiederholungsanzahl (Zählschleife).

Bei den ersten beiden Wiederholungskonstrukten handelt es sich um bedingte Wiederholungen.

Zusätzlich gibt es noch den Sonderfall, dass eine Wiederholung abgebrochen werden muss, z. B., wenn ein Fehler aufgetreten ist. Diesen Sonderfall bezeichnet man als $n + 1/2$ -Schleife.

Wie die Abb. 10.1-5 für die bedingte Wiederholung zeigt, gibt es bei der UML sowohl eine Fluss-Notation als auch eine Knoten-Notation zur Darstellung der bedingten Wiederholung. Bei der Knoten-Notation wird eine Wiederholung durch einen sogenannten **Schleifenknoten** (*loop node*) dargestellt. Er wird durch ein Rechteck mit abgerundeten Ecken und gestrichelten Linien gezeichnet, das in zwei Bereiche aufgeteilt ist. Der `while`-Bereich enthält die Bedin-

III 10 Dynamik

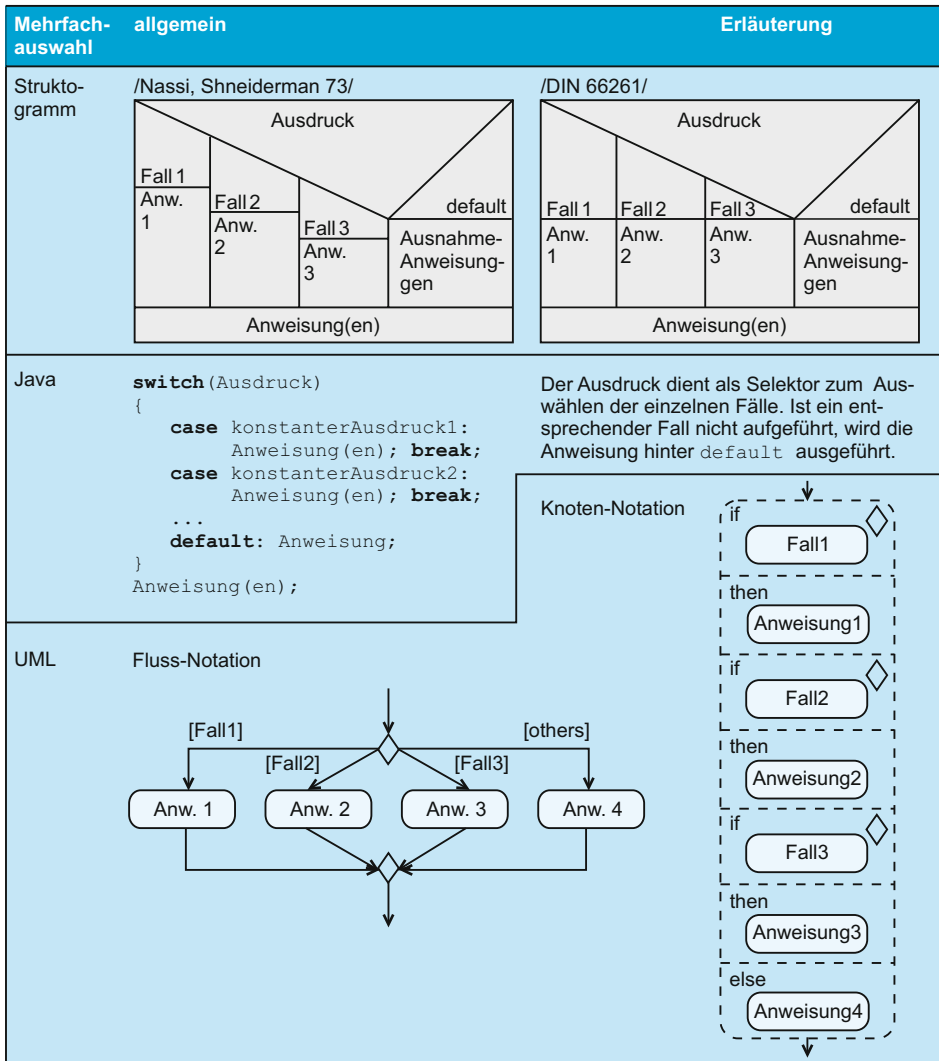


Abb. 10.1-4: Darstellungsformen für die Mehrfach-Auswahl.

gung. Ist diese Bedingung wahr, dann wird der Schleifenrumpf erneut durchlaufen. Andernfalls wird die nächste Anweisung nach der Wiederholung ausgeführt.

Bei der Wiederholung mit Abfrage *vor* jedem Wiederholungsdurchlauf wird solange wiederholt, wie die Bedingung erfüllt ist. Dann wird hinter der zu wiederholenden Anweisung bzw. Anweisungsfolge fortgefahren. Ist die Bedingung bereits am Anfang *nicht* erfüllt, dann wird die Wiederholungsanweisung *keinmal* ausgeführt. Die Bedingung muss daher am Anfang der Wiederholung bereits einen eindeutigen Wert besitzen.

Wiederholung	allgemein	Erläuterung
Struktogramm	<div> <div>Ausdruck</div> <div>Wiederholungsanweisung(en)</div> <div>Anweisung(en)</div> </div>	Wiederholung mit Abfrage vor jedem Wiederholungsdurchlauf (Abweisende Schleife)
	<div> <div>Wiederholungsanweisung(en)</div> <div>Ausdruck</div> <div>Anweisung(en)</div> </div>	Wiederholung mit Abfrage nach jedem Wiederholungsdurchlauf (Akzeptierende Schleife)
Java	<pre>while (Ausdruck) { Wiederholungsanweisungen; } Anweisung(en);</pre>	Wiederholung mit Abfrage vor jedem Wiederholungsdurchlauf
	<pre>do { Wiederholungsanweisungen; } while (Ausdruck); Anweisung(en);</pre>	Wiederholung mit Abfrage nach jedem Wiederholungsdurchlauf
<div> <div>UML</div> <div> <div> <div>Abweisende Schleife</div> <div> <div>Fluss-Notation</div> </div> <div> <div>Knoten-Notation</div> </div> </div> <div> <div>Akzeptierende Schleife</div> <div> <div>Fluss-Notation</div> </div> <div> <div>Knoten-Notation</div> </div> </div> </div> </div>		

Bei der Wiederholung mit Abfrage *nach* jedem Wiederholungsdurchlauf wird solange wiederholt, wie die Bedingung erfüllt bzw. der Ausdruck wahr ist. Die zu wiederholenden Anweisungen werden also in jedem Fall einmal ausgeführt, da die Bedingung erst am Ende abgefragt wird. Eine Wiederholung mit Abfrage nach jedem Durchlauf lässt sich auf eine Wiederholung mit Abfrage vor jedem Durchlauf zurückführen, wenn die Schleife so initialisiert wird, dass die Bedingung am Anfang erfüllt ist.

Es gibt Fälle, in denen es notwendig ist, innerhalb der Wiederholungsanweisungen die laufende Wiederholung abzubrechen. Dies ist insbesondere dann sinnvoll, wenn z. B. bei einer Berechnung innerhalb einer Wiederholung Fehler auftreten, die eine weitere Verarbeitung der Wiederholungsanweisungen überflüssig machen. Die

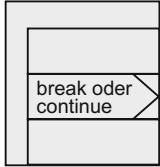
Abb. 10.1-5: Notationen für bedingte Wiederholungen.

$n + 1/2$ -Schleife

III 10 Dynamik

Abbildung in der Marginalspalte zeigt eine mögliche Darstellung in einem Struktogramm. Diese Darstellung wird aber *nicht* einheitlich verwendet.

Struktogramm-Darstellung (nicht einheitlich verwendet)



Konzeptionell hat man die Kontrollstruktur Wiederholung so verallgemeinert, dass

- innerhalb des Wiederholungsteils ein oder mehrere Unterbrechungen (*breaks*) oder Aussprünge programmiert werden können, die bewirken, dass aus dem Wiederholungsteil hinter das Ende der Wiederholung verzweigt bzw. gesprungen wird,
- die aktuelle Wiederholung abgebrochen wird und sofort eine neue Wiederholung beginnt (*continue*), d. h. es wird zur jeweiligen Wiederholungsbedingung verzweigt, und
- die Methode, in der sich die Wiederholung befindet, durch ein *return* beendet wird.

Zählschleife Sollen eine oder mehrere Anweisungen (Blöcke) für eine gegebene Zahl von Wiederholungen durchlaufen werden, so ist das Konzept der **Zählschleife** – auch Laufanweisung genannt – zu verwenden (Abb. 10.1-6).

Wiederholung	allgemein	Erläuterung
Struktogramm	<pre> for (Startausdruck; Testausdruck; Schrittweite) Wiederholungsanweisung(en) Anweisung(en) </pre>	Wiederholung mit fester Wiederholungszahl (Zählschleife, Laufanweisung)
Java	<pre> for (Startausdruck; Testausdruck; Schrittweite) { Wiederholungsanweisungen; } Anweisung(en); </pre>	Wiederholung mit fester Wiederholungszahl (Zählschleife, Laufanweisung)
UML	<div style="display: flex; justify-content: space-around;"> <div> <p>Fluss-Notation</p> </div> <div> <p>Knoten-Notation</p> </div> </div>	

Abb. 10.1-6:
Notationen für
Zählschleifen.

Die Anzahl der Wiederholungen wird durch eine Zählvariable mitgezählt und die Bedingung so gewählt, dass nach der geforderten Wiederholungszahl der Abbruch erfolgt. Einige Probleme erfordern auch ein Abwärtszählen. Auch dies kann mit der Zählschleife programmiert werden.

10.1.4 Der Aufruf

In vielen Systemen werden Teilaufgaben oft mehrmals benötigt. Solche Teilaufgaben werden daher als eigenständige Operationen bzw. Methoden formuliert und von der Operation/Methode aus, in der die Teilaufgabe benötigt wird, aufgerufen. **Aufruf** bedeutet, dass an der Stelle, an der die Teilaufgabe auszuführen ist, eine Verzweigung zu der Teilaufgabe, d. h. zu der entsprechenden Operation/Methode, erfolgt. Nach Abschluss der aufgerufenen Operation/Methode wird das Programm hinter der Aufruf- bzw. Verzweigungsstelle fortgesetzt. Der Operation/Methode, die aufgerufen wird, können noch aktuelle Parameter übergeben werden. Ebenso können Ergebnisse an die aufrufende Operation/Methode übergeben werden. Die verschiedenen Notationen zeigt die Abb. 10.1-7. Da durch einen Aufruf die sequenzielle Ausführung der Anweisungen in der rufenden Operation/Methode unterbrochen wird, zählt der Aufruf zu den Kontrollstrukturen.

In der UML spricht man von Operationen, in der objekt-orientierten Programmierung von Methoden

10.1.5 Die Nebenläufigkeit

In manchen Fällen spielt die Reihenfolge, in der Aktionen oder Anweisungen ausgeführt werden sollen, aus fachlicher Sicht keine Rolle. Aktionen oder Anweisungen können sowohl in beliebiger Sequenz oder auch zeitlich parallel ausgeführt werden. Man spricht dann von **Nebenläufigkeit** (*concurrency*). Sowohl die UML als auch einige Programmiersprachen erlauben die Modellierung von Nebenläufigkeit.

In der Abb. 10.1-8 ist die Reihenfolge der Aktionen »Rechnung drucken« und »Kreditkarte belasten« gleichgültig.

Beispiel

Die UML bietet dafür die Notation des **Splitting** (*fork node*) und der **Synchronisation** (*join node*) an. Bei einem Splitting wird der Kontrollfluss in mehrere parallele Ströme aufgeteilt und bei der Synchronisation wieder zusammengeführt. Eine nachfolgende Aktion kann erst ausgeführt werden, wenn alle parallelen Pfade durchlaufen wurden. Splitting und Synchronisation werden durch einen Balken modelliert. Der Splitting-Balken besitzt einen Eingangs- und mehrere Ausgangspfeile, bei einem Synchronisationsbalken werden mehrere Eingangs- und ein Ausgangspfeil angetragen. Folgt auf eine Synchro-



III 10 Dynamik

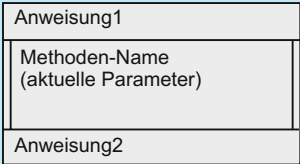
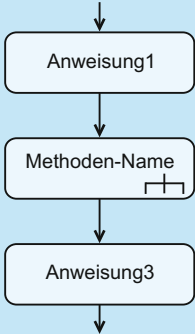
Aufruf	allgemein	Erläuterung
Struktogramm	 <pre> graph TD A[Anweisung1] --> B[Methoden-Name (aktuelle Parameter)] B --> C[Anweisung2] </pre>	Nach Ausführung der aufgerufenen Methode wird die rufende Methode hinter der Aufrufstelle fortgesetzt (Anweisung2).
Java	<pre> Anweisung1; Methoden-Name (aktuelleParameter); Anweisung2; </pre>	Ein Aufruf erfolgt durch Angabe des Methoden-Namens, gefolgt von der Liste der aktuellen Parameter.
UML	 <pre> graph TD Start(()) --> A[Anweisung1] A --> B[Methoden-Name] B --> C[Anweisung3] C --> End(()) </pre>	Durch ein Rechensymbol rechts unten in einem abgerundeten Rechteck wird ein Aktivitätsaufruf angegeben (sub activity indicator).

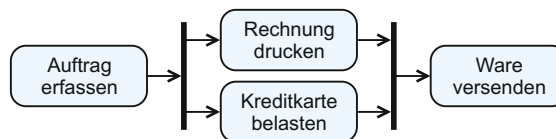
Abb. 10.1-7: Darstellungsformen des Aufrufs.

nisation direkt ein Splitting, dann können beide auch in einem Balken mit mehreren Eingangs- und Ausgangspfeilen kombiniert werden.

Java

In Java können durch sogenannte **Threads** unabhängige Kontrollflüsse initiiert werden. Durch Synchronisationskonzepte kann die geordnete Zusammenarbeit zwischen verschiedenen **Threads** programmiert werden.

Abb. 10.1-8: Aufsplitten des Kontrollflusses in parallele Pfade.



Literatur

[Zies04] mit E-Learning-Kurs.

10.1.6 Aktivitätsdiagramm

Eine **Aktivität** (activity) beschreibt in der UML die Ausführung von Funktionalität bzw. Verhalten. Sie wird durch mehrere Knoten modelliert, die durch gerichtete Kanten miteinander verbunden sind. Es lassen sich Aktionsknoten, Kontrollknoten und Objektknoten unterscheiden. Aktivitäten besitzen sowohl ein Kontrollfluss- als auch

ein Datenmodell. Das Kontrollflussmodell spezifiziert die Reihenfolge von Aktionen und das Datenmodell die Daten, die zwischen den Aktionen ausgetauscht werden.

Aktionsknoten

Die kleinste ausführbare Einheit innerhalb einer Aktivität wird als **Aktion** (*action*) bezeichnet. Eine Aktion kann ausgeführt werden, wenn die Vorgänger-Aktion beendet ist, wenn notwendige Daten zur Verfügung stehen (z. B. Bestellung liegt vor) oder wenn ein Ereignis auftritt (z. B. Jahreswechsel). Eine Aktion kann nicht nur ein elementarer Verarbeitungsschritt, sondern auch ein Aktivitätsaufruf sein, d. h. von der Ausführung her gesehen, kann sich hinter einem Aktionsknoten eine sehr komplexe Verarbeitung verbergen.

Aktionen werden durch Rechtecke mit abgerundeten Ecken dargestellt. In das Rechteck wird der Name oder eine kurze Beschreibung der Aktion eingetragen. Steht eine Aktion für einen Aktivitätsaufruf, dann wird rechts unten ein kleines gabelähnliches Symbol (bzw. Harken- oder Rechensymbol) eingetragen (Abb. 10.1-9).

Notation

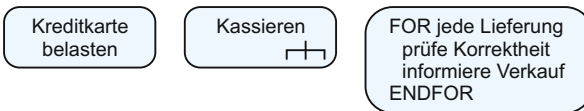


Abb. 10.1-9:
Notation für
Aktionen.

Eine Aktivität wird durch ein großes Rechteck mit abgerundeten Ecken modelliert, wobei links oben der Name der Aktivität eingetragen wird. Die Verarbeitungsschritte der Aktivität werden durch einen Graphen dargestellt, der aus Knoten (*nodes*) und Pfeilen (*edges*) besteht. Die Knoten entsprechen im einfachsten Fall den Aktionen. Die Pfeile (gerichtete Kanten) verbinden die Knoten und stellen im einfachsten Fall den Kontrollfluss der Aktivität dar. Viele Aktivitäten benötigen Eingaben und produzieren Ausgaben. Sie werden durch Parameterknoten beschrieben. Der gleiche Aktionsname kann in einer Aktivität mehrfach vorkommen, beispielsweise wenn die gleiche Aktion mehrere Male ausgeführt wird. Die Aktivität ist als Namensraum für die in ihr enthaltenen Aktionen zu sehen. Die Abb. 10.1-10 zeigt die Notation für eine einfache Aktivität. Sie wird in einem **Aktivitätsdiagramm** (*activity diagram*) modelliert.

Aktivität

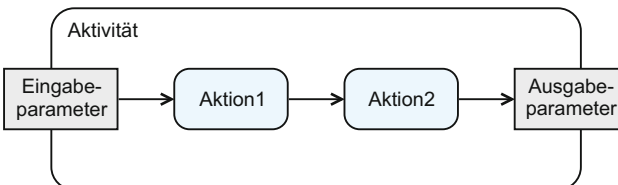
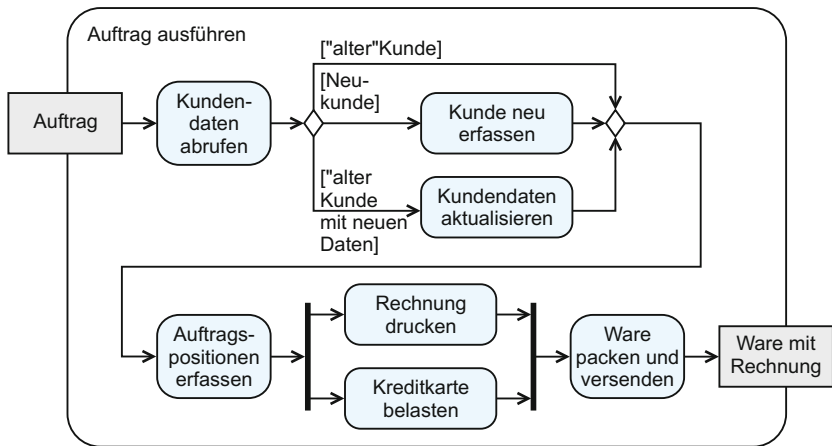


Abb. 10.1-10:
Einfache Aktivität
mit Ein-/Ausgabe-
parametern.

III 10 Dynamik

Beispiel 1a Als Beispiel für eine Aktivität wird die Bearbeitung eines Auftrags betrachtet (Abb. 10.1-11). Eingabeparameter ist der vom Kunden erteilte Auftrag. Im ersten Schritt wird geprüft, ob der Kunde schon im System vorhanden ist (Kundendaten abrufen). Ist es ein Neu-Kunde, dann wird die Aktion Kunde neu erfassen ausgeführt. Haben sich bei einem »alten« Kunden die Daten geändert, dann werden die Kundendaten aktualisiert. Andernfalls ist nichts zu tun. Wenn die Auftragspositionen erfasst sind, können die Aktionen Rechnung drucken und Kreditkarte belasten in beliebiger Reihenfolge ausgeführt werden. Sind beide abgeschlossen, kann die Ware verpackt und versandt werden. Ausgabeparameter dieser Aktivität ist die Ware mit der Rechnung. In diesem Fall beginnt die Aktivität, wenn Eingabedaten (Auftrag) vorliegen. Analog endet sie, wenn der Ausgabeparameter erzeugt wurde.

Abb. 10.1-11:
Aktivität zur
Ausführung eines
Auftrags.



Objektknoten

Außer den Aktions- und Kontrollknoten gibt es das Konzept des Objektknotens. **Objektknoten** werden durch Rechtecke dargestellt und häufig mit dem Namen der Klasse benannt (siehe »Zusammenfassung von Funktionen«, S. 131). Der Objektknoten kann während der Ausführung der Aktivität nur Werte annehmen, die konform mit der Klassenspezifikation sind.

Objektfluss Mit Hilfe von Objektknoten können Daten von einer Aktion zur nächsten gereicht werden. Sie realisieren somit das Datenmodell einer Aktivität. Anstelle vom Kontrollfluss (*control flow*) spricht man hier vom Objektfluss (*object flow*). Während Kontrollflüsse die Ausführungsreihenfolge von Aktionen bestimmen, modellieren Objektflüsse Objekte und Daten, die von einem Aktionsknoten erzeugt und

von einem anderen benötigt werden. Objektflüsse werden als Pfeile dargestellt, die an mindestens einem Ende einen Objektknoten besitzen.

In der Abb. 10.1-12 »fließen« Auftragsobjekte von der Aktion Auftrag erteilen zur Aktion Auftrag bearbeiten. Beispiel

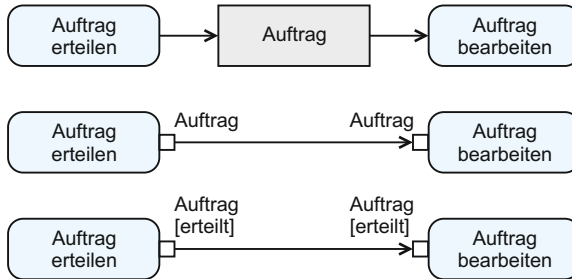


Abb. 10.1-12:
Notationen für den
Objektfluss.

Objektknoten können alternativ zur Rechteck-Notation als »Pins« modelliert werden. Das sind kleine Quadrate, die direkt auf der linken und rechten Seite einer Aktion angetragen werden. Sie stellen der Aktion Eingabewerte zur Verfügung und nehmen Ausgabewerte von ihr entgegen. Neben die Pins wird der Name des Objektknotens (z. B. Klassenname) angetragen. Zusätzlich kann der Zustand, in dem sich das Objekt befindet, unter dem Namen in eckigen Klammern angegeben werden. Sollten zwischen zwei Aktionen mehrere Objektflüsse stattfinden, so kann man auf beiden Seiten der Aktionen eine ganze Reihe von Pins eintragen.

Pin-Notation

Eine besondere Art von Objektknoten sind die Parameterknoten, die auf den Grenzen einer Aktivität angetragen werden. Eingangsparameter besitzen nur ausgehende, Ausgangsparameter nur eingehende Pfeile. Besitzt eine Aktion einen Eingangsparameter, dann muss sie keinen Startknoten besitzen, sondern sie wird durch die im Objektknoten verfügbaren Daten initiiert.

Parameter der
Aktivität

Außer den Daten, die in einem System von einer Aktion zur nächsten durchgereicht werden, gibt es »ruhende« Daten. Man spricht von einem Datenspeicher, der eine Sonderform des Objektknotens ist. Der Datenspeicher enthält alle Daten, die in ihn eingetragen werden, und kopiert sie bei Bedarf in ausgehende Objektflüsse. Eingehende Daten können bereits vorhandene Daten im Datenspeicher ersetzen. Die Abb. 10.1-13 modelliert eine Artikel-Datenbank als Datenspeicher. Er wird von der Aktion Artikel erfassen/ändern gefüllt und von der Aktion Artikelkatalog erstellen gelesen. Im Unterschied zum »normalen« Objektknoten werden hier persistente, d. h. dauerhaft gespeicherte Daten modelliert.

Datenspeicher

III 10 Dynamik

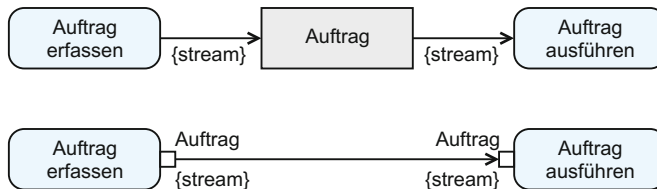
Abb. 10.1-13:
Datenspeicher zur
Modellierung
persistenter Daten.



Streaming

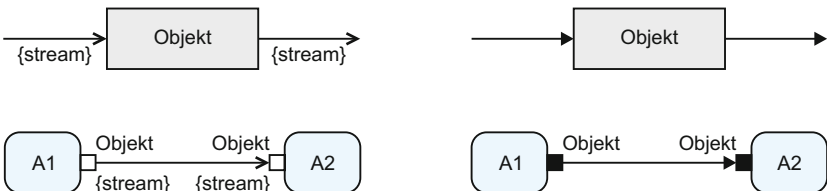
Objektknoten können in einem Streaming-Modus verwendet werden. In der Abb. 10.1-14 bedeutet dies, dass die Aktion Auftrag erfassen kontinuierlich durchgeführt wird und neue Aufträge produziert. Die Aktion Auftrag ausführen ist ebenfalls eine kontinuierlich durchgeführte Aktion, die die erzeugten Aufträge verbraucht. Eine Aktion, die mit einem kontinuierlichen Datenstrom verbunden ist, beginnt, wenn das erste Datenelement vorhanden ist, und verarbeitet alle weiteren Objekte, sobald sie vorliegen. Die Abb. 10.1-14 zeigt, wie *streams* für Objektknoten und Pins spezifiziert werden.

Abb. 10.1-14:
Modellierung von
streams.



Um kontinuierliche Datenströme beim Objektfluss darzustellen, bietet die UML auch eine spezielle Notation mit gefüllten Pfeilspitzen bzw. mit gefüllten Eingabe- und Ausgabe-Pins an. In der Abb. 10.1-15 sind die linke und rechte Darstellung jeweils äquivalent.

Abb. 10.1-15:
Alternative
Modellierung von
Streams.



Ereignisse

Sowohl im betrachteten System selbst als auch außerhalb können Ereignisse auftreten. Sie werden – unabhängig von irgendwelchen Aktionen – von Objekten entdeckt und in den Objekten gespeichert.

Ereignis-
empfänger

Aktionen, die auf das Eintreten eines Ereignisses warten, werden Ereignisempfänger (*accept event actions*) genannt. Sie werden durch ein konkaves Fünfeck dargestellt. Eine solche Aktion ist beispielsweise Zahlungseingang in der Abb. 10.1-16. Sie wartet darauf, dass das Ereignis Zahlungseingang eintritt. Anschließend wird die nächste Aktion Ware versenden aktiviert. Besitzt ein Ereignisempfänger keine

eingehenden Kanten, dann startet diese Aktion, wenn die umgebende Aktivität beginnt. Sie ist während der gesamten Ausführungszeit der Aktivität in der Lage, Ereignisse entgegenzunehmen – gleichgültig, wie viele auftreten. Sie endet also nicht, wenn das erste Ereignis aufgetreten ist.

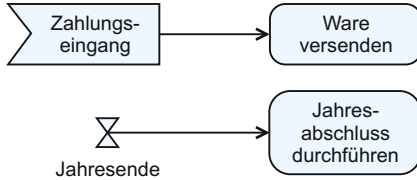


Abb. 10.1-16:
Ereignisempfänger
und Empfänger
von
Zeitereignissen.

Eine Sonderform sind Zeitereignisse. Der Empfänger von Zeitereignissen (*accept time event actions*, *wait time actions*) wird durch eine Sanduhr symbolisiert. In der Abb. 10.1-16 löst das zeitliche Ereignis Jahresende die Aktion Jahresabschluss durchführen aus.

Zeitereignisse

Das Gegenstück zum Warten auf ein Ereignis ist eine Aktion, die ein Signal an ein Zielobjekt sendet, das dort eine weitere Verarbeitung auslösen kann. Der Signalsender (*signal send action*) wird durch ein konvexes Fünfeck dargestellt. In der Abb. 10.1-17 wird nach dem Erfassen eines Auftrags eine Versandanweisung an das Auslieferungslager geschickt. Diese Aktion aktiviert dann die Aktion Ware versenden.

Signalsender

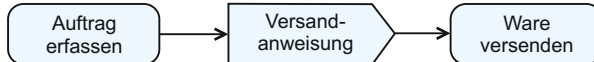


Abb. 10.1-17:
Signalsender.

Das Senden eines Signals und das Warten auf ein Ereignis können auch kombiniert auftreten. In der Abb. 10.1-18 wird der Buchhaltung ein Signal geschickt, um einen Zahlungseingang abzufragen. Wenn das Ereignis Zahlungseingang auftritt, wird die Ware versendet.

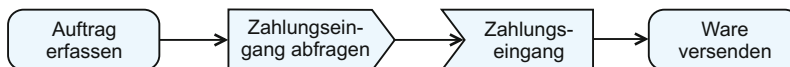


Abb. 10.1-18:
Kombination
Signalsender und
Signalempfänger.

Start- und Endeknoten

Die Aktion, die innerhalb einer Aktivität zuerst ausgeführt wird, wird mit einem Startknoten (*initial node*) markiert, der als kleiner schwarzer Kreis dargestellt wird. Der Startknoten wird auch als Anfangsknoten bezeichnet. Er besitzt keinen eingehenden Pfeil, kann jedoch mehrere ausgehende Kanten besitzen. Eine Aktivität kann mehrere Startknoten besitzen. In diesem Fall startet die Aktivität parallel in

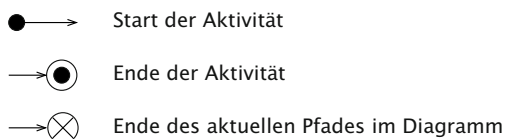
Startknoten

III 10 Dynamik

mehreren Pfaden. Ein Startknoten muss nicht unbedingt vorhanden sein. In diesem Fall startet die Aktivität, wenn notwendige Daten verfügbar sind (siehe oben Parameter).

Endknoten Das Gegenstück zum Startknoten sind die Endknoten, die einerseits das Beenden einer Aktivität und andererseits das Beenden des Kontrollflusses anzeigen. Endknoten besitzen nur eingehende Pfeile. Das Bullauge bzw. ein kleiner schwarzer Kreis mit einem umschließenden Ring kennzeichnet das Ende einer Aktivität. Wird dieser Endknoten (*final node*) erreicht, dann werden sämtliche Aktionen innerhalb der Aktivität sofort beendet. Eine Aktivität kann mehr als einen Endknoten besitzen. Sie wird in diesem Fall dann beendet, wenn der erste Knoten erreicht ist. Außerdem gibt es einen Endknoten für Kontrollflüsse (*flow final*) an. Er wird auch als Ablaufende bezeichnet. Dieser Knoten beendet den Pfad im Aktivitätsdiagramm und hat keinerlei Auswirkungen auf andere Pfade. Das bedeutet, dass Aktionen in anderen Pfaden *weiter ausgeführt* werden können. Das Ablaufende wird durch ein X mit einem umschließenden Kreis modelliert. Die Abb. 10.1-19 zeigt die Notation für Start- und Endknoten.

Abb. 10.1-19:
Notation für Start-
und Endknoten.



Aktivitätsbereiche

Aktivitätsbereiche (*activity partitions*) fassen Aktionen zusammen, die bestimmte Gemeinsamkeiten besitzen, z. B. organisatorische Einheit, Standort, Verantwortungsbereich. Sie werden auch als Verantwortlichkeitsbereiche oder Partitionen bezeichnet. Aktivitätsbereiche werden durch parallele Linien spezifiziert. Jeder Bereich wird mit einer Box an einem Ende gekennzeichnet. Da diese Darstellung an die einzelnen Bahnen in einem Schwimmbecken erinnert, wird auch von Schwimmbahnen (*swimlanes*) gesprochen. In der Abb. 10.1-20 werden die ersten drei Aktionen innerhalb des Aktivitätsbereichs Verkauf ausgeführt und die Aktion Ware versenden innerhalb der Partition Lager.

Hierarchie von
Aktivitäts-
bereichen

Schwimmbahnen können auch hierarchisch angeordnet sein. In der Abb. 10.1-21 gehören die Aktivitätsbereiche Verkauf, Lager und Buchhaltung zum übergeordneten Aktivitätsbereich Shop.

Externe Aktivitäts-
bereiche

In einer Aktivität können nicht nur die Aktionen in dem betrachteten System, sondern auch extern durchgeführte Verarbeitungsschritte spezifiziert werden, um das Zusammenwirken beider zu modellieren. Mit dem Schlüsselwort «external» kann spezifiziert werden,

10.1 Kontrollstrukturen III

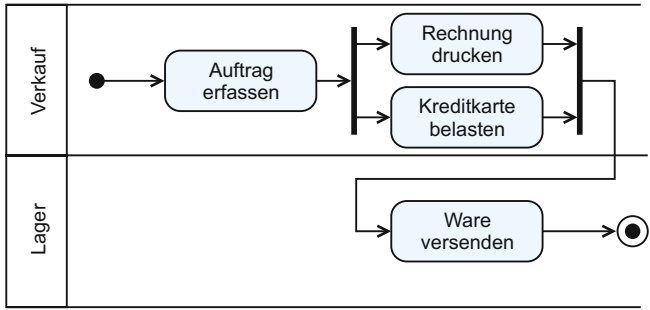


Abb. 10.1-20:
Einfache
Aktivitätsbereiche.

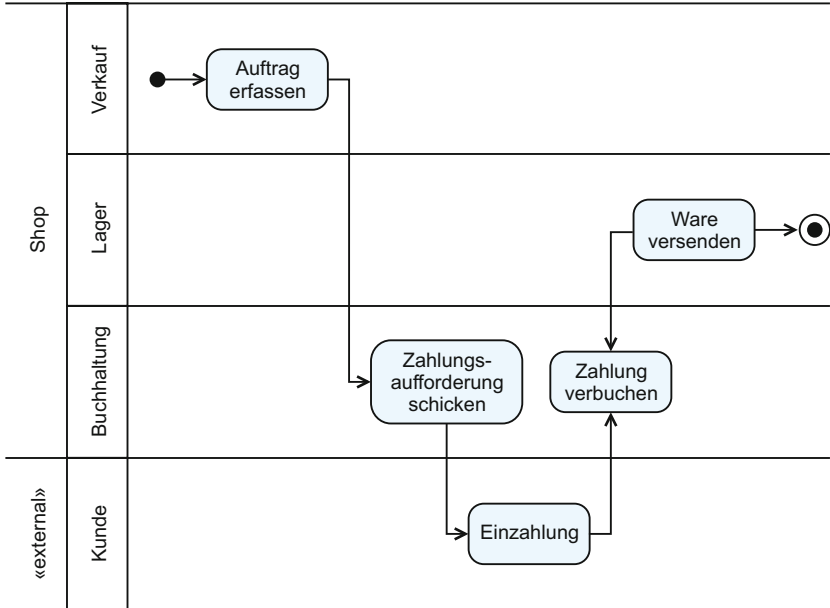


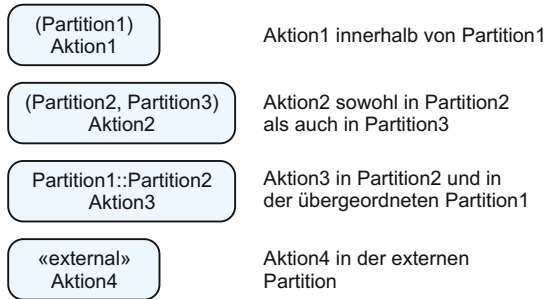
Abb. 10.1-21:
Hierarchie und
externer
Aktivitätsbereich.

dass eine Aktion außerhalb des eigentlichen Modells liegt. In der Abb. 10.1-21 wird außer den beschriebenen Aktivitätsbereichen der externe Aktivitätsbereich Kunde und die von ihm durchgeführte Aktion Einzahlung modelliert.

Vor allem in komplexeren Diagrammen ist es nicht immer sinnvoll, Aktivitätsbereiche in der Schwimmbahn-Notation zu modellieren. Daher bietet die UML als alternative Notation an, dass der Partitionsname in runden Klammern direkt über dem Aktionsnamen eingetragen wird (Abb. 10.1-22). Gehört eine Aktion zu mehr als einer Partition, dann werden diese durch Kommata getrennt. Die Doppelpunkt-Notation zeigt, dass es sich um eine hierarchisch strukturierte Partition handelt. Diese alternative Notation kann auch mit der Schwimmbahn-Notation kombiniert werden.

III 10 Dynamik

Abb. 10.1-22:
Alternative
Notation für
Aktivitätsbereiche
(Partitionen).



Beispiel 1b Die Auftragsbearbeitung aus Beispiel 1a kann nun verfeinert werden. Zuerst werden alle Daten des Kundenauftrags erfasst. Verlangt der Kunde Artikel, die nicht vorrätig sind, dann müssen sie erst beim Lieferanten bestellt werden. Sobald sie eingetroffen sind, wird die Kreditkarte des Kunden belastet oder alternativ geprüft, ob der Rechnungsbetrag per Vorkasse eingegangen ist. Die Rechnung wird gedruckt und zusammen mit der Ware versandt. Der Einfachheit halber soll gelten, dass keine unvorhergesehenen Probleme, wie z. B. Lieferant kann nicht liefern oder Kreditkarte gesperrt, auftreten.

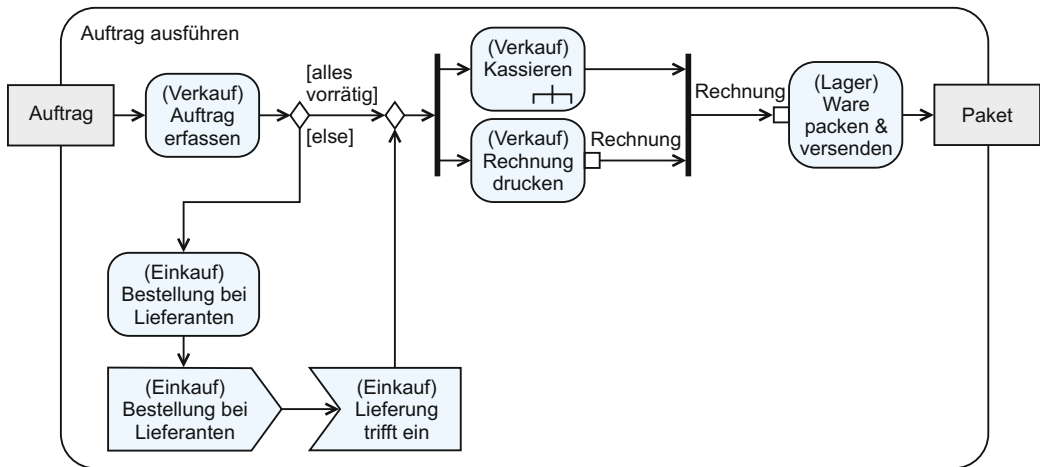


Abb. 10.1-23:
Aktivität zum
Ausführen eines
Auftrags mit
Aktivitätsaufruf
Kassieren.

Die Abb. 10.1-23 beschreibt die Aktivität Auftrag ausführen. Eingabeparameter ist der vorliegende Auftrag. Nach dem Erfassen der Daten verzweigt der Kontrollfluss. Sind alle bestellten Artikel vorrätig, dann finden die Aktionen Kassieren und Rechnung drucken statt, wobei deren zeitliche Reihenfolge beliebig ist. Sind nicht alle bestellten Artikel vorrätig, dann schickt der Einkauf eine Bestellung an den jeweiligen Lieferanten. Der Einkauf sendet ein Signal, um den Liefereingang abzu prüfen. Wenn das Ereignis Lieferung trifft ein

auftritt, geht der Kontrollfluss wieder zum Verkauf zurück. Ist der Auftrag vom Verkauf vollständig bearbeitet, erhält das Lager einen Auftrag zum Versenden der Ware.

Die Aktion Kassieren enthält rechts unten ein kleines grafisches Symbol. Es bedeutet, dass diese Aktion eine Aktivität aufruft, die in einem separaten Diagramm beschrieben wird (Abb. 10.1-24). Nach dem Prüfen der Zahlungsart erfolgt eine Entscheidung. Zahlt der Kunde per Kreditkarte, dann wird diese belastet und der Kassivorgang abgeschlossen. Diese Aktion wird vom Verkauf durchgeführt. Will der Kunde seinen Auftrag per Vorkasse bezahlen, dann erhält die Buchhaltung ein Signal, um einen Zahlungseingang abzufragen. Wenn das Ereignis Zahlungseingang auftritt, ist der Kassivorgang auch für diesen Fall abgeschlossen.

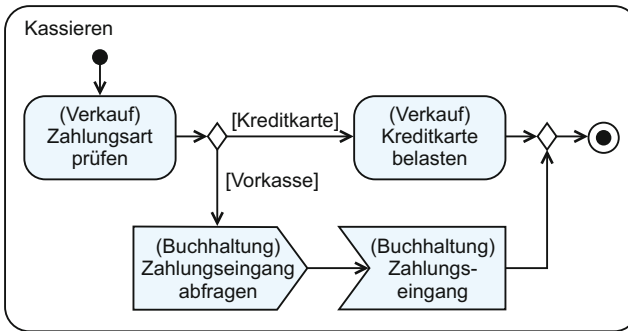


Abb. 10.1-24:
Aktivität
Kassieren.

Die Abb. 10.1-25 zeigt die Notationselemente für UML-Aktivitätsdiagramme ohne strukturierte Knoten im Überblick.

Aktivitäten besitzen in der UML 2 eine Semantik, die den Petrinetzen ähnelt (siehe »Petrinetze«, S. 303, »Aktivitätsdiagramme und Petrinetze«, S. 317).

Petrinetze

10.1.7 Box: Aktivität – Methode und Checkliste

Zum Erstellen und Überprüfen von Aktivitätsdiagrammen hilft die Box »Checkliste Aktivität«. Auf einige Aspekte zu der Box wird im Folgenden eingegangen.

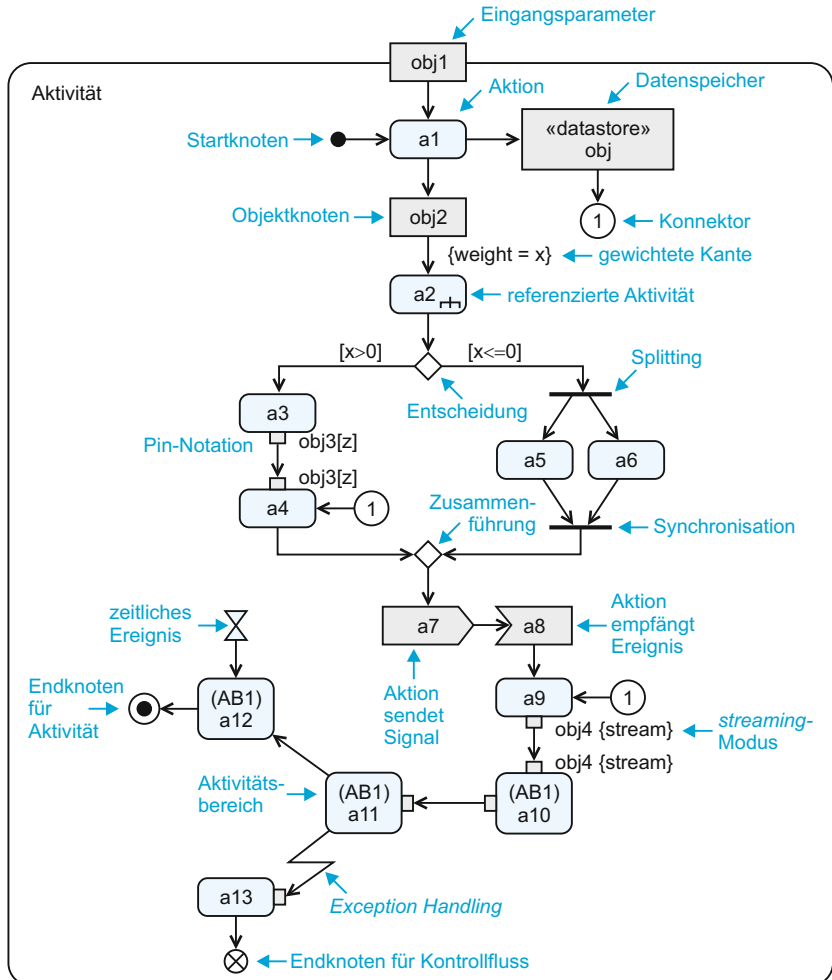
Konstruktive Schritte

Überlegen Sie, welches Ereignis die Verarbeitung auslöst. Tragen Sie den Startknoten links oben in das Diagramm ein. Welche Vorbedingungen (*preconditions*) müssen erfüllt sein, damit die Verarbeitung ausgeführt werden kann?

1 Start

III 10 Dynamik

Abb. 10.1-25:
Notationselemente
des Aktivitäts-
diagramms.



Beispiel 1a Aktivität: in Urlaub fliegen
Auslösendes Ereignis: Wecker klingelt
Vorbedingungen: Koffer gepackt, Taxi bestellt

2 Ende Welches Ziel soll mit der Verarbeitung im Erfolgsfall erreicht werden? Wann terminiert die Verarbeitung? Soll die gesamte Aktivität beendet werden (Endknoten) oder nur der jeweilige Pfad (*flow final*). Es sind auch zyklische Verarbeitungen möglich, die – zumindest theoretisch – nie terminieren. In diesem Fall entfällt der Endknoten.

Beispiel 1b Aktivität: in Urlaub fliegen
Im Erfolgsfall endet die Aktivität, wenn der Urlauber ins Flugzeug einsteigt.

Ergebnis

■ Aktivitätsdiagramm

Konstruktive Schritte

1 Welches Ereignis löst die Verarbeitung aus?

- Startknoten links oben eintragen.
- Vorbedingungen (*preconditions*) ermitteln.

2 Wann terminiert die Verarbeitung?

- Welches Ziel soll im Erfolgsfall erreicht werden?
- Terminiert die gesamte Verarbeitung oder nur der aktuelle Pfad?

3 Wie sieht der Standardfall aus?

4 Welche Erweiterungen zum Standardfall sind möglich?

- Werden Aktionen nur unter bestimmten Bedingungen ausgeführt?
- Tragen Sie Bedingungen an die Ausgangspfeile des Diamanten an.
- Wo wird der Kontrollfluss wieder zusammengeführt?

5 Ist parallele Verarbeitung möglich?

- Ist für bestimmte Verarbeitungsschritte die Reihenfolge irrelevant?
- Ist echte Parallelarbeit möglich?
- Wo wird der Kontrollfluss wieder synchronisiert?

6 Objektknoten

- Erzeugen die Aktionen Ausgabedaten oder benötigen sie Eingabedaten?
- Tragen Sie Objektknoten ein, um die Aussagefähigkeit des Diagramms zu erhöhen.
- Bei Objekten, die in verschiedenen Bearbeitungszuständen vorkommen, ist zusätzlich der jeweilige Zustand einzutragen.

7 Ein-/Ausgabeparameter

- Die Angabe von Ein- und Ausgabeparametern ist nur notwendig, wenn sie zusätzliche Informationen liefern.

8 Aktivitätsbereiche

- Verwenden Sie die Schwimmbahn-Notation, wenn die Aktivität höchstes 5 Bereiche enthält.
- Modellieren Sie die wichtigsten Schritte im ersten Aktivitätsbereich.
- Knoten, die zwei Bereiche betreffen, werden auf der Grenzlinie dargestellt.

Analytische Schritte

9 Aktionsnamen

- Aktionsnamen sollen ein starkes Verb enthalten.
- Aktionsnamen sollen im Kontext der Aktivität verständlich und eindeutig sein.

10 Black-Hole- und Miracle-Aktionen

- Gibt es Aktionsknoten, die keine Ausgabepfeile besitzen (*black holes*)?
- Gibt es Aktionsknoten, die keine Eingabepfeile besitzen (*miracles*)?

11 Trifft bei Entscheidungen genau eine Alternative zu?

- Sind die Bedingungen eindeutig definiert?

12 Ist die Menge der Bedingungen vollständig?

- Wird bei einer Entscheidung jede mögliche Bedingung durch einen Ausgabepfeil abgedeckt?
- Spezifizieren Sie eine Alternative mit *else*.

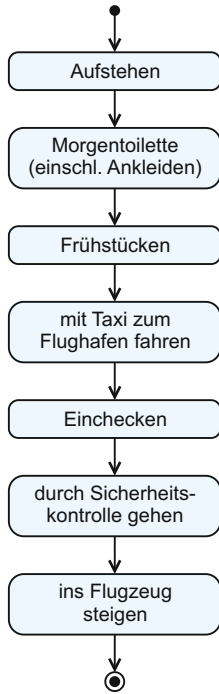
Box 1a: Checkliste Aktivität

Modellieren Sie zuerst den Hauptpfad der Aktivität, d. h., legen Sie eine Folge von Aktionsknoten für den Normalfall oder den einfachsten Fall fest. **3** Standardfall

III 10 Dynamik

Beispiel 1c Die Abb. 10.1-27 zeigt die Aktionen, die durchlaufen werden, wenn bei der Fahrt zum Flughafen alles optimal verläuft.

Abb. 10.1-27: Aktivitätsdiagramm für den optimalen Fall eines Starts in den Urlaub.



4 Entscheidungen

Welche Erweiterungen zum Standardfall sind möglich? Werden Aktionsknoten nur unter bestimmten Bedingungen ausgeführt? Tragen Sie einen Entscheidungsknoten (Diamant) ein. Jeder Ausgang aus dem Diamanten ist durch eine Bedingung zu spezifizieren. Spielen Sie die Alternativen gedanklich durch und tragen Sie die verschiedenen Bedingungen an die Ausgangspfeile des Diamanten an. Prüfen Sie, wo der Kontrollfluss wieder zusammengeführt wird (Diamant).

Beispiel 1d Aktivität: in Urlaub fliegen
Jeder weiß, dass beim Start in den Urlaub vieles schief gehen kann: Die Abb. 10.1-28 zeigt das Aktivitätsdiagramm mit den diversen Alternativen.

5 Parallele Verarbeitung

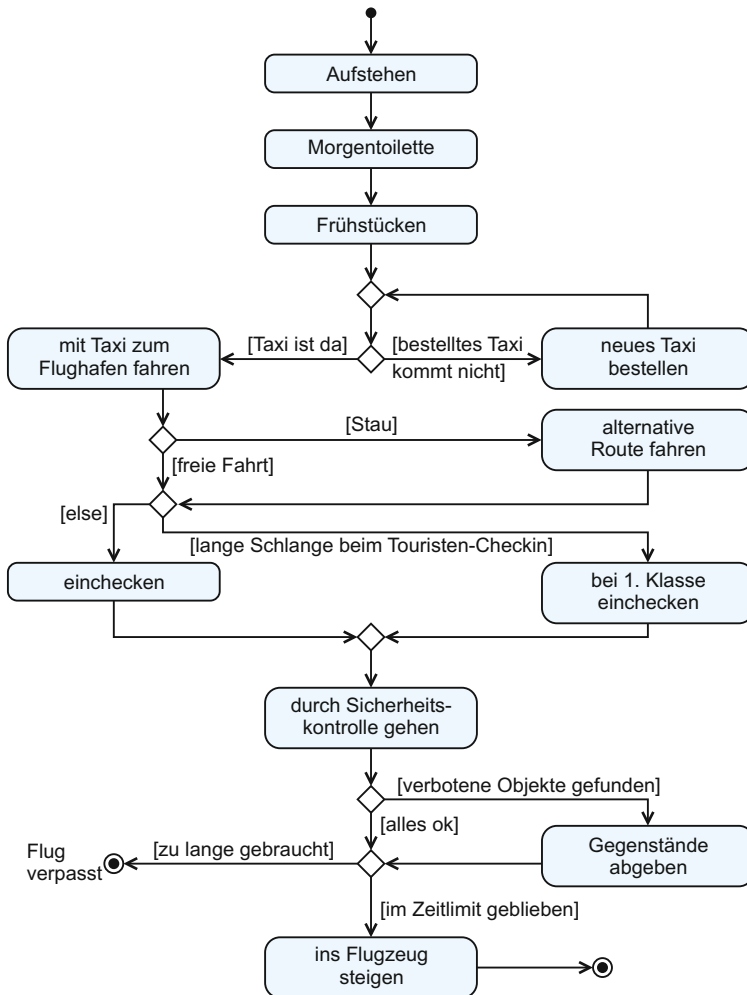
Ist für bestimmte Verarbeitungsschritte die Reihenfolge irrelevant oder ist Parallelarbeit möglich? Tragen Sie die Kontrollknoten zum Splitten (Balken) und Synchronisieren (Balken) des Kontrollflusses ein. Stellen Sie sicher, dass der Splitting-Balken genau einen Eingang und der Synchronisations-Balken genau einen Ausgang besitzt. Im Gegensatz zur Zusammenführung nach einer Entscheidung geht die Verarbeitung nach einer Synchronisation erst dann weiter, wenn alle Teilpfade durchlaufen wurden.

6 Objektknoten

Erzeugen die Aktionen Daten oder benötigen Sie Eingabedaten? Tragen Sie Objektknoten ein, wenn sich dadurch der Informationsgehalt des Aktivitätsdiagramms erhöht. Wenn ein Objekt in einem Aktivitätsdiagramm mehrmals, jedoch in verschiedenen Bearbeitungszuständen vorkommt, dann sollten Sie den Objektnamen um den jeweiligen Zustand (in eckigen Klammern) ergänzen.

10.1 Kontrollstrukturen III

Abb. 10.1-28: Aktivitätsdiagramm mit den wichtigsten Sonderfällen bei einem Start in den Urlaub.



Eine Rechnung besitzt in einem Use Case die Bearbeitungszustände ungeprüft, geprüft und bezahlt. Entsprechend werden für den Objektknoten Rechnung die folgenden Namen gewählt: Rechnung[ungeprüft], Rechnung[geprüft] und Rechnung[bezahlt].

Beispiel

Tragen Sie Ein- und Ausgabeparameter dann in ein Aktivitätsdiagramm ein, wenn deren Modellierung zusätzliche Information liefert.

7 Eingabe-/Ausgabeparameter

10.1.8 Zusammenfassung

Kontrollstrukturen legen innerhalb eines Algorithmus fest, in welcher Reihenfolge, ob und wie oft Anweisungen ausgeführt werden sollen. Die strukturierte Programmierung erlaubt nur solche Kon-

III 10 Dynamik

trollstrukturen, die genau einen Ein- und einen Ausgang haben. Man nennt solche Kontrollstrukturen daher lineare Kontrollstrukturen. Es lassen sich fünf verschiedene Typen unterscheiden:

- Die Sequenz,
- die Auswahl,
- die Wiederholung,
- der Aufruf und
- die Nebenläufigkeit

Alle Typen lassen sich beliebig miteinander kombinieren und ineinander schachteln.

In der UML ist es in Aktivitätsdiagrammen zusätzlich möglich, Objektflüsse durch Objektknoten zu spezifizieren. Durch Parameterknoten können Eingangs- und Ausgangsparameter beschrieben werden. Kontinuierliche Datenströme können durch einen Streaming-Modus modelliert werden. Außerdem können Ereignisse, Start- und Endeknoten und Aktivitätsbereiche spezifiziert werden.

Zur Klassifikation

- Kontrollstrukturen werden sowohl textuell (in Programmiersprachen, als Pseudocode) als auch grafisch (mit textuellen Annotationen) beschrieben (Struktogramme, UML).
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt – im Detaillierungsgrad aber je nach Phase unterschiedlich.
- Sie werden in allen Anwendungsbereichen verwendet.

10.2 Geschäftsprozesse und *Use Cases*

Geschäftsprozesse und *Use Cases* – zwei Begriffe, die in der Softwaretechnik immer wieder auftauchen.

Beide Begriffe werden teilweise synonym, teilweise überlappend und manchmal disjunkt verwendet. Daher wird zunächst eine begriffliche Klärung vorgenommen, und es werden verschiedene Notationen vorgestellt:

- »Konzepte und Notationen«, S. 251

Die Modellierung von Geschäftsprozessen wird an »ereignisgesteuerten Prozessketten« und an Aktivitätsdiagrammen beispielhaft gezeigt:

- »EKPs und Aktivitätsdiagramme«, S. 253

Use Cases können in der UML grafisch und textuell mit Hilfe von Schablonen spezifiziert werden:

- »*Use Case*-Diagramme und -Schablonen«, S. 255

Für die Erstellung und Überprüfung von *Use Cases* gibt es konstruktive und analytische Schritte:

■ »Box: *Use Case* – Methode und Checkliste«, S. 262

Zum Abschluss wird das Wichtigste zusammengefasst:

■ »Zusammenfassung«, S. 268

10.2.1 Konzepte und Notationen

Terminologie

Die Begriffe **Geschäftsprozess** (*business process*) und **Use Case** werden sowohl in der Literatur als auch in der Praxis unterschiedlich verwendet. Eine aus der Systemtheorie kommende Abgrenzung beider Begriffe wird in [UmMe06] vorgenommen, der sich hier angeschlossen wird.

»[...] ein Geschäftsprozess [ist] [...] eine durchgängige Folge von Aktivitäten entlang einer Wertschöpfungskette innerhalb eines Unternehmens zwischen einem internen oder externen Lieferanten und einem internen oder externen Kunden mit einem festgelegten Ergebnis von Wert für den Kunden und wirtschaftlichem Mehrwert für das Unternehmen« [UmMe06, S. 424].

Definition

»Ein Use Case ist das Beziehen einer Systemleistung durch die Außenwelt, i.e. eine durchgängige, zielgetriebene und von diesem Ziel abgeleitete ergebnisorientierte Menge an Interaktion« [UmMe06, S. 429]. Ein *Use Case* gibt also die Art und Weise an, wie ein Akteur die Benutzung des Systems aus der **Außensicht** wahrnimmt.

Definition

Use Cases werden im Deutschen auch als **Anwendungsfall** bezeichnet.

Akteure sind Rollen von Menschen oder Systeme, insbesondere Computersysteme, die als externe Beteiligte mit einem Unternehmen (Kunden) oder einem Softwareprodukt (Benutzer) kommunizieren und Daten austauschen. In der UML werden Akteure durch Strichmännchen oder selbst gewählte Symbole dargestellt.

Akteure

Weder durch einen Geschäftsprozess noch durch einen *Use Case* darf die Benutzungsoberfläche spezifiziert werden. Die Oberfläche ändert sich aufgrund neuer Techniken und aufgrund von *Usability*-Studien sehr schnell, und oft muss die Software auf verschiedenen Plattformen laufen.



III 10 Dynamik

Geschäftsprozesse vs. *Use Cases*

Bei *Use Cases* wird ein System aus der Außensicht betrachtet. Wie das System intern die durch die *Use Cases* spezifizierten Anforderungen umsetzt, ist irrelevant. Es kommt nur darauf an, dass die Anforderungen erfüllt werden.

Zitat »Ein Use Case ist die Artikulation des Wunsches eines Aktors [richtig Akteurs, *der Autor*] (eingenommen durch einen Menschen, eine Software, eine Hardware oder einen Zeitstempel), der die Unterstützung eines Systems für seine operativen Aufgaben wünscht. Das Use Case Ziel leitet sich daher von den Bedürfnissen des Aktors ab. Alle von diesem Ziel getriebenen Interaktionen zwischen Akteur und System führen zu einem messbaren Ergebnis, welches einem Akteur oder Stakeholder einen angestrebten Mehrwert (Wertschöpfung) bringen muss« [UmMe06, S. 424].

Im Gegensatz dazu beschreibt ein Geschäftsprozess die Prozesssteuerung *hinter* der Systemgrenze, gibt an, wie die Systemziele umgesetzt werden und wie die zielgetriebene Wertschöpfung im Systeminneren aussieht.

Use Cases und Geschäftsprozesse können dabei auf jeder der Stufen »Unternehmen«, »Unternehmensausschnitt« und »Software« eingesetzt werden.

Beide Begriffe sind notwendig, komplementär, aber disjunkt.

Obwohl ein *Use Case* die Außensicht modelliert, ist es möglich, dass ein *Use Case* White-Box-Aussagen, d. h. Aussagen aus dem Inneren des Systems, enthält.

Beispiel Bestellt ein Kunde in einem Online-Shop auf Rechnung, dann muss er u. U. das Einholen von Kreditinformationen gestatten. Da der Kunde einen Rechtsanspruch darauf hat zu wissen, wie das System mit seinen personenbezogenen Daten umgeht, muss das System Informationen aus seinem »Inneren« offenbaren. Dadurch ist aber die Art der Realisierung noch nicht festgelegt.

Notationen

Für die Beschreibung von **Geschäftsprozessen** gibt es eine Vielzahl unterschiedlicher Notationen, z. B.

- eEPK (erweiterte Ereignisgesteuerte Prozesskette): Erweiterte Form der grafischen Modellierungssprache EPK zur Darstellung von Geschäftsprozessen einer Organisation. Für jede Aufgabe kann die verantwortliche Organisationseinheit spezifiziert werden (siehe »EKPs und Aktivitätsdiagramme«, S. 253).
- BPMN (*Business Process Modeling Notation*): Unterscheidet Kontroll- und Nachrichtenfluss. Dadurch können verschiedene Prozesse in einem Diagramm dargestellt werden, die durch den Nach-

richtenfluss verbunden sind. Durch den Nachrichtenfluss können die Grenzen verschiedener prozessausführender Organisationseinheiten überschritten werden, siehe Website BPMN (<http://www.bpmn.org/>).

- Let's Dance: Choreografiesprache zur Modellierung des Kommunikationsverhaltens zwischen Organisationen [ZBD+06].
- BPEL (*Business Process Execution Language*): Eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, deren Aktivitäten durch Webservices implementiert werden (Beschreibung von Serviceorchestrierungen).
- UML-Aktivitätsdiagramme (siehe »Aktivitätsdiagramm«, S. 236)

Use Cases werden i. Allg. beschrieben durch

- UML-Use-Case-Diagramme und
- Use Case-Schablonen (siehe »Use Case-Diagramme und -Schablonen«, S. 255).

10.2.2 EKPs und Aktivitätsdiagramme

Eine weit verbreitete Notation für Geschäftsprozesse ist die **ereignisgesteuerte Prozesskette** (EPK)¹. Zentrale Elemente der EPK sind Funktionen, Ereignisse und Konnektoren. Mit Konnektoren lassen sich Verzweigungen und Zusammenführungen modellieren. Die Abb. 10.2-1 zeigt wichtige Notationselemente der EPK.

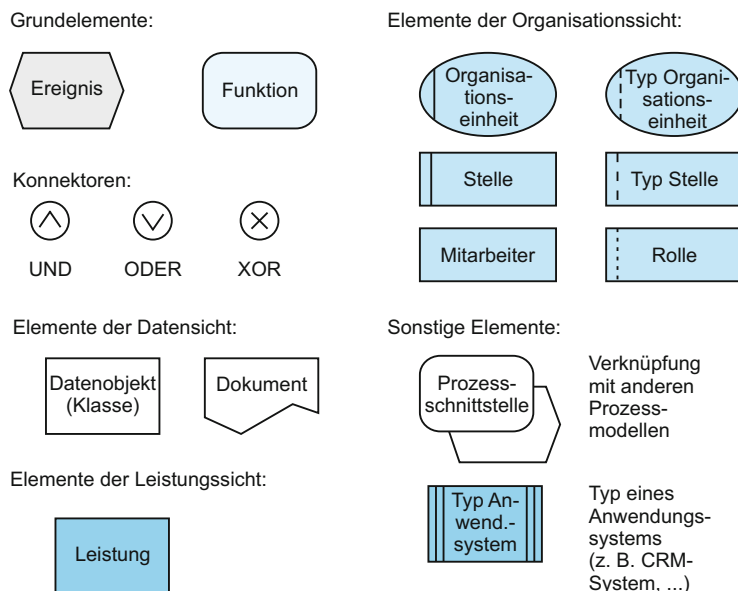


Abb. 10.2-1: Ereignis und Funktion sind die wichtigsten Konstrukte der EPK. Hinzu kommen Konnektoren zur Abbildung von Verzweigungen und Zusammenführungen sowie Elemente der Organisations-, Daten- und Leistungssicht, außerdem Prozess-Schnittstellen zur Verknüpfung mit anderen Prozessen.

¹Ein Teil der Inhalte wurde mit freundlicher Genehmigung der W3L GmbH dem Buch »Geschäftsprozessmanagement« von Thomas Allweyer entnommen.

III 10 Dynamik

Die Abb. 10.2-2 zeigt ein einfaches Beispiel eines Geschäftsprozesses Auftragsbearbeitung.

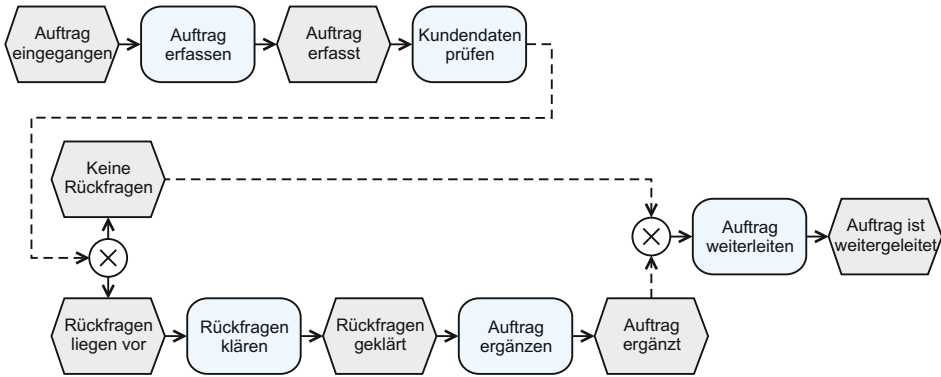


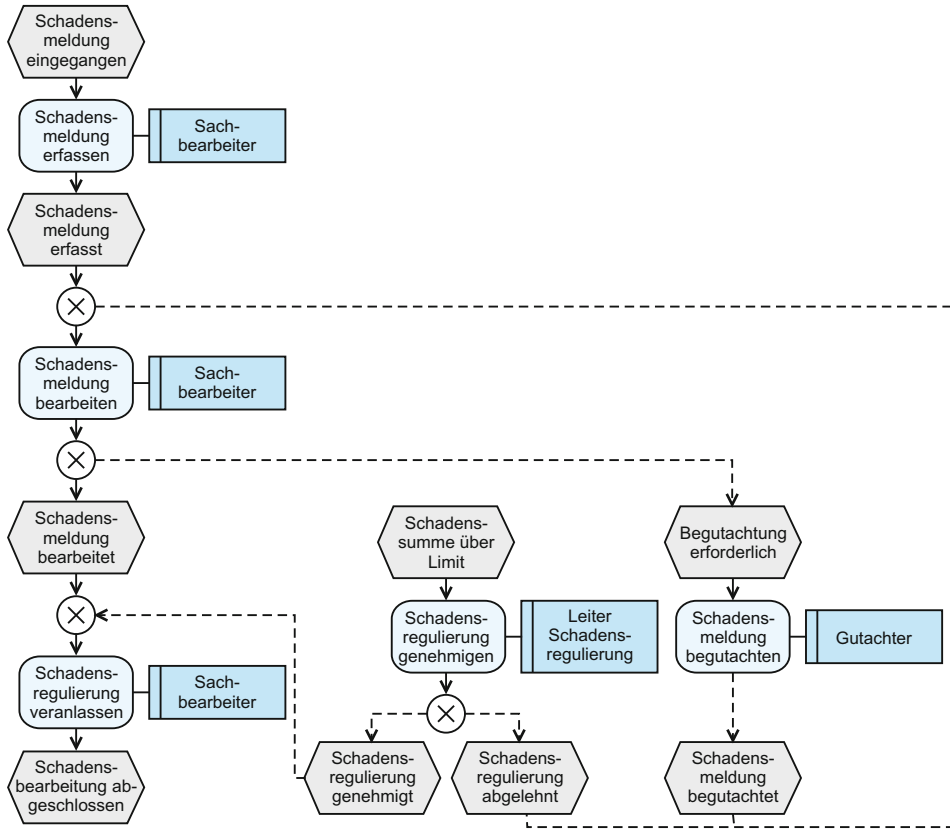
Abb. 10.2-2:
Beispiel für eine
einfache EPK aus
den Elementen
Ereignis, Funktion
und Konnektor.

Ein mit einer EPK modellierter Geschäftsprozess kann simuliert werden.

Der in der Abb. 10.2-3 dargestellte Beispielprozess zur Schadensabwicklung in einer Versicherung läuft wie folgt ab: Geht eine Schadensmeldung bei der Versicherung ein, so wird sie zunächst von einem Sachbearbeiter erfasst. Anschließend wird sie von einem Sachbearbeiter bearbeitet. Ein mögliches Ergebnis ist, dass die Schadensmeldung vollständig bearbeitet werden konnte. In diesem Fall veranlasst der Sachbearbeiter die Schadensregulierung, d. h. die Information des Versicherten sowie ggf. die Auszahlung der zu erstattenden Summe. Ist eine Begutachtung durch einen spezialisierten Gutachter erforderlich, dann führt einer der Gutachter des Unternehmens die Begutachtung durch, bevor ein Sachbearbeiter erneut die Bearbeitung der Schadensmeldung übernimmt. Überschreitet die zu erstattende Summe eine gewisse Höhe, dann ist eine Genehmigung durch den Leiter der Schadensregulierung erforderlich. Wird die Erstattung genehmigt, geht es mit der Veranlassung der Schadensregulierung weiter, ansonsten muss die Schadensmeldung erneut durch den Sachbearbeiter bearbeitet werden, um einen anderen Vorschlag zur Regulierung zu erarbeiten [Allw05, S. 244].

Geschäftsprozesse können auch durch UML-Aktivitätsdiagramme spezifiziert werden (siehe »Aktivitätsdiagramm«, S. 236).

Literatur [Allw05] mit E-Learning-Kurs.



10.2.3 Use Case-Diagramme und -Schablonen

Ein **Use Case** spezifiziert eine Sequenz von Aktionen, einschließlich möglicher Varianten, die das System in Interaktion mit Akteuren ausführt. Ein Use Case wird durch ein bestimmtes Ereignis ausgelöst und ausgeführt, um ein Ziel zu erreichen oder ein gewünschtes Ergebnis zu erstellen. Ein Use Case ist immer als *Black Box* zu verstehen: Er beschreibt das extern wahrnehmbare Verhalten eines Systems oder Subsystems, ohne auf die interne Struktur oder Details der Realisierung einzugehen.

Ein Akteur ist eine Rolle, die ein Benutzer des Systems spielt. Jeder Akteur hat einen gewissen Einfluss auf das System. Ein Akteur ist häufig eine Person. Es kann sich ebenso um eine Organisationseinheit oder ein externes System handeln, das mit dem zu modellierenden System kommuniziert. Akteure befinden sich stets außerhalb des Systems.

Abb. 10.2-3: Der Beispielprozess einer Schadensabwicklung in einer Versicherung soll mit Hilfe der Simulation untersucht werden.

Akteur

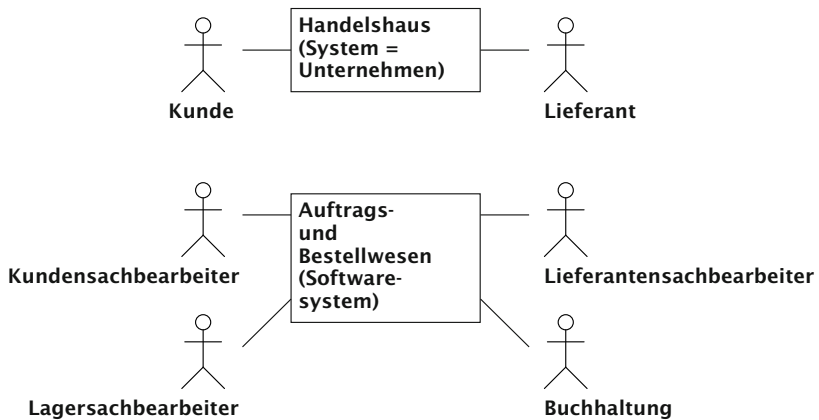
Stellt das betrachtete System ein Handelshaus dar (Abb. 10.2-4), dann sind Kunde und Lieferant Akteure. Die Buchhaltung ist dagegen kein Akteur des Handelshauses, denn sie befindet sich inner-

Beispiel

III 10 Dynamik

halb dieses Systems. Bei dem Softwaresystem, das Auftrags- und Bestellwesen unterstützt, ist dagegen die Buchhaltung ein Akteur, denn diese Abteilung ist außerhalb dieses Systems, muss jedoch mit dem Softwaresystem kommunizieren. In einer kleinen Firma können die Aufgaben des Kunden- und Lieferantensachbearbeiters durchaus von dergleichen Person ausgeführt werden. Trotzdem werden zwei Akteure – die Rollen, die diese Person spielt – identifiziert. Bei der Modellierung von Softwaresystemen sind die Akteure also diejenigen, die das System später bedienen bzw. Ergebnisse dieses Systems erhalten.

Abb. 10.2-4: Wer ist Akteur.



Subject In der UML wird das System, mit dem die Akteure interagieren, *Subject* genannt. »The subject of a use case could be a physical system or any other element that may have behavior, such as a component, subsystem, or class« [OMG09a, S. 596]. Ein *Subject* wird durch ein Rechteck dargestellt, in das oben der Name des *Subject* eingetragen wird (siehe Abb. 10.2-4).

Use Case-Diagramm



Das Zusammenspiel mehrerer *Use Cases* untereinander und mit den Akteuren wird in der UML in einem **Use Case-Diagramm** beschrieben (Abb. 10.2-5). Es gibt auf hohem Abstraktionsniveau einen guten Überblick über das System und seine Schnittstellen zur Umgebung.

Ein *Use Case* wird durch ein Oval dargestellt. Der *Use Case*-Name kann entweder in das Oval oder darunter geschrieben werden. Alternativ kann er auch in Klassennotation (Rechteck) mit einem *Use Case*-Piktogramm in der rechten oberen Ecke modelliert werden. Für einen *Use Case* können auch Attribute und Operationen angege-

10.2 Geschäftsprozesse und Use Cases III

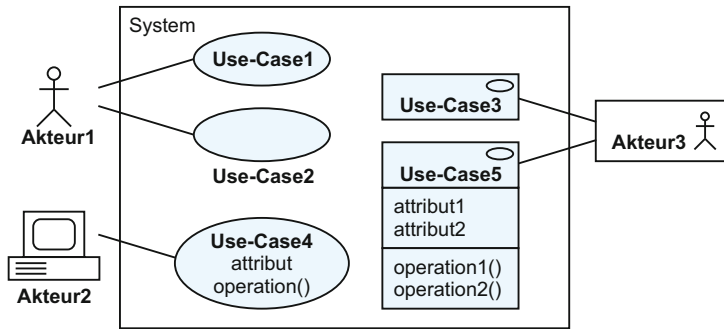


Abb. 10.2-5:
Notation für Use
Case-Diagramm.

ben werden. Bei einer größeren Anzahl von Attributen und Operationen ist es praktisch, wenn die Klassennotation (Rechteck) für den Use Case verwendet wird.

Akteure werden häufig – auch wenn es sich um ein externes System handelt – als Strichmännchen eingetragen. Alternativ können sie durch ein Symbol dargestellt werden, was oft sinnvoll ist, wenn es sich um ein technisches System (z. B. Computer) handelt. Auch hier kann alternativ ein Rechteck mit dem Akteur-Symbol oder dem Stereotypen «actor» gewählt werden. Eine Linie zwischen Akteur und Use Case bedeutet, dass eine Kommunikation stattfindet. Sie wird in der UML als Assoziation modelliert. Das betrachtete System (*subject*) wird im Use Case-Diagramm durch ein großes Rechteck modelliert, das alle Use Cases einschließt.

Die Abb. 10.2-6 modelliert das System »SemOrg« als Use Case-Diagramm.

Beispiel:
SemOrg

Als Name für einen Use Case sollte die Gerundiumform eines Verbs, ein Substantiv gefolgt von einem Verb oder der Anfangs- und Endpunkt des Prozesses gewählt werden. Charakterisiert ein anderer Begriff einen Use Case besser, dann kann von dieser Empfehlung auch abgewichen werden.

Empfehlung

Gerundium: Buchen

Substantiv und Verb: Buchung durchführen

Anfang und Ende: Von Anmeldung bis Buchung

Beispiel:
SemOrg

Strukturierung von Use Cases

In der UML gibt es drei Möglichkeiten, Use Cases zu strukturieren:

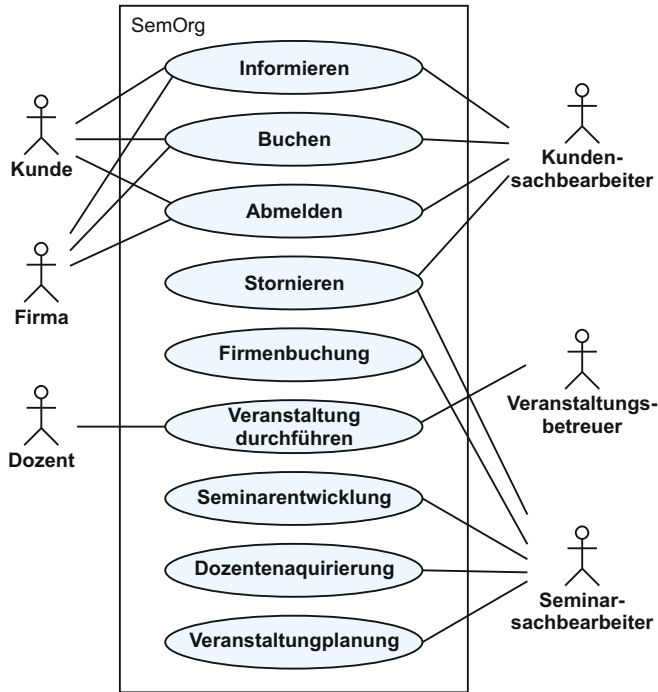
- Die *extend*-Beziehung,
- die *include*-Beziehung und
- die Generalisierungs-Beziehung.

Eine *extend*-Beziehung bzw. Erweiterungs-Beziehung zwischen einem Use Case A und einem Use Case B bedeutet, dass von dem Use Case A in den Use Case B verzweigt wird, wenn gegebene Bedin-

extend-Beziehung

III 10 Dynamik

Abb. 10.2-6:
Use Case-
Diagramm für das
Produkt SemOrg.



gungen erfüllt sind. Ist der *Use Case* B beendet, wird in den *Use Case* A zurückgekehrt. Wichtig ist, dass der *Use Case*, der erweitert wird (hier A), der ursprüngliche und fundamentale *Use Case* ist. Die Erweiterung wird vorgenommen, um die Komplexität des ursprünglichen *Use Case* zu reduzieren.

Eine *extend*-Beziehung wird durch einen gestrichelten Pfeil mit offener Pfeilspitze dargestellt, der von dem *Use Case*, der die Erweiterung zur Verfügung stellt, zum Standard-*Use Case* zeigt. Der Pfeil wird mit dem Stereotypen «extend» beschriftet. Die Bedingung der Erweiterung kann optional bei der Pfeilbeschriftung angegeben werden. Außerdem ist es möglich, die Stelle, an der die Erweiterung ausgeführt wird, im Oval des *Use Case* anzugeben (siehe unten). Die *extend*-Beziehung ermöglicht es also, einen komplexen *Use Case* zunächst in vereinfachter Form zu spezifizieren und komplexe Sonderfälle in die Erweiterungen zu verlagern.

Beispiel: Der *Use Case* Informieren (Von Anfrage bis Auskunft) lässt sich durch eine Erweiterung besser strukturieren (Abb. 10.2-7).

include-Beziehung Oft kommt es vor, dass einige *Use Cases* die gleichen Unter-*Use Cases* besitzen. Um einen solchen Unter-*Use Case* nicht doppelt zu beschreiben, wird er als selbstständiger *Use Case* modelliert. Dieser Unter-*Use Case* kann dann von mehreren *Use Cases* be-

10.2 Geschäftsprozesse und Use Cases III

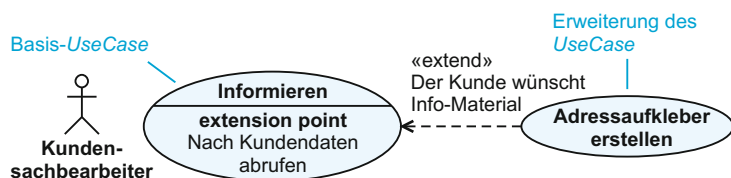


Abb. 10.2-7:
Strukturierung
von Use Cases mit
der extend-
Beziehung.

nutzt werden. Ein Unter-Use Case ist analog einem Unterprogramm in einer Programmiersprache. Der eingeschlossene Use Case steht *nie* für sich allein.

Für die Darstellung wird ebenfalls ein gestrichelter Pfeil mit offener Spitze verwendet, der mit dem Stereotypen «include» beschriftet wird. Der Pfeil zeigt vom Standard-Use Case zum eingeschlossenen Use Case.

In der Abb. 10.2-8 verwenden die beiden Use Cases Buchen und Firmenbuchung beide den Use Case Zahlungsmoral überprüfen.

Beispiel:
SemOrg



Abb. 10.2-8:
Strukturierung
von Use Cases mit
der include-
Beziehung.

Mit Hilfe der Generalisierungs-Beziehung können zu einem Use Case Kind-Use Cases modelliert werden. Ein Kind-Use Case erbt das Verhalten des Eltern-Use Case, analog wie bei der Vererbung zwischen Klassen. Der Kind-Use Case kann zusätzliches Verhalten hinzufügen oder Verhalten des Eltern-Use Case überschreiben. Der Kind-Use Case kann an jeder Stelle eingesetzt werden, an dem der Eltern-Use Case steht. Eine Generalisierungs-Beziehung wird durch einen Pfeil mit einer weißen bzw. transparenten Dreiecksspitze, die zum Eltern-Use Case zeigt, gekennzeichnet (siehe »Vererbung«, S. 150).

Generalisierung

In der Abb. 10.2-9 besitzt der Eltern-Use Case Krankenhaus aufnahme die zwei Kind-Use Cases Einweisung durch Arzt und Notfallaufnahme.

Beispiel



Abb. 10.2-9:
Strukturierung
von Use Cases mit
der
Generalisierungs-
Beziehung.

Es lassen sich **konkrete** und **abstrakte Use Cases** unterscheiden. Der Use Case Krankenhausaufnahme ist z. B. ein künstliches Gebilde, das nur zum Zweck der Verwendung durch die konkreten Kind-Use Cases existiert. Abstrakte Use Cases werden in der UML-Notation durch kursive Namen gekennzeichnet.

Konkret & abstrakt

III 10 Dynamik

Use Case-Schablonen

Im ersten Schritt kann jeder *Use Case* durch einen kurzen Text beschrieben werden. Außerdem sollten die beteiligten Akteure angegeben werden.

Beispiel 1a: **Use Case:** Buchen: Von Anmeldung bis Buchung

SemOrg **Akteure:** Kundensachbearbeiter

Beschreibung: Ein Kunde oder eine Firma meldet sich oder Mitarbeiter zu einer Veranstaltung an.

Während bei einfachen *Use Cases* eine umgangssprachliche Beschreibung ausreicht, können bei umfangreicheren Spezifikationen **Use Case-Schablonen** (*use case templates*) sinnvoll eingesetzt werden. Die Box »Schablone für *Use Cases*« zeigt ein Beispiel für eine *Use Case*-Schablone. Eine solche Schablone sollte als Checkliste betrachtet werden, d.h. sie ist nicht für jeden *Use Case* vollständig auszufüllen.

Beispiel 1b: **Use Case:** Buchen: Von Anmeldung bis Buchung /F30/, /F130/,
SemOrg /F150/

Ziel: Anmeldebestätigung und Rechnung an Kunden geschickt

Vorbedingung: -

Nachbedingung Erfolg: Kunde ist angemeldet

Nachbedingung Fehlschlag: Mitteilung an Kunden, dass Veranstaltung ausgebucht oder ausfällt oder nicht existiert oder Kunde ist im Zahlungsverzug

Akteure: Kundensachbearbeiter

Auslösendes Ereignis: Anmeldung des Kunden liegt vor

Beschreibung:

1 Kundendaten abrufen

2 Veranstaltung prüfen

3 Anmeldebestätigung und Rechnung erstellen

Erweiterung:

1a Kundendaten aktualisieren

1b Wenn Kunde Mitarbeiter einer Firma ist, dann Firmendaten erfassen bzw. wenn vorhanden, dann abrufen und aktualisieren

1c Zahlungsmoral überprüfen

3a Rechnungskopie an Buchhaltung

Alternativen:

1a Neukunden erfassen

2a Auf alternative Veranstaltungen hinweisen, wenn ausgebucht

2b Mitteilung »falsche Veranstaltung«, wenn nicht existierende Veranstaltung

Use Case: Name (was wird getan?)

Ziel: Globale Zielsetzung bei erfolgreicher Ausführung des *Use Case*.

Kategorie: primär, sekundär oder optional

Vorbedingung: Erwarteter Zustand, bevor der *Use Case* beginnt.

Nachbedingung Erfolg: Erwarteter Zustand nach erfolgreicher Ausführung des *Use Case*, d. h. Ergebnis des *Use Case*.

Nachbedingung Fehlschlag: Erwarteter Zustand, wenn das Ziel nicht erreicht werden kann.

Akteure: Rollen von Personen oder andere Systeme, die den *Use Case* auslösen oder daran beteiligt sind.

Auslösendes Ereignis: Wenn dieses Ereignis eintritt, dann wird der *Use Case* initiiert.

Beschreibung:

1 Erste Aktion

2 Zweite Aktion

Erweiterungen:

1a Erweiterung des Funktionsumfangs der ersten Aktion

Alternativen:

1a Alternative Ausführung der ersten Aktion

1b Weitere Alternative zur ersten Aktion

Der betrachtete *Use Case* kann nur ausgeführt werden, wenn die genannte **Vorbedingung** erfüllt ist. Die **Nachbedingung** eines *Use Case* A kann für einen *Use Case* B eine Vorbedingung bilden. Diese Angaben bestimmen also, in welcher Reihenfolge *Use Cases* ausgeführt werden können.

Unter **Beschreibung** erfolgt eine umgangssprachliche Spezifikation des *Use Case*. Die einzelnen Aufgaben werden der besseren Übersicht halber nummeriert. Wichtig ist, dass hier zunächst der **Standardfall**, d. h. der Fall, der am häufigsten auszuführen ist, beschrieben wird. Alle seltener eingesetzten Fälle werden unter **Erweiterungen** ausgeführt, wenn sie zusätzlich zu einer Aktion der Standardverarbeitung ausgeführt werden und unter **Alternativen**, wenn sie eine Aktion der Normalverarbeitung ersetzen.

Die Kategorie eines *Use Case* ist

- primär, wenn er notwendiges Verhalten beschreibt, das häufig benötigt wird,
- sekundär, wenn er notwendiges Verhalten beschreibt, das selten benötigt wird,
- optional, wenn er ein Verhalten beschreibt, das für den Einsatz des Systems zwar nützlich, aber nicht unbedingt notwendig ist.

Quelle: [Cock97]

**Box: Schablone
für Use Cases**

Bewertung

- + Im Mittelpunkt stehen die zentralen Arbeitsabläufe und nicht elementare Funktionen.
- + Konzentration auf die Standardabläufe.
- + Verschiedene Notationen für verschiedene Abstraktionsniveaus.
- + Auch für den Auftraggeber bzw. die Fachabteilung leicht verständlich.
- Gefahr, zu sehr ins Detail zu gehen.
- Gefahr, *Use Cases* als Kontrollstrukturen zu spezifizieren.
- Gefahr, dass Verbindungen zur Benutzungsoberfläche hergestellt werden.

III 10 Dynamik

Literatur In [DPF05] werden ein Fehlerklassifikationsschema für *Use Cases* und ein integriertes Qualitäts sicherungskonzept vorgestellt. Weiterführende Informationen zu *Use Cases* sind enthalten in [Cock03], [ScWi98] und [BRJ99].

10.2.4 Box: *Use Case* – Methode und Checkliste

Zum Erstellen und Überprüfen von *Use Cases* helfen die Boxen »Checkliste *Use Cases*«. Um einen guten Überblick über die *Use Cases* zu erhalten, sollten Sie zunächst *Use Case*-Diagramme erstellen. Für ein kleines System ist eventuell nur ein Diagramm notwendig. Für mittlere bis große Systeme sind mehrere Diagramme zu modellieren.

Das Formulieren von *Use Cases* bietet eine gute Möglichkeit, die Anforderungen an ein System besser zu verstehen. Zunächst sollte man sich auf die primären *Use Cases* konzentrieren, um ein Verständnis für die Kernanforderungen an das System zu erhalten. Die Anzahl der *Use Cases* hängt stark vom jeweiligen Anwendungstyp ab. Bei einem umfangreichen System müssen Sie zuvor Subsysteme bilden.

Use Case-
Diagramm Arbeiten Sie zu einem Zeitpunkt immer nur an einem *Use Case*. Interviewen Sie die Benutzerrepräsentanten und die Experten des jeweiligen Fachgebiets. *Use Cases* sollen so dokumentiert werden, dass sie sowohl für die Interviewten als auch für andere Systemanalytiker verständlich sind. Tragen Sie jeden identifizierten *Use Case* in das *Use Case*-Diagramm ein, um einen besseren Überblick zu erhalten

Schablone *Use Cases* sollen zunächst auf einer hohen Abstraktionsebene formuliert werden. Sonderfälle werden zunächst *nicht* betrachtet. Die Verwendung einer Schablone zwingt dazu, einen Standardfall festzulegen und getrennt über mögliche Erweiterungen und Alternativen nachzudenken.

Konstruktive Schritte

- 1 Wer ist der Akteur? Akteure befinden sich immer außerhalb des betrachteten Systems und kommunizieren mit den *Use Cases* des Systems. Handelt es sich bei dem zu modellierenden System um ein Softwaresystem, dann ist der Akteur derjenige, der später die entsprechenden Aufgaben mit dem Softwaresystem durchführt. Obwohl diese Definition zunächst einleuchtend klingt, ist es in der Praxis nicht immer einfach, den Akteur zu ermitteln.

Beispiel: Der *Use Case* Buchen: Von Anmeldung bis Rechnung legt zunächst nahe, dass der Kundensachbearbeiter diese Tätigkeit durchführt. Ist das System jedoch internetfähig, dann können auch Kunden und Firmen ihre Buchung selbst durchführen, d. h. sie sind auch Akteure.

Ergebnisse

- *Use Case*-Diagramm
Alle *Use Cases* und Akteure sind eingetragen.
- Beschreibung der *Use Cases*
Alle *Use Cases* sind umgangssprachlich oder mittels Schablone beschrieben.

Konstruktive Schritte

1 Akteure ermitteln

- Welche Personen führen diese Aufgaben zurzeit durch und besitzen daher wichtige Kenntnisse über die durchzuführenden Arbeitsabläufe? Welche Rollen spielen diese Personen?
- Welche Personen werden zukünftig diese Aufgaben durchführen und auf welche Vorkenntnisse muss die Benutzungsoberfläche abgestimmt werden? Welche Rollen spielen diese Personen?
- Wo befindet sich die Schnittstelle des betrachteten Systems bzw. was gehört nicht mehr zu dem System?

2 Use Cases für die Standardverarbeitung ermitteln

- Primäre und ggf. sekundäre *Use Cases* betrachten
- Welche Standardverarbeitung besitzen sie?

2a mittels Akteuren

- Sind die Akteure Personen?
- Welche Arbeitsabläufe lösen sie aus?
- An welchen Arbeitsabläufen wirken sie mit?

2b mittels Ereignissen (Akteure sind externe Systeme)

- Erstellen einer Ereignisliste.
- Für jedes Ereignis einen *Use Case* identifizieren.
- Externe und zeitliche Ereignisse unterscheiden.

2c mittels Aufgabenbeschreibungen

- Was sind die Gesamtziele des Systems?
- Welches sind die zehn wichtigsten Aufgaben?
- Was ist das Ziel jeder Aufgabe?

3 Use Cases für die Sonderfälle formulieren

- Erweiterungen und Alternativen mittels Schablone erstellen.
- Aufbauend auf Standardfunktionalität mit *extend* die Sonderfälle formulieren, d. h. erweiterte *Use Cases* beschreiben.

4 Aufteilen komplexer Use Cases

- Komplexe Schritte als eigene *Use Cases* spezifizieren (*include*).
- Komplexe *Use Cases* (viele Sonderfälle) in mehrere *Use Cases* zerlegen und Gemeinsamkeiten mit *include* modellieren.
- Umfangreiche Erweiterungen als *Use Cases* spezifizieren (*extend*).

5 Gemeinsamkeiten von Use Cases ermitteln

- Auf redundanzfreie Beschreibung achten (*include*).
- Prüfen, ob eine Generalisierungsstruktur möglich ist.

Box 1a: Checkliste Use Cases

In diesem Beispiel lassen sich die Akteure daher nur dann ermitteln, wenn Informationen über den Einsatz des Gesamtsystems bekannt sind.

In Abhängigkeit vom Typ der Anwendung gibt es mehrere Wege, um *Use Cases* zu modellieren (vergleiche [Hrus98]):

Identifikation von
Use Cases

III 10 Dynamik

2 Zuerst Standardfälle Im ersten Schritt konzentriert man sich auf die Standardfälle und ignoriert die Sonderfälle. Ein häufiger Fehler beim Ermitteln von *Use Cases* besteht darin, in der Flut von Sonderfällen und Details zu »ertrinken«.

2a Akteure [JCJ+92] empfehlen, von den Akteuren des Systems auszugehen. Bei Personen werden deren typische Arbeitsabläufe analysiert. In Interviews sind folgende Fragen zu stellen:

- Welches Ereignis löst den Arbeitsablauf aus?
- Welche Eingabedaten werden benötigt?
- Welche Schritte sind auszuführen?
- Ist eine Reihenfolge der Schritte festgelegt?
- Welche Zwischenergebnisse werden erstellt?
- Welche Endergebnisse werden erstellt?
- Welche Vorbedingungen müssen erfüllt sein?
- Welche Nachbedingungen (Vorbedingungen anderer *Use Cases*) werden sichergestellt?
- Wie wichtig ist diese Arbeit?
- Warum wird diese Arbeit durchgeführt?
- Kann die Durchführung verbessert werden?

Beispiel: SemOrg In der Seminarorganisation ist ein Akteur der Veranstaltungsbetreuer, dessen Aufgabe es ist, Veranstaltungen zu betreuen. Daraus lässt sich der *Use Case* »Veranstaltung durchführen« ableiten.

2b Ereignisse Sind Akteure beispielsweise organisatorische Einheiten oder technische Schnittstellen, dann sollte eine Ereignisliste erstellt werden. Es ist zu überlegen, welche Ereignisse der Umgebung für das System relevant sind. Für jedes Ereignis muss ein *Use Case* (UC) existieren, der darauf reagiert bzw. entdeckt, dass ein entsprechendes Ereignis vorliegt. Es lassen sich externe Ereignisse und zeitliche Ereignisse unterscheiden. Externe Ereignisse treten außerhalb des betrachteten Systems auf. Zeitliche Ereignisse werden im Allgemeinen im System produziert.

Beispiel: SemOrg Eine Ereignisliste kann beispielsweise folgendermaßen aussehen:

- Kunde schreibt (Brief, Fax, email) oder ruft an: →UC Informieren.
- Anmeldung des Kunden liegt vor: →UC Buchen.
- Abmeldung des Kunden liegt vor: →UC Abmelden.
- Dozent sagt wegen Krankheit ab: →UC Stornieren.
- Anmeldung einer Firma liegt vor: →UC Firmenbuchung.
- Anfangstermin der Veranstaltung (zeitliches Ereignis): →UC Veranstaltung durchführen.
- Beginn der Planungsperiode (zeitliches Ereignis): →UC Seminarentwicklung.
- Beginn der Planungsperiode oder sporadisch (zeitliches Ereignis): →UC Dozentenakquirierung.

10.2 Geschäftsprozesse und Use Cases III

- Beginn der Planungsperiode oder sporadisch (zeitliches Ereignis):
→UC Veranstaltungsplanung.

In dem Kapitel »Fallstudie: Fensterheber – Die fachliche Lösung«, S. 575, sind eine Reihe von *Use Cases* aufgeführt, die von verschiedenen Ereignissen ausgelöst werden. Beispiel:
Fensterheber

Ist es schwierig anhand von Akteuren oder Ereignissen *Use Cases* zu identifizieren, dann kann man die Aufgaben des Systems beschreiben. Zunächst formuliert man den Zweck bzw. die Ziele des Systems (siehe »Visionen und Ziele«, S. 456). Aus diesen Zielen werden die notwendigen Aufgaben abgeleitet. Die zehn wichtigsten Aufgaben sind zu überlegen. Jede Aufgabe wird umgangssprachlich mit 25 – oder weniger – Worten beschrieben. Zu beantworten ist die Frage: Was ist das Ziel dieser Aufgabe bzw. der Nutzen dieser Aufgabe für das Gesamtsystem? 2c Aufgaben

Nach der Erstellung der Standardfälle werden im zweiten Schritt die Erweiterungen und Sonderfälle modelliert, die nur unter bestimmten Bedingungen auftreten. 3 Sonderfälle

Erweiterungen sind beispielsweise

- optionale Teile eines *Use Case*,
- komplexe oder alternative Möglichkeiten und
- Aufgaben, die nur selten durchgeführt werden.

Diese Vorgehensweise hat den Vorteil, dass die Basisfunktionalität leicht zu verstehen ist und erst im zweiten Schritt die Komplexität in das System integriert wird. Sonderfälle können bei Verwenden der Schablone unter *Erweiterungen* und *Alternativen* aufgeführt werden. Umfangreiche Sonderfälle sind als eigenständige *Use Cases* zu modellieren und mit *extend* an die Standardverarbeitung anzubinden.

Bei einer Versicherungsgesellschaft ist ein Schadensfall zu bearbeiten (vgl. [Cock97]). Beispiel

Use Case: bearbeite Schadensfall

Ziel: Bezahlung des Schadens durch die Versicherung

Kategorie: primär

Vorbedingung: -

Nachbedingung Erfolg: Schaden ganz oder teilweise bezahlt

Nachbedingung Fehlschlag: Forderung abgewiesen

Akteure: Schadenssachbearbeiter

Auslösendes Ereignis: Schadensersatzforderung des Antragstellers, d.h. der versicherten Person

Beschreibung:

1 Der Sachbearbeiter prüft die Forderung auf Vollständigkeit.

2 Der Sachbearbeiter prüft, ob eine gültige Police vorliegt.

3 Der Sachbearbeiter prüft alle Details der Police.

4 Der Sachbearbeiter errechnet den Betrag und überweist ihn an den Antragsteller.

III 10 Dynamik

Erweiterungen:

1a Die vorliegenden Daten vom Antragsteller sind nicht vollständig. Dann muss der Sachbearbeiter diese Informationen nachfordern.

2a Der Antragsteller besitzt keine gültige Police. Der Sachbearbeiter teilt ihm mit, dass keine Ansprüche bestehen, und schließt den Fall ab.

4a Der Schaden wird durch die Police nicht abgedeckt. Der Sachbearbeiter teilt dies dem Antragsteller mit und schließt den Fall ab.

4b Der Schaden wird durch die Police nur unvollständig abgedeckt. Der Sachbearbeiter verhandelt mit dem Antragsteller, bis zu welchem Grad der Schaden bezahlt wird.

Alternativen: -

4 Aufteilen Im Allgemeinen besteht ein *Use Case* aus mehreren Teilaufgaben. Unter Umständen ist ein einzelner Schritt eines *Use Case*, der auf einer hohen Abstraktionsebene formuliert wurde, so komplex, dass er selbst ein *Use Case* ist. Dieser Zusammenhang wird im Diagramm mittels der *include*-Beziehung angegeben.

Treten in einem *Use Case* viele Sonderfälle auf, dann sollte ebenfalls geprüft werden, ob das Verhalten nicht besser durch mehrere *Use Cases* beschrieben wird.

Sind die Erweiterungen eines *Use Case* sehr umfangreich, dann sollte ein eigener *Use Case* spezifiziert werden, der mittels *extend* mit dem Standard-*Use Case* verbunden wird.

5 Gemeinsamkeiten Besitzen zwei *Use Cases* einen gemeinsamen Teil, dann ist dieser herauszulösen und mit *include* zu verknüpfen. Die *include*-Beziehung dient in erster Linie der redundanzfreien Beschreibung von *Use Cases*. Man kann *include*-Beziehungen auch als eine Art Funktionsaufruf ansehen.

Mit der *include*-Beziehung kann die funktionale Zerlegung eines Systems beschrieben werden. *Use Cases* dürfen jedoch nicht zu stark verfeinert werden. Es entsteht sonst eine Art Funktionsbaum (siehe »Funktionsbaum«, S. 143). Das ist jedoch nicht der Sinn dieses Konzepts.

Prüfen, ob zwischen den Akteuren und zwischen den *Use Cases* jeweils Generalisierungsstrukturen vorhanden sind?

Use Case vs. Funktion

Zwischen *Use Cases* und der klassischen funktionalen Zerlegung gibt es folgende Unterschiede:

- *Use Cases* beschreiben die mit dem System auszuführenden Arbeitsabläufe, d. h., welche Aufgaben mit dem System durchgeführt werden sollen. Diese Aufgaben werden meistens vom Softwaresystem ausgeführt, können aber auch organisatorischer Natur sein. Sie werden auch dazu verwendet, um Testfälle zu spezifizieren.

- Die klassische funktionale Zerlegung gibt an, welche Funktionen das System – *unabhängig* von den jeweiligen Arbeitsabläufen – zur Verfügung stellt.
- *Use Cases* werden nur in der Anforderungsspezifikation eingesetzt und dienen als *high level documentation* des Systemverhaltens.
- Die klassische funktionale Zerlegung kann dagegen sowohl in der Spezifikation als auch im Entwurf eingesetzt werden.
- Ein *Use Case* beschreibt immer einen kompletten Ablauf auf hohem Abstraktionsniveau von Anfang bis Ende. Er besteht daher im Allgemeinen aus mehreren Schritten oder Transaktionen. Jeder Schritt kann einen weiteren *Use Case* oder eine Operation (z. B. drucke Rechnung) darstellen. Im Extremfall kann ein *Use Case* auf eine einzige Operation abgebildet werden.
- In der klassischen funktionalen Zerlegung wird eine Funktion in der Regel auf genau eine Operation abgebildet.

Anzahl Use Cases

Die Anzahl der *Use Cases* hängt wesentlich vom gewählten Abstraktionsniveau und vom Anwendungsbereich ab:

- Ein kleineres System (zwei bis fünf Mitarbeiterjahre) besteht aus 3 bis 20 *Use Cases*.
- Ein mittleres System (10 bis 100 Mitarbeiterjahre) kann 10 bis 60 *Use Cases* enthalten.
- Größere Systeme, z. B. Anwendungen für Banken, Versicherungen, Verteidigung und Telekommunikation können Hunderte von *Use Cases* enthalten.
- [Booc96] erwartet bei einem Projekt mittlerer Komplexität etwa ein Dutzend *Use Cases*.
- [Cock97] gibt folgende Größen an: Ein Projekt von 50 Mitarbeiterjahren mit 50 *Use Cases* und ein Projekt mit 30 Mitarbeiterjahren (18 Monate Entwicklungsdauer) mit 200 *Use Cases*.

Wie die unterschiedlichen Zahlen zeigen, gibt es über das Abstraktionsniveau und die genaue Definition von *Use Cases* verschiedene Auffassungen.

Analytische Schritte

Die *Use Cases* sind so zu formulieren, dass der Auftraggeber sie lesen und verstehen kann. Die Kommunikation der Akteure mit dem System steht im Mittelpunkt. Es wird weder die interne Struktur noch werden die Algorithmen beschrieben. Der Standardfall ist immer komplett zu spezifizieren. Eine Beschreibung sollte maximal eine Seite umfassen.

6 »Gute«
Beschreibung

Box 1b:
Checkliste Use
Cases

Analytische Schritte

6 »Gute« Beschreibung

- Verständlich für den Auftraggeber.
- Extern wahrnehmbares Verhalten.
- Fachliche Beschreibung des Arbeitsablaufs auf hohem Abstraktionsniveau.
- Beschreibt Standardfall vollständig und Sonderfälle separat.
- Maximal eine Seite.

7 Fehlerquellen

- Zu kleine und damit zu viele *Use Cases*.
- Zu frühe Betrachtung von Sonderfällen.
- Zu detaillierte Beschreibung der *Use Cases*.
- Verwechseln von *include*- und *extend*-Beziehungen.
- *Use Cases* beschreiben Dialogabläufe.

10.2.5 Zusammenfassung

Zitat »Use Cases beschreiben den dynamischen Leistungs- und Nachrichtenaustausch des Systems mit seiner Umwelt aus Außensicht. [...] Die **Menge aller Use Cases** begründet die Systemgrenze und damit den Zweck des Systems. [...]

Die Menge aller Geschäftsprozesse ist eine vollständige Beschreibung, wie das System zu einem *bestimmten Zeitpunkt* seine internen Einzelteile dynamisch miteinander verknüpft hat, um die Anforderungen, die sich aus der Differenz zur Umwelt ergeben (die Use Cases) zu erfüllen. Die Systemleistung muss nach außen hin zielbefriedigend sein, um die Existenz des Systems sichern zu können« [UmMe06, S. 429].

Geschäftsprozesse

Zur Modellierung von Geschäftsprozessen gibt es verschiedene Konzepte und Notationen, die sich durch verschiedene Zielsetzungen und Detaillierungsgrade unterscheiden. Oft werden ereignisgesteuerte Prozessketten (EKPs) verwendet. Bedingt einsetzbar sind dafür auch UML-Aktivitätsdiagramme.

Zur Klassifikation

- Geschäftsprozesse werden in der Regel grafisch (mit textuellen Annotationen) in unterschiedlichen Notationen beschrieben.
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden in der Spezifikations- und der Entwurfsphase eingesetzt.
- Sie werden vorwiegend in kaufmännisch/administrativen Anwendungen verwendet.

Use Cases

Durch *Use Cases* können die Arbeitsabläufe von Akteuren mit dem System auf einer hohen Abstraktionsebene aus der Außensicht beschrieben werden. Die Dokumentation erfolgt in der UML durch *Use Case*-Diagramme. Zur Spezifikation einzelner *Use Cases* kann eine *Use Case*-Schablone sinnvoll eingesetzt werden. Vorbedingungen geben dabei an, welche Bedingungen vor dem Ausführen einer Verarbeitung erfüllt sein müssen, Nachbedingungen beschreiben die bewirkten Veränderungen.

Ausgangspunkt beim Identifizieren von *Use Cases* sind Akteure. Dann werden die Standardfälle betrachtet und erst danach die Sonderfälle. Aufteilungen des *Use Case* und/oder Nutzung von vorhandenen *Use Case* sind möglich.

Zur Klassifikation

- *Use Cases* werden sowohl textuell (in *Use Case*-Schablonen) als auch grafisch (mit textuellen Annotationen) beschrieben (UML-*Use-Case*-Diagramme).
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden in der Spezifikationsphase eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

10.3 Zustandsautomaten

Bei vielen Systemen hängt das Ergebnis oder die Ausgabe nicht nur von einer Eingabe oder einem Ereignis ab, sondern auch von der Historie, die das System bis dahin durchlaufen hat. Zum Beschreiben solcher Zusammenhänge eignen sich Zustandsautomaten, auch endliche Automaten genannt.

Unter einem **Automaten** versteht man im Allgemeinen ein technisches Gerät, das zu einer Eingabe ein bestimmtes Ergebnis ausgibt. Beispiele dafür sind Münzwechselautomaten, Fahrkartenautomaten und Warenautomaten. In der Informatik steht der Automatenbegriff vorwiegend für mathematische Modelle von Geräten oder Systemen, die Informationen verarbeiten und dabei Antworten auf Ereignisse oder Eingaben geben.

Ein endlicher Automat oder **Zustandsautomat** (*finite automaton*, *finite state machine*, *sequential machine*) besteht aus einer endlichen Anzahl von internen Konfigurationen – **Zustände** genannt. Der Zustand eines Systems beinhaltet implizit die Informationen, die sich aus den bisherigen Eingaben ergeben haben und die benötigt werden, um die Reaktion des Systems auf noch folgende Eingaben zu bestimmen.

Endlicher
Automat,
Zustandsautomat

III 10 Dynamik

i Ein Zustandsautomat lässt sich systematisch erstellen:

■ »Erstellung eines Zustandsautomaten«, S. 270

Für Zustandsautomaten gibt es sowohl grafische als auch tabellarische Darstellungen:

■ »Notationen«, S. 272

Es gibt zyklische Zustandsautomaten und Zustandsautomaten, die einen oder mehrere Endzustände besitzen:

■ »Zustandsautomat mit Endzuständen«, S. 274

Außerdem werden Zustandsautomaten danach unterschieden, ob eine Aktion an den Zustandsübergang oder den Zustand gekoppelt ist:

■ »Mealy-Automat vs. Moore-Automat«, S. 275

Zur Modellierung komplexer Probleme wurden Zustandsautomaten um weitere Konzepte erweitert:

■ »Zustandsautomat nach Harel«, S. 277

In der objektorientierten Softwareentwicklung werden zusätzlich zwei Arten von Zustandsautomaten unterschieden:

■ »Verhaltens- vs. Protokollzustandsautomaten«, S. 287

Stochastische Abhängigkeiten zwischen Systemzuständen können durch Markov-Ketten – dargestellt durch Zustandsautomaten – spezifiziert werden:

■ »Markov-Ketten«, S. 292

Zustandsautomaten können methodisch entwickelt und überprüft werden:

■ »Box: Zustandsautomat – Methode und Checkliste«, S. 295

Zum Abschluss wird das Wichtigste zusammengefasst:

■ »Zusammenfassung«, S. 301

Zur Historie Endliche Automaten wurden von Huffman, Moore und Mealy in den Jahren 1954 bis 1956 entwickelt. Die Idee entstand in Anlehnung an neurale Netze (1943) und Schaltkreise.

10.3.1 Erstellung eines Zustandsautomaten

Beispiel 1a: Es soll das Stellen der Uhrzeit einer Digitaluhr mit zwei Einstell-
Digitaluhr Druckknöpfen modelliert werden. Knopf 1 erlaubt es, den Stellmodus (Normalzeit, Stunden stellen, Minuten stellen, Sekunden stellen) sequenziell zu wählen. Knopf 2 ermöglicht das Einstellen der Zeit entsprechend dem gewählten Stellmodus.

1. Schritt: Um Anforderungen als Zustandsautomaten zu modellieren, müssen in der Aufgabenstellung die Zustände, die Eingaben bzw. Ereignisse und die Ausgaben bzw. Ergebnisse identifiziert werden.

Beim Stellen der Digitaluhr ergeben sich folgende Zustände:

- Zustand Normalzeit:
In diesem Zustand befindet sich die Uhr nach Einlegen der Batterie (Startsignal).
- Zustand Stunden stellen.
- Zustand Minuten stellen.
- Zustand Sekunden stellen.

Folgende Ereignisse können eintreten:

- Startsignal:
Tritt auf, wenn die Batterie eingelegt wird.
- Knopf 1 gedrückt.
- Knopf 2 gedrückt.

Es wird davon ausgegangen, dass nicht beide Knöpfe gleichzeitig gedrückt werden können. Folgende Ausgaben können auftreten:

- Stunden blinken
Um anzuzeigen, dass der Bediener jetzt in dem Zustand ist, in dem er die Stunden verstellen kann.
- Minuten blinken
- Sekunden blinken
- Stunden erhöhen
Um 1 erhöhte Anzeige der Stunde
- Minuten erhöhen
- Sekunden stellen
Anzeige von 00 als Sekundenanzeige
- Initialisierung
Anzeige von 00:00:00.

Beispiel 1b



Im nächsten Schritt muss festgelegt werden, wie die **Übergänge** – auch Transitionen genannt – zwischen den Zuständen in Abhängigkeit von Eingaben oder Ereignissen aussehen sollen und welche Ausgaben erzeugt oder welche Aktionen ausgelöst werden sollen.

2. Schritt:
Übergänge
festlegen

- Wenn das Startsignal auftritt, dann soll das System in den Zustand Normalzeit übergehen und die Aktion Initialisierung durchführen.
- Wenn der Knopf 1 gedrückt wird und das System im Zustand Normalzeit ist, dann soll als Aktion Stunden blinken ausgeführt und der Zustand Stunden stellen eingenommen werden.

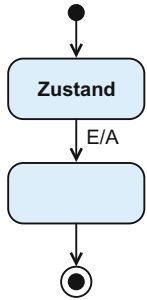
Beispiel 1c

In dieser Form überlegt man sich die Übergänge.

III 10 Dynamik

10.3.2 Notationen

3. Schritt:
Zustands-
automat zeichnen
oder als Tabelle
beschreiben



Ein Zustandsautomat kann als **Zustandsdiagramm** (*statechart diagram*) gezeichnet und als **Zustandstabelle** oder **Zustandsmatrix** dargestellt werden. Für Zustandsdiagramme gibt es verschiedene grafische Notationen. Im Folgenden wird die UML-Notation verwendet.

In der UML werden Zustände als Rechtecke mit abgerundeten Ecken und die **Zustandsübergänge** – auch Transitionen genannt – als beschriftete Pfeile dargestellt. Die Beschriftung besteht aus zwei Teilen. Der erste Teil gibt die Eingabe bzw. das Ereignis an, das den Zustandsübergang bewirkt. Der zweite Teil – durch einen Schrägstrich »/« vom ersten getrennt – gibt das Ergebnis bzw. die durchzuführende **Aktion** an, das während des Übergangs auszugeben bzw. die durchzuführen ist.

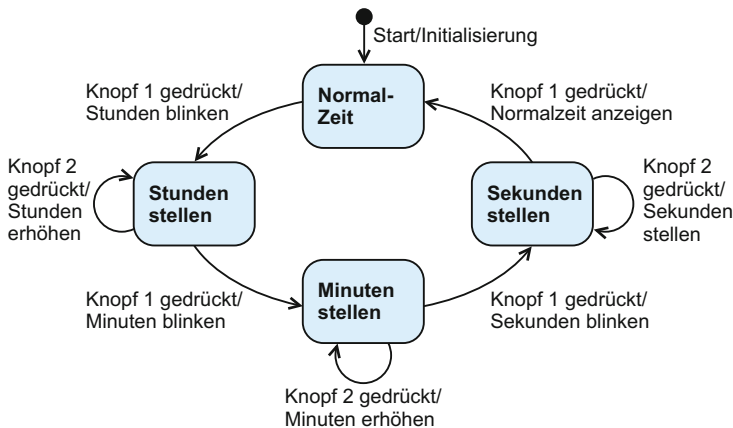
Der Anfangs- oder Startzustand (*initial state*) wird durch einen kleinen schwarzen Kreis dargestellt. Es handelt sich um einen Pseudozustand, der mit einem »echten« Zustand durch eine Transition verbunden ist. Der Endzustand (*final state*) wird durch ein »Bullauge« dargestellt und kann optional beschriftet werden.

Beispiel:
Digitaluhr

Es ergibt sich das Zustandsdiagramm der Abb. 10.3-1. Die Dynamik dieses Diagramms ist folgendermaßen zu interpretieren:

Wenn das Signal Start eintrifft (hier beim Einlegen der Batterie) wird die Initialisierung ausgeführt (Null anzeigen) und das System geht in den Zustand Normalzeit.

Abb. 10.3-1:
Zustands-
diagramm
Uhr stellen.



Wird jetzt der Knopf 1 gedrückt, dann blinken die Stunden und das System geht in den Zustand Stunden stellen.

Befindet sich das System im Zustand Normalzeit und es wird der Knopf 2 gedrückt, dann passiert nichts, da es von diesem Zustand aus keinen Pfeil gibt, der Knopf 2 gedrückt als Ereignis enthält.

Ist das System im Zustand *Stunden stellen* und wird Knopf 2 gedrückt, dann werden die Stunden erhöht und das System geht wieder in den alten Zustand *Stunden stellen* über. Befindet sich das System jedoch im Zustand *Minuten stellen* und es wird Knopf 2 gedrückt, dann werden die Minuten erhöht.

Das gleiche Ereignis kann also unterschiedliche Ergebnisse erzeugen bzw. Aktionen auslösen, in Abhängigkeit von dem Zustand, in dem es sich befindet. Ein bestimmter Zustand wird oft erst nach dem Durchlaufen anderer Zustände erreicht. Insofern kennzeichnet ein aktueller Zustand eines Systems auch die bis dahin durchlaufene Historie. Einen ausgezeichneten Endzustand gibt es bei diesem Beispiel nicht, da nach dem Erreichen des Zustands *Normalzeit* der Zyklus erneut durchlaufen werden kann.

Bei einer Zustandstabelle werden die Zusammenhänge tabellarisch dargestellt. Dies ist dann erforderlich, wenn eine zeichnerische Darstellung zu unübersichtlich wird. Wenn möglich, sollte man auf Tabellen- oder Matrizen-Darstellungen aber verzichten. Die Abb. 10.3-2 und die Abb. 10.3-3 zeigen die Tabellen- und Matrix-Darstellung des Beispiels. Eine alternative Matrixdarstellung zeigt die Abb. 10.3-4. Die Tabellen- und Matrixdarstellungen können – wenn nötig – auf mehrere Seiten verteilt werden.

Aktueller Zustand	Ereignis	Aktion	Folgezustand
	Start	Initialisierung	Normalzeit
Normalzeit	Knopf 1 gedrückt	Stunden blinken	Stunden stellen
Stunden stellen	Knopf 1 gedrückt	Minuten blinken	Minuten stellen
	Knopf 2 gedrückt	Stunden erhöhen	Stunden stellen
Minuten stellen	Knopf 1 gedrückt	Sekunden blinken	Sekunden stellen
	Knopf 2 gedrückt	Minuten erhöhen	Minuten stellen
Sekunden stellen	Knopf 1 gedrückt	Normalzeit anzeigen	Normalzeit
	Knopf 2 gedrückt	Sekunden stellen	Sekunden stellen

Abb. 10.3-2:
Zustandstabelle
von Uhr stellen.

Der Vorteil der ersten Matrixdarstellung liegt darin, dass man visuell schnell erfasst, was geschieht, wenn ein bestimmtes Ereignis eintritt. Da außerdem alle Kombinationen zwischen Zuständen und Ereignissen aufgeführt sind, erlaubt dies eine Vollständigkeitsprüfung.

Der Nachteil dieser Darstellung liegt darin, dass die Matrix oft dünn besetzt ist und daher mehr Platz benötigt als die Zustandstabelle.

III 10 Dynamik

Zustand ↓ Ereignis →	Start	Knopf 1 gedrückt	Knopf 2 gedrückt
	Initialisierung		
	Normalzeit		
Normalzeit		Stunden blinken	
		Minuten blinken	
Stunden stellen		Minuten blinken	Stunden erhöhen
		Minuten stellen	Stunden stellen
Minuten stellen		Sekunden blinken	Minuten erhöhen
		Sekunden stellen	Minuten stellen
Sekunden stellen		Normalzeit anzeigen	Sekunden stellen
		Normalzeit	Sekunden stellen

Legende:
Aktion
 Folgezustand

Abb. 10.3-3: Zustandsmatrix von Uhr stellen. Die zweite Matrixdarstellung zeigt alle Kombinationen zwischen den Zuständen. Sie erlaubt daher eine andere Vollständigkeitsüberprüfung als die erste Darstellungsform. Allerdings ist sie oft auch nur schwach besetzt.

zu Zustand → von Zustand ↓	Normalzeit	Stunden stellen	Minuten stellen	Sekunden stellen
(Start)	Start			
	Initialisierung			
Normalzeit		Knopf 1 gedrückt		
		Stunden blinken		
Stunden stellen		Knopf 2 gedrückt	Knopf 1 gedrückt	
		Stunden erhöhen	Minuten blinken	
Minuten stellen			Knopf 2 gedrückt	Knopf 1 gedrückt
			Minuten stellen	Minuten stellen
Sekunden stellen	Knopf 1 gedrückt			Knopf 2 gedrückt
	Normalzeit anzeigen			Sekunden stellen

Legende:
Ereignis
 Aktion

Abb. 10.3-4:
 Alternative
 Zustandsmatrix
 von Uhr stellen.

10.3.3 Zustandsautomat mit Endzuständen

Einen Zustandsautomaten mit Endzuständen zeigt folgendes Beispiel.

Beispiel Ein Teach-Roboter (Abb. 10.3-5) soll vom Benutzer mit folgenden Kommandos gesteuert werden:

BSA = Beugen Schulter Absolut (Ausgabe: 1)

BSR = Beugen Schulter Relativ (Ausgabe: 2)

RSA = Rotieren Schulter Absolut (Ausgabe: 3)

RSR = Rotieren Schulter Relativ (Ausgabe: 4)

An die Roboteranimation soll je nach erkanntem Kommando die entsprechende Kommandonummer übergeben werden (1, 2, 3 oder 4). Im Anschluss an das Kommando kommt optional ein + oder – Zeichen gefolgt von einer dreistelligen Zahl. Die Abb. 10.3-6 zeigt das

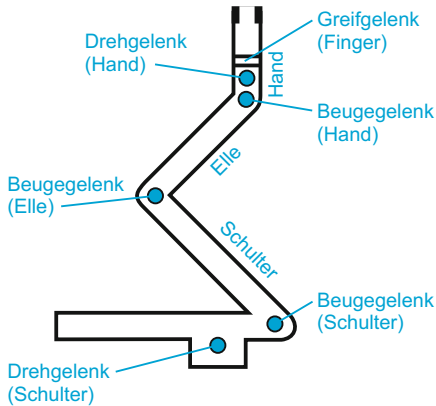


Abb. 10.3-5:
Teach-Roboter:
Seitenansicht
(physikalische
Skizze).

entsprechende Zustandsdiagramm. Dieser Zustandsautomat besitzt drei Endzustände, davon zwei Fehlerzustände. Nicht zu jeder Eingabe gehört auch eine Ausgabe.

10.3.4 Mealy-Automat vs. Moore-Automat

Die bisher beschriebenen Zustandsautomaten sind sogenannte **Mealy-Automaten**. Charakteristisch für einen Mealy-Automaten ist, dass die Ausgaben bzw. Aktionen an die Zustandsübergänge gekoppelt sind. Dies entspricht der Vorstellung, die man sich bezüglich der Zeitdauer von Übergängen, Zuständen und Ausgaben macht. Zustände repräsentieren Zeitperioden bzw. Zeitintervalle. Übergänge stellen Zeitpunkte dar, an denen sich die Verhaltensweise ändert. Ausgaben treten zu diskreten Zeitpunkten auf und können deshalb gut mit Übergängen assoziiert werden.

Mealy-Automat

Im Gegensatz zum Mealy-Automaten sind beim **Moore-Automaten** die Ausgaben bzw. Aktionen an die Zustände gebunden. In der UML spricht man dann von **Aktivitäten**. Eine Aktivität beginnt, wenn in den Zustand eingetreten wird und endet, wenn der Zustand verlassen wird. In einem Zustandsdiagramm werden die Aktivitäten unterhalb der Zustandsnamen – getrennt durch eine Linie – eingetragen. Vor den Aktivitätsnamen wird *do/* geschrieben (Abb. 10.3-7).

Moore-Automat

Mealy- und Moore-Automaten sind äquivalent und können jeweils ineinander überführt werden (siehe [HoUl88, S. 55 f.]). Voraussetzung für einen Moore-Automaten ist, dass in dem jeweiligen Zustand *genau eine* Aktivität durchgeführt wird. Der Zustandsautomat der Abb. 10.3-8 ist *kein* geeigneter Moore-Automat, da die durchzuführende Aktion davon abhängt, von welchem Zustand in einen anderen Zustand gewechselt wird.

Äquivalenz

III 10 Dynamik

Abb. 10.3-6:
Zustands-
diagramm von
Kommando erkennen.

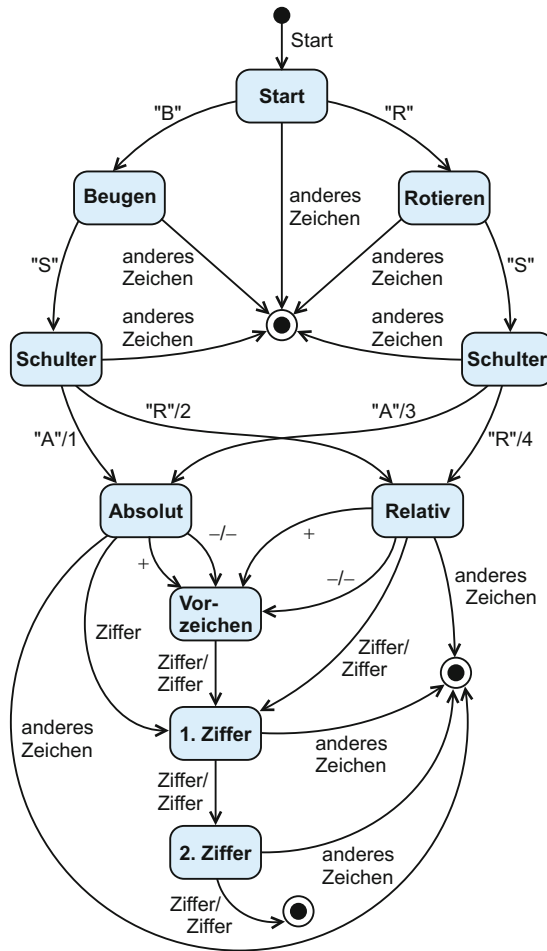
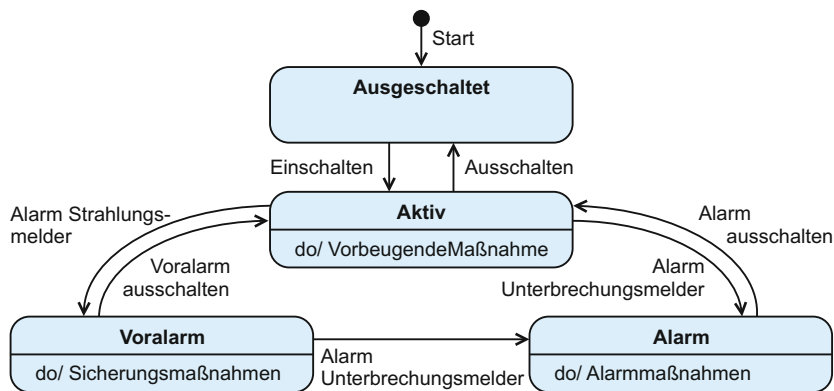


Abb. 10.3-7:
Moore-Automat
für eine
Alarmanlage.



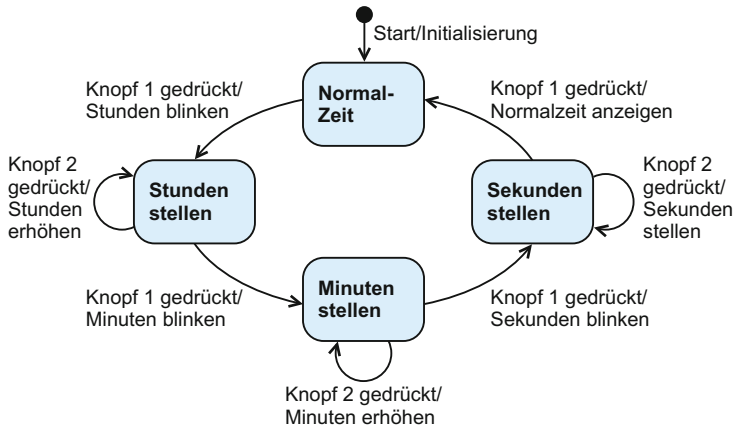


Abb. 10.3-8: Zustandsdiagramm Uhr stellen.

Eine Alarmanlage soll entsprechend dem Moore-Automaten der Abb. 10.3-7 gesteuert werden. Für diese Problemstellung stellt der Moore-Automat die adäquate Beschreibungsform dar, da die Maßnahmen (Vorbeugung, Sicherung, Alarm) so lange aktiv sein sollen, wie der Zustand beibehalten wird.

Beispiel

Ein Moore-Automat lässt sich in einen äquivalenten Mealy-Automaten transformieren, indem an jeden Zustandsübergang diejenige Ausgabe gekoppelt wird, die im Moore-Automat von dem Zustand generiert wird, der nach dem Zustandsübergang angenommen wird.

Transformation

Bei den Mealy- und Moore-Automaten handelt es sich um deterministische Automaten. Von einem Zustand aus gibt es zu einer Eingabe höchstens einen Zustandsübergang.

Deterministischer Automat

Nichtdeterministische Automaten (Abb. 10.3-9) erlauben demgegenüber für einen Zustand bei derselben Eingabe null, einen oder mehr Zustandsübergänge. Nichtdeterministisches Verhalten wird in der Softwaretechnik in der Regel durch Petrinetze modelliert (siehe »Petrinetze«, S. 303).

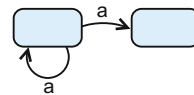


Abb. 10.3-9: Nicht-deterministischer Automat.

10.3.5 Zustandsautomat nach Harel

Zur Modellierung komplexer Zusammenhänge reichen die bisher behandelten Konzepte nicht aus. David Harel erweiterte 1987 mit seinen *statecharts* [Hare87] die Zustandsautomaten um folgende Konzepte:

- Hybride Zustandsautomaten
- Bedingte Zustandsübergänge
- Hierarchische Zustandsautomaten
- Zustände mit Gedächtnis
- Nebenläufige Zustände



Prof. Dr. D. Harel

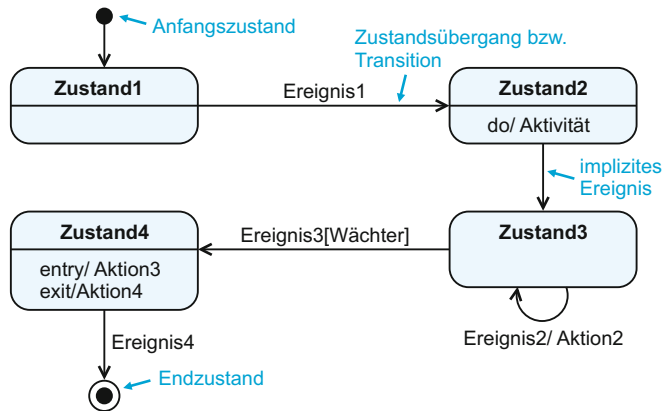
III 10 Dynamik

Diese Konzepte werden im Folgenden näher betrachtet. Automaten mit diesen Konzepten bezeichnet man als **Harel-Automat**.

Hybride Zustandsautomaten

Harel erlaubt es, den Mealy- und den Moore-Automaten zu kombinieren. Man spricht daher auch von hybriden Zustandsautomaten. Die UML-Notation zeigt die Abb. 10.3-10.

Abb. 10.3-10: UML-Notation für Zustandsautomaten.



Ein Zustandsübergang wird durch ein Ereignis ausgelöst. Die Bezeichnung für das auslösende Ereignis wird an den Zustandsübergang angetragen.

Bedingte
Zustands-
übergänge

Optional kann ein Ereignis mit einem **Wächter** (*guard condition*) – eingeschlossen in eckige Klammern – kombiniert werden. Ein Zustandsübergang findet nur statt, wenn zu dem Zeitpunkt, zu dem das Ereignis eintritt, auch die entsprechende Bedingung gültig ist (*guarded transition*).

Beispiel Die Abb. 10.3-11 beschreibt einen einfachen Kartenautomaten, der in einem Parkhaus die Parkkarten abrechnet. Tritt im Zustand wartet auf Geld das Ereignis Geld eingeworfen ein, so hängt die Reaktion von der zusätzlichen Bedingung [reicht nicht] oder [reicht aus] ab.

Aktionen Mit einem Zustandsübergang kann eine Aktion verbunden sein. Eine Aktion benötigt den Bruchteil einer Sekunde. Sie wird sofort ausgeführt und benötigt im Idealfall keine Zeit.

Aktivitäten Im Gegensatz zu Aktionen benötigen Aktivitäten eine bestimmte Zeitdauer, wie die Ausführung einer längeren Berechnung, das Anzeigen von Informationen oder das Aussenden eines Pieptons. Ein Beispiel für diese Situation ist die Modellierung der Alarmanlage (siehe »Mealy-Automat vs. Moore-Automat«, S. 275).

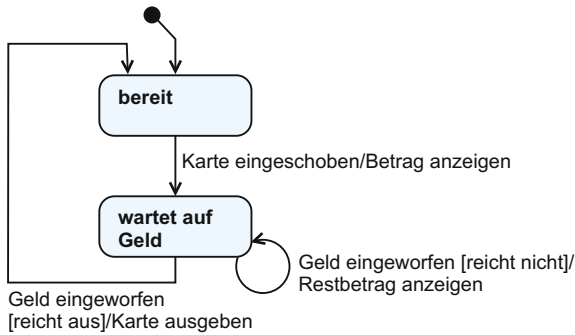


Abb. 10.3-11:
Kartenautomat im
Parkhaus.

Mit einem Zustand können auch Aktionen verbunden sein (Abb. 10.3-12). Sie können beim Eintritt in einen Zustand (*entry action*) und beim Verlassen eines Zustands (*exit action*) ausgelöst werden. Die linke und die rechte Beschreibungsform in der Abb. 10.3-12 sind identisch.

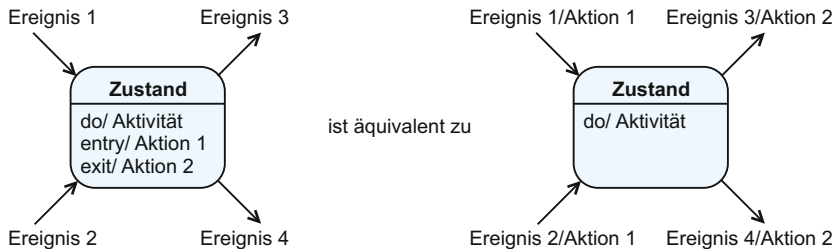


Abb. 10.3-12:
Aktionen in
Zuständen.

Eine *entry action* sagt das Gleiche aus, als wenn jeder Übergang in diesem Zustand die entsprechende Aktion enthalten würde. Eine *exit action* bedeutet, dass diese Aktion bei jedem Zustandsübergang, der den Zustand verlässt, ausgeführt würde. Während Aktivitäten durch Ereignisse vorzeitig beendet werden können, werden Aktionen immer ausgeführt.

Aktionen in
Zuständen

Durch die Möglichkeit, in einem Zustandsdiagramm Aktionen und Aktivitäten gleichzeitig zu notieren, können die Konzepte des Mealy- und des Moore-Automaten kombiniert benutzt werden.

Hierarchische Zustandsautomaten

Zustandsautomaten können sehr umfangreich und unübersichtlich werden. Zustände können daher geschachtelt, d.h. durch Unterzustände (*substates*) verfeinert werden, sodass **hierarchische Zustandsautomaten** entstehen. Alle Unterzustände sind disjunkt, d.h. sie schließen sich gegenseitig aus. Ein Zustand, der verfeinert wird, heißt **zusammengesetzter Zustand**.

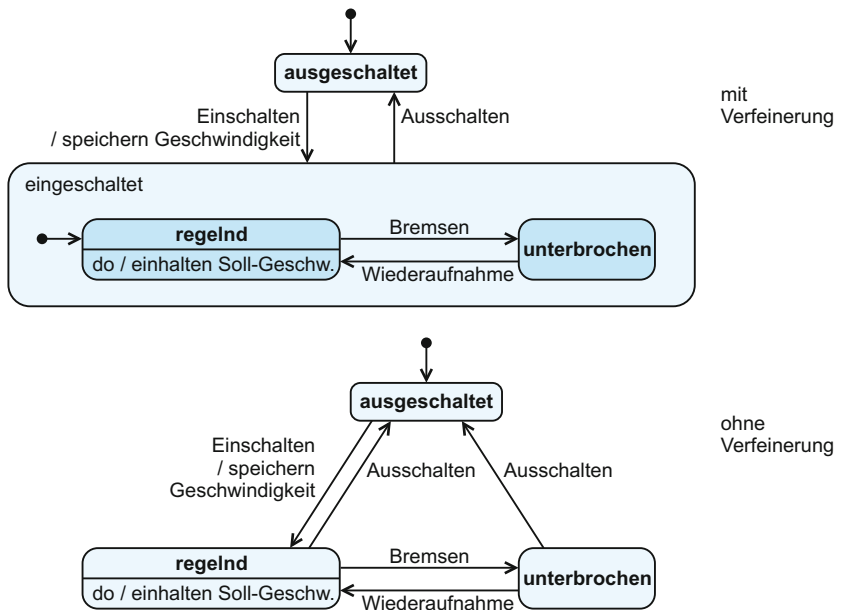
III 10 Dynamik

Auch jede Verfeinerung besitzt genau einen Anfangszustand. Ein Übergang in einen verfeinerten Zustand entspricht dem Übergang in den Anfangszustand der Verfeinerung. Das Verlassen eines verfeinerten Zustands wird im entsprechenden Zustandsdiagramm durch den Endzustand angezeigt.

Wird ein verfeinerter Zustand durch einen Übergang verlassen, dann wird jeder Unterzustand – egal auf welcher Verfeinerungsstufe – verlassen und die entsprechende exit-Aktivität ausgeführt. Wird ein Zustand mit einem rekursiven Übergang verfeinert, dann wird beim erneuten Zustandseintritt der Anfangszustand eingenommen und die entry-Aktivität ausgeführt.

Beispiel Die Abb. 10.3-13 wendet die Verfeinerung auf einem Tempomaten an. Beide Zustandsdiagramme modellieren die gleiche Problemstellung.

Abb. 10.3-13:
Zustandsautomat
eines
Tempomaten.



Pseudozustände

Für die Modellierung der erweiterten Möglichkeiten gibt es in der UML weitere **Pseudozustände**. Sie dienen dazu, eine bestimmte Ablauflogik in den Zustandsautomaten einzufügen. Übergänge verbinden nicht ausschließlich Zustände, sondern können auch Pseudozustände einbeziehen. Die Abb. 10.3-14 zeigt die im Folgenden verwendeten Pseudozustände.

- Anfangszustand (*initial pseudo state*)
- ◇ Entscheidung (*choice*)
- Kreuzung (*junction*)
- × Terminator (*terminate pseudo state*)
- ⊕ flache Historie (*shallow history*)
- ⊕^o tiefe Historie (*deep history*)
- Eintrittspunkt (*entry point*)
- ⊗ Austrittspunkt (*exit point*)

Abb. 10.3-14:
Pseudozustände.

Der Anfangszustand (*initial state*) (auch Startzustand genannt) kennzeichnet den Startpunkt für die Ausführung des Zustandsautomaten. Er wird durch einen kleinen ausgefüllten Kreis dargestellt. Vom Anfangszustand erfolgt genau ein Übergang in den ersten »echten« Zustand. Im Allgemeinen wird dieser Übergang *nicht* mit einem Ereignis beschriftet.

Ein Entscheidungszustand (*choice*) ermöglicht die Modellierung von Alternativen, die von dem Ergebnis einer zuvor ausgeführten Aktivität abhängen. Die Abb. 10.3-15 zeigt verschiedene Möglichkeiten, um einen Entscheidungszustand darzustellen. Da die jeweils gewählte Alternative immer aktuell berechnet wird, spricht man hier von einer dynamischen Verzweigung (*dynamic conditional branch*).

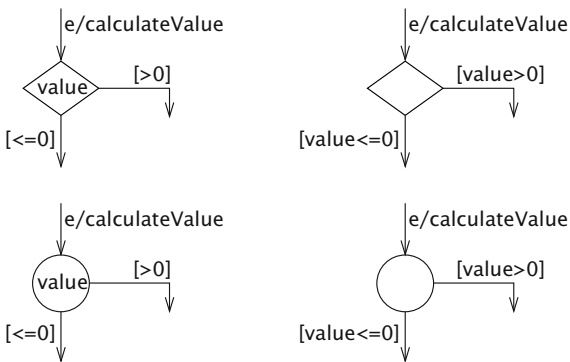


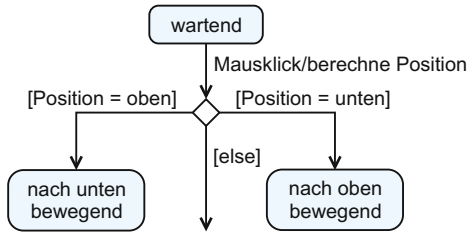
Abb. 10.3-15:
Notationsalternativen für einen Entscheidungszustand.

In der Abb. 10.3-16 reagiert der Zustandsautomat im Zustand wartend auf das Ereignis Taste und führt die Aktivität berechne Position aus. In Abhängigkeit von diesem Ergebnis erfolgt dann eine Verzweigung in den Zustand nach unten bewegend oder nach oben bewegend. Führt die errechnete Position zu keiner dieser Bedingungen, dann wird die else-Alternative ausgeführt.

Beispiel

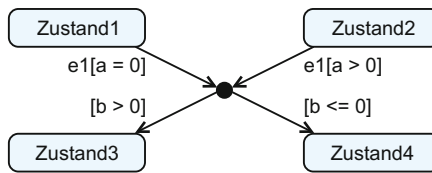
III 10 Dynamik

Abb. 10.3-16:
Entscheidungs-
zustand zur
Modellierung
dynamischer
Verzweigungen.



Die Kreuzung (*junction*) wurde eingeführt, um mehrere aufeinander folgende Übergänge ohne dazwischen liegende Zustände zu verknüpfen. Jede Kreuzung muss mindestens einen eingehenden und einen ausgehenden Übergang besitzen. Im Gegensatz zu dem Entscheidungszustand handelt es sich um eine statische Verzweigung (*static conditional branch*). In der Abb. 10.3-17 wird der Zustand1 verlassen, wenn das Ereignis e1 eintrifft und die Bedingung (*guard*) $[a=0]$ erfüllt ist. Dann wird geprüft, ob die Bedingung $[b>0]$ gilt. Falls ja, erfolgt ein Übergang in den Zustand3, andernfalls in den Zustand4. Analoges gilt für den Zustand2.

Abb. 10.3-17:
Kreuzung zur
Verknüpfung von
Übergängen.



Beim Erreichen eines Terminators (*terminate pseudo state*) endet die Verarbeitung des Zustandsautomaten. Im Gegensatz zum Endzustand (Bullauge) gilt in der UML zusätzlich, dass die Lebensdauer des beschriebenen Modellelements beendet wird. Diesen Pseudozustand sollten Sie verwenden, wenn Sie modellieren wollen, dass Objekte dynamisch vernichtet werden.

Beim Endzustand, der durch einen kleinen gefüllten Kreis mit einem Ring dargestellt wird (Bullauge), handelt es sich in der UML *nicht* um einen Pseudozustand, sondern um einen echten Zustand.

Beispiel In der Abb. 10.3-18 wird der Zugriff auf eine Datenbank beschrieben. Im Zustand *konnectierend* wird versucht, eine Verbindung zur Datenbank aufzubauen. Gelingt dies nicht, dann erfolgt ein Übergang zum Terminator und das zugehörige Objekt wird wieder gelöscht.

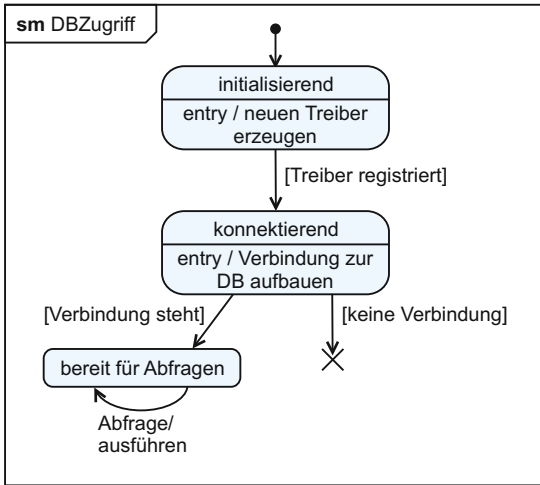


Abb. 10.3-18:
Übergang bei
erfolglosem Ver-
bindungsaufbau.

Untenzustandsautomaten

Hierarchische Zustandsautomaten können auch durch eigenständige **Untenzustandsautomaten** (*submachine*) verfeinert werden. Im übergeordneten Zustandsdiagramm wird ein Untenzustandsautomatenzustand (*submachine state*) eingetragen. Dieser Zustand kann beliebig viele Ein- und Austrittspunkte besitzen.

Ein Eintrittspunkt (*entry point*) kennzeichnet den Eintritt in einen Zustandsautomaten und wird meistens bei geschachtelten Zustandsdiagrammen verwendet. Er wird durch einen kleinen, nicht ausgefüllten Kreis dargestellt und oft auf dem Rahmen des entsprechenden Automaten angetragen.

Der Austrittspunkt (*exit point*) bildet das Gegenstück zum Eintrittspunkt. Er wird durch einen kleinen, nicht ausgefüllten Kreis mit einem »x« dargestellt und im Allgemeinen ebenfalls auf dem Rahmen des entsprechenden Automaten angetragen.

Ein- und Austrittspunkte werden jeweils benannt, wobei die Namen im Untenzustandsautomatenzustand und im verfeinernden Automaten übereinstimmen müssen.

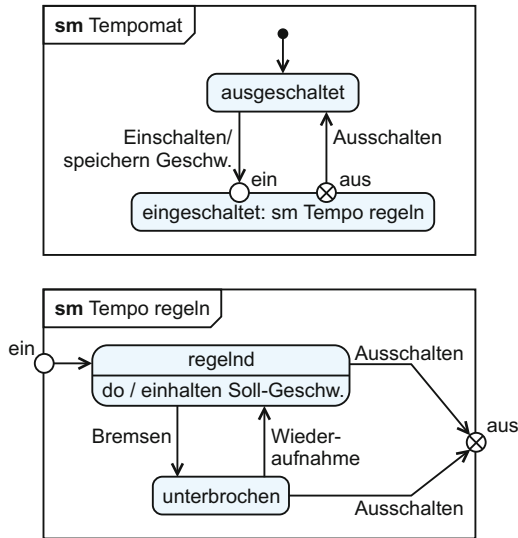
Ein Untenzustandsautomatenzustand ist semantisch äquivalent zu einem zusammengesetzten Zustand. Die Bildung von Untenzustandsautomaten erlaubt jedoch darüber hinaus, häufig benötigtes Verhalten in einem separaten Zustandsautomaten zu spezifizieren und beliebig oft zu referenzieren.

In der Abb. 10.3-19 reagiert der Tempomat im Zustand *ausgeschaltet* auf das Ereignis *Einschalten* und geht über den Eintrittspunkt ein in den Untenzustandsautomatenzustand über. Das bedeutet, dass er im Untenzustandsautomaten den Zustand *regelnd* einnimmt. Durch das Ereignis *Ausschalten* werden bei diesem Beispiel alle Unterzustände über den Austrittspunkt aus verlassen.

Beispiel 1a

III 10 Dynamik

Abb. 10.3-19:
Untenzustands-
automat mit
Eintritts- und
Austrittspunkten.



Ein Untenzustandsautomat kann auch durch seinen Anfangszustand betreten werden und benötigt dann keinen Eintrittspunkt. Ein Austrittspunkt ist in folgenden Fällen *nicht* notwendig: Der Automat wird verlassen, weil die Verarbeitung beendet ist, oder es erfolgt ein Austritt durch einen Gruppenübergang. Das ist ein Übergang, der von der Grenze des Untenzustandsautomatenzustands ausgeht und über die *alle* darin enthaltenen Zustände verlassen werden.

Beispiel 1b Die Abb. 10.3-20 zeigt den Tempomaten in der alternativen Notation. Durch das Ereignis Ausschalten werden die Zustände *regelnd* und *unterbrochen* des Untenzustandsautomaten verlassen.

Man sieht in diesem Beispiel aber deutlich, dass eine hierarchische Anordnung z. B. dann sinnvoll ist, wenn mehrere Zustände durch das gleiche Ereignis verlassen werden.

UML-Regeln In der UML gelten für hierarchische Zustandsautomaten folgende Regeln:

- Auch jede Verfeinerung besitzt genau einen Anfangszustand.
- Ein Zustandsübergang in einen verfeinerten Zustand entspricht dem Zustandsübergang in den Anfangszustand der Verfeinerung. Das Verlassen eines verfeinerten Zustands wird im entsprechenden Zustandsdiagramm durch den Endzustand angezeigt.
- Wird ein verfeinerter Zustand durch einen Zustandsübergang verlassen, dann wird jeder Unterzustand – egal auf welcher Verfeinerungsstufe – verlassen und die entsprechenden *exit*-Aktionen ausgeführt. Wird ein Zustand mit einem rekursiven Zustandsübergang verfeinert, dann wird beim erneuten Zustandseintritt der Anfangszustand eingenommen und die *entry*-Aktion ausgeführt.

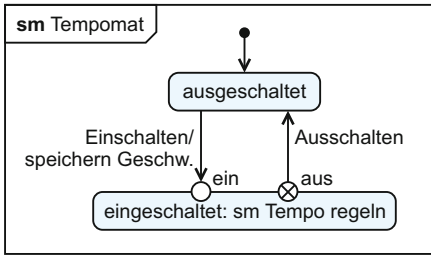
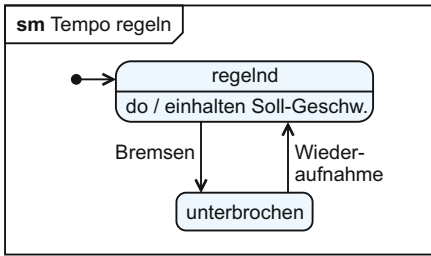


Abb. 10.3-20:
Unterzustands-
automat mit
Anfangszustand
und
Gruppentransition
für den Austritt.



Zustände mit Gedächtnis

Mithilfe von Historienzuständen kann sich ein Zustandsautomat »merken«, welcher Unterzustand zuletzt eingenommen wurde. Die UML unterscheidet zwischen einer flachen und einer tiefen **Historie**. Die flache Historie (*shallow history*) wird durch einen Kreis mit einem »H« dargestellt, die tiefe Historie (*deep history*) durch einen Kreis mit der Beschriftung »H*«. Flache und tiefe Historie können sowohl bei zusammengesetzten Zuständen als auch bei Unterzustandsautomaten verwendet werden.

Die Abb. 10.3-21 modelliert eine flache Historie. Der Automat startet im Zustand Z1 bzw. in dem darin enthaltenen Zustand Z2. Durch das Ereignis e1 gelangt er in den Zustand Z3 und von dort erfolgt durch das Ereignis e3 ein Übergang in Z4 (Gruppenübergang). In Z4 löst das Ereignis e4 einen Übergang in den Historienzustand aus. Da Z3 der zuletzt eingenommene Unterzustand von Z1 war, wird dies als Übergang in den Zustand Z3 interpretiert.

Flache Historie

Ein analoges Verhalten tritt auf, wenn Z2 der zuletzt eingenommene Zustand war. Dann wird der Übergang von Z4 in den Historienzustand als Übergang in Z2 interpretiert.

Die Abb. 10.3-22 zeigt zum Vergleich eine tiefe Historie. Wenn sich der Automat im Zustand Z5 befindet, erfolgt durch das Ereignis e4 ein Übergang in den Zustand Z6. Von Z6 führt das Ereignis e5 in den tiefen Historienzustand. Das heißt, dass der Zustand Z5 wieder eingenommen wird.

Tiefe Historie

III 10 Dynamik

Abb. 10.3-21:
Flache Historie.

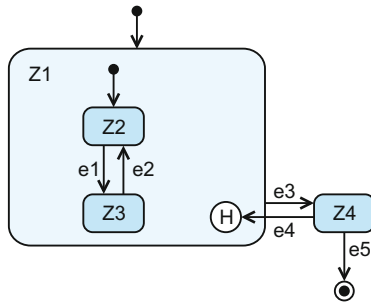
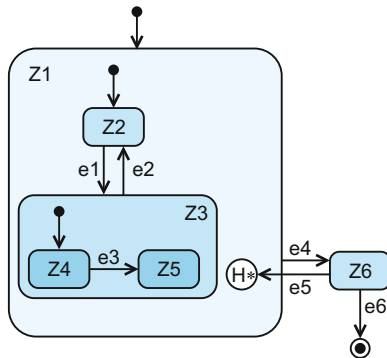


Abb. 10.3-22:
Tiefe Historie.



Der Unterschied zwischen flacher und tiefer Historie wird deutlich, wenn Sie sich vorstellen, dass die Abb. 10.3-22 nur eine flache Historie enthielte. In diesem Fall würde in der gleichen Situation nur ein Übergang in den Zustand Z3 erfolgen und dort der Zustand Z4 (Folgebzustand des Anfangszustands) eingenommen.

Nebenläufige Zustände

Ein Oberzustand kann aus Unterzuständen bestehen, in denen sich das System gleichzeitig befindet. Um diese Situation darzustellen, wird ein Oberzustand durch gestrichelte Linien in mehrere nebenläufige Regionen (*regions*) aufgeteilt.

Die Abb. 10.3-23 [Hare87, S. 242] zeigt den Oberzustand Y mit seinen beiden Komponenten A und D. Wird in den Oberzustand Y eingetreten, dann hat dies zur Folge, dass sich das System im Zustand B und gleichzeitig im Zustand F befindet.

Tritt das Ereignis a ein, dann erfolgen simultane Zustandsübergänge von B nach C und von F nach G. Befindet sich das System im kombinierten Zustand (B,F) und das Ereignis b tritt ein, dann hat dies nur eine Auswirkung auf die D-Komponente. Es ergibt sich der kombinierte Zustand (B,E).

Der Zustandsübergang c in A hat die Bedingung »in G«, d. h., nur wenn sich die Komponente D im Zustand G befindet, wird der Zustandsübergang c ausgeführt.

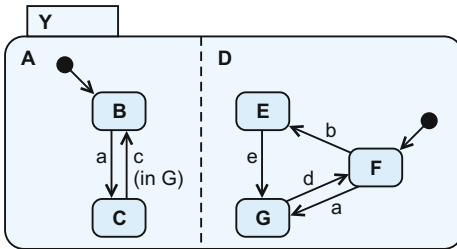


Abb. 10.3-23:
Nebenläufige
Zustände.

Das Beispiel »Digitaluhr« (siehe »Erstellung eines Zustandsautomaten«, S. 270) soll erweitert werden. Die Digitaluhr kann alternativ zur Uhrzeit das Datum in der Form Wochentag, Tag, Monat anzeigen. Ein Druckknopf 3 erlaubt das Umschalten von Datum auf Uhrzeit und umgekehrt. Das Datum kann analog wie die Uhrzeit mit den Knöpfen 1 und 2 gestellt werden. Nach dem Einlegen der Batterie werden Datum und Uhrzeit angezeigt (Abb. 10.3-24), die Uhrzeit kann eingestellt werden. In der Region Stellen wird durch Drücken von Knopf 3 festgelegt, ob die Uhrzeit oder das Datum gestellt werden kann.

Beispiel:
Digitaluhr



10.3.6 Verhaltens- vs. Protokollzustandsautomaten

Zustandsautomaten eignen sich gut dazu, das Verhalten von Elementen, z. B. von Objekten oder Interaktionen, zu beschreiben. In der UML werden zwei Arten von Zustandsautomaten unterschieden:

- Der Verhaltenszustandsautomat (*behavioral state machines*) und
- der Protokollzustandsautomat (*protocol state machines*).

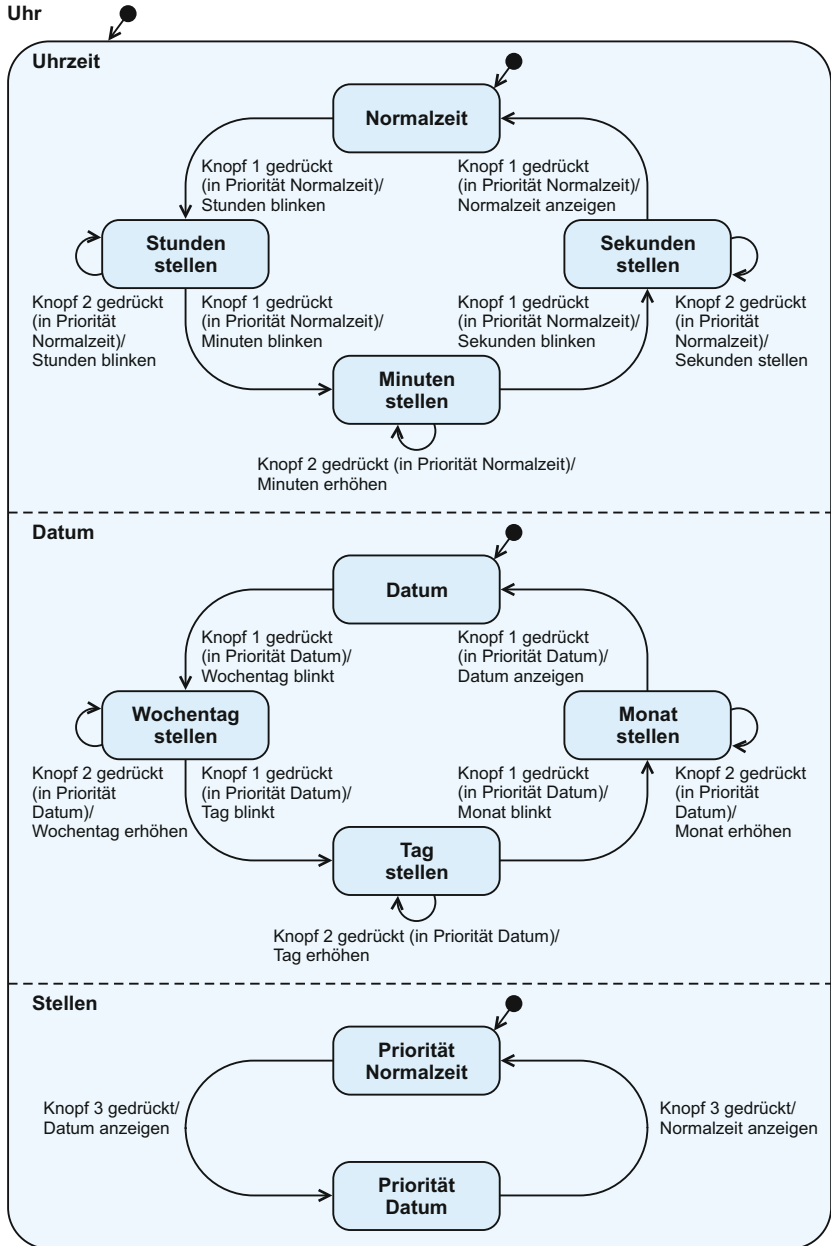
Der Verhaltenszustandsautomat ermöglicht es, das dynamische Verhalten von verschiedenen Modellelementen, z. B. Objekten, zu modellieren. Er ist eine objektorientierte Variante des Harel-Zustandsautomaten.

Der Protokollzustandsautomat dient zur Spezifikation von Benutzungsprotokollen, z. B. zur Modellierung des Lebenszyklus eines Objekts oder die Aufrufreihenfolge seiner Operationen. *Classifier*, Schnittstellen und *Ports* können einem solchen Zustandsautomat zugeordnet werden.

In der UML können Diagramme mit einem optionalen Rahmen ergänzt werden. Jeder Rahmen besitzt links oben ein Fünfeck, in das der Diagrammtyp und der Name des Diagramms eingetragen wird. Für den Diagrammtyp gibt die UML einige Kürzel vor, z. B. *sm* für Zustandsdiagramm (*state machine*).

III 10 Dynamik

Abb. 10.3-24: Uhr
mit nebenläufigen
Zuständen.



Verhaltenszustandsautomaten

Wenn Zustandsautomaten dazu dienen, das Verhalten von Modellelementen zu spezifizieren, dann spricht man von **Verhaltenszustandsautomaten**. Ist ein solcher Automat unabhängig vom Kontext eines UML-Classifier spezifiziert, dann sind die Ereignisse (*trigger*) unabhängig von einem Classifier.

Die Abb. 10.3-25 zeigt, wie die Ausfahrt aus einem Parkhaus funktioniert. Die Schranke benötigt für das Heben und Senken eine bestimmte Zeitspanne. Daher wird hier eine *do*-Aktivität zur Modellierung verwendet. Bei diesem Beispiel beenden sich beide Aktivitäten selbstständig nach einer gewissen Zeit. Es wäre aber auch denkbar, dass während des Senkens der Schranke plötzlich ein weiteres Auto durchfährt. Hier könnte modelliert werden, wie die Schranke in diesem Fall reagieren soll. *Do*-Aktivitäten können durch auftretende Ereignisse abgebrochen werden.

Beispiel

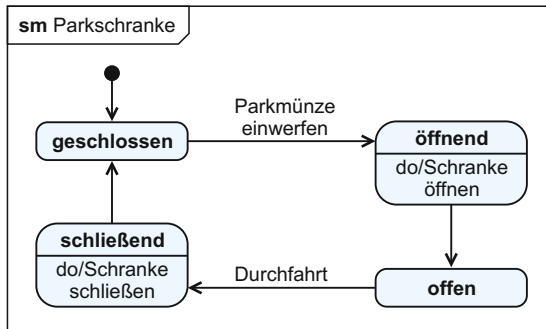


Abb. 10.3-25:
Zustandsautomat
mit *do*-Aktivitäten.

Protokollzustandsautomaten

Am häufigsten werden Zustandsautomaten dazu eingesetzt, um den **Lebenszyklus eines Objekts** zu modellieren. Alle Objekte einer Klasse besitzen dabei *denselben* Zustandsautomaten, jedoch kann jedes Objekt einen individuellen Zustand einnehmen. Im Allgemeinen ist es *nicht* notwendig, für jede Klasse einen Zustandsautomaten aufzustellen, sondern nur für Klassen mit Operationen, die nur ausgeführt werden dürfen, wenn bestimmte Bedingungen erfüllt sind, d. h., wenn sich das Objekt vor der Ausführung der Operation in einem bestimmten Zustand befindet.

Wenn in einer Bibliothek ein Buch beschafft wird, dann werden seine Daten erfasst und ein neues Objekt der Klasse Buch erzeugt (Abb. 10.3-26). Der Einfachheit halber gebe es von jedem Buch nur ein einziges Exemplar. Jedes Buch kann ausgeliehen werden. Wird ein geliehenes Buch von einem anderen Leser gewünscht, dann muss es vor-

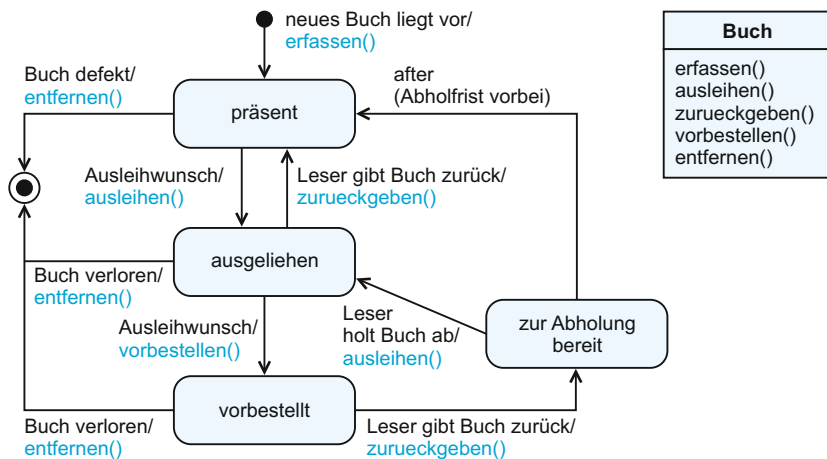
Beispiel

III 10 Dynamik

bestellt werden. Nicht vorbestellte Bücher stehen nach der Rückgabe sofort für eine erneute Ausleihe bereit. Vorbestellte Bücher werden nach der Ausleihe für den entsprechenden Leser zur Abholung bereitgelegt, und der Leser wird informiert. Wird das Buch nicht fristgemäß abgeholt, dann steht es für eine neue Ausleihe bereit. Beschädigte Bücher oder Bücher, die nicht zurückgegeben wurden, werden aus dem Bestand entfernt.

Wenn ein neues Buch im System gespeichert wird, dann befindet es sich zunächst im Zustand präsent. Das Löschen von Büchern im System wird durch den Übergang in den Endzustand (Bullauge) angezeigt. Er sagt aus, dass das Objekt aufhört zu existieren.

Abb. 10.3-26:
Zustandsautomat
der Klasse Buch
[Balz99].



Zustandsname

In der UML-Notation ist der Name des **Zustands** optional. Zustände ohne Namen heißen *anonyme Zustände* und sind alle voneinander verschieden. Ein benannter Zustand kann dagegen – der besseren Lesbarkeit wegen – mehrmals in das Diagramm eingetragen werden. Diese Zustände sind alle identisch. Der Zustandsname soll kein Verb sein, sondern wenn möglich ein Adjektiv, auch wenn mit dem Zustand eine Verarbeitung verbunden sein kann. Im Beispiel wurde daher der Name »ausgeliehen« statt »ausleihen« gewählt. Innerhalb eines Zustandsautomaten muss jeder Zustandsname eindeutig sein.

Zustands-
übergang

Ein Zustandsübergang bzw. eine Transition verbindet zwei Zustände. Er kann *nicht* unterbrochen werden und wird stets durch ein Ereignis ausgelöst. Tritt ein Ereignis ein und das Objekt befindet sich *nicht* in einem Zustand, in dem es darauf reagieren kann, dann wird das Ereignis ignoriert. Meistens ist mit einem Zustandsübergang ein Zustandswechsel verbunden. Es ist aber auch möglich, dass Ausgangs- und Folgezustand identisch sind. In einem solchen Fall werden die *entry*- und *exit*-Aktionen bei jedem neuen Eintritt – in denselben Zustand – ausgeführt.

Ein **Ereignis** kann sein:

- Eine Bedingung, die wahr wird,
- ein Signal,
- eine Botschaft (Aufruf einer Operation),
- eine verstrichene Zeit (*elapsed time event*) oder
- das Eintreten eines bestimmten Zeitpunkts.

In den beiden letzten Fällen handelt es sich um zeitliche Ereignisse.

Eine Bedingung ist beispielsweise: `when (Kontostand < 0)`. Der Zustandsübergang findet statt, wann immer diese Bedingung wahr wird. Signale und Botschaften sind durch die Notation nicht unterscheidbar. Sie werden durch einen Namen beschrieben und können Parameter besitzen, z. B. rechte Maustaste gedrückt (Mausposition). Der Zustandsübergang erfolgt, wenn das Signal oder die Botschaft gesendet wird. Eine verstrichene Zeitspanne ist beispielsweise: `after (10 Sekunden)`. Ist die angegebene Zeitspanne nach einem definierten Zeitpunkt (z. B. Eintritt in den Ausgangszustand des Zustandsübergangs) verstrichen, dann wird der Zustandsübergang durchgeführt. Ein Zeitpunkt wird beispielsweise durch `when (Datum = 4.2.2010)` beschrieben.

Jeder Zustand darf einen – ausgehende – Übergang ohne explizite Angabe eines Ereignisses besitzen. Dieser Übergang wird ausgeführt, wenn die mit dem Zustand verbundene Verarbeitung beendet ist. Es handelt sich um ein implizites Ereignis.

Wird durch einen Zustandsautomaten ein Objektlebenszyklus modelliert, dann müssen das Klassendiagramm und der Zustandsautomat konsistent sein.

Folgende Konsistenzregeln sind einzuhalten, die über die Notation der UML hinausgehen [Balz05]:

- Als Aktionen und Aktivitäten sind nur Operationen der jeweiligen Klasse zulässig. Diese Operationen können dann auch Botschaften an andere Objekte schicken.
- Operationsnamen werden in der Form `Operation()` eingetragen.
- Wenn eine Operation in mehreren Zuständen aktiviert werden kann, so kann sie in Abhängigkeit vom jeweiligen Zustand eine unterschiedliche Wirkung besitzen.
- Erhält ein Objekt in einem Zustand eine Botschaft, wobei die Operation weder als Aktivität noch als Aktion zur Verfügung steht, dann besitzt die Botschaft keine Wirkung, d. h. »das Objekt tut nichts«.

Ereignisse

Notation

Implizites Ereignis

Konsistenz

QS-Checkliste
UML-Protokoll-
zustands-
automat



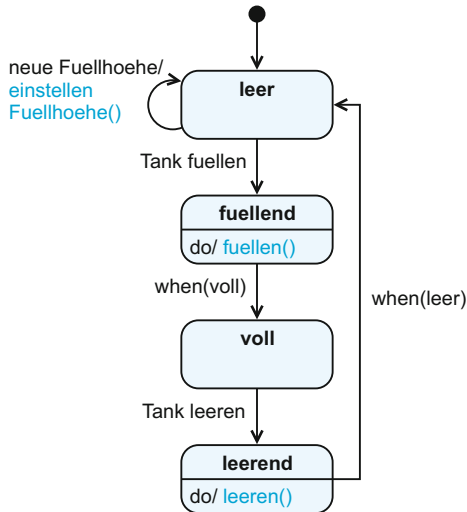
Die Abb. 10.3-27 zeigt den zyklischen Zustandsautomaten (*circular lifecycle*) eines Tanks. Im Zustand `leer` kann die Operation `einstellenFuellhöhe()` aktiviert werden. Weiterhin kann bei leerem Tank die Operation `fuellen()` gestartet werden. Der Zustand `fuellend` wird verlassen, wenn die Bedingung `when(voll)` wahr wird. Die Operation `fuellen()` wird als Aktivität eingetragen, weil sie auch

Beispiel
Lebens-
zyklus

III 10 Dynamik

durch die Aktionen `starte fuellen` und `terminiere fuellen` beschrieben werden könnte. Analog wird bei der Operation `leeren()` verfahren.

Abb. 10.3-27:
Zustandsautomat
eines Tanks.



Tank
fuellen() leeren() einstellenFuellhoehe()

Vererbung von
Objektlebens-
zyklen

Eine Klasse vererbt ihren Objektlebenszyklus an ihre Unterklassen. Unterklassen können darüber hinaus eigene Objektlebenszyklen besitzen. Um Konflikte zu vermeiden, sind einige Restriktionen zu beachten. Eine ausführliche Beschreibung dieser Problematik findet man in [MCDy93].

10.3.7 Markov-Ketten

Markov-Ketten sind spezielle stochastische Prozesse, die zur Modellierung und Analyse komplexer Systeme verwendet werden können. Insbesondere die Möglichkeit stochastische Abhängigkeiten zwischen Systemzuständen beschreiben zu können, erlaubt die Anwendung von Markov-Ketten im Bereich hochzuverlässiger und sicherheitskritischer Systeme, die zum Beispiel spezielle Systemzustände mit eingeschränkter Funktionalität besitzen können. Markov-Ketten eignen sich ebenfalls zur Beschreibung des Ausfallverhaltens von Systemen.

Man unterscheidet zwischen zeitdiskreten und zeitkontinuierlichen Markov-Ketten mit endlichen oder unendlichen Zustandsräumen. Im Folgenden werden zeitdiskrete Markov-Ketten mit endlichem Zustandsraum genauer betrachtet, da sie für die Softwaretechnik von besonderem Interesse sind.

Eine Markov-Kette M kann mathematisch als Tupel beschrieben werden:

Mathematische Beschreibung

$M = (S, P)$ wobei

S eine endliche Menge von Zuständen und

P eine $|S| \times |S|$ -Matrix, die die Übergangswahrscheinlichkeiten zwischen allen Zuständen enthält.

Bezeichne p_{ij} die Wahrscheinlichkeit, dass M vom Zustand i in den Zustand j übergeht. Da p_{ij} eine Wahrscheinlichkeit ist, gilt $0 \leq p_{ij} \leq 1$. Die Summe $\sum_{j \in S} p_{ij}$ aller aus einem Zustand i ausgehenden Übergangswahrscheinlichkeiten p_{ij} ist immer eins.

Anschaulich können Markov-Ketten als Zustandsautomaten dargestellt werden, wobei die Übergangswahrscheinlichkeiten an die Zustandsübergänge annotiert werden.

Für die Software »Seminarorganisation« soll ein Lasttest durchgeführt werden. Dazu wird das typische Benutzungsverhalten mit Hilfe einer zeitdiskreten Markov-Kette modelliert. Um den Lasttest durchzuführen, wird dann eine große Zahl von Benutzern simuliert, die sich gemäß der Beschreibung dieser Markov-Kette verhalten. Bei der Modellierung der Markov-Kette für das Verhalten des Kunden geht man folgendermaßen vor:

Beispiel:
SemOrg

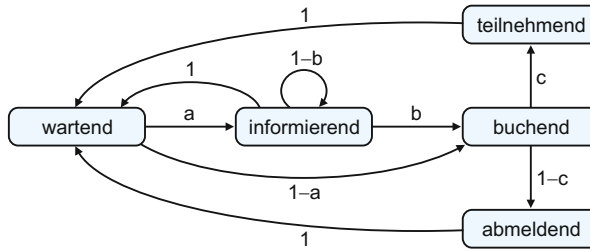
Es gibt drei *Use Cases*: Informieren, Buchen und Abmelden. Jedem dieser *Use Case* wird ein Zustand der Markov-Kette zugeordnet. Weiterhin wird angenommen, dass der Kunde auch an einem Seminar, für das er angemeldet ist, teilnehmen kann (Zustand »teilnehmend«) oder zu keinem Seminar angemeldet ist und momentan auch kein Interesse hat, das zu tun (Zustand »wartend«). Die Übergänge ergeben sich folgendermaßen:

Ein wartender Benutzer kann sich über Seminare informieren oder ein Seminar buchen. Hat sich der Benutzer gerade informiert, so kann er weitere Informationen einholen, wieder warten (beispielsweise, wenn er kein interessantes Seminar gefunden hat) oder eine Seminarveranstaltung buchen. Ist eine Veranstaltung gebucht, so kann der Kunde teilnehmen oder absagen. Nach Teilnahme oder Absage wartet der Kunde wieder. Die einzelnen Übergangswahrscheinlichkeiten können dann zum Beispiel dadurch ermittelt werden, dass man einigen potenziellen Benutzern einen Prototypen zur Verfügung stellt und aus den Auswahlhäufigkeiten der einzelnen Funktionen die Wahrscheinlichkeiten berechnet.

Die Abb. 10.3-28 zeigt die sich ergebende Markov-Kette. Die Übergänge sind mit Übergangswahrscheinlichkeiten annotiert.

III 10 Dynamik

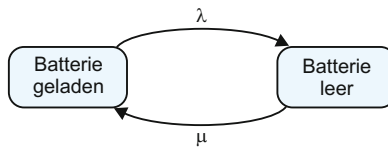
Abb. 10.3-28:
Markov-Kette für
das Beispiel »Semi-
narorganisation«.



Es gibt Systeme bei denen Zustandsänderungen nur zu bestimmten Zeitpunkten sprunghaft sind. Deswegen kann man dort Zeit nicht mehr als einen diskreten Parameter, sondern als kontinuierlichen Parameter ansehen. Es besteht die Möglichkeit die durchschnittliche Zeitspanne für einen Zustandswechsel zu spezifizieren. Diese Zeitspanne ist exponentiell verteilt. Markov-Ketten dieser Art nennt man zeitkontinuierliche Markov-Ketten (im Gegensatz zu den oben beschriebenen zeitdiskreten Markov-Ketten).

Beispiel: Ein möglicher Fehlerfall liegt vor, wenn die Batteriespannung unter 10V liegt. Um die Zuverlässigkeit des Systems einschätzen zu können, ist es somit relevant zu wissen, wie wahrscheinlich es ist, dass die Batterie im normalen Betrieb leer wird. Dazu modelliert man den Zustand der Batterie als zeitkontinuierliche Markov-Kette. Die Batterie kann, wenn sie geladen ist, mit einer Rate λ entladen werden. Wenn sie leer ist, werden die Verbraucher ausgeschaltet, und die Batterie mit Rate μ geladen. Daraus ergibt sich die Markov-Kette in der Abb. 10.3-29.

Abb. 10.3-29:
Markov-Kette für
das Beispiel
»Fensterheber«.



Für die zeitliche Änderung der Aufenthaltswahrscheinlichkeiten in den Zuständen »Batterie geladen« bzw. »Batterie leer« gelten die folgenden Gleichungen, wobei der Index A den Zustand »Batterie geladen« und der Index B den Zustand »Batterie leer« bezeichnet:

$$\frac{dP_A(t)}{dt} = -\lambda \cdot P_A(t) + \mu \cdot P_B(t)$$

$$\frac{dP_B(t)}{dt} = \lambda \cdot P_A(t) - \mu \cdot P_B(t) = -\frac{dP_A(t)}{dt}$$

$$P_A(t) + P_B(t) = 1$$

Das Differentialgleichungssystem besitzt die folgende Lösung:

$$P_A(t) = \frac{\mu}{\mu+\lambda} + \left(c - \frac{\mu}{\mu+\lambda}\right) \cdot e^{-(\mu+\lambda)t}$$

$$P_B(t) = 1 - P_A(t) = 1 - \left[\frac{\mu}{\mu+\lambda} + \left(c - \frac{\mu}{\mu+\lambda}\right) \cdot e^{-(\mu+\lambda)t}\right]$$

Für $t = 0$ erhält man die initialen Aufenthaltswahrscheinlichkeiten in den Zuständen. Falls das Batterie zum Zeitpunkt 0 sicher geladen ist, so besitzt die Konstante c den Wert 1.

$$P_A(t=0) = \frac{\mu}{\mu+\lambda} + \left(c - \frac{\mu}{\mu+\lambda}\right) = c$$

$$P_B(t=0) = 1 - c$$

Für t gegen unendlich erhält man den eingeschwungenen Zustand des Systems.

$$\lim_{t \rightarrow \infty} P_A = \frac{\mu}{\mu+\lambda}$$

$$\lim_{t \rightarrow \infty} P_B(t) = 1 - \frac{\mu}{\mu+\lambda}$$

Diese Aufenthaltswahrscheinlichkeiten sind von den initialen Aufenthaltswahrscheinlichkeiten unabhängig. Wenn μ viel größer ist als λ , dann geht die Wahrscheinlichkeit gegen 1, dass die Batterie sich im geladenen Zustand befindet. Ist μ klein im Vergleich zur λ , so geht die Wahrscheinlichkeit gegen Null, dass die Batterie geladen ist.

10.3.8 Box: Zustandsautomat – Methode und Checkliste

Insbesondere bei der objektorientierten Modellierung von Klassen und *Use Cases* können Zustandsautomaten zu einer detaillierteren und vollständigeren Spezifikation beitragen. Die folgenden Ausführungen beschränken sich daher auf Klassen und *Use Cases*.

Die Modellierung eines Zustandsautomaten hilft, Operationen von Klassen zu identifizieren und die Abhängigkeiten der Operationen in einer Klasse zu verstehen. Normalerweise ist nur für wenige Klassen der Lebenszyklus zu modellieren. [IBM97] gibt an, dass bei typischen Anwendungen (Informationssystemen) nur ein bis zwei Prozent der Klassen einen nicht-trivialen Lebenszyklus besitzen, während deren Anzahl bei Echtzeitanwendungen wesentlich zunimmt. Die Boxen »Checkliste Zustandsautomat« zeigen, wie Zustandsautomaten erstellt und geprüft werden können.

Klassen

Box 1a:
Checkliste Zu-
standsautomat

Ergebnis

- Zustandsdiagramm

Das Zustandsdiagramm kann u. a. das dynamische Verhalten von Klassen und die Verarbeitung von *Use Cases* beschreiben.

Konstruktive Schritte

1 Soll die Klasse durch einen Zustandsautomaten spezifiziert werden?

- Das gleiche Ereignis kann – in Abhängigkeit vom aktuellen Zustand – unterschiedliches Verhalten bewirken.
- Operationen sind nur in bestimmten Situationen (Zuständen) auf ein Objekt anwendbar und werden sonst ignoriert.
- Der Zustandsautomat beschreibt, welche Operationen der Klasse unter welchen Bedingungen aufgerufen werden können.
- Der Zustandsautomat muss alle Operationen der Klasse enthalten, deren Ausführung zustandsabhängig ist, d. h. mit einem Übergang verbunden ist.
- Der Zustandsautomat beschreibt den Lebenszyklus für die Objekte der Klasse.

2 Soll ein Use Case durch einen Zustandsautomaten spezifiziert werden?

- Zustandsautomaten sind bei der *Use Case*-Spezifikation eine Alternative zu Aktivitätsdiagrammen.
- Zustandsdiagramme sind zu wählen, wenn der Fokus *nicht* auf der Verarbeitung, sondern auf den eingenommenen Zuständen liegt.

3 Brainstorming

- Erstellen Sie in einer Brainstorming-Sitzung eine Tabelle mit folgenden Spalten:
 1. Spalte: Alle Zustände.
 2. Spalte: Alle Ereignisse, die extern oder intern auftreten können.
 3. Spalte: Alle Verarbeitungsschritte, die ausgeführt werden müssen.

4 Welche Zustände enthält der Automat?

- Ausgangsbasis ist der Anfangszustand.
- Durch welche Ereignisse wird ein Zustand verlassen?
- Welche Folgezustände treten auf?
- Wodurch wird der Zustand definiert (Attributwerte, Objektbeziehungen)?

5 Benötigt der Zustandsautomat einen Endzustand?

- Wird der Endzustand erreicht, endet die Bearbeitung des Zustandsautomaten.
- Beschreibt der Automat den Lebenszyklus, dann kann das Beenden des Zustandsautomaten gleichgesetzt werden mit dem Lebensende des Objekts.
- In einem Endzustand darf keine Verarbeitung durchgeführt werden und er darf keine Ausgabepfeile besitzen.

6 Welche Aktivitäten sind zu modellieren?

- Ist mit einem Zustandsübergang eine Verarbeitung verbunden?
- Besitzen alle eingehenden Übergänge eines Zustands die gleiche Aktivität? Modellieren als *entry*-Aktivität.
- Besitzen alle ausgehenden Übergänge eines Zustands die gleiche Aktivität? Modellieren als *exit*-Aktivität.
- Ist eine Verarbeitung an die Dauer des Zustands gekoppelt? Modellieren als *do*-Aktivität.

7 Welche Ereignisse sind zu modellieren?

- Externe Ereignisse:
 - ☐ vom Benutzer,
 - ☐ von anderen Objekten.
- Zeitliche Ereignisse:
 - ☐ Zeitdauer,
 - ☐ Zeitpunkt.
- Intern generierte Ereignisse der Klasse oder des *Use Case*.

Analytische Schritte**8 Geeigneter Zustandsname**

- Beschreibt eine bestimmte Zeitspanne.
- Enthält kein Verb.

9 Ist der Zustandsautomat konsistent mit dem Klassendiagramm (wenn eine Klasse spezifiziert wird)?

- Sind alle Operationen auch im Klassendiagramm enthalten?
- Ist jede zustandsabhängige Operation im Zustandsautomaten enthalten?
- Können alle Operationen, die nicht im Zustandsautomaten spezifiziert sind, in jedem beliebigen Zustand aufgerufen werden?

10 Sind alle Übergänge korrekt eingetragen?

- Ist jeder Zustand erreichbar?
- Kann jeder Zustand – mit Ausnahme der Endzustände – verlassen werden?
- Sind die Ereignisse der Übergänge eindeutig?
- Können Ereignisse auftreten, die durch die spezifizierten Ereignisse *nicht* abgedeckt sind?

11 Fehlerquellen

- Modellierung der Benutzungsoberfläche im Lebenszyklus.
- Gedankengut aus den Programmablaufplänen übernommen.

Box 1b:
Checkliste Zu-
standsautomat

Konstruktive Schritte

Die Modellierung eines Lebenszyklus ist sinnvoll, wenn eine der folgenden Situationen zutrifft:

- Ein Objekt kann auf eine bestimmte Botschaft – in Abhängigkeit vom aktuellen Zustand – unterschiedlich reagieren.
- Einige Operationen sind nur in bestimmten Situationen (Zuständen) auf ein Objekt anwendbar und werden sonst ignoriert.

Verzichten Sie auf den Lebenszyklus, wenn seine Beschreibung nichts zum Verständnis der Problematik beiträgt.

1 Zustands-
automat für
Klassen

Für den Seminartyp ergibt sich nur ein trivialer Lebenszyklus (Abb. 10.3-32). Es kann daher auf diesen Zustandsautomaten verzichtet werden.

Beispiel 1a:
SemOrg

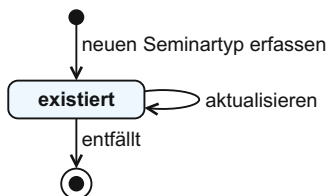


Abb. 10.3-32:
Trivialer
Zustandsautomat
aus der Klasse
Seminartyp.

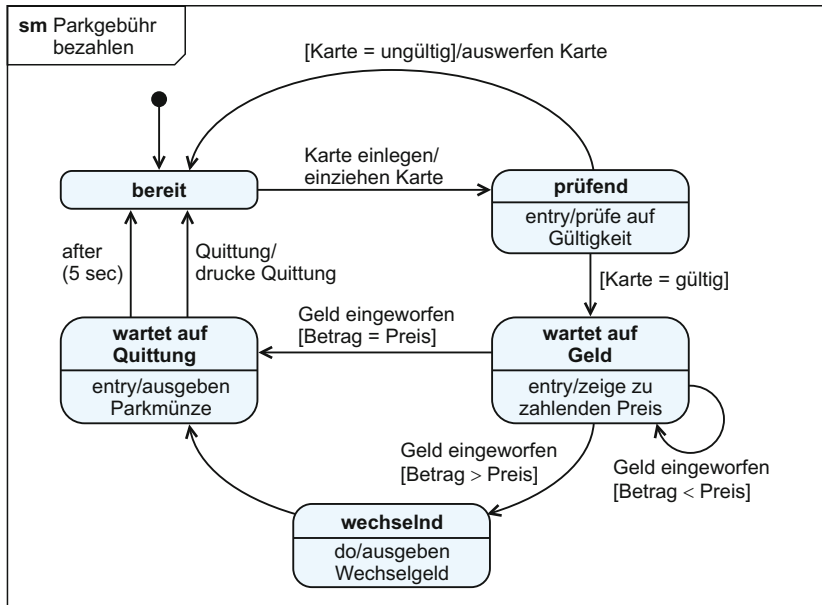
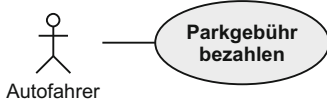


Abb. 10.3-34: Zustandsautomat für einen Use Case.



Spalte alle Verarbeitungsschritte, die im Rahmen eines *Use Case* auftreten oder von Objekten der Klasse ausgeführt werden. Formulieren Sie alle Informationen beliebig in Umgangssprache.

Beginnen Sie mit dem Anfangszustand, d. h. dem Zustand, in dem sich das Objekt befindet, nachdem es erzeugt wurde. Entwickeln Sie ausgehend vom Anfangszustand weitere Zustände. Betrachten Sie dazu jeweils einen Zustand und überlegen Sie, durch welche Ereignisse er verlassen wird. Auf welche Ereignisse muss gewartet werden? In welche Folgezustände erfolgt ein Übergang? Prüfen Sie, wodurch ein Zustand definiert wird, z. B. das Vorliegen bestimmter Attributwerte oder Beziehungen zu anderen Objekten.

In einem Endzustand endet die Bearbeitung des Zustandsautomaten. Der Endzustand impliziert nicht automatisch das Löschen eines Objekts. Dafür sieht die UML ein spezielles Element vor (X-Zeichen).

Prüfen Sie, ob Endzustände existieren, d. h., ob es Zustände gibt, aus denen kein Übergang herausführt. Dazu ist es hilfreich, zyklische und lineare Automaten zu unterscheiden. Endzustände existieren nur bei einem linearen Lebenszyklus (*born and die*). Bei einem zyklischen Zustandsautomaten (*circular lifecycle*) werden die Zustände iterativ durchlaufen. Es gibt keinen Endzustand. Diese Art Lebenszyklus ist typisch für Geräte.

4 Zustände identifizieren

5 Endzustände

III 10 Dynamik

Beispiel: SemOrg Eine Veranstaltung wird erfasst, anschließend werden Anmeldungen und Absagen eingetragen. Wird die Veranstaltung storniert oder durchgeführt, dann müssen im Rahmen der Stornierung alle gebuchten Teilnehmer benachrichtigt oder es muss allen Teilnehmern eine Rechnung zugesandt werden. Anschließend werden die Informationen zu dieser Veranstaltung nicht mehr benötigt und können ins Archiv eingetragen werden. Hier stehen die Veranstaltungen nur noch für statistische Abfragen zur Verfügung, während ihr dynamisches Verhalten nicht mehr von Interesse ist.

6 Aktivitäten Wenn Sie den Automaten mit den Zuständen und Übergängen dargestellt haben, sollten Sie wichtige Aktivitäten eintragen. Überlegen Sie:

- Ist mit einem Zustandsübergang eine Verarbeitung verbunden? Tragen Sie diese Verarbeitung als Aktivität an den Übergang ein, wenn sie von kurzer Dauer ist.
- Besitzen alle eingehenden Übergänge eines Zustands die gleiche Aktivität? Modellieren Sie diese Verarbeitung als *entry*-Aktivität.
- Besitzen alle ausgehenden Übergänge eines Zustands die gleiche Aktivität? Modellieren Sie diese Verarbeitung als *exit*-Aktivität dieses Zustands.
- Ist eine Verarbeitung an die Dauer des Zustands gekoppelt? Wenn die Verarbeitung beendet ist, wird der Zustand durch ein implizites Ereignis verlassen bzw. die Verarbeitung wird beendet, wenn der Zustand durch ein Ereignis verlassen wird. Modellieren Sie diese Verarbeitung als *do*-Aktivität.

7 Ereignisse Außer den externen Ereignissen, die vom Benutzer oder anderen Objekten ausgehen, sind auch zeitliche Ereignisse zu modellieren. Eventuell müssen einige Ereignisse selbst generiert werden, um Nicht-Endzustände zu verlassen.

Überlegen Sie, welche Fehlersituationen auftreten können und wie das Objekt darauf reagieren soll. Unter einem Fehler wird hier jede Abweichung vom normalen bzw. gewünschten Verhalten verstanden. Ein besserer Begriff dafür ist Ausnahmebehandlung (*exception handling*). Beachten Sie, dass zuerst immer das Normalverhalten und erst im zweiten Schritt das Fehlerverhalten betrachtet wird.

Beispiel: SemOrg Wird der letzte Platz belegt, dann muss das Ereignis [Seminar voll] erzeugt werden, um den Zustand *buchend* zu verlassen. Hier handelt es sich um ein Ereignis, das durch die Änderung von Attributwerten ($\text{Teilnehmerzahl} = \text{MaxTeilnehmerzahl}$) erzeugt wird.

Analytische Schritte

Der Zustandsname soll die Zeitspanne beschreiben, in der sich das Objekt in einem Zustand befindet. Der Zustand ist eine Konsequenz früherer Eingaben und beschreibt damit die Historie dieses Objekts. Auch wenn mit dem Zustand eine Verarbeitung verbunden ist, soll der Zustandsname diese Verarbeitung nicht direkt ausdrücken. Nennen Sie einen Zustand beispielsweise prüfend und nicht prüfen. **8 Zustandsname**

Modelliert der Zustandsautomat den Lebenszyklus einer Klasse, dann müssen die Aktivitäten im Zustandsautomaten mit den Operationen des Klassendiagramms konsistent sein. Das bedeutet, dass sich alle Aktivitäten auf Operationen des Klassendiagramms abbilden lassen. **9 Konsistenz**

Alle Zustände müssen erreichbar sein. Besitzt ein Zustand keinen ausgehenden Übergang, dann muss es sich um einen Endzustand handeln. Alle Übergänge, die von *einem* Zustand ausgehen, müssen mit unterschiedlichen Ereignissen bzw. bei gleichen Ereignissen mit unterschiedlichen Bedingungen beschriftet sein. Fehlt bei einem Übergang das Ereignis, dann muss ein implizites Ereignis vorliegen. Prüfen Sie für jeden ausgehenden Übergang eines Zustands: **10 Übergänge**

- Kann eine Situation eintreten, die durch keinen Ausgangspfeil abgedeckt ist? Dann könnte der Zustand niemals über einen Übergang verlassen werden.
- Kann ein Fall eintreten, in dem mehr als ein Ausgangspfeil zutrifft? Dann wäre der Übergang in den nächsten Zustand – und damit das dynamische Verhalten – nichtdeterministisch, sondern rein zufällig.

Lassen Sie Ihre Zustandsdiagramme nicht zu Programmablaufplänen bzw. Flussdiagrammen entarten. Modellieren Sie *keinen* Kontrollfluss mit Entscheidungen und Schleifen. Auch schlecht gewählte Zustandsnamen, die Verben enthalten, bergen die Gefahr, dass keine reinen Zustandsautomaten, sondern Mischungen aus Aktivitätsdiagrammen und Programmablaufplänen entstehen. **11 Keine Flussdiagramme**

10.3.9 Zusammenfassung

Viele Systeme und Geräte zeigen ein Verhalten, das von der bis dahin durchlaufenen Historie abhängt. Das aktuelle Verhalten wird durch den internen Zustand bestimmt, der durch vorausgegangene Eingaben oder Ereignisse erreicht worden ist. Zur Modellierung solcher Systeme eignen sich Zustandsautomaten (*finite state machine*), auch endliche Automaten (*finite automaton, sequential machine*) genannt. **Automat**

Gegenüber einem allgemeinen Automaten besitzen endliche Automaten nur eine endliche Zahl von Zuständen. Bei deterministischen endlichen Automaten gibt es zu einer Eingabe von einem Zustand **Endlicher Automat**

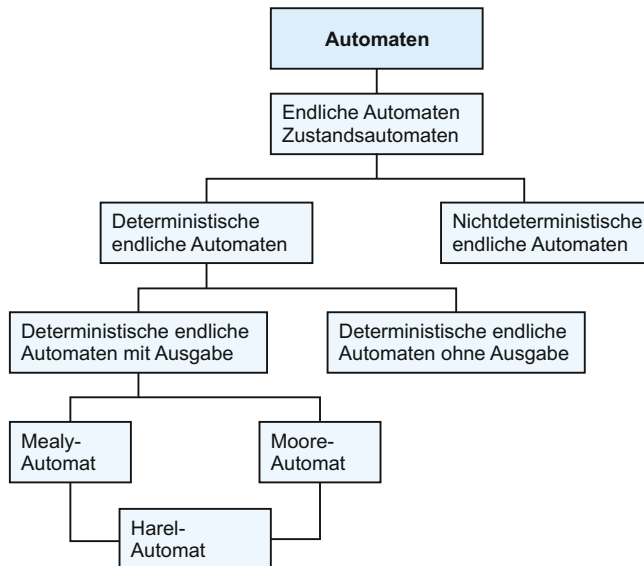
III 10 Dynamik

aus höchstens einen Zustandsübergang (*transition*) in einen anderen Zustand, während bei nichtdeterministischen endlichen Automaten mehrere Übergänge für dieselbe Eingabe möglich sind.

Deterministischer
endlicher Automat

Es gibt deterministische endliche Automaten mit und ohne Ausgabe bzw. Aktionen. In der Softwaretechnik benötigt man im Allgemeinen Automaten mit Aktionen. Beim Mealy-Automaten ist die Aktion an den Zustandsübergang, beim Moore-Automaten an den Zustand gekoppelt. Man spricht dann von Aktivitäten. Einen Überblick über die Klassifikation endlicher Automaten gibt die Abb. 10.3-35.

Abb. 10.3-35:
Klassifikation von
endlichen
Automaten.



Harel-Automat Der Harel-Automat (*statechart*) erlaubt eine Kombination von Mealy- und Moore-Automaten (hybride Automaten). Außerdem können Zustände ineinander geschachtelt werden, was zu hierarchischen Zustandsautomaten führt. Diese unterstützen eine Top-down- sowie Bottom-up-Entwicklung von Automaten. Außerdem kann Nebenläufigkeit beschrieben werden.

OO-Welt In der objektorientierten Welt kann der Lebenszyklus von Objekten und anderen Modellelementen durch Zustandsautomaten modelliert werden. Es werden Verhaltenszustandsautomaten und Protokollzustandsautomaten unterschieden.

Markov-Ketten Zur Modellierung und Analyse hochzuverlässiger und sicherheitskritischer Systeme eignen sich Markov-Ketten – darstellbar als Zustandsautomaten. Es können Übergangswahrscheinlichkeiten zwischen allen Zuständen spezifiziert werden.

Ziel: möglichst wenig Zustände Ziel beim Aufstellen eines Zustandsautomaten ist es, mit möglichst wenig Zuständen auszukommen.

Zur Klassifikation

- Zustandsautomaten werden sowohl tabellarisch (Zustandstabelle, Zustandsmatrix) als auch grafisch (Zustandsdiagramm mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt – in der Implementierungsphase jedoch ausprogrammiert.
- Sie werden in allen Anwendungsbereichen – besonders aber in technischen und softwareintensiven Systemen – verwendet.

10.4 Petrinetze

Petrinetze eignen sich zur Modellierung, Analyse und Simulation von dynamischen Systemen mit nebenläufigen und nichtdeterministischen Vorgängen.

Ein **Petrinetz** ist ein gerichteter Graph, der aus zwei verschiedenen Sorten von Knoten besteht, aus **Stellen** und **Transitionen**. Eine Stelle entspricht einer Zwischenablage von Informationen, eine Transition beschreibt die Verarbeitung von Informationen.

Stellen (auch Plätze oder Zustände genannt) werden durch Kreise, Transitionen (auch Hürden oder Zustandsübergänge genannt) durch Rechtecke oder Balken dargestellt (Abb. 10.4-1). Die Kanten dürfen jeweils nur von einer Sorte zur anderen führen.



Prof. Dr. Petri

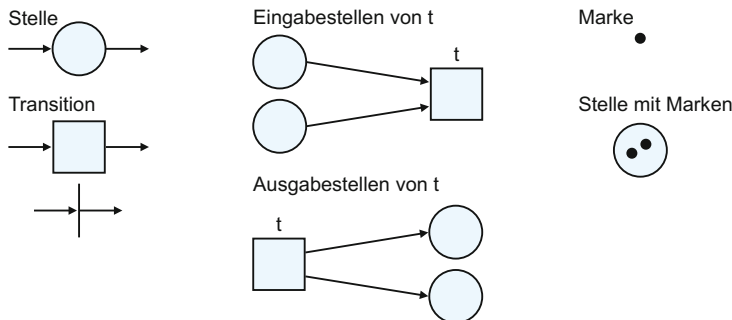


Abb. 10.4-1:
Elemente zum
Aufbau von
Petrinetzen.

Stellen, von denen Kanten zu einer Transition t laufen, heißen **Eingabestellen** von t (auch Vorbereich von t genannt).

Eingabestellen

Stellen, zu denen – von einer Transition t aus – Kanten führen, heißen **Ausgabestellen** von t (auch Nachbereich von t genannt).

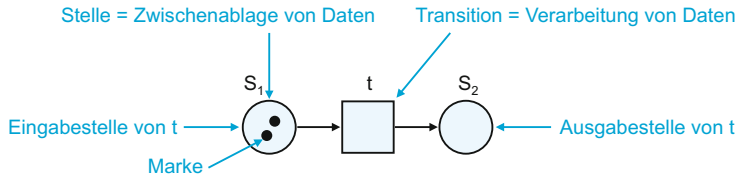
Ausgabestellen

Zur Beschreibung der dynamischen Vorgänge in einem Petrinetz werden die Stellen mit Objekten belegt. Diese Objekte werden durch die Transitionen weitergegeben. Objekte werden als **Marken** (*tokens*) bezeichnet und als kleine schwarze Kreise in die Stellen des Petrinetzes eingetragen (Abb. 10.4-2).

Marken

III 10 Dynamik

Abb. 10.4-2:
Notation und
Symbolik eines
Petrinetzes.

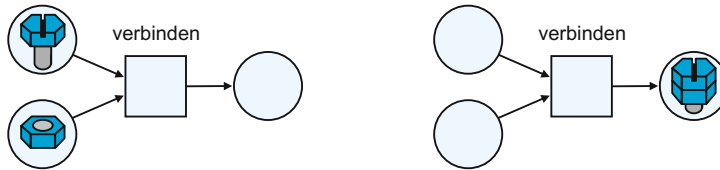


Schaltregel Der Bewegungsablauf der Marken im Netz wird durch folgende **Schaltregel** festgelegt:

- a** Eine Transition t kann schalten oder »feuern«, wenn jede Eingabestelle von t mindestens eine Marke enthält.
- b** Schaltet eine Transition, dann wird aus jeder Eingabestelle eine Marke entfernt und zu jeder Ausgabestelle eine Marke hinzugefügt.

Beispiel 1 Eine anschauliche Vorstellung der Schaltregel zeigt die Abb. 10.4-3. Erst wenn die Schraube und die Mutter in der jeweiligen Eingabestelle der Transition »verbinden« vorhanden sind, kann die Transition ausgeführt, d. h. Schraube und Mutter verbunden werden. Das Ergebnis wird in die Ausgabestelle gelegt, aus der jeweiligen Eingabestelle wird eine Marke entnommen.

Abb. 10.4-3:
Schalten einer
Transition.



In Abhängigkeit von der Art der Objekte unterscheidet man verschiedene Netzarten, die im Folgenden behandelt werden:

- »Bedingungs/Ereignis-Netze«, S. 305
- »Stellen/Transitions-Netze«, S. 309
- »Prädikat/Transitionsnetze«, S. 311

Petrinetze können auch hierarchisch strukturiert, zeitbehaftet und stochastisch sein:

- »Hierarchische Petrinetze«, S. 312
- »Zeitbehaftete Petrinetze«, S. 314
- »Generalisierte stochastische Petrinetze«, S. 316

UML-Aktivitätsdiagramme basieren semantisch auf Petrinetzen:

- »Aktivitätsdiagramme und Petrinetze«, S. 317

Die Strukturelemente der Petrinetze lassen sich zu Strukturen und Anwendungsmustern zusammensetzen:

- »Strukturelemente und Strukturen von Petri-Netzen«, S. 320

Beim Erstellen von Petrinetzen ist es wichtig, methodisch vorzugehen:

- »Box: Petrinetze – Methode«, S. 322

Petrinetze können analysiert und simuliert werden, um sie auf Korrektheit zu überprüfen:

■ »Analyse und Simulation von Petrinetzen«, S. 327

Petrinetze besitzen viele Vorteile, aber auch Nachteile:

■ »Wertung«, S. 328

Petrinetze lassen sich klassifizieren:

■ »Zusammenfassung«, S. 330

[Reis85], [HRV 84]

Literatur

10.4.1 Bedingungs/Ereignis-Netze

Ein Petrinetz ist ein **Bedingungs/Ereignis-Netz** (B/E-Netz) (*Condition/Event Net, C/E Net*), wenn die Objekte bzw. Marken vom Datentyp boolean sind (dargestellt durch eine vorhandene bzw. nicht vorhandene Marke). Die Transitionen werden als Ereignisse interpretiert. Die Stellen werden als Bedingungen bezeichnet. Jede Stelle kann entweder genau eine oder keine Marke enthalten.

B/E-Netz

Daraus folgt eine zusätzliche Bedingung für das Schalten einer Transition in einem B/E-Netz:

Schaltregel

- c Eine Transition t kann schalten, wenn jede Eingabestelle von t eine Marke enthält und wenn jede Ausgabestelle von t leer ist.

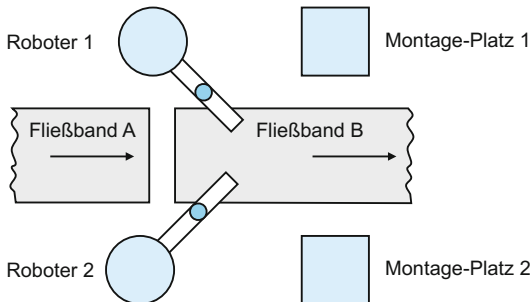


Abb. 10.4-4: Bestückungsroboter mit Fließband.

Zwei Roboter bestücken Leiterplatten mit elektronischen Bauelementen, die auf einem Fließband A antransportiert werden (Abb. 10.4-4). Ist ein Roboter frei, dann kann er eine antransportierte Leiterplatte vom Fließband nehmen. Sind beide Roboter frei, dann wird nichtdeterministisch entschieden, welcher Roboter die Leiterplatte nimmt. Beide Roboter »konkurrieren« also um die antransportierte Leiterplatte. Sie arbeiten parallel, da die Bestückung mehr Zeit erfordert als die anderen Bearbeitungsschritte der Leiterplatte. Die Zeit zur Bestückung einer Leiterplatte kann unterschiedlich lang sein, da jede Leiterplatte anders zu bestücken sein kann. Ist eine Leiterplatte bestückt, dann wird sie von dem jeweiligen Roboter auf das Fließband B gelegt und abtransportiert. Sind beide Roboter gleichzeitig

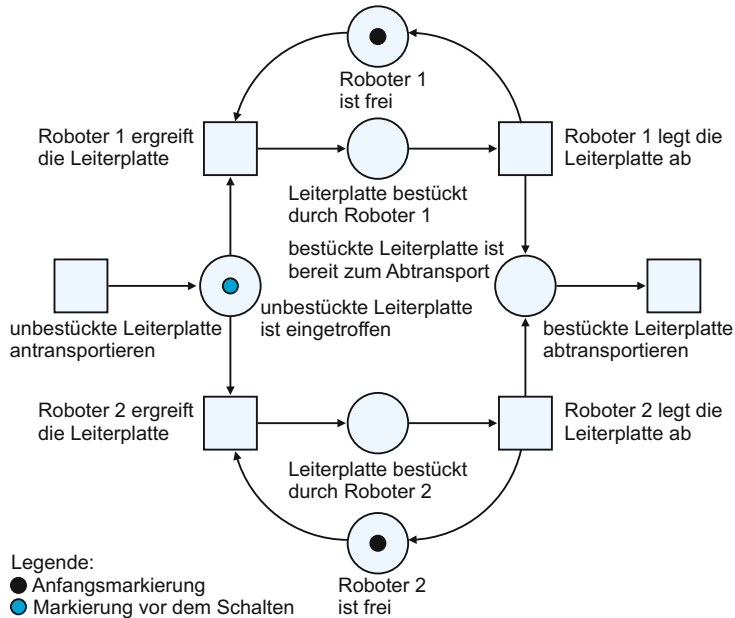
Beispiel 1a:
Roboter

III 10 Dynamik

fertig, dann wird nichtdeterministisch bestimmt, welcher Roboter die Leiterplatte zuerst ablegt. Durch Lichtschranken gesteuert werden die Fließbänder gestartet und gestoppt.

Die Abb. 10.4-5 zeigt die **Anfangsmarkierungen** (schwarze Marken) des Netzes (Initialisierungszustand): Roboter 1 und Roboter 2 sind frei.

Abb. 10.4-5: B/E-Netz des Bestückungsroboters vor dem Schalten.



Keine Transition kann schalten, da keine Vorbedingung einer Transition vollständig erfüllt ist. Die Situation ändert sich, nachdem eine unbestückte Leiterplatte antransportiert wurde (Abb. 10.4-5, blaue Marke). Vor- und Nachbedingungen der Transitionen Roboter 1 ergreift die Leiterplatte und Roboter 2 ergreift die Leiterplatte sind erfüllt. Diese Transitionen heißen aktiviert, da die Ereignisse eintreten können.

Konflikt Wenn zwei aktivierte Ereignisse mindestens eine gemeinsame Vorbedingung (hier: unbestückte Leiterplatte wird antransportiert) oder Nachbedingung besitzen, können sie miteinander in Konflikt geraten, sie konkurrieren miteinander. Tritt nun eines der Ereignisse ein, dann ist das andere nicht mehr aktiviert (Abb. 10.4-6).

Nicht-deterministisch Das tatsächliche Eintreten konkurrierender Ereignisse ist zufällig und das Verhalten des Netzes ist somit nicht mehr deterministisch.

Beispiel 1b Wird nun angenommen, dass die Transition Roboter 1 ergreift die Leiterplatte feuert, dann ergibt sich nach dem Feuern die Markenbelegung der Abb. 10.4-6. Die Transition Roboter 2 ergreift die

Leiterplatte kann nun nicht mehr schalten, da in ihrer Eingangsstelle unbestückte Leiterplatte ist eingetroffen keine Marke mehr vorhanden ist.

Die Transition Roboter 1 legt die Leiterplatte ab kann nun schalten, da ihre Vorbedingung jetzt erfüllt ist. Die Stellen Roboter 1 ist frei und bestückte Leiterplatte ist bereit zum Abtransport erhalten eine Marke.

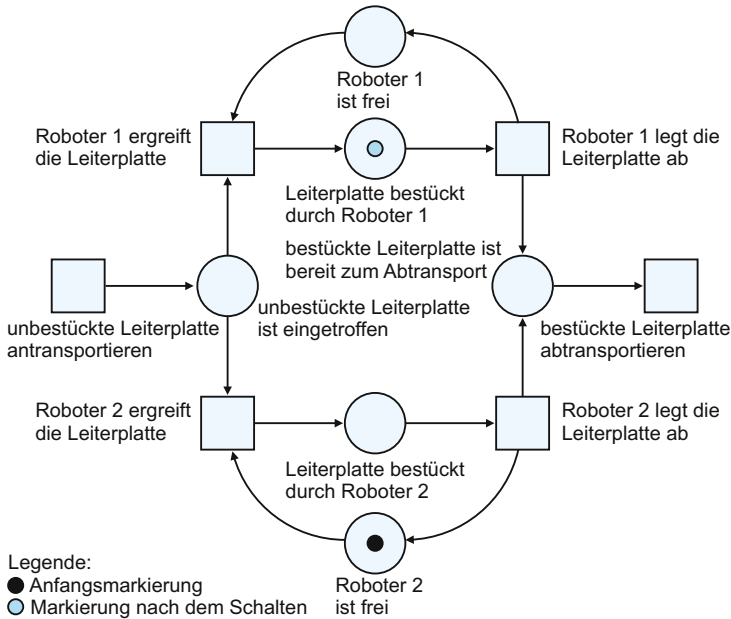


Abb. 10.4-6: B/E-Netz des Bestückungsroboters nach dem Schalten.

Wie das Beispiel 1b zeigt, erlauben Petrinetze eine Modellierung nicht-deterministischer Vorgänge. Der Nichtdeterminismus spiegelt hier die Realität wider. Dauert das Bestücken einer Leiterplatte unterschiedlich lang, dann können die Roboter in beliebiger Reihenfolge aktiv werden. Dies wird durch das Petrinetz ausgedrückt.

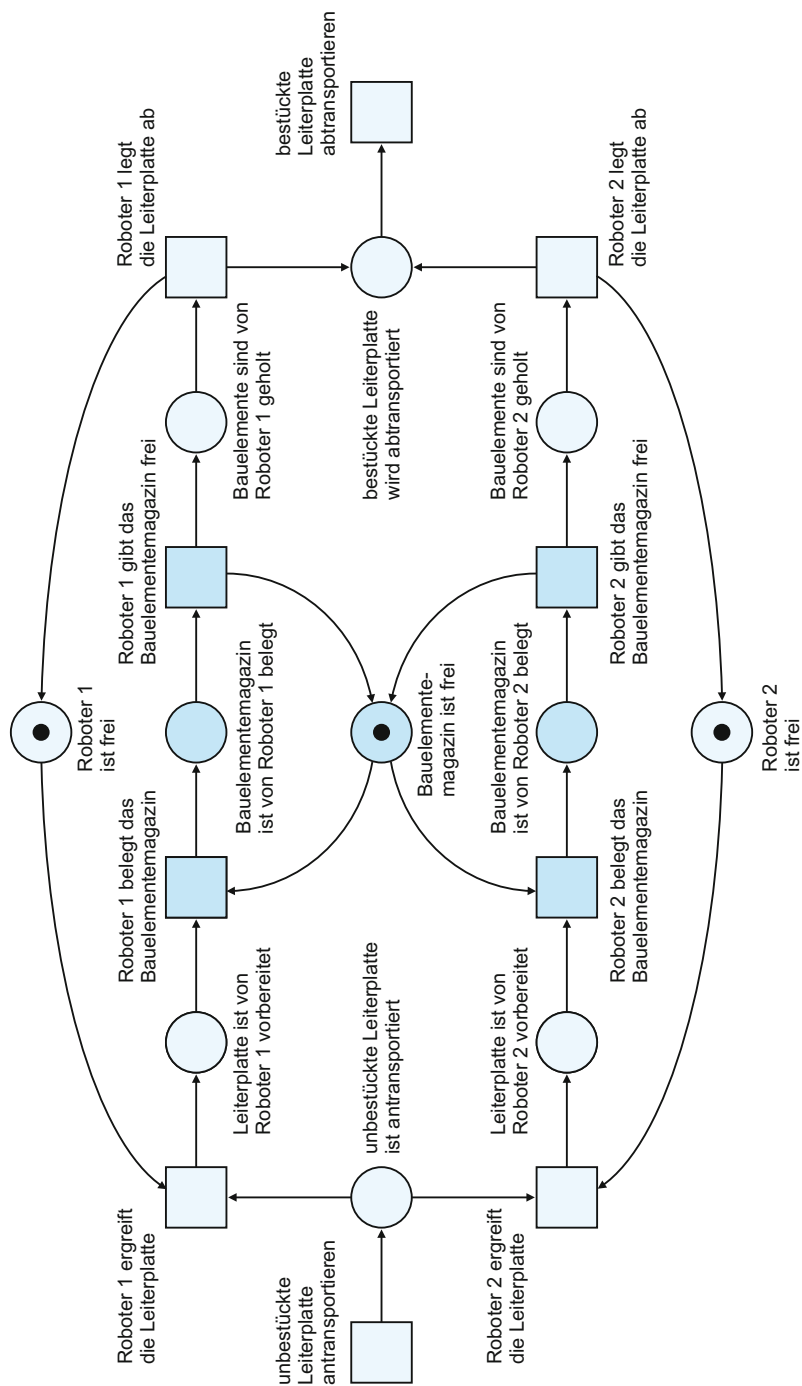
Angenommen, beide Roboter greifen auf ein gemeinsames Bauelementemagazin zu, holen dort alle zur Bestückung einer Leiterplatte benötigten Bauelemente und bestücken dann die Leiterplatte, so ergibt sich das Petrinetz nach der Abb. 10.4-7.

Beispiel 1c
Roboter 2

Da B/E-Netze in ihrer Modellierungskraft begrenzt sind, hat man sie zu S/T-Netzen weiterentwickelt.

III 10 Dynamik

Abb. 10.4-7: B/E-
Netz des Be-
stückungsroboters
mit gemeinsamem
Bauelemente-
magazin.



10.4.2 Stellen/Transitions-Netze

In **Stellen/Transitions-Netzen** (S/T-Netze) (*Place/Transition Net*, *P/T Net*) können Stellen mehr als eine Marke enthalten (in B/E-Netzen nur eine Marke). Die Transitionen müssen so viele Marken beim Schalten wegnehmen oder hinzufügen, wie die Gewichte an den Pfeilen angeben (in B/E-Netzen nur eine Marke).

Soll eine Stelle eine Kapazität größer 1 erhalten, dann wird dies durch »K = ...« an der Stelle notiert. Die Kapazität definiert die maximale Anzahl von Marken, die auf einer Stelle liegen dürfen. Eine Transition darf nur feuern, wenn dadurch die Kapazität der Ausgangsstellen nicht überschritten wird. Die Abb. 10.4-8 zeigt ein Beispiel für ein S/T-Netz.

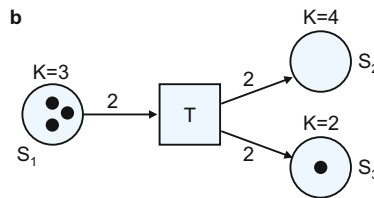
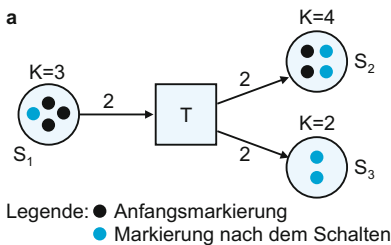


Abb. 10.4-8: Schaltbedingungen bei S/T-Netzen.

Im Fall a der Abb. 10.4-8 kann T schalten. Anschließend sind in S1 eine Marke, in S2 vier Marken und in S3 zwei Marken. Beispiel

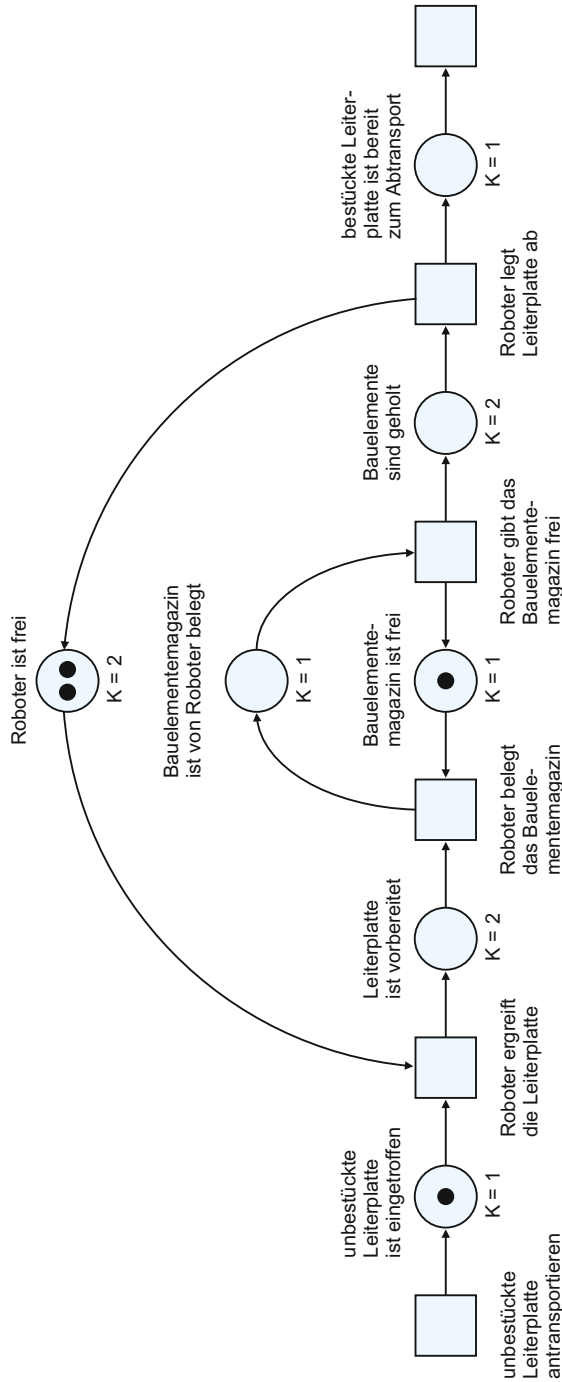
Im Fall b kann T nicht schalten, da in S3 dann drei Marken liegen würden. Dies ist wegen $K = 2$ von S3 nicht erlaubt.

Aktiviert werden Transitionen nur, wenn bei keiner Eingabestelle das Kantengewicht die Anzahl vorhandener Marken übersteigt und nach dem Schalten die Kapazitäten der Ausgabestellen nicht überschritten werden. Eine Transition kann feuern, wenn sie aktiviert ist. Schaltet sie, so werden die Marken auf den Eingangsstellen um die Gewichte der Pfeile vermindert und die Marken der Ausgangsstellen um die Gewichte der Pfeile erhöht.

B/E-Netze können also durch S/T-Netze mit Kapazitäten und Gewichten gleich Eins dargestellt werden.

Das Beispiel »Roboter« (siehe »Bedingungs/Ereignis-Netze«, S. 305) lässt sich mit einem S/T-Netz vereinfacht modellieren. Von den individuellen Robotern 1 und 2 wird abstrahiert. Dass es sich um zwei Roboter handelt, wird durch die Kapazität $K = 2$ an den entsprechenden Stellen ausgedrückt (Abb. 10.4-9). Beispiel: Roboter

Abb. 10.4-9: S/T-
Netz des Be-
stückungsroboters.



10.4.3 Prädikat/Transitions-Netze

B/E- und S/T-Netze verwenden nur »schwarze« Marken, die alle gleich sind. Werden individuelle, »gefärbte« Marken verwendet, dann spricht man von **Prädikat/Transitions-Netzen** (Pr/T-Netze). Pr/T-Netz

Individuelle Marken werden durch einen Wert, z. B. eine Zahl, repräsentiert. Transitionen können nur noch beim Vorliegen bestimmter Marken schalten. Den Transitionen werden Schaltbedingungen mit Variablen zugeordnet, die die von den Eingabestellen erhaltenen Marken repräsentieren. Die Bindung von Marken an Variablen geschieht dadurch, dass die Pfeile mit den Namen der Variablen versehen werden. Im oberen Teil der Transition wird eine Schaltbedingung (*firing condition*) angegeben. In ihr werden die Variablen verwendet, durch die die aus den Stellen erhaltenen Marken dargestellt werden.

Die Transition kann nur für die Marken schalten, für die die Schaltbedingung erfüllt ist. Beim Schalten verbraucht sie die Marken aus den Eingabestellen und erzeugt neue Marken in den Ausgabestellen. Die Werte der neuen Marken ergeben sich in der Regel aus den Marken der Eingabestellen. Zu ihrer Berechnung gibt man im unteren Rechteckbereich der Transition die Schaltwirkung (*firing result*) an. Die so erzeugten Marken werden entsprechend der Kennzeichnung der Pfeile auf die Ausgabestellen verteilt.

Die Abb. 10.4-10 zeigt ein Pr/T-Netz. Eine Marke wird durch eine Zahl dargestellt. Beispiel

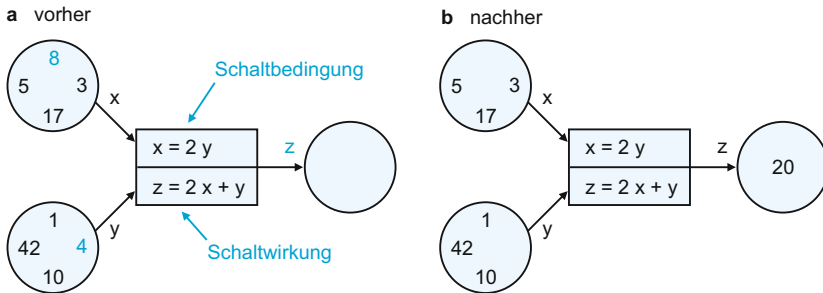


Abb. 10.4-10:
Schalten eines
Netzes mit
unterscheidbaren
Marken.

Ist das Schalten nicht von bestimmten Marken abhängig, dann entfällt die Schaltbedingung. Werden Marken durch die Transition nicht verändert, sondern lediglich von den Eingangs- an die Ausgangsstellen weitergegeben, dann entfällt die Schaltwirkung.

Das Beispiel »Roboter 2« (siehe »Bedingungs/Ereignis-Netze«, S. 305) lässt sich durch Pr/T-Netze modellieren, wenn die Aufgabenstellung sich folgendermaßen ändert: Beispiel:
Roboter 2

III 10 Dynamik

Roboter 2 kann – im Gegensatz zu Roboter 1 – auch übergroße Leiterplatten bestücken. Den Leiterplatten werden daher zwei Mengen zugeordnet, L_g und L_n . Die Roboter werden durch die individuellen Marken R_1 und R_2 gekennzeichnet.

Durch konstante (hier: R_1, R_2) und variable Pfeilbeschriftungen (hier: L_g, L_n) wird jetzt der Markenfluss beschrieben (Abb. 10.4-11). Zusätzliche Schaltbedingungen und Schaltwirkungen sind in diesem Beispiel nicht erforderlich.

Eine konstante Pfeilbeschriftung bedeutet, dass dort nur entsprechend bezeichnete Marken »fließen« dürfen. Bei einer variablen Pfeilbezeichnung müssen die Marken zu dem Wertebereich der angetragenen Variablen gehören, z. B. $L_g = \{L_1, L_2, L_3\}$.

Die Notation der Pr/T-Netze ist nicht einheitlich.

10.4.4 Hierarchische Petrinetze

Um umfangreiche Petrinetze besser zu strukturieren und zu entwickeln, können Netzkanten durch detaillierte Unternetze verfeinert werden. Umgekehrt können Netzknoten zu einer höheren Einheit zusammengefasst werden. Grundsätzlich können sowohl Stellen als auch Transitionen verfeinert bzw. zusammengefasst werden. Eine Verfeinerung der Transitionen ist jedoch vorzuziehen, da dies einer besseren zeitlichen Auflösung entspricht.

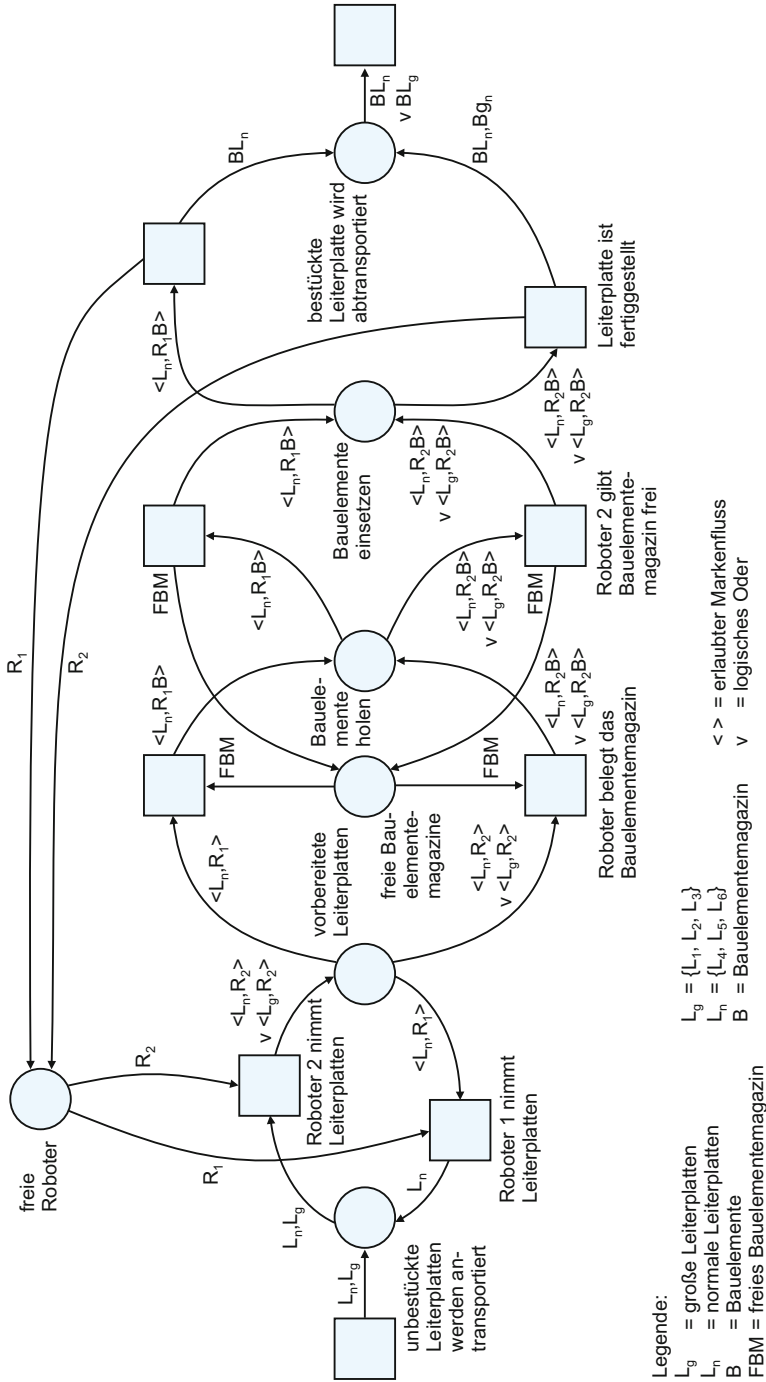
Kanal-Instanzen-
Netz
Kanal = Stelle

Die oberen Netzebenen bei hierarchischen Petrinetzen werden oft als **Kanal-Instanzen-Netze** (*channel agency nets*) bezeichnet. Eine Stelle wird als Kanal interpretiert. Einen solchen Kanal kann man als passive Systemkomponente ansehen, in der Informationen oder Material abgelegt werden können. Eine Transition wird als Instanz interpretiert. Instanzen repräsentieren aktive Komponenten, die Informationen oder Material verarbeiten. Instanzen kommunizieren über Kanäle. Die Abb. 10.4-12 zeigt ein hierarchisches Petrinetz mit verfeinerten Transitionen.

Hierarchische Petrinetze erweitern nicht das entsprechende Netz-Modell, sondern erlauben nur eine strukturierte Darstellung eines komplexen Netzes. Bei der Hierarchiebildung sind jedoch Regeln zur Konsistenzerhaltung des Modells zu beachten:

- Regeln
- Zusätzliche Pfeile können nur innerhalb eines Baumknotens zwischen Stellen und Transitionen eingefügt werden, d. h., es dürfen zwischen Baumknoten auf tieferen Ebenen keine neuen Pfeile gezogen werden.
 - Pfeile, die von einem Unternetz weg- oder zu ihm hinführen, müssen bereits vor der Verfeinerung vorhanden sein und die vorgegebenen Richtungen müssen berücksichtigt werden.
 - Eine Verfeinerung muss markengetreu erfolgen, d. h., sie soll gleich viele Marken abgeben, wie sie verbraucht (Abb. 10.4-13).

Abb. 10.4-11: Netz mit variablen Pfeilanschriften.



III 10 Dynamik

Abb. 10.4-12:
Hierarchisches
Petrinetz.

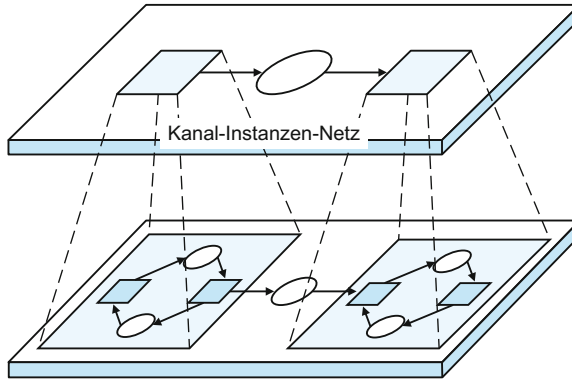
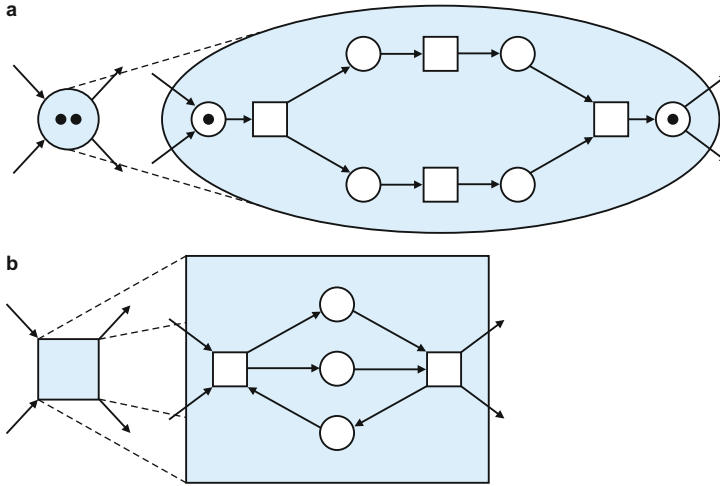


Abb. 10.4-13:
Markengetreue
Verfeinerung einer
Stelle (a) und einer
Transition (b).

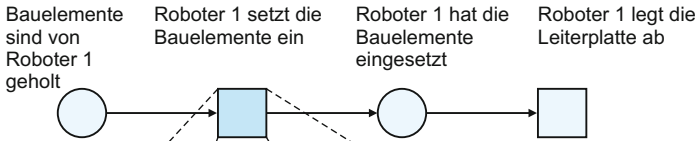


Beispiel: Roboter Das Modell der Leiterplattenbestückung soll erweitert werden, so-
dass das Einsetzen der Bauelemente detailliert beschrieben wird. Dazu werden die Transition Roboter 1 setzt Bauelemente ein und die Stelle Roboter 1 hat die Bauelemente eingesetzt ergänzt (Abb. 10.4-14).

10.4.5 Zeitbehaftete Petrinetze

Die bisher betrachteten Petrinetze arbeiten ohne Zeitbegriff. Alle Transitionen werden zum gleichen Zeitpunkt abgearbeitet, an dem die dafür erforderlichen Bedingungen erfüllt sind. Zur Untersuchung der Leistungsfähigkeit oder zur Simulation eines Systems muss jedoch oft die Dauer einer Aktion oder eines Ereignisses festgelegt werden.

a vor der Verfeinerung



b nach der Verfeinerung

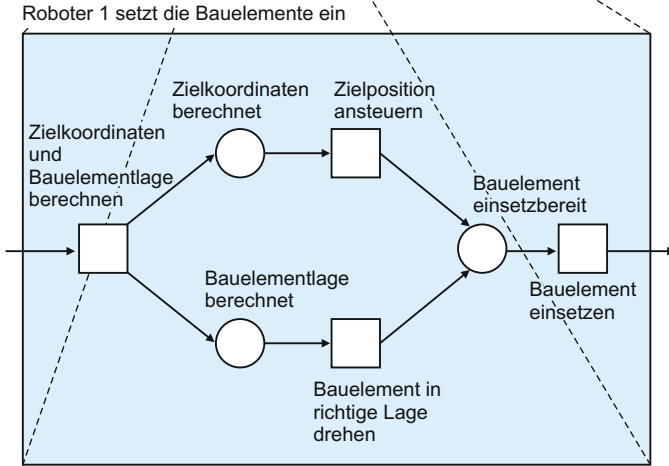


Abb. 10.4-14:
Beispiel einer Stellenverfeinerung.

Atomare Ereignisse oder Transitionen sind per Definition beliebig schnell und benötigen keine Zeit. Ein zusammengesetztes Ereignis oder eine zusammengesetzte Transition kann jedoch eine bestimmte Zeitdauer benötigen. Die Marken müssen daher eine bestimmte Zeitdauer ($t > 0$) auf einer Stelle verharren, bevor sie von einer Transition verbraucht werden können.

Die Zeitintervalle können in den Stellen oder den Transitionen festgelegt werden, verharren müssen die Marken aber jeweils auf den Stellen.

In der Abb. 10.4-15 zeigen **a** und **b** verschiedene Notationsmöglichkeiten, das Zeiterhalten eines Petrinetzes darzustellen. **a** bedeutet, dass eine Marke erst nach einer Verzögerung von vier Zeiteinheiten zur Verfügung steht, **b** bedeutet, dass die Transition erst vier Zeiteinheiten nach dem Eintreffen der Marke in der Eingangsstelle feuern kann.

Beispiel

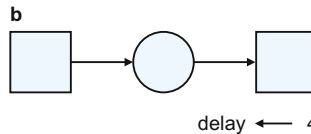
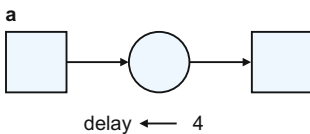
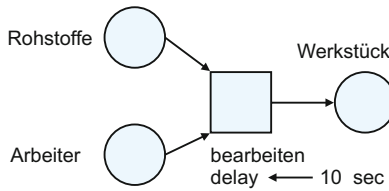


Abb. 10.4-15:
Verschiedene Notationsmöglichkeiten für zeitbehaftete Petrinetze.

III 10 Dynamik

Beispiel Die Abb. 10.4-16 zeigt ein zeitbehaftetes Petrinetz.

Abb. 10.4-16:
Zeitbehaftetes
Petrinetz.



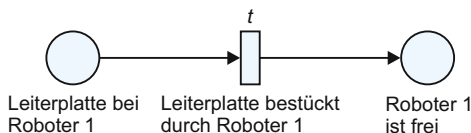
Zeitbehaftete Petrinetze lassen sich nochmals gliedern in zeitattributierte Netze (Transitionen mit deterministischem Zeitverbrauch) und stochastische Netze (Transitionen mit stochastischem Zeitverbrauch).

10.4.6 Generalisierte stochastische Petrinetze

SPN Stochastische Petrinetze (SPN) erweitern die klassischen Petrinetze um so genannte *zeitbehaftete* Transitionen. Die Schalt- oder Verzögerungszeit einer Transition, d.h. die Zeit, die zwischen Aktivieren und Feuern der Transition verstreicht, wird durch eine exponentiell verteilte Zufallsvariable beschrieben. Stochastische Petrinetze können in äquivalente zeitkontinuierliche Markov-Ketten (siehe auch »Markov-Ketten«, S. 292) umgewandelt werden. Wenn eine bestimmte Markierung mehrere Transitionen aktiviert, dann feuert diejenige Transition mit der (durch das jeweilige Zufallsexperiment ermittelten) kürzesten Verzögerungszeit als erste. Im neuen Zustand können die anderen Transitionen immer noch feuerbereit sein (in Abhängigkeit von den Marken). Ihre Restverzögerungszeiten genügen den ursprünglichen Verteilungen.

Beispiel Im Beispiel »Roboter« (siehe »Bedingungs/Ereignis-Netze«, S. 305) kann angenommen werden, dass die Bestückungszeit exponentiell verteilt ist. Deswegen kann man die Transition »Leiterplatte bestücken« mit einer **Rate** t annotieren. Diese Rate gibt an, wie lange es im Durchschnitt dauert, bis der Roboter 50% einer Menge von Leiterplatten bestückt hat.

Abb. 10.4-17:
Beispiel für ein
stochastisches
Petrinetz.



GSPN Generalisierte stochastische Petrinetze (GSPN) erhält man, indem stochastische Petrinetze um zeitlose Transitionen und hemmende Kanten erweitert werden.

Zeitlose Transitionen werden meist durch dünne Balken dargestellt. Sie haben *keine* Verzögerungszeit. Wenn eine zeitlose Transition aktiviert ist, schaltet sie sofort.

Eine **hemmende Kante** verbindet eine Stelle mit einer Transition. Über diese Kante selbst fließen *keine* Marken. Sie dient nur dazu, zu bestimmen, ob die Transition feuern kann. Eine hemmende Kante übt auf eine Transition ihre hemmende Wirkung aus, wenn eine bestimmte Anzahl von Marken in der zugeordneten Stelle überschritten ist. Diese Anzahl wird durch das Kantengewicht spezifiziert. Damit eine Transition aktiv werden kann, müssen folgende Bedingungen erfüllt sein:

- 1 Auf allen Eingangsstellen der normalen Kanten befinden sich genug Marken.
- 2 Auf den Stellen mit hemmenden Kanten befinden sich nicht zu viele Marken.

Schaltet die Transition, so ändert sich dabei *nicht* die Anzahl der Marken dieser Stelle. Als grafische Repräsentation wird eine Linie mit einem Kreis an der Seite der Transition verwendet statt einem Pfeil bei gewöhnlichen Kanten. Generalisierte stochastische Petrinetze, in denen zusätzlich Transitionen mit fester Schaltzeit enthalten sind, werden deterministische und stochastische Petrinetze (DSPN) genannt.

Das Roboter-Beispiel (siehe »Bedingungs/Ereignis-Netze«, S. 305) kann man mit Hilfe eines GSPN modellieren. Dabei wird angenommen, dass der Roboter 1 die Leiterplatte mit Rate t_1 bestückt und Roboter 2 diese Aufgabe mit Rate t_2 erledigen kann. Das sich so ergebende GSPN ist in Abb. 10.4-18 dargestellt. Beispiel

10.4.7 Aktivitätsdiagramme und Petrinetze

UML-Aktivitätsdiagramme (siehe »Aktivitätsdiagramm«, S. 236) beschreiben dynamisches Verhalten. Das exakte dynamische Verhalten wird durch den Markenfluss spezifiziert, der sich durch die Aktionen des Aktivitätsdiagramms bewegt – analog wie beim Petrinetz.

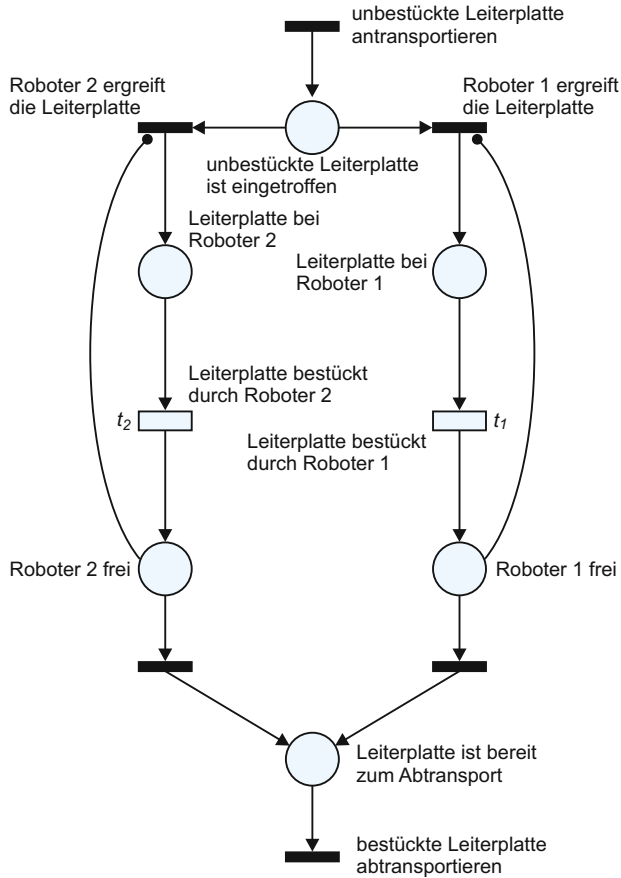
Marken-Konzept

Im Gegensatz zum Petrinetz werden die Marken *nicht* in das Aktivitätsdiagramm eingetragen. Da das Aktivitätsdiagramm mehr Elemente als ein Petrinetz besitzt, gibt es auch zusätzliche Regeln für den Markenfluss [OMG09a, S. 311 ff.]. Im Folgenden werden wichtige Regeln erklärt und durch Markenannotationen verdeutlicht.

Kanten und Knoten können in einem Aktivitätsdiagramm mit Marken (*token*) behaftet sein. Knoten bestimmen, wann eine Marke angenommen wird und wann sie den Knoten verlassen darf. Die Regeln an den Kanten kontrollieren, wann eine Marke von einem Ausgangsknoten entfernt und einem Zielknoten übergeben wird.

III 10 Dynamik

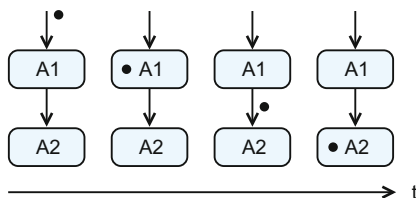
Abb. 10.4-18:
Beispiel für ein
generalisiertes
stochastisches
Petrinetz.



Marken &
Aktionsknoten

Die Abb. 10.4-19 zeigt eine einfache Marken-Übergabe von der eingehenden Kante in die Aktion A1. Wenn A1 vollständig abgearbeitet ist, wird die Marke an die ausgehende Kante gegeben und kann von Aktion A2 entgegengenommen werden. Die Marken werden hier analog zur Petrinetz-Notation durch kleine gefüllte Kreise dargestellt. Hierbei handelt es sich *nicht* um eine UML-Notation, sondern soll nur zur Veranschaulichung des Marken-Flusses dienen.

Abb. 10.4-19:
Marken von
Kanten und
Aktionen.



Marken &
Kontrollknoten

Kontrollknoten steuern den Fluss der Marken zwischen den Objektknoten und den Aktionen. Marken können sich *nicht* in Kontrollknoten (Auswahl usw.) »ausruhen« und darauf warten, sich später

weiter zu bewegen. Kontrollknoten agieren wie Verkehrsleitsysteme und verteilen die Marken auf die entsprechenden Kanten. Nur in Aktionen und Objektknoten können sich Marken kürzere oder längere Zeit aufhalten. In einer Aktivität können zu einem Zeitpunkt mehrere Marken enthalten sein. Objektknoten können Marken sammeln bzw. puffern. Diese Marken repräsentieren Daten, die in den Objektknoten eingetragen werden.

Die Abb. 10.4-20 zeigt den Fluss der Marken durch eine Auswahl mit anschließender Zusammenführung. In Abhängigkeit davon, ob Bedingung B1 oder B2 erfüllt ist, kann die Marke nur auf eine der beiden alternativen Kanten »fließen«. Bei der Zusammenführung wird diese eine Marke weitergegeben und steht anschließend für die Aktion A4 bereit.

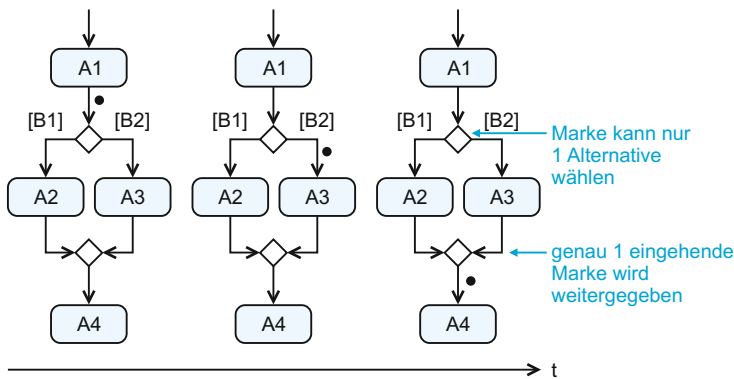


Abb. 10.4-20:
Marken bei einer
Entscheidung und
einer Zusammen-
führung.

Eine andere Situation ergibt sich beim Splitting-Knoten (Abb. 10.4-21). Hier wird die Marke von der Ausgangskante von A1 vervielfältigt und »fließt« dann sowohl in die Aktion A2 als auch in A3. Der Synchronisationsknoten wartet, bis alle Marken eintreffen, verschmilzt sie zu einer einzigen Marke und gibt diese eine Marke an die Aktion A4 weiter.

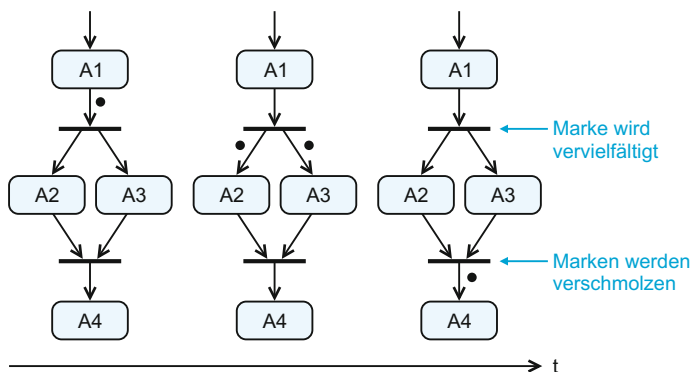
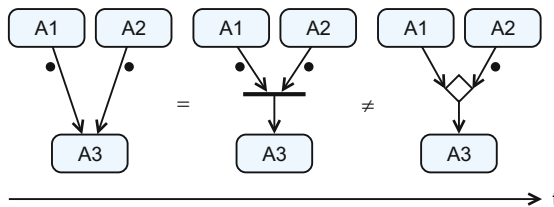


Abb. 10.4-21:
Marken beim
Splitting und bei
der
Synchronisation.

III 10 Dynamik

Die Abb. 10.4-22 zeigt, dass die Synchronisation mehrerer Kanten auch ohne Synchronisationsbalken modelliert werden kann. In beiden Fällen werden die eingehenden Marken zu einer neuen Marke verschmolzen. Man spricht bei der linken Darstellung auch von einem impliziten UND, während die mittlere ein explizites UND darstellt. Anders sieht es bei einer Zusammenführung aus. Hier wird entweder die Marke der einen oder die Marke der anderen eingehenden Kante weitergegeben (explizites ODER).

Abb. 10.4-22:
Unterschiedlicher
Markenfluss bei
der Zusammen-
führung von
Kanten.

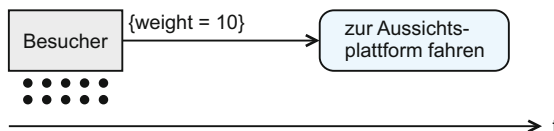


Gewichtete Kanten

In engem Zusammenhang mit dem Marken-Konzept steht die Notation der gewichteten Kanten, die für Objektflüsse definiert werden kann. Das Gewicht definiert, wie viele Marken vorliegen müssen, damit die nachfolgende Aktion ausgeführt wird.

Beispiel Die Abb. 10.4-23 modelliert, dass ein Aufzug erst dann zur Aussichtsplattform fährt, wenn mindestens 10 Besucher eingetroffen sind. Enthält eine Kante keine Gewichtung, dann gilt implizit $\{\text{weight} = 1\}$. Die Gewichtung $\{\text{weight} = 10\}$ spezifiziert, dass der nachfolgenden Aktion alle vorliegenden Marken angeboten werden.

Abb. 10.4-23:
Gewichtete
Kanten.



10.4.8 Strukturelemente und Strukturen von Petri-Netzen

Petrinetze kann man aus zehn Strukturelementen zusammensetzen (Abb. 10.4-24).

Aus diesen Grundelementen ergeben sich Strukturen (Abb. 10.4-25).

Mit Hilfe dieser Strukturen lassen sich typische Anwendungsmuster modellieren (Abb. 10.4-26, Abb. 10.4-27 und Abb. 10.4-28).

10.4 Petrinetze III

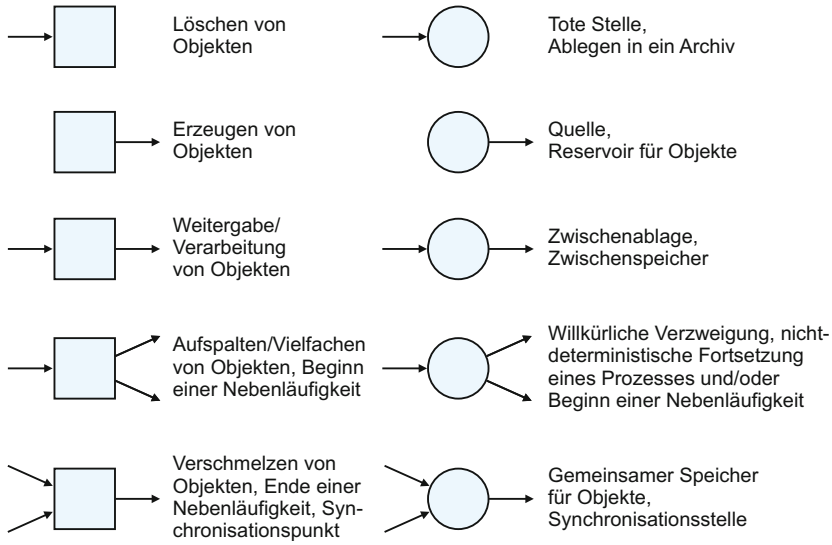


Abb. 10.4-24:
Strukturelemente
von Petri-Netzen.

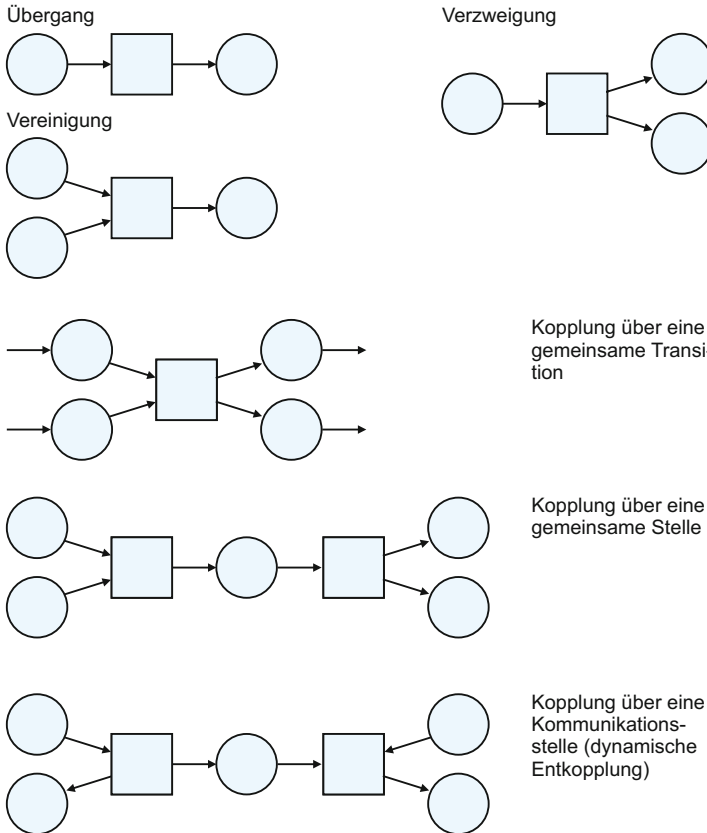
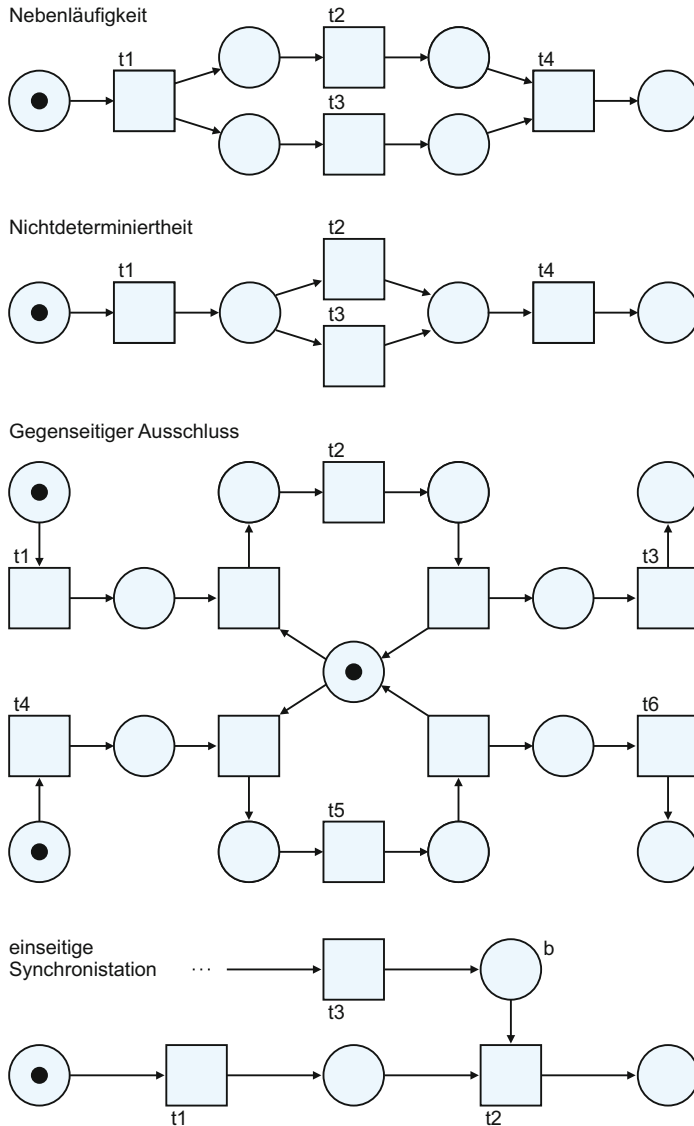


Abb. 10.4-25:
Aus den
Grundelementen
herleitbare
Strukturen.

III 10 Dynamik

Abb. 10.4-26:
Typische Anwendungsmuster für Petrinetze.



10.4.9 Box: Petrinetze – Methode

Eine allgemein anerkannte Methode zur Erstellung von Petrinetzen gibt es nicht. Ist ein umfangreiches Problem zu modellieren, dann empfiehlt es sich, schrittweise vorzugehen und ein hierarchisches Petrinetz zu verwenden.

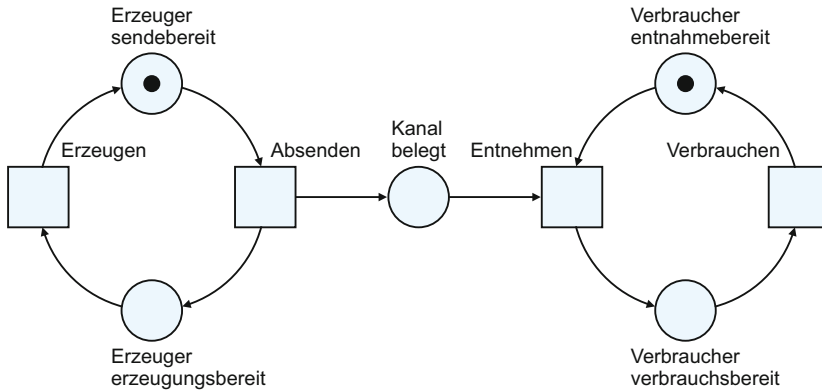


Abb. 10.4-27:
Produzenten und
Konsumenten bzw.
Erzeuger und
Verbraucher im
Petrinetz.

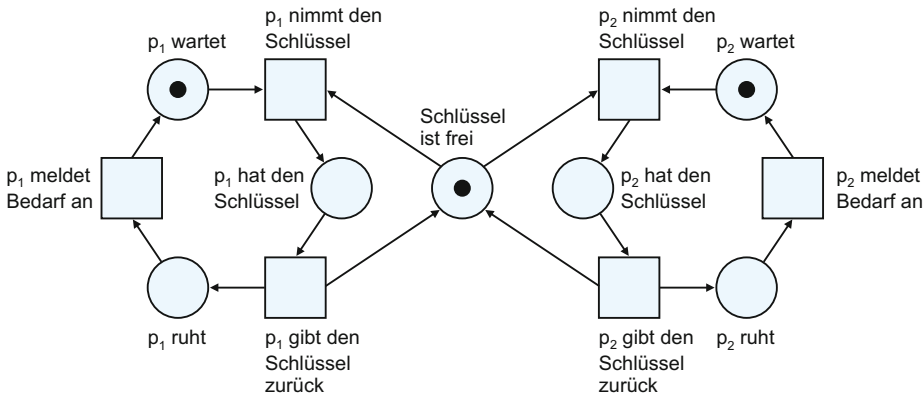


Abb. 10.4-28:
Leser und
Schreiber im
Petrinetz
(gegenseitiger
Ausschluss).

1. Schritt: Aktive & passive Komponenten identifizieren

In der Problemstellung werden daher zunächst Instanzen (=Transitionen) und Kanäle (=Stellen) auf einem hohen Abstraktionsniveau identifiziert. Instanzen repräsentieren aktive Komponenten, Kanäle passive Komponenten.

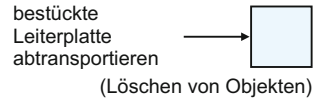
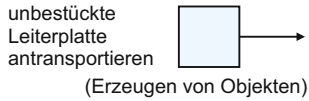
Aktive Komponenten tun etwas. Sie können Objekte erzeugen, transportieren oder verändern. Daher beschreibt man sie durch Verben. Ist es möglich, die erzeugten, transportierten oder veränderten Objekte anzugeben, dann sollte vor dem Verb ein Substantiv oder ein Adjektiv gefolgt von einem Substantiv stehen. Liegt eine verbale Problembeschreibung vor, dann geben Verben Hinweis auf mögliche aktive Komponenten.

Die im Kapitel »Strukturelemente und Strukturen von Petri-Netzen«, S. 320, angegebenen ersten fünf Strukturelemente helfen bei der Zuordnung des richtigen Elementtyps.

Man sollte zunächst die Schnittstellen des Systems mit seiner Umwelt modellieren.

III 10 Dynamik

Abb. 10.4-29:
Beispiel zur
Schnittstellen-
modellierung.



Beispiel 1a Analysiert man die Problembeschreibung des Beispiels »Roboter« (siehe »Bedingungs/Ereignis-Netze«, S. 305), dann werden die Leiterplatten vom Fließband A antransportiert und vom Fließband B abtransportiert (Abb. 10.4-29). Da es für diese Problemstellung uninteressant ist, was vor dem Antransport und nach dem Abtransport mit den Leiterplatten geschieht, ist diese Schnittstellenmodellierung problemgerecht.

Passive Komponenten können Objekte lagern, speichern oder sichtbar machen. Sie können sich in bestimmten Zuständen befinden. Man beschreibt sie durch Substantive oder durch Aussagen, die ihren Zustand angeben. Zustandsaussagen sollten so formuliert werden, dass sie mit Ja oder mit Nein beantwortet werden können. Verbale Problembeschreibungen geben durch Substantive und Zustandsaussagen Hinweise auf passive Komponenten. Die im Kapitel »Strukturelemente und Strukturen von Petri-Netzen«, S. 320, angegebenen letzten fünf Strukturelemente helfen bei der Auswahl des richtigen Elementtyps.

Abb. 10.4-30:
Identifizierte
passive
Komponenten.



Beispiel 1b Anhand der Problembeschreibung lassen sich passive Komponenten identifizieren (Abb. 10.4-30). Es ergeben sich weitere aktive Komponenten (Abb. 10.4-31).

Abb. 10.4-31:
Identifizierte
aktive
Komponenten.



Die Pfeile beschreiben eine abstrakte, gedankliche Beziehung zwischen passiven und aktiven Komponenten oder umgekehrt. Bei den Beziehungen kann es sich z.B. um logische Zusammenhänge, Zugriffsrechte, räumliche Nähe oder eine unmittelbare Kopplung handeln.

2. Schritt: Beziehungen ermitteln

Im zweiten Schritt müssen nun zwischen den Komponenten diese Beziehungen ermittelt werden. Bei der Art der Beziehung helfen wieder die Strukturelemente und Strukturen. In verbalen Problembeschreibungen ist dabei auf Formulierungen wie »nichtdeterministische Fortsetzung«, »Vereinigung«, »Verzweigung«, »Synchronisation«, »Nebenläufigkeit« u.ä. zu achten.

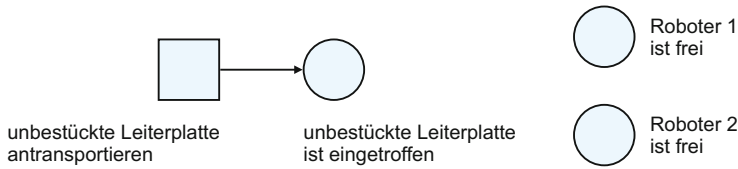


Abb. 10.4-32:
Komponenten zur
Modellierung des
Bestückungsauto-
maten.

In der Problemstellung heißt es: »Ist ein Roboter frei, dann kann er eine antransportierte Leiterplatte vom Fließband nehmen. Sind beide Roboter frei, dann wird nichtdeterministisch entschieden, welcher Roboter die Leiterplatte nimmt«. Zur Modellierung dieser Situation liegen bereits Komponenten vor (Abb. 10.4-32).

Beispiel 1c

Wenn eine unbestückte Leiterplatte eingetroffen ist, dann wird nichtdeterministisch auf Roboter 1 oder Roboter 2 verzweigt (Abb. 10.4-33).

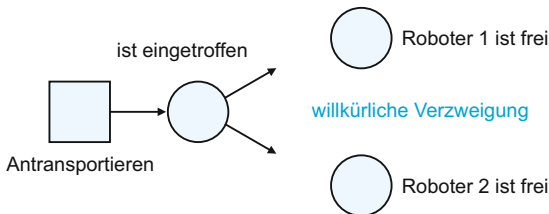


Abb. 10.4-33:
Nichtdeterministi-
sche Verzweigung
beim Eintreffen
einer Leiterplatte.

Ist eine Leiterplatte eingetroffen und ein Roboter frei, dann wird die Leiterplatte vom Roboter ergriffen und anschließend bestückt. Der Roboterarm und die Leiterplatte verschmelzen sozusagen in einem Objekt (Abb. 10.4-34).

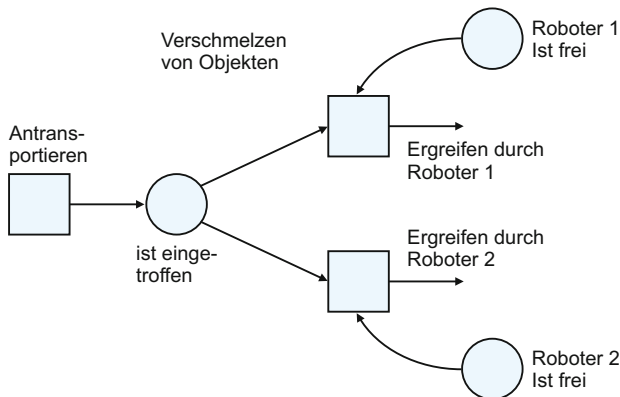


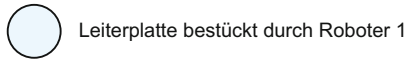
Abb. 10.4-34:
Verschmelzen von
Roboterarm und
Leiterplatte.

An dieser Stelle stößt man auf eine Schwierigkeit. Im Anschluss an das Ergreifen muss eine passive Komponente folgen. Der Roboter bestückt aber die Leiterplatte. Dies ist eine Aktivität und würde eine aktive Komponente erfordern. In Petrinetzen müssen sich aber aktive und passive Komponenten immer abwechseln. Da die Bestückung

III 10 Dynamik

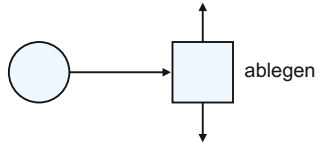
aber laut Problemstellung selbst nicht betrachtet wird, wird nur der Zustand formuliert, ob die Leiterplatte bestückt ist oder nicht (Abb. 10.4-35).

Abb. 10.4-35:
Leiterplatte
bestückt?



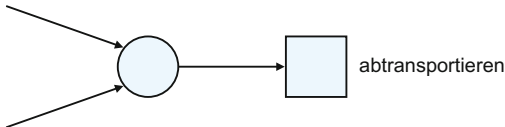
Nachdem die Bestückung erfolgt ist, kann die Leiterplatte abgelegt werden; es erfolgt eine Aufspaltung von Objekten (Roboterarm und Leiterplatte werden getrennt) (Abb. 10.4-36).

Abb. 10.4-36:
Aufspalten von
Objekten.



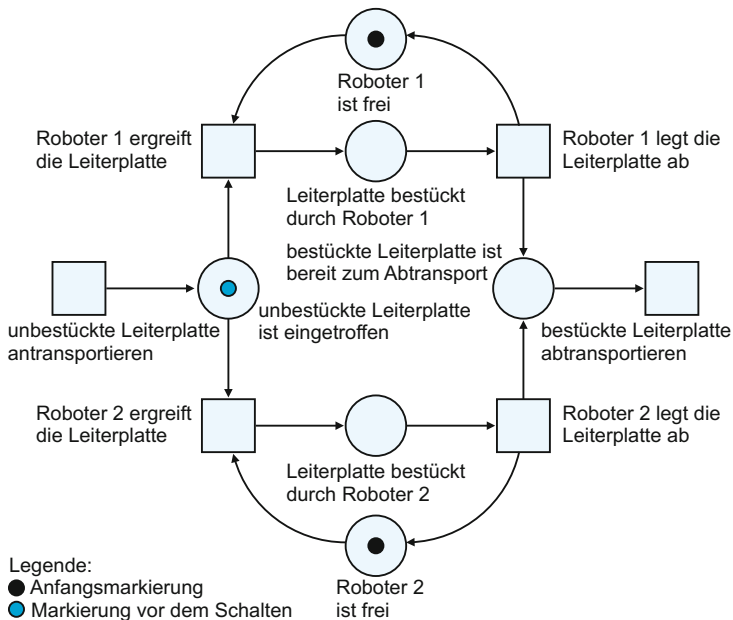
Die getrennt bearbeiteten Leiterplatten werden anschließend wieder auf dem Transportband B vereinigt (Abb. 10.4-37).

Abb. 10.4-37:
Vereinigen von
Objekten.



Es ergibt sich das Netz der Abb. 10.4-38 (siehe auch »Bedingungs/Ereignis-Netze«, S. 305).

Abb. 10.4-38: B/E-
Netz des Be-
stückungsroboters
vor dem Schalten.



3. Schritt: Verfeinerung & Ergänzung

Nach dem Erstellen eines Kanal-Instanzen-Netzes kann man iterativ folgendermaßen fortfahren:

- Verfeinerung von Instanzen und Kanälen,
- Ergänzung des bis dahin erstellten Netzes.

4. Schritt: Festlegung der Objekte

Im Anschluss daran ist zu überlegen, welche konkreten Objekte die Kanäle bzw. Stellen beinhalten können. Reichen anonyme Objekte zur Modellierung aus oder werden individuelle Objekte benötigt? Diese Überlegungen führen zu möglichen Netztypen.

5. Schritt: Überlegungen zu Schaltregeln & Schaltwirkungen

Anschließend ist zu überlegen, welche Schaltregeln und Schaltwirkungen zur Modellierung nötig sind.

6. Schritt: Netztyp festlegen

Aus den Objekt- und Schaltüberlegungen ergibt sich ein benötigter Netztyp.

7. Schritt: Anfangsmarkierung festlegen

Entsprechend dem festgelegten Netztyp sind die Objekte, Pfeilbeschriftungen, Schaltregeln und Schaltwirkungen festzulegen.

8. Schritt: Analyse, Simulation

Dann ist eine Anfangsmarkierung zu überlegen und das Netz zu analysieren oder zu simulieren.

10.4.10 Analyse und Simulation von Petrinetzen

Nach der Beschreibung eines Systems als Petrinetz kann es unter verschiedenen Gesichtspunkten analysiert werden. Typische Fragestellungen lauten:

- *Terminiert* das Netz, d. h., können, ausgehend von einer Anfangsmarkierung, stets nur endlich viele Transitionen schalten?
- Ist jede Transition *lebendig*, d. h., können, ausgehend von einer Anfangsmarkierung, die Transitionen stets so schalten, dass eine vorgegebene Transition t im weiteren Verlauf nochmals schalten kann?
- Treten vermeidbare *Verklemmungen* auf, d. h., gibt es Situationen, in denen keine Transition schalten kann, die aber bei anderer Schaltreihenfolge hätten vermieden werden können?

Da viele solcher Fragestellungen algorithmisch schwierig zu beantworten sind, werden Petrinetze oft simuliert, um das Verhalten zu studieren.

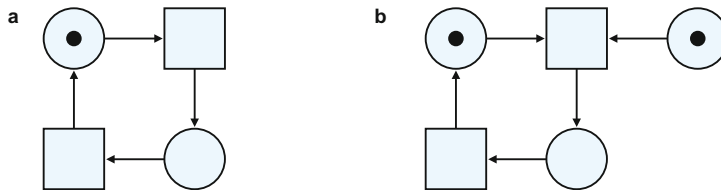
Betrachtet man S/T-Netze, dann kann man verschieden hohe Anforderungen an die **Lebendigkeit** stellen.

III 10 Dynamik

Wenn in einem S/T-Netz der Zustand eintreten kann, dass keine Transition aufgrund leerer Eingangsstellen oder wegen voller Ausgangsstellen schalten kann, dann ist das Netz todesgefährdet, sonst ist es lebendig. Anders ausgedrückt: Ein S/T-Netz mit der Markierung M ist lebendig, wenn es mindestens eine Transition gibt, die schalten kann, und wenn für jede solche Transition das Netz mit der entstehenden Folgemarkierung wieder lebendig ist.

Beispiel Die Abb. 10.4-39 zeigt ein lebendiges und ein todesgefährdetes Petrinetz. Das Netz **a** ist lebendig, da die Transitionen immer abwechselnd schalten. Das Netz **b** ist todesgefährdet, da nach dem zweiten Schalten keine weitere Transition mehr schalten kann.

Abb. 10.4-39:
Lebendige und
todesgefährdete
Petrinetze.



Eine stärkere Lebendigkeitsforderung wäre, dass jede Transition immer schalten können muss oder dass sie immer wieder schalten können muss.

Lebendige Netze sorgen dafür, dass es weder zu einem Mangel an Marken noch zu einem Überfluss kommt. Ein System, das im Endlosbetrieb laufen soll, kann nur durch ein lebendiges Netz beschrieben werden.

Bei Lebendigkeitsuntersuchungen sind Netzteile relevant, die nie markiert werden oder die nie alle Marken verlieren.

Verklemmung Eine Stellenmenge S heißt **Verklemmung** (deadlock), wenn sie – einmal ohne Marken – nie mehr markiert werden kann.

Beispiel Schaltet t_1 (Abb. 10.4-40), dann kann auch t_3 schalten. Schaltet t_2 , dann kann auch t_4 schalten. Schalten jedoch t_1 und t_2 gleichzeitig, dann liegt eine Verklemmung vor. In S_2 und S_3 liegt jeweils eine Marke. S_1 ist leer. t_3 und t_4 können niemals mehr schalten.

10.4.11 Wertung

Petrinetze eignen sich besonders gut zur Modellierung von Systemen mit kooperierenden Prozessen. Das Anwendungsspektrum umfasst daher insbesondere diskrete, ereignisorientierte, verteilte Systeme. In jüngerer Zeit werden sie auch für die Vorgangsmodellierung von Bürovorgängen eingesetzt (*work flow*).

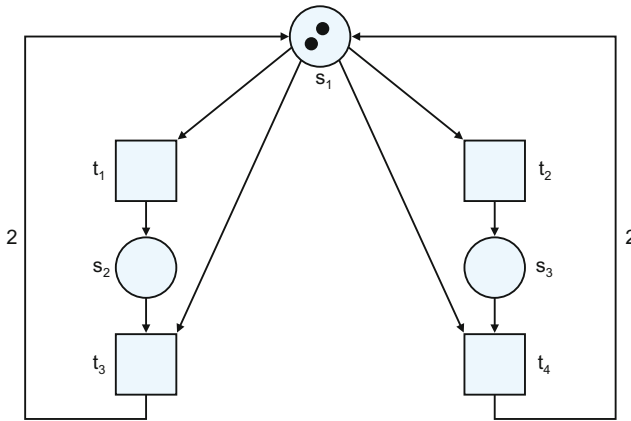


Abb. 10.4-40:
Petrinetz mit
Verklemmung.

Gewisse Ähnlichkeiten gibt es mit Zustandsautomaten. Die Stellen lassen sich als Zustände, die Transitionen als Zustandsänderung interpretieren (siehe »Zustandsautomaten«, S. 269).

Parallelen zu
Zustands-
automaten

Ein durch einen Zustandsautomaten beschriebenes System befindet sich zu jedem Zeitpunkt in genau einem Zustand, außer bei nebenläufigen Harel-Automaten (siehe »Zustandsautomat nach Harel«, S. 277).

Ein durch ein Petrinetz beschriebenes System kann sich zu einem Zeitpunkt aber in mehreren Zuständen, dargestellt durch die aktuelle Markenbelegung, befinden. Je nach Struktur des Petrinetzes können in einem System unabhängig voneinander Zustandsübergänge erfolgen. Synchronisationen zwischen nebenläufigen Systemen können durch eine geeignete Netzstruktur erzwungen werden. Außerdem kann Nichtdeterminismus modelliert werden.

Petrinetze besitzen gegenüber Zustandsautomaten eine größere Mächtigkeit und erlauben die Modellierung von Problemklassen, für die Zustandsautomaten nicht geeignet sind.

B/E-Netze sind gut geeignet für die Beschreibung des Kontrollflusses bei kooperierenden Prozessen. Dabei wird der augenblickliche Zustand jedes Prozesses durch eine Marke repräsentiert.

S/T-Netze erweitern die Modellierungsmöglichkeiten, da beliebig viele, aber weiterhin anonyme Marken pro Stelle abgelegt werden können.

Pr/T-Netze sind übersichtlicher als S/T-Netze. Die Modellierungsmächtigkeit ist größer, zumal auch beliebige, kontinuierliche Funktionen benutzt werden können, wie $\text{Höhe} > 4000 \text{ m}$ als Schaltbedingung.

Zeitbehaftete Transitionen können durch stochastische Petrinetze (SPN), zeitlose Transitionen und hemmende Kanten durch generalisierte stochastische Petrinetze (GSPN) dargestellt werden. Nachteilig ist die schwierigere Handhabung dieser Netze. Es ist nicht einfach, korrekte Schaltbedingungen und Schaltwirkungen zu entwerfen.

III 10 Dynamik

Generell gilt, je größer die Modellierungsmächtigkeit, desto geringer die Analysemöglichkeiten.

Als vorteilhaft erweisen sich folgende Punkte:

- + Petrinetze bestehen aus wenigen und einfachen Elementen.
- + Petrinetze sind grafisch gut darstellbar.
- + Die Marken erlauben eine gute Visualisierung des jeweiligen Systemzustands.
- + Petrinetze besitzen ein solides theoretisches Fundament.
- + Petrinetze können – in einem beschränkten Rahmen – analysiert und simuliert werden.
- + Es gibt mehrere Petrinetz-Werkzeuge, die die Erstellung, Analyse, Simulation und Code-Generierung erlauben.
- + Petrinetze sind das einzige, weit verbreitete Basiskonzept zur Modellierung kooperierender Prozesse.

Nachteilig sind:





- Für praktische Anwendungen müssen höhere Petrinetze eingesetzt werden, für die es keine einheitliche Notation gibt.
- Höhere Petrinetze sind schwer zu erstellen und zu analysieren.
- Petrinetze sind mit anderen Basiskonzepten bisher kaum kombiniert worden, d. h., sie bilden ein weitgehend für sich stehendes Konzept.
- Petrinetze besitzen eine statische Struktur. Will man z. B. Vorgänge beschreiben, bei denen dauernd neue Prozesse erzeugt werden (z. B. task-Konzept in Ada), dann erweisen sich Petrinetze als ungeeignet.
- Es gibt keine allgemeine Methode, wie man Petrinetze erstellt.

10.4.12 Zusammenfassung

Petrinetze ermöglichen die Modellierung, Analyse und Simulation nebenläufiger Systeme. In Abhängigkeit von dem zu modellierenden System können verschieden mächtige Klassen von Petrinetzen eingesetzt werden. Es werden einfache Petrinetze (Bedingungs/Ereignis-Netze, Stellen/Transitions-Netze) und höhere Petrinetze (Prädikat/Transitions-Netze, zeitbehaftete Petrinetze, stochastische Petrinetze) unterschieden. Wesentliche Unterschiede zwischen diesen Netztypen zeigt die Abb. 10.4-41.

Stellen und Transitionen werden je nach Netztyp unterschiedlich bezeichnet und interpretiert. Die Stellen unterscheiden sich im Wesentlichen dadurch, ob sie eine oder mehrere Marken enthalten können und ob ein Zeitintervall angegeben ist, das festlegt, wie lange die Marken auf der Stelle verharren müssen.

Für B/E- und S/T-Netze gibt es jeweils eine allgemeine Schaltregel für Transitionen. Bei Pr/T-Netzen kann jede Transition mit einer Schaltbedingung und einer Schaltwirkung versehen werden.

Allgemeine Bezeichnung	grafische Repräsentation	B/E - Netze	S/T - Netz	Pr/T - Netz	Zeitbehaftetes Netz	Hierarchisches Netz
Stelle		Bedingung <i>markiert oder unmarkiert</i>	Stelle <i>Kapazität > 0</i>	Prädikat <i>Kapazität > 0</i>	Stelle <i>Zeitintervall</i>	Kanal
Transition	 <i>oder</i>	Ereignis	Transition	Ereignis <i>Schaltbedingung & Schaltwirkung</i>	Transition <i>Zeitintervall</i>	Instanz
Pfeil (Flussrelation)		ungewichtet	gewichtet	gewichtet <i>konstante und/ oder variable Beschriftung</i>	je nach Netztyp	je nach Netztyp
Marke		Marke <i>uniform</i>	Marke <i>uniform</i>	Objekt <i>individuell</i>	je nach Netztyp	je nach Netztyp

In zeitbehafteten Netzen können Zeitintervalle an Transitionen notiert werden. In stochastische Petrinetzen (SPN) wird die Schalt- oder Verzögerungszeit einer Transition durch eine exponentiell verteilte Zufallsvariable beschrieben. Generalisierte stochastische Petrinetze (GSPN) liegen vor, wenn stochastische Petrinetze um zeitlose Transitionen und hemmende Kanten erweitert werden.

Die Pfeile, die Stellen und Transitionen miteinander verbinden, können ungewichtet, gewichtet und mit Konstanten oder Variablen beschriftet sein. Marken können uniform oder individuell sein.

Jeden Netztyp kann man als hierarchisches Petrinetz strukturieren. Die oberen Netzebenen heißen dann auch Kanal-Instanzen-Netze.

Die UML-Aktivitätsdiagramme basieren semantisch auf Petrinetzen.

Ein Petrinetz kann auf verschiedene Eigenschaften hin analysiert werden. Wichtig sind die Lebendigkeit und Verklemmungsfreiheit eines Netzes.

*Abb. 10.4-41:
Klassifikation von
Petrinetzen.*

Zur Klassifikation

- Petrinetze werden sowohl textuell als auch grafisch (mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von semiformal bis formal.
- Sie werden vorwiegend in der Spezifikation eingesetzt.
- Sie werden vorwiegend in technischen und softwareintensiven Anwendungen verwendet.

10.5 Szenarien

Bei allgemeinen Modellierungen – insbesondere auf der Typebene – ist es oft schwierig, sich zeitliche Abläufe vorzustellen und auf Richtigkeit zu überprüfen. Daher ist es nützlich, sich zeitliche Schnappschüsse von der Dynamik eines Systems – insbesondere auf der Exemplarebene – zu veranschaulichen und dann zu überprüfen. Man spricht in diesem Zusammenhang oft von der Erstellung eines Szenarios oder von Szenarien.

Definition

Szenario: hypothetische Aufeinanderfolge von Ereignissen, die zur Beachtung kausaler Zusammenhänge konstruiert wird [...] [Frem07].



Die UML stellt für das Modellieren von Szenarien verschiedene Diagrammtypen zu Verfügung.

Mit Hilfe eines UML-Sequenzdiagramms ist es möglich, den zeitlichen Ablauf von Methoden und die Kommunikation mit Akteuren für einen oder mehrere Abläufe darzustellen:

- »Sequenzdiagramm«, S. 333

Mit Hilfe eines UML-Kommunikationsdiagramms ist es möglich, Beziehungen zwischen Objekten sowie die Reihenfolge und Verschachtelung von Operationen, die diese Objekte benutzen, darzustellen:

- »Kommunikationsdiagramm«, S. 343

Für die Erstellung und Qualitätsüberprüfung können für Sequenz- und für Kommunikationsdiagramme eine Methode und eine Checkliste angegeben werden:

- »Box: Sequenz- und Kommunikationsdiagramm – Methode und Checkliste«, S. 346

Für die Fälle in denen das exakte zeitliche Verhalten und die zeitliche Synchronisation der Zustandsübergänge (z. B. von Objekten) wichtig ist, können UML-Zeitverlaufsdiagramme verwendet werden:

- »Timing-Diagramm«, S. 352

Zum Abschluss wird ein Überblick gegeben:

- »Zusammenfassung«, S. 355

In der UML werden unter dem Oberbegriff *Interactions* [OMG09a, S. 459 ff.] folgende Diagrammart und Tabellen aufgeführt:

- **Sequence Diagram**

- **Communication Diagram**

- **Timing Diagram**

- *Interaction Overview Diagram*

- *Interaction Table*

Auf die letzten beiden wird hier *nicht* eingegangen.

10.5.1 Sequenzdiagramm

Ein **UML-Sequenzdiagramm** (*sequence diagram*, abgekürzt **sd**) zeigt die Interaktion zwischen mehreren Kommunikationspartnern. Oft handelt es sich bei diesen Partnern um die Objekte von Klassen. Dann wird anstelle von Kommunikationspartnern von Objekten gesprochen, die miteinander kommunizieren. Jeder Kommunikationspartner wird durch ein Rechtecksymbol zusammen mit einer gestrichelten Linie dargestellt: der **Lebenslinie** (*lifeline*). In das Rechteck wird der Name des Kommunikationspartners eingetragen. In der Abb. 10.5-1 findet eine Interaktion zwischen drei Objekten statt, die zu den Klassen C1, C2 und C3 gehören. Sie wird initiiert durch einen Benutzer. Bei diesem Kommunikationspartner handelt es sich um einen Akteur. Die Namen der Kommunikationspartner werden bei diesem Sequenzdiagramm mit dem Klassennamen bezeichnet, dem ein Doppelpunkt vorausgeht. Alternativ könnten auch Objekt-namen gewählt werden. Das Sequenzdiagramm besitzt zwei Dimensionen: Die Vertikale repräsentiert die Zeit (von oben nach unten), auf der Horizontalen werden die Interaktionen zwischen den Kommunikationspartnern in Form von Nachrichten (*messages*) bzw. Botschaften eingetragen. Jede Nachricht wird durch einen Pfeil vom Sender zum Empfänger dargestellt.

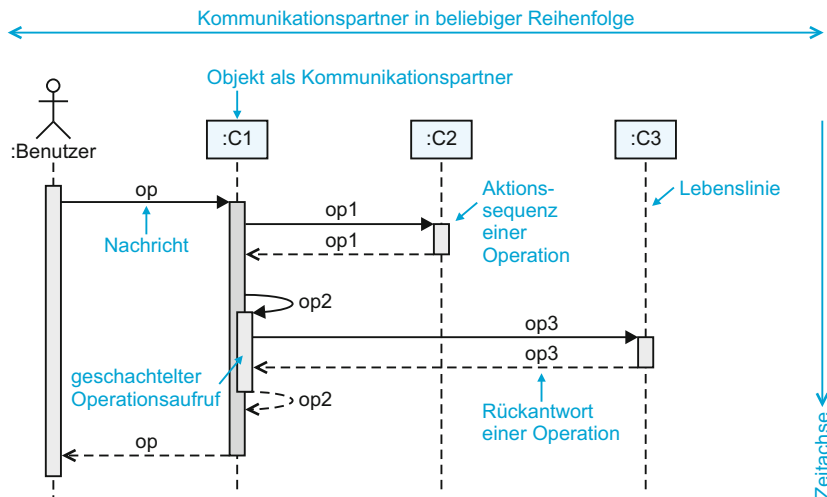


Abb. 10.5-1:
Notation Sequenz-
diagramm.

In der Abb. 10.5-1 handelt es sich bei der Nachricht um den Aufruf einer Operation (*method call*), die von dem jeweiligen Empfängerobjekt ausgeführt wird. Operationsaufrufe werden mit einer gefüllten Pfeilspitze dargestellt. In diesem Fall wird der Name der gerufenen Operation an den Pfeil angetragen. Die Zeitspanne, in der eine Operation vom Empfänger ausgeführt wird, wird in der Abb. 10.5-1 als **Aktionsequenz** bezeichnet und als längliches Rechteck auf der

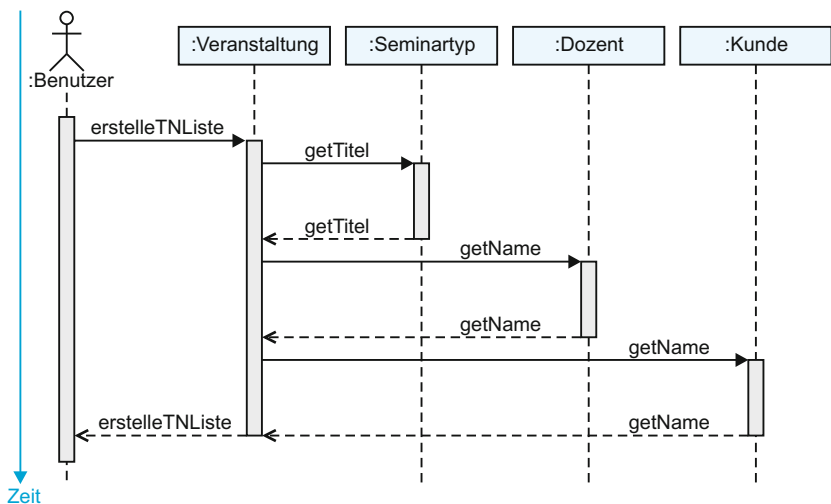
III 10 Dynamik

Lebenslinie angetragen. Nach Beendigung der Operation kann eine Rückantwort (*reply message*) an das Senderobjekt geschickt werden. Diese Rückantwort wird als gestrichelte Linie modelliert und mit dem Namen der gerufenen Operation beschriftet. Operationsaufrufe können auch »geschachtelt« werden. In der Abb. 10.5-1 ruft die Operation op die Operation op2 auf, die ebenfalls auf ein Objekt der Klasse C1 angewandt wird.

Die Abb. 10.5-1 beschreibt somit folgendes Szenario. Der Akteur mit dem Namen Benutzer schickt die Nachricht op an ein Objekt der Klasse C1, die dort eine gleichnamige Operation aufruft. Die Operation op sendet dann zuerst die Nachricht op1 an ein Objekt der Klasse C2 und dann die Operation op2 an ein Objekt der Klasse C1. Die Operation op2 sendet wiederum eine Nachricht op3 an ein Objekt der Klasse C3. Nach Beendigung einer jeden Operation geht der Kontrollfluss zurück an den Aufrufer.

Beispiel: Die Abb. 10.5-2 veranschaulicht, wie eine Teilnehmerliste erstellt wird. Am linken Rand der Abbildung wird die Zeitachse dargestellt, die jedoch nicht zum UML-Diagramm gehört.

Abb. 10.5-2:
Zeitlicher Ablauf
bei der Erstellung
einer
Teilnehmerliste.



Nachrichten Die Interaktion zwischen zwei Kommunikationspartnern wird in der UML durch eine synchrone oder eine asynchrone Nachricht dargestellt.

Bei der **synchronen Nachricht** wartet der Sender, bis der Empfänger die geforderte Verarbeitung komplett durchgeführt hat. Der Empfänger schickt daraufhin dem Sender eine Antwortnachricht, die implizit das Ende der geforderten Verarbeitung mitteilt und außerdem Antwortdaten enthalten kann. Synchrone Nachrichten sind oft

Operationsaufrufe, die die Ausführung der entsprechenden Methode beim Empfänger auslösen. Es ist aber auch möglich, dass synchrone Nachrichten durch Signale modelliert werden.

Bei der **asynchronen Nachricht** wartet der Sender *nicht* auf das Verarbeitungsende durch den Empfänger, sondern setzt parallel dazu seine eigene Verarbeitung fort. Es entsteht eine nebenläufige Verarbeitung. Asynchrone Nachrichten werden immer durch **Signale** realisiert.

Synchrone Nachrichten werden durch einen Pfeil mit gefüllter Pfeilspitze, asynchrone durch einen Pfeil mit offener Pfeilspitze angegeben. Die Rückantwort einer synchronen Nachricht ist ein gestrichelter Pfeil (Abb. 10.5-3). Unabhängig davon, ob es sich um eine synchrone oder asynchrone Nachricht handelt, kann die Aktionssequenz eingetragen werden, um zu verdeutlichen, wie lange der jeweilige Kommunikationspartner aktiv ist.

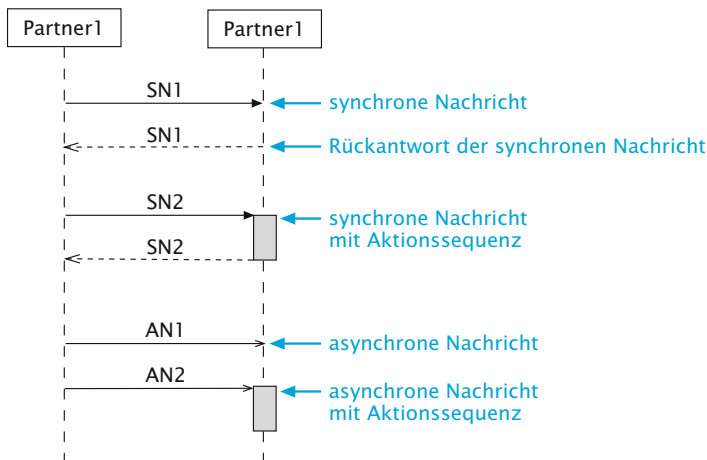


Abb. 10.5-3:
Nachrichten im Se-
quenzdiagramm.

Sowohl Operationsaufrufe als auch Signale werden durch einen Namen spezifiziert. Klammern werden – bei Operationsaufrufen und Signalen – nur angegeben, wenn durch die Nachricht Daten übergeben werden. Bei einer Antwortnachricht wird der Name der ursprünglichen Aufrufnachricht wiederholt.

Beschriftung der
Nachrichten

Um ein Sequenzdiagramm in anderen Diagrammen referenzieren zu können, kann ein Sequenzdiagramm – wie auch bei den anderen Diagrammartentypen der UML – durch einen Rahmen umgeben werden, in dessen linken oberen Ecke der Name des Diagramms eingetragen wird. Davor wird **sd** für **sequence diagram** eingetragen.

Umrandung

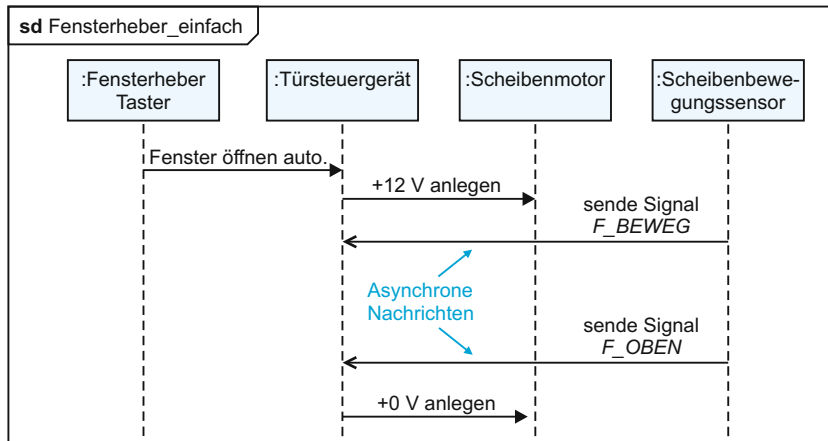
Das Sequenzdiagramm beschreibt das vollautomatische Öffnen des Türfensters. Die Interaktion beginnt mit der Übermittlung des Signals Fenster öffnen auto., welches das Türsteuergerät dazu veranlasst den Scheibenmotor anzuschalten. Der Scheibenbewegungssensor

Beispiel:
Fensterheber

III 10 Dynamik

registriert daraufhin die Bewegung der Scheibe und meldet dies dem Türsteuergerät. Nachdem das Fenster vollständig geschlossen wurde, übermittelt der Bewegungssensor auch diese Information, woraufhin das Türsteuergerät den Scheibenmotor abschaltet.

Abb. 10.5-4: Ein einfaches Sequenzdiagramm zur Fallstudie »Fensterheber«.



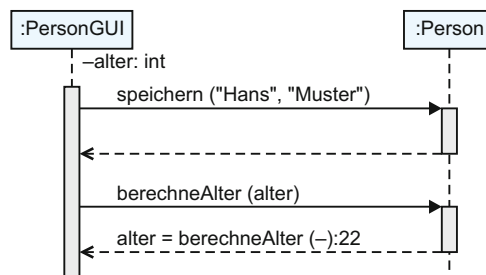
Parameter von Nachrichten

Mit vielen Nachrichten ist ein Datenaustausch verbunden. In der UML gibt es verschiedene und vielfältige Möglichkeiten Parameter zu annotieren [OMG09b, S. 495 ff.].

Beispiel In der Abb. 10.5-5 werden die folgenden Operationen der Klasse Person aufgerufen:

speichern (in vorname: String, in nachname: String)
berechneAlter (return alter: int)

Abb. 10.5-5: Parameterübergabe von Nachrichten.



Beim Operationsaufruf müssen die aktuellen Parameter in der gleichen Reihenfolge angegeben oder die formalen Parameter müssen mit aufgeführt werden. Die Operation speichern() besitzt in der Abb. 10.5-5 die aktuellen Parameterwerte "Hans" und "Muster".

Die Operation berechneAlter() besitzt nur einen Rückgabeparameter, der beim Operationsaufruf durch den formalen Ausgabeparameter alter angegeben wird. Für die Modellierung des Rückgabewerts bietet die UML verschiedene Möglichkeiten. Hier wird der Rückgabe-

wert dem Attribut *alter* des Kommunikationspartners zugewiesen. Hinter der runden schließenden Klammer wird durch einen Doppelpunkt getrennt der Ergebniswert – hier 22 – angegeben.

Umfangreiche Szenarien können auf mehrere Sequenzdiagramme aufgeteilt werden. Dazu bietet die UML die Notation der verlorenen (*lost message*) und gefundenen Nachrichten (*found message*). In der Abb. 10.5-6 wird im Sequenzdiagramm seq1 die Nachricht s2 gesendet, deren Empfänger nicht eingetragen ist. Bei der **verlorenen Nachricht** zeigt die Pfeilspitze dann *nicht* auf eine Lebenslinie, sondern auf einen Punkt. Dieser Begriff bedeutet also nicht, dass hier eine gesendete Nachricht verloren geht, sondern dass sich der Empfänger *nicht* im gleichen Diagramm befindet. Das Sequenzdiagramm seq2 enthält die Nachricht s2, deren Sender nicht eingetragen ist. Stattdessen geht der entsprechende Pfeil von einem Punkt aus. Hier liegt eine **gefundene Nachricht** vor.

Verlorene & gefundene Nachrichten

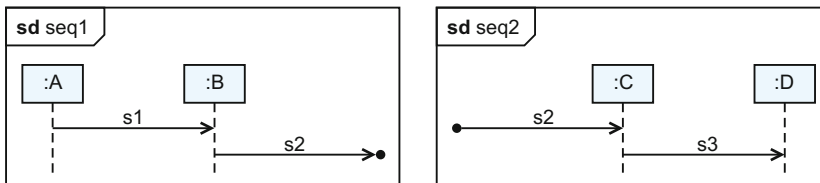


Abb. 10.5-6:
Verlorene und
gefundene
Nachrichten.

Ein Sequenzdiagramm bietet ebenfalls Notationselemente, die es erlauben eine **Zeitdauer** im Diagramm zu vermerken. Hierfür wird eine Lebenslinie an zwei Stellen um eine horizontale Hilfslinie erweitert. Zwischen die Hilfslinien zeichnet man einen Pfeil mit zwei offenen Spitzen und beschriftet diesen mit der gewünschten Zeitdauer. Die Angabe der Zeiteinheit kann direkt am Pfeil vorgenommen werden oder die Zeiteinheit wird global für die gesamte Interaktion in einer Notiz vermerkt.

Zeitdauer

Die Abb. 10.5-7 zeigt das aus der Abb. 10.5-4 bekannte Beispiel, erweitert um einige neue Notationselemente. Insbesondere ist das Modell um eine Angabe einer Zeitdauer erweitert worden. Das Sequenzdiagramm beinhaltet nun die Information, dass zwischen dem Senden der Nachricht *F_BEWEG* und *F_OBEN* zwischen null und drei Sekunden vergehen können (je nach Stellung der Fensterscheibe).

Beispiel:
Fensterheber

Weiterhin erlaubt die Sequenzdiagrammnotation das Modellieren von **Bedingungen** oder **Invarianten**, die während eines Szenarios gelten müssen. Zur Umsetzung bietet die UML mehrere Notationselemente an. Zum einen können Bedingungen in geschweiften Klammern auf einer Lebenslinie notiert werden. Dies bedeutet, dass die formulierte Bedingung gelten muss, sobald alle Ereignisse der Lebenslinie aufgetreten sind, welche zeitlich vor der Bedingung ste-

III 10 Dynamik

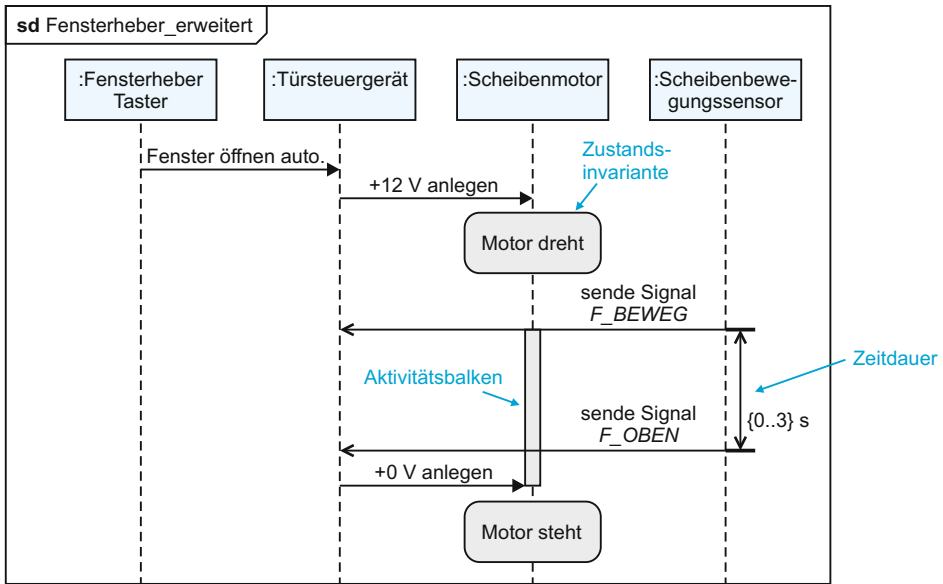


Abb. 10.5-7: Ein Sequenzdiagramm zur Fallstudie »Fensterheber«, erweitert um einige Notationselemente.

hen. Ist dies nicht der Fall, so ist die Interaktion ungültig und wird nicht weiter ausgeführt. Des Weiteren existiert die Möglichkeit **Zustandsinvarianten** zu definieren, in dem man die Lebenslinie um ein Zustandssymbol erweitert. Vergleichbar mit der Definition von Bedingungen in geschweiften Klammern bedeutet dies nun, dass sich das Element (welches durch die Lebenslinie repräsentiert wird) in dem abgebildeten Zustand befinden muss, wenn die Interaktion fortgesetzt wird.

Beispiel: Betrachtet man das Beispiel in Abb. 10.5-7, so sind hier zwei Zustände auf der Lebenslinie des Scheibenmotors modelliert. Durch diese Erweiterung stellt das Sequenzdiagramm nun die Forderung, dass sich der Scheibenmotor drehen muss, nachdem +12V angelegt wurden und wieder zum Stillstand kommen soll, nachdem die Spannung auf 0V gefallen ist.

Fragmente Mithilfe von kombinierten Fragmenten (*combined fragment*) können auch Kontrollstrukturen (siehe »Kontrollstrukturen«, S. 227) beschrieben werden. Ein kombiniertes Fragment wird wie ein Sequenzdiagramm mit einem rechteckigen Rahmen umgeben, in dem links oben der Interaktionsoperator eingetragen wird. Es werden hier folgende Operatoren betrachtet:

- opt: optionale Interaktion (if-then)
- alt: alternative Abläufe (if-then-else, switch)
- loop: Schleife (for, while-do, do-while)
- break: Interaktion bei Ausnahmebehandlung (exception, goto)
- par: Definition paralleler Interaktionssequenzen

Darüber hinaus können kritische Bereiche, Negationen, lose oder strenge Ordnung, Filter für unwichtige oder relevante Nachrichten sowie Zusicherungen (*assertions*) beschrieben werden.

Die Abb. 10.5-8 zeigt die kombinierten Fragmente für den alt-Operator und den loop-Operator.

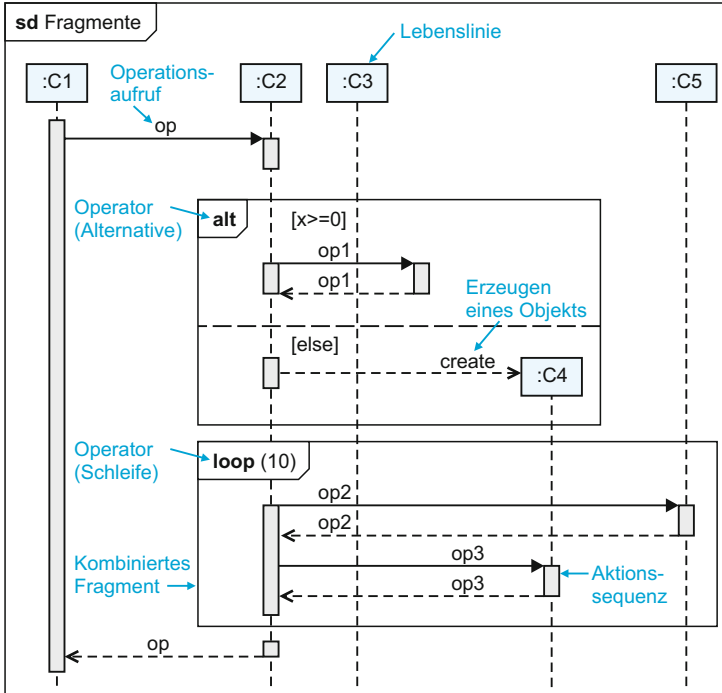


Abb. 10.5-8:
Sequenzdiagramm
mit kombinierten
Fragmenten.

In der Abb. 10.5-9 werden in Abhängigkeit, ob es sich um einen Neukunden oder einen Altkunden handelt, verschiedene Operationen aktiviert.

Beispiel:
SemOrg

Der opt-Operator kennzeichnet optionale Interaktionen. Sie werden ausgeführt, wenn die angegebene Bedingung erfüllt ist, und sonst übersprungen (Abb. 10.5-10).

opt-Operator

Der alt-Operator spezifiziert zwei oder mehr alternative Abläufe, die durch gestrichelte Linien getrennt sind (Abb. 10.5-10). Für jeden Ablauf wird die Bedingung in eckigen Klammern angegeben. Der Modellierer muss darauf achten, dass sich diese Bedingungen nicht überschneiden, d. h., es darf höchstens eine Alternative zutreffen. Fehlt die Bedingung, dann gilt implizit [true], d. h., die betreffende Alternative wird immer ausgeführt. Für eine Alternative kann [else] angegeben werden. Sie wird dann durchlaufen, wenn alle anderen Bedingungen nicht zutreffen.

alt-Operator

III 10 Dynamik

Abb. 10.5-9:
Unterschiedliche
Behandlung von
Alt- und
Neukunden.

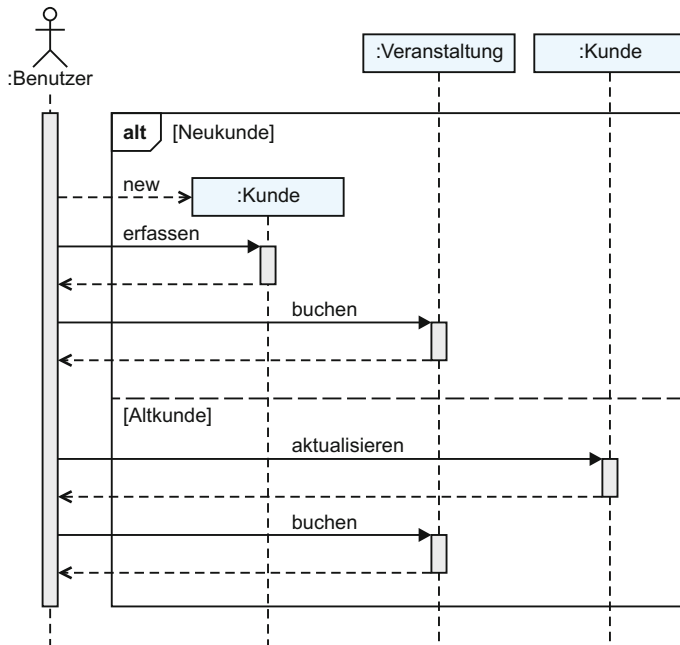
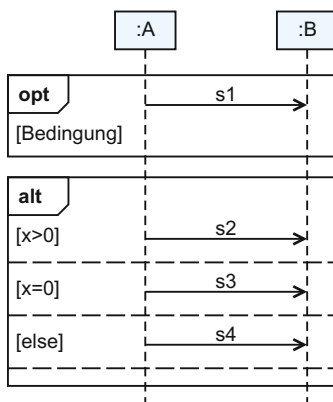


Abb. 10.5-10:
Kombinierte
Fragmente mit
opt- und alt-
Operator.



loop-Operator

Der loop-Operator spezifiziert, dass die Interaktionen wiederholt ausgeführt werden (Abb. 10.5-11). Er kann wie folgt spezifiziert werden:

- `loop (min,max)`, wobei `min` und `max` ganzzahlige Werte sind, die die Mindest- und Höchstzahl der Wiederholungen angeben. Es gilt `max` \geq `min`.
- `loop (min,*)`, wobei `*` für unendlich viele Wiederholungen steht, z.B. `loop (1,*)`.
- `loop (min)`, wobei `min` die Anzahl der Iterationen angibt, z.B. `loop(10)`.
- `loop`, wobei die Schleife 0 bis unendlich mal ausgeführt wird.

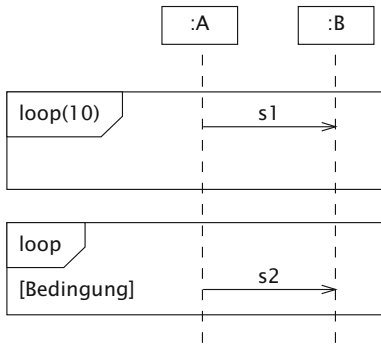


Abb. 10.5-11:
Kombinierte
Fragmente mit
loop-Operator.

Sowohl für min als auch für max können berechenbare Ausdrücke oder Variablen angegeben werden, z.B. loop (i), loop (i=1, i<=100). Zusätzlich zu min und max kann innerhalb des Fragments eine Iterationsbedingung angegeben werden. Solange sie mit true ausgewertet wird, wird die Schleife wiederholt.

Der break-Operator kennzeichnet Fragmente zur Ausnahmebehandlung. Wird die angegebene Bedingung mit true ausgewertet, dann werden die enthaltenen Interaktionen (Anweisungen zur Ausnahmebehandlung) ausgeführt und anschließend die umgebende Interaktion komplett verlassen (Abb. 10.5-12).

break-Operator

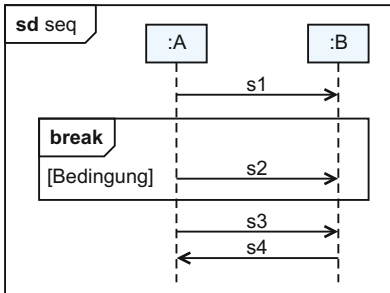


Abb. 10.5-12:
Sequenzdiagramm
mit break-
Operator.

Unter Zuhilfenahme des par-Operators lassen sich parallele Interaktionssequenzen definieren. Der par-Operator erlaubt es ebenfalls, mehrere voneinander separierte Abläufe zu definieren, in der Darstellung vergleichbar mit dem alt-Operator.

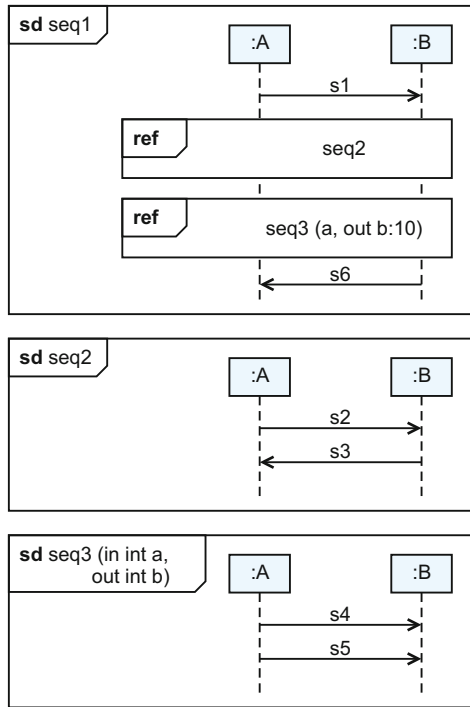
par-Operator

Die Interaktionsreferenz beschreibt eine Referenz auf ein anderes Sequenzdiagramm oder anderes Interaktionsdiagramm (Abb. 10.5-13). Häufig wiederkehrende Abläufe können durch ein eigenständiges Diagramm beschrieben und an beliebigen Stellen eingefügt werden. Wie beim Funktionsaufruf in Programmen können Parameter übergeben werden. Sie werden genau so spezifiziert wie beim Operationsaufruf.

Interaktions-
referenz

III 10 Dynamik

Abb. 10.5-13: Interaktionsreferenz ohne und mit Parameterübergabe.



Schachtelung Kombinierte Fragmente können ineinander geschachtelt werden. Die Abb. 10.5-14 zeigt, wie eine Schleife innerhalb einer Alternative verwendet wird.

Beispiel: Fensterheber Das Sequenzdiagramm in Abb. 10.5-15 modelliert mit Hilfe des loop-Operators, dass der Scheibenbewegungssensor über den kompletten Schließungsvorgang hinweg das Signal F_BEWEG sendet. Die modellierte Schleife kann null bis unendlich (angezeigt durch [0,*]) viele Durchläufe haben. Sie bricht ab, sobald das Fenster komplett geschlossen ist.

- Bewertung**
- + Sequenzdiagramme erlauben eine relativ detaillierte Darstellung einer Interaktion.
 - Die Vielzahl der Notationselemente und ihre zugehörige Semantik erschweren die Erlernbarkeit und die Lesbarkeit.
 - Die Erstellung eines Sequenzdiagramms ist zeitintensiv und nicht jeder Ablauf in einem komplexen System kann durch ein solches Diagramm modelliert werden.
 - Aus dem hohen Detailgrad ergibt sich ein hoher Wartungsaufwand, da es auch nach kleinen Änderungen oftmals nötig ist, die Konsistenz des Sequenzdiagramms wiederherzustellen.

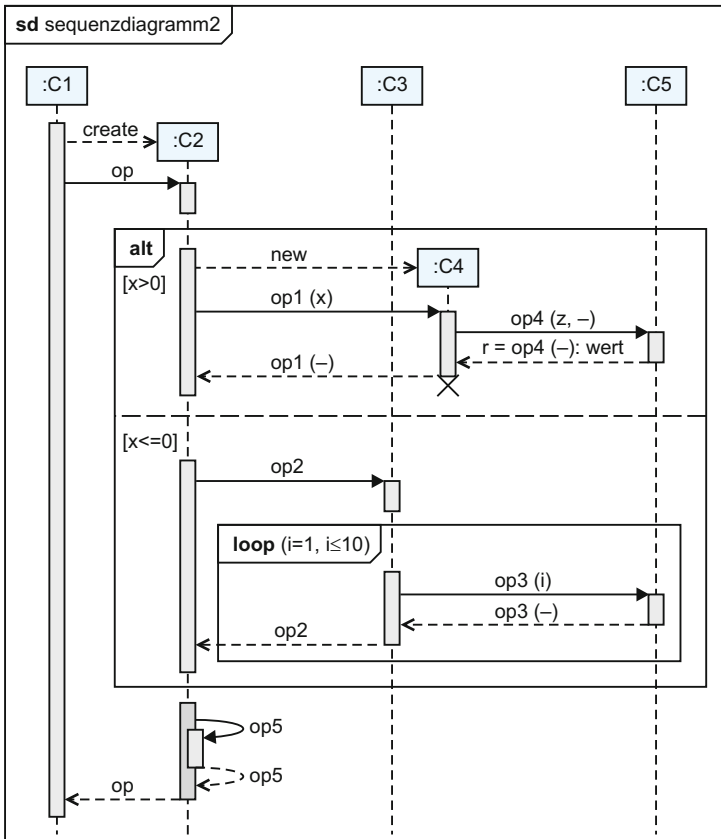


Abb. 10.5-14:
Sequenzdiagramm
mit geschachtelten
kombinierten
Fragmenten.

Manche Abläufe lassen sich durch Pseudocode einfacher und schneller veranschaulichen.

Empfehlung

10.5.2 Kommunikationsdiagramm

Ein **UML-Kommunikationsdiagramm** (*communication diagram*) erweitert ein **Objektdiagramm** um Nachrichten bzw. Botschaften. Es zeigt diejenigen Objekte, die für die Ausführung einer oder mehrerer Operationen relevant sind. Im Gegensatz zum Objektdiagramm modelliert es jedoch *nicht* einen Schnappschuss der Systemstruktur, sondern zeigt, wie Objekte für die Ausführung bestimmter Operationen zusammenarbeiten.

Während Sie bei der Erstellung des Sequenzdiagramms von vornherein über die Reihenfolge der Operationsaufrufe nachdenken müssen, haben Sie beim Kommunikationsdiagramm größeren Freiraum. Die Ausgangsbasis bilden die Objekte. Beispielsweise kann ein Objektdiagramm als Grundlage genommen werden (siehe »Assoziation«, S. 158). Die Reihenfolge der Operationen können Sie zum

III 10 Dynamik

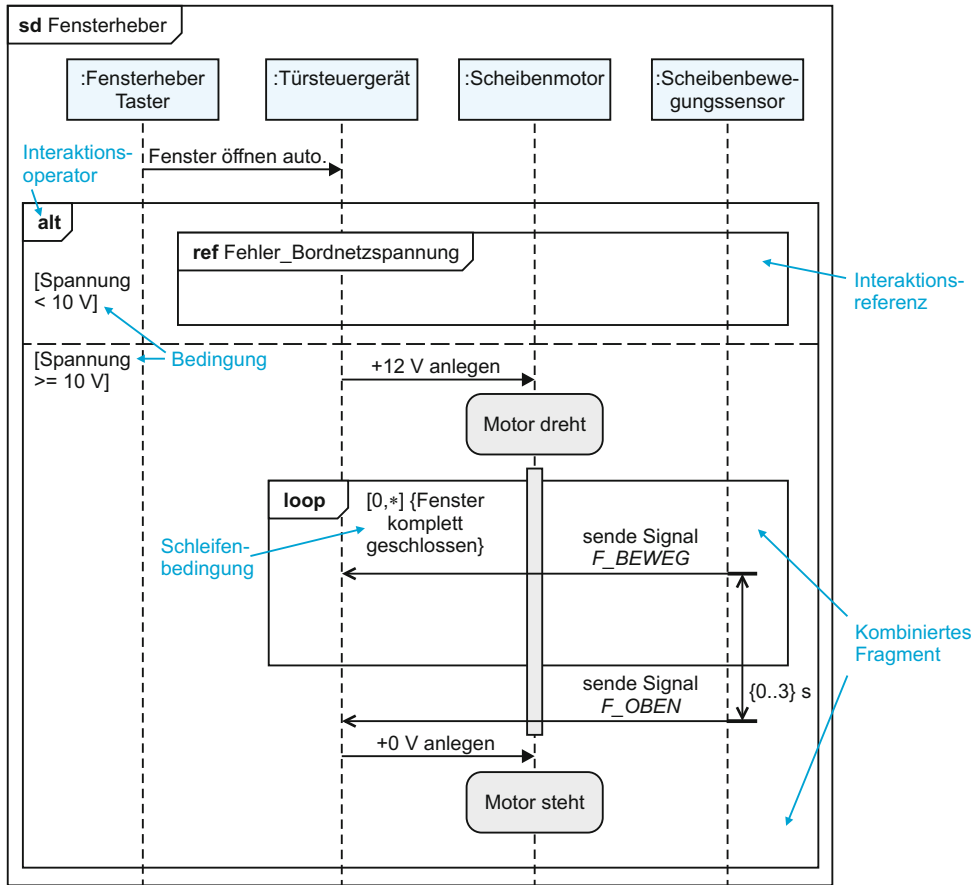


Abb. 10.5-15: Die finale Version des Sequenzdiagramms zur Fallstudie Fensterheber. Notation

Schluss durch entsprechende Nummern hinzufügen. Operationsnamen können ggf. mit Parametern und Ergebniswerten eingetragen werden. Das Kommunikationsdiagramm beschreibt zusätzlich die Objektbeziehungen zwischen den Objekten, die durch die Navigierbarkeit ergänzt werden können.

Die Abb. 10.5-16 zeigt die Notationselemente von Kommunikationsdiagrammen. An jede Objektbeziehung kann eine Nachricht angetragen werden, deren Richtung durch den Pfeil spezifiziert wird. Die Nummern bedeuten, dass nach der Operation 1 zuerst die Operation mit der Nummer 1.1 und dann die Operation mit der Nummer 1.2 ausgeführt wird. Die Schachtelung 1.2 und 1.2.1 drückt aus, dass die Operation n6 als erste Operation innerhalb der Operation n3 aufgerufen wird. Die Nummern 1.3a und 1.3b geben an, dass die entsprechenden Operationen *parallel* ausgeführt werden können. Schleifen werden durch einen Stern (*) zusammen mit einer

Bedingung spezifiziert, z. B. $*[i=1..10]$. Die Syntax für die Formulierung der Bedingungen ist in der UML *nicht* festgelegt. Oft verwendet man eine Programmiersprachen-Notation.

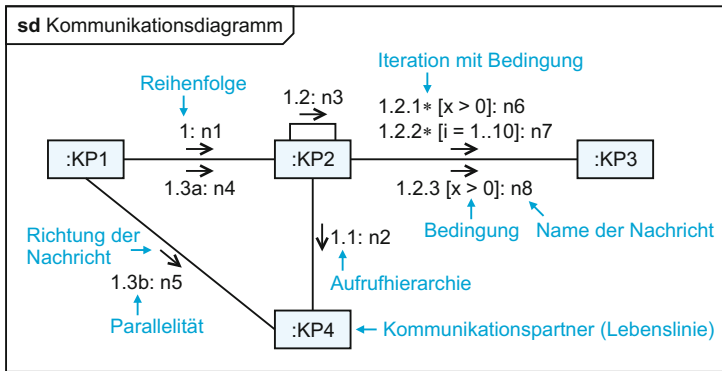


Abb. 10.5-16: Notation von Kommunikationsdiagrammen.

Im Kommunikationsdiagramm der Abb. 10.5-17 stellt jedes aufgeführte Objekt einen Platzhalter für ein beliebiges Objekt der Klasse dar. Dagegen modelliert das Objektdiagramm exemplarisch die Veranstaltung mit der Nummer 12, zu denen der angegebene Seminar- und Dozent sowie die aufgeführten Kunden existieren.

Beispiel:
SemOrg

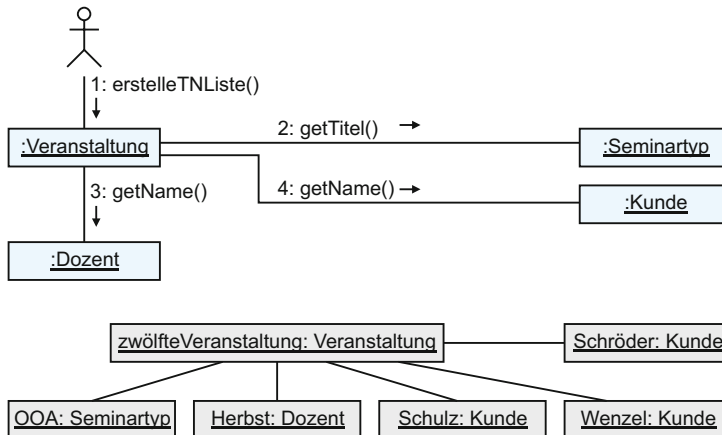


Abb. 10.5-17: Kommunikationsdiagramm (oben) im Vergleich zu einem Objektdiagramm (unten).

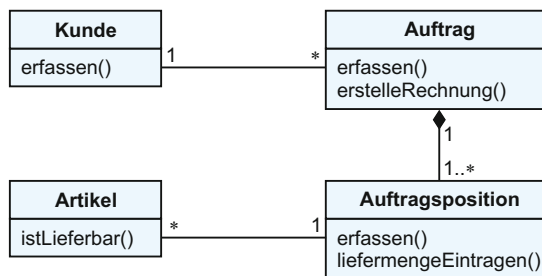
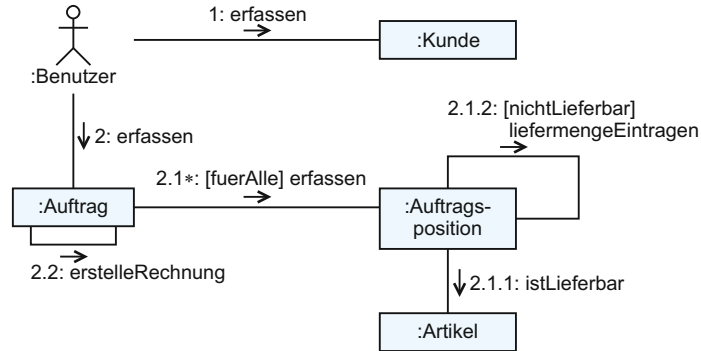
Die Abb. 10.5-18 modelliert das Szenario zum Bearbeiten eines Auftrags für einen Neukunden mit Hilfe eines Kommunikationsdiagramms. Zuerst wird der Kunde, dann dessen Auftrag erfasst. Diese Operation besteht aus dem Erfassen jeder einzelnen Auftragsposition. Es wird hier ausgedrückt, dass diese Kommunikation in einer Schleife stattfindet. Da der Kunde von einem Artikel mehr bestellen kann, als momentan auf Lager ist, muss erst dessen Lieferbarkeit geprüft werden. Ist ein Artikel nicht in der gewünschten Anzahl lieferbar (Bedingung), wird die tatsächliche Liefermenge eingetragen

Beispiel

III 10 Dynamik

werden. Ist die gewünschte Anzahl des Artikels lieferbar, so wird in diesem Fall nichts eingetragen. Sind alle Auftragspositionen erfasst, kann die Rechnung erstellt werden. Die Operationen, die im Kommunikationsdiagramm verwendet werden, sind auch im Klassendiagramm aufgeführt.

Abb. 10.5-18:
Kommunikations-
diagramm:
Auftrag für einen
Neukunden
bearbeiten.



Hinweis

In den Programmiersprachen gibt es keine Notation, um die Zusammenarbeit der Objekte zu dokumentieren, sondern ein Szenario kann nur aus der Reihenfolge der Operationsaufrufe gewonnen werden.

10.5.3 Box: Sequenz- und Kommunikationsdiagramm – Methode und Checkliste

Sequenz- und Kommunikationsdiagramme tragen dazu bei, die Operationen von Klassen zu identifizieren und den Fluss der Nachrichten durch das System zu definieren. Außerdem dienen sie dazu, die Vollständigkeit und Korrektheit des statischen Modells zu validieren. Darüber hinaus lassen sich Szenarien gleichzeitig als Test-Szenarien verwenden. Die Boxen »Checkliste Sequenz- & Kommunikationsdiagramme« zeigen, wie diese Diagramme erstellt und geprüft werden können.

Ergebnisse**■ Sequenzdiagramm, Kommunikationsdiagramm**

Für jedes relevante Szenario ist ein Sequenzdiagramm zu erstellen. Alternativ können Kommunikationsdiagramme verwendet werden.

Konstruktive Schritte**1 Wenn Use Cases vorliegen, dann aus jedem Use Case mehrere Szenarien entwickeln**

- Variationen eines Use Case ermitteln (positive Ausgänge, Fehlschläge).
- Standardausführung und Alternativen.
- Positive und negative Fälle unterscheiden.
- Prüfen, welche Szenarien wichtig sind.
- Diagramme benennen und beschreiben.

2 Beschreiben Sie das Szenario in Textform.

- Name.
- Alle Vorbedingungen, die zu dieser Ausführung des Use Case führen.
- Ergebnisse bzw. Wirkung des Szenarios.

3 Geeignete Diagrammart wählen.

- Sequenzdiagramm: Zeigt zeitliche Dimension sehr deutlich, Reihenfolge leicht nachzuvollziehen.
- Kommunikationsdiagramm: Zeigt Assoziationen, ermöglicht Bedingungen und Iterationen auf einfache Art.

4 Welche Akteure sind an dem Szenario beteiligt?

- Aus dem Use Case-Diagramm entnehmen.
- Meistens nur ein Akteur, aber auch mehrere möglich.

5 Welche Kommunikationspartner sind beteiligt?

- Beteiligte Klassen.
- Müssen verschiedene Objekte einer Klasse verwendet werden, dann Objekt-namen vergeben.
- Ordnen Sie die Kommunikationspartner so an, dass die Nachrichtenpfeile überwiegend von links nach rechts gezeichnet werden.

6 Wie läuft das Szenario ab?

- In welcher Reihenfolge müssen die Operationen ablaufen?
- Welcher Kommunikationspartner führt eine Operation aus?
- Konzentrieren Sie sich auf die essenzielle Verarbeitung.
- Vermeiden Sie es, jeden Sonderfall zu beschreiben.

7 Wie ist das Szenario zu strukturieren?

- Eine oder mehrere Transaktionen.
- Zentrale Struktur (*fork diagram*).
- Dezentrale Struktur (*stair diagram*).

Box 1a:
Checkliste
Sequenz- &
Kommunikationsdiagramme

Konstruktive Schritte

Szenarien können direkt aus Use Cases abgeleitet werden. Wählen Sie einen Use Case aus und überlegen Sie, welche Variationen auftreten können. Jede Variation führt zu einem unterschiedlichen Ergebnis des Use Case und bildet ein Szenario.

Szenarien sollten zunächst in Textform dokumentiert werden, damit man einen schnellen Überblick über deren Funktionalität erhält. Das kann beispielsweise in folgender Form erfolgen:

- Name des Szenarios.

1 Use Case →
Szenarien

2 Szenario in
Textform

III 10 Dynamik

- Bedingungen, die zu dieser Variation des *Use Case* führen (Unter welchen Voraussetzungen wird dieses Szenario ausgeführt?).
 - Ergebnis des Szenarios.
- Beschreiben Sie zunächst nicht, *wie* das Szenario abläuft. Verwenden Sie hier noch keine Objekte oder Attribute.

Beispiel 1a Aus dem *Use Case* bearbeite Schadensfall (siehe »Box: *Use Case* – Methode und Checkliste«, S. 262) werden folgende Szenarien abgeleitet: **Szenario 1:**

bearbeite Schadensfall (Schaden bezahlt)

Bedingungen:

- notwendige Daten vorhanden
- Police gültig
- Police deckt Schaden ab

Ergebnis:

- Schadensersatzanspruch wird voll beglichen

Szenario 2: bearbeite Schadensfall (Police ungültig)

Bedingungen:

- notwendige Daten vorhanden
- Police ungültig

Ergebnis:

- Antragsteller erhält Schreiben

Szenario 3: bearbeite Schadensfall (unvollständige Deckung)

Bedingungen:

- notwendige Daten vorhanden
- Police gültig
- Police deckt Schadensersatzforderung nur unvollständig ab
- erfolgreiches Verhandeln mit Antragsteller

Ergebnis:

- Schaden entsprechend Vergleich bezahlt

Achten Sie darauf, alle Bedingungen, die zu einem Szenario führen, explizit anzugeben. Oft lassen sich neue Szenarien durch Variationen der schon dokumentierten Szenarien finden. In diesem Fall sollten Sie das ursprüngliche Szenario um die neue Bedingung ergänzen.

3 Welches Diagramm?

Sequenzdiagramme bieten den Vorteil, dass sie die zeitliche Reihenfolge transparent dokumentieren. Man kann sehr schön nachvollziehen, wie die einzelnen Operationen einander aufrufen. Das Kommunikationsdiagramm besitzt den Vorteil, dass Entscheidungen und Iterationen einfach modelliert werden können. Außerdem sind die Objektbeziehungen bzw. die Assoziationen ersichtlich, die die Basis der Kommunikation darstellen. Kommunikationsdiagramme besitzen darüber hinaus den Vorteil, dass sie einfacher zu erstellen sind als Sequenzdiagramme.

Die Akteure, die Szenarien auslösen, lassen sich aus dem *Use Case*-Diagramm und der *Use Case*-Schablone ermitteln. Akteure können die gleichen Namen wie Klassen besitzen. Beispielsweise kann in einem Szenario eines Online-Shops der Akteur Kunde mit einem Objekt der Klasse Kunde kommunizieren. Beim ersten Kunden handelt es sich um das externe Objekt, beim zweiten um das interne Objekt, das eine Abstraktion des externen Kunden bildet.

4 Akteure

Der wichtigste Schritt für das Erstellen eines Sequenz- oder Kommunikationsdiagramms ist die Ermittlung der benötigten Kommunikationspartner. In den meisten Sequenzdiagrammen ist es üblich, die Kommunikationspartner so anzuordnen, dass die meisten Nachrichtenpfeile von links nach rechts zeigen.

5 Kommunikationspartner

- Überlegen Sie, welche Klassen bzw. Objekte an dem Szenario beteiligt sind. Fügen Sie diese Objekte in das Diagramm ein. Die meisten Klassen, die Sie hier verwenden, sind bereits im Klassendiagramm enthalten. Eventuell finden Sie auch neue Klassen, die dann im statischen Modell nachzutragen sind.
- Wenn mehrere Objekte einer Klasse verschiedene Aufgaben im Szenario haben, dann wird jedes Objekt aufgeführt und entsprechend seiner Bedeutung benannt, z. B. können beim Kopieren von einem Verzeichnis in ein anderes die Namen *Quelle:Verzeichnis* und *Ziel:Verzeichnis* verwendet werden.

Bei der Erstellung der Diagramme im *Requirements Engineering* kommt es weniger darauf an, den exakten Fluss der Operationen durch das System zu ermitteln, wie er später vom System ausgeführt wird, sondern das fachliche Verhalten soll möglichst präzise beschrieben werden.

6 Ablauf des Szenarios

- Zerlegen Sie die Aufgaben des *Use Case* in Teile, bis jeder Teil einer Operation entspricht. Fragen Sie immer wieder »Was ist nun zu tun?« und »Welcher Kommunikationspartner führt diese Operation aus?«
- Im Kommunikationsdiagramm können Sie auf einfache Weise Bedingungen angeben. Damit können mehrere Variationen durch ein einziges Diagramm beschrieben werden. Auf die Angabe von Iterationen kann im *Requirements Engineering* meistens verzichtet werden.

Konzentrieren Sie sich bei der Spezifikation auf die essenzielle Verarbeitung. Im Allgemeinen ist es im *Requirements Engineering* nicht notwendig, jeden Sonderfall eines Szenarios, der zu einem Fehlerausgang führt, zu beschreiben. Halten Sie die Diagramme so einfach wie möglich, aber spezifizieren Sie alles Wichtige.

III 10 Dynamik

Beispiel: Die Abb. 10.5-20 zeigt das Sequenzdiagramm für folgendes Szenario der Seminarorganisation:
SemOrg

Szenario: bearbeite Anmeldung (positiver Fall)

Bedingungen:

- Seminar existiert
- neuer Kunde

Ergebnis:

- Anmeldebestätigung erstellt

Die Abb. 10.5-21 zeigt das gleiche Szenario als Kommunikationsdiagramm.

Abb. 10.5-20:
Sequenzdiagramm
für eine
erfolgreiche
Anmeldung eines
neuen Kunden
zum Seminar.

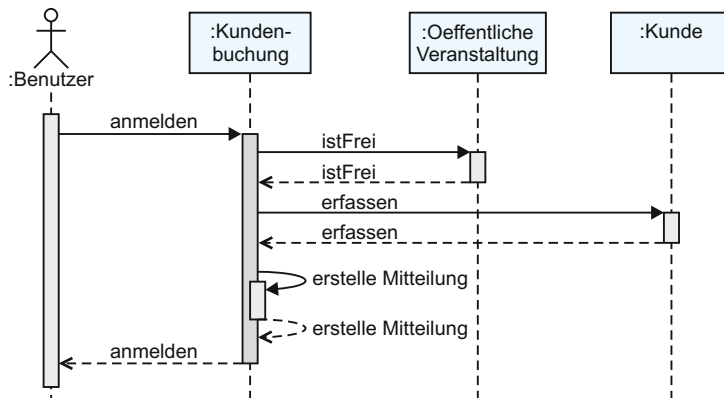
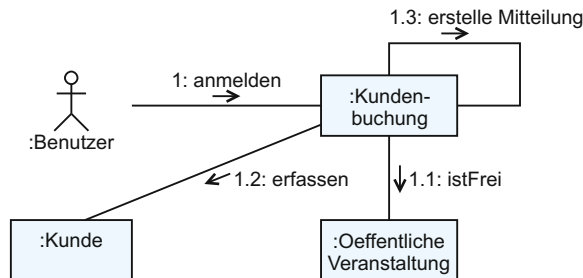


Abb. 10.5-21:
Kommunikations-
diagramm für eine
erfolgreiche
Anmeldung eines
neuen Kunden
zum Seminar.



7 Struktur des Szenarios

Bei Szenarien werden oft die Benutzer als Akteure angetragen. Die Funktionen, die diese Akteure direkt aktivieren, werden Benutzerfunktionen genannt. Jede Benutzerfunktion kann weitere Funktionen aktivieren (Abb. 10.5-22).

Diese weiteren Aufrufe können auf verschiedene Arten strukturiert werden (Abb. 10.5-23).

Bei der zentralen Organisation gibt es ein Objekt, das die Steuerung aller anderen Objekte übernimmt (*fork diagram*). Dieses Objekt verkapselt das Wissen, welche Operationen in welcher Reihenfolge aufgerufen werden. Wenn sich diese Abläufe häufig ändern, dann wirken sich diese Änderungen nur auf das Steuerungsobjekt aus. In einer dezentralen Struktur kommuniziert jedes beteiligte Ob-

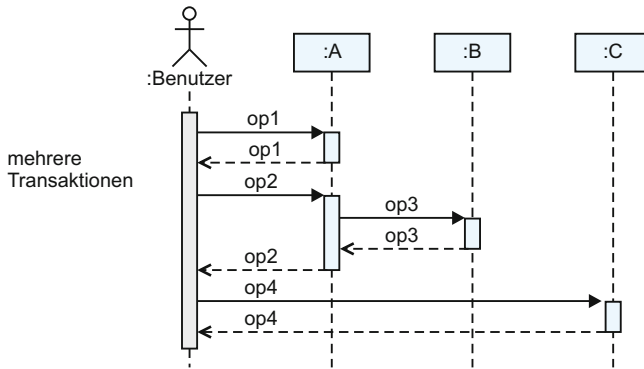


Abb. 10.5-22:
Szenario
bestehend aus
mehreren
Transaktionen.

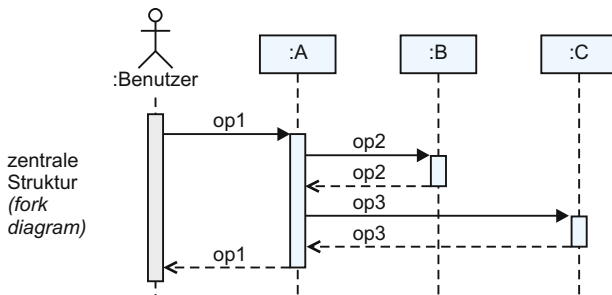
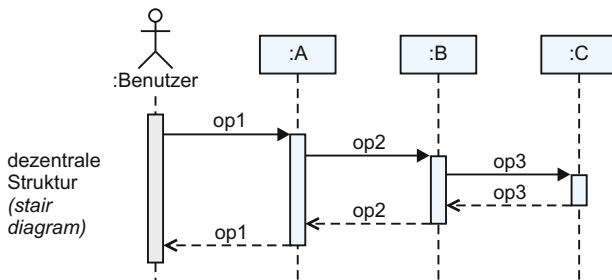


Abb. 10.5-23: fork
diagram und stair
diagram.



jekt lediglich mit einem Teil der anderen Objekte (*stair diagram*). Bei dieser Organisationsform ist die Steuerungslogik im System verteilt. Sie vermeidet, dass einige Objekte besonders aktiv und andere übermäßig passiv sind.

Analytische Schritte

In vielen Fällen existiert zwischen Klassen, deren Objekte miteinander kommunizieren, auch eine Assoziation, d.h. eine permanente Objektbeziehung zwischen den Objekten. In diesem Fall kennt jedes Senderobjekt seine zugehörigen Empfängerobjekte. Es ist jedoch auch möglich, dass Objekte ohne eine solche permanente Objektbeziehung kommunizieren können. Beispielsweise kann eine Operati-

8 Empfänger
erreichbar?

Box 1a:
Checkliste
Sequenz- &
Kommunikationsdiagramme

Analytische Schritte

8 Sind die Empfänger-Objekte erreichbar?

- Assoziation existiert (permanente Objektbeziehung).
- Identität kann dynamisch ermittelt werden (temporäre Objektbeziehung).

9 Ist das Diagramm konsistent mit dem Klassendiagramm?

- Alle Klassen sind auch im Klassendiagramm enthalten.
- Mit Ausnahme von Verwaltungsoperationen werden nur Operationen aus dem Klassendiagramm eingetragen.

10 Fehlerquellen

- Vollständigen Kontrollfluss beschreiben.
- Datenfluss beschreiben.
- Benutzungsoberfläche beschreiben.
- Zu viele Details beschreiben.

on die Identität eines Objekts ermitteln und als Ergebnis zurückliefern. Müssen Objekte miteinander kommunizieren, die *nicht* durch eine Assoziation miteinander verbunden sind, so ist zu prüfen, woher das Senderobjekt die Identität des Empfängerobjektes erhält. Sie können auf eine Assoziation verzichten, wenn sich die Objekte *nicht* permanent kennen müssen, weil eine entsprechende Kommunikation nur selten stattfindet und das gewünschte Objekt auch anders identifiziert werden kann.

- 9 Konsistenz** Beide Diagramme stellen eine Verbindung zum Klassendiagramm her. Sie lassen sich daher einerseits zur Modellbildung und andererseits zur Validierung des Klassendiagramms verwenden. Können die beteiligten Klassen die genannte Aufgabe wirklich gemeinsam lösen? Enthält das Diagramm nur Elemente des Klassendiagramms?

Beispiel Die Abb. 10.5-25 zeigt zwei Szenarien und einen Ausschnitt aus dem Klassendiagramm einer Bibliotheksverwaltung.

- 10 Fehlerquellen** Weder Sequenz- noch Kommunikationsdiagramme sind dazu gedacht, vollständige Kontrollflüsse und Datenflüsse zu modellieren. Für diese Aufgaben bietet die UML das Aktivitätsdiagramm. Szenarien sollen keine Beschreibungen der Benutzungsoberfläche enthalten. Beschränken Sie sich auf die wichtigsten Szenarien. Beschreiben Sie *nicht* jeden Sonderfall. Vermeiden Sie die Überspezifikation von Szenarien, d. h., verzichten Sie auf unnötige Details.

10.5.4 Timing-Diagramm

Das **Timing-Diagramm** – auch Zeitverlaufsdiagramm genannt – ermöglicht die Darstellung des zeitlichen Verhaltens eines oder mehrerer Kommunikationspartner im Zusammenspiel. Dieser Diagrammtyp kann zum Beispiel genutzt werden um

- die Reaktionszeit einer Komponente auf den Erhalt einer Nachricht oder einer Wertänderung zu modellieren,

Szenario: Ausleihen von Büchern (registrierter Leser)

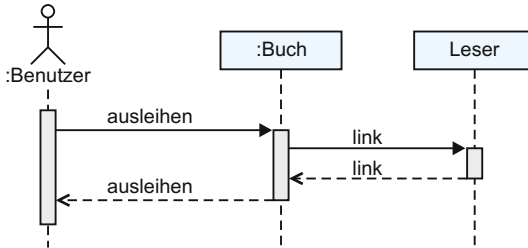
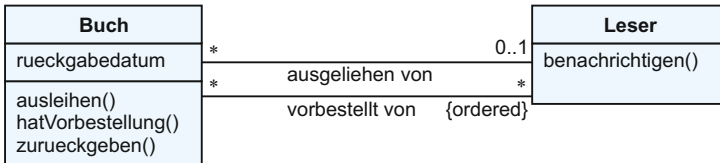
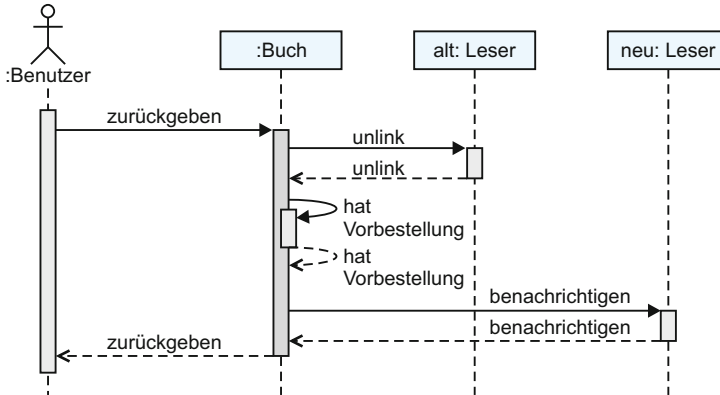


Abb. 10.5-25:
Szenarien und
Klassendiagramm
einer Bibliotheks-
verwaltung.

Szenario: Zurückgeben von Büchern (Vorbestellung liegt vor)



- die Veränderung von Werten oder Zuständen einer Komponente über die Zeit darzustellen oder
- die Bearbeitungszeit einer Aufgabe abzubilden.

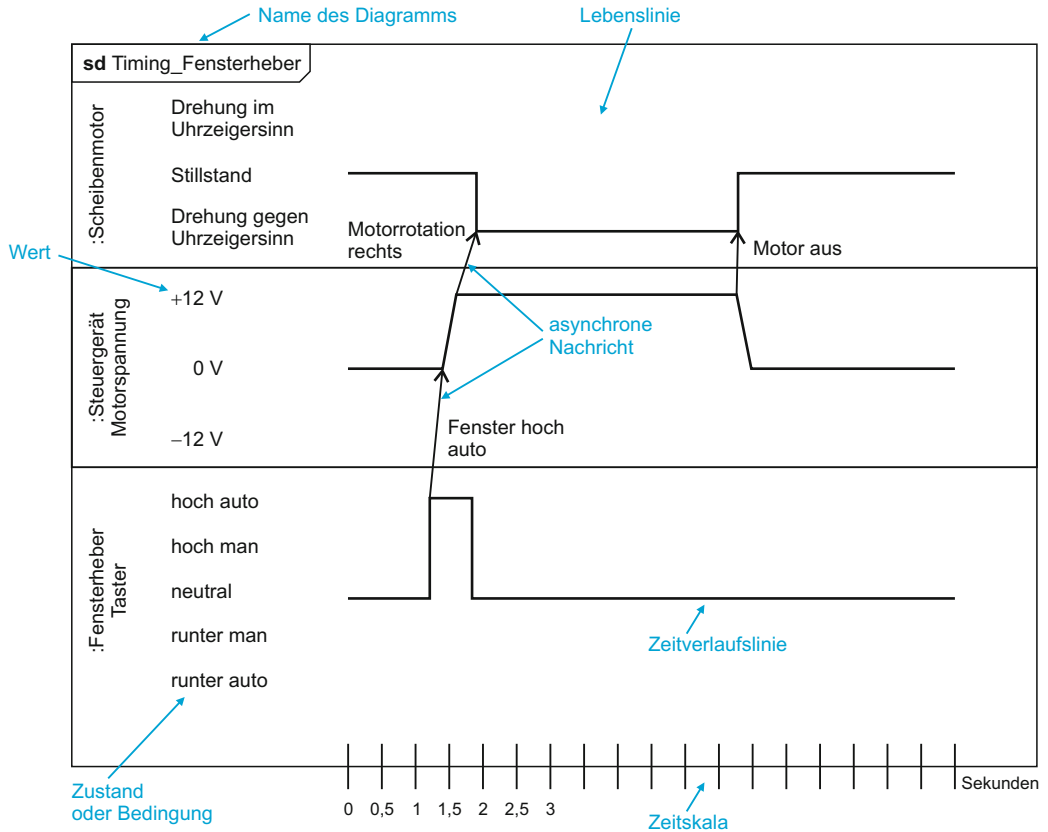
Timing-Diagramme finden häufig ihre Anwendung bei der Entwicklung eingebetteter Systeme, da dort das zeitliche Verhalten von Komponenten oftmals von sehr hoher Bedeutung ist. Ein Airbag hat beispielsweise keinerlei Nutzen, wenn sich dieser nicht rechtzeitig öffnet. Weiterhin sollte beachtet werden, dass Timing-Diagramme keinerlei Notationselemente zur Verfügung stellen, welche den Nutzer bei der übersichtlichen Abbildung komplexer Abläufe unterstützen (siehe »Sequenzdiagramm«, S. 333). Demnach sollte die zu modellierende Interaktion einen relativ einfachen Ablauf besitzen. Sollen zum Beispiel mehrere Varianten einer Interaktion modelliert werden, so muss dies durch mehrere Timing-Diagramme abgebildet werden.

Typische
Anwendungs-
fälle

III 10 Dynamik

Notation	Der grundlegende Aufbau eines Timing-Diagramms lässt sich am besten beschreiben, indem man die horizontale und die vertikale Achse getrennt betrachtet.
Kommunikationspartner & Lebenslinien	Entlang der Vertikalen sind die an der Interaktion beteiligten Kommunikationspartner aufgereiht, wie in der Abb. 10.5-26 zu sehen. Die Bereiche der an der Interaktion beteiligten Elemente sind durch horizontale Linien unterteilt. Die dadurch entstehenden länglichen Quadrate werden als Lebenslinien bezeichnet. Am linken Ende, also am Anfang dieser Lebenslinie wird der Name des entsprechenden Kommunikationspartners festgelegt. Direkt daneben werden nun die Zustände oder Variablenwerte übereinander aufgereiht, welche man im abgebildeten Szenario unterscheiden möchte.
Zeitskala & Zeitverlaufslinie	Am Fuß des Diagramms befindet sich in horizontaler Richtung eine Zeitskala . Diese ist in der Regel in äquidistante Zeitabschnitte unterteilt und wird passend mit Werten und einer Einheit beschriftet. Die Zeit schreitet innerhalb dieser Skala immer von links nach rechts fort. Innerhalb jeder Lebenslinie wird weiterhin eine sogenannte Zeitverlaufslinie eingezeichnet. Diese Linie zeigt in Abhängigkeit der Zeit, die Wert- oder Zustandsänderung der abgebildeten Größen, wie in der Abb. 10.5-26 zu erkennen ist. Die Wertänderungen in der Abb. 10.5-26 werden teilweise durch einen vertikalen Ausschlag der Zeitverlaufslinie abgebildet. Diese Wertänderungen passieren somit in Nullzeit, sprich sofort. Möchte man darstellen, dass die Veränderung eines Wertes oder ein Zustandsübergang eine gewisse Zeit benötigt, so kann dies durch eine Schräge in der Zeitverlaufslinie modelliert werden.
Nachrichten	Um die Verläufe der verschiedenen Zeitverlaufslinien in Zusammenhang zu bringen, können Nachrichten in Form von Pfeilen verwendet werden. Diese können überaus flexibel eingesetzt werden, um die kausalen Abhängigkeiten zwischen Wertänderungen verschiedener Kommunikationspartner zu verdeutlichen. Dieser Flexibilität fällt jedoch eine klare Semantik zum Opfer, sodass Nachrichten einen rein dokumentativen Charakter in Timing-Diagrammen besitzen.

Die Abb. 10.5-26 zeigt die Reaktion des Türsteuergeräts auf das Betätigen des FensterheberTasters. Wie in der Illustration zu sehen, reagiert das Steuergerät mit leichter Verzögerung auf die Fahrereingabe indem die Versorgungsspannung des Scheibenmotors auf +12V angehoben wird. Dies passiert *nicht* in Nullzeit, sondern benötigt einen gewissen Augenblick. Dies ist abgebildet durch eine Anschrägung der Zeitverlaufslinie. Daraufhin beginnt sich der Motor im Uhrzeigersinn zu drehen, solange bis das Steuergerät den Motor wieder abschaltet.



10.5.5 Zusammenfassung

Sequenzdiagramme betonen den zeitlichen Aspekt des dynamischen Verhaltens. Die Reihenfolge und die Verschachtelung der Operationen sind gut zu erkennen. In ein Sequenzdiagramm können mehrere externe Operationen, die von einem Akteur nacheinander aktiviert werden, eingetragen werden.

Im *Requirements Engineering* werden Sequenzdiagramme verwendet, um einzelne Abläufe so präzise zu beschreiben, dass deren fachliche Korrektheit diskutiert werden kann und um eine geeignete Vorgabe für Entwurf und Implementierung zu erstellen. Im Entwurf werden Sequenzdiagramme für eine detaillierte Spezifikation der Operationsaufrufe verwendet und enthalten dann alle beteiligten Operationen.

Kommunikationsdiagramme zeigen Operationsaufrufe zwischen Objekten. Die Reihenfolge und die Verschachtelung der Operationen werden durch eine hierarchische Nummerierung angegeben. Dadurch wird die Reihenfolge weniger deutlich sichtbar.

Abb. 10.5-26:
Beispiel für ein
Timing-
Diagramm,
angelehnt an die
Fallstudie
»Fensterheber«.

Kommunikations-
diagramme

III 10 Dynamik

Der Vorteil für den *Requirements Engineer* ist, dass er sich beim Erstellen des Diagramms noch nicht auf die Ausführungsreihenfolge festlegen muss, sondern zunächst die Objekte und ihre Kommunikation beschreiben und in einem weiteren Schritt die Reihenfolge hinzufügen kann. Kommunikationsdiagramme eignen sich gut dazu, die Wirkung komplexer Operationen zu beschreiben.

Sequenzdiagramme heben den zeitlichen Aspekt des dynamischen Verhaltens deutlich hervor. Die Reihenfolge und die Verschachtelung der Operationen sind anhand der Zeitachse sehr leicht zu erkennen. Kommunikationsdiagramme betonen die Aufrufe von Operationen zwischen Objekten. Reihenfolge und Verschachtelung werden nur durch eine hierarchische Nummerierung angegeben und sind daher für den Leser nicht so schnell ersichtlich. Im Kommunikationsdiagramm können Schleifen und Bedingungen einfacher modelliert werden als im Sequenzdiagramm.

Gibt es zu Sequenz- und Kommunikationsdiagrammen zugehörige Klassen, dann sollten die Operationen mit dem Klassendiagramm konsistent sein, d. h., alle Botschaften, die an ein Objekt einer Klasse gesendet werden, sollen im Klassendiagramm in der Operationsliste dieser Klasse enthalten sein.

Timing-
Diagramme

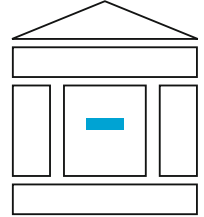
Das Timing-Diagramm modelliert das zeitliche Verhalten der Kommunikationspartner. Es zeigt alle an der Interaktion beteiligten Elemente in einer gemeinsamen Zeitskala und ähnelt der Anzeige eines Oszilloskops. Es erlaubt zum Beispiel die Visualisierung von Wertverläufen oder von Reaktionen auf eingehende Nachrichten über der Zeit. Die darzustellende Interaktion sollte jedoch eine überschaubare Komplexität besitzen, da Schleifen, parallele Ausführung oder das Referenzieren von anderen Timing-Diagrammen nicht unterstützt werden.

Zur Klassifikation

- UML-Sequenz-, Kommunikations- und Timing-Diagramme werden grafisch (mit textuellen Annotationen) dargestellt.
- Der Grad der Formalität reicht von informal bis semiformal.
- Sie werden in der Spezifikations- und Entwurfsphase eingesetzt.
- Sequenz- und Kommunikationsdiagramme werden in allen Anwendungsbereichen verwendet, Timing-Diagramme in softwareintensiven Anwendungen.

11 Logik

Neben statischen Strukturen und dynamischen Abläufen zwischen Systemkomponenten gibt es noch logische Abhängigkeiten, die unabhängig von der Statik und der Dynamik immer gelten. Solche logischen Abhängigkeiten können auf verschiedene Art und Weise formuliert werden. Mit Hilfe der Logik ist es auch möglich, Schlüsse zu ziehen, Widersprüche zu erkennen und Vollständigkeitsprüfungen zu durchführen. Im Bereich der Programmierung können mit Hilfe der Logik Programme verifiziert werden.



Die mathematische Logik ermöglicht es, logische Aussagen formal zu spezifizieren und Schlüsse daraus zu ziehen:

■ »Formale Logik«, S. 357

In der UML können logische Zusammenhänge einzelnen Modellelementen oder mehreren Modellelementen zugeordnet werden. Alternativ oder ergänzend kann die Sprache OCL zur Spezifikation von logischen Formeln verwendet werden:

■ »Constraints und die OCL in der UML«, S. 377

Eine der ältesten Basiskonzepte der Softwaretechnik sind Entscheidungstabellen, die es ermöglichen, den logischen Zusammenhang zwischen kombinierten Bedingungen und Aktionen zu beschreiben:

■ »Entscheidungstabellen und Entscheidungsbäume«, S. 386

Noch modularer sind logische Wenn-Dann-Regeln, die von Regelmäschinen ausgewertet werden:

■ »Regeln«, S. 404

i



11.1 Formale Logik

Unter einer formalen Logik versteht man eine Menge von Regeln, mit denen Schlussfolgerungen formalisiert werden können. Spezielle formale Logiken wie die Aussagenlogik oder die Prädikatenlogik bilden die Grundlage für die formale Spezifikation von technischen und softwareintensiven Systemen. Im Wesentlichen umfasst jede formale Logik

- eine Klasse von mathematischen Modellen, zum Beispiel Mengen, Relationen, Funktionen oder Zustandsautomaten,
- eine präzise definierte Sprache mit der Aussagen über Modelle formuliert werden können und
- eine mathematische Relation zwischen Modellen und Aussagen, die angibt, ob ein Modell eine Aussage erfüllt oder nicht.

III 11 Logik

i Im Folgenden werden drei spezielle formale Logiken genauer vorgestellt, die für die exakte Spezifikation von Anforderungen und für die Verifikation von Programmen besonders wichtig sind:

- »Aussagenlogik«, S. 358
- »Prädikatenlogik«, S. 367
- »Temporale Logik«, S. 370

Die Beispiele beziehen sich vorwiegend auf die Fallstudie »Fensterheber«.

11.1.1 Aussagenlogik

Die Aussagenlogik bildet die einfachste der formalen Logiken. In ihr wird untersucht, welches Wissen sich über Aussagen gewinnen lässt, die aus einfachen Grundaussagen mittels einfacher Verknüpfungen gewonnen werden. Dabei ist hier nur erheblich, ob diese Grundaussagen wahr oder falsch sind. Ihnen wird keine tiefere Bedeutung beigemessen. Praktische Vorteile einer Formalisierung von Anforderungen in der Aussagenlogik liegen in der Präzisierung und Analysierbarkeit. Zum Beispiel können Anforderungen nach ihrer Formalisierung auf **Widerspruchsfreiheit** und **Vollständigkeit** geprüft werden.

Syntax der Aussagenlogik

Eine Formel der Aussagenlogik besteht aus **atomaren Aussagen**, die durch **Junktoren** verbunden werden. Ein Junktore beschreibt, wie die von ihm verknüpften Teilaussagen verbunden werden.

Beispiel 1a:
Fensterheber

Die folgende Anforderung für einen Fensterheber soll formalisiert werden (siehe »Fallstudie: Fensterheber – Die Spezifikation«, S. 117):
/F40/ Falls ein Signal anliegt, das das Öffnen einer Scheibe zur Folge hat, wird auf dem entsprechenden Motor die Spannung -12 V angelegt. Liegt die Batteriespannung zum Beginn der Scheibenbewegung unterhalb von 10V, so wird die Scheibenbewegung nicht durchgeführt. Statt dessen wird die CAN-Botschaft B_LOW_WIN = 1 gesendet. Für die Öffnen-Bewegung sind folgende Fälle zu beachten:
/F50/ Ist die relevante Schalterstellung gleich Fenster runter man. oder ist die relevante CAN-Botschaft WIN_x_OP = 01, so wird die Scheibe nach unten bewegt. Die Bewegung endet, wenn
/F51/ das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird) oder
/F52/ sich die Scheibe in der unteren Position befindet (d.h. F_UNTEN bzw. FF_UNTEN) oder
/F53/ ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht (in diesem Fall wird der neue Bewegungsbe-
fehl bearbeitet) oder

/F54/ der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der unteren Position befindet (in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen) oder

/F55/ die Ansteuerung länger als 3 sec. dauert, ohne dass erkannt wird, dass sich die Scheibe in der unteren Position befindet (in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen).

Als erstes Beispiel wird der Satz **/F40/** »Falls ein Signal anliegt, das das Öffnen einer Scheibe zur Folge hat, wird auf dem entsprechenden Motor die Spannung -12 V angelegt.« formalisiert.

Der Satz hat die Form »Falls A, wird B«.

Somit ist »Ein Signal, das das Öffnen einer Scheibe zur Folge hat, liegt an« eine atomare Aussage, und

»Auf dem entsprechenden Motor wird die Spannung -12 V angelegt.« eine weitere atomare Aussage.

Der Junktor in diesem Fall ist »Falls«.

In der Aussagenlogik werden solche Formeln nicht in natürlicher Sprache dargestellt, sondern als Formel:

»Ein Signal, das das Öffnen einer Scheibe zur Folge hat, liegt an«
→ »Auf dem entsprechenden Motor wird die Spannung -12 V angelegt.«

Es gibt verschiedene Schreibweisen für Junktoren. Die Tab. 11.1-1 stellt die wichtigsten Junktoren in ihrer gängigsten Schreibweise dar.

Junktoren

Tab. 11.1-1:
Schreibweisen für
Junktoren.

Symbol	Bezeichnung
\wedge	Und
\vee	(inklusives) Oder
\rightarrow	Impliziert, wenn – dann
\leftrightarrow	Äquivalenz, genau dann – wenn
\neg	nicht

Als atomare Formeln können i. Allg. beliebige Ausdrücke gewählt werden. Dem Namen einer atomaren Formel in der Aussagenlogik wird *keine* tiefere Bedeutung zugemessen. Die einzige Ausnahme bilden die beiden besonderen Formeln *wahr* und *falsch*, die als grundsätzlich wahr oder grundsätzlich falsch betrachtet werden.

Atomare
Formeln

Aussagenlogische Formeln kann man mit Hilfe der folgenden abstrakten Grammatik darstellen:

Formel ::= atomare Formel | (Formel) | \neg Formel | falsch | wahr | Formel \vee Formel | Formel \wedge Formel | Formel \rightarrow Formel | Formel \leftrightarrow Formel

Syntax

III 11 Logik

Zwar gibt es Vorrangregeln für verschiedene logische Junktoren. Da diese nicht so geläufig sind, ist es üblich, die Struktur logischer Formeln mit Hilfe von Klammern zu verdeutlichen.

Beispiel 1b Mit Hilfe der Junktoren kann das »Öffnen eines Fensters der Autotür« im Detail beschrieben werden. Zunächst legt man die atomaren Formeln fest:

- 1 Die relevante Schalterstellung ist gleich »Fenster runter man.«, kurz: *SchalterRunter*.
- 2 Die relevante CAN-Botschaft *WIN_x_OP* ist = 01, kurz: *CANRunter*.
- 3 Die Scheibe befindet sich in der unteren Position, kurz: *ScheibeUnten*.
- 4 Der Scheibenbewegungssensor sendet keine Signale, kurz: *SensorStumm*.
- 5 Die Botschaft *ERROR_WIN* = 1 wird gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen, kurz: *Fehler*.
- 6 Die Spannung -12 V wird auf dem entsprechenden Motor angelegt, kurz: *MotorAn*.
- 7 Die Batteriespannung liegt unterhalb von 10 V, kurz: *BatterieSchwach*.
- 8 Die CAN-Botschaft *B_LOW_WIN* = 1 wird gesendet, kurz: *BatterieWarnung*.
- 9 Die Ansteuerung dauert bereits länger als 3 Sekunden, kurz: *LangAn*.
- 10 Die Bewegung endet, kurz: *Ende*.
- 11 Eine Bewegung der Scheibe wird gerade veranlasst, kurz: *ScheibeSollBewegtWerden*.

Man kann jetzt die Anforderung formalisieren, indem man diese atomaren Formeln entsprechend zusammensetzt. Man beginnt am einfachsten mit den Endbedingungen der Bewegung:

$(\text{ScheibeUnten} \vee (\text{SensorStumm} \wedge \text{MotorAn} \wedge \neg \text{ScheibeUnten}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten})) \rightarrow \text{Ende}$

Als nächstes formalisiert man, wann die Scheibe bewegt werden kann:

$(\text{SchalterRunter} \vee \text{CANRunter}) \rightarrow \text{ScheibeSollBewegtWerden}$

Die Bewegung findet nur statt, wenn sie nicht am Ende angekommen ist und nicht zu wenig Spannung vorhanden ist:

$(\text{ScheibeSollBewegtWerden} \wedge \neg \text{Ende} \wedge \neg \text{BatterieSchwach}) \rightarrow \text{MotorAn}$

Wenn zu wenig Spannung vorhanden ist, soll stattdessen eine Warnung ausgegeben werden:

$(\text{ScheibeSollBewegtWerden} \wedge \text{BatterieSchwach}) \rightarrow \text{BatterieWarnung}$

Schließlich wird in zwei Fällen eine Fehlermeldung erzeugt:

$((\text{SensorStumm} \wedge \text{MotorAn} \wedge \neg \text{ScheibeUnten}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten})) \rightarrow \text{Fehler}$

Die Formalisierung von Anforderungen kann beim Finden von Fehlern hilfreich sein. Im Weiteren werden u.a. die Konzepte der disjunktiven und konjunktiven Normalform eingeführt, die als Hilfsmittel diesen Sachverhalt veranschaulichen.

Um zu zeigen, wie durch die Formalisierung Fehler gefunden werden können, wird zusätzlich angenommen, statt der Anforderung /F54/ stände die folgende fehlerhafte Anforderung /F54f/ (für »F54 falsch«) in der Anforderungsspezifikation:

Beispiel 1c

/F54f/ [... oder] der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird (in diesem Fall wird die Botschaft ERROR_WIN = 1 gesendet und der Fehlercode 0x35 in den Fehlerspeicher eingetragen Mit anderen Worten:

Es wird »oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird und sich die Scheibe noch nicht in der unteren Position befindet« in /F54/ ersetzt durch »oder der Scheibenbewegungssensor (F_BEWEG bzw. FF_BEWEG) keine Signale sendet, obwohl der Scheibenmotor angesteuert wird« (d.h. der kursive Satzteil fehlt in /F54f/). Aus dieser fehlerhaften Anforderung und /F55/ erhält man statt dessen als letzte Formel:

$((\text{SensorStumm} \wedge \text{MotorAn}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten})) \rightarrow \text{Fehler}$

Vergleichen Sie diese Formel mit der letzten Formel aus Beispiel 1b.

Semantik der Aussagenlogik

Da die atomaren Formeln in der Aussagenlogik per se keinen Wahrheitswert besitzen (Ausnahme: *wahr* und *falsch*), kann man allein auf Grund des Namens einer atomaren Formel nicht sagen, ob sie wahr oder falsch ist. Daher kann man den meisten aussagenlogischen Formeln auch nicht ohne weiteres einen Wahrheitswert zuordnen.

Um dennoch Aussagen über den Wahrheitswert einer Formel gewinnen zu können, muss man festlegen, welche atomaren Formeln wahr und welche falsch sind. Zu diesem Zweck bedient man sich einer **Belegung**, die jeder (relevanten) atomaren Formel einen Wahrheitswert von 0 (falsch) oder 1 (wahr) zuweist. Dabei gilt für jede Belegung, dass sie *wahr* den Wert 1 und *falsch* den Wert 0 zuordnet. Der Wahrheitswert einer aussagenlogischen Formel kann dann rekursiv mit Hilfe der folgenden Tabellen berechnet werden, wobei B eine Belegung und p, p₁ und p₂ aussagenlogische Formeln sind.

Belegung
Wahrheits-
tabelle

In der Praxis ist es oft interessant, zu wissen, für welche Belegungen eine bestimmte Formel wahr und für welche sie falsch ist. Eine systematische Darstellung dafür ist die sogenannte **Wahrheitstabelle**. Für eine aussagenlogische Formel p mit atomaren Formeln a₁

III 11 Logik

Tab. 11.1-2:
Wahrheitstabelle
für binäre
Junktoren.

$B(p_1)$	$B(p_2)$	$B(p_1 \wedge p_2)$	$B(p_1 \vee p_2)$	$B(p_1 \rightarrow p_2)$	$B(p_1 \leftrightarrow p_2)$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Tab. 11.1-3:
Wahrheitstabelle
für unäre
Junktoren.

$B(p)$	$B(\neg p)$
0	1
1	0

bis a_k hat diese Tabelle $k+1$ Spalten, eine für jede atomare Formel und eine für die Gesamtformel. Die Zeilen der Tabelle entsprechen den verschiedenen möglichen Belegungen der atomaren Formeln (es gibt insgesamt 2^k verschiedene). In den Einträgen für die atomaren Formeln steht jeweils die Belegung der zugehörigen atomaren Formel und in der Spalte für die Gesamtformel der sich daraus ergebende Wahrheitswert der Gesamtformel. Zur Bestimmung einer Wahrheitstabelle ist es oft nützlich, zusätzliche Spalten für Teilformeln der Gesamtformel einzufügen. Auch kann man mehrere Wahrheitstabellen durch Einführung entsprechender Spalten zu einer zusammenzufassen.

Beispiel 1d Für die Formel, die beschreibt, wann die Scheibe bewegt werden soll, erhält man eine Wahrheitstabelle (Tab. 11.1-4).

$B(\text{ScheibeSoll BewegtWerden})$	$B(\text{SchalterRunter})$	$B(\text{CANRunter})$	$B(\text{SchalterRunter} \vee \text{CANRunter})$	$B((\text{SchalterRunter} \vee \text{CANRunter}) \rightarrow \text{ScheibeSoll BewegtWerden})$
0	0	0	0	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Tab. 11.1-4: Scheibenbewegung.

Oft interessiert man sich dafür, ob bestimmte aussagenlogische Formeln überhaupt eine Belegung haben, die wahr ist (also 1 ergibt), und wenn ja, ob das sogar für alle Belegungen gilt. Daraus ergibt sich die folgende Klassifikation:

- Eine Formel, bei der alle Belegungen falsch ergeben, nennt man **unerfüllbar**. Die Wahrheitstabelle einer solchen Formel hat eine Spalte voller Nullen als letzte Spalte.

- Eine Formel, bei der alle Belegungen wahr ergeben, nennt man **allgemeingültig**. Die Wahrheitstabelle einer solchen Formel hat eine Spalte voller Einsen als letzte Spalte.
- Eine Formel, bei der mindestens eine Belegung wahr ergibt, nennt man **erfüllbar**. Die Wahrheitstabelle einer solchen Formel hat in der letzten Spalte mindestens eine Eins.

Offenbar sind allgemeingültige Formeln immer erfüllbar. Die Formeln aus Beispiel 1 sind allesamt erfüllbar, aber nicht allgemeingültig. Die Formel $p \wedge \neg p$ ist ein Beispiel für eine unerfüllbare Formel, und $p \vee \neg p$ ist ein Beispiel für eine allgemeingültige.

Beispiele

Wenn zwei Formeln die gleichen Wahrheitstabellen haben, spricht man von **äquivalenten** Formeln. Beispielsweise kann man leicht nachrechnen, dass $\neg(p \wedge q)$ und $\neg p \vee \neg q$ äquivalent sind (De Morgan'sche Regel).

Normalformen

Häufig ist es interessant, zu wissen, welche Struktur die Belegungen einer aussagenlogischen Formel haben, die diese Formel erfüllen. Ebenso interessant ist die Frage, ob zwei gegebene Formeln äquivalent sind. Auch ist es oft nützlich, zu einer gegebenen Formel eine einfachere äquivalente Formel zu finden. Eine wichtige Rolle für die Lösung solcher Probleme spielen die sogenannten **Normalformen** von aussagenlogischen Formeln. Eine Normalform beschreibt dabei eine gewisse Art, Formeln aufzuschreiben. Formeln in Normalform stehen einer algorithmischen Weiterverarbeitung zur Verfügung und sind oft einfacher zu verstehen als beliebige Formeln. Außerdem gilt für einige Normalformen, dass äquivalente Formeln alle die gleiche Normalform haben. Die praktischen Anwendungen von Normalformen sind dementsprechend vielfältig:

- Erkennung unerwünschter erfüllender oder nicht erfüllender Belegungen, wie sie zum Beispiel bei fehlerhaften Anforderungen auftauchen,
- Vereinfachung der Formeln für eine einfachere Implementierung,
- Entdeckung unnötiger Formelbestandteile (und damit Anforderungsbestandteile) und so weiter.

DNF und KNF

Die zwei bekanntesten Normalformen sind die **konjunktive Normalform** und die **disjunktive Normalform**. Diese beiden Normalformen sind sich vom Prinzip her sehr ähnlich, und das zugrundeliegende Konstruktionsverfahren funktioniert für beide auf sehr ähnliche Weise. Als ersten Begriff zur Erklärung beider Formen benötigt man das **Literal**.

DNF, KNF

III 11 Logik

Literal Ein **Literal** ist entweder eine atomare Formel oder die Negation einer atomaren Formel. Mit anderen Worten: Wenn p eine atomare Formel ist, so sind p und $\neg p$ Literale.

Klausel, KNF Eine Formel in **konjunktiver Normalform** (kurz **KNF**) hat die Form $(L_{1,1} \vee \dots \vee L_{1,n(1)}) \wedge \dots \wedge (L_{m,1} \vee \dots \vee L_{m,n(m)})$, wobei die $L_{i,j}$ Literale sind. Einen Ausdruck der Form $L_1 \vee \dots \vee L_n$ nennt man auch **Klausel**. Somit kann die konjunktive Normalform auch als Formel der Form $K_1 \wedge \dots \wedge K_m$ beschrieben werden, wobei die K_i Klauseln sind.

Beispiel 1e Die Formel

$(\text{ScheibeUnten} \vee (\text{SensorStumm} \wedge \text{MotorAn} \wedge \neg \text{ScheibeUnten}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten})) \rightarrow \text{Ende}$

ist äquivalent zur Formel (siehe auch die Tab. 11.1-5)

$(\neg \text{ScheibeUnten} \vee \text{Ende}) \wedge (\neg \text{SensorStumm} \vee \neg \text{MotorAn} \vee \neg \text{ScheibeUnten} \vee \text{Ende}) \wedge (\neg \text{LangAn} \vee \text{ScheibeUnten} \vee \text{Ende})$

Diese Formel besteht aus drei Klauseln:

- 1** $(\neg \text{ScheibeUnten} \vee \text{Ende})$ mit den Literalen $\neg \text{ScheibeUnten}$ sowie Ende ,
- 2** $(\neg \text{SensorStumm} \vee \neg \text{MotorAn} \vee \neg \text{ScheibeUnten} \vee \text{Ende})$ mit den Literalen $\neg \text{SensorStumm}$, $\neg \text{MotorAn}$, $\neg \text{ScheibeUnten}$ sowie Ende und
- 3** $(\neg \text{LangAn} \vee \text{ScheibeUnten} \vee \text{Ende})$ mit den Literalen $\neg \text{LangAn}$, ScheibeUnten sowie Ende ,

Zu einer Formel kann es mehr als eine äquivalente konjunktive Normalform geben.

Monom, DNF Ganz analog ist eine Formel in **disjunktiver Normalform** (kurz **DNF**), wenn sie die Form $(L_{1,1} \wedge \dots \wedge L_{1,n(1)}) \vee \dots \vee (L_{m,1} \wedge \dots \wedge L_{m,n(m)})$, wobei die $L_{i,j}$ Literale sind. Einen Ausdruck der Form $L_1 \wedge \dots \wedge L_n$ nennt man auch **Monom**. Somit kann die disjunktive Normalform auch beschrieben als Formel der Form $M_1 \vee \dots \vee M_m$, wobei die M_i Monome sind.

Beispiel 1f Die Formel aus Beispiel 1e hat unter anderem die folgende DNF:

$(\neg \text{ScheibeUnten} \wedge \neg \text{SensorStumm} \wedge \neg \text{LangAn}) \vee (\neg \text{ScheibeUnten} \wedge \neg \text{MotorAn} \wedge \neg \text{LangAn}) \vee \text{Ende}$.

Sie hat drei Terme:

- 1** $(\neg \text{ScheibeUnten} \wedge \neg \text{SensorStumm} \wedge \neg \text{LangAn})$
- 2** $(\neg \text{ScheibeUnten} \wedge \neg \text{MotorAn} \wedge \neg \text{LangAn})$
- 3** Ende

Für jede Formel kann man eine zu ihr äquivalente Formel in konjunktiver Normalform und eine in disjunktiver Normalform finden. Dazu gibt es zwei Verfahren: Durch Ermittlung der Wahrheitstabelle oder durch Umformung. Das zweite Verfahren wird am folgenden Beispiel für eine disjunktive Normalform demonstriert. Dazu benutzt man die Umformungsregeln aus der Tab. 11.1-5, die für die Umformung von Formeln verwendet werden.

Formel	Äquivalente Formel
$(p \wedge q) \vee r$	$(p \vee r) \wedge (q \vee r)$
$(p \vee q) \wedge r$	$(p \wedge r) \vee (q \wedge r)$
$p \vee q$	$q \vee p$
$p \wedge q$	$q \wedge p$
$(p \vee q) \vee r$	$p \vee (q \vee r)$
$(p \wedge q) \wedge r$	$p \wedge (q \wedge r)$
$\neg (p \vee q)$	$\neg p \wedge \neg q$
$\neg (p \wedge q)$	$\neg p \vee \neg q$
$p \vee p$	p
$p \vee \text{falsch}$	p
$p \vee \text{wahr}$	wahr
$p \wedge p$	p
$p \wedge \text{falsch}$	falsch
$p \wedge \text{wahr}$	p
$p \vee \neg p$	wahr
$p \wedge \neg p$	falsch
$p \rightarrow q$	$\neg p \vee q$
$p \leftrightarrow q$	$(p \rightarrow q) \wedge (q \rightarrow p)$
$\neg \neg p$	p

Tab. 11.1-5:
Umformungen.

Um zu zeigen, wie die DNF zum Finden *fehlerhafter* Anforderungen benutzt werden kann, soll nun eine *unvollständige Variante der Anforderung* betrachtet werden. Dazu wird die Anforderung von oben leicht modifiziert, indem man /F54/ durch /F54f/ ersetzt.

Es wird eine DNF von der Formalisierung der *fehlerhaften* Anforderung

$f = (\text{SensorStumm} \wedge \text{MotorAn}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten}) \rightarrow \text{Fehler}$ bestimmt. Durch Anwendung der oben angegebenen Regeln bestimmt man folgendermaßen eine DNF:

- 1 $(\text{SensorStumm} \wedge \text{MotorAn}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten}) \rightarrow \text{Fehler}$
- 2 $\neg ((\text{SensorStumm} \wedge \text{MotorAn}) \vee (\text{LangAn} \wedge \neg \text{ScheibeUnten})) \vee \text{Fehler}$
- 3 $(\neg (\text{SensorStumm} \wedge \text{MotorAn}) \wedge \neg (\text{LangAn} \wedge \neg \text{ScheibeUnten})) \vee \text{Fehler}$
- 4 $((\neg \text{SensorStumm} \vee \neg \text{MotorAn}) \wedge (\neg \text{LangAn} \vee \neg \neg \text{ScheibeUnten})) \vee \text{Fehler}$
- 5 $((\neg \text{SensorStumm} \vee \neg \text{MotorAn}) \wedge (\neg \text{LangAn} \vee \text{ScheibeUnten})) \vee \text{Fehler}$
- 6 $((\neg \text{SensorStumm} \wedge (\neg \text{LangAn} \vee \text{ScheibeUnten})) \vee (\neg \text{MotorAn} \wedge (\neg \text{LangAn} \vee \text{ScheibeUnten}))) \vee \text{Fehler}$
- 7 $((\neg \text{SensorStumm} \wedge \neg \text{LangAn}) \vee (\neg \text{SensorStumm} \wedge \text{ScheibeUnten})) \vee ((\neg \text{MotorAn} \wedge \neg \text{LangAn}) \vee (\neg \text{MotorAn} \wedge \text{ScheibeUnten})) \vee \text{Fehler}$
- 8 $(\neg \text{SensorStumm} \wedge \neg \text{LangAn}) \vee (\neg \text{SensorStumm} \wedge \text{ScheibeUnten}) \vee (\neg \text{MotorAn} \wedge \neg \text{LangAn}) \vee (\neg \text{MotorAn} \wedge \text{ScheibeUnten}) \vee \text{Fehler}$

Beispiel 1g

III 11 Logik

Die DNF bietet die Möglichkeit sehr einfach erfüllende Belegungen abzulesen. Dazu überlegt man sich Folgendes: Wenn eines der Monome erfüllt ist, so ist auch die ganze Formel erfüllt. Ein Monom ist genau dann unerfüllbar, wenn eine atomare Formel p darin sowohl negiert als auch unnegiert vorkommt, also wenn das Monom die Form $\dots \wedge p \wedge \dots \wedge \neg p \wedge \dots$ hat. Für alle anderen Monome kann man sehr leicht eine erfüllende Belegung konstruieren: Wenn p im Monom unnegiert vorkommt (also als p ohne \neg davor), so belegt man p mit 1. Kommt p negiert vor (also als $\neg p$), so belegt man p mit 0, und wenn es gar nicht vorkommt, ist jede Belegung möglich.

Beispiel 1h An der DNF der fehlerhaften Anforderung liest man die folgenden erfüllenden Belegungen ab:

- 1 SensorStumm und LangAn werden mit 0 belegt, alle anderen mit beliebigen Werten.
- 2 SensorStumm wird mit 0 belegt, ScheibeUnten mit 1, alle anderen mit beliebigen Werten.
- 3 MotorAn und LangAn werden mit 0 belegt, alle anderen mit beliebigen Werten.
- 4 MotorAn wird mit 0 belegt, ScheibeUnten mit 1, alle anderen mit beliebigen Werten.
- 5 Fehler wird mit 1 belegt, alle anderen mit beliebigen Werten.

In diesem Fall findet man eine **Unvollständigkeit**:

Wenn Fehler mit 0 belegt werden soll, dann muss entweder SensorStumm oder MotorAn mit 0 belegt werden. Wenn aber der Motor an ist, ist es denkbar, dass die Scheibe schon unten angekommen ist (das liegt nahe, weil es eine atomare Formel ScheibeUnten gibt), und in diesem Fall ist der Scheibenbewegungssensor stumm, obwohl das *kein* Fehler ist.

Für die KNF gibt es wieder ein ähnliches Verfahren, um nicht erfüllende Belegungen abzulesen.

Hornformeln

Eine andere Normalform für bestimmte aussagenlogische Formeln sind die so genannten Hornformeln. Eine Hornformel hat dabei immer die Form $\langle (p_1 \wedge \dots \wedge p_n) \rightarrow k \rangle$. Viele Aussagen der Form »Wenn ..., dann ...« können mit Hilfe von Hornformeln ausgedrückt werden. Allerdings gibt es aussagenlogische Formeln (z.B. $p \vee q$), die *nicht* mit Hilfe von Hornformeln ausgedrückt werden können.

Der Folgerungsbegriff

Folgerung Eine weitere wichtige Frage in der Logik ist die Folgende: Wenn man eine Menge aussagenlogischer Formeln gegeben hat, kann man aus diesen Formeln auf weitere, neue Aussagen schließen?

Eine mögliche Definition des Folgerungsbegriffs ist die Folgende:

Eine aussagenlogische Formel G folgt aus den Formeln F_1 bis F_n , wenn unter jeder Belegung, die die Formeln F_1 bis F_n erfüllen, auch G erfüllt wird.

Diese Aussage kann man äquivalent so ausdrücken:

$(F_1 \wedge \dots \wedge F_n) \rightarrow G$ ist allgemeingültig.

Es werden einige der Formeln betrachtet, die die Anforderung an die Autofenstersteuerung beschreiben: Beispiel 1i

$F_1 = (\text{SchalterRunter} \vee \text{CANRunter}) \rightarrow \text{ScheibeSollBewegtWerden}$

$F_2 = (\text{ScheibeSollBewegtWerden} \wedge \neg \text{Ende} \wedge \neg \text{BatterieSchwach}) \rightarrow \text{MotorAn}$

$F_3 = (\text{ScheibeSollBewegtWerden} \wedge \text{BatterieSchwach}) \rightarrow \text{BatterieWarnung}$
Man kann zeigen (z. B. durch Betrachtung der Belegungen), dass aus diesen drei Formeln folgende Formel folgt:

$\text{SchalterRunter} \wedge \neg \text{Ende} \wedge \neg \text{BatterieWarnung} \rightarrow \text{MotorAn}$

Man kann diese Folgerung benutzen, um die Anforderungen zu validieren: Durch die formale Überprüfung dieser Formel ist sichergestellt, dass auf das Ereignis `SchalterRunter` immer die Scheibe bewegt wird, ausser, sie ist bereits ganz unten oder die Batterie ist leer. Auf diese Art und Weise kann man sich vergewissern, dass ein Druck auf den Schalter unter gegebenen Rahmenbedingungen immer zu einer Scheibenbewegung führt, was ja auch den Erwartungen an das System entspricht.

Weiterführende Themen

Aufbauend auf der oben vorgestellten Aussagenlogik lassen sich viele weitere Begriffe, Konzepte und Verfahren beschreiben. Als Beispiel seien hier die Minimierung aussagenlogischer Formeln, Äquivalenzregeln und Algorithmen zur Bestimmung von erfüllenden Belegungen genannt. Besonders wichtig ist auch die weitergehende Behandlung des Folgerungsbegriffs sowie der Begriff der Herleitung. Da die Behandlung dieser Fragen hier jedoch zu weit ginge, sei auf die Fachliteratur zur mathematischen Logik verwiesen, zum Beispiel auf [Schö00] oder [Erk02].

11.1.2 Prädikatenlogik

Im Gegensatz zur Aussagenlogik wird in der Prädikatenlogik nicht nur die Verknüpfung elementarer Aussagen betrachtet, sondern es werden komplexere Ausdrücke verwendet, die das Verhältnis von Objekten zueinander beschreiben können. Ein wesentlicher Unterschied zur Aussagenlogik liegt in der Möglichkeit in Formeln über Wertebereiche quantifizieren zu können. So ist es zum Beispiel mög-

III 11 Logik

lich, eine Aussage der Form »Für jede natürliche Zahl n gibt es eine natürliche Zahl m , die größer ist als n « in der Prädikatenlogik zu formalisieren.

Prädikatenlogische Formeln werden prinzipiell ähnlich wie aussagenlogische Formeln aufgebaut. Es gibt allerdings zwei Änderungen: Zum einen werden statt atomarer Formeln sogenannte **Prädikate** verwendet, und zum anderen gibt es neben Junktoren auch **Quantoren**.

Beispiel 1a:
Fensterheber

Die Sätze /F40/ »Falls ein Signal anliegt, das das Öffnen der Scheibe zur Folge hat, wird auf dem entsprechenden Motor -12 Volt angelegt. Liegt die Batteriespannung zu Beginn der Scheibenbewegung unterhalb von 10 Volt, so wird die Scheibenbewegung nicht durchgeführt.« könnte prädikatenlogisch folgendermaßen formalisiert werden:

$\forall \text{ signal: (ScheibeÖffnen(signal) } \wedge \text{ Batteriespannung} \geq 10 \text{ V}) \rightarrow \text{Motorspannung} = -12 \text{ V}$

Diese Zeile kann gelesen werden als »Für alle Signale gilt: Wenn das Signal besagt, dass die Scheibe geöffnet werden soll, und die Batteriespannung ist größer als 10 V, dann wird an den Motor -12 V angelegt.«

Anhand dieses Beispiels sollen nun die Bestandteile einer prädikatenlogischen Formel im Detail beschrieben werden. Die Formel hat folgende Bestandteile:

- Eine **Quantifizierung**, nämlich » $\forall \text{ signal:}$ «.
 - Drei **Prädikate**, nämlich »ScheibeÖffnen(signal)«, »Batteriespannung $\geq 10 \text{ V}$ « und »Motorspannung $= -12 \text{ V}$ «.
 - Zwei bereits aus der Aussagenlogik bekannte Junktoren: \wedge und \rightarrow .
- Um zu beschreiben, wie prädikatenlogische Formeln gebildet werden, benötigt man zuerst den Begriff des Terms. Ein Term ist in der Prädikatenlogik entweder eine Variable, eine Konstante oder eine Funktionsanwendung. Eine Funktionsanwendung hat im Allgemeinen die Form » $f(\text{Term}_1, \dots, \text{Term}_N)$ «, wobei f ein **Funktionssymbol** ist und Term_1 bis Term_N weitere Terme.

Beispiel 1b:
Terme

Im Fensterheber-Beispiel tauchen insgesamt fünf Terme auf, nämlich die drei Variablen »signal«, »Batteriespannung« und »Motorspannung« sowie die beiden Konstanten »10 V« und »-12 V«. Im Beispiel taucht kein Funktionsterm auf.

Die Funktion eines Terms ist es, bei der Auswertung einer prädikatenlogischen Formel einen Wert aus einer vorgegebenen Menge, dem sogenannten Universum anzunehmen. Aufbauend auf Termen können Prädikate definiert werden: Ein Prädikat setzt sich zusammen aus einem **Prädikatsymbol** sowie einer Anzahl von Termen.

Die Prädikate aus dem Beispiel haben folgende Struktur:

Beispiel 1b:
Terme

- 1 »ScheibeÖffnen(signal)« besteht aus dem Prädikatsymbol »ScheibeÖffnen« und dem Term »signal«.
- 2 »Batteriespannung ≥ 10 V« hat das Prädikatsymbol \geq und die Terme »Batteriespannung« und »10 V«.
- 3 »Motorspannung = -12 V« hat das Prädikatsymbol = und die Terme »Motorspannung« und »-12 V«.

Ein Prädikat hat die Aufgabe, abhängig von seinen Termen und seinem Prädikatsymbol einen Wahrheitswert anzunehmen. Schließlich kann man Prädikate zu prädikatenlogischen Formeln zusammensetzen. Eine prädikatenlogische Formel besteht dabei entweder aus einem einzelnen Prädikat, oder wird aus einfacheren Formeln zusammengesetzt. Das kann entweder mit Hilfe von Junktoren wie in der Aussagenlogik geschehen, oder aber durch **Quantifizierung**. Eine Quantifizierung hat dabei die Struktur »Quantor Variable Formel«. Es gibt zwei Quantoren, \forall (für alle Werte aus dem Universum gilt ...) sowie \exists (es gibt mindestens einen Wert im Universum, sodass ...). Um einer prädikatenlogischen Formel eine Bedeutung zuzuordnen, benutzt man sogenannte **Interpretationen**. Aufgabe einer Interpretation ist es, jedem Term einen Wert aus einer vorgegebenen Menge, dem sogenannten **Universum**, zuzuweisen, und darauf aufbauend zu ermitteln, ob die betrachtete prädikatenlogische Formel wahr oder falsch ist. Eine Interpretation besteht aus drei Teilen: Dem Universum, der Interpretation der Konstanten und der Belegung der Variablen. Die Interpretation der Konstanten legt fest, was die einzelnen Konstanten-, Funktions- und Prädikatsymbole konkret bedeuten.

Für die Formel zur Beschreibung des Fensterhebers wäre eine mögliche Interpretation:

Beispiel 1d

- 1 Das Universum umfasst alle denkbaren Signale.
- 2 Das =-Zeichen hat seine übliche Bedeutung, also: $a=b$ ist genau dann wahr, wenn a und b das gleiche Signal beschreiben.
- 3 Das \geq -Zeichen beschreibt, dass ein Signal eine Mindestspannung aufweisen muss. Mit anderen Worten: Wenn $a \geq b$ gilt, dann muss das Signal a immer mindestens die Spannung des Signals b haben.
- 4 Die Konstanten 10 V und -12 V werden als Signale mit konstanter Spannung verstanden.
- 5 Das Prädikatsymbol ScheibeÖffnen wird nur dann wahr, wenn sein Argument eine CAN-Bus-Nachricht ist, in der das entsprechende Bit gesetzt ist, oder ein Signal vom Benutzer-Bedienfeld, das das Fensteröffnen zur Folge hat.
- 6 Für »Batteriespannung« und »Motorspannung« können beliebige Werte gewählt werden. Im Folgenden sollen die Wirkungen einiger Werte betrachtet werden.

III 11 Logik

Der einfachste Fall wäre, für die Batteriespannung einen Wert von 12 V und für die Motorspannung den Wert -12 V anzunehmen. Da sicherlich $12\text{ V} \geq 10\text{ V}$ gilt, wird in diesem Fall die Formel wahr.

Nimmt man stattdessen eine Motorspannung von 0 V an, wird die Formel offenbar falsch.

Man beachte, dass kein Wert für die Variable »signal« angegeben wird. Das liegt daran, dass diese Variable quantifiziert ist. Um den Wahrheitswert der Formel zu berechnen, muss man also streng genommen alle Werte, die signal annehmen kann, einzeln durchspielen.

Aufbauend auf diesen Sprachmitteln lassen sich sehr viele komplexe Aussagen formalisieren. Eine genauere Behandlung würde zu weit führen, weswegen hier auf die entsprechende Fachliteratur (z. B. [Schö00] oder [Ehri01]) verwiesen wird.

11.1.3 Temporale Logik

Während bei der Aussagenlogik und der Prädikatenlogik einzelne statische Situationen betrachtet werden, hat die temporale Logik die Dynamik eines Systems im Blick. Beispielsweise können Aussagen wie »Das Autofenster bewegt sich so lange, bis es sich in der unteren Position befindet« formalisiert werden.

Syntax Die Syntax (und Semantik) der temporalen Logik baut wesentlich auf der Syntax der Aussagenlogik auf. Es gibt mehrere Formen der temporalen Logik. Hier werden drei Varianten betrachtet, denen gemeinsam ist, dass sie einen Zeitbegriff haben, der nur auf der Kausalitätsreihenfolge beruht und *nicht* auf physikalischer Zeit. Die drei Varianten sind CTL*, CTL und LTL. Dabei ist CTL* die allgemeinste der drei Logiken, die sowohl CTL als auch LTL umfasst. Im praktischen Einsatz findet man aus Effizienzgründen allerdings meistens CTL und LTL.

CTL*

In CTL* betrachtet man zwei Sorten von Formeln:

- Eine **Zustandsformel** beschreibt dabei eine Bedingung, die zu einem bestimmten Zeitpunkt gelten soll.
- Eine **Pfadformel** beschreibt eine Bedingung, die während eines ganzen Zeitabschnitts gelten soll.

Die einfachsten Zustandsformeln sind die atomaren Formeln der Aussagenlogik, die dann zu komplexeren aussagenlogischen Formeln zusammengesetzt werden können. Eine weitere Möglichkeit, eine Zustandsformel zu gewinnen, ist es, eine Pfadformel mit einem

Pfadquantor zu verstehen. Umgekehrt ist jede Zustandsformel auch eine Pfadformel, und man kann Pfadformeln mit Hilfe von (aussagenlogischen) **Junktoren** und **temporalen Operatoren** verbinden.

Dieses Beispiel bezieht sich auf die Anforderungen /F50/, /F51/ und /F52/ (siehe »Fallstudie: Fensterheber – Die Spezifikation«, S. 117): Ist die relevante Schalterstellung gleich »Fenster runter man.« oder ist die relevante CAN-Botschaft $WIN_x_OP = 01$, so wird die Scheibe nach unten bewegt. Die Bewegung endet, wenn

Beispiel 1a:
Fensterheber

- 1 das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird) oder
- 2 sich die Scheibe in der unteren Position befindet (d. h. F_UNTEN bzw. FF_UNTEN).

Die Formalisierung sieht folgendermaßen aus:

Als atomare Formeln werden gewählt:

- FRM für »Schalterstellung gleich Fenster runter man.«,
- $WINxOP01$ für »relevante CAN-Botschaft ist $WIN_x_OP = 01$ «,
- UNTEN für »F_UNTEN bzw. FF_UNTEN« und
- SNU für »Scheibe nach unten bewegen«.

Damit kann man die Anforderungen schreiben als:

- »SNU, so lange FRM oder $WINxOP01$ und nicht UNTEN«

In dieser Form handelt es sich bei der Formalisierung um eine Pfad-Formel. Da die Anforderungen immer gelten sollen, kann man die Formalisierung wie folgt erweitern:

- »Bei jeder Benutzung soll immer gelten: SNU, so lange FRM oder $WINxOP01$ und nicht UNTEN«.

In dieser Formalisierung findet sich ein Pfad-Quantor, nämlich »Bei jeder Benutzung soll gelten«, und zwei temporale Operatoren, nämlich »immer« und »so lange, wie«.

Es gibt potenziell viele verschiedene temporale Operatoren, aber lediglich zwei Pfadquantoren. Die folgenden Tabellen zeigen die gängigsten.

temporale Operatoren	Bezeichnung
X	Im nächsten Schritt
F	Irgendwann
G	Immer
U	So lange, bis

Tab. 11.1-6:
Temporale
Operatoren.

Pfadquantoren	Bezeichnung
A	Auf jedem Pfad gilt
E	Es gibt einen Pfad, für den gilt

Tab. 11.1-7:
Pfadquantoren.

III 11 Logik

Beispiel 1b Das Beispiel 1a kann wie folgt als temporallogische Formel dargestellt werden:

○ $AG (SNU \ U \neg ((FRM \vee WINxOP01) \wedge \neg UNTEN))$.

Hierbei werden die erwähnten temporalen Operatoren »Bei jeder Benutzung soll gelten« und »immer« durch die Operatoren A bzw. G formalisiert.

CTL und LTL

Die temporalen Logiken CTL und LTL sind Einschränkungen von CTL*. Sie sind nicht so ausdrucksstark wie CTL*, dafür aber einfacher zu verarbeiten, sodass es hier Werkzeuge für die Arbeit mit diesen Logiken gibt.

Bei CTL gilt die Einschränkung, dass vor jedem Ausdruck, der einen temporalen Operator enthält, ein Pfadquantor stehen muss. Konkret bedeutet das: Anstelle von Pfadformeln gibt es in CTL nur quantifizierte Pfadformeln, also $AX\ p$, $AF\ p$, $AG\ p$ und $A(p\ U\ q)$ sowie $EX\ p$, $EF\ p$, $EG\ p$ und $E(p\ U\ q)$, wobei p und q Zustandsformeln sind. Eine Formel wie $AFG\ p$ ist kein CTL, da $G\ p$ keine Zustandsformel ist.

Beispiel 1c Die Formel $AG (SNU \ U \neg ((FRM \vee WINxOP01) \wedge \neg UNTEN))$ ist *keine* CTL-Formel, da $(SNU \ U \neg ((FRM \vee WINxOP01) \wedge \neg UNTEN))$ keine Zustandsformel ist.

Bei LTL gibt es *keine* Pfadquantoren. Man kann implizit davon ausgehen, dass vor der gesamten Formel ein A steht, das allerdings üblicherweise einfach weggelassen wird.

Beispiel 1d Die Formel aus dem Beispiel 1b ist eine LTL-Formel. Allerdings ist die Formel $E (SNU \ U \neg ((FRM \vee WINxOP01) \wedge \neg UNTEN))$ *keine* LTL-Formel, da es in LTL keinen Pfadquantor E gibt.

Semantik

Zur Auswertung von CTL*-Formeln verwendet man so genannte Kripke-Strukturen, die Systeme von Zuständen beschreiben. Zu jedem Zustand gehört nicht nur eine Belegung der atomaren Formeln, sondern auch eine Menge von Zuständen, die als nächstes von diesem Zustand aus erreicht werden können. Durch »Aufrollen« dieser Struktur erhält man eine Baumstruktur, den sogenannten **Berechnungsbaum**.

Beispiel 1d Die Abb. 11.1-1 zeigt zwei mögliche Berechnungsbäume für den Fensterheber, wobei nur Zustandsfolgen der Länge 3 oder kürzer betrachtet werden.

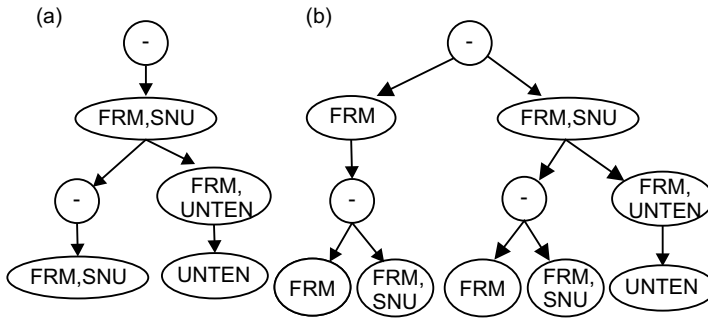


Abb. 11.1-1: Zwei Berechnungsbäume für den Fensterheber. Der linke Berechnungsbaum gehört zu einer korrekten Implementierung, der rechte zu einer fehlerhaften.

Einen Berechnungsbaum kann man folgendermaßen lesen: Es gibt **Zustände** (dargestellt durch Kreise bzw. Ellipsen) und **Übergänge** (dargestellt durch Pfeile). In jedem Berechnungsbaum gibt es genau einen **Anfangszustand**. Dieser wird üblicherweise ganz oben dargestellt und hat nur Übergänge, die von ihm weg zeigen. In jedem Zustand steht jeweils, welche atomaren Formeln in diesem Zustand wahr sind. So bedeutet »FRM, UNTEN« beispielsweise, dass die atomaren Formeln »FRM« und »UNTEN« wahr sind, alle anderen (z. B. »SNU«) falsch. Ein Strich zeigt an, dass keine atomare Formel wahr ist.

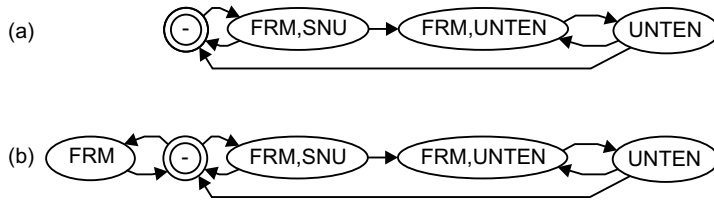
Zu jedem Zeitpunkt befindet sich das durch den Berechnungsbaum beschriebene System in genau einem der Zustände des Baums. Zum Systemstart befindet es sich insbesondere im Anfangszustand. Immer, wenn Zeit vergeht, geht das System von seinem gegenwärtigen Zustand in einen der Zustände über, die mit Hilfe einer Transition von dort aus erreichbar sind. Beispielsweise kann das System, das durch den Baum (a) dargestellt wird, zwei mögliche Verhaltensweisen zeigen. Die eine fängt im Anfangszustand von (a) an; hier ist keine atomare Formel wahr. Nach einem Zeitschritt sind FRM und SNU wahr, und im nächsten Schritt gibt es zwei Möglichkeiten: Entweder ist wieder keine Formel wahr, oder FRM und UNTEN werden wahr. Im letzten Schritt sind dann entweder FRM und SNU wahr (wenn vorher keine Formel wahr war) oder nur UNTEN (im anderen Fall).

Der erste Berechnungsbaum beschreibt einen korrekt implementierten Fensterheber. Der zweite beschreibt einen Fensterheber mit einem einfachen Fehler: Es ist möglich, dass zwar der Knopf »Fenster nach unten« gedrückt wird, sich das Fenster aber nicht bewegt.

Die beiden Berechnungsbäume wurden durch Ausrollen aus den Automaten der Abb. 11.1-2 gewonnen. Diese Darstellung kann man folgendermaßen lesen: Es gibt wieder Zustände und Zustandsübergänge. Der Anfangszustand ist diesmal durch Doppelkreise beziehungsweise Doppelellipsen markiert. Ansonsten ist die Semantik die gleiche wie für Berechnungsbäume.

III 11 Logik

Abb. 11.1-2: Die Kripke-Strukturen, aus denen die Berechnungsbäume ermittelt wurden.

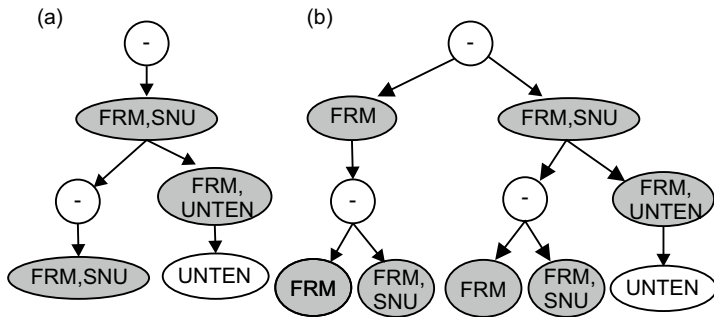


Die Darstellung als Berechnungsbaum hat allerdings den Vorteil, dass man Folgen von Zuständen besser darstellen kann. Deswegen werden im Weiteren nur Berechnungsbäume dargestellt.

Aussagenlogische Formeln können in diesem System für jeden Zustand ausgewertet werden, indem man einfach die zum Zustand gehörende Belegung auf die Formel anwendet.

Beispiel 1e Die Formel $FRM \vee SNU$ gilt auf allen grau markierten Zuständen der beiden Berechnungsbäume (Abb. 11.1-3).

Abb. 11.1-3: Die Berechnungsbäume von oben, in denen jeweils gezeigt wird, welche Zustände die Formel » $FRM \vee UNTEN$ « erfüllen.



Wenn der Berechnungsbaum nur ein Pfad ist, also jeder Knoten nur ein Kind hat, kann man Pfadformeln sehr einfach auswerten, und zwar entsprechend der Abb. 11.1-4. Für die Beschreibung der Semantik allgemeiner temporallogischer Formeln betrachtet man als nächstes die Pfadformeln. Um die Erklärung zu vereinfachen, macht man zunächst die Annahme, dass der Berechnungsbaum nur einen Pfad enthält. Mit anderen Worten: Von jedem Zustand geht höchstens ein Zustandsübergang aus. In diesem Fall kann man die Bedeutung der einzelnen Pfadformeln folgendermaßen erklären:

- Ist die Pfadformel eine aussagenlogische Formel, so gilt das oben über aussagenlogische Formeln Gesagte.
- Die Pfadformel » $F p$ « gilt, wenn in mindestens einem zukünftigen Zustand p erfüllt ist.
- Im Fall » $G p$ « gilt die Pfadformel, wenn alle zukünftigen Zustände p erfüllen.

- Wenn die Pfadformel die Form $\gg X p \ll$ hat, dann gilt die Pfadformel in jedem Zustand, in dessen Nachfolgezustand (also dem Zustand, der mit Hilfe eines einzigen Übergangs erreicht werden kann) p gilt.
- Schließlich ist die Pfadformel $\gg p U q \ll$ wahr, wenn in irgendeinem zukünftigen Zustand q gilt und in allen Zuständen vom aktuellen Zustand bis zum Vorgänger des Zustandes, in dem q gilt, die Formel p wahr ist.

Diese Bedingungen sind in der Abb. 11.1-4 grafisch dargestellt: Es wird immer ein Pfad der Länge 3 gezeigt, der die Mindestanforderungen erfüllt, damit die darüber stehende Pfadformel wahr wird.

Enthält der Berechnungsbaum mehrere mögliche Pfade, so muss man einen Pfad zur Betrachtung auswählen.

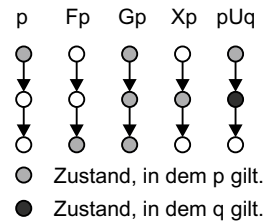


Abb. 11.1-4: Semantik der temporalen Operatoren auf einfachen Pfaden. Zu jedem Operator ist ein Berechnungspfad angegeben, der die Mindestanforderungen des Operators erfüllt.

Im Rückgriff auf das Beispiel 1e kann man die Auswertung der Pfadformel $SNU \ U \neg ((FRM \vee WINxOP01) \wedge \neg UNTEN)$ auf zwei verschiedenen Pfaden betrachten. Diese Pfade sind in der Abb. 11.1-5 dargestellt durch eine graue Markierung.

Beispiel 1f

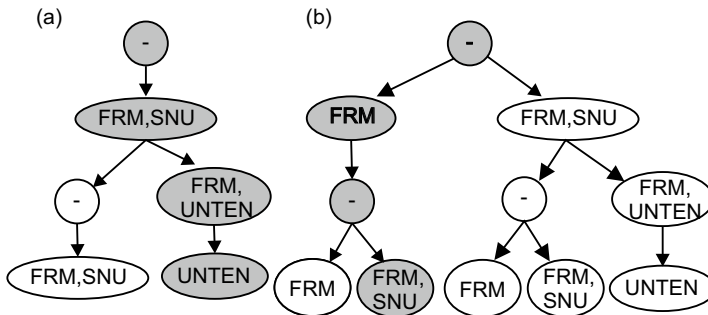


Abb. 11.1-5: Pfade in den Berechnungsbäumen von oben.

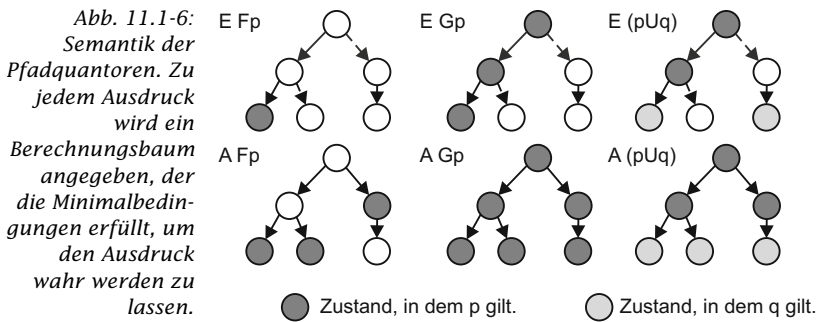
Die Formel ist in beiden Berechnungsbäumen erfüllt: Es gilt ja in beiden Zuständen \neg neg; FRM, und \neg neg; SNU, also auch \neg neg; $(FRM \vee SNU)$ und damit auch \neg neg; $((FRM \vee SNU) \wedge \neg UNTEN)$. Damit ist die gesamte Bedingung beide Mal im Anfangszustand erfüllt.

Betrachtet man andererseits $G (SNU \ U \neg ((FRM \vee WINxOP01) \wedge \neg UNTEN))$, sieht das anders aus. Hier muss die Behauptung von oben in jedem Zustand gelten. Da das im ersten Zustand zutrifft, muss die Bedingung als nächstes im zweiten Zustand überprüft werden. Für (a) gilt die Behauptung immer noch, während bei (b) die Formel *nicht* mehr erfüllt ist, denn nun gilt $((FRM \vee WINxOP01) \wedge \neg UNTEN)$, aber nicht SNU. Somit ist diese strengere Formel in der Lage, die Berechnungsbäume (a) und (b) zu unterscheiden. Nun wurde in Beispiel 1d

III 11 Logik

definiert, dass (a) zu einer korrekten Implementierung gehört und (b) zu einer fehlerhaften. Da die Formel auf (a) gilt, kann die Formel den Fehler in (b) erkennen.

Schließlich bleibt noch die Erklärung der Pfadquantoren: Wenn eine Pfadformel f mit einem A -Quantor versehen ist, so gilt $A f$ in einem Zustand, wenn f für alle von diesem Zustand ausgehenden Pfade gilt. Ist f mit einem E -Quantor versehen, so gilt $E f$ in einem Zustand, wenn es einen von diesem Zustand ausgehenden Pfad gibt, auf dem f gilt. Einige Beispiele für solche Ausdrücke findet man in der Abb. 11.1-6.



Beispiel 1g Im Folgenden werden einige CTL*-Formeln auf den oben beschriebenen Berechnungsbäumen ausgewertet. ($SNU \ U \neg (FRM \vee WINxOP01) \wedge \neg UNTEN$) wird dazu mit p abgekürzt. Die Tab. 11.1-8 gibt die Ergebnisse an.

Tab. 11.1-8: Ergebnisse.

Formel	(a)	(b)
$A p$	wahr	wahr
$E p$	wahr	wahr
$AG p$	wahr	falsch
$EG p$	wahr	wahr

Diese Ergebnisse erklären sich folgendermaßen: Da (a) eine fehlerfreie Implementierung des Fensterhebers beschreibt, sind alle Formeln wahr. Die erste und zweite Formel sind auf (b) wahr, da p schon im ersten Zustand abgearbeitet ist. Die dritte Formel ist auf (b) falsch, denn es gibt einen Pfad (siehe oben), sodass nicht immer p gilt. Allerdings gibt es auch Pfade (nämlich diejenigen, die Pfaden in (a) entsprechen), sodass p immer erfüllt ist. Deswegen gilt auch für (b): $EG p$.

11.1.4 Zusammenfassung

Die formale Logik erlaubt die Beschreibung des Verhältnisses von Aussagen oder Objekten zueinander. Dies geschieht durch Angabe logischer Formeln, die diese Verhältnisse präzise beschreiben. Einer Formel kann durch Festlegung, welchen Wahrheitswert eine Aussage beziehungsweise welchen Wert ein Objekt hat, ein Wahrheitswert (wahr oder falsch) zugeordnet werden.

Es gibt verschiedene Bereiche der Logik. Stellvertretend wurden Aussagenlogik, Prädikatenlogik und temporale Logik betrachtet. Die Aussagenlogik ist Basis der beiden anderen Logiken und beschreibt das Verhältnis einzelner Aussagen zueinander. Die Prädikatenlogik beschreibt das Verhältnis komplexerer Objekte und die Temporallogik betrachtet das Verhältnis von Aussagen, deren Wahrheitswert sich mit ablaufender Zeit verändern kann.

Zur Klassifikation

- Formeln der Aussagenlogik, der Prädikatenlogik und der temporalen Logik werden textuell und formal beschrieben.
- Sie werden in der Regel in der Spezifikations- und Entwurfsphase eingesetzt, in der Implementierungsphase aber auch zur Programmverifikation.
- Sie werden in technischen und softwareintensiven Anwendungen benutzt.

11.2 Constraints und die OCL in der UML

In der UML ist es möglich, UML-Modellelementen *Constraints* – übersetzt logische Einschränkungen, logische Bedingungen – zuzuordnen, die die möglichen Inhalte, Zustände oder die Semantik von Modellelementen einschränken und immer erfüllt sein müssen:

»A constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element « [UML07, S. 41]

Constraints können folgendermaßen geschrieben werden:

Notationen

- In natürlicher Sprache,
- in einer Programmiersprache,
- in einer mathematischen Notation oder
- in OCL (*Object Constraint Language*).

Zunächst werden die ersten drei Möglichkeiten beschrieben:

- »Constraints in der UML«, S. 378

Ergänzend zur UML gibt es die OCL zur formalen Spezifikation von *Constraints*:

- »OCL«, S. 380

Constraints für Assoziationen

Auch für Assoziationen können *Constraints* spezifiziert werden.

- In der Abb. 11.2-2 legt {xor} fest, dass ein Artikel entweder in einem Hochregallager *oder* in einem Lagerraum lagert. Allgemein ausgedrückt: Zu jedem beliebigen Zeitpunkt kann nur eine der Assoziationen, die von »Artikel« ausgehen, gelten.
- Die reflexive Assoziation der Klasse Angestellter enthält ein selbstdefiniertes *Constraint*. Es besagt, dass für jede Objektbeziehung zwischen einem Chef und einem Mitarbeiter gilt: `chef.gehalt > mitarbeiter.gehalt`. Hier wird der Rollennamen verwendet, um ein Attribut eindeutig zu kennzeichnen. Wie das Objektdiagramm zeigt, ist das Gehalt des Mitarbeiters a2 geringer als das Gehalt des eigenen Chefs, kann aber durchaus höher sein als das Gehalt des Angestellten a3, der ebenfalls Chef ist.

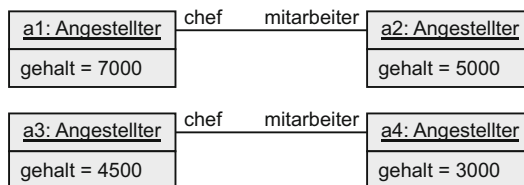
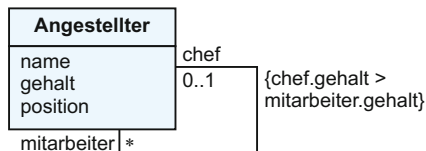
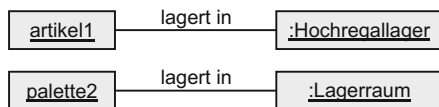
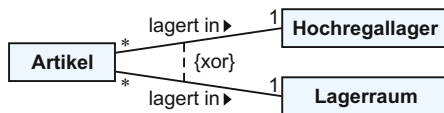


Abb. 11.2-2:
Constraints für
Assoziationen.

Wie die Beispiele zeigen, kann sich ein *Constraint* auf eine einzige Assoziation oder auf mehrere Assoziationen beziehen. Ein {xor} kann auch mehr als zwei Assoziationen umfassen.

III 11 Logik

Beispiel Wäre in der Abb. 11.2-3 kein *Constraint* angegeben, so könnte jeder Student bei allen Professoren – auch außerhalb seines Fachbereichs – Vorlesungen belegen. Mit *Constraint* kann der Student s1 beim Professor p1 eine Vorlesung hören, aber nicht beim Professor p3.

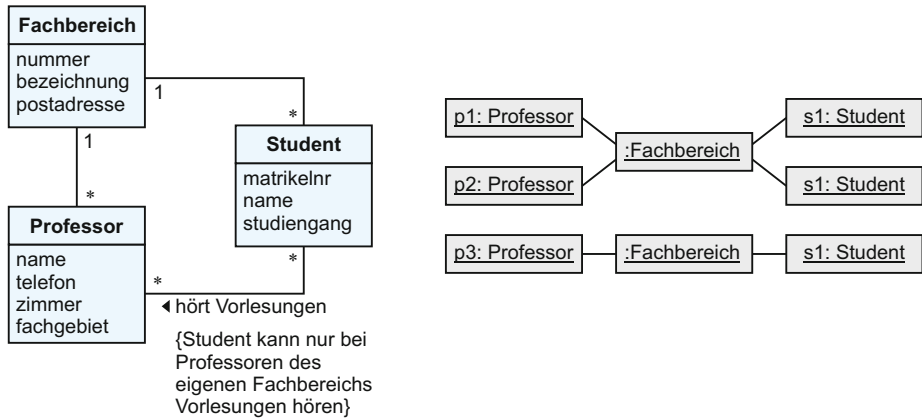


Abb. 11.2-3: Besteht eine Restriktion zwischen den Attributwerten zweier Objekte, dann ist damit auch immer eine Restriktion für die Assoziation zwischen diesen Objekten vorhanden. Ein Beispiel zeigt die Abb. 11.2-4.



11.2.2 OCL

»The Object Constraint Language (OCL) [is] a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model« [OCL06, S. 5].

Die **OCL** ist Bestandteil der UML und erlaubt es, UML-Modellen zusätzliche präzierte Semantik zuzuordnen, die mit UML-Elementen nicht oder nur umständlich ausgedrückt werden kann. OCL ist eine standardisierte, typisierte, deklarative und seiteneffektfreie Abfrage- und Constraint-Sprache mit einer mathematisch fundierten Semantik. Sie basiert auf der Mengentheorie und hat Ähnlichkeit mit der Programmiersprache Smalltalk und mit der Prädikatenlogik (siehe »Prädikatenlogik«, S. 367).

Möglichkeiten Mit Hilfe der OCL ist es möglich,

- Anfragen zu spezifizieren,
- Werte zu referenzieren,

- entlang von Assoziationen zu navigieren,
- Zustände zu definieren.

OCL besteht aus einzelnen Ausdrücken. Vor dem Ausdruck wird der Kontext angegeben, auf den sich der Ausdruck bezieht: Syntax

```
[ context Modellelement ] OCL-Ausdruck
```

Die Kontext-Angabe entfällt, wenn der OCL-Ausdruck in einem UML-Diagramm durch eine gestrichelte Linie einem UML-Modellelement zugeordnet ist (Formulierung als Notiz (*note*)).

Ein Kommentar wird durch zwei Minuszeichen begonnen und endet mit dem Zeilenumbruch (Zeilenumkommentar): -- OCL-Kommentar

Invarianten

Häufig werden Invarianten durch einen OCL-Ausdruck spezifiziert. Eine Invariante muss für alle Exemplare des entsprechenden Typs oder eine Assoziation zu jedem Zeitpunkt wahr sein. Die Referenz auf ein Exemplar eines Typs wird durch `self` angegeben, d.h. `self` bezieht sich immer auf das Exemplar, für das das *Constraint* berechnet wird. Sichtbarkeiten von Attributen u.ä. werden durch die OCL ignoriert.

In einer Klasse `Artikel` gibt es das Attribut `lagermenge`. Wenn die Lagermenge für einen Artikel immer größer 100 sein soll, dann kann dies, wie folgt, in der OCL spezifiziert werden: Beispiel 1a

```
context Artikel inv: self.lagermenge > 100
```

Das Schlüsselwort `inv` gibt an, dass es sich um eine Invariante handelt. Diese Spezifikation ist identisch mit der Angabe `lagermenge {lagermenge > 100}` in einem Klassendiagramm (siehe »Constraints in der UML«, S. 378).

Der Invarianten kann zusätzlich auch ein eigener Name gegeben werden (benannte Invariante):

```
context Artikel inv Mindestlagermenge: self.lagermenge > 100
```

```
[ context Modellelement ]
```

```
inv [Constraint-Name] : self.eigenschaft
```

Syntax

Vor- und Nachbedingungen

Für Operationen oder andere Verhaltenselemente können Vorbedingungen (*preconditions*) und Nachbedingungen (*postconditions*) spezifiziert werden, ohne dass dafür ein Algorithmus angegeben wird. Vorbedingungen (Schlüsselwort `pre`) müssen vor der Ausführung, Nachbedingungen (Schlüsselwort `post`) nach der Ausführung gültig sein.

III 11 Logik

Beispiel 1b In der Klasse Artikel gibt es die Operation einlagern (menge:Integer):Integer. Folgende Nachbedingung lässt sich zu dieser Operation spezifizieren:

```
context Artikel::einlagern(menge:Integer):Integer  
post: self.lagermenge = self.lagermenge@pre + menge
```

Mit dem Schlüsselwort @pre wird angegeben, dass der Wert von lagermenge (vor Ausführung der Operation) nach der Ausführung um den Wert von menge erhöht ist. @pre ist nur in Nachbedingungen erlaubt.

Syntax [context Typname::
 operationName(param1 : Type, ...).
 Returntype]
 pre: param1 > ...
 post: [result =]...
 result ist ein Schlüsselwort in der OCL und steht für den Ergebniswert der Operation (falls vorhanden).

Beispiel In einer Klasse Veranstaltung gibt es die Attribute tagesrasterAnfang:Integer und tagesrasterEnde:Integer, wobei als Wert die Stunde als ganze Zahl angegeben wird, z. B. 10 für 10 Uhr. Die Operation dauerErmitteln():Integer ermittelt die Dauer pro Tag:

```
context Veranstaltung::dauerErmitteln():Integer  
inv: self.tagesrasterEnde > self.tagesrasterAnfang  
post: result = self.tagesrasterEnde - self.tagesrasterAnfang  
result ist der Ergebniswert der Operation.
```

Voreinstellungswerte und abgeleitete Werte

Beispiel 2a Die Abb. 11.2-5 zeigt ein Beispiel für eine Initialisierung und eine Ableitung.

Typen und Operationen in der OCL

Jeder OCL-Ausdruck hat einen vordefinierten oder einen benutzerdefinierten Typ. Vordefinierte Typen sind die Basistypen Boolean, Integer, Real und String. Benutzerdefinierte Typen sind alle Klassentypen und Aufzählungstypen.

OCL definiert ein eigenes *Collection Framework* mit *Collection Types* (Abb. 11.2-6).

In Abhängigkeit vom *Collection Type* gibt es 22 Operationen mit unterschiedlicher Semantik. Wichtige Operationen sind:

- Verschiedene including- und excluding-Operationen:
- count(object): Kardinalität von object in der *Collection*
- excludes(object): true, wenn object nicht in der *Collection* enthalten

11.2 Constraints und die OCL in der UML III

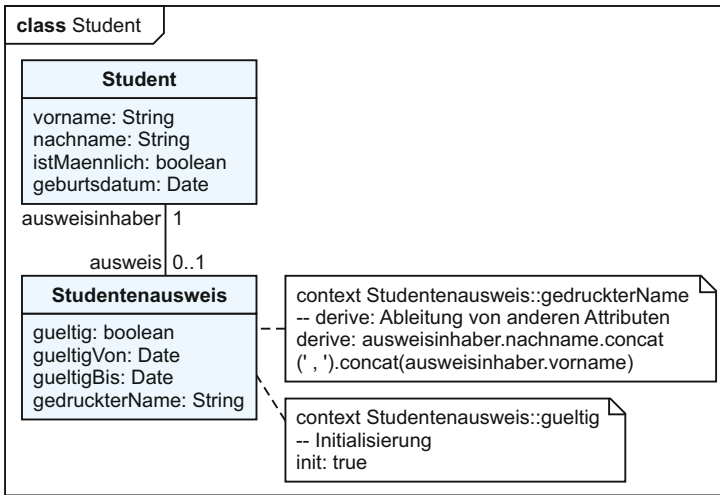


Abb. 11.2-5:
Beispiel für die
OCL-Spezifikation
einer
Initialisierung und
Ableitung.

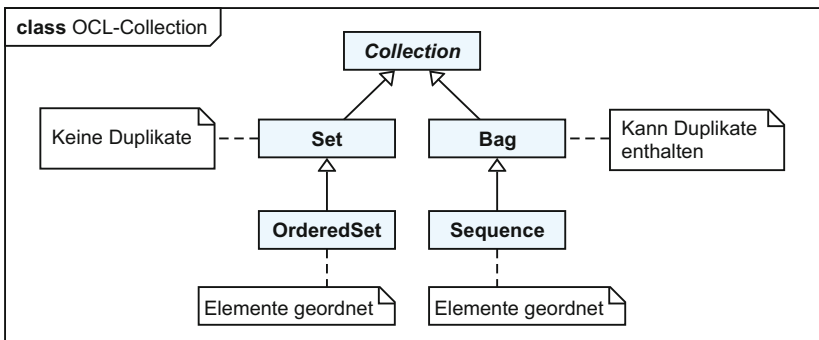


Abb. 11.2-6: Das
»Collection
Framework« der
OCL.

- ❑ `excludesAll(collection)`: true, wenn alle Elemente aus *collection* nicht enthalten
- ❑ `includes(object)`, `includesAll(collection)`: analog zu `excludes`
- ❑ `isEmpty()`: true für leere *Collection*
- ❑ `notEmpty()`: true für nicht leere *Collection*
- ❑ `size()`: Anzahl der Elemente in der *Collection*
- ❑ `sum()`: Summe der Elemente in der *Collection*
- Vergleichsoperationen: `=`, `<>`
- Mengenoperationen: `union`, `intersection`, `minus`, `symmetricDifference`
- Operationen auf sortierten *Collections*: `first()`, `last()`, `indexOf()` u. a.
- Konvertierungsoperationen: `asBag()`, `asSet()`, `asOrderedSet()`, `asSequence()`
- Iterierende Operationen auf allen *Collection Types*: `any(expr)`, `collect(expr)`, `exists(expr)`, `forAll(expr)`, `isUnique(expr)`, `one(expr)`, `select(expr)`, `reject(expr)`, `sortedBy(expr)` u. a.

III 11 Logik

Mengenoperationen auf *Collections* werden in der Pfeilnotation -> geschrieben.

OCL-Ausdruck im Operationsrumpf

Ein OCL-Ausdruck kann im Operationsrumpf (*body*) angegeben werden, um das Ergebnis einer Abfrageoperation zu spezifizieren. Der Ausdruck muss kompatibel zum Ergebnistyp der Operation sein. Vor- und Nachbedingungen können mit dem Rumpf-Ausdruck kombiniert werden.

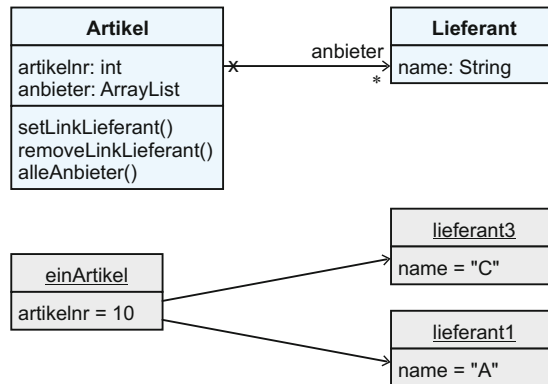
Beispiel 2b Es soll ermittelt werden, wie viele Studenten männlich sind (siehe Abb. 11.2-5):

context Student::ermittleAnzahlMaennlicheStudenten(): Integer
body: self.istMaennlich -> size()

Beispiel Es sollen die Namen aller anbieter ermittelt werden (Abb. 11.2-7):

context Artikel::ermittleAnbiernamen(): Bag(String)
body: anbieter -> collect(name)

Abb. 11.2-7:
Unidirektionale
Assoziation mit
der Multiplizität *
sowie ein
Objektszenario.



Syntax [context Typname::
operationName(param1 : Type, ...). Returntype]
body: -- OCL-Ausdrücke

Navigation über Assoziationen

Ausgehend von einer Klasse kann über Assoziationen zu anderen Klassen und deren Eigenschaften navigiert werden. Um über eine Assoziation zu einer anderen Klasse zu navigieren, wird der Rollenname am gegenüberliegenden Ende der Assoziation zum Navigieren verwendet. Fehlt der Rollenname, dann wird der Klassenname kleingeschrieben verwendet. Navigationen werden in der OCL als Attribute behandelt (Punkt-Notation).

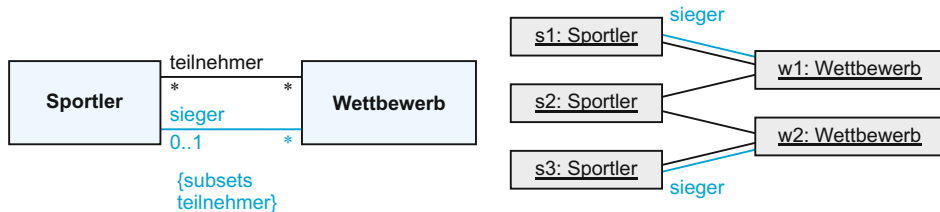
Der OCL-Ausdruck

Beispiel

context Wettbewerb

inv: self.teilnehmer -> includes(self.sieger)

besagt (Abb. 11.2-8), dass die Mengenoperation `includes()`, die auf die Menge der `teilnehmer` angewandt wird und der als Argument der `sieger` mitgegeben wird, `true` ergeben muss. `self` ist hier ein Exemplar der Klasse `Wettbewerb`. D.h. der Sieger eines Wettbewerbs muss zur Menge der Teilnehmer gehören.



Folgende Constraints sollen bei der Klasse `Person` (Abb. 11.2-9) gelten:

- Das Alter einer `Person` ist nicht negativ:

context `Person` **inv:** `self.alter >= 0`

- Verheiratete Personen sind älter als 18 Jahre:

context `Person` **inv:**

`self.ehefrau -> notEmpty()` implies `self.ehefrau.alter >= 18`
and

`self.ehemann -> notEmpty()` implies `self.ehemann.alter >= 18`

a implies b bedeutet: Wenn a `true` ist, dann soll auch b `true` sein.

Abb. 11.2-8:

Darstellung von zwei Assoziationen zwischen zwei Klassen, ergänzt um eine mögliche Objekt-Konstellation.

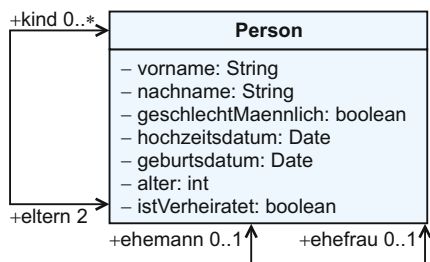


Abb. 11.2-9:

Beispiel für eine Klasse mit zwei reflexiven Assoziationen.

- + Mehrdeutigkeiten in UML-Modellen können durch OCL-Ausdrücke aufgelöst werden.

Bewertung

- + Fügt UML-Modellen eine präzisierte Semantik hinzu.
- Inkonsistente Spezifikationen, d.h. eine Kombination von *Constraints*, die sich widersprechen, sind schwer aufzufinden.
- Die Verwendung von OCL für eine modellgetriebene Codegenerierung (MDA) ist begrenzt, da ausführbare Operationen, wie Zuweisungen und Objekterzeugung, fehlen.

III 11 Logik

- Nachbedingungen beziehen sich nur auf lokale Attribute und Assoziationen.

Literatur [OCL06], [WaKl99]

11.2.3 Zusammenfassung

Constraints in der UML sind Bedingungen oder Zusicherungen, die immer wahr sein müssen. Ihr Zweck ist es, die Semantik eines Elements genauer zu modellieren. Bestimmte *Constraints* sind in der UML schon vordefiniert, weitere können hinzugefügt werden.

Die OCL (*Object Constraint Language*) ist eine deklarative, präzise, textuelle Sprache zur Beschreibung von Regeln, die Einschränkungen für UML-Modellelemente festlegt, die grafisch nicht ausgedrückt werden können. Außerdem können Abfrage-Ausdrücke formuliert werden.

Zur Klassifikation

- *Constraints* werden als textuelle Annotationen UML-Modellelementen zugeordnet.
- Der Grad der Formalität von *Constraints* reicht von informal bis formal.
- Die OCL wird in textueller, formaler Form unabhängig von Modellelementen oder zugeordnet zu Modellelementen beschrieben.
- *Constraints* und die OCL werden in der Spezifikations- und Entwurfsphase eingesetzt.
- Sie werden in allen Anwendungsbereichen verwendet.

11.3 Entscheidungstabellen und Entscheidungsbäume

Mit Hilfe von **Entscheidungstabellen** (ET) können vorzunehmende Aktionen oder Handlungen, die von der Erfüllung oder Nichterfüllung mehrerer Bedingungen abhängen, kompakt und übersichtlich definiert werden.

Eine Entscheidungstabelle besteht aus vier Quadranten (Abb. 11.3-1). Die Bedingungen werden oben links, die Aktionen links unten jeweils in sich durchnummeriert in die Tabelle eingetragen. Die beiden rechten Quadranten dienen dazu, die Bedingungen mit den Aktionen zu verknüpfen (siehe unten). Die Verknüpfung geschieht spaltenweise. Jede Spalte wird als Regel bezeichnet und durch eine Regelnummer identifiziert. Im Prinzip ist eine Entscheidungstabelle nichts anderes als eine übersichtliche Darstellung des Sachwissens in der Form »wenn ..., dann ...«.

11.3 Entscheidungstabellen und Entscheidungsbäume III

	Name der ET	Regelnummern
wenn	Bedingungen	Bedingungsanzeiger
dann	Aktionen	Aktionsanzeiger

Abb. 11.3-1:
Struktur einer ET.

Eine Entscheidungstabelle kann systematisch erstellt werden:

- »Erstellung einer Entscheidungstabelle«, S. 387

Die Anwendung einer Entscheidungstabelle erfolgt in mehreren Schritten:

- »Anwendung einer Entscheidungstabelle«, S. 389

Erstellte Entscheidungstabellen können überprüft und optimiert werden:

- »Überprüfung und Optimierung von Entscheidungstabellen«, S. 391

Es gibt mehrere Möglichkeiten, Entscheidungstabellen zu beschreiben:

- »Darstellungsformen für Entscheidungstabellen«, S. 393

Oft werden mehrere Entscheidungstabellen in Kombination benötigt:

- »Entscheidungstabellen-Verbunde«, S. 394

Erweiterte Bedingungs- und Aktionsanzeiger erlauben eine kompakte Darstellung:

- »Erweiterte Entscheidungstabellen«, S. 399

Können zu einem Zeitpunkt mehrere Regeln zutreffen, dann handelt es sich um Mehrtreffer-Tabellen:

- »Eintreffer- und Mehrtreffer-Entscheidungstabellen«, S. 400

Zusammenfassend wird ein Überblick gegeben:

- »Zusammenfassung & Bewertung«, S. 402

Entscheidungstabellen wurden 1957 in einer Projektgruppe der General Electric Company entwickelt und sind in [DIN66241] genormt.

Zur Historie

11.3.1 Erstellung einer Entscheidungstabelle

Die konkrete Erstellung einer Entscheidungstabelle wird an einem Beispiel »Scheck einlösen« gezeigt.

Methode zur
Erstellung von ET

Ein Sachbearbeiter in einer Bank soll bei der Einlösung von Schecks folgende Regeln beachten:

Beispiel 1a

/1/ Wenn die vereinbarte Kreditgrenze des Ausstellers eines Schecks überschritten wird, das bisherige Zahlungsverhalten aber einwandfrei war und der Überschreibungsbetrag kleiner als € 500,- ist, dann soll der Scheck eingelöst werden.

III 11 Logik

/2/ Wenn die Kreditgrenze überschritten wird, das bisherige Zahlungsverhalten einwandfrei war, aber der Überschreibungsbetrag über € 500,- liegt, dann soll der Scheck eingelöst und dem Kunden sollen neue Konditionen vorgelegt werden.

/3/ War das Zahlungsverhalten nicht einwandfrei, wird der Scheck nicht eingelöst.

/4/ Der Scheck wird eingelöst, wenn der Kreditbetrag nicht überschritten ist.

1. Schritt: Ermittlung der Aktionen

Als Erstes werden aus einer verbalen Problembeschreibung die durchzuführenden Aktionen ermittelt.

Beispiel 1b A1: Scheck einlösen /1/, /2/, /4/
A2: Scheck nicht einlösen /3/
A3: Neue Konditionen vorlegen /2/

2. Schritt: Ermittlung der Bedingungen

Beispiel 1c B1: Kreditgrenze überschritten? /1/, /2/, /4/
B2: Zahlungsverhalten einwandfrei? /1/, /3/
B3: Überschreibungsbetrag < € 500,-? /2/

3. Schritt: Eintrag der Bedingungen & Aktionen in die ET

Die Aktionen werden in den linken unteren, die Bedingungen in den linken oberen Quadranten eingetragen. Ist die Anzahl der Bedingungen kleiner 5, dann empfiehlt es sich, zunächst eine vollständige Entscheidungstabelle anzulegen.

4. Schritt: Eintrag aller Bedingungskombinationen

Eine formal vollständige Entscheidungstabelle liegt vor, wenn alle möglichen Bedingungskombinationen im Quadranten »Bedingungsanzeiger« eingetragen sind. Bei n Bedingungen gibt es 2^n mögliche Bedingungskombinationen.

Formal vollständige ET Ist eine Bedingung erfüllt, dann lautet der Bedingungsanzeiger J (für Ja) oder Y (für Yes); ist eine Bedingung nicht erfüllt, dann wird N (für Nein bzw. No) eingetragen. Ist es gleichgültig (irrelevant), ob die Bedingung erfüllt ist, dann wird (-) eingesetzt (siehe unten).

5. Schritt: Eintrag der Aktionsanzeiger

Im letzten Schritt wird jede Bedingungskombination betrachtet und entsprechend der Problembeschreibung im Aktionsanzeiger-Quadranten ein Kreuz (X) eingetragen, wenn eine entsprechende Aktion auszuführen ist.

11.3 Entscheidungstabellen und Entscheidungsbäume III

Für die Scheckeinlösung ergibt sich die Entscheidungstabelle der Abb. 11.3-2, wobei der Aktionsteil noch um eine zusätzliche »unlogisch«-Aktion erweitert wurde. Die Regeln R6 und R8 sind »unlogisch« bzw. fachlich nicht möglich. Ist die Kreditgrenze nicht überschritten (B1), dann gibt es keinen Überschreibungsbetrag (B3).

Beispiel 1d

		Regel							
ET1: Scheckeinlösung		R1	R2	R3	R4	R5	R6	R7	R8
B1	Kreditgrenze überschritten?	J	J	J	J	N	N	N	N
B2	Zahlungsverhalten einwandfrei?	J	J	N	N	J	J	N	N
B3	Überschreibungsbetrag < 500,- €?	J	N	J	N	J	N	J	N
A1	Scheck einlösen	X	X			X		X	
A2	Scheck nicht einlösen			X	X				
A3	neue Konditionen vorlegen		X						
A4	unlogisch						X		X

Bedingungs-
kombinationen
($2^3 = 8$ Kombi-
nationen)

Durch-
zuführende
Aktionen

Abb. 11.3-2: Entscheidungstabelle für eine Scheckeinlösung.

11.3.2 Anwendung einer Entscheidungstabelle

Liegt eine fertige Entscheidungstabelle vor, dann kann sie folgendermaßen ausgewertet werden:

1. Schritt: Ermittlung der konkreten Bedingungsanzeiger

Im ersten Schritt ermittelt man für eine konkrete Situation, ob die aufgeführten Bedingungen erfüllt sind oder nicht. Man erhält eine bestimmte Konstellation der Bedingungsanzeiger, auch Auswertungsvektor genannt.

Ein Kunde will einen Scheck einlösen (siehe Beispiel »Scheck einlösen«, »Erstellung einer Entscheidungstabelle«, S. 387). Der Sachbearbeiter stellt fest, dass die Kreditgrenze überschritten ist, das Zahlungsverhalten bisher einwandfrei war und der Überschreibungsbetrag nicht kleiner € 500,- ist. Es ergibt sich der Auswertungsvektor: (J, J, N).

Beispiel 1a

2. Schritt: Auswertungsvektor über Bedingungsanzeiger schieben

Eine Möglichkeit, eine Entscheidungstabelle auszuwerten, besteht darin, den Auswertungsvektor z.B. von links nach rechts über die Bedingungsanzeiger der Regeln zu schieben. Stimmen die Vektorwerte mit den Bedingungsanzeigern einer Regel überein, dann wird die entsprechende Regel angewandt.

3. Schritt: Bei Gleichheit angekreuzte Aktionen ausführen

Es wird im Aktionsanzeigerteil der Regel nachgesehen, wo ein Kreuz (X) steht und dann werden die Aktionen in den angekreuzten Zeilen ausgeführt. Alle angekreuzten Aktionen sind mit »und« verknüpft, die Reihenfolge der Ausführung ist aber nicht vorgeschrieben. Diese Art der Auswertung zeigt, dass die Bedingungen ebenfalls immer mit »und« verknüpft sind.

Beispiel 1b Der Auswertungsvektor (J, J, N) ist identisch mit dem Bedingungsanzeigervektor der Regel R2. Dementsprechend werden die Aktionen A1 und A3 ausgeführt.

Die oben angegebene Auswertungsreihenfolge ist für eine Entscheidungstabelle aber *nicht* vorgeschrieben. Der Auswertungsvektor kann auch von rechts nach links oder in beliebiger Reihenfolge über die Bedingungsanzeiger der Regeln geschoben werden.

Eine andere Möglichkeit besteht darin, die Entscheidungstabelle zeilenorientiert auszuwerten. Zuerst wird geprüft, ob die erste Bedingung zutrifft oder nicht. Ergibt sich beispielsweise ein J, dann werden alle mit N in der ersten Zeile beginnenden Regeln gestrichen. Dann wird die zweite Bedingung geprüft. Ergibt sich beispielsweise wieder ein J, dann werden alle mit N in der zweiten Zeile beginnenden Regeln von den verbliebenen Regeln gestrichen. Die Fortsetzung dieses Verfahrens ergibt bei n Bedingungen nach n Schritten die zutreffende Regel.

ET ist kein Algorithmus

Diese verschiedenen Auswertungsmöglichkeiten zeigen deutlich, dass eine Entscheidungstabelle nicht gleichzusetzen ist mit einem Algorithmus, bei dem die Auswertungsreihenfolge eindeutig vorgegeben ist.

ET ist Menge aussagenlogischer Regeln

Eine Entscheidungstabelle ist vielmehr eine Menge von aussagenlogischen Regeln. Bei der Auswertung werden die Regeln bestimmt, deren Prämissen erfüllt sind. Die Alternativen liegen dann in der Konfliktauflösungsstrategie.

Werden in der Implementierungsphase prozedurale oder objektorientierte Programmiersprachen verwendet, dann erfolgt eine Transformation der Entscheidungstabelle in eine Kontrollstruktur, die eine eindeutige Abarbeitungsfolge festlegt. Werden deklarative oder regelbasierte Sprachen verwendet, dann ist dies *nicht* notwendig.

11.3.3 Überprüfung und Optimierung von Entscheidungstabellen

Bei der Entscheidungstabelle »Scheck einlösen« (siehe »Anwendung einer Entscheidungstabelle«, S. 389) handelt es sich um eine formal **vollständige Entscheidungstabelle**, da im Bedingungsanzeigerteil alle möglichen Kombinationen auftreten. Formal vollständig

Betrachtet man die Regeln R6 und R8, dann sieht man, dass diese Bedingungskombinationen nicht möglich sind. Wenn beispielsweise die Kreditgrenze nicht überschritten ist, dann gibt es auch keinen Überschreibungsbetrag. Die Regeln R6 und R8 können daher gestrichen werden. Es ergibt sich eine inhaltlich vollständige Entscheidungstabelle. Beispiel 1a

Eine Entscheidungstabelle ist inhaltlich vollständig, wenn alle praktisch möglichen Problemkombinationen in der Entscheidungstabelle aufgeführt sind. Inhaltlich vollständig

Vollständige Entscheidungstabellen kann man versuchen zu optimieren. Eine Verdichtung überführt eine vollständige Entscheidungstabelle in eine konsolidierte Entscheidungstabelle. Eine Konsolidierung führt man in folgenden Schritten durch: Optimierung, Konsolidierung

- 1 Identische Aktionen? Zuerst wird nachgesehen, ob es Regeln mit identischen Aktionen gibt.
- 2 Paarweise Betrachtung: Wenn ja, dann werden jeweils zwei dieser Regeln paarweise betrachtet.
- 3 Irrelevanzanzeiger einfügen: Wenn sich die Bedingungsanzeiger der zwei betrachteten Regeln nur in einer Zeile unterscheiden, dann werden die beiden Regeln zu einer Regel zusammengefasst und die beiden unterschiedlichen Bedingungsanzeiger durch einen so genannten Irrelevanzanzeiger (dargestellt durch »?«) ersetzt.

Betrachtet man die Entscheidungstabelle »Scheck einlösen«, dann sieht man, dass beispielsweise die Aktionen der Regeln R3 und R4 identisch sind. Die zugehörigen Bedingungsanzeiger sind bis auf B3 gleich. Ob die Bedingung B3 erfüllt ist oder nicht, spielt für die durchzuführende Aktion keine Rolle. Unabhängig von B3 wird immer A2 ausgeführt. Daher kann man die Regeln R3 und R4 löschen und zu einer neuen Regel R3/4 zusammenfassen mit den Bedingungsanzeigern (J,N,-). Die Aktionen der Regeln R5 und R7 sind ebenfalls identisch. Eine Zusammenfassung ergibt die Bedingungsanzeiger (N, -, J). Es ergibt sich die konsolidierte Entscheidungstabelle der Abb. 11.3-3. Aus acht Regeln sind vier Regeln geworden. Beispiel 1b

III 11 Logik

Abb. 11.3-3:
Konsolidierte Entscheidungstabelle
für eine
Scheckeinlösung.

ET1: Scheckeinlösung		R1	R2	R3/4	R5/7
B1	Kreditgrenze überschritten?	J	J	J	N
B2	Zahlungsverhalten einwandfrei?	J	J	N	–
B3	Überschreibungsbetrag < 500,– €?	J	N	–	J
A1	Scheck einlösen	X	X		X
A2	Scheck nicht einlösen			X	
A3	neue Konditionen vorlegen		X		

Eine Entscheidungstabelle konsolidiert man, indem man die Regeln zusammenfasst, die unerhebliche Bedingungen enthalten. Dadurch enthält die Tabelle jetzt neben J und N auch das Unerheblichkeitszeichen »–«.

Else-Regel Eine weitere Möglichkeit zur Reduzierung der Regelanzahl bietet die *Else*-Regel. In ihr können alle Regeln zusammengefasst werden, in denen die gleichen Aktionen angekreuzt sind. Allerdings ist pro Tabelle nur eine *Else*-Regel möglich. Die *Else*-Regel muss als solche gekennzeichnet werden. Durch das Einfügen einer *Else*-Regel in eine Entscheidungstabelle wird aus einer unvollständigen eine vollständige Entscheidungstabelle, da alle nicht beachteten Entscheidungssituationen in der *Else*-Regel enthalten sind.

Die *Else*-Regel legt also fest, welche Aktionen durchzuführen sind, wenn keine andere Regel der Entscheidungstabelle zutrifft. Bei der Anwendung einer Entscheidungstabelle werden daher immer erst alle anderen Regeln überprüft. Eine *Else*-Regel besitzt folgende Eigenschaften:

- Sie wird durch *Else* gekennzeichnet.
- Sie hat keine Bedingungsanzeiger.
- Ihre Position innerhalb des Regelteils ist beliebig, es empfiehlt sich aber, sie ganz rechts anzuordnen.
- Sie kann, muss aber nicht verwendet werden.

Beispiel 1c In der Entscheidungstabelle aus Beispiel 1b können die Regeln R1 und R5/7 zu einer *Else*-Regel zusammengefasst werden (Abb. 11.3-4).

Abb. 11.3-4: Entscheidungstabelle mit *Else* für eine Scheckeinlösung.

ET1: Scheckeinlösung (optimiert)		R2	R3/4	Else
B1	Kreditgrenze überschritten?	J	J	
B2	Zahlungsverhalten einwandfrei?	J	N	
B3	Überschreibungsbetrag < 500,– €?	N	–	
A1	Scheck einlösen	X		X
A2	Scheck nicht einlösen		X	
A3	neue Konditionen vorlegen	X		

11.3 Entscheidungstabellen und Entscheidungsbäume III

Der Vorteil einer optimierten Entscheidungstabelle liegt darin, dass sie leichter angewendet werden kann, während die vollständige Entscheidungstabelle leichter zu erstellen ist.

11.3.4 Darstellungsformen für Entscheidungstabellen

Neben der Darstellung einer Entscheidungstabelle durch vier Quadranten (vertikale Anordnung der Regeln) gibt es noch eine horizontale Anordnung der Regeln, die in der Praxis auch oft angewandt wird.

Horizontale Anordnung der Regeln

Die Abb. 11.3-5 zeigt die Entscheidungstabelle »Scheck einlösen« (siehe »Erstellung einer Entscheidungstabelle«, S. 387) in der alternativen Darstellungsform.

Beispiel 1a

Kreditgrenze überschritten	Zahlungsverhalten einwandfrei?	Überschreibungsbetrag < 500,-?	Scheck einlösen	Scheck nicht einlösen	neue Konditionen vorlegen	unlogisch
J	J	J	X			
		N	X		X	
	N	J		X		
		N		X		
N	J	J	X			
		N				X
	N	J	X			
		N				X

Die optimierte Entscheidungstabelle mit *Else*-Regel (siehe »Anwendung einer Entscheidungstabelle«, S. 389) zeigt die Abb. 11.3-6.

Abb. 11.3-5: Entscheidungstabelle mit horizontaler Anordnung für eine Scheckeinlösung.

Um einem Gesprächspartner, der die Notation einer Entscheidungstabelle nicht kennt, das Lesen zu erleichtern, gibt es noch die Möglichkeit, eine Entscheidungstabelle als **Entscheidungsbaum** darzustellen. Ausgangspunkt ist die alternative Darstellungsform. Jedoch werden alle Alternativen explizit ausformuliert. Der Entscheidungsbaum wird von links nach rechts durchlaufen.

Die erste Entscheidungstabelle aus Beispiel 1a sieht als Entscheidungsbaum wie in der Abb. 11.3-7 aus.

Beispiel 1b

III 11 Logik

Kreditgrenze überschritten	Zahlungsverhalten einwandfrei?	Überschreibungsbetrag < 500,-?	Scheck einlösen	Scheck nicht einlösen	neue Konditionen vorlegen
J	J	N	X		
	N	–		X	X
	Else		X		

Abb. 11.3-6:
Optimierte Entscheidungstabelle für eine Scheckeinlösung.

Die alternativen Darstellungsformen einer Entscheidungstabelle sind semantisch nicht vollständig identisch mit der Darstellungsform durch vier Quadranten. Die alternativen Darstellungsformen legen eine Abarbeitung von links nach rechts nahe, während dies bei der Quadrantenform nicht festgelegt ist.

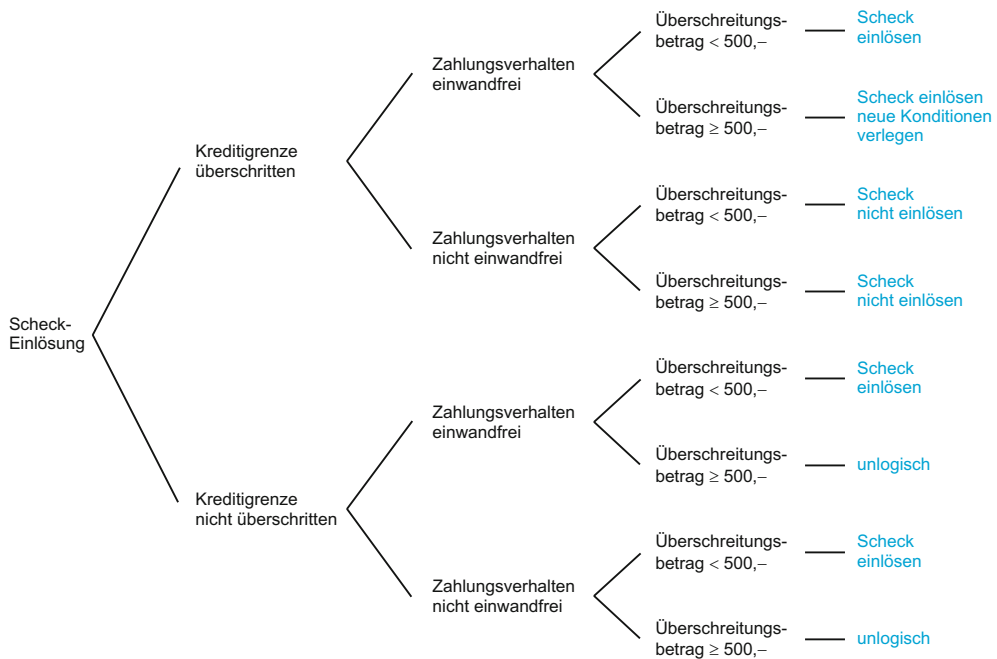


Abb. 11.3-7:
Darstellung einer Entscheidungstabelle als Entscheidungsbaum.

11.3.5 Entscheidungstabellen-Verbunde

Bei mehr als fünf Bedingungen ergibt sich eine so große Anzahl an Bedingungskombinationen, dass vollständige Entscheidungstabellen nicht mehr aufgestellt werden können. Es gibt zwei Möglichkeiten, dieses Problem zu lösen.

Die eine Möglichkeit besteht darin, den Entscheidungsprozess in zusammengehörende Teilentscheidungsprozesse aufzuteilen. Jeder Teilentscheidungsprozess wird durch eine eigene Entscheidungstabelle beschrieben. Die Zusammenhänge werden durch Entschei-

11.3 Entscheidungstabellen und Entscheidungsbäume III

dungstabellen-Verbunde hergestellt. Die andere Möglichkeit besteht darin, so genannte erweiterte Bedingungs- und Aktionsanzeiger in einer Entscheidungstabelle zu verwenden. Im Folgenden wird zunächst die erste Möglichkeit behandelt.

Ein Entscheidungsprozess kann durch die Verknüpfung mehrerer Entscheidungstabellen oder einer Entscheidungstabelle mit sich selbst beschrieben werden. Die Verknüpfungen werden durch geeignete Textteile, die in die Entscheidungstabelle eingetragen werden, hergestellt. Eine grafische Verknüpfung ist ebenfalls möglich. miteinander verknüpfte Entscheidungstabellen bilden einen **Entscheidungstabellen-Verbund**. Folgende Verknüpfungsformen sind möglich:

- Sequenz
- Verzweigung
- Schleife
- Schachtelung

Zwei Entscheidungstabellen bilden eine **Sequenz**, wenn die zweite Entscheidungstabelle der einzige unmittelbare Nachfolger der ersten ist (Abb. 11.3-8).

Eine **Verzweigung** liegt vor, wenn auf eine Entscheidungstabelle alternativ mehrere Entscheidungstabellen unmittelbar folgen (Abb. 11.3-9).

Eine Entscheidungstabelle bildet eine **Schleife**, wenn mindestens eine Regel unmittelbar zur erneuten Anwendung dieser Entscheidungstabelle führt (Abb. 11.3-10). Damit keine unendlichen Schleifen entstehen, müssen die Aktionen eine Rückwirkung auf die Bedingungen haben, so dass eine Terminierung sichergestellt ist.

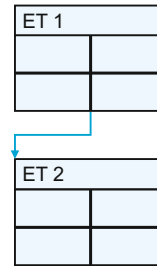


Abb. 11.3-8:
Sequenz von ET.

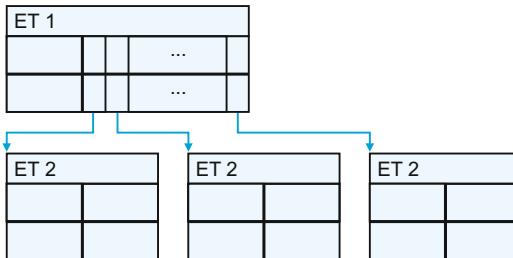


Abb. 11.3-9:
Verzweigung von
ET.

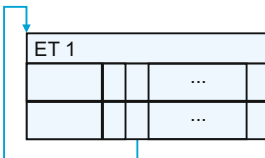
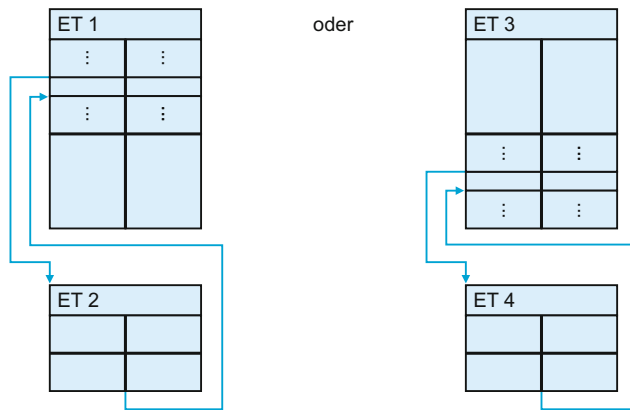


Abb. 11.3-10: ET-
Schleife .

III 11 Logik

Zwei Entscheidungstabellen sind **geschachtelt**, wenn zur Prüfung einer Bedingung oder bei der Ausführung einer Aktion die zweite Entscheidungstabelle angewandt wird (Abb. 11.3-11).

Abb. 11.3-11:
Schachtelung von
ET.



Beispiel 1a: Meldet sich ein Teilnehmer zu einer Seminarveranstaltung an, dann muss eine entsprechende Buchung durchgeführt werden. Im Pflichtenheft und im *Use Case* Buchen (siehe »*Use Case*-Diagramme und -Schablonen«, S. 255, Beispiel 1b) werden zur Buchung folgende Aussagen gemacht:

SemOrg

Anmeldung eines Kunden mit Überprüfung /F30/

- ob er bereits angemeldet ist
- ob die angegebene Veranstaltung existiert
- ob für die Veranstaltung noch Plätze frei sind
- wie die Zahlungsmoral ist

Folgende Aktionen werden gefordert:

- Buchung vornehmen
- Versand der Anmeldebestätigung und Rechnung /F60/
- Rechnungskopie an Buchhaltung /F130/

Folgende Aktionen lassen sich daraus ableiten:

- A1: Anmeldedaten eintragen
- A2: Teilnehmerzahl erhöhen
- A3: Anmeldebestätigung erstellen
- A4: Rechnung erstellen
- A5: Rechnungskopie erstellen
- A6: Kunden neu eintragen
- A7: Mitteilung »falsche Veranstaltung«
- A8: Mitteilung »ausgebucht«
- A9: Mitteilung »Zahlungsverzug«
- A10: Kundensachbearbeiter über Höhe Zahlungsverzug informieren und entscheiden lassen, ob A1 bis A5 erfolgt
- A11: Mitteilung »bereits angemeldet«

11.3 Entscheidungstabellen und Entscheidungsbäume III

Die Aktion A2 steht nicht explizit im Pflichtenheft. In /F100/ wird jedoch angegeben, dass zu jeder Veranstaltung die minimale, maximale und aktuelle Teilnehmerzahl gespeichert wird. Die Aktion A6 ergibt sich, wenn ein Teilnehmer bisher noch kein Kunde war. Da im Pflichtenheft nicht angegeben ist, wie bei einem Zahlungsrückstand zu verfahren ist, wird A10 eingeführt, damit der Sachbearbeiter entscheiden kann.

Folgende Bedingungen lassen sich aufstellen:

B1: Kunden-Nr. ok? /F70/

B2: Veranstaltungs-Nr. ok? /F100/

B3: Bereits angemeldet?

B4: Teilnehmerzahl aktuell < max?

B5: Zahlungsverzug? /F150/

Bevor eine Entscheidungstabelle aufgestellt wird, muss noch überprüft werden, ob es Abhängigkeiten in der Reihenfolge der Bedingungen und Aktionen gibt.

Abhängigkeiten der Reihenfolge

Analysiert man die Bedingungen, dann stellt man folgende Abhängigkeiten fest (Abb. 11.3-12):

Beispiel 1b

- B3 kann nur geprüft werden, wenn B1 und B2 erfüllt sind,
- B4 kann nur geprüft werden, wenn B2 erfüllt ist,
- B5 kann nur geprüft werden, wenn B1 erfüllt ist.

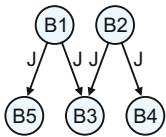


Abb. 11.3-12:
Abhängigkeiten zwischen Bedingungen.

Daraus ergibt sich, dass nur B1 und B2 unabhängig von anderen Bedingungen sind. Es lässt sich daher die Entscheidungstabelle der Abb. 11.3-13 aufstellen.

ET0: Erfasse Anmeldungen		R1	R2	R3	R4
B1	Personal-Nr. ok?	J	J	N	N
B2	Veranstaltungs-Nr. ok?	J	N	J	N
A6	Kunden neu eintragen			X	X
A7	Mitteilung »falsche Veranstaltung«		X		X
Weiter bei ET:		ET1	ET2	ET3	

Abb. 11.3-13: ET für eine Seminaranmeldung.

Trifft die Regel R1 zu, dann ist der Kunde bereits bekannt, und es gibt die gewählte Veranstaltung.

In der Entscheidungstabelle ET1 müssen nun noch die Bedingungen B3, B4 und B5 geprüft werden (Abb. 11.3-14).

III 11 Logik

Abb. 11.3-14: ET für »Buchung verarbeiten« und »Zahlungsmoral entscheiden«.

ET1: Buchung verarbeiten		R1	R2	R3	R4	Else
B3	Bereits angemeldet?	N	N	N	N	
B4	Teilnehmerzahl aktuell < max?	J	J	N	N	
B5	Zahlungsverzug?	J	N	J	N	
A11	Mitteilung »bereits angemeldet«					X
A8	Mitteilung »ausgebucht«			X	X	
A9	Mitteilung »Zahlungsverzug«	X		X		
Buchung vornehmen (A1 bis A5)			X			
Weiter bei ET:		ET1.1				

ET1.1: Zahlungsmoral entscheiden		R1
Zahlungsverzug kritisch? (Kundensachbearbeiter Höhe anzeigen)		N
Buchung vornehmen (A1 bis A5)		X

Die Regel R2 der Entscheidungstabelle ET0 gibt die Situation wieder, dass der Kunde bereits bekannt ist, er sich aber zu einer nicht existierenden Veranstaltung anmelden will. Da er sowieso eine Mitteilung erhält, kann gleich noch überprüft werden, ob er im Zahlungsverzug ist (Abb. 11.3-15).

Abb. 11.3-15: ET für »Zahlungsverzug prüfen«.

ET2: Zahlungsverzug prüfen		R1
B5	Zahlungsverzug?	J
A9	Mitteilung »Zahlungsverzug«	X

Die Regel R3 in der Entscheidungstabelle ET0 beschreibt die Situation, dass es sich um einen neuen Kunden handelt und dass die gewünschte Veranstaltung vorhanden ist. Es muss nur geprüft werden, ob die Veranstaltung ausgebucht ist (Abb. 11.3-16).

Abb. 11.3-16: ET für »Teilnehmerzahl prüfen«.

ET3: Teilnehmerzahl prüfen		R1	R2
B4	Teilnehmerzahl aktuell < max?	J	N
Buchung vornehmen (A1 bis A5)		X	
A8	Mitteilung »ausgebucht«		X

Aufgrund der Reihenfolgeabhängigkeiten der Bedingungen ergibt sich ein Entscheidungstabellen-Verbund, der die fachlich notwendige Entscheidungsreihenfolge widerspiegelt.

Analysiert man die in Beispiel 1 beschriebene fachliche Problemlösung, dann kann man zwei Arten von Wissen unterscheiden, das beschrieben wird.

11.3 Entscheidungstabellen und Entscheidungsbäume III

Die einzelnen Regeln in den Entscheidungstabellen beschreiben das **Fachwissen** (auch Sachwissen genannt). Das Fachwissen ist das dynamische Wissen über das Anwendungsgebiet, das zum Handeln (hier: Ausführen von Aktionen) oder zum Bilden von Schlussfolgerungen benötigt wird.

Fachwissen

Das Problemlösungswissen (auch Kontrollwissen genannt) bestimmt, in welcher Reihenfolge und unter welchen Randbedingungen eine Regel des Fachwissens anzuwenden ist. Es beschreibt also die Strategie und Taktik zum Erreichen einer Problemlösung.

Problemlösungs-
wissen

Das Problemlösungswissen steckt im Beispiel 1 in der Verknüpfung der verschiedenen Entscheidungstabellen: Die Regel R1 von ET0 sagt aus, dass zu ET1 verzweigt werden soll, wenn B1 und B2 zutreffen. Das Problemlösungswissen bestimmt also, welche Regelgruppen geprüft werden und wann ein Regelgruppenwechsel vollzogen wird.

Wissen kann auch zwischen Problemlösungswissen und Fachwissen liegen. Beispiele dafür sind die Regeln R2 und R3 in ET0. Beide beinhalten sowohl durchzuführende Aktionen als auch Verweise zu anderen Entscheidungstabellen.

11.3.6 Erweiterte Entscheidungstabellen

Eine andere Möglichkeit, Entscheidungstabellen mit vielen Bedingungen kompakt darzustellen, besteht darin, erweiterte Bedingungs- und Aktionsanzeiger zu verwenden. Es liegt dann eine **erweiterte Entscheidungstabelle** vor. Die bisher behandelten Tabellen waren **begrenzte Entscheidungstabellen**.

Erweiterte
Bedingungs- &
Aktionsanzeiger

Anstelle von (J, N, –) und (X) kann ein beliebiger Text im Anzeigerteil verwendet werden. Der Text im Bedingungsanzeigerteil muss erfüllt sein, damit die Regel zutreffen kann. Die durch den Aktions-text angegebene Aktion ist beim Zutreffen der Regel auszuführen.

Die Fluggesellschaft Softflight Worldwide Airline (SoftWAir) bietet ausschließlich die Flugziele Miami und New York an. Aus den Angaben Flugziel, Abflugdatum, Aufenthaltsdauer und Alter des Passagiers soll der Preisnachlass in Prozent bezogen auf den normalen Flugpreis aus einer Entscheidungstabelle entnommen werden können. Es gelten folgende Tarifbestimmungen:

Beispiel

- Personen ab dem 18. Lebensjahr erhalten für den Zielort Miami einen Preisnachlass von 20% (Ferientarif), falls das Abflugdatum nicht zwischen dem 21. und dem 30. Dezember liegt und die Aufenthaltsdauer mindestens 6 Tage beträgt.
- Für den Zielort New York gibt es keinen Ferientarif.
- Keinen Ferientarif gibt es für Personen, die jünger sind als 18 Jahre.

III 11 Logik

- Personen, die bereits 2 Jahre, aber noch nicht 18 Jahre alt sind, erhalten einen Preisnachlass von 30 %.
 - Kinder unter zwei Jahren fliegen zum Nulltarif (Preisnachlass 100 %).
 - Erfolgt kein Preisnachlass, dann wird der Wert 0 % angegeben.
- Es ergibt sich die Entscheidungstabelle der Abb. 11.3-17 mit erweiterten Anzeigern.

ET: SoftWAir	R1	R2	R3	R4	R5	R6
Alter?	≥18	≥18	≥18	≥18	≥2 u. <18	<2
Flugziel?	Miami	N.Y.	Miami	Miami	–	–
Aufenthalt ≥ 6 Tage?	J	–	N	–	–	–
Abflug zw. 21. u. 30.12.?	N	–	–	J	–	–
Preisnachlass in %	20	0	0	0	30	100

Abb. 11.3-17: **Durch Anwendung der Else-Regel lässt sich diese Tabelle noch weiter optimieren (Abb. 11.3-18).**
 Beispiel für eine ET mit erweiterten Anzeigern.

Abb. 11.3-18: ET mit Else für SoftWAir.

ET: SoftWAir	R1	R5	R6	Else
Alter?	≥18	≥2 u. <18	<2	
Flugziel?	Miami	–	–	
Aufenthalt ≥ 6 Tage?	J	–	–	
Abflug zw. 21. u. 30.12.?	N	–	–	
Preisnachlass in %	20	30	100	0

11.3.7 Eintreffer- und Mehrtreffer-Entscheidungstabellen

Eintreffer-ET Bei den bisher betrachteten Entscheidungstabellen handelt es sich um **Eintreffer-Tabellen**. Bei diesen ist zu einem Zeitpunkt höchstens eine der vorhandenen Regeln anwendbar. Egal, in welcher Reihenfolge man eine Entscheidungstabelle auswertet, gilt: Hat man eine gültige Regel gefunden, dann kann die Auswertung beendet werden. Bei Eintreffer-Tabellen schließen sich die Bedingungsanzeigerteile aller zugehörigen Regeln gegenseitig aus.

Mehrtreffer-ET Im Gegensatz dazu sind bei **Mehrtreffer-Tabellen** nicht-disjunkte Bedingungsanzeigerteile erlaubt. Diese können sowohl identisch sein, sich gegenseitig einschließen oder auch überschneiden. Ist bekannt, dass eine Mehrtreffer-Tabelle vorliegt, dann müssen bei der Auswertung alle Regeln geprüft werden. Die Aktionen aller zutreffenden Regeln werden vereint und nacheinander ausgeführt. Dies setzt jedoch voraus, dass sich die Aktionen nicht widersprechen.

11.3 Entscheidungstabellen und Entscheidungsbäume III

Die Entscheidungstabelle der Abb. 11.3-19 hat begrenzte Anzeiger und ist formal vollständig (Else-Regel). Liegt die Bedingungskonstellation (J, J, N, N, J) vor, dann treffen die Regeln R3 und R4 zu.

Beispiel

ET: Überprüfung von Eingaben		R1	R2	R3	R4	R5	Else
B1	Ist A numerisch?	N	–	J	–	J	
B2	Ist B numerisch?	–	N	–	J	J	
B3	Ist A größer Null?	–	–	N	–	J	
B4	Ist B größer Null?	–	–	–	N	J	
Berechne C und prüfe, ob C kleiner als 40; $C=A*100/B$		–	–	–	–	N	
A1	Fehler: A ist nicht numerisch	X					
A2	Fehler: B ist nicht numerisch		X				
A3	Fehler: A ist nicht größer Null			X			
A4	Fehler: B ist nicht größer Null				X		
A5	Fehler: C ist nicht größer als 40					X	
A und B sind richtig! Weitere Verarbeitung möglich							X

Abb. 11.3-19:
Beispiel für eine
Mehrtreffer-ET.

Bei Mehrtreffer-Entscheidungstabellen können zwischen den Bedingungen der Regeln vier verschiedene Beziehungstypen bestehen:

Beziehungen
zwischen den
Bedingungen

- 1 Gleichheit (Identität)
- 2 Ausschluss (Exklusion)
- 3 Einschluss (Inklusion)
- 4 Überschneidung (Intersektion)

Sind die Bedingungsanzeiger zweier Regeln gleich, dann sind die Regeln in Abhängigkeit von den Aktionsanzeigern entweder redundant, widersprüchlich oder zusammenfassbar. Folgende Maßnahmen sind möglich:

Gleichheit

- Löschen von redundanten Regeln (gleiche Bedingungs- und Aktionsanzeiger),
- Modifikation der Tabelle bei widersprüchlichen Regeln (gleiche Bedingungs-, aber unterschiedliche Aktionsanzeiger),
- Zusammenfassung von Regeln mit gleichen Bedingungsanzeigern, deren Aktionsteile unterschiedlich, aber widerspruchsfrei sind.

Ein gegenseitiger **Ausschluss** zweier Regeln liegt vor, wenn sie in mindestens einer Bedingungszeile unterschiedliche Bedingungsanzeiger haben. Es kann jeweils nur je eine von beiden Regeln zutreffen.

Ausschluss

Ein **Einschluss** zweier Regeln R1 und R2 liegt vor, wenn mindestens ein Bedingungsanzeiger B1 von R1 auch in R2 enthalten ist und alle übrigen Bedingungen gleich sind. Dies ist z.B. dann der Fall, wenn B1 von R1 einen bestimmten Wert und B1 von R2 den Irrelevanzanzeiger enthält. Die Regel R1 ist dann in der Regel R2 enthalten

Einschluss

R1	R2
J	–
N	N
J	J

III 11 Logik

oder R1 ist kleiner als R2. Trifft die kleinere Regel R1 zu, dann gilt auch die größere Regel R2. Eine eingeschlossene Regel beschreibt häufig einen Sonderfall.

Überschneidung

R1	R2
–	J
N	–
J	J
N	N

Zwei Regeln R1 und R2 überschneiden sich, wenn mindestens ein Bedingungsanzeiger B1 von R1 auch in R2 und ein weiterer Bedingungsanzeiger B2 von R2 auch in R1 enthalten ist und alle übrigen Bedingungsanzeiger gleich sind. Dies ist z.B. dann der Fall, wenn bei der einen Regel B1 und bei der anderen Regel B2 den Irrelevanzanzeiger enthält. Überschneiden sich zwei Regeln, so gibt es mindestens je einen Fall, in dem nur eine der beiden Regeln zutrifft. Gelten beide Regeln, so ist auf sachliche Unstimmigkeit zu prüfen. Außerdem muss sichergestellt werden, dass sich die Aktionen nicht widersprechen.

11.3.8 Zusammenfassung & Bewertung

Entscheidungstabellen erlauben eine tabellarische oder grafische Darstellung (Entscheidungsbäume genannt) von durchzuführenden (durch und verknüpften) Aktionen in Abhängigkeit von Bedingungskonstellationen, wobei die Bedingungen wiederum durch »und« verknüpft werden.

Bedingungen und Aktionen werden durch Regeln verknüpft. Sind alle möglichen Bedingungskombinationen berücksichtigt, dann handelt es sich um eine vollständige Entscheidungstabelle. Tritt eine Bedingungskonstellation zu einer Zeit genau einmal ein, dann handelt es sich um eine Eintreffer-Tabelle, sonst um eine Mehrtreffer-Tabelle. Eine begrenzte Entscheidungstabelle liegt vor, wenn als Bedingungsanzeiger nur J (für Ja), N (für Nein) und »-« (als Irrelevanzanzeiger) und als Aktionsanzeiger nur »X« (für Durchführen) verwendet werden, sonst handelt es sich um eine erweiterte Entscheidungstabelle. Erfordert eine Problembeschreibung eine Kombination von Entscheidungstabellen (möglich sind Sequenz, Verzweigung, Schleife, Schachtelung), dann geschieht dies durch einen Entscheidungstabellen-Verbund. Einen Überblick über Entscheidungstabellen gibt die Abb. 11.3-20.

Bewertung

- + Entscheidungstabellen erlauben eine übersichtliche und kompakte Darstellung von Aktionen, die von mehreren Bedingungen abhängig sind.
- + Durch manuelle und automatisierbare Verfahren können Entscheidungstabellen auf verschiedene Eigenschaften hin überprüft und optimiert werden. Die Gefahr, dass man bei komplexen Entscheidungssituationen Sonderfälle übersieht, wird durch eine Entscheidungstabellendarstellung reduziert.

Abb. 11.3-20:
Überblick über ET.

Erstellen

- 1 Ermittlung der Aktionen
- 2 Ermittlung der Bedingungen
- 3 Eintrag der Bedingungen und Aktionen in die ET
- 4 Eintrag aller Bedingungskombinationen
- 5 Eintrag der Aktionsanzeiger

Konsolidieren

- 1 Gibt es Regeln mit identischen Aktionen?
- 2 Wenn ja, dann paarweise Betrachtung dieser Regeln
- 3 Unterscheiden sich die Bedingungsanzeiger dieser Regeln nur in einer Zeile, dann beide Regeln zu einer Regel zusammenfassen und unterschiedliche Bedingungsanzeiger durch Irrelevanzanzeiger ersetzen.

e/se-Regel

- Alle Regeln, in denen die gleichen Aktionen angekreuzt sind, können zu einer e/se-Regel zusammengefasst werden.
- Eine e/se-Regel legt fest, welche Aktionen durchzuführen sind, wenn keine andere Regel der ET zutrifft.

Anwenden

- 1 Ermittlung der konkreten Bedingungsanzeiger
- 2 Auswertungsvektor über Bedingungsanzeiger schieben
- 3 Bei Gleichheit angekreuzte Aktionen ausführen
- 4 Bei Mehrtreffer-Tabelle alle Regeln prüfen

Arten von ET

formal vollständig: alle 2^n -Bedingungskombinationen vorhanden
inhaltlich vollständig: fachlich nicht mögliche Bedingungen sind nicht vorhanden

ET-Verbund:	Verknüpfung von ET durch Sequenz, Verzweigung, Schleife und Schachtelung; reduziert die Anzahl der Bedingungskombinationen pro ET; möglich, wenn es Abhängigkeiten in der Reihenfolge der Bedingungen und Aktionen gibt.
Erweiterte ET:	Bedingungs- und Aktionszeiger enthalten beliebigen Text
Eintreffer-ET:	Zu einem Zeitpunkt nur eine Regel anwendbar. Hat man eine gültige Regel gefunden, dann ist man fertig.
Mehrtreffer-ET:	Enthalten nicht-disjunkte Bedingungsanzeigerteile. Alle Regeln müssen bei der Anwendung der ET überprüft werden.

- + Werkzeuge unterschiedlicher Mächtigkeit unterstützen die Erstellung, Analyse und Optimierung von Entscheidungstabellen. Werden Entscheidungstabellen in der Implementierungsphase eingesetzt, dann generieren Werkzeuge den fertigen Quellcode in der gewünschten Programmiersprache.
- + Der Vorteil von Entscheidungstabellen in der Definitionsphase liegt darin, dass die Abarbeitungsreihenfolge – im Gegensatz zu Algorithmen – nicht festgelegt wird. Damit bleiben Freiräume erhalten, die bei der Beschreibung mit Algorithmen nicht mehr vorhanden sind.
- Eintreffer-Tabellen sind schwierig zu erstellen. Ihre Überprüfung auf Eindeutigkeit und Vollständigkeit der Bedingungen sowie auf Widersprüchlichkeit der Aktionen und Redundanzfreiheit aller Regeln kann jedoch automatisch erfolgen.

III 11 Logik

- Mehrtreffertabellen sind leichter zu erstellen, da Eindeutigkeit und Vollständigkeit der Bedingungstexte nicht berücksichtigt werden müssen. Die Überprüfung auf Widerspruchsfreiheit ist problematisch. Jedoch kann die Redundanzfreiheit der Regeln automatisch überprüft werden.

Zur Klassifikation

- Entscheidungstabellen werden grafisch (mit textuellen Annotationen) beschrieben.
- Der Grad der Formalität reicht von informal bis formal.
- Sie werden in allen Entwicklungsphasen eingesetzt – vorzugsweise im *Requirements Engineering*.
- Sie werden in allen Anwendungsbereichen verwendet.

11.4 Regeln

Interviewt man einen Auftraggeber oder einen Fachexperten, dann formulieren sie ihre Anforderungen oder ihr Wissen mitunter in Form von Regeln, wobei die Regeln logische Zusammenhänge beschreiben. Softwareexperten können nun versuchen, dieses regelbasierte Wissen in Algorithmen zu überführen. Dies ist aus folgenden Gründen nicht immer möglich oder sinnvoll:

- Eine algorithmische Lösung des Problems ist *nicht* bekannt oder die bekannten Algorithmen sind zu zeitintensiv. Eine Lösung benötigt zu viel Zeit, es dauert zu lange. Es existieren aber Erfahrungen, wie dieses Problem gelöst werden kann. Diese Erfahrungen, dieses Wissen, steckt in den formalisierten Regeln und kann einer maschinellen Verarbeitung zugänglich gemacht werden.
- Große Teile von Berechnungen oder Abläufen unterliegen sehr starken Veränderungen. Die gewünschte Softwarelösung muss daher hochgradig flexibel sein und auf schnell wechselnde Anforderungen reagieren können. In solchen Fällen ist es sinnvoll, Berechnungen oder Abläufe mittels Regeln zu beschreiben. Veränderte Anforderungen erfordern dann nur Änderungen in den Regeln, jedoch nicht in der Software selbst.

Die Entwicklung regelbasierter Software kann aus zwei Blickwinkeln betrachtet werden. Das Problem muss mittels **Regeln** beschrieben werden – **Wissensrepräsentation** genannt. Die beschriebenen Regeln müssen maschinell geprüft und verkettet werden – dies erledigen sogenannte **Inferenzmaschinen** oder **Regel-Interpreter**. Eine Inferenzmaschine, verallgemeinert spricht man auch von einer *Shell*, ist eine problemunabhängige Lösung.

Regeln besitzen eine festgelegte Struktur:

- »Aufbau von Regeln«, S. 405

Voraussetzung für die Verarbeitung von Regeln ist die Auswahl anwendbarer Regeln:

- »Auswahl von Regeln«, S. 407

Ausgewählte Regeln werden von Expertensystemen ausgeführt:

- »Regelbasierte Software«, S. 409

Dazu wird ein effizienter Algorithmus verwendet:

- »Der Rete-Algorithmus«, S. 412

Regeln können vorwärts- und rückwärtsverkettet ausgeführt werden:

- »Verkettung von Regeln«, S. 414

Verschiedene Strategien können eingesetzt werden, um eine Lösung zu finden:

- »Lösungssuche«, S. 417

Regeln und Fakten können bewertet werden, um zu einer Lösung zu gelangen:

- »Bewertete Regeln«, S. 426

Business Rules erlauben es, betriebswirtschaftliche Prozesse zu beschreiben:

- »Geschäftsregeln«, S. 428

Beispiele aus zwei Anwendungsbereichen verdeutlichen den Einsatzbereich von regelbasierten Systemen:

- »Anwendungen«, S. 429

Zusammenfassend wird ein Überblick gegeben:

- »Zusammenfassung«, S. 430

[LäCl08], [ScGr06], [Freg1879], [HSB07], [RuNo03], [GRS03].

Weiterführende
Literatur

11.4.1 Aufbau von Regeln

Eine Regel ist eine Form der Wissensrepräsentation, die im Alltag vielfach gegenwärtig ist.

○ Vorschrift:

»Auf der Fahrbahn dürfen sie nur gehen, wenn die Straße weder einen Gehweg noch einen Seitenstreifen hat.« StVO, §25(1)

Beispiele

○ Naturwissenschaftliches Gesetz:

»Wenn Wasser auf 100° erhitzt wird, dann verdampft es.«

○ Handlungsanweisung z. B. in einer Bedienungsanweisung:

»Wenn das Fenster angezeigt wird, dann klicken Sie auf die Schaltfläche Weiter.«

○ Androhung:

»Wenn Du mir nicht zuhörst, dann gehe ich.«

○ Regel in Form einer Kontrollstruktur in Programmiersprachen:

»if (x>0) then ...«

III 11 Logik

Regel Verallgemeinert hat eine **Regel** die Form: **wenn A dann B**.
Dabei lassen sich A sowie B verschieden interpretieren:

- **wenn** Situation **dann** Aktion.
- **wenn** Bedingung **dann** Ergebnis.
- **wenn** Voraussetzung **dann** Folgerung.

Modus ponens Alle Formen und Anwendungen einer Regel basieren auf der allgemeinen Schlussweise des **Modus ponens**, der besagt, dass B sicher eintritt bzw. als sicher angenommen werden kann, falls die Regel »**wenn A dann B**« als solche gültig ist, und die Voraussetzung A oder die Bedingung A oder die Situation A erfüllt bzw. eingetreten ist. Der Modus ponens ist eine Schlussregel, die bereits seit der Antike bekannt ist:

$$\begin{array}{c} A \rightarrow B \\ A \\ \hline B \end{array}$$

Regel & Prädikatenlogik Wenn aus A B folgt und A wahr ist, dann muss auch B wahr sein. Gottlob Frege (1848–1925) entwickelte in seiner Begriffsschrift von 1879 die Grundlagen der Prädikatenlogik erster Stufe (siehe »Prädikatenlogik«, S. 367) und stellte dabei die Implikation, die die Basis der Regeln bildet, in den Mittelpunkt [Frege 1879].

Implikation Gegenüber der allgemeinen Form einer logischen **Implikation** $A \rightarrow B$, die beliebige prädikatenlogische Ausdrücke für A sowie B zulässt, gelten für die praktische Wissensrepräsentation mittels Regeln einige Besonderheiten:

- Produktionsregel
- Im Folgerungsteil B einer Regel **wenn A dann B** sind nicht nur logische Ausdrücke, sondern auch Aktionen zugelassen. Man spricht dann von **Produktionsregeln**.
 - In den Regeln sind keine Existenzquantifizierungen von Variablen erlaubt.
 - Alle Variablen werden als allquantifiziert angesehen, sodass das Symbol der Quantifizierung nicht angegeben wird.

Beispiel 1a Für alle x gilt, wenn x eine gerade Zahl größer als 2 ist, dann existieren zwei Primzahlen p und q, deren Summe x ergibt: $x = p + q$. In formaler Darstellung als prädikatenlogische Formel lautet der Satz:

$$\forall x \, x > 2 \wedge \text{gerade}(x) \rightarrow \exists p \, \exists q (x = p + q)$$

Eine derartige Implikation kann nur dann sinnvoll in eine Regel umgewandelt werden, wenn ein Algorithmus zur Bestimmung der existenzquantifizierten Variablen (hier p und q) angegeben werden kann.

Bei der formalen Umwandlung eines prädikatenlogischen Ausdrucks in eine Menge von Klauseln (Fakten und Regeln) wird eine **Existenzquantifizierung** durch eine so genannte **Skolemisierung** beseitigt. Praktisch kann man sich darunter eine Funktion vorstellen, die einen Wert für die existenzquantifizierte Variable berechnet.

In diesem Beispiel sind das die beiden einstelligen Funktionen $fp/1$ sowie $fq/1$, die dann die Primzahlen der Belegung berechnen müssen:

Beispiel 1b:
allquantifizierte Variablen

wenn $x > 2 \wedge \text{gerade}(x)$ **dann** $x = fp(x) + fq(x)$.

Interpretation: Für alle x gilt, wenn ..., dann $x = fp(x) + fq(x)$.

Im Sinne der Logik sind Regeln **Klauseln**, die nur genau ein positives Literal besitzen.

Regeln sind Klauseln

Regeln sind ein leicht verständliches Mittel für die formale Darstellung von Wissen, da Regeln aus dem Alltag geläufig sind. Dennoch dürfen die Schwierigkeiten einer formalen Wissensrepräsentation *nicht* unterschätzt werden.

wenn container A steht auf platz X
 und container A steht oben **und** platz Y ist frei
 dann Setze container A von platz X nach platz Y.

Beispiele für Regeln

wenn istKunde(X) **dann** sendeRechnung(X).
wenn nicht istKunde(X) **dann** Nachnahme(X).
wenn istGuterKunde(X) **dann** erhaeltRabatt(X).
wenn umsatz(X) > 1.000 **dann** istGuterKunde(X).

11.4.2 Auswahl von Regeln

Die Verarbeitung von Regeln beruht auf der **Auswahl einer Regel**, die ausgeführt wird. Diese Auswahl aus einer Menge von anwendbaren Regeln ist eines der grundlegenden Probleme bei der Ausführung regelbasierter Software.

Aufgrund der Faktenlage wird überprüft, ob eine Regel anwendbar ist, das heißt der Bedingungsteil dieser Regel erfüllt ist. Dabei kann sich herausstellen, dass mehrere Regeln anwendbar sind.

Regel anwendbar?

Die Menge der anwendbaren Regeln wird als **Konfliktmenge** bezeichnet. Gemäß einer bestimmten Ordnung ist aus dieser Menge eine Regel für die Ausführung auszuwählen. Die Schwierigkeit liegt in der Bestimmung einer geeigneten Ordnung. Je nach Anwendung werden folgende Ansätze in Betracht gezogen:

- Es wird die erste anwendbare Regel gemäß der Reihenfolge, in der die Regeln formuliert wurden, ausgewählt (textuelle Ordnung).

1. Regel

III 11 Logik

Speziell vor allgemein ■ Es werden spezielle Regeln vor allgemeinen Regeln ausgewählt. Eine Regel wird als spezieller als eine andere angesehen, falls sie mehr mittels *und* verknüpfte Bestandteile im Bedingungsteil aufweist.

Beispiel 1. **wenn** istKunde(X) **dann** sendeRechnung(X).
 2. **wenn** istKunde(X) **und** umsatz(X) > 1.000 **dann** erhaeltRabatt(X).

Regel 2 ist spezieller als Regel 1.

■ Auch für den umgekehrten Ansatz, allgemeinere Regeln vorzuziehen, gibt es denkbare Situationen.

Metaregel ■ Es werden zuerst **Metaregeln** ausgeführt, die bestimmen, welche weiteren Regeln oder Regelmengen für die Bearbeitung in Betracht gezogen werden.

Neu vor alt ■ Um insbesondere bei Planungsaufgaben Wiederholungen zu vermeiden, die zu nicht optimalen Lösungen führen, werden zuerst die bis dato noch nicht benutzten Regeln bevorzugt.

Alt vor neu ■ Auch hier kann der entgegen gesetzte Ansatz mehr Erfolg zeigen: Man greife zuerst auf die zuletzt benutzten Regeln zurück, die einen Fortschritt zur Problemlösung gebracht haben.

Aktuellste Regel ■ Es wird die aktuellste Regel ausgewählt. Dies geschieht anhand einer Zeitmarkierung, die jeder Fakt erhält. Die Zeitmarkierung registriert den Zeitpunkt der letzten Änderung des Wertes.

Bewertung ■ Regeln werden mit Bewertungen versehen, die dann bei der Regelauswahl berücksichtigt werden.

Zufall ■ Eine zufällige Auswahl einer anwendbaren Regel ist denkbar.

Beispiel 1. **wenn** istKunde(X) **dann** sendeRechnung(X).
 2. **wenn** nicht istKunde(X) **dann** Nachnahme(X).
 3. **wenn** istKunde(X) **und** umsatz(X) > 1.000 **dann** erhaeltRabatt(X).

Je nach Datenlage können anwendbar sein:

- Nur Regel 1,
- nur Regel 2,
- die Regeln 1 und 3.

Eine Regel, die bereits eingesetzt wurde, darf hierbei in keinem Fall erneut ausgeführt werden. Die Reihenfolge der beiden Regeln 1 und 3 beeinflusst das Ergebnis dann nicht.

Beispiel Container-Umschlag **wenn** container X auf platz P **dann** setze container X auf platz Q.

Abb. 11.4-1: Start- und Zielzustand des Container-Umschlag-Problems.



Diese Regel hat für die Startsituation vier konkrete anwendbare Regeln mit instanziierten Variablen (Abb. 11.4-1):

wenn A auf platz 1 dann setze A auf platz 2.
 wenn A auf platz 1 dann setze A auf platz 3.
 wenn C auf platz 2 dann setze C auf platz 1.
 wenn C auf platz 2 dann setze C auf platz 3.

11.4.3 Regelbasierte Software

In einer **regelbasierten Software** existiert eine Trennung zwischen der Faktenbasis und der Regelbasis sowie der Verarbeitung.

Trennung Wissen
von Verarbeitung

Ein Fakt gilt ohne Wenn und Aber, z. B.

Fakt

umsatz(meier, 5.000), gerade(6).

Diese Fakten, meist im herkömmlichen Sinne Daten, werden in einer Datenbank, der **Faktenbasis** verwaltet. Im Sinne der Logik sind Fakten Klauseln, die nur aus einem positiven **Literal** bestehen.

Die Menge der Regeln wird in einer **Regelbasis** verwaltet. Getrennt von der Wissensbasis – die aus der Fakten- und Regelbasis besteht –, übernimmt eine **Inferenzmaschine** die Verarbeitung des Wissens. Damit wird eine Trennung der Logik von der Verarbeitung erzielt. Das WAS einer Lösung wird in der Wissensbasis dargestellt und verwaltet. Die Verarbeitung – das WIE – übernimmt die Inferenzmaschine und ist weitgehend problemunabhängig. Eine regelbasierte Software ist sehr flexibel an geänderte Aufgabenstellung anpassbar. Änderungen und Erweiterungen finden nur in der Wissensbasis statt.

WAS
anstatt
WIE

Die Struktur eines **Expertensystems** zeigt deutlich die Trennung der Wissensbasis von der Steuerung und der Benutzungsoberfläche (Abb. 11.4-2).

Experten-
system

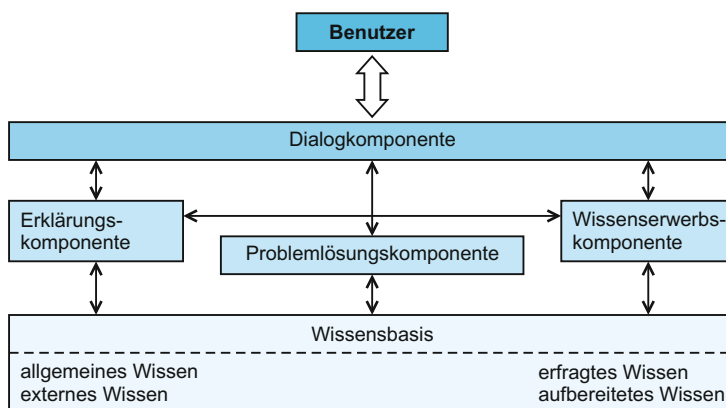


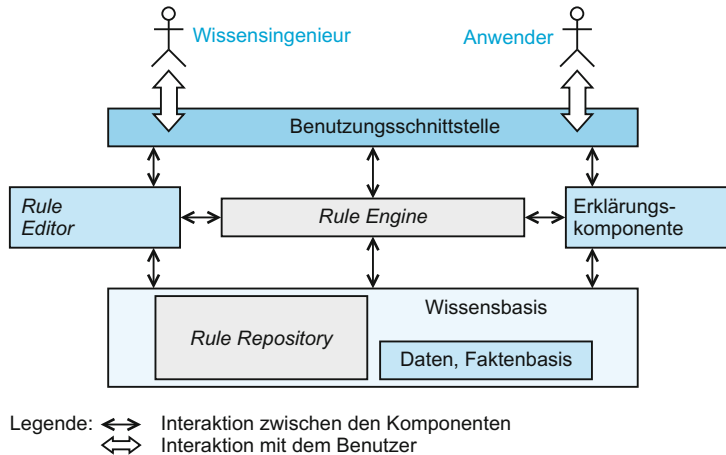
Abb. 11.4-2:
Struktur eines
Expertensystems.

Legende: \leftrightarrow Interaktion zwischen den Komponenten
 \rightleftarrows Interaktion mit dem Benutzer

III 11 Logik

BRMS **BRMS** (*Business Rules Management Systems*) sind die aktuelle Form einer Expertensystem-Shell. Sie besitzen eine analoge Struktur, indem sie die Fakten und Regeln in einer Datenbank verwalten und eine davon getrennte Steuerungskomponente, die Inferenzmaschine oder *Rule Engine*, nutzen. Erweiterungen gegenüber den Expertensystemen früherer Jahre sind ein grafischer Regeleditor sowie die Möglichkeit Regeln in natürlicher Sprache zu formulieren. Darüber hinaus sind BRMS heutzutage keine proprietären Lösungen, sondern lassen sich in betriebliche IT-Strukturen einbetten (Abb. 11.4-3).

Abb. 11.4-3:
Struktur eines
Business Rules
Management
Systems.



Die Schritte zur Anpassung der Software an neue Kundenwünsche oder Änderungen, die anderweitig erzwungen werden, z. B. durch Gesetze, zeigt die Abb. 11.4-3. Im Idealfall ist der Benutzer selbst in der Lage, neue Regeln hinzuzufügen bzw. bestehende Regeln abzuändern. Obwohl viele Systeme diesen hohen Anspruch an die Benutzungsfreundlichkeit formulieren, wird in der Realität stets ein Informatiker, ein Wirtschaftsinformatiker oder ein Wissensingenieur diese Aufgaben übernehmen.

Prolog Die logische Programmiersprache **Prolog** kann als eine Expertensystem-Shell angesehen werden. In einer Datenbasis werden Fakten und Regeln gehalten, ein Inferenzmechanismus ist in der Lage, Anfragen zu beweisen. Nach wie vor eignet sich Prolog sehr gut für die regelbasierte Beschreibung von Problemen. Der Vorteil: Eine regelbasierte Beschreibung eines Problems ist zumeist zugleich eine Beschreibung der Lösung des Problems. Es ist in erster Linie eine psychologische Barriere, die dem breiten Einsatz von Prolog als Werkzeug regelbasierter Systeme entgegensteht. Die rein rekursive Programmierung erfordert eine entsprechende Denkweise der Softwareentwickler.

So konnten sich Prolog-basierte Systeme nicht allgemein durchsetzen, leistungsfähige Anwendungen sind dennoch bekannt und werden weiter entwickelt, siehe z. B. Logic Programming Associates (<http://www.lpa.co.uk/>).

Für den Anwender einer regelbasierten Lösung ergeben sich folgende Vorteile:

- + Neue Ideen der Geschäftslogik, z. B. neue Preis- oder Tarifmodelle, können schneller realisiert werden, da die Implementation dieser neuen Ideen in die entsprechende Anwendungssoftware schneller erfolgen kann. Geschäftslogik
- + Es ist *keine* Veränderung an der Software an sich erforderlich. Es wird nur die Menge der Regeln angepasst. Das verkürzt den Zyklus der Anpassung der Software an die neuen Wünsche beziehungsweise Forderungen erheblich. Ein Programmiereingriff verbunden mit langen Testzeiten auf dem Weg zu einem neuen Release ist *nicht* erforderlich.
- + Eine Veränderung der Menge der Regeln erfordert zwar ebenso einen Test, dies beschränkt sich jedoch auf die inhaltlichen Wirkungen der Regeln. Die Ablaufsteuerung ist bei einem regelbasierten System nicht von den Änderungen der Regeln betroffen (Abb. 11.4-4). Schnelle Änderung

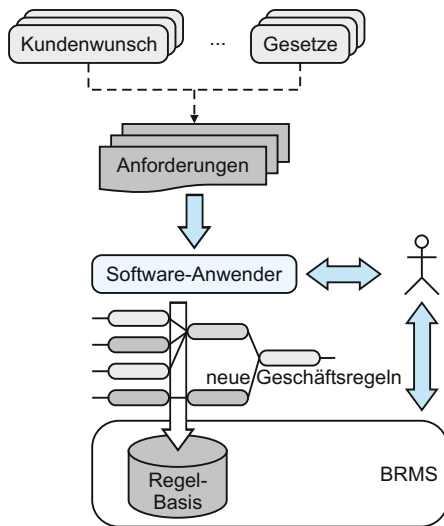


Abb. 11.4-4:
Ablauf bei der
Aktualisierung
regelbasierter
Software.

- + Ständig neue Anforderungen ergeben sich aus Kundenwünschen oder gesetzlichen Regelungen (gestrichelte Linien, Abb. 11.4-4). Nicht mehr der Softwareentwickler, sondern der Anwender wandelt die Anforderungen in Geschäftsregeln und fügt diese der Regelbasis hinzu. Der Benutzer interagiert (blaue Blockpfeile) mit dem BRMS und dem Anwender.

III 11 Logik

Trennung WAS – WIE Die Trennung des WAS von der Steuerung, dem WIE, bringt diesen entscheidenden Gewinn an Zeit und damit auch eine Verringerung der Kosten.

Regel = Dokumentation + Eine regelbasierte Lösung führt zu einer besseren Dokumentation der Geschäftslogik. Die Regeln selbst werden für die Dokumentation genutzt. Damit können die Differenzen zwischen tatsächlichen und dokumentierten Abläufen verringert oder gar beseitigt werden. Dies ist von besonderer Bedeutung für Abläufe, für die gesetzliche Regelungen greifen, und deren Einhaltung nachgewiesen werden muss. Regelbasierte Lösungen können damit zur Qualitätssicherung beitragen.

+ Als weiteren Vorteil kann der Prozess der Entwicklung einer regelbasierten Lösung angesehen werden. Die Herausarbeitung der Regeln erfordert einen intensiven Klärungsprozess, der zu klaren Strukturen, Abläufen oder einer klaren Geschäftslogik führen kann.

11.4.4 Der Rete-Algorithmus

Das Problem Um die **Konfliktmenge**, d.h. die Menge der anwendbaren Regeln, zu bestimmen, sind generell alle Regeln zu untersuchen und deren Bedingungssteile zu prüfen. Das erfordert einen hohen Rechenaufwand, insbesondere wenn in realistischen Anwendungen Regelbasen mit einigen Hunderttausenden oder Millionen von Regeln zu bearbeiten sind. Eine effiziente Bestimmung der anwendbaren Regeln ist für einen praktischen Einsatz regelbasierter Systeme notwendig.

Rete Unter dem Namen Rete (lateinisch für Netz) ist ein effizienter Algorithmus bekannt, der in den meisten regelbasierten Systemen, so auch in BRMS, eingesetzt wird. Der **Rete-Algorithmus** wurde von Charles Forgy Ende der 70er Jahre im Rahmen der Arbeiten an der Expertensystem-Shell OPS5 entwickelt.

Ziel ist es, die Überprüfungen der Regeln auf Anwendbarkeit soweit zu reduzieren, dass lediglich für neue beziehungsweise geänderte Fakten neue Berechnungen erforderlich sind. Dazu werden die Ergebnisse der Überprüfungen der Bedingungen der Regeln gespeichert und nur für diejenigen Regeln neu berechnet, deren Bedingungen auf geänderten Fakten basieren.

Idee des Algorithmus Die Grundidee des Rete-Algorithmus wird anhand eines Beispiels erläutert.

Beispiel **wenn** wochenende(heute) **und** sonnenschein **und** temperatur > 23
 dann fahreBaden.
wenn wochenende(heute) **und** sonnenschein **und** temperatur < -5
 dann geheEislaufen.

Die Bedingungen und deren Ergebnisse werden nun in einem Graphen verwaltet. Dazu wird für jede Regel ein Graph aufgebaut, wobei dann gleiche Bedingungen miteinander verschmolzen werden (Abb. 11.4-5).

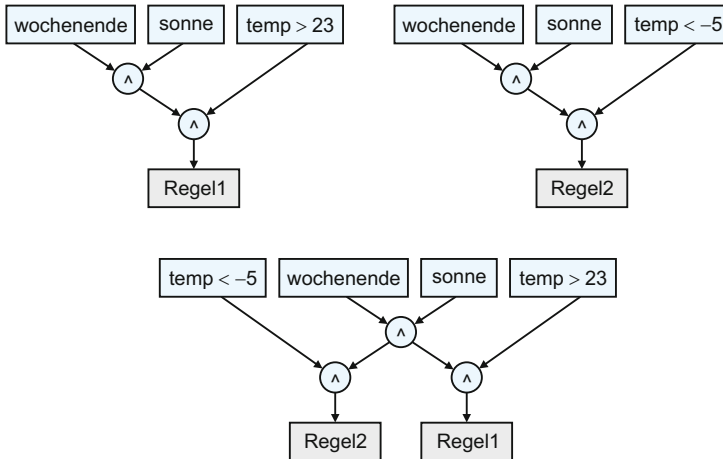


Abb. 11.4-5: RETE-Netz der Regeln.

Es werden zwei verschiedene Arten von Knoten unterschieden: die α -Knoten repräsentieren jeweils eine Bedingung während die β -Knoten die Verknüpfung übernehmen. Für jeden Knoten werden die Ergebnisse gespeichert, sodass bei einer Änderung nur die von dieser Änderung betroffenen Knoten erneut zu berechnen sind.

Nicht mehr als nötig

Für das erste Werte-Tupel, zum Beispiel (sonntag, sonnenschein, 25) werden alle Bedingungen geprüft und die Ergebnisse gespeichert, eventuell werden entsprechende Variablenbelegungen mit verwaltet (Abb. 11.4-6).

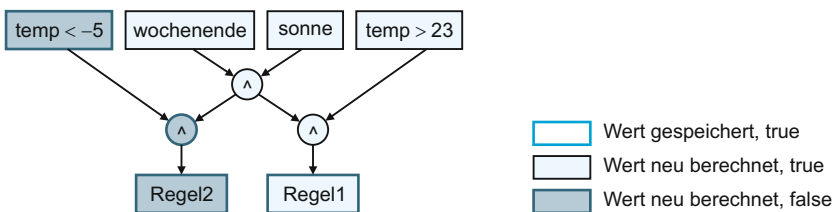


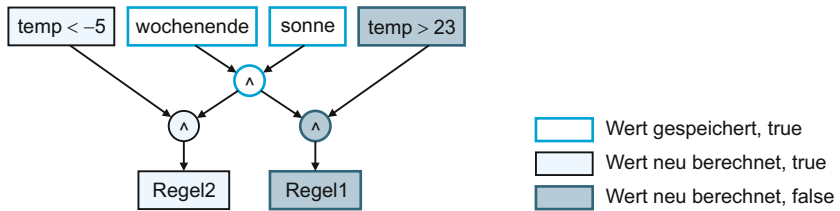
Abb. 11.4-6: Erster Zustand des Netzes.

Für ein weiteres Tupel, nun (sonntag, sonnenschein, -6) werden nun nur die Knoten betrachtet, die von einem geänderten Wert abhängen: [temp < -5] sowie [temp > 23] (Abb. 11.4-7).

Der Rete-Algorithmus wurde von Forgy weiter entwickelt und von ihm unter dem Namen ReteII kommerziell als Teil regelbasierter Software vermarktet.

III 11 Logik

Abb. 11.4-7:
Zweiter Zustand
des Netzes.



11.4.5 Verkettung von Regeln

Eine Lösung erhält man meist nicht allein aus der einmaligen Auswahl und Abarbeitung einer einzigen Regel. Die Ausführung einer Regel bewirkt im Allgemeinen, dass weitere Regeln anwendbar werden. Man spricht hierbei von der Verkettung von Regeln. Es gibt zwei unterschiedliche Strategien:

- Vorwärtsverkettung** ■ Zum einen kann man von den Fakten ausgehen, anwendbare Regeln ermitteln, diese ausführen und dieses dann zyklisch wiederholen. Man spricht von einer Vorwärtsverkettung.
- Rückwärtsverkettung** ■ Im Umkehrschluss bezeichnet man als Rückwärtsverkettung, wenn von einem Ergebnis, einer Hypothese, ausgegangen wird und man diese Hypothese unter Nutzung von Regeln auf die Fakten zurückführt.

Eine Regel-Verkettung tritt nur dann auf, wenn nach der Ausführung einer Regel, eine weitere Regel anwendbar wird. Somit scheiden **Produktionsregeln** für eine Verkettung aus, da Produktionsregeln Aktionen ausführen. Diese Aktionen verändern nicht die Daten- oder Faktenlage und führen daher nicht dazu, dass weitere Regeln anwendbar werden. **Schlussfolgerungsregeln** hingegen führen zu neuen Fakten und ermöglichen eine Verkettung von Regeln.

Planen Die Lösung von Planungsaufgaben, wie bei dem Beispiel Container-Umschlag (siehe »Auswahl von Regeln«, S. 407), erfordert eine Verkettung von Regeln, ja die Kette von Regeln, die von einem Anfangs- zu einem Endzustand führt, ist in einem derartigen Fall die tatsächlich gesuchte Lösung. Man beachte, dass die Container-Regeln im obigen Beispiel keine Produktionsregeln im engeren Sinne sind: »setze C auf 1« liest sich zwar wie eine Aktion, ist de facto aber eine Schlussfolgerung, die die Faktenbasis verändert. Nur so können andere Regeln anwendbar werden.

Beispiel 1a R1: wenn istKunde(X) dann sendeRechnung(X).
 R2: wenn nicht istKunde(X) dann Nachnahme(X).
 R3: wenn istGuterKunde(X) dann erhaeltRabatt(X).
 R4: wenn umsatz(X) > 1.000 dann istGuterKunde(X).

Die Faktenbasis enthalte die Angabe, dass Schmidt ein Kunde ist und aus der Datenbank kann der Umsatz von Schmidt mittels einer Abfrage als 1.600 bestimmt werden: istKunde(schmidt), umsatz(schmidt) = 1.600.

Vorwärtsverkettung

Bei einer **Vorwärtsverkettung** wird ausgehend von der Daten- bzw. Faktenlage überprüft, welche der Regeln anwendbar ist. Es wird die Konfliktmenge zusammengestellt und gemäß einer Ordnung werden die anwendbaren Regeln ausgeführt. Dies geschieht solange, bis ein gewünschtes Ergebnis ermittelt wurde oder keine Regel mehr anwendbar ist: Ein Match-Select-Act-Zyklus wird durchlaufen. Die Abb. 11.4-8 zeigt das Vorgehen anhand eines UML-Zustandsdiagramms.

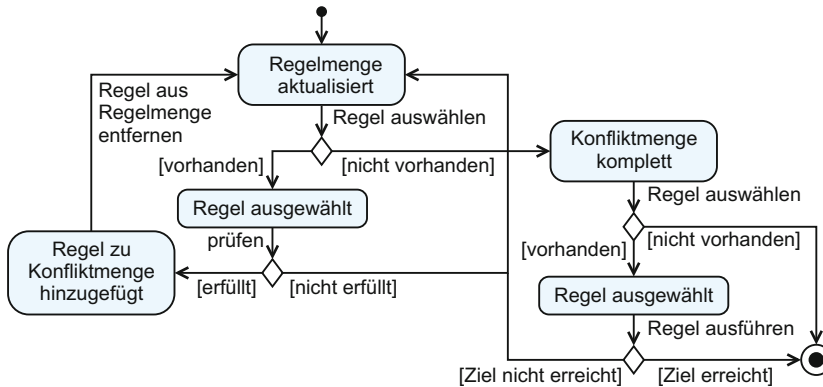


Abb. 11.4-8:
Vorgehensweise
bei der Vorwärts-
verkettung.

Die Konfliktmenge besteht im ersten Durchlauf aus den Regeln R1 und R4: {R1,R4}. Nach der Ausführung von R1 wird `sendeRechnung(schmidt)` in die Datenbasis hinzugefügt und dann erneut die Konfliktmenge bestimmt, die sich nicht geändert hat. Werden Regeln ausgeschlossen, die gerade gefeuert wurden, dann besteht die Konfliktmenge aktuell nur aus der Regel R4: {R4}. Es wird R4 ausgeführt und damit `istGuterKunde(schmidt)` zur Faktenbasis hinzugefügt. Die Konfliktmenge ändert sich nun in {R3}, die nun aktiviert wird: Der Kunde Schmidt erhält Rabatt.

Beispiel 1b
Konflikt-
menge

Zuerst sind die Schlussregeln zu bearbeiten, da von diesen eine Veränderung der Faktenbasis zu erwarten ist. Danach können Produktionsregeln ausgeführt werden, deren Aktionen dann keinen Einfluss auf die Faktenbasis und somit auf die Bedingungsteile von Regeln haben.

Vorwärtsverkettung wird häufig bei der Lösung von Planungsaufgaben eingesetzt. Hierbei besteht eine gesuchte Lösung nicht aus einem bestimmten Endzustand, dieser kann sogar vorher bekannt sein, sondern in einer Folge von Regeln, die von dem Start- zu einem Endzustand führen.

Viele Business-Rules-Management-Systeme bieten eine vorwärtsverkettende Regel-Maschine an, darunter auch die frei verfügbare Software JBoss Drools.

BRMS

III 11 Logik

Rückwärtsverkettung

Bei einer **Rückwärtsverkettung** werden die Regeln rückwärts, somit vom dann-Teil in Richtung wenn-Teil, von der Schlussfolgerung in Richtung der Voraussetzung angewendet.

Beispiel 1c R1: **wenn** istKunde(X) **dann** sendeRechnung(X).
Diese Regel kann ebenso interpretiert werden als:
R1: sendeRechnung(X) **falls** istKunde(X).

Beweis einer Hypothese Es wird mit einer Hypothese, d.h. einer Behauptung, begonnen und versucht, diese zu beweisen. »Beweisen« bedeutet, die Behauptung mittels der Regeln auf die vorhandenen Fakten zurückzuführen.

Die Hypothese ist gültig, falls diese ein Fakt aus der Faktenbasis ist oder es eine Regel gibt, in der die Hypothese als Schlussfolgerung auftritt und alle Bedingungen dieser Regel erfüllt, d.h. gültig sind.

Algorithmus

```
boolean istGueutig(Hypothese x) {
    return
        istInFaktenbasis(x) ||
        ( existiert Regel: wenn bedingungen dann x
          && für alle b aus bedingungen gilt: istGueutig(b) )
} //istGueutig
```

Die Verkettung der Regeln wird durch eine andere Definition von istGueutig() unter Nutzung einer weiteren Funktion rueckwaertsverkettung() deutlich:

```
boolean istGueutig(Hypothese x) {
    return istInFaktenbasis(x) || rueckwaertsverkettung(x);
} // istGueutig

boolean rueckwaertsverkettung(Hypothese x) {
    return
        existiert Regel: wenn bedingungen dann x
        && für alle b aus bedingungen gilt: istGueutig(b);
} // rueckwaertsverkettung
```

Beispiel 1d R1: **wenn** istKunde(X) **dann** sendeRechnung(X).
R2: **wenn nicht** istKunde(X) **dann** Nachnahme(X).
R3: **wenn** istGuterKunde(X) **dann** erhaeltRabatt(X).
R4: **wenn** umsatz(X) > 1000 **dann** istGuterKunde(X).

Die Faktenbasis enthalte die Angabe, dass Schmidt ein Kunde ist und aus der Datenbank kann der Umsatz von Schmidt mittels einer Abfrage als 1.600 bestimmt werden:

```
istKunde(schmidt),
umsatz(schmidt) = 1.600.
```

Wird die Hypothese erhaeltRabatt(schmidt) untersucht, so ist dieser Fakt *nicht* in der Faktenbasis. Es existiert jedoch eine Regel (R3), die durch geeignete Instanziierung der Variablen anwendbar wird: **wenn** istGuterKunde(schmidt) **dann** erhaeltRabatt(schmidt).

Somit ist nun die Bedingung von R2 zu prüfen: *istGuterKunde(schmidt)*. Dies führt unter Nutzung von R4 zur Prüfung des Umsatzes von *schmidt*. Da dieser die Bedingung von R4 erfüllt, ist die Hypothese *erhaeltRabatt(schmidt)* gültig. Die Verkettung der Regeln R3 und R4 führt unter Nutzung der Fakten zum gewünschten Ergebnis.

Die Rückwärtsverkettung ist zielgerichtet, denn sie prüft genau die zum Beweisen der Hypothesen notwendigen Regeln beziehungsweise Fakten. Eine Aufsammlung anwendbarer Regeln und eine Auswahl einer Regel aus der Konfliktmenge erfolgt bei der Rückwärtsverkettung üblicherweise nicht. Stattdessen wird die erste anwendbare Regel eingesetzt, bei einem Misserfolg wird mittels **Backtracking** nach einer weiteren anwendbaren Regel gesucht.

Zielgerichtet

Die logische Programmiersprache Prolog setzt eine Rückwärtsverkettung zur Lösungsfindung ein.

Prolog

11.4.6 Lösungssuche

Die Suche nach einer Lösung wird zuerst anhand der Suche nach dem Weg von einem Ausgangspunkt zu einem Ziel erläutert und dann anschließend verallgemeinert.

Welchen Weg?

Will man von einem Ausgangspunkt zu einem Ziel gelangen (Abb. 11.4-9) und hat keine weitere Information zur Verfügung, so steht man an jeder Kreuzung vor derselben Frage: Welcher Weg ist der richtige? Geht man nun nach dem Schema vor, an jeder Kreuzung zuerst den Weg nach links auszuprobieren, dann den Weg geradeaus und dann nach rechts, so verfolgt man eine **Tiefe-zuerst-Strategie**. Eine Entscheidung ist dabei nicht erfolgreich, wenn man keine weitere Kreuzung erreicht oder man an eine Kreuzung gelangt, an der man schon vorher gewesen ist. Dann gehe man zurück zur letzten Kreuzung und probiere den nächsten Weg (**Backtracking**).

Tiefe-zuerst

Es ist schnell einzusehen, dass man sehr lange benötigt, um einen Weg vom Startpunkt, der zuerst zur Kreuzung b2 führt bis zum Ziel (zwischen d4 und e4) zu finden. Ganz sicher ist dieser Weg *nicht* der kürzeste.

Ist man in einer Gruppe unterwegs und sind genügend viele Mitstreiter dabei, so kann man versuchen, den Weg »parallel« zu suchen. An jeder Kreuzung gehen gleichzeitig jeweils einige in jede Richtung. Hat jemand das Ziel erreicht, informiert er alle anderen über seinen Weg und die Suche ist beendet. Dieser Ansatz wird als **Breite-zuerst-Strategie** bezeichnet.

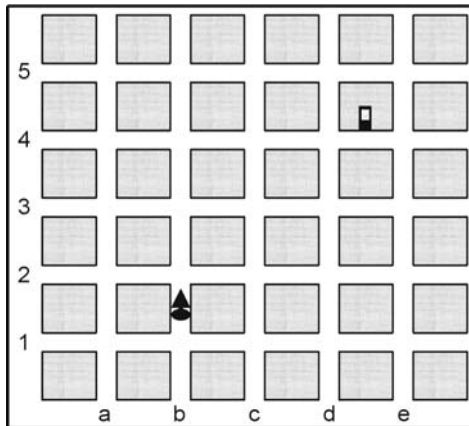
Breite-zuerst

Allemaal besser als die beiden bisher vorgestellten Suchverfahren ist es, wenn Information über die Richtung der Suche zur Verfügung steht. Sind Wegweiser vorhanden, liegt kein Suchproblem mehr vor: Die Lösung, der Weg vom Start zum Ziel, ist in Form der Wegwei-

Heuristische Suche

III 11 Logik

Abb. 11.4-9: Suche eines Weges.



ser angegeben. Angenommen es gibt zwar keine Wegweiser, dafür aber an jeder Kreuzung eine Information, wie weit diese vom Zielort entfernt ist. Diese dann informierte oder heuristische Suche führt schneller zum Ziel.

Algorithmus

Der Algorithmus für die Suche verwaltet die aufgebauten Wege, d. h. Pfade, vom Startzustand z_0 bis zum Ziel. Am Anfang besteht die Folge der Suchpfade nur aus einem Pfad, der wiederum nur eine Folge aus einem Zustand, dem Startzustand, ist. Das Ziel wurde erreicht, falls am Ende des gewählten Pfades p der Zustand z_m der Zielzustand ist. Falls das Ziel noch nicht erreicht wurde, werden alle Nachfolger s_i von z_m (Zustände, die mittels Anwendung einer Regel aus z_m entstehen) bestimmt. Der untersuchte Pfad p wird gestrichen und es werden alle Pfade ps_i in die zu untersuchende Menge aufgenommen, die durch anhängen eines Nachfolgezustands s_i an p entstehen. Der hier angegebene Algorithmus stellt einen Rahmen dar, aus dem sich in einfacher Weise die Algorithmen für die konkreten Suchverfahren ableiten lassen (Abb. 11.4-10).

Anhand der Antworten auf die Frage, wie der Suchbaum aufgebaut wird, unterscheiden sich die Vorgehensweisen: Tiefensuche (Tiefenzuerst-Strategie), Breitensuche (Breite-zuerst-Strategie) und heuristische Suche.

Im Algorithmus wird dies anhand der Auswahl des Pfades p aus der Folge der Suchpfade entschieden. Wählt man stets den ersten Pfad aus der Folge aus, dann verlagert sich das Problem auf das Einfügen der neu generierten Suchpfade $p-s_i$ in die Folge der Suchpfade: an den Anfang, an das Ende, eine andere Reihenfolge?

i Die verschiedenen Strategien werden nun näher betrachtet:

- »Der Suchbaum«, S. 419
- »Tiefe-zuerst-Suche«, S. 420
- »Breite-zuerst-Suche«, S. 422
- »Heuristische Suche«, S. 425

Suche

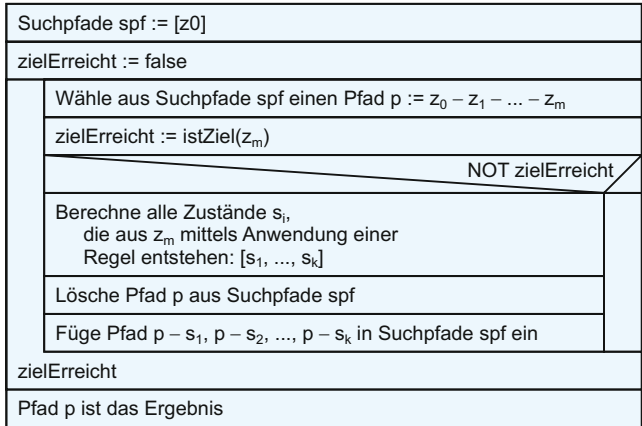


Abb. 11.4-10:
Struktogramm für
den
Suchalgorithmus.

11.4.6.1 Der Suchbaum

Betrachtet man die Menge der Fakten der Faktenbasis zu einem bestimmten Zeitpunkt als einen Zustand, so kann die Anwendung einer Regel diesen Zustand verändern und einen neuen Zustand erzeugen.

Die Zustände können grafisch als Knoten eines Graphen visualisiert werden. Eine Regelanwendung stellt dann eine Kante von einem Zustand zum Folgezustand dar.

R1: **wenn** istKunde(X) **dann** sendeRechnung(X).
R2: **wenn** nicht istKunde(X) **dann** Nachnahme(X).
R3: **wenn** istGuterKunde(X) **dann** erhaeltRabatt(X).
R4: **wenn** umsatz(X) > 1000 **dann** istGuterKunde(X).

Beispiel

Die Faktenbasis enthalte die Angabe, dass Schmidt ein Kunde ist und aus der Datenbank kann der Umsatz von Schmidt mittels einer Abfrage als 1600 bestimmt werden:

$z_0 = \{\text{istKunde}(\text{schmidt}), \text{umsatz}(\text{schmidt}) = 1600\}$. Grafisch gesehen entsteht ein Baum, dessen Wurzel der Startzustand ist (Abb. 11.4-11).

Der komplette Baum, der so genannte **Suchbaum**, kann nur für kleine Probleme vollständig dargestellt werden. Für das Kunden-Beispiel ist dies möglich, wie die Abb. 11.4-11 zeigt. Ein Zustand ist jeweils durch die Menge von Fakten charakterisiert:

$z_0 = \{\text{istKunde}(\text{schmidt}), \text{umsatz}(\text{schmidt}) = 1600\}$
 $z_1 = \{\text{istKunde}(\text{schmidt}), \text{umsatz}(\text{schmidt}) = 1600,$
 $\quad \text{sendeRechnung}(\text{schmidt})\}$
 $z_2 = \{\text{istKunde}(\text{schmidt}), \text{umsatz}(\text{schmidt}) = 1600,$
 $\quad \text{istGuterKunde}(\text{schmidt})\}$
 $z_3 = \{\text{istKunde}(\text{schmidt}), \text{umsatz}(\text{schmidt}) = 1600,$
 $\quad \text{sendeRechnung}(\text{schmidt}), \text{istGuterKunde}(\text{schmidt})\}$

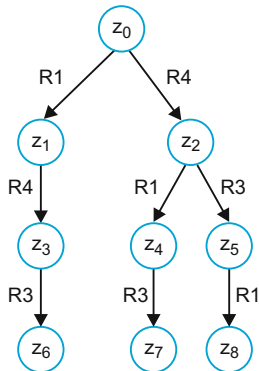
III 11 Logik

```

z4 = { istKunde(schmidt), umsatz(schmidt) = 1600,
       istGuterKunde(schmidt), sendeRechnung(schmidt) }
z5 = { istKunde(schmidt), umsatz(schmidt) = 1600,
       istGuterKunde(schmidt), erhaeltRabatt(schmidt) }
z6 = z7 = z8 = { istKunde(schmidt), umsatz(schmidt) = 1600,
                  sendeRechnung(schmidt),
                  istGuterKunde(schmidt), erhaeltRabatt(schmidt) }

```

Abb. 11.4-11: Ein vollständiger Suchbaum.



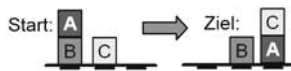
Ein Suchbaum existiert für jedes Suchproblem, nur wird dieser Suchbaum so gut wie niemals vollständig aufgebaut. Die entscheidende Frage lautet: Wie wird der Suchbaum Schritt für Schritt erzeugt, damit einerseits so wenig wie möglich Knoten unnötig erzeugt werden und andererseits ein Weg zum Ziel, besser noch der kürzeste Weg zum Ziel, schnell gefunden wird. Wie verläuft der kürzeste Pfad vom Start zu einem Zielknoten?

Beispiel:
Container-
Umschlag

Es seien die Container A, B und C – wie in der Abb. 11.4-12 dargestellt – umzustapeln. Dazu wird eine Muster-Regel eingesetzt, aus der sich durch Instanziierung der Variablen mehrere konkrete Regeln erzeugen lassen:

wenn container X auf platz P dann setze container X auf platz Q.

Abb. 11.4-12: Start- und Zielzustand des Container-Umschlag-Problems.



11.4.6.2 Tiefe-zuerst-Suche

Nachdem eine Regel auf den Startzustand angewendet wurde und somit ein neuer Zustand erzeugt wurde, stellt sich die Frage, welcher Zustand wird nun weiter auf die Anwendbarkeit von Regeln untersucht. Wird eine weitere Regel auf den Startzustand oder wird eine Regel auf den neu erzeugten Zustand angewendet?

Der Algorithmus für die Tiefensuche unterscheidet sich von dem allgemeinen Suchverfahren nur in der Art der Sortierung der Pfade in der Folge der Suchpfade (Abb. 11.4-13).

Sortierung der
Suchpfade

Bei der Tiefensuche werden die neu generierten Pfade $p-s_i$, die den Pfad p jeweils um einen Zustand s_i verlängern, vor die anderen Pfade p_2, \dots, p_n eingeordnet. Das führt dazu, dass im nächsten Schritt zuerst die neu entstandenen Pfade weiter bearbeitet werden. Es sind die längeren Pfade, die somit zuerst verlängert werden.

Tiefensuche

Suchpfade $\text{spf} := [z_0]$
zielErreicht := false
Sei Suchpfade $\text{spf} = [p_1, p_2, \dots, p_n]$ sowie $p = p_1 = z_0 - z_1 - \dots - z_m$
zielErreicht := istZiel(z_m)
NOT zielErreicht
Berechne alle Zustände s_i , die aus z_m mittels Anwendung einer Regel entstehen: $[s_1, \dots, s_k]$
Lösche Pfad p aus Suchpfade spf
Füge neue Pfade $p - s_1, p - s_2, \dots, p - s_k$ an den Anfang von Suchpfade spf hinzu: $\text{spf} := [p - s_1, p - s_2, \dots, p - s_k, p_2, p_3, \dots, p_n]$
zielErreicht
Pfad p ist das Ergebnis

Abb. 11.4-13:
Algorithmus für
die Tiefensuche.

Im Suchbaum entstehen zuerst immer längere Äste, längere Pfade von der Wurzel ausgehend, man spricht von der Tiefe-zuerst-Strategie oder der Tiefensuche. Die **Tiefe-zuerst-Strategie** behandelt stets den gerade neu erzeugten Zustand (Knoten) zuerst und entfernt sich damit bildlich gesprochen schnell vom Startknoten, es geht zuerst in die Tiefe.

Lange Pfade
zuerst

Drei Plätze für Container-Stapel seien von links beginnend durchnummeriert. Die Reihenfolge der Regeln sei so, dass stets zuerst die Regeln für den Container auf Platz 1, danach Platz 2, dann Platz 3 angewendet werden. Regeln für den obersten Container eines Platzes seien so geordnet, dass dieser Container in der Reihenfolge der Platznummern auf einen anderen Platz versetzt wird. Für den Startzustand ergibt sich damit diese Folge von anwendbaren Regeln:

Beispiel:
Container-
Umschlag

- R1: wenn A auf platz 1 dann setze A auf platz 2.
- R2: wenn A auf platz 1 dann setze A auf platz 3.
- R3: wenn C auf platz 2 dann setze C auf platz 1.
- R4: wenn C auf platz 2 dann setze C auf platz 3.

Die Regeln werden auf den Startzustand angewendet und es wird nun der Zustand 1 weiter bearbeitet.

Die Abb. 11.4-14 zeigt den Aufbau des Suchbaums nach der Tiefe-zuerst-Strategie. Zustände, die bereits in dem aktuellen Pfad enthalten sind, werden gelöscht, um zyklisches Suchen ohne Fortschritt zu verhindern.

Werden für einen Zustand keine gültigen Nachfolger erzeugt, so wird der nächste Kandidat bearbeitet. Diese Situation tritt im Zustand 7 auf, wenn der Suchbaum weiter entwickelt wird.

III 11 Logik

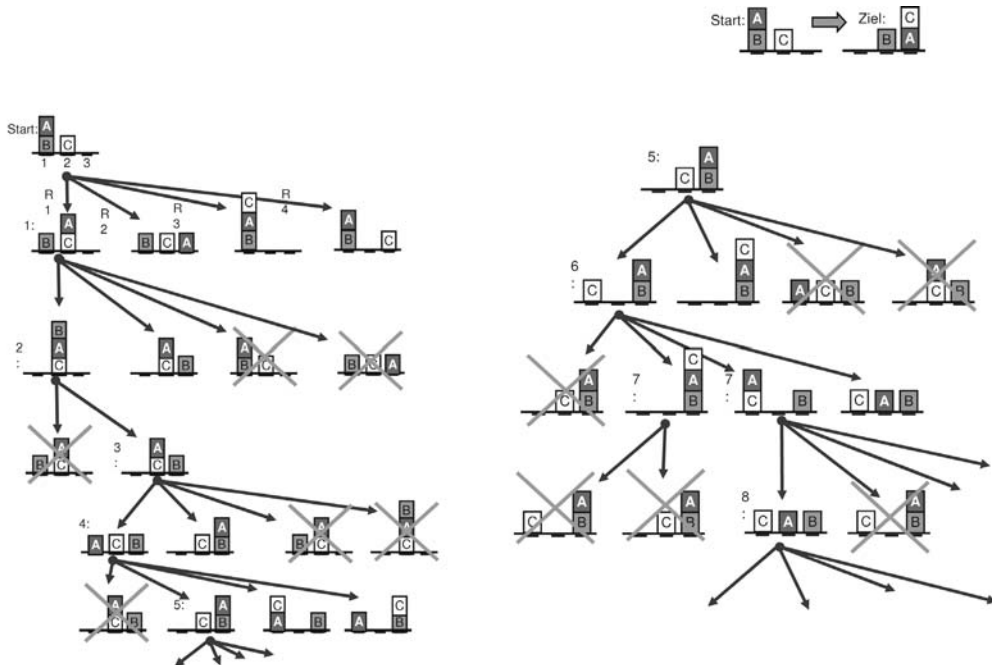


Abb. 11.4-14: Die Tiefe-zuerst-Strategie kommt zu einer Lösung, bei der man nach 19 Schritten vom Start- zum Zielzustand gelangt (Abb. 11.4-15). Offensichtlich ist das *nicht* die optimale Lösung.

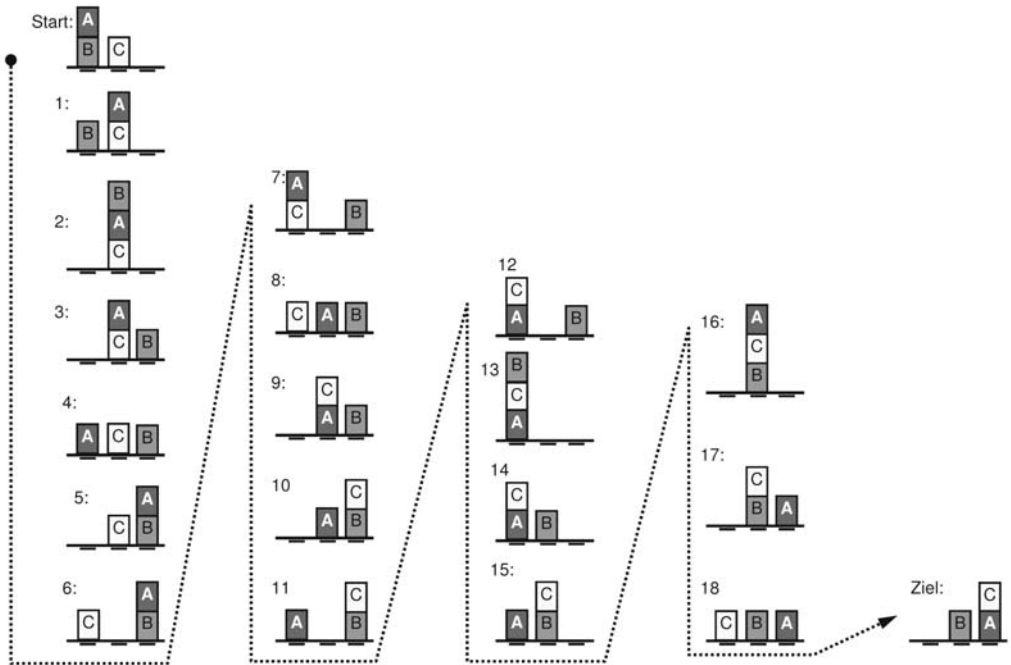
Der Aufwand für eine Tiefe-zuerst-Suche lässt sich dahingehend verringern, dass *nicht* alle Nachfolgezustände berechnet werden, sondern nur der jeweils für die weitere Bearbeitung notwendige Nachfolgezustand. Damit sinkt die Verwaltung erzeugter Zustände erheblich. Es wird stets nur ein Pfad, der aktuelle Pfad betrachtet.

11.4.6.3 Breite-zuerst-Suche

Kurze Pfade
zuerst

Wird die Frage nach dem nächsten zu bearbeitenden Zustand so beantwortet, dass stets ein Zustand gewählt wird, der den kürzesten Pfad zum Startzustand aufweist, so ergibt sich die **Breite-zuerst-Strategie** oder Breitensuche.

Der Algorithmus für die Breitensuche ordnet die neu erzeugten Pfade an das Ende der Folge aller zu behandelnden Suchpfade ein. Damit wird sichergestellt, dass beim Zugriff auf das erste Element der Folge der Suchpfade spf stets ein Pfad p mit der kürzesten Pfadlänge für die weitere Bearbeitung bestimmt wird. Der Pfad p enthalte als letzten Zustand z_m . Für diesen Zustand werden alle Nachfolger s_i bestimmt und es entstehen neue Pfade $p-s_i$ durch das Anhängen von s_i an p . Diese neuen Pfade werden an das Ende der Folge Suchpfade eingeordnet.



Zuerst steht somit nur der Pfad $p=z_0$ in der Folge der Suchpfade. Nach der Ausführung aller anwendbaren Regeln auf z_0 sind die Pfade z_0s_i für alle Nachfolgerzustände s_i von z_0 in Suchpfade enthalten. Im nächsten Schritt wird z_0-z_1 bearbeitet und die neuen Pfade $z_0-z_1-s_i$ werden an das Ende der Suchpfade angefügt.

Nun wird der Unterschied zur Tiefensuche deutlich: Bei der Breitensuche wird als nächstes der Pfad z_0-z_2 bearbeitet, während bei der Tiefensuche nun der Pfad $z_0-z_1-s_1$ weiter entwickelt wird.

Den Algorithmus zur Breitensuche zeigt die Abb. 11.4-16.

Abb. 11.4-15:
Lösung des
Container-
Problems nach
Tiefe-zuerst-
Strategie.
Tiefe-zuerst vs.
Breite-zuerst

Algorithmus

Es entsteht der in der Abb. 11.4-17 dargestellte Suchbaum. Hierbei wurde auf die Darstellung von Knoten, die bereits im Pfad enthalten sind, verzichtet. Die Nummerierung zeigt an, in welcher Reihenfolge die Zustände im Zuge der Bearbeitung entstehen. Es wird eine Situation während der Abarbeitung des Algorithmus betrachtet: Sei $m=8$. Der Zustand 8 wird für die Bearbeitung ausgewählt. Dann ist folgende Situation im System anzutreffen:

suchpfade =

$[z_0-z_2-z_8, z_0-z_2-z_9, z_0-z_2-z_{10}, z_0-z_2-z_{11}, z_0-z_3-z_{12}, z_0-z_4-z_{13}, z_0-z_4-z_{14}, z_0-z_4-z_{15}, z_0-z_1-z_5-z_{16}, z_0-z_1-z_6-z_{17}, z_0-z_1-z_6-z_{18}]$

$p = z_0-z_2-z_8$

$m = 8$

Beispiel:
Container-
Umschlag

III 11 Logik

Abb. 11.4-16: Breittensuche
Algorithmus für
die Breittensuche.

Suchpfade $spf := [z_0]$
zielErreicht := false
Sei Suchpfade $spf = [p_1, p_2, \dots, p_n]$ sowie $p = p_1 = z_0 - z_1 - \dots - z_m$
zielErreicht := istZiel(z_m)
NOT zielErreicht
Berechne alle Zustände s_i , die aus z_m mittels Anwendung einer Regel entstehen: $[s_1, \dots, s_k]$
Lösche Pfad p aus Suchpfade spf
Füge neue Pfade $p - s_1, p - s_2, \dots, p - s_k$ an das Ende von Suchpfade spf hinzu: $spf := [p_2, p_3, \dots, p_n, p - s_1, p - s_2, \dots, p - s_k]$
zielErreicht
Pfad p ist das Ergebnis

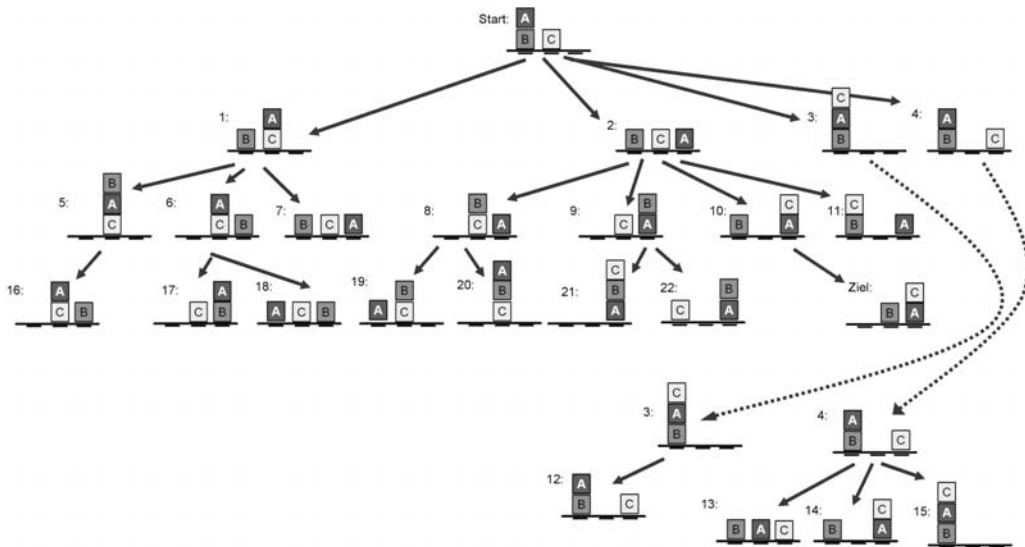


Abb. 11.4-17: Es wird der Knoten z_8 bearbeitet und alle auf diesen Zustand z_8 anwendbaren Regeln ausgeführt. Es entstehen die Zustände: z_{19} und z_{20} . Somit werden die beiden neuen Pfade $z_0 z_2 z_8 z_{19}$, $z_0 z_2 z_8 z_{20}$ an das Ende der Folge suchpfade angefügt. Der Pfad p wird aus Suchpfade gelöscht:

$suchpfade = suchpfade - p + [z_0 - z_2 - z_8 - z_{19}, z_0 - z_2 - z_8 - z_{20}]$

Damit sind nun die folgenden Suchpfade weiter zu untersuchen:

$suchpfade =$

$[z_0 - z_2 - z_8, z_0 - z_2 - z_9, z_0 - z_2 - z_{10}, z_0 - z_2 - z_{11}, z_0 - z_3 - z_{12}, z_0 - z_4 - z_{13}, z_0 - z_4 - z_{14}, z_0 - z_4 - z_{15}, z_0 - z_1 - z_5 z_{16}, z_0 z_1 z_6 z_{17}, z_0 - z_1 z_6 z_{18}, z_0 z_2 z_8 z_{19}, z_0 - z_2 - z_8 - z_{20}]$

Als 23. Zustand wird der Zielzustand erzeugt. Betrachtet man den Algorithmus genauer, so stellt man fest, dass nun auch der Pfad mit dem Zielzustand am Ende $z_0 z_2 z_{10} z_{23}$ an die Folge der Suchpfade angehängt wird, danach erst alle Pfade mit den Zuständen z_{11}, \dots, z_{22} zu bearbeiten sind, bevor der Pfad $z_0 z_2 z_{10} z_{23}$ als Lösung erkannt wird.

Im Unterschied zur Tiefensuche findet die Breitensuche immer die optimale Lösung. Der Aufwand (in Zeit und Speicher) ist für eine Breitensuche jedoch in den meisten Fällen erheblich höher als für eine Tiefensuche.

Optimale Lösung

- Wie ist der Algorithmus für die Breitensuche zu verändern, damit der Algorithmus sofort, wenn ein Zielzustand erreicht wurde, die Bearbeitung beendet und die Lösung bereitstellt?
- Warum ist das geschilderte Problem für den Algorithmus der Tiefensuche nicht so dramatisch?
- Unter welchen Bedingungen ist eine Tiefensuche genauso aufwendig wie eine Breitensuche?

Fragen

11.4.6.4 Heuristische Suche

Die uninformierte Suche, wie die Tiefe-zuerst-Suche oder die Breite-zuerst-Suche, hat keine Informationen über das Suchproblem zur Verfügung. Tiefen- und Breitensuche sind so allgemein, dass sie für jedes Problem eingesetzt werden können. Der Nachteil der Allgemeinheit: Es gibt kein zielgerichtetes Suchen in Abhängigkeit von der Problemstellung.

Nutzt man Wissen über das zu lösende Problem für die Lösung des Suchproblems so spricht man von **heuristischer Suche**. Als eine **Heuristik** bezeichnet man eine Idee, wie die Suche für eine Problemklasse zielgerichteter und damit effizienter durchgeführt werden kann. Eine Heuristik entsteht durch Erfahrung oder detaillierte Problemanalyse. Mit einer Heuristik kann der Suchraum strukturiert und eingegrenzt werden.

Mit Wissen
schneller

Bei der Strategie des **Nächsten Nachbarn** (*Nearest Neighbour Heuristic*) werden die Kosten einer Regelanwendung zugrunde gelegt. Es wird die Regel angewendet, die am kostengünstigsten, d. h. am »billigsten« ist.

Billigste Regel

Bei der Strategie **Bergsteiger** (*Hill Climbing*) erfolgt ein Zustandsübergang nur dann, wenn dieser dichter am Ziel ist. Alle Zweige werden im Suchraum abgeschnitten, bei denen der Nachfolger schlechter als der Vorgänger ist. Die bisherigen Kosten, die der Suchpfad aufweist, spielen wie bei der Bestensuche keine Rolle.

Immer nach oben

Bei der Strategie **Bestensuche** (*Best First*) wird der Pfad gewählt, der gemäß einer Heuristik h am dichtesten am Ziel liegt. Die bisherigen Kosten, die der Suchpfad aufweist, spielen wie bei der Bergsteiger-Strategie keine Rolle.

Am dichtesten am
Ziel

III 11 Logik

Heuristik Die A*-Suche zieht sowohl die bisherigen Kosten eines Suchpfades g als auch den geschätzten Abstand zum Ziel, die Heuristik h in Betracht. Es wird der Pfad, der die beste Bewertung $f=g+h$ aufweist, weiter entwickelt. Der zu durchsuchende Raum hängt wesentlich von der Qualität der Information, d. h. von der Heuristik ab.

11.4.7 Bewertete Regeln

Die bisher betrachteten Regeln sowie Fakten gelten absolut. Aufgrund der Faktenlage ist eine Regel entweder anwendbar oder nicht.

Beispiel **wenn** $\text{umsatz}(X) > 1000$ **dann** $\text{istGuterKunde}(X)$. Ein Kunde mit einem Umsatz von genau 1000 ist danach kein guter Kunde, da die Bedingung nicht erfüllt und die Regel somit nicht anwendbar ist.

Mitunter ist es sinnvoll, die scharfe wahr-falsch-Trennung aufzuheben.

Beispiel Es sind Ferien vom 12. bis zum 27. Februar (einschließlich der Wochenenden). Familie Meier aus Rostock möchte von einem nahe gelegenen Flugplatz in die Sonne fliegen.

Folgende Angebote liegen vor:

- Gran Canaria vom 11. bis zum 25.2. von Hamburg, Berlin oder Frankfurt.
- Lanzarote vom 14. bis zum 28.2. von Hamburg oder Berlin.
- Teneriffa vom 15. bis zum 1.3. von Hamburg oder Berlin.
- Las Palmas vom 14. bis zum 25.2. von Frankfurt oder Berlin.
- Teneriffa vom 31.1. bis zum 16.2. von Rostock-Laage, Hamburg oder Frankfurt.

Leider gibt es kein Angebot, welches alle Bedingungen voll erfüllt: Der Ferienzeitraum stimmt nicht überein oder der Flugplatz ist nicht nahe gelegen.

Hier helfen Bewertungen der Regeln und der Fakten. Die Verknüpfung der Regeln bewirkt dann auch eine Verknüpfung der Bewertungen, sodass am Ende eine Reihe von Lösungen mit Bewertungen angeboten werden können.

Fuzzy

Beispiel 1a Die Fakten, z.B. $\text{nahe}(\text{frankfurt})$, werden durch eine so genannte Fuzzy-Menge beschrieben. Mittels einer Zugehörigkeitsfunktion wird definiert, wie der Begriff »nahe« zu interpretieren ist: Die Abb. 11.4-18 sagt aus, dass eine Entfernung bis zu 100 km definitiv als »nah« angesehen wird und deshalb die Bewertung 1,0 erhält. Orte mit einer Entfernung bis zu 100 km gehören voll zur Menge der nahe gelegenen Orte. Analog wird definiert, dass alle Orte, die über

400 km entfernt sind, nicht zur Menge der nahen Orte gehören. Für die Entfernungen dazwischen wird ein Zugehörigkeitswert (gemäß Kurve) ermittelt.

Fuzzy-Werte können unterschiedlich miteinander verknüpft werden. Eine Möglichkeit ist:

Negation:

$$\mu(\neg x) = 1 - \mu(x)$$

Disjunktion:

$$\mu(x_1 \vee x_2) = \max(\mu(x_1), \mu(x_2))$$

Konjunktion:

$$\mu(x_1 \wedge x_2) = \min(\mu(x_1), \mu(x_2))$$

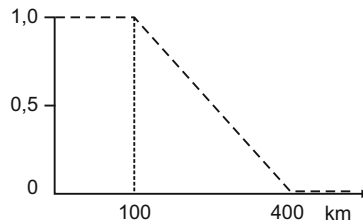


Abb. 11.4-18: Zugehörigkeitsfunktion zur Fuzzy-Menge »nahe«.

Sicherheitsfaktoren

Sicherheitsfaktoren (*certainty factors*) können ebenso wie Fuzzy-Werte eingesetzt werden, um *unscharfe* Sachverhalte zu beschreiben. Dabei wird ein Sicherheitsfaktor (CF) einer Aussage/Hypothese aufgrund der Glaubwürdigkeit MB (*measure of belief*) der Aussage sowie des Misstrauens MD (*measure of disbelief*) gegenüber der Aussage bestimmt.

»Hamburg ist nahe gelegen.« Aus Sicht eines Rostockers spricht dafür (MB), dass es in ca. 2 Stunden mit Bahn oder Auto erreichbar ist. MB(»Hamburg ist nahe«) = 0,9. Der gleiche Grund kann auch als »dagegen« interpretiert werden, schließlich wird der Flughafen Rostock dagegen in 20 Minuten erreicht: MD(»Hamburg ist nahe«) = 0,1. Damit wird der Sicherheitsfaktor CF(h) = MB(h) – MD(h) mit 0,8 ermittelt.

Beispiel 1b

Für die Verknüpfung von Sicherheitsfaktoren gilt:

$$CF(A \wedge B) = \min(CF(A), CF(B))$$

$$CF(\text{Folgerung}) = CF(\text{Vor}) \cdot CF(\text{R: wenn Vor dann Folgerung})$$

Liegen zwei Sicherheitsfaktoren CF1 und CF2 für eine Aussage vor, so konstruiert man daraus einen gemeinsamen Sicherheitsfaktor CF nach folgender Regel:

$$CF = CF1 + CF2 - CF1 \cdot CF2, \text{ falls } CF1 < 0, CF2 < 0$$

$$CF = CF1 + CF2 - CF1 \cdot CF2, \text{ falls } CF1 > 0, CF2 > 0$$

$$CF = (CF1 + CF2) / (1 - \min(|CF1|, |CF2|)), \text{ sonst}$$

Sprechen mehrere Fakten oder Regeln für einen Sachverhalt, so führt die Berechnung des gemeinsamen Sicherheitsfaktors zu einer Verstärkung: $CF > CF1 \wedge CF > CF2$.

Sicherheitsfaktoren wurden zuerst im Expertensystem MYCIN eingesetzt.

Rechnen
mit CF

III 11 Logik

11.4.8 Geschäftsregeln

Der **Geschäftsregel**-Ansatz (*business rules*) ist leicht zu definieren: Es ist die regelbasierte Wissensdarstellung und -verarbeitung für betriebswirtschaftliche Prozesse (siehe »Aufbau von Regeln«, S. 405, »Verkettung von Regeln«, S. 414).

Eine regelbasierte Wissensdarstellung bietet eine Reihe von Vorteilen für Unternehmen: Regeln werden im tagtäglich häufig verwendet, sie sind intuitiv verständlich, entsprechen der menschlichen Denkweise und können als Handlungsanleitung oder auch zum Nachweis eines bestimmten Ergebnisses herangezogen werden.

Schnelle
Reaktionen

Die Unternehmen sind sich immer schneller ändernden Rahmenbedingungen ausgesetzt. Zum einen erfordert eine erfolgreiche Positionierung auf dem Markt ein schnelles Anpassen an das Marktgeschehen. Kürzere Zyklen in der Entwicklung, Produktion und Vermarktung von Produkten und Dienstleistungen sind notwendig. Diese dynamischen Prozesse sind von den IT-Systemen zu unterstützen. Eine Anpassung der IT-Systeme kann jedoch langwierig und kostenintensiv sein. Hier sind flexible Lösungen erforderlich. Die Dynamik wird nicht nur durch die Märkte, sondern auch durch sich ständig ändernde Regelungen erhöht.

Regel =
Dokumentation

In jedem Unternehmen existieren Regelungen für die ablaufenden Prozesse: für die Beschaffung, für die Vermarktung, für die Kundenbeziehungen, für die Abrechnung, für das Personal usw. Die Geschäftsregeln sind vorhanden, sie sind jedoch *nicht* einheitlich formuliert, sondern verbal in staatlichen Regelungen, in Betriebsvereinbarungen, in Arbeitsanweisungen festgeschrieben oder auch in IT-Systemen implementiert.

Beispiel Wer ist verantwortlich, zeichnungsberechtigt? Geschäftsregeln machen deutlich, was sonst in Anweisungen des Managements, in Protokollen oder einfach in den Köpfen der Mitarbeiter mehr oder weniger genau festgehalten ist:

```
R1: wenn X ist_chef und nicht X ist_krank  
    dann unterschreibt X.  
R2: wenn X ist_chef und X ist_krank und Y vertritt X  
    dann unterschreibt Y.
```

Es sei weiterhin die Faktenbasis mit folgenden Fakten gefüllt:
schulz ist_chef; schulz ist_krank; meyer vertritt schulz

Der Geschäftsregel-Ansatz ist geeignet, die Flexibilität der IT-Systeme und damit auch der Unternehmen insgesamt zu erhöhen, indem die vorhandenen Geschäftsregeln extrahiert, explizit dokumentiert und nach Möglichkeit einer computertechnischen Verwaltung und Verarbeitung zugänglich gemacht werden. In [ScGr06] werden die Vorteile des Geschäftsregel-Ansatzes in drei Punkten zusammengefasst:

- + Höhere Transparenz durch explizite Darstellung von fachlichen Begriffen, auf denen die Geschäftsregeln aufbauen, und den Geschäftsregeln selbst.
- + Höhere Flexibilität durch Geschäftsregeln, da explizit dargestellte Regeln leichter geändert werden können.
- + Höhere Effizienz durch automatisierte Verarbeitung der Regeln.

11.4.9 Anwendungen

Regelbasierte Lösungen werden in vielen Bereichen eingesetzt. Einen Überblick kann man sich auf den Seiten der Anbieter von Business-Rules-Management-Systemen verschaffen, in dem man sich deren Referenzanwendungen oder Fallstudien anschaut.

Eine grobe Unterteilung kann in Diagnose- bzw. Planungsaufgaben vorgenommen werden.

Diagnose

In diese Rubrik können sowohl Klassifikations- wie auch Prognose-Aufgaben eingeordnet werden. Die regelbasierte Berechnung von Leistungen oder Tarifen wird ebenso als eine Form einer Diagnose gesehen.

Typisch sind eine hohe Anzahl von Regeln und eine geringe Anzahl von Regelverkettungen. Das Ergebnis ist hier in erster Linie durch eine Menge von Fakten am Ende der Regelverkettung gegeben.

- Kreditkunden-Bewertung Beispiele
- Tarif-Berechnungen in Telekommunikations- oder Energieversorgungsunternehmen
- Diagnose im engeren Sinne (Technik, Medizin, Geologie, Chemie usw.)
- Angebotserstellung

»Echte Regeln« aus einem Diagnosesystem zu Erkrankungen im Bereich der Mundhöhle: Beispiel

wenn veraenderungen(alveolarfortsatz) und
 zahnlockerung und nicht zahnVital und schmerzhaft
 dann Akute Exazerbation einer
 Periodontitis apicalis chronica

wenn gesichtsveränderungen und gesichtsblässe
 und akutEntstanden
 dann vagoasale Synkope oder Schock.

wenn gesichtsveränderung und pigmentstörung und
 nicht fleckenartigBegrenzt und haut(pergamentartig)
 dann Leberzirrose.

III 11 Logik

Mittels Regeln kann eine Hierarchie von Begriffen aufgebaut werden.

Beispiel Eine simple Hierarchie von Bücher-Genres könnte wie folgt aussehen:

1. wenn krimi(X) dann unterhaltung(X).
2. wenn comic(X) dann unterhaltung(X).
3. wenn ki(X) dann fachbuch(X).
4. wenn geographie(X) dann fachbuch(X).
5. wenn kochen(X) dann ratgeber(X).
6. wenn unterhaltung(X) dann (X).
7. wenn fachbuch(X) dann buch(X).
8. wenn ratgeber(X) dann buch(X).

Planung

Planungsaufgaben basieren im Allgemeinen auf einer relativ geringen Anzahl von flexibel verknüpfbaren Regeln. Die Suche nach einer Folge von Regelverknüpfungen, die von der Ausgangssituation zum Ziel führt, steht im Mittelpunkt und kann aufwendig sein. Das Ergebnis wird hierbei durch eine Folge von Regeln gegeben, die den Weg zum Ziel beschreiben.

- Beispiele
- Logistik
 - Reiseplanung
 - Steuerung

11.4.10 Zusammenfassung

Der Einsatz einer regelbasierten Software stellt in zwei Fällen eine Alternative dar: Ist eine algorithmische Lösung *nicht* bekannt oder *nicht* praktikabel, kann eine Problembeschreibung mittels Wenn-Dann-Regeln sowie eine entsprechende Regelverarbeitung doch noch eine Lösung ermöglichen. Insbesondere in wirtschaftlichen Anwendungen sind extrem flexible Softwarelösungen erwünscht, die innerhalb kürzester Zeit neue Geschäftslogiken, wie zum Beispiel Tarifmodelle oder Marketing-Strategien, abbilden können. Hier werden zunehmend regelbasierte Lösungen eingesetzt.

Verfügbare Systeme für die Regelverarbeitung

Insbesondere unter dem Stichwort BRMS (*Business Rules Management System*) sind viele, meist kommerziell vertriebene, Systeme zu finden.

Für das Experimentieren mit Regeln und regelbasierter Wissensverarbeitung bietet sich nach wie vor die Sprache Prolog an, da diese die Denkweise in Regeln nicht nur ermöglicht sondern erzwingt. Frei verfügbar ist SWI-Prolog, für das zudem viele Erweiterungen und Schnittstellen verfügbar sind, siehe Website SWI-Prolog (<http://www.swi-prolog.org/>). Mittels des Prolog-Programms »vorwaertsVerknuepfung.pl« ist eine wenn-dann-Regel-Darstellung möglich.

Prolog

Jess (*Java Expert System Shell*) ist eine in Java programmierte und mit entsprechenden Schnittstellen versehene Umgebung, die die Formulierung und Verarbeitung von Regeln ermöglicht. Die Jess-Notation ist ähnlich wie die Programmiersprache Lisp:

Jess

```
(defrule repariere-Fahrrad
  (answer (ident fahrad-kaputt) (text yes))
  (answer (ident ist-winter) (text no))
  =>
  (recommend-action "Repariere das Fahrrad!")
  (halt)
)
```

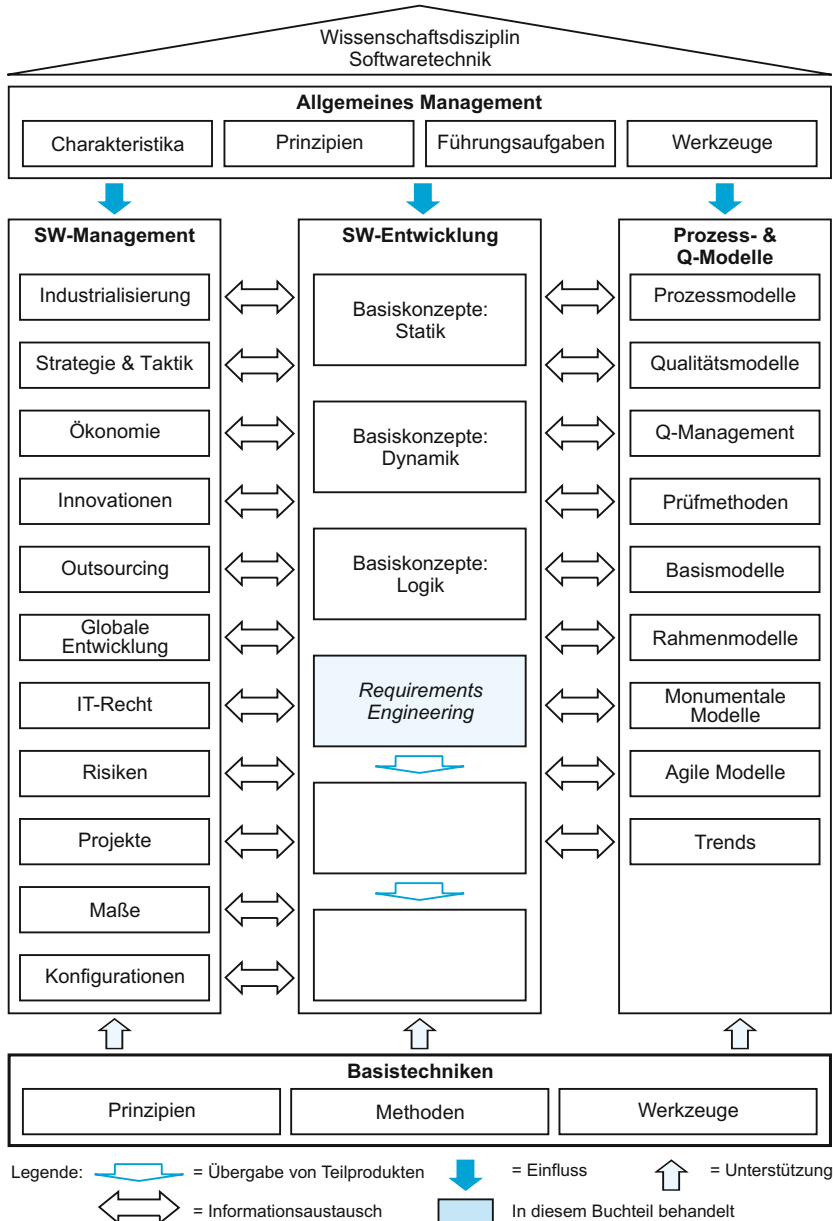
Praxisnäher aber für einfache Beispiele erheblich komplizierter ist das frei verfügbare, in Java programmierte BRMS JBoss-Rules, siehe Website JBoss Enterprise BRMS (<http://www.jboss.com/products/platforms/brms/>) und Website JBoss-Drools (<http://www.jboss.org/drools/>). In [Star07a] wird ein Beispiel mit JBoss-Rules vorgestellt.

JBoss-Rules

Zur Klassifikation

- Regeln werden textuell beschrieben.
- Der Grad der Formalität reicht von informal bis formal, wobei verarbeitbare Regeln formal spezifiziert sein müssen.
- Sie werden in der Spezifikation eingesetzt.
- Sie werden in speziellen Anwendungsbereichen, z.B. bei Planungs- und Diagnoseproblemen, verwendet.

IV Requirements Engineering



IV IV Requirements Engineering

Terminologie

Die Anforderungen an ein neues Softwareprodukt zu ermitteln, zu spezifizieren, zu analysieren, zu validieren und daraus eine fachliche Lösung abzuleiten bzw. ein Produktmodell zu entwickeln, gehört mit zu den anspruchsvollsten Aufgaben innerhalb der Softwaretechnik. Für diese Tätigkeiten gibt es unterschiedliche Begriffe. Lange Zeit wurde in Deutschland dafür hauptsächlich der Begriff »Systemanalyse« verwendet. Der Begriff ist aber etwas missverständlich, da der Systembegriff oft mehr umfasst als ein Softwaresystem. In der deutschsprachigen Literatur wird heute fast ausschließlich der englische Begriff **Requirements Engineering** verwendet (abgekürzt **RE**) – den ich deswegen ebenfalls verwende. Ergänzend oder untergeordnet wird oft noch der Begriff *Requirements Management* benutzt, der auf die Managementaktivitäten im Rahmen des *Requirements Engineering* fokussiert. Die Managementaktivitäten werden *nicht* in diesem Buch, sondern in dem Buch »Lehrbuch der Softwaretechnik – Softwaremanagement« behandelt.

i Bevor auf das *Requirements Engineering* eingegangen wird, wird der Sichtwechsel zwischen Problem und Lösung im Rahmen einer Softwareentwicklung aufgezeigt:

■ »Problem vs. Lösung«, S. 437

Die Bedeutung des *Requirements Engineering* erkennt man daran, wie stark der Erfolg einer Softwareentwicklung davon abhängt:

■ »Bedeutung, Probleme und Best Practices«, S. 439

Im Rahmen des *Requirements Engineering* sind eine Reihe von Aktivitäten durchzuführen, die zu Ergebnissen führen:

■ »Aktivitäten und Artefakte«, S. 443

Alle diese Aktivitäten sind eingebettet in einen Prozess, der je nach Produktart, Produktumfang, Firmenorganisation und Firmengröße sowie Mitarbeiterqualifikation sehr unterschiedlich sein kann:

■ »Der Requirements Engineering-Prozess«, S. 449

Bevor eine Vorgehensweise im *Requirements Engineering* festgelegt werden kann, muss geklärt werden, was Anforderungen überhaupt sind, wer die Anforderungen festlegt und welche Eigenschaften für ein zu entwickelndes Softwareprodukt zu spezifizieren sind:

■ »Anforderungen und Anforderungsarten«, S. 455

Damit Anforderungen analysiert, überprüft und validiert werden können, müssen sowohl jede Anforderung für sich als auch alle Anforderungen zusammen gewisse Qualitätskriterien erfüllen:

■ »Anforderungen an Anforderungen«, S. 475

Neben den Qualitätskriterien muss jede Anforderung es ermöglichen, bestimmte Attribute zu erfassen:

■ »Anforderungsattribute«, S. 479

In der Praxis werden Anforderungen natürlichsprachlich formuliert. Die damit verbundenen Probleme und Möglichkeiten ihrer Vermeidung sollten bekannt sein:

■ »Natürlichsprachliche Anforderungen«, S. 481

Um beim Spezifizieren der Anforderungen wichtige Gesichtspunkte nicht zu vergessen und um immer gleichartig aufgebaute Produktspezifikationen zu haben, gibt es geeignete Schablonen:

■ »Anforderungsschablonen«, S. 485

Eine Vorgehensweise, um systematisch von der Anforderungsermittlung bis zum fertigen Produktmodell zu gelangen, hängt von vielen Randbedingungen ab:

- Liegt eine Ausschreibung vor, dann gibt es in der Regel bereits ein vom Auftraggeber erstelltes Lastenheft.
- Beauftragt eine Fachabteilung die interne IT-Abteilung mit der Softwareerstellung, dann gibt es außer Ideen der Fachabteilung vielleicht noch keine strukturierte schriftliche Unterlage.
- Gibt es bereits ein eingesetztes Softwaresystem, das abgelöst oder verbessert werden soll?
- Ist eine Individualsoftware zu entwickeln oder sind kundenspezifische Anpassungen einer Standardsoftware vorzunehmen?
- Handelt es sich um eine innovative Softwareentwicklung, für die es keine Vorbilder gibt?

Diese Randbedingungen beeinflussen die Methodik. Es kann daher *keine* allgemeingültige Vorgehensweise geben. Im Folgenden wird ein Vorgehensrahmen vorgestellt, der an die jeweiligen Rahmenbedingungen geeignet angepasst werden muss.

Das Ermitteln und Spezifizieren der Anforderungen gliedert sich in mehrere Aktivitäten:

■ »Anforderungen ermitteln und spezifizieren«, S. 503

Anschließend müssen die spezifizierten Anforderungen überprüft werden:

■ »Anforderungen analysieren, validieren und abnehmen«, S. 513

Um ein *Go* oder ein *Stop* für eine Produktentwicklung geben zu können, ist es nötig, den voraussichtlichen Aufwand zu schätzen:

■ »Schätzen des Aufwands«, S. 515

Außerdem ist es nötig, die Anforderungen mit Prioritäten zu versehen:

■ »Anforderungen priorisieren«, S. 543

Sind alle Anforderungen spezifiziert, analysiert, validiert und abgenommen, dann kann auf dieser Grundlage eine fachliche Lösung erarbeitet werden. Dies geschieht in der Regel durch eine Modellierung:

■ »Anforderungen modellieren«, S. 547

Die fachliche Lösung der beiden Fallstudien wird präsentiert:

■ »Fallstudie: SemOrg V1.0 – Die fachliche Lösung«, S. 565

■ »Fallstudie: Fensterheber – Die fachliche Lösung«, S. 575

Anschließend müssen die modellierten Anforderungen einer Qualitätsüberprüfung unterzogen werden:

IV IV Requirements Engineering

- »Modellierte Anforderungen analysieren, verifizieren und abnehmen«, S. 587

Nach der Abnahme der fachlichen Lösung sind die Aufgaben des *Requirements Engineering* beendet.

Ausbildung Ein einheitliches Berufsbild für einen *Requirements Engineer* gibt es noch nicht [Paec08]. Ein »International Requirements Engineering Board (IREB) e.V.« hat Vorschläge für ein Curriculum veröffentlicht. Der »Lehrplan IREB Certified Professional for Requirements Engineering – Foundation Level« definiert ein Basiswissen und kann durch ein Zertifikat bescheinigt werden, siehe IREB-Website (<http://certified-re.de/>). Die Inhalte dieses Lehrbuchs sowie des »Lehrbuchs der Softwaretechnik – Softwaremanagement« decken weite Teile dieses Lehrplans ab.

12 Problem vs. Lösung

Was vs. Wie oder Problem vs. Lösung

Anforderungen legen fest, welche Eigenschaften ein zu entwickelndes Softwaresystem besitzen soll. Sie beschreiben also das Problem, das gelöst werden soll. Die Lösung des Problems, d.h. das »Wie« ist die fachliche Lösung. Die fachliche Lösung wiederum stellt das Problem für die technische Lösung dar usw. Die Abb. 12.0-1 veranschaulicht die verschiedenen Sichtweisen.

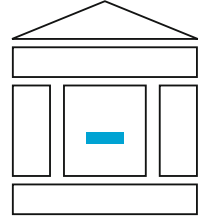
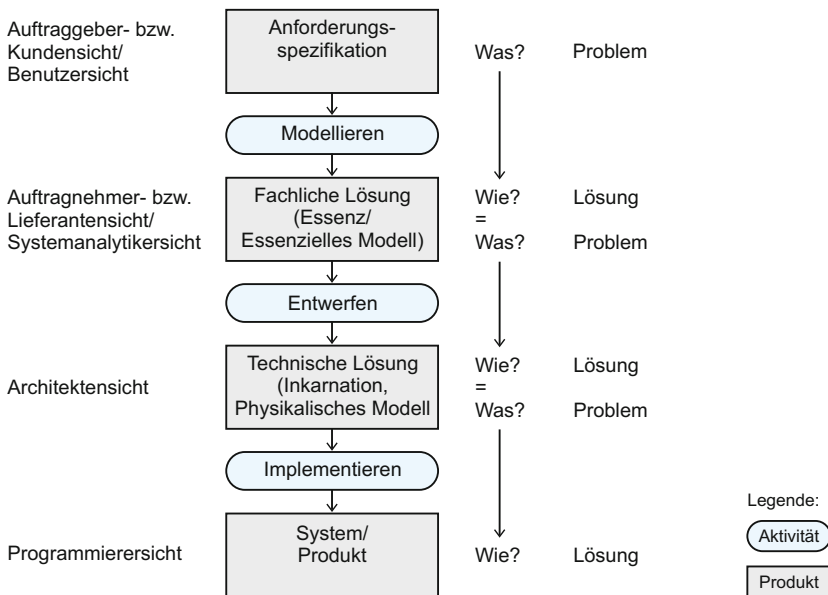


Abb. 12.0-1: Was vs. Wie im Entwicklungsprozess.



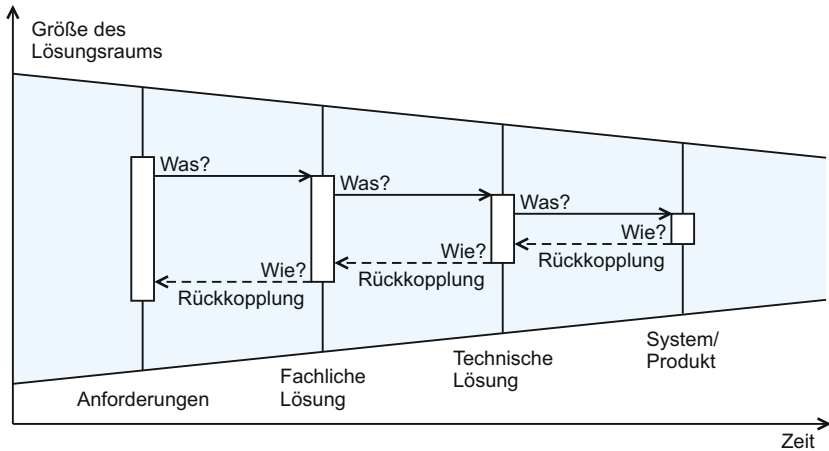
Wie die Abb. 12.0-1 zeigt, alterniert der Entwicklungsprozess zwischen Problem und Lösung. Die Lösung auf einem höheren Abstraktionsniveau ist das Problem für das jeweils niedrigere Abstraktionsniveau.

Lösungsraum

Im Laufe der Softwareentwicklung wird der Lösungsraum immer mehr eingeschränkt. Daher ist es wichtig, dass bei der Aufstellung der Anforderungen möglichst *keine* Einschränkungen für die nachfolgenden Aktivitäten erfolgen. Die Abb. 12.0-2 zeigt, wie der Lösungsraum zunehmend verringert wird.

IV 12 Problem vs. Lösung

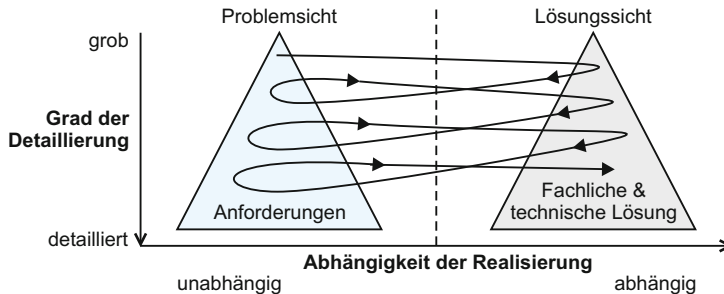
Abb. 12.0-2:
Einschränkung des
Lösungsraums
während der
Entwicklung.



Wechselwirkungen

Zwischen dem jeweiligen Problemraum und dem Lösungsraum gibt es Abhängigkeiten. Es kann durchaus sein, dass beim Versuch, für ein Problem eine Lösung zu finden, es sich herausstellt, dass das Problem so nicht gelöst werden kann. Es muss dann das Problem modifiziert werden oder die Softwareentwicklung muss eingestellt werden. Diese Abhängigkeiten zeigt die Abb. 12.0-3.

Abb. 12.0-3:
Wechselwirkung
zwischen
Anforderungen
und fachlicher
und technischer
Lösung (in
Anlehnung an
[Nuse01]).



Technik vs. Management

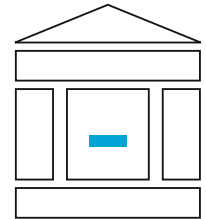
Bei einer Softwareentwicklung sollte klar zwischen den technischen Aspekten und den Managementaspekten unterschieden werden. Anforderungen und Festlegungen, die den Entwicklungsprozess betreffen, gehören in eigene Dokumente. Beispiele für solche Anforderungen sind: Kosten, Fertigstellungstermine, Berichtsverfahren, Festlegung von Entwicklungs- und Qualitätssicherungsmethoden, Kriterien für die Validation und Verifikation, Abnahmeverfahren [IEEE830, S. 10].

13 Bedeutung, Probleme und *Best Practices*

Es gibt eine Vielzahl von Untersuchungen darüber, wie erfolgreich oder nicht erfolgreich Softwareprojekte waren und sind. Eine berühmte Untersuchung, die immer wieder zitiert wird, ist der CHAOS-Report von 1994 [Stan94]. In diesem Report wird gesagt, dass 31 Prozent aller Softwareprojekte *vor* der Fertigstellung abgebrochen werden. Dieser Report wurde wegen fehlender Transparenz, nicht offengelegter Methodik und anderer Mängel kritisiert. Eine neue Untersuchung kommt zu folgenden Erkenntnissen [EmKo08]:

- Es wurden in den Jahren 2005 und 2007 zwei internationale Befragungen durchgeführt.
- Die Hälfte aller Projekte dauerte neun Monate oder weniger.
- Die Anzahl der Entwickler lag zwischen drei und 10, wobei die meisten Projekte weniger als 10 Entwickler hatten.
- Von allen Projekten wurden 2005 **16 Prozent** und 2007 **12 Prozent** abgebrochen, *bevor* irgendetwas ausgeliefert wurde.
- Es wurde *keine* statistisch signifikante Differenz der Abbruchwahrscheinlichkeit bezogen auf die Projektdauer und die Anzahl der Mitarbeiter festgestellt.
- Die vier Hauptgründe für einen Projektabbruch waren:
 - **Änderungen der Anforderungen** und des Umfangs (33 Prozent).
 - Mangelnde Einbindung des höheren Managements (33 Prozent).
 - Engpass im Budget (28 Prozent).
 - Fehlende Projektmanagement-Fähigkeiten (28 Prozent).
- Zwischen 48 Prozent (2005) und 55 Prozent (2007) der ausgelieferten Projekte waren erfolgreich, während zwischen 17 und 22 Prozent der ausgelieferten Projekte *nicht* erfolgreich waren.
- Kombiniert man die abgebrochenen Projekte mit den nicht erfolgreichen Projekten, dann ergeben sich für 2005 **34 Prozent** und für 2007 **26 Prozent**. Dies ist eine hohe Misserfolgsrate für eine angewandte Disziplin.
- Seit 1994 gibt es einen klaren Trend hin zu abnehmenden Abbruchraten – belegbar über alle Studien hinweg.

»Stolze 43 Prozent der im Betrieb festgestellten Fehler in Steuerungs- und Regelungssoftware sind auf Unzulänglichkeiten in der Analysephase oder der Systemspezifikation zurückzuführen. 'Sie wirken sich viel später aus und kosten eine Menge Geld, wenn man sie beheben will'« [cz06].



Empirie

Zitat

IV 13 Bedeutung, Probleme und *Best Practices*

Eine Befragung von 80 Unternehmen im Jahr 2008 durch die Fachhochschule St. Gallen führte zu folgenden Ergebnissen ([Ott08], [Huth09]):

- Fehlerquellen ■ Die häufigste Fehlerquelle bei der Anforderungserfassung sind mit 83 Prozent **sprachliche Fehler** (Unverständlichkeit, Missverständlichkeit, Unquantifizierbarkeit).
- **Logische Fehler** (Widersprüchlichkeit, Redundanz) bei der Anforderungserfassung treten mit 75 Prozent auf.
- **Inhaltliche Fehler** (Falsche Sachverhalte, Unvollständigkeit) machen 73 Prozent aus.

- Änderungen ■ Änderungen der Anforderungen sind die Regel.
- 77 Prozent der Änderungen werden durch Missverständnisse in der Kommunikation der **Stakeholder** verursacht (siehe auch »Anforderungen ermitteln und spezifizieren«, S. 503).
- Wachsende oder sich ändernde Anforderungen an das Gesamtsystem (Produkt ändert sich, Zielgruppe ändert sich usw.) führen in 70 Prozent aller Fälle zu geänderten Anforderungen.
- Pilotbetrieb, Prototypen, Analysen, zusätzliche Know-how-Träger usw. führen zu neuen Erkenntnissen, die in 66 Prozent der Fälle zu Anforderungsänderungen führen.
- 56 Prozent der Unternehmen gaben an, dass eine Änderung im Budget, der Priorisierung oder der Marketingstrategie immer oder oft die Ursache für eine geänderte Anforderung ist.
- In 50 Prozent aller Fälle müssen Anforderungen wegen ungenauen Formulierungen oder falschen Einschätzungen bzgl. der Machbarkeit geändert werden.

Nutzen von RE ■ Zwei Studien der NASA von 1997 und 2001 haben Folgendes ergeben [Eber08, S. 9 f.]:

- Wenn die NASA in ihre Projekte weniger als 5 Prozent Vorbereitungsaufwand – insbesondere für die Anforderungen – investiert, dann führen die Projekte zu starken Verzögerungen. Liegt der Vorbereitungsaufwand zwischen 10 und 20 Prozent des gesamten Projektaufwands, dann liegen die Terminverzögerungen unter 30 Prozent.
- Projekte mit 5 Prozent Aufwand für RE führen zu Kostenüberschreitungen zwischen 80 und 200 Prozent. Liegt der RE-Aufwand zwischen 8 und 14 Prozent, dann liegt die Kostenüberschreitung unter 60 Prozent.

Aufwand von RE ■ Eine Feldstudie mit 15 RE-Teams, davon sechs Teams, die Standardsoftware entwickelt haben, und neun Teams, die Individualsoftware entwickelt haben, führte zu folgenden Erkenntnissen [HoLe01]:

■ 1981 wurden 6 Prozent der Projektkosten und 9 bis 12 Prozent der Projektdauer für das *Requirements Engineering* verwendet.

- 2001 wurden **16 Prozent der Projektkosten** und **39 Prozent der Projektdauer** für RE aufgewendet. In 20 Jahren sind die Ressourcen für RE also signifikant gestiegen.
- Die Größe des RE-Teams liegt im Durchschnitt bei 5 bis 6 Personen, während die gesamte Projektteamgröße bei ca. 17 Personen liegt.

In [Eber08, S. 4 ff.] und [LWE01] werden sieben Risiken aufgeführt, die im RE vermieden werden sollten: 7 RE-Risiken

- Risiko 1: Kunden sind im Projekt unzureichend repräsentiert.
- Risiko 2: Kritische Anforderungen werden übersehen.
- Risiko 3: Es werden nur funktionale Anforderungen berücksichtigt.
- Risiko 4: Anforderungen werden unkontrolliert geändert.
- Risiko 5: Anforderungen beschreiben den Entwurf.
- Risiko 6: Anforderungen werden nicht auf Qualität geprüft.
- Risiko 7: Anforderungen werden perfektioniert.

Neben diesen Risiken gibt es ein generelles interkulturelles bzw. kommunikatives Problem bei der Ermittlung der Anforderungen (siehe z. B. [Dahm00]). In der Regel ist die Kommunikation zwischen dem Auftraggeber bzw. der Fachabteilung und dem Auftragnehmer, d. h. den Softwareentwicklern, problembehaftet. Folgende Probleme treten oft auf: Probleme

- Der Auftraggeber weiß zu Projektbeginn *nicht* genau, was er will. Oft hört man die Aussage des Auftraggebers: »Wenn ich die Software sehe, dann kann ich sagen, was ich will.«
- Der Auftraggeber kann das, wovon er weiß, dass er es will, *nicht* vollständig mitteilen.
- Der Auftraggeber versteht *nicht*, was der Softwareentwickler außer den vorgelegten Beispielen noch leisten könnte.
- Der Auftraggeber weiß *nicht*, welche Software möglich wäre, wenn der Softwareentwickler besser über seine Bedürfnisse informiert wäre.

»Eine recht naive und gefährliche Annahme versteckt sich außerdem in der Zuversicht, daß Entwickler und Anwender es schon rechtzeitig merken werden, wenn Mißverständnisse auftreten, und diese durch Nachfragen leicht beseitigen können. Vielmehr ist es die Regel, daß Verständnislücken auf beiden Seiten viel zu spät und manchmal gar nicht aufgeklärt werden, so daß für den Anwender Unnützes mit viel Mühe implementiert wird, und zugleich zentrale, aber bislang implizit gebliebene Anforderungen gerade noch nicht erfüllt sind. [...] Um brauchbare Software herstellen zu können, sollte die Kultur der Anwender berücksichtigt werden, da die Software für den Anwender entwickelt wird und in seine Kultur eingebettet werden soll. Dazu müssen sich aber beide dieser Situation bewußt sein.« [Dahm00, S. 175]. Zitat

IV 13 Bedeutung, Probleme und *Best Practices*

Best Practices Nach [HoLe01, S. 65] wenden die besten RE-Teams die in der Tab. 13.0-1 aufgeführten *Best Practices* an.

Fokus	<i>Best Practice</i>	Kosten der Einführung	Kosten der Anwendung	Hauptnutzen
Wissen	Kunden & Benutzer einbinden	Gering	Mittel	Besseres Verständnis der »echten Bedürfnisse«
Wissen	Identifizieren & Konsultieren aller möglichen RE-Quellen	Gering – Mittel	Mittel	Verbesserte Überdeckung der Anforderungen
Wissen	Erfahrene Projektmanager & Teammitglieder einsetzen	Mittel – Hoch	Mittel	Besser voraussagbare Leistung
Ressourcen	15–30 % der gesamten Ressourcen für RE	Gering	Mittel – Hoch	Beibehaltung einer qualitativ hochwertigen Spezifikation während des Projekts
Ressourcen	Anforderungsschablonen & Beispiele zur Verfügung stellen	Gering – Mittel	Gering	Verbesserte Spezifikationsqualität
Ressourcen	Gute Beziehungen zu allen Beteiligten & Betroffenen herstellen	Gering	Gering	Bessere Zufriedenstellung der Kundenbedürfnisse
Prozess	Anforderungen priorisieren	Gering	Gering – Mittel	Aufmerksamkeit auf die wichtigsten Kundenwünsche fokussieren
Prozess	Ergänzende Modelle zusammen mit Prototypen entwickeln	Gering – Mittel	Mittel	Eliminieren von Spezifikationswidersprüchen & Inkonsistenzen
Prozess	Eine Nachverfolgungsmatrix pflegen	Mittel	Mittel	Explizite Verweise zwischen Anforderungen & Ergebnissen
Prozess	<i>Peer Reviews</i> durch Benutzer, Szenarien & <i>Walk-Throughs</i> zum Validieren & Verifizieren der Anforderungen	Gering	Mittel	Genauere Spezifikationen & höhere Kundenzufriedenheit

Tab. 13.0-1: *Best Practices im RE.*

14 Aktivitäten und Artefakte

Im Rahmen einer Softwareentwicklung müssen Aktivitäten durchgeführt werden, die zu Ergebnissen – im Folgenden **Artefakte** (*artifacts*) genannt – führen. Eine Aktivität wird durch Mitarbeiter ausgeführt, die definierte **Rollen** einnehmen. Unter Beachtung von Methoden, Richtlinien, Konventionen, Checklisten und Schablonen wird – ausgehend von einem oder mehreren gegebenen Artefakten oder Informationen – ein neues Artefakt erstellt oder der Zustand oder der Inhalt gegebener Artefakte geändert. Für die Durchführung der Aktivität werden in der Regel vom Softwaremanagement vorgegebene Werkzeuge eingesetzt (Abb. 14.0-1).

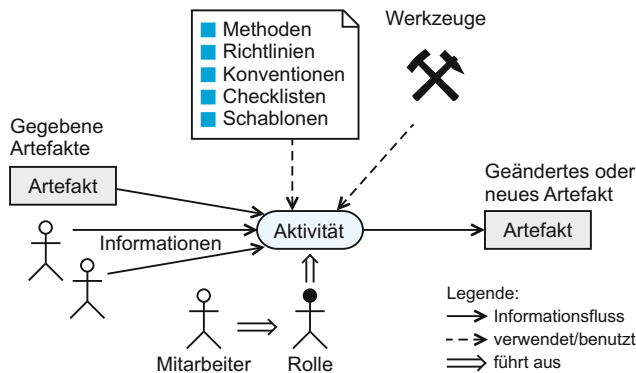
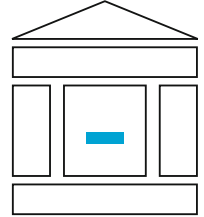


Abb. 14.0-1:
Durchführung
einer Aktivität.

Eine Artefaktbeschreibung legt die Inhalte und das Layout des Artefakts fest. Sie erfolgt nach einer festen Schablone, der **Artefakt-Schablone** (*artifact template*). Jedes Artefakt enthält einen Artefakt-Vorspann, der die Historie des Artefakts aufzeigt.

Rollen im RE

Die Aufgaben, die im *Requirements Engineering* zu erledigen sind, werden von der **Rolle** des **Requirements Engineer** erledigt, im Deutschen auch Systemanalytiker, Anforderungsingenieur oder Anforderungsanalytiker genannt.

Welche Kompetenzen sollte Ihrer Meinung nach ein *Requirements Engineer* besitzen? Frage

Ein *Requirements Engineer* sollte über folgende Kompetenzen verfügen [IREB07, S. 9]: Antwort

- Analytisches Denken
- Selbstbewusstes Auftreten

IV 14 Aktivitäten und Artefakte

- **Empathische** Fähigkeiten
- Moderationsfähigkeit
- Überzeugungsfähigkeit
- Kommunikationsfähigkeiten
- Sprachliche Kompetenz
- Methodische Kompetenz

In [Balz08] werden folgende Kompetenzen genannt:

- Abstraktes Denken (Vom Konkreten zum Abstrakten)
- Flexibilität
- Hohe Kommunikationsbereitschaft und -fähigkeit
- Hineindenken in andere Begriffs- und Vorstellungswelten
- Fachwissen aus den Anwendungsgebieten

Ein *Requirements Engineer* wird oft durch einen oder mehrere **Anwendungsspezialisten** unterstützt, die spezielles Wissen der Domäne besitzen, für die das Produkt entwickelt werden soll. Ein Anwendungsspezialist sollte über folgende Kompetenzen verfügen:

- Abstraktes Denken (Vom Konkreten zum Abstrakten)
- Ganzheitliches Denken, z. B. in Geschäftsprozessen
- Hohe Kommunikationsbereitschaft
- Breites Modellierungswissen über das Anwendungsgebiet
- Kenntnis vorhandener Softwaresysteme für das Anwendungsgebiet
- Fähigkeit, das Anwendungsgebiet aufgabengerecht zu strukturieren
- Automatisierungsmöglichkeiten des Anwendungsgebiets einschätzen können unter Berücksichtigung von wirtschaftlichen und ergonomischen Gesichtspunkten

Die Rollen *Requirements Engineer* und Anwendungsspezialist werden oft nicht unterschieden, sondern zu einer Rolle zusammengefasst.

Aktivitäten im RE

Unabhängig von einem Prozessmodell sind im *Requirements Engineering* folgende Aktivitäten durchzuführen:

- **Anforderungen ermitteln:** Die Anforderungen müssen von den Beteiligten und Betroffenen sowie sonstigen Quellen, z.B. Normen, Gesetzestexte, Standards, systematisch gewonnen werden.
- **Anforderungen spezifizieren:** Die ermittelten Anforderungen müssen spezifiziert werden, d.h. unter Berücksichtigung von festgelegten Methoden, Richtlinien, Konventionen, Checklisten und Schablonen beschrieben werden.
- **Anforderungen analysieren, validieren und abnehmen:** Die spezifizierten Anforderungen müssen anhand von Richtlinien und Checklisten analysiert werden, um Fehler, Missverständnis-

se und Unklarheiten zu beseitigen. Die **Validation** hat das Ziel, die Eignung bzw. den Wert des spezifizierten Produkts auf seinen Einsatzzweck hin zu überprüfen.

- **Anforderungen modellieren:** Die analysierten und validierten Anforderungen bilden den Ausgangspunkt für die Modellierung der fachlichen Lösung.
- **Anforderungen analysieren, verifizieren und abnehmen:** Das entstehende Modell wird permanent analysiert, führt zu Rücksprachen und evtl. Änderungen an der Spezifikation. Außerdem wird das Modell gegen die Spezifikation verifiziert. Der Begriff **Verifikation** bedeutet in diesem Kontext die Überprüfung der Übereinstimmung zwischen einem Softwareprodukt und seiner Spezifikation.
- **Anforderungen verwalten und managen:** Alle Anforderungen müssen verwaltet werden – was in der Regel mit Hilfe einer Softwareentwicklungsumgebung oder eines Softwarewerkzeugs passiert. Unter Managen ist die Änderung, Nachverfolgbarkeit, die Versionierung usw. der Anforderungen zu verstehen.

Alle Aktivitäten zusammen werden oft auch **Spezifizieren** oder **Definieren der Anforderungen** genannt.

Artefakte des RE

In Abhängigkeit vom verwendeten Prozessmodell, von der Art der Auftraggeber-Auftragnehmer-Situation und von der zu entwickelnden Produktart sind die Artefakte sowohl bezogen auf die Anzahl als auch bezogen auf den Aufbau unterschiedlich. Grobgranular lassen sich zwei Artefakte unterscheiden:

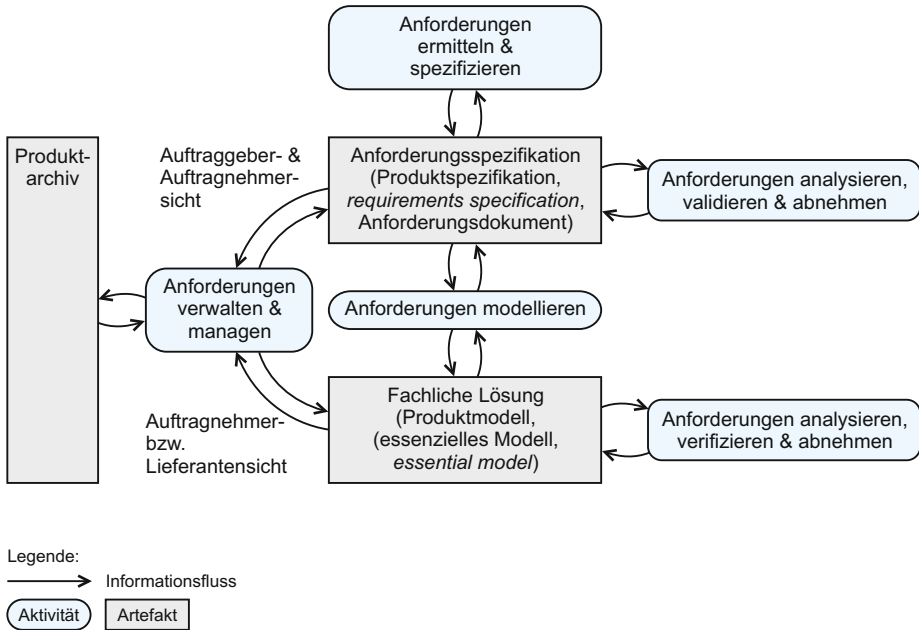
- Die **Anforderungsspezifikation** (*requirements specification*) bzw. die **Produktspezifikation**: Enthält alle ermittelten, spezifizierten, analysierten und validierten Anforderungen aus der Kunden- bzw. Auftraggebersicht. In die Anforderungsspezifikation integriert oder als eigenständiges Artefakt ist ein **Glossar** der verwendeten domänenspezifischen Fachbegriffe.
- Die **fachliche Lösung** bzw. das Produktmodell oder das **essenzielle Modell** (*essential model*): Beschreibt die analysierte und verifizierte fachliche Lösung des Produkts aus der Auftragnehmer- bzw. Lieferantensicht.

Einen grobgranularen Überblick über die Aktivitäten und Artefakte gibt die Abb. 14.0-2.

Oft gibt es keine klare Trennung zwischen Anforderungsspezifikation und fachlicher Lösung. Oder zum *Requirements Engineering* wird nur die Anforderungsspezifikation gezählt. Die fachliche

Hinweis

IV 14 Aktivitäten und Artefakte



*Abb. 14.0-2:
Grobgranulare
Übersicht über die
Aktivitäten und
Artefakte des
Requirements
Engineering.*

Lösung und die technische Lösung, d. h. die Produktarchitektur, fließen dann oft ineinander, was aber unbedingt vermieden werden sollte.

Die Anforderungsspezifikation

Bei der Anforderungsspezifikation – oft auch Anforderungsdokument genannt – kann es sich um ein Artefakt, aber auch um zwei Artefakte handeln.

In manchen Prozessmodellen ist das Ermitteln und Spezifizieren der Anforderungen in zwei Teile geteilt (Abb. 14.0-3):

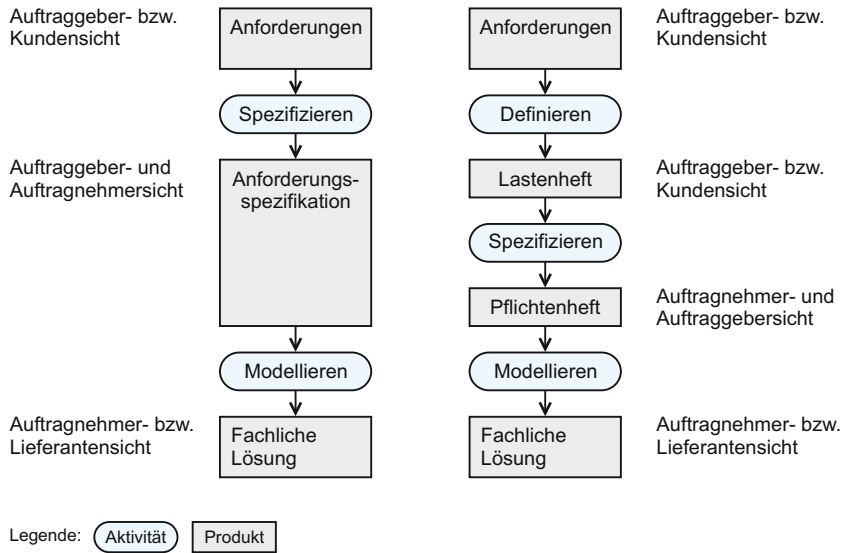
- In eine Ausschreibungs- oder Planungsphase und
- in eine Spezifikationsphase.

Diese Zweiteilung ist immer dann notwendig, wenn ein Auftraggeber eine Softwareentwicklung ausschreibt, d. h. der Auftragnehmer noch nicht feststeht. Der Auftraggeber erstellt dann selbstständig – oder mit Hilfe von Beratern – ein Lastenheft.

Definitionen

Lastenheft: »Zusammenstellung aller Anforderungen des Auftraggebers hinsichtlich Liefer- und Leistungsumfang. Im Lastenheft sind die Anforderungen aus Anwendersicht einschließlich aller Randbedingungen zu beschreiben. Diese sollen quantifizierbar und prüfbar sein. Im Lastenheft wird definiert WAS und WO-

14 Aktivitäten und Artefakte IV



FÜR zu lösen ist. Das Lastenheft wird vom Auftraggeber oder in dessen Auftrag erstellt. Es dient als Ausschreibungs-, Angebots- und/oder Vertragsgrundlage« [VDI2519, S. 2].

Lastenheft: »Vom Auftraggeber festgelegte Gesamtheit der Forderungen an die Lieferungen und Leistungen eines Auftragnehmers innerhalb eines Auftrags« [DIN69905, S. 3].

Ist der Auftrag erteilt, dann erstellt der Auftragnehmer zusammen mit dem Auftraggeber ein Pflichtenheft, das wesentlich ausführlicher als das Lastenheft ist.

Pflichtenheft: »Beschreibung der Realisierung aller Anforderungen des Lastenheftes. Das Pflichtenheft enthält das Lastenheft. Im Pflichtenheft werden die Anwendervorgaben detailliert und die Realisierungsanforderungen beschrieben. Im Pflichtenheft wird definiert, WIE und WOMIT die Anforderungen zu realisieren sind. [...] Das Pflichtenheft wird in der Regel nach Auftragserteilung vom Auftragnehmer erstellt, falls erforderlich unter Mitwirkung des Auftraggebers. Der Auftragnehmer prüft bei der Erstellung des Pflichtenhefts die Widerspruchsfreiheit und Realisierbarkeit der im Lastenheft genannten Anforderungen. Das Pflichtenheft bedarf der Genehmigung durch den Auftraggeber« [VDI2519, S. 2 f.].

Abb. 14.0-3: Einstufiges (links) und zweistufiges (rechts) Vorgehen im Requirements Engineering.

Definitionen

IV 14 Aktivitäten und Artefakte

Pflichtenheft: »Vom Auftragnehmer erarbeitete Realisierungsvorgaben aufgrund der Umsetzung des vom Auftraggeber vorgegebenen Lastenheftes« [DIN69905, S. 3].

Liegt eine Trennung Auftraggeber – Auftragnehmer vor, dann ist in der Regel eine zweistufige Spezifikation notwendig. Aber auch wenn diese Situation nicht vorliegt, kann eine zweistufige Spezifikation sinnvoll sein. Bevor eine Softwareentwicklung genehmigt wird – z. B. innerbetrieblich – werden die Anforderungen oft zunächst grob zusammengestellt und danach eine Wirtschaftlichkeitsbetrachtung angestellt. Am Ende einer solchen Planungsphase steht dann das »Stop or Go« für die eigentliche Entwicklung. Für eine solche Planungsphase ist der Aufwand zur Erstellung eines Pflichtenhefts zu hoch. Daher wird zunächst nur ein Lastenheft erstellt.

- + Vorteilhaft bei diesem Vorgehen ist, dass es ein klar abgegrenztes Ende der RE-Phase gibt, zu dessen Zeitpunkt auch der Auftraggeber sich klar für die fachliche Lösung entscheiden muss.
- Nachteilig ist, dass spätere Änderungen immer über den Änderungsprozess abgewickelt werden müssen.

Wichtig ist, dass definierte Anforderungen und fachliche Lösungen in einem Wiederverwendungsarchiv gesammelt werden, um sie projektübergreifend nutzen zu können.

Die fachliche Lösung

Die fachliche Lösung bzw. das Produktmodell besteht – in Abhängigkeit von der verwendeten Methode – in der Regel aus verschiedenen Artefakten:

- OOA-Modell (siehe »OOA-Methode«, S. 559)
- GUI-Konzept oder -Prototyp
- evtl. Benutzerhandbuch

15 Der Requirements Engineering-Prozess

In großen Unternehmen und Softwarehäusern ist die Softwareentwicklung eingebettet in die Unternehmens- und Produktstrategie. Der Weg von einer Idee hin zu einem Produkt besteht aus vielen Schritten, in denen Annahmen gemacht, geprüft und substantiiert werden. Aus vielen ursprünglichen Ideen und Visionen werden am Ende nur einzelne wenige Produkte entwickelt. Dies liegt an den stark steigenden Kosten, sobald Entwicklungsprojekte und das begleitende Marketing gestartet werden. Man versucht innerhalb der Strategieentwicklung daher, möglichst viele Ideen frühzeitig zu bewerten und nur für jene mit großem Potenzial die Entwicklung zu starten. Die Abb. 15.0-1 zeigt diese Entwicklung vom Ideenmanagement hin zum konkreten Portfolio von Produkten.

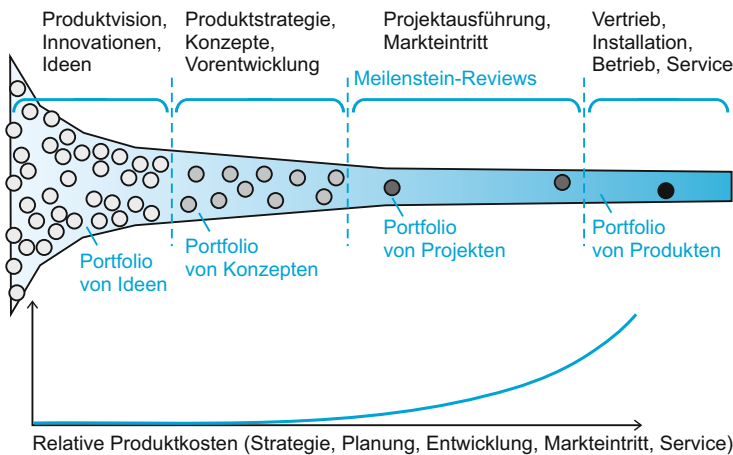
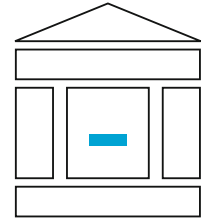


Abb. 15.0-1: Ideen bewerten.

Auf die Aktivitäten im Vorfeld einer Softwareentwicklung wird im »Lehrbuch der Softwaretechnik – Softwaremanagement« eingegangen.

Im Folgenden wird davon ausgegangen, dass die prinzipielle Entscheidung für die Entwicklung eines Softwareprodukts gefallen ist und dass es nun darum geht, die Anforderungen zu ermitteln und eine fachliche Lösung zu erstellen. Nach der Ermittlung der Anforderungen und einer Aufwandsschätzung kann u. U. noch eine Überprüfung dahingehend erfolgen, dass ein explizites »Go« oder auch ein »Stop« der weiteren Entwicklung erfolgt.

Annahme

IV 15 Der *Requirements Engineering*-Prozess

Granularität Im Rahmen einer Softwareentwicklung kann der RE-Prozess grob- oder feingranular sein. Traditionell endet der RE-Prozess mit Fertigstellung und Abnahme der **fachlichen Lösung**. Anschließend folgt der Entwurfs-Prozess.

Alle Erfahrungen haben aber gezeigt, dass Anforderungen während der Entwicklung von den *Stakeholdern* noch häufig geändert werden.

Frage Wie hoch schätzen Sie die Änderungsquote der Anforderungen pro Monat?

Antwort Anforderungen ändern sich während einer laufenden Softwareentwicklung typischerweise mit 1 bis 5 Prozent des Projektumfangs (Aufwand) pro Monat [Eber08, S. 331].

Daraus ergibt sich, dass entweder der RE-Prozess parallel zur weiteren Entwicklung weiterlaufen muss oder dass es ein definiertes Änderungsmanagement gibt.

Alternativen Für den RE-Prozess gibt es daher folgende Alternativen (vgl. [Pohl07, S. 30 ff.]):

- RE als eigenständige Entwicklungsphase + definiertes Änderungsmanagement
- RE entwicklungsbegleitend
- RE produktübergreifend

RE als eigenständige Entwicklungsphase und definiertes Änderungsmanagement

In vielen Prozessmodellen wird das RE als eigenständige Phase am Anfang jeder Softwareentwicklung durchgeführt. Die Anforderungen werden für jede Softwareentwicklung unabhängig von anderen Softwareentwicklungen durchgeführt. Muss die RE-Phase abgeschlossen sein, bevor mit der Entwurfsphase begonnen wird, dann liegt ein sequenzielles Prozessmodell vor. Beginnt die Entwurfsphase bereits, bevor die RE-Phase abgeschlossen ist, dann liegt ein nebenläufiges Prozessmodell vor (Abb. 15.0-2).

Unabhängig von diesen beiden Modellen, kann das RE entweder *alle* Anforderungen an ein Produkt enthalten oder *nur* Anforderungen an ein Teilprodukt. Da nach Abschluss des RE-Prozesses in der Regel kontinuierlich Anforderungsänderungen gefordert werden, muss parallel zu den weiteren Entwicklungsphasen ein definiertes Änderungsmanagement (*Change Management*) etabliert sein oder werden, um systematisch mit Änderungen umzugehen. Im Änderungsmanagement werden Änderungswünsche, Fehler und Probleme, die während der Systementwicklung oder -nutzung auftreten, behandelt und gelöst. Die Abb. 15.0-3 zeigt, wie ein Änderungsmanagement aussehen kann.

15 Der Requirements Engineering-Prozess IV

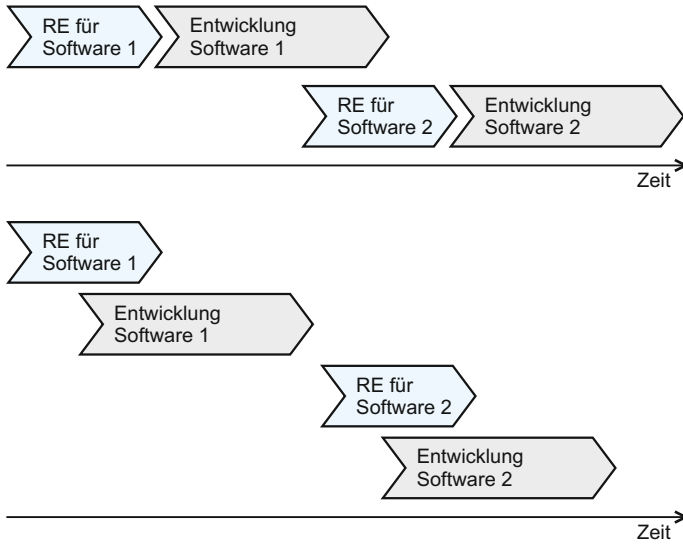


Abb. 15.0-2:
Sequenzielles
(oben) und
nebenläufiges
(unten),
phasenbezogenes
Requirements
Engineering (RE).

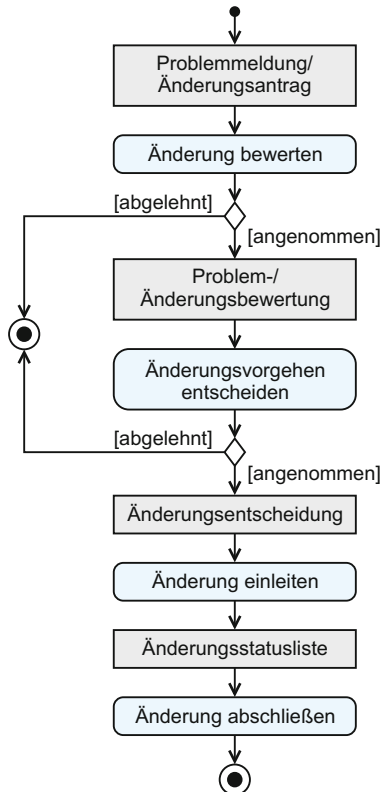


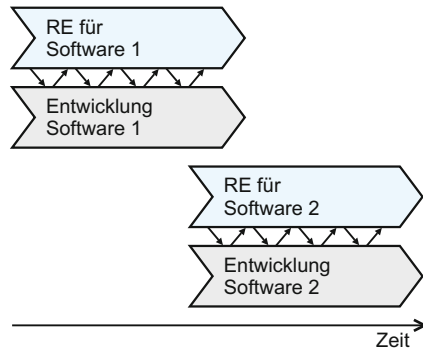
Abb. 15.0-3:
Mögliche Zustände
einer Änderung
und zugehörige
Formulare.

IV 15 Der *Requirements Engineering*-Prozess

RE entwicklungsbegleitend

Beim entwicklungsbegleitenden RE liegt ein phasenübergreifendes *Requirements Engineering* vor, d.h. es handelt sich um eine Querschnittstätigkeit, die den gesamten Entwicklungsprozess begleitet und eine konsistente Erfassung und Verwaltung von Anforderungen sicherstellt (Abb. 15.0-4). Zwischen dem RE und den anderen Entwicklungsphasen gibt es Wechselwirkungen. Änderungen, die sich z.B. aus dem Softwareentwurf ergeben, werden an das RE übergeben.

Abb. 15.0-4: Phasenübergreifendes *Requirements Engineering*.

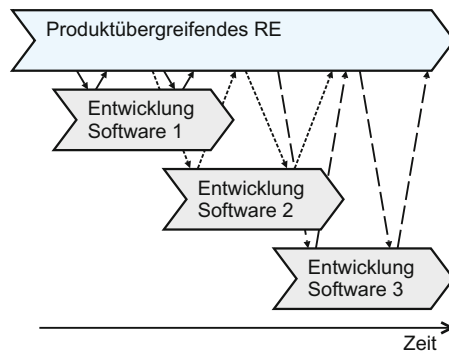


Der Aufwand für das *Requirements Engineering* fällt also nicht konzentriert am Entwicklungsanfang an, sondern ist über die gesamte Entwicklungszeit verteilt (siehe auch »Schablonen für agile Entwicklungen«, S. 497).

RE produktübergreifend

Befinden sich die zu entwickelnden Softwareprodukte in einer gemeinsamen Domäne, dann ist es sinnvoll, das RE produktübergreifend auszulegen (Abb. 15.0-5). Dies ist in der Regel bei einer Produktlinien-Entwicklung der Fall (siehe auch »Lehrbuch der Softwaretechnik – Softwaremanagement«).

Abb. 15.0-5: Produktübergreifendes *Requirements Engineering*.



15 Der *Requirements Engineering*-Prozess IV

- + Vorteilhaft ist, dass eine fortlaufende Wissensbasis für die Produktdomäne entsteht, auf die bei neuen Produkten sofort zugegriffen werden kann.
- + Die für das RE zuständigen Mitarbeiter können sich ganz auf die Entwicklung und Pflege der RE-Wissensbasis konzentrieren.
- Nachteilig ist, dass ein erhöhter Koordinierungsaufwand zwischen den Projekten entsteht.

16 Anforderungen und Anforderungsarten

Bevor ein Softwaresystem entwickelt werden kann, muss festgelegt werden, welche Anforderungen es erfüllen soll.

Wie definieren Sie den Begriff »Anforderungen«?

Frage

Im Duden wird der Begriff wie folgt definiert:

Antwort

»Anforderung: <meist Plural> das, was man von jmdm. als [Arbeits]leistung erwartet, von ihm verlangt.« (Duden Bedeutungen, PC-Bibliothek).

Übertragen auf ein Softwaresystem lässt sich der Begriff wie folgt definieren:

Anforderungen (*requirements*) legen fest, was man von einem Softwaresystem als Eigenschaften erwartet.

Definition

Der Begriff Leistung wurde bewusst nicht in die Definition übernommen, da Leistung oft einen Bezug zurzeit hat. Dies trifft auf Softwaresysteme aber nur in Teilbereichen zu.

In dieser Definition sind zwei Dinge noch offen:

1 Wer ist »man«?

2 Was sind »Eigenschaften«?

Beides wurde in der Definition bewusst *nicht* genauer festgelegt, da es dazu verschiedene Antworten gibt. Man kann die Worte »man« und »Eigenschaften« auch als Platzhalter verstehen, die je nach Perspektive durch konkretere Begriffe substituiert werden.

Wer ist »man«?

An jeder Softwareentwicklung im industriellen Maßstab wirken verschiedene und meist auch viele Personen mit. Alle Personen und Organisationen, die ein Interesse an einer Softwareentwicklung haben und von einer Softwareentwicklung bzw. dem Einsatz des Softwaresystems betroffen sind, werden mit dem englischen Begriff **Stakeholder** bezeichnet. Der Begriff bedeutet im Deutschen etwa: Akteur, Interessenvertreter. »*The term stakeholder generalizes the traditional notion of customer or user in requirements engineering to all parties involved in a system's requirements*« [GIWi07, S. 18].

Gibt es einen individuellen Auftraggeber, dann wird dieser wesentliche Anforderungen festlegen. Wird ein Produkt für den anonymen Markt entwickelt, dann werden die Marketingabteilung und der Vertrieb die wichtigen Anforderungen bestimmen. Je nach Situation ist also »man« festzulegen.

IV 16 Anforderungen und Anforderungsarten

Was sind »Eigenschaften«?

Frage Welche Eigenschaften müssen Ihrer Meinung nach für ein neues Softwaresystem festgelegt werden? Versuchen Sie eine Klassifikation.

Antwort Visionen und Ziele, die mit dem Produkt erreicht werden sollen, stehen oft am Anfang einer Produktspezifikation:

■ »Visionen und Ziele«, S. 456

Rahmenbedingungen legen Restriktionen für das zu entwickelnde System oder den Entwicklungsprozess fest:

■ »Rahmenbedingungen«, S. 459

Die Umgebung, in die das zu entwickelnde System eingebettet wird, muss definiert werden:

■ »Kontext und Überblick«, S. 461

Die eigentlichen Eigenschaften eines Softwaresystems werden in der Regel in **funktionale** und **nichtfunktionale** Anforderungen unterteilt.

Eine funktionale Anforderung lässt sich wie folgt definieren:

Definition Eine **funktionale Anforderung** legt eine vom Softwaresystem oder einer seiner Komponenten bereitzustellende Funktion oder bereitzustellenden Service fest.

Funktionale Anforderungen lassen sich gliedern in (siehe »Basiskonzepte«, S. 99):

- ☐ Anforderungen, die die **Statik** des Systems beschreiben,
- ☐ Anforderungen, die die **Dynamik** des Systems beschreiben und
- ☐ Anforderungen, die die **Logik** des Systems beschreiben.

Unter einer nichtfunktionalen Anforderung versteht man oft alles, was *keine* funktionale Anforderung ist, was aber nicht ganz stimmt:

■ »Nichtfunktionale Anforderungen«, S. 463

Nichtfunktionale Anforderungen lassen sich unter dem Begriff Qualitätsanforderungen subsumieren:

■ »Box: Qualitätsmerkmale nach ISO/IEC 9126-1«, S. 468

Bereits bei der Festlegung von Anforderungen muss überlegt werden, wie die Anforderungen auf Einhaltung überprüft werden sollen:

■ »Abnahmekriterien«, S. 471

TBD Anforderungen, zu denen noch keine Festlegungen getroffen werden können, werden oft mit TBD gekennzeichnet. TBD steht für *to be defined* (noch nicht definiert) oder *to be determined* (noch nicht festgelegt).

16.1 Visionen und Ziele

Bevor funktionale und nichtfunktionale Anforderungen sowie Rahmenbedingungen festgelegt werden, sollten die Visionen und Ziele des Systems aufgestellt werden.

Warum sollten Visionen und Ziele vorher definiert werden?

Frage

Sind Visionen und Ziele beschrieben, dann können funktionale und nichtfunktionale Anforderungen sowie die Rahmenbedingungen immer gegen die Visionen und Ziele abgeglichen werden. Es kann dann immer gefragt werden: »Ist diese Anforderung zielführend, d. h., trägt sie dazu bei, das Ziel zu erreichen?«

Antwort

Vision

Eine **Vision** ist eine realitätsnahe Vorstellung der gewünschten Zukunft. Sie beschreibt, was erreicht werden soll, sagt aber nicht wie.

- Die Brüder Albrecht hatten folgende Vision: »Wir verkaufen qualitativ hochwertige Produkte möglichst preiswert.« Die Umsetzung dieser Vision führte zu den ALDI-Läden (*Albrecht Discount*).
- Artur Fischer, der Erfinder des Dübel, hatte die Vision: »Ein Fotograf kann auch bei Nacht fotografieren.« Diese Vision brachte ihn dazu, das Blitzlicht zu erfinden.
- Die Paketverteilfirma Federal Express formulierte in ihrer Anfangszeit folgende Vision: »Wir liefern das Paket am nächsten Morgen bis 10.30 Uhr aus«.

Beispiele 1

Bezogen auf die Softwaretechnik sollte eine Vision als Leitgedanke für alle *Stakeholder* dienen.

/V10/ Die Firma Teachware soll durch das System in die Lage versetzt werden, die von ihr veranstalteten Seminare sowie Kunden und Dozenten effizient rechnerunterstützt zu verwalten.

Beispiel:
SemOrg

/V20/ Die Kunden der Firma Teachware sollen über das Web möglichst viele Vorgänge selbst durchführen können.

/V20/ Die Fensterheber-Komponente soll das komfortable Heben und Senken der Seitenfenster des Fahrzeugs ermöglichen.

Beispiel:
Fensterheber

Ziele

Ausgehend von einer Vision dienen **Ziele** dazu, die Vision zu verfeinern und zu operationalisieren.

/Z10/ Ein Interessent oder ein Kunde kann mindestens 20 Stunden jeden Tag Seminare und Veranstaltungen über das Web selektieren und eine Veranstaltung online buchen, damit die Mitarbeiter der Fa. Teachware von solchen Tätigkeiten entlastet werden.

Beispiel:
SemOrg

In [Pohl07, S. 100 ff.] werden sieben Regeln zur Formulierung von Zielen aufgeführt:

■ Regel 1: Ziele kurz und prägnant formulieren

Füllwörter und Allgemeinplätze vermeiden.

IV 16 Anforderungen und Anforderungsarten

Beispiel: Falsch: Es wäre gut, wenn zu Seminarveranstaltungen vielfältige
SemOrg Auswertungen möglich wären.

Besser: Das System soll dem Benutzer die Möglichkeit bieten, Seminarveranstaltungen nach den vorhandenen Attributen zu selektieren und zu sortieren.

■ Regel 2: Aktivformulierungen verwenden

Den Akteur klar benennen.

Beispiel »Federal Express liefert das Paket aus« (siehe Beispiele 1).

■ Regel 3: Überprüfbare Ziele formulieren

Das Zielerfüllung muss im späteren System überprüfbar sein.

Beispiel »Federal Express liefert bis 10.30 Uhr aus« (siehe Beispiele 1).

■ Regel 4: Ziele, die nicht überprüft werden können, nicht verfeinern

Ziel in überprüfbare Teilziele aufgliedern.

Beispiel »Wir können jederzeit feststellen, wo sich ein Paket zurzeit befindet.«

Besser: »Wir können zwischen 7.00 Uhr und 20.00 Uhr innerhalb von maximal 5 Minuten feststellen, wo sich ein Paket zurzeit befindet.«

»Wir können zwischen 20.00 Uhr und 7.00 Uhr innerhalb von maximal 10 Minuten feststellen, wo sich ein Paket zurzeit befindet.«

■ Regel 5: Den Mehrwert eines Ziels hervorheben

Möglichst genau beschreiben, welchen Mehrwert das Ziel bringt.

Beispiel Die Erstellungszeit von Rechnungen soll von 5 auf 2 Arbeitstage verkürzt werden.

■ Regel 6: Das Ziel sollte begründet werden

Eine Zielbegründung führt zur Identifikation weiterer Ziele.

Beispiel Als Webanwendung kann das System auch durch externe Benutzer benutzt werden.

■ Regel 7: Keine Lösungsansätze angeben

Lösungsansätze schränken den Lösungsraum zu früh ein, daher den maximalen Lösungsraum offen lassen.

Beispiel: Falsch: Die Software »Seminarorganisation« muss mit der Software
SemOrg »Buchhaltung« permanent über eine Netzverbindung verbunden sein.

Richtig: Die Software »Seminarorganisation« muss mit der Software »Buchhaltung« in der Regel innerhalb von 24 Stunden einmal Daten austauschen können, mindestens aber einmal innerhalb von 5 Arbeitstagen. Begründung: Aus fachlicher Sicht reicht diese Festlegung, da nach einer Seminaranmeldung nicht sofort eine Mitteilung

an die Buchhaltung erfolgen muss. Außerdem ist eine Festlegung auf eine Netzverbindung nicht nötig. Es könnte ja auch sein, dass beide Programme auf einem Computersystem laufen.

In [Rupp07, S. 100 f.] werden zwei weitere Regeln aufgeführt:

■ Regel 8: Einschränkung der Rahmenbedingungen aufführen

Rahmenbedingungen, die den Lösungsraum einschränken, müssen aufgeführt werden, damit die spätere Lösung nicht außerhalb des Lösungsraums liegt.

Wenn der Kollisionssensor anschlägt, muss das System in weniger als 20 Millisekunden den Airbag auslösen. Beispiel

■ Regel 9: Realistische Ziele formulieren

Unrealistische, nicht erreichbare Ziele diskreditieren die gesamten Anforderungen und werden nicht ernst genommen.

Die Spracheingabe muss alle deutschen Dialekte ohne Trainingsphase 100%ig erkennen. Beispiel

In [NuEa07] werden die Probleme beschrieben, die durch den inkonsistenten Gebrauch der Terminologie auftreten, wenn *Stakeholder* zielorientierte Anforderungen formulieren. Es wird zwischen harten Zielen und weichen Zielen unterschieden. Es wird eine Methode vorgestellt, um diese Probleme zu lösen. Probleme

Weiterführende Konzepte zu Visionen und Zielen finden Sie in [Pohl07, S. 89 ff.] Literatur

16.2 Rahmenbedingungen

Eine **Rahmenbedingung** (*constraint*) – auch Restriktion genannt – legt organisatorische und/oder technische Restriktionen für das Softwaresystem und/oder den Entwicklungsprozess fest.

Definition

Zu den **organisatorischen Rahmenbedingungen** gehören:

- **Anwendungsbereiche:** Legt fest, für welche Anwendungsbereiche das System vorgesehen ist, z. B. Textverarbeitung im Büro.
- **Zielgruppen:** Legt fest, für welche Zielgruppen das System vorgesehen ist, z. B. Sekretärinnen, Schreibkräfte. Diese Rahmenbedingung ist wichtig für die Gestaltung der Benutzungsoberfläche. Unter Umständen sollte auch festgelegt werden, von welchen Voraussetzungen, z. B. bezüglich des Qualifikationsniveaus des Benutzers, ausgegangen wird. Ebenfalls kann es sinnvoll sein, explizit anzugeben, für welche Anwendungsbereiche und Zielgruppen das Produkt *nicht* vorgesehen ist, z. B. für den IT-unkundigen Benutzer.

IV 16 Anforderungen und Anforderungsarten

■ **Betriebsbedingungen:** Folgende Punkte werden beschrieben:

- Physikalische Umgebung des Systems, z. B. Büroumgebung, Produktionsanlage oder mobiler Einsatz.
- Tägliche Betriebszeit, z. B. Dauerbetrieb bei Telekommunikationsanlagen.
- Ständige Beobachtung des Systems durch Bediener oder unbeaufsichtigter Betrieb.

Deckt das System verschiedene Anwendungsbereiche und Zielgruppen ab, dann ist eine Auflistung der unterschiedlichen Bedürfnisse und Anforderungen nötig.

Beispiel:
SemOrg

○ **Anwendungsbereich:** Kaufmännisch/administrativer Anwendungsbereich

○ **Zielgruppen:**

Mitarbeiter der Firma Teachware lassen sich gliedern in:

Kundensachbearbeiter, Seminarsachbearbeiter, Veranstaltungsbe-
treuer.

Kunden der Firma Teachware:

Kunden und Firmen können sich über das Internet über Seminare
und Veranstaltungen informieren und selbst Buchungen durch-
führen.

○ **Betriebsbedingungen:** Büroumgebung

Zu den **technischen Rahmenbedingungen** gehören:

■ Technische Produktumgebung

■ Anforderungen an die Entwicklungsumgebung

Bei der **technischen Produktumgebung** sind folgende Festlegun-
gen zu treffen:

- **Software:** Welche Softwaresysteme (Betriebssystem, Laufzeitsys-
tem, Datenbank, Fenstersystem usw.) sollen auf der Zielmaschine
(Maschine, auf der das fertiggestellte System eingesetzt werden
soll) zur Verfügung stehen, z. B. Web-Browser auf dem Client.
- **Hardware:** Welche Hardware-Komponenten (CPU, Peripherie, z. B.
Grafikbildschirm, Drucker) sind in minimaler und maximaler Kon-
figuration für den Systemeinsatz vorgesehen.
- **Orgware:** Unter welchen organisatorischen Randbedingungen
bzw. Voraussetzungen soll das Produkt eingesetzt werden (z. B.
»Aktuelle Wetterdaten sind nur dann einbindbar, wenn das Sys-
tem über einen permanenten Internet-Zugriff verfügt«).

Bei Client/Server- oder Web-Anwendungen ist die Umgebung jeweils
für Clients und Server getrennt anzugeben.

Bei den **Anforderungen an die Entwicklungsumgebung** sind
folgende Festlegungen zu treffen:

- Software
- Hardware
- Orgware

□ Entwicklungsschnittstellen

Es wird die Entwicklungsumgebung des Systems beschrieben. Es wird festgelegt, welche Konfiguration bzgl. Software, Hardware und Orgware für die Entwicklung des Systems benötigt wird. Diese Festlegungen sind insbesondere dann notwendig, wenn Entwicklungs- und Zielmaschine unterschiedlich sind. Bei Entwicklungsschnittstellen ist unter Umständen aufzuführen, über welche einzuhaltenden Hardware- und Softwareschnittstellen Entwicklungs- und Zielrechner gekoppelt sind. Unter Software ist insbesondere aufzuführen, welche Software-Werkzeuge, z. B. integrierte Entwicklungsumgebungen, Programmierungsumgebungen, Compiler usw., benötigt werden.

■ **Technische Produktumgebung:** Das System ist eine Web-Anwendung.

Beispiel:
SemOrg

□ **Software:** Server-Betriebssystem: Windows. Client: Web-Browser (Es werden die drei marktführenden Browser unterstützt).

□ **Hardware:** Server: PC, Client: Browserfähiges Gerät mit Grafikbildschirm

□ **Orgware:** Zugriff des Servers auf die Buchhaltungssoftware.

■ **Anforderungen an die Entwicklungsumgebung:** Keine Abweichungen von der Einsatzumgebung.

Rahmenbedingungen können

- keine der Anforderungen einschränken,
- mögliche Realisierungen von Anforderungen einschränken,
- zur Änderung von Anforderungen führen,
- zu neuen Anforderungen führen oder
- zu nicht realisierbaren Anforderungen führen.

16.3 Kontext und Überblick

Jedes Softwaresystem ist in eine materielle und immaterielle Umgebung eingebettet – mehr oder weniger stark in Abhängigkeit von der Art der Software.

Materielle Umgebung: Sensoren, Gebäude, Personen, andere technische Systeme, physikalische Kanäle und Übertragungsmedien.

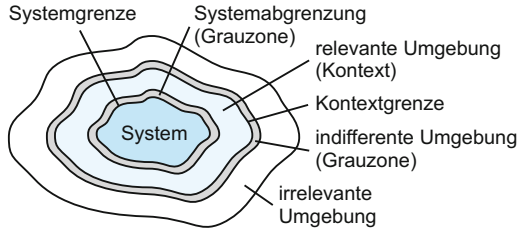
Beispiele

Immaterielle Umgebung: Schnittstellen zu anderen Softwaresystemen, Internet.

Die Systemumgebung hat einen maßgeblichen Einfluss auf die Anforderungen. Daher ist es wichtig, die Systemumgebung festzulegen. Die Abb. 16.3-1 zeigt, wie ein System und seine Umgebung abgegrenzt werden können (in Anlehnung an [Pohl07, S. 56 ff.]).

IV 16 Anforderungen und Anforderungsarten

Abb. 16.3-1: Das System und seine Umgebung.



Systemgrenze Das zu entwickelnde System besitzt eine **Systemgrenze**, die es von den Teilen der Umgebung abgrenzt, die durch die Entwicklung *nicht* verändert werden können. Liegt bereits fest, was *nicht* zum System gehört, dann sollte dies explizit aufgeführt werden.

Quellen und Senken interagieren mit dem System. Quellen liefern Eingaben, Senken erhalten Ausgaben. Beispiele für Quellen und Senken sind Personen, andere Systeme, Sensoren und Aktoren. Die Interaktion erfolgt i. Allg. über Benutzungsschnittstellen und Softwareschnittstellen.

Da am Anfang des *Requirements Engineering* die Systemgrenze noch nicht genau festliegt, wird sie von einer Grauzone umgeben.

Beispiel: Das System SemOrg soll mit der Buchhaltungssoftware Daten austauschen. Da zum Zeitpunkt der Anforderungserstellung das Buchhaltungssystem noch nicht feststeht, kann auch die Schnittstelle noch nicht fertig spezifiziert werden.

Kontext Um das System herum gibt es eine relevante Umgebung – **Kontext** genannt –, die für die Systementwicklung zu beachten ist, und eine irrelevante Umgebung, die keinen Einfluss auf die Entwicklung hat. Zwischen beiden gibt es ebenfalls eine Grauzone.

Der Kontext beeinflusst die Interpretation einer Anforderung, daher ist die Festlegung des Kontextes so wichtig.

Beispiel Die Reaktion des Systems auf Benutzereingaben muss in weniger als 2 Sekunden erfolgen. Legt der Kontext fest, dass die Anwendung auf einem Handy läuft und die Internet-Verbindung über das Handy erfolgt, dann hat die Anforderung andere Konsequenzen, als wenn die Anwendung auf einem Notebook oder PC läuft und über einen DSL-Anschluss verfügt.

Basieren Anforderungen auf Annahmen des Systemkontextes, dann sollten diese explizit gekennzeichnet werden.

Notation Neben der natürlichsprachlichen Beschreibung des Kontextes eignen sich auch folgende Basiskonzepte zur Beschreibung:

- UML-Use-Case-Diagramm (siehe »Use Case-Diagramme und -Schablonen«, S. 255)
- UML-Klassendiagramm (siehe »Zusammenfassung von Funktionen«, S. 131)
- UML-Sequenzdiagramm (siehe »Sequenzdiagramm«, S. 333)

16.4 Nichtfunktionale Anforderungen IV

■ UML-Verteilungsdiagramm (für physikalische Kanäle)

Den Kontext zur Seminarorganisation wird in der Abb. 16.3-2 durch ein *Use Case-Diagramm* dargestellt – beispielsweise im Lastenheft. Eine Verfeinerung – beispielsweise im Pflichtenheft – zeigt die Abb. 16.3-3.

Beispiel:
SemOrg

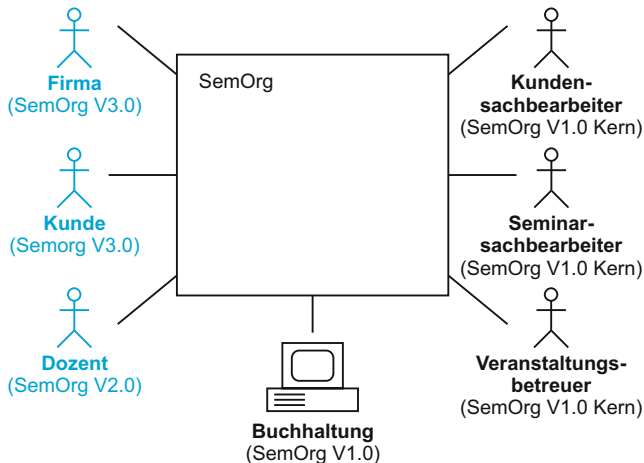


Abb. 16.3-2:
Umwelt des
Produkts SemOrg
(Umweltdia-
gramm) als UML-
Use-Case-
Diagramm.

Weiterführend wird dieses Thema in [Pohl07, S. 55 ff.] behandelt.

Literatur

16.4 Nichtfunktionale Anforderungen

Unter dem Begriff nichtfunktionale Anforderungen wird oft alles subsumiert, was nicht zu den funktionalen Anforderungen gehört.

Was verstehen Sie unter nichtfunktionalen Anforderungen?

Frage

Die Auffassungen, was nichtfunktionale Anforderungen sind, differieren.

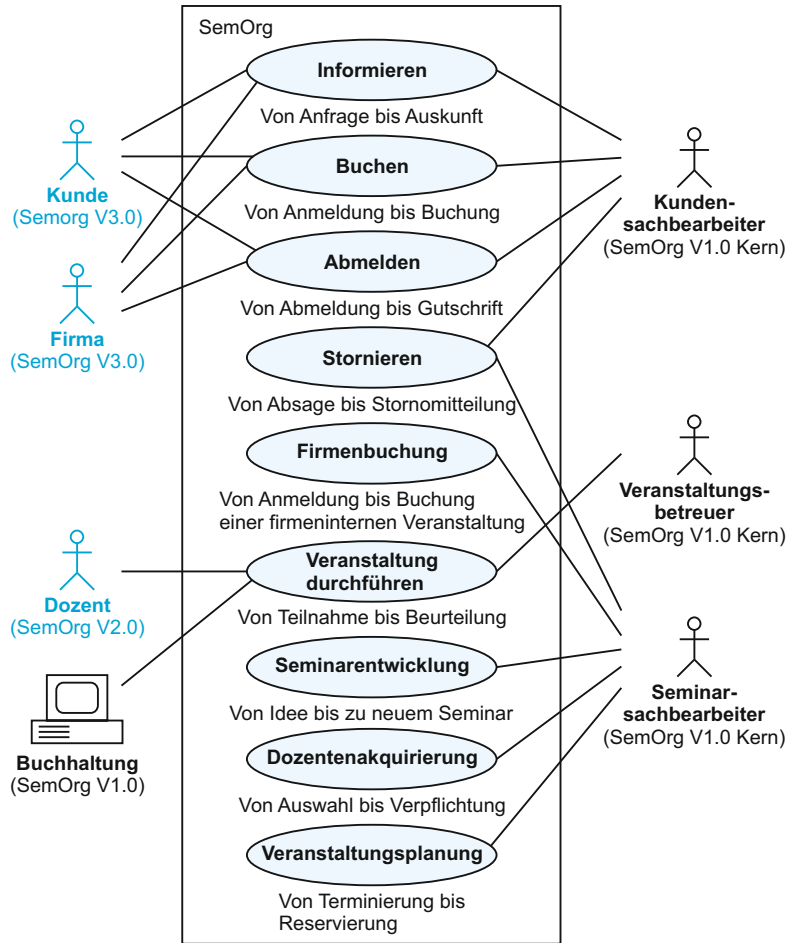
Antwort

Nichtfunktionale Anforderungen (*nonfunctional requirements*, kurz NFRs), auch Technische Anforderungen oder *Quality of Service* (QoS) genannt, beschreiben Aspekte, die typischerweise mehrere oder alle funktionale Anforderungen betreffen bzw. überschneiden (*cross-cut*). Zu den NFRs zählen u. a. Genauigkeit, Verfügbarkeit, Nebenläufigkeit, Konsumierbarkeit (eine Obermenge der Benutzbarkeit), Internationalisierung, Betriebseigenschaften, Zuverlässigkeit, Sicherheit, Service-Anforderungen, Support (siehe auch [Ambl08, S. 64]). Nichtfunktionale Anforderungen haben großen Einfluss auf die Softwarearchitektur.

Außerdem beeinflussen sich nichtfunktionale Anforderungen. Sicherheit (*security*) steht häufig im Konflikt mit Benutzbarkeit, Speichereffizienz ist meist gegenläufig zur Laufzeiteffizienz.

IV 16 Anforderungen und Anforderungsarten

Abb. 16.3-3: UML-
Use Cases des
Produkts SemOrg
(Übersichtsdiagramm).



In [Pohl07, S. 16 ff.] wird darauf hingewiesen, dass nichtfunktionale Anforderungen entweder Qualitätsanforderungen oder unter-spezifizierte funktionale Anforderungen sind.

Definition

Eine **Qualitätsanforderung** legt eine qualitative und/oder quantitative Eigenschaft des Softwaresystems oder einer seiner Komponenten fest.

Qualitätsanforderungen legen fest, welche Qualitätsmerkmale für das Softwaresystem als relevant erachtet werden und in welcher Qualitätsstufe sie erreicht werden sollen. Qualitätsanforderungen haben in der Regel Auswirkungen auf den Softwareentwurf, da sie aufgrund ihrer systemübergreifenden Eigenschaft die Struktur einer Architektur wesentlich beeinflussen.

16.4 Nichtfunktionale Anforderungen IV

Unterspezifizierte Anforderungen lassen sich bei entsprechender Detaillierung in funktionale Anforderungen und ggf. Qualitätsanforderungen überführen.

Allgemein legen funktionale Anforderungen fest, »was« ein System tut bzw. tun soll, während die Qualitätsanforderungen beschreiben, »wie gut« die Funktionen erledigt werden (sollen). In [Glin08] wird folgende Klassifikation der Anforderungen vorgeschlagen:

■ **Funktionale Anforderungen:** Funktionalität und Verhalten

- ☐ Funktionen
- ☐ Daten
- ☐ Stimuli
- ☐ Reaktionen
- ☐ Verhalten

■ **Leistungsanforderungen:** Zeit- und Speichergrenzen

- ☐ Zeit
- ☐ Geschwindigkeit
- ☐ Umfang
- ☐ Durchsatz

■ **Spezifische Qualitätsanforderungen:** »-keiten«

- ☐ Zuverlässigkeit
- ☐ Benutzbarkeit
- ☐ Sicherheit
- ☐ Verfügbarkeit
- ☐ Portabilität
- ☐ Wartbarkeit usw.

■ **Randbedingungen/Einschränkungen**

- ☐ Physikalisch
- ☐ Rechtlich
- ☐ Kulturell
- ☐ Umgebung
- ☐ Entwurf und Implementierung
- ☐ Schnittstellen usw.

In der internationalen Norm ISO/IEC 9126-1:2001 »*Software engineering – Product quality – Part 1: Quality model*« wird ein Qualitätsmodell beschrieben, das aus sechs Qualitätsmerkmalen (*characteristics*) besteht, die wiederum in insgesamt 26 Teilmerkmale (*subcharacteristics*) untergliedert sind. Den Merkmalen bzw. Teilmerkmalen sind Qualitätsattribute (*quality attributes*) zugeordnet. Alle Qualitätsmerkmale sind auf jede Art von Software anwendbar.

ISO/IEC 9126-1

Die sechs Qualitätsmerkmale sind:

6 Q-Merkmale

■ **Funktionalität** (*functionality*)

■ **Zuverlässigkeit** (*reliability*)

■ **Benutzbarkeit** (*usability*)

■ **Effizienz** (*efficiency*)

■ **Wartbarkeit** (*maintainability*)

IV 16 Anforderungen und Anforderungsarten

■ Portabilität (*portability*)

Die Definition dieser Qualitätsmerkmale sowie aller Teilmerkmale sind im Kapitel »Box: Qualitätsmerkmale nach ISO/IEC 9126-1«, S. 468, aufgeführt. Anhand dieser Qualitätsmerkmale können die nichtfunktionalen Anforderungen gegliedert aufgeführt werden. Den Qualitätsmerkmalen kann man Qualitätsstufen zuordnen, z. B.:

- Sehr gute Systemqualität
- Gute Systemqualität
- Normale Systemqualität

Beispiel 1a: Im Lastenheft wird die in der Tab. 16.4-1 angegebene Qualitätszielbestimmung vorgenommen.
SemOrg

Tab. 16.4-1:
Qualitätszielbestimmung für
SemOrg im
Lastenheft.

Systemqualität	sehr gut	gut	normal	nicht relevant
Funktionalität		X		
Zuverlässigkeit			X	
Benutzbarkeit		X		
Effizienz			X	
Wartbarkeit			X	
Portabilität			X	

In einem Pflichtenheft kann eine Qualitätszielbestimmung für alle 26 Teilmerkmale vorgenommen werden.

Beispiel 1b: Im Pflichtenheft wird eine detaillierte Qualitätszielbestimmung vorgenommen, siehe »Fallstudie: SemOrg – Die Spezifikation«, S. 107.
SemOrg

Spezifikation von nichtfunktionalen Anforderungen

Nichtfunktionale Anforderungen werden in der Regel in natürlicher Sprache formuliert (siehe auch »Natürlichsprachliche Anforderungen«, S. 481).

Beispiel 1c: /QEZ10/ Alle Reaktionszeiten auf Benutzeraktionen müssen unter 5 Sekunden liegen (EZ steht für Effizienz / Zeitverhalten).
SemOrg

In der UML können nichtfunktionale Anforderungen – über das Modell verstreut – in Form von Zusicherungen an Modellelemente spezifiziert werden (siehe auch »Constraints und die OCL in der UML«, S. 377). Diese Art und Weise, nichtfunktionale Anforderungen zu spezifizieren, ist aber unübersichtlich. Eine Alternative dazu bietet die Sprache **SysML** (siehe »Basiskonzepte«, S. 99). In ihr gibt es das Modellelement *Requirement* und die neue Diagrammart *Requirement Diagram*. Dieses Modellelement wird als stereotypisierte UML-Klasse mit dem Stereotyp «requirement» und den Stereotypattributen *id* und *text* dargestellt. *id* legt eine eindeutige Kennung fest,

16.4 Nichtfunktionale Anforderungen IV

text einen beschreibenden Text, der auch Verweise auf Quellen außerhalb des UML-Modells enthalten kann. Damit können funktionale und nichtfunktionale Anforderungen spezifiziert werden.

Da es sich bei dem Modellelement *Requirement* um eine Klasse handelt, sind auch alle Klassenbeziehungen erlaubt. Folgende Abhängigkeitsbeziehungen (gestrichelter Pfeil in der UML und in SysML) können modelliert werden:

- «trace»: Die Anforderung A bezieht sich auf die Anforderung B. Sie kann benutzt werden, um den Zusammenhang zwischen funktionalen und nichtfunktionalen Anforderungen zu modellieren.
- «deriveReq»: Die Anforderung A ist von der Anforderung B abgeleitet. «deriveReq» ist ein Stereotyp der UML-Abstraktionsbeziehung.
- «verify»: Das Testelement A überprüft die Anforderung B. «verify» ist eine Spezialisierung des UML-Stereotyps «trace».

Die Abb. 16.4-1 zeigt die grafische Spezifikation von nichtfunktionalen Anforderungen.

Beispiel 1d:
SemOrg

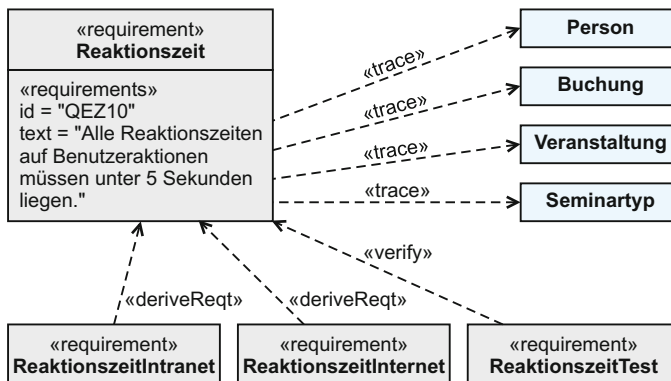


Abb. 16.4-1:
Beispiel für die
Spezifizierung
nichtfunktionaler
Anforderungen in
der UML 2 und der
SysML.

Neben speziellen SysML-Werkzeugen können diese Modellelemente mit jedem UML-Werkzeug, das Stereotypen mit Eigenschaften (Stereotypattribute) unterstützt, umgesetzt werden.

Werkzeuge

[Weil06], [FMS08], [JaMa02], [OBN+08].

Weiterführende
Literatur

16.5 Box: Qualitätsmerkmale nach ISO/IEC 9126-1

Funktionalität (*functionality*)

Fähigkeit des Softwareprodukts, Funktionen bereitzustellen, damit die Software unter den spezifizierten Bedingungen die festgelegten Bedürfnisse erfüllt.

- **Angemessenheit (*suitability*)**

Fähigkeit des Softwareprodukts, für spezifizierte Aufgaben und Benutzerziele geeignete Funktionen bereitzustellen.

- **Genauigkeit (*accuracy*)**

Fähigkeit des Softwareprodukts, die richtigen oder vereinbarten Ergebnisse oder Wirkungen mit der benötigten Genauigkeit bereitzustellen.

- **Interoperabilität (*interoperability*)**

Fähigkeit des Softwareprodukts, mit einem oder mehreren vorgegebenen Systemen zusammenzuwirken.

- **Sicherheit (*security*)**

Fähigkeit des Softwareprodukts, Informationen und Daten so zu schützen, dass nicht autorisierte Personen oder Systeme sie nicht lesen oder verändern können, und autorisierten Personen oder Systemen der Zugriff nicht verweigert wird.

- **Konformität der Funktionalität (*functionality compliance*)**

Fähigkeit des Softwareprodukts, Standards, Konventionen oder gesetzliche Bestimmungen und ähnliche Vorschriften bezogen auf die Funktionalität einzuhalten.

Zuverlässigkeit (*reliability*)

Fähigkeit des Softwareprodukts, ein spezifiziertes Leistungsniveau zu bewahren, wenn es unter festgelegten Bedingungen benutzt wird.

- **Reife (*maturity*)**

Fähigkeit des Softwareprodukts, trotz Fehlzuständen in der Software nicht zu versagen.

- **Fehlertoleranz (*fault tolerance*)**

Fähigkeit des Softwareprodukts, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren.

- **Wiederherstellbarkeit (*recoverability*)**

Fähigkeit des Softwareprodukts, bei einem Versagen ein spezifiziertes Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen.

- **Konformität der Zuverlässigkeit (*reliability compliance*)**

Fähigkeit des Softwareprodukts, Standards, Konventionen oder Vorschriften bezogen auf die Zuverlässigkeit einzuhalten.

Benutzbarkeit (*usability*)

Fähigkeit des Softwareprodukts, vom Benutzer verstanden und benutzt zu werden sowie für den Benutzer erlernbar und »attraktiv« zu sein, wenn es unter den festgelegten Bedingungen benutzt wird.

- **Verständlichkeit (*understandability*)**

Fähigkeit des Softwareprodukts, den Benutzer zu befähigen, zu prüfen, ob die Software angemessen ist und wie sie für bestimmte Aufgaben und Nutzungsbedingungen verwendet werden kann.

- **Erlernbarkeit (*learnability*)**

Fähigkeit des Softwareprodukts, den Benutzer zu befähigen, die Anwendung zu erlernen.

- **Bedienbarkeit (*operability*)**

Fähigkeit des Softwareprodukts, den Benutzer zu befähigen, die Anwendung zu bedienen und zu steuern.

- **Attraktivität (*attractiveness*)**

Fähigkeit des Softwareprodukts für den Benutzer attraktiv zu sein, z. B. durch die Verwendung von Farbe oder die Art des grafischen Designs.

- **Konformität der Benutzbarkeit (*usability compliance*)**

Fähigkeit des Softwareprodukts, Standards, Konventionen, Stilvorgaben (*style guides*) oder Vorschriften bezogen auf die Benutzbarkeit einzuhalten.

Effizienz (*efficiency*)

Fähigkeit des Softwareprodukts, ein angemessenes Leistungsniveau bezogen auf die eingesetzten Ressourcen unter festgelegten Bedingungen bereitzustellen.

- **Zeitverhalten (*time behaviour*)**

Fähigkeit des Softwareprodukts, angemessene Antwort- und Bearbeitungszeiten sowie Durchsatz bei der Funktionsausführung unter festgelegten Bedingungen sicherzustellen.

- **Verbrauchsverhalten (*resource utilisation*)**

Fähigkeit des Softwareprodukts, eine angemessene Anzahl und angemessene Typen von Ressourcen zu verwenden, wenn die Software ihre Funktionen unter festgelegten Bedingungen ausführt.

- **Konformität der Effizienz (*efficiency compliance*)**

Fähigkeit des Softwareprodukts, Standards oder Konventionen bezogen auf die Effizienz einzuhalten.

IV 16 Anforderungen und Anforderungsarten

Wartbarkeit (*maintainability*)

Fähigkeit des Softwareprodukts änderungsfähig zu sein. Änderungen können Korrekturen, Verbesserungen oder Anpassungen der Software an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen.

- **Analysierbarkeit** (*analyzability*)

Fähigkeit des Softwareprodukts, Mängel oder Ursachen von Versagen zu diagnostizieren oder änderungsbedürftige Teile zu identifizieren.

- **Änderbarkeit** (*changeability*)

Fähigkeit des Softwareprodukts, die Implementierung einer spezifizierten Änderung zu ermöglichen.

- **Stabilität** (*stability*)

Fähigkeit des Softwareprodukts, unerwartete Wirkungen von Änderungen der Software zu vermeiden.

- **Testbarkeit** (*testability*)

Fähigkeit des Softwareprodukts, die modifizierte Software zu validieren.

- **Konformität der Wartbarkeit** (*maintainability compliance*)

Fähigkeit des Softwareprodukts, Standards oder Konventionen bezogen auf die Wartbarkeit einzuhalten.

Portabilität (*portability*)

Fähigkeit des Softwareprodukts, von einer Umgebung in eine andere übertragen zu werden. Umgebung kann organisatorische Umgebung, Hardware- oder Software-Umgebung einschließen.

- **Anpassbarkeit** (*adaptability*)

Fähigkeit des Softwareprodukts, die Software an verschiedene, festgelegte Umgebungen anzupassen, wobei nur Aktionen oder Mittel eingesetzt werden, die für diesen Zweck für die betrachtete Software vorgesehen sind.

- **Installierbarkeit** (*installability*)

Fähigkeit des Softwareprodukts, in einer festgelegten Umgebung installiert zu werden.

- **Koexistenz** (*co-existence*)

Fähigkeit des Softwareprodukts, mit anderen unabhängigen Softwareprodukten in einer gemeinsamen Umgebung gemeinsame Ressourcen zu teilen.

- **Austauschbarkeit** (*replaceability*)

Fähigkeit des Softwareprodukts, diese Software anstelle einer spezifizierten anderen in der Umgebung jener Software für denselben Zweck zu verwenden.

■ **Konformität der Portabilität** (*portability compliance*)

Fähigkeit des Softwareprodukts, Standards oder Konventionen bezogen auf die Portabilität einzuhalten.

16.6 Abnahmekriterien

Welche Vorteile bringt es mit sich, wenn mit der Festlegung der Anforderungen bereits die Festlegung von Abnahmekriterien für diese Anforderungen verbunden wird?

Frage

Das bringt folgende Vorteile mit sich:

Antwort

- + Es ist von vornherein klar, wie überprüft werden kann, ob die Anforderungen korrekt realisiert wurden.
- + Es wird sich bei der Abnahme nicht nur auf das realisierte System gestützt.
- + Bereits bei der Formulierung der Anforderungen wird darauf geachtet, dass sie auch überprüft werden können, was nach aller Erfahrung zu einer operationalisierten Festlegung und qualitativen Verbesserung der Anforderungen führt.
- + Die Formulierung von Abnahmekriterien führt zu besserer Veranschaulichung und Verständlichkeit der meist abstrakt formulierten Anforderungen.

Abnahmekriterien lassen sich nach ihrem **Abstraktionsgrad** klassifizieren:

Klassifikation

- Abstrakte Abnahmekriterien enthalten keine konkreten Werte.
- Konkrete Abnahmekriterien enthalten konkrete Werte, die z. B. für einen Abnahmetest verwendet werden können.

Weiterhin lassen sich die Abnahmekriterien nach ihrer **Reichweite** klassifizieren:

- Abnahmekriterien gelten für genau eine Anforderung.
- Abnahmekriterien gelten für mehrere Anforderungen.
- Abnahmekriterien gelten nur für Teile einer Anforderung (Regelfall).

Analog, wie es Anforderungen an Anforderungen gibt, so gibt es auch (Qualitäts-)Anforderungen an Abnahmekriterien (vgl. [Rupp07, S. 326 f.]):

Anforderungen

- Abnahmekriterien müssen *wirtschaftlich überprüfbar* sein, d. h. eine Prüfung, ob die Anforderung realisiert ist, muss mit vertretbarem Aufwand möglich sein.
- Abnahmekriterien müssen auf Korrektheit überprüfbar sein, d. h. eine Prüfung, ob die Anforderung *korrekt realisiert* ist, muss möglich sein.
- Abnahmekriterien müssen so formuliert sein, dass sie für **Regressionstests** verwendbar sind.

IV 16 Anforderungen und Anforderungsarten

- Abnahmekriterien dürfen nicht mehr aber auch nicht weniger festlegen, als in der jeweiligen Anforderung gefordert wird. Es dürfen in einem Abnahmekriterium *keine* zusätzlichen Eigenschaften oder Leistungen versteckt sein, die in der Anforderung selbst nicht aufgeführt sind. Umgekehrt dürfen Details, die in Anforderungen stehen, bei Abnahmekriterien nicht weggelassen werden.
- Abnahmekriterien sollen *minimal*, aber *vollständig* sein, d. h. sie sollen alle Anforderungen überdecken, aber aus Wirtschaftlichkeitsgründen nicht mehrfach.

Notation Abnahmekriterien lassen sich natürlichsprachlich, semiformal und formal beschreiben.

Natürlichsprachliche Abnahmekriterien können in drei Teile gegliedert werden:

- Ausgangssituation: Zustand des Prüflings vor der Abnahmeprüfung
- Ereignis: Welches Ereignis löst die Prüfung aus?
- Erwartetes Ergebnis: Soll-Zustand bei korrektem Verhalten.

Beispiel: **Anforderung:**

SemOrg

/F12/ Wenn ein Kunde oder eine Firma sich von einer bereits gebuchten Veranstaltung später als X Wochen vor der Veranstaltung abmeldet, dann muss das System Stornogebühren in Höhe der Veranstaltungsgebühr berechnen oder nach einem Ersatzteilnehmer fragen.

Abnahmekriterium (abstrakt):

Ausgangssituation: Ein Kunde, eine Firma, ein Seminar und eine Veranstaltung sind angelegt. Der Kunde und die Firma haben die Veranstaltung gebucht.

Ereignis: Der Kunde und die Firma stornieren die Veranstaltung X Wochen vor Veranstaltungsbeginn (wobei festgelegt sein muss, ab wann Stornogebühren in Höhe der Veranstaltungsgebühr anfallen, X muss in diesen Zeitraum fallen).

Erwartetes Ergebnis: Das System fragt, ob der Kunde und die Firma einen Ersatzteilnehmer stellen (Name usw. wird angefordert). Wenn nein, dann wird mitgeteilt, dass der volle Veranstaltungsbetrag fällig wird. Die Stornierung wird bestätigt.



Formulieren Sie das Abnahmekriterium des Beispiels in konkreter Form.

Sind Abnahmekriterien komplex, dann ergeben sich daraus oft viele natürlichsprachliche Abnahmekriterien.

Bei der Festlegung von Anforderungen wird oft nur der Standardfall beschrieben. Sonderfälle werden nicht spezifiziert und Fehlerfälle oft nicht betrachtet. Das hat zur Folge, dass bei der Realisierung des Systems die Entwickler oft nicht wissen, wie sie diese Fälle realisieren sollen.

Als Alternativen für die Beschreibung von Abnahmekriterien bieten sich folgende Basiskonzepte an:

■ »Entscheidungstabellen und Entscheidungsbäume«, S. 386

■ »Geschäftsprozesse und *Use Cases*«, S. 250

Für die möglichst umfassende, aber redundanzarme Prüfung der spezifizierten Funktionalität bieten sich **funktionale Testverfahren** (Black-Box-Tests) an, bei denen das zu testende System ein »schwarzer Kasten« ist, d. h. die interne Programmstruktur ist dem Tester nicht bekannt. Ein vollständiger Funktionstest ist i. Allg. nicht durchführbar. Ziel bei der Ermittlung der Abnahmekriterien muss es daher sein, Testfälle so auszuwählen, dass die Wahrscheinlichkeit groß ist, Fehler zu finden. Für die Testfallbestimmung gibt es folgende wichtige Verfahren:

■ **Funktionale Äquivalenzklassenbildung**,

■ **Grenzwertanalyse**,

■ **Test spezieller Werte**,

■ Zustandsbasierter Test,

■ Ursache-Wirkungs-Analyse.

Auch – oder gerade – für nichtfunktionale Anforderungen sind Abnahmekriterien zu formulieren. Oft werden nichtfunktionale Anforderungen nur vage beschrieben. Die Formulierung von Abnahmekriterien hilft, die Anforderung zu operationalisieren.

Qualitätsanforderung:

/QEZ10/ Alle Reaktionszeiten auf Benutzeraktionen müssen unter 5 Sekunden liegen.

Beispiel:
SemOrg

Abnahmekriterium:

Beim Aufruf der Funktionen /F10/, /F11/, /F12/, /F20/, /F30/, /F40/, /F50/, /F60/, /F61/ (siehe »Fallstudie: SemOrg – Die Spezifikation«, S. 107) muss die Ausgabe des Ergebnisses innerhalb von 5 Sekunden erfolgen.

Die Prüfung von Geschäftsabläufen und *Use Cases* erfordern oft das Anwenden mehrerer Abnahmekriterien in einer bestimmten Reihenfolge: Ein **Abnahmeszenario** ist eine chronologisch geordnete Sammlung von Abnahmekriterien, das für eine Abnahme verwendet wird.

Ein Seminar neu erfassen, eine Veranstaltung neu erfassen, die Veranstaltung dem Seminar zuordnen, einen Dozenten erfassen und dem Seminar und der Veranstaltung zuordnen.

Beispiel:
SemOrg

IV 16 Anforderungen und Anforderungsarten

Es kann zwischen gültigen und ungültigen Abnahmeszenarien unterschieden werden. Ein gültiges Szenario führt zu einer korrekten Ausführung, ein ungültiges muss zu einem definierten Fehlerstatus führen.

Resümee Die sorgfältige Formulierung von Abnahmekriterien ist aufwendig. Es werden dabei jedoch Lücken und Unvollständigkeiten aufgedeckt und Präzisierungen vorgenommen, die insgesamt zu einer wesentlichen Qualitätsverbesserung der Anforderungsspezifikation führen.

Zeitpunkt Abnahmekriterien sollen zeitlich versetzt zur Festlegung der Anforderungen ermittelt werden, wenn eine gewisse Stabilisierung der Anforderungen eingetreten ist. Rückwirkungen, die sich aus den Abnahmekriterien auf die Anforderungen ergeben, können noch während der Analyse durch Änderung der betroffenen Anforderungen behoben werden.

Die Formulierung von Abnahmekriterien wird erleichtert, wenn jeweils fachlich zusammenhängende Anforderungspakete betrachtet werden.

Besitzen Anforderungen eine hohe Kritikalität (siehe »Anforderungsattribute«, S. 479), dann sind u. U. zusätzliche Abnahmekriterien zu formulieren, um sicher zu sein, dass die Anforderung exakt realisiert wird.

Die Abnahmekriterien sollten Bestandteil der Anforderungsspezifikation sein und damit auch Vertragsbestandteil zwischen Auftraggeber und Auftragnehmer.

Die Abnahmekriterien sollten von einer Person geschrieben werden, die Fachexperte in der betreffenden Domäne ist, aber die Anforderungen *nicht* erstellt hat. Die Überprüfung der Abnahmekriterien wiederum sollte durch diejenigen erfolgen, die die Anforderungen erstellt haben.



Überlegen Sie, warum dies sinnvoll ist.

17 Anforderungen an Anforderungen

Welche Anforderungen würden Sie an jede einzelne Anforderung stellen? Frage

Jede *einzelne* Anforderung soll eine Reihe von Kriterien erfüllen (in Anlehnung an [IEEE830, S. 4 ff.]). Sie soll sein: Antwort

- **Korrekt** (*correct*): Die Anforderung ist korrekt, wenn das zu erstellende Softwaresystem sie erfüllen soll. Der Auftraggeber und/oder die Benutzer können sagen, ob die Anforderung die Bedürfnisse reflektiert.
- **Eindeutig** (*unambiguous*): Die Anforderung wird von allen *Stakeholdern* gleich interpretiert. Als Minimum sollte die beschriebene Charakteristik durch einen einzigen, eindeutigen Begriff beschrieben werden. Wird ein Begriff in verschiedenen Kontexten unterschiedlich verwendet, dann sollte er in einem Glossar definiert werden. Alle Erfahrungen haben gezeigt, dass in natürlicher Sprache beschriebene Anforderungen meist nicht eindeutig interpretiert werden. Das hat zur Entwicklung formalisierter Beschreibungsverfahren geführt.
- **Vollständig** (*complete*): Die Anforderung muss die geforderte Funktionalität vollständig beschreiben. Ist die Anforderung noch unvollständig, dann ist dies zu kennzeichnen, z. B. durch TBD (*to be determined*).
- **Konsistent** (*consistent*): Die Anforderung muss in sich widerspruchsfrei sein.
- **Klassifizierbar nach Wichtigkeit** (*ranked for importance*): Der Anforderung kann eine Wichtigkeitsstufe zugeordnet werden. Einige Anforderungen können essenziell sein, z. B. bei lebenskritischen Anwendungen, während andere nur wünschenswert sind. Mögliche Klassen sind:
 - ☐ Essenziell (*essential*): Das Softwaresystem kann ohne die Erfüllung der Anforderung nicht eingesetzt werden.
 - ☐ Bedingt (*conditional*): Die Anforderung wertet das Softwaresystem auf, aber die Nichtrealisierung macht das System nicht unbrauchbar.
 - ☐ Optional (*optional*): Anforderung, die das System u.U. wertvoller macht. Gibt dem Auftragnehmer die Möglichkeit, etwas Zusätzliches dem Auftraggeber anzubieten.

IV 17 Anforderungen an Anforderungen

- **Klassifizierbar nach Stabilität** (*ranked for stability*): Der Anforderung kann eine Stabilitätsstufe zugeordnet werden. Unter Stabilität wird hierbei die Anzahl der erwarteten Änderungen bei dieser Anforderung verstanden. Dieser Wert kann ein Erfahrungswert sein oder das Wissen über vorausszusehende Veränderungen, die die Organisation, die Funktionen oder die Personen, die durch die Software unterstützt werden, betreffen.
- **Überprüfbar** (*verifiable*): Die Anforderung muss so beschrieben sein, dass sie nach der Realisierung überprüfbar bzw. testbar ist. Es muss ein kosteneffektives Verfahren vorhanden sein, mit der eine Person oder eine Maschine überprüfen kann, dass das System die Anforderung erfüllt. Jede inkonsistente Anforderung ist nicht überprüfbar. Nicht überprüfbare Anforderungen enthalten Aussagen wie »Funktioniert gut«, »gute Benutzungsoberfläche«, »soll normalerweise eintreten«.
- **Verfolgbar** (*traceable*): Die Anforderung muss sich eindeutig identifizieren lassen, z. B. durch eine eindeutige Anforderungsnummer, die unverändert bleibt.

Frage Welche Anforderungen soll eine Anforderungsspezifikation, die alle Anforderungen zu einem Softwaresystem enthält, erfüllen?

Antwort Alle Anforderungen zusammen sollen sein (teilweise nach [IEEE830, S. 4 ff.]):

- **Korrekt**: Alle Anforderungen zusammen sind korrekt, wenn das zu erstellende Softwaresystem sie erfüllen soll. Der Auftraggeber und/oder die Benutzer können sagen, ob die Anforderungen die Bedürfnisse reflektieren.
- **Vollständig**: Alle Anforderungen zusammen müssen alle relevanten Eigenschaften des Softwaresystems festlegen. Eine solche Anforderung ist sinnvoll, wenn eine Softwareentwicklung nach dem Wasserfallmodell oder dem inkrementellen Modell erfolgt. Wird jedoch ein evolutionäres Modell oder ein agiles Modell verwendet, dann werden zu Beginn nur ein Teil der Anforderungen festgelegt (siehe »Lehrbuch für Softwaretechnik – Softwaremanagement«). Auch ist zu Beginn der Gesamtumfang des Softwaresystems noch nicht bekannt.
- **Konsistent**: Alle Anforderungen müssen untereinander widerspruchsfrei sein.
- **Abhängigkeiten angebbar**: Abhängigkeiten zwischen Anforderungen müssen sichtbar und nachverfolgbar sein, z. B. durch Angabe der Anforderungsnummern abhängiger Anforderungen.
- **Modifizierbar** (*modifiable*): Änderungen an den Anforderungen können einfach, vollständig und konsistent unter Beibehaltung der vorhandenen Struktur und des verwendeten Stils durchgeführt werden. Die Anforderungsspezifikation muss dazu ein In-

haltsverzeichnis, einen Index und explizite Querverweise besitzen. Anforderungen dürfen nicht redundant sein, d.h. eine Anforderung darf nur einmal vorhanden sein.

- **Erweiterbar:** Neue Anforderungen können einfach unter Beibehaltung der vorhandenen Struktur und des verwendeten Stils hinzugefügt werden.
- **Im Umfang angemessen:** Die notwendigen Anforderungen müssen in ausreichender Detaillierung und ausreichendem Präzisierungsgrad beschrieben werden. Es darf nur das *Was*, nicht das *Wie* beschrieben werden.
- **Nach verschiedenen Sichten auswertbar:** Die Anforderungsspezifikation muss nach verschiedenen Sichten – entsprechend dem Bedarf der einzelnen *Stakeholder* – sortierbar und gruppierbar sowie in verschiedenen Granularitätsstufen darstellbar sein.

Anforderungsspezifikation als Teil einer Systemspezifikation

Wird eine Anforderungsspezifikation parallel zu einer Systemspezifikation entwickelt, d.h. das zu entwickelnde Softwaresystem ist Bestandteil eines softwareintensiven Systems, dann muss darauf geachtet werden, dass die Anforderungsspezifikation kompatibel mit der Systemspezifikation ist, sonst ist sie nicht korrekt. Liegt bereits eine Systemspezifikation vor, dann muss von vornherein darauf geachtet werden, dass die Anforderungen der Systemspezifikation durch die Anforderungsspezifikation erfüllt werden.

Von der Theorie zur Praxis

In der Praxis ist man in der Regel weit davon entfernt, die obigen Qualitätskriterien für Anforderungen einzuhalten. Kunden – ob intern oder extern – werden die Wünsche an das neue Softwaresystem weder systematisch, noch strukturiert, noch vollständig beschreiben. Normalerweise wird zwischen abstrakten Angaben und konkreten Wünschen, zwischen allgemeinen Aussagen und spezifischen Festlegungen hin- und hergesprungen. Oft erfolgt eine fallorientierte Darstellung der Wünsche: »In der Situation A sollte das System dies und jenes tun«. Diese Problematik entsteht dadurch, dass die Gesprächspartner kein vollständiges Modell des zu entwickelnden Systems »im Kopf« haben.

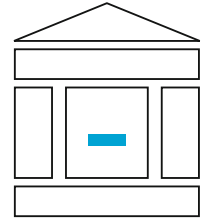
Aufgabe des *Requirements Engineer* ist es, in Zusammenarbeit mit den *Stakeholdern* die oben aufgeführten Qualitätskriterien für Anforderungen möglichst zu erreichen.

18 Anforderungsattribute

Zu jeder Anforderung sind Attribute zu dokumentieren, die bei der Anforderungserfassung eingetragen oder im Laufe des *Requirements Engineering* ergänzt werden. Erfolgt die Anforderungsdokumentation werkzeuggestützt, dann kann auf die Angabe der Attribute hingewiesen werden bzw. können Voreinstellungen vorgenommen werden. Bei der Verwendung von Textsystemen können Anforderungsschablonen verwendet werden. Die Attribute hängen teilweise von der Anforderungsart ab.

Folgende Attribute können erfasst werden (siehe auch [Pohl07, S. 257 ff.], [Rupp07, S. 385 f.], [Eber08, S. 161 f.]):

- **Identifikator**: Dient zur eindeutigen Identifikation und Referenzierung, z. B. durch eine Nummer oder einen Bezeichner.
- **Kurzbezeichnung der Anforderung** (optional): Eindeutige, charakterisierende Bezeichnung, z. B. sprechender Name.
- **Anforderungstyp**: Gibt den Typ der Anforderung an, z. B. funktionale Anforderung.
- **Beschreibung der Anforderung**: Die Anforderung wird informal, semiformal oder formal beschrieben, z. B. in Form eines strukturierten Satzes (siehe »Natürlichsprachliche Anforderungen«, S. 481) oder eines Klassendiagramms.
- **Anforderungssicht** (bei funktionalen Anforderungen): Angabe, ob die Statik, Dynamik oder Logik des Produkts oder eine Kombination davon beschrieben wird, z. B. Angabe, dass die Anforderung die Dynamik beschreibt.
- **Querbezüge** (optional): Bezüge und Abhängigkeiten zu anderen Anforderungen, z. B. hängt ein Erfassungsformular von den Datenanforderungen ab.
- **Status des Inhalts**: Gibt an, wie vollständig die Anforderung beschrieben ist, z. B. Idee, grober Inhalt, detaillierter Inhalt.
- **Abnahmekriterien**: Angaben, wie die Erfüllung der Anforderung bei der Abnahme überprüft werden soll, z. B. durch Angabe eines Testszenarios.
- **Schlüsselwörter** (optional): Charakteristische Schlüsselwörter, die es erlauben die Anforderungen zu filtern, z. B. GUI, Persistenz.
- **Priorität der Anforderung**: Festlegung der Priorität aus Auftraggeber- und Auftragnehmersicht, z. B. hohe, mittlere oder niedrige Priorität.
- **Stabilität der Anforderung**: Maß für die Wahrscheinlichkeit, dass sich die Anforderung im Entwicklungsverlauf ändert, z. B. fest, gefestigt, volatil.



Identifikation

Inhalt

Management

IV 18 Anforderungsattribute

- **Kritikalität der Anforderung:** Bei einer fehlerhaften Umsetzung der Anforderung können Schäden entstehen, z. B. kann bei einer hohen Kritikalität eines technischen Systems ein Fehlverhalten zum Verlust von Menschenleben führen.
- **Entwicklungsrisiko:** Gibt an, wie hoch das Entwicklungsrisiko für die Umsetzung dieser Anforderung eingeschätzt wird, z. B. hohes Risiko bzgl. der Termineinhaltung.
- **Aufwand:** Geschätzter Aufwand für die Realisierung der Anforderung, z. B. zwei Personenmonate.
- **Konflikte** (optional): Beschreibung identifizierter Konflikte bezogen auf diese Anforderung, z. B. Sicherheitsaspekte vs. Benutzungsfreundlichkeit.
- Dokumentation ■ **Autor:** Person, die die Anforderung beschrieben hat bzw. für die Anforderung verantwortlich ist.
- **Quelle:** Ursprung der Anforderung und Begründung, warum die Anforderung berücksichtigt wurde, z. B. Name des *Stakeholders*, Bezeichnung des Gesetzes.
- **Version und Änderungsbeschreibung:** Versionsnummer der Anforderung und Angabe welche Änderungen gegenüber der Vorgängerversion vorgenommen wurde einschl. Begründung, z. B. V02 (Detaillierung der Anforderung).
- **Bearbeitungsstatus:** Gibt an, in welchem Bearbeitungsstatus sich die Anforderung befindet, z. B. erzeugt, in Bearbeitung, der QS vorgelegt, fertig gestellt.

Kritikalität	Art des Fehlverhaltens
hoch	Fehlverhalten kann zum Verlust von Menschenleben führen.
mittel	Fehlverhalten kann die Gesundheit von Menschen gefährden oder zur Zerstörung von Sachgütern führen.
niedrig	Fehlverhalten kann zur Beschädigung von Sachgütern führen, ohne jedoch Menschen zu gefährden.
keine	Fehlverhalten gefährdet weder die Gesundheit von Menschen noch Sachgüter.

Tab. 18.0-1: Attribute für Anforderungen können unternehmensspezifisch oder projektbezogen festgelegt werden. Für jedes Anforderungsattribut ist ein Attributierungsschema festzulegen, das Folgendes festlegt (siehe [Pohl07, 257 f.]):

Eindeutiger Attributnamen, Attributsemantik, Wertebereich, Wertesemantik

- Beispiel **Attributname:** Kritikalität der Anforderung
Attributsemantik: Maß für die Schäden, die entstehen können, wenn die Anforderung fehlerhaft umgesetzt wird.
Wertebereich: {hoch, mittel, niedrig, keine}
Wertesemantik [V-Modell XT 06]:

19 Natürlichsprachliche Anforderungen

In den meisten Fällen werden Anforderungen in natürlicher Sprache formuliert.

Überlegen Sie, welche Vor- und Nachteile die Verwendung natürlicher Sprache für die Formulierung von Anforderungen mit sich bringt.

Frage

Natürliche Sprachen besitzen vielfältige Vorteile:

Antwort

- + Die Verwendung der natürlichen Sprache ist **einfach** – vorausgesetzt alle *Stakeholder* beherrschen die jeweilige Sprache, z. B. Deutsch.
- + Eine natürliche Sprache ist **flexibel** – sie kann abstrakte und konkrete Dinge beschreiben.
- + Und sie ist für jede Anwendungsdomäne einsetzbar – sie ist **universell**.

Sie sind jedoch in mehrfacher Hinsicht **mehrdeutig**:

- **Synonyme** (z. B. Innenstadt – City) und **Homonyme** (z. B. Bank) führen zu einer **lexikalischen Mehrdeutigkeit**.
- Eine **syntaktische Mehrdeutigkeit** zeigt das folgende Beispiel: Die letzten 10 Buchungen und die Stornierungen des Kunden werden im Fenster angezeigt. Diese Anforderung kann so interpretiert werden, dass die letzten 10 Buchungen und die letzten 10 Stornierungen angezeigt werden. Eine andere Interpretation besteht darin, die letzten 10 Buchungen und alle Stornierungen anzuzeigen.
- Kann eine Anforderung in einem Kontext unterschiedlich interpretiert werden – auch wenn keine lexikalische oder syntaktische Mehrdeutigkeit vorliegt –, dann liegt eine **semantische Mehrdeutigkeit** vor, wie folgendes Beispiel zeigt: Jeder Sensor ist mit einem Service verbunden. Dies kann bedeuten: Es gibt viele Services – jeder ist mit einem anderen Sensor verbunden. Oder: Es gibt genau einen Service, mit denen alle Sensoren verbunden sind. Die Mehrdeutigkeit entsteht durch das Verhältnis der Quantoren »jeder« und »einem«.
- Eine **referentielle Mehrdeutigkeit** zeigt das folgende Beispiel: Beim Login muss zuerst das Benutzerkennzeichen und dann das Passwort eingegeben werden. Ist dies nicht korrekt, schlägt die Anmeldung fehl.

IV 19 Natürlichsprachliche Anforderungen

- *Vage Begriffe*, wie z.B. »neben«, können von unterschiedlichen Personen verschieden interpretiert werden: Der Sensor muss neben der Tür angebracht werden.

Es gibt im Wesentlichen drei Möglichkeiten, um die Probleme der natürlichen Sprache zu reduzieren oder zu vermeiden:

- Erstellung eines Glossars
- Benutzung sprachlicher Anforderungsschablonen
- Weitgehende Vermeidung natürlichsprachlicher Anforderungen

Erstellung eines Glossars

Durch die Definition von Begriffen in Form eines Glossars werden lexikalische Mehrdeutigkeiten vermieden. Ein Glossareintrag sollte folgende Struktur besitzen:

- **Begriff**: zu definierender Begriff
- **Definition**: Text der Definition
- **Synonyme**: Begriffe mit gleicher Bedeutung
- **Plural**: Wenn ungewöhnlich, z.B. Algorithmus, Algorithmen
- **Kurzform**: Wenn vorhanden, z.B. QM für Qualitätsmanagement
- **Langform**: Wenn der Begriff in Kurzform angegeben ist, z.B. World Wide Web für WWW.
- **Übersetzung**: In eine oder mehrere Sprachen, wenn die Projektsprache oder die Benutzungsoberfläche mehrsprachig sind.

Zusätzlich kann ein Glossarbegriff Metainformationen wie Autor, Version usw. besitzen.

Ein Glossar kann ein eigenständiges Anforderungsartefakt sein oder Bestandteil eines anderen Artefakts – in der Regel Bestandteil der Anforderungsspezifikation. Damit auf das Glossar alle *Stakeholder* Zugriff haben, hat das Glossar als eigenständiges Artefakt Vorteile.

Benutzung sprachlicher Anforderungsschablonen

Um die syntaktische und semantische Mehrdeutigkeit bei natürlichsprachlichen Anforderungen zu reduzieren, haben sich in der Praxis Anforderungsschablonen bewährt. Die Abb. 19.0-1 zeigt eine Anforderungsschablone ohne Bedingungen (in Anlehnung an [Rupp07, S. 233]).

- Das **Prozesswort** (ganz rechts) beschreibt die geforderte Funktionalität durch ein Verb wie anlegen, löschen, berechnen.
- Im mittleren Teil wird angegeben, um welche Art von Systemaktivität es sich handelt:
- ☐ Selbstständige Systemaktivität: Das System führt den Prozess selbstständig durch.

19 Natürlichsprachliche Anforderungen IV

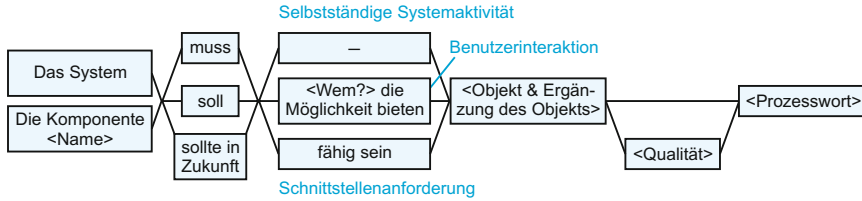


Abb. 19.0-1:
Aufbau einer
Anforderungsscha-
blone ohne
Bedingungen.

- Benutzerinteraktion: Das System stellt dem Benutzer die Prozessfunktionalität zur Verfügung.
- Schnittstellenanforderung: Das System führt den Prozess in Abhängigkeit von einem anderen System aus. Es ist passiv und wartet auf ein externes Ereignis.
- Die rechtliche Verbindlichkeit einer Anforderung kann mit Hilfsverben wie folgt festgelegt werden:
 - muss: rechtlich bindend
 - soll (kann): dringend empfohlen
 - sollte in Zukunft: wahrscheinlich ein zukünftige Anforderung
- Die nähere oder ergänzende Bestimmung des Prozesswortes wird durch <Objekt & Ergänzung des Objekts> angegeben.
- Nichtfunktionale Anforderungen legen in der Regel <Qualitäten> in Bezug auf das Prozesswort fest.

- Benutzerinteraktion:
Das System muss *dem Kunden* die Möglichkeit bieten, sich über *Seminare und Veranstaltungen* zu informieren.
- Benutzerinteraktion:
Das System soll *dem Seminarsachbearbeiter* die Möglichkeit bieten, *Seminare und Veranstaltungen* mit *selbst erstellten Suchanfragen* auszuwerten.
- Selbstständige Systemaktivität:
Das System muss die *Kundendaten permanent speichern*.
- Schnittstellenanforderung:
Das System muss *fähig sein*, dem *Buchhaltungssystem Rechnungsdatensätze mindestens einmal am Tag* zur Verfügung zu stellen.

Beispiele:
SemOrg

Mit manchen Anforderungen sind logische und zeitliche Bedingungen verknüpft. Eine entsprechend erweiterte Schablone zeigt die Abb. 19.0-2.

- Selbstständige Systemaktivität:
Falls ein Kunde im Zahlungsverzug ist, muss das System *eine neue Buchung nicht erlauben*.

Beispiel

- + Wie die Beispiele zeigen, führt der Einsatz einer Schablone zu einer Vereinheitlichung der Anforderungsbeschreibungen und zu einer eindeutigeren Semantik.

IV 19 Natürlichsprachliche Anforderungen

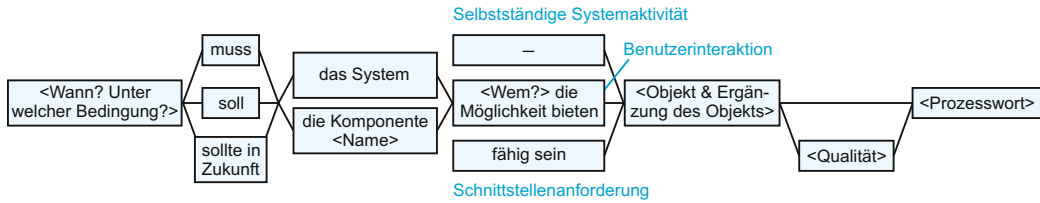


Abb. 19.0-2:
Aufbau einer
Anforderungsscha-
blone mit
Bedingungen.

- Auf der anderen Seite ist es schwierig, alle Anforderungen in Form solcher Sätze zu formulieren – manche Festlegungen sind schwer zu lesen.

Weitgehende Vermeidung natürlichsprachlicher Anforderungen

In der Softwaretechnik gibt es etablierte formale und semiformale Basiskonzepte (siehe »Basiskonzepte«, S. 99), die eine präzisere und eindeutige Festlegung von Anforderungen erlauben, auch verglichen mit schablonenbasierten Anforderungsbeschreibungen. Da auf der Auftraggeberseite auch immer häufiger Informatiker tätig sind, sollte es bei Kenntnis der Basiskonzepte kein Problem sein, die Anforderungen auf der Auftraggeber- und der Auftragnehmerseite eindeutig zu interpretieren.

Nach Auffassung des Autors sollten daher auch in der Anforderungsspezifikation immer Basiskonzepte eingesetzt werden – immer, wenn dies möglich ist – und dadurch weitgehend auf natürlichsprachliche Anforderungen verzichtet werden.

20 Anforderungsschablonen

Anforderungsspezifikationen lassen sich gliedern in:

- Spezifikationen ohne festgelegte Regeln,
- Spezifikationen mit nummerierten oder markierten Anforderungen,
- Spezifikationen mit festgelegten Gliederungsschemata.

Die beiden letzten Formen können kombiniert verwendet werden.

Um zu einer einheitlichen Struktur und Gliederung von Anforderungsspezifikationen zu gelangen, werden in der Regel Schablonen – auch Muster oder Gliederungsschemata genannt – verwendet.

Der Standard ANSI/IEEE Std. 830–1998 enthält eine Anforderungsschablone mit verschiedenen Varianten:

- »Anforderungsschablone der IEEE 830–1998«, S. 485

In dem Prozessmodell »V-Modell XT« gibt es sowohl für Lasten- als auch für Pflichtenhefte Schablonen:

- »Anforderungsschablonen im V-Modell XT«, S. 487

Insbesondere für kleinere Softwareentwicklungen sind die Schablonen von ANSI/IEEE und dem V-Modell XT zu umfangreich. Für kleinere Entwicklungen werden vom Autor entwickelte Schablonen vorgestellt:

- »Schablonen für Lastenheft, Pflichtenheft und Glossar«, S. 492

Bei agilen Softwareentwicklungen sind Anforderungsspezifikationen überhaupt nicht oder nur in geringem Umfang vorhanden. In der Regel werden *User Stories* beschrieben:

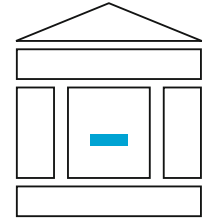
- »Schablonen für agile Entwicklungen«, S. 497

Für softwareintensive Systeme wird in IEEE Std 1362–1998 ein Gliederungsschema für ein System-Lastenheft (*System Definition – Concept of Operations (ConOps) Document*) vorgeschlagen [IEEE1362].

IEEE 1362

Ein System-Pflichtenheft für softwareintensive Systeme wird in dem IEEE Std 1233–1998 angegeben [IEEE1233].

IEEE 1233



i

20.1 Anforderungsschablone der IEEE 830–1998

Als Muster für eine Anforderungsspezifikation mit festgelegtem Gliederungsschema kann der *IEEE Recommended Practice for Software Requirements Specifications (SRS)* angesehen werden (genormt

IV 20 Anforderungsschablonen

als ANSI/IEEE Std 830-1998) [IEEE830]¹. Der Standard ANSI/IEEE Std 830-1998 wurde 1984 von der IEEE verabschiedet und ein Jahr später auch von der ANSI (American National Standard Institute) übernommen. Heute gilt die Fassung von 1998. Insgesamt besteht die Richtlinie aus 31 Seiten. In der Praxis werden oft einzelne Gliederungspunkte der IEEE 830 nicht durch verbale Beschreibungen, sondern durch Einsatz einzelner Konzepte beschrieben. Aufbau und Struktur einer Anforderungsspezifikation nach IEEE 830 sehen wie folgt aus (Kurzfassung, Übersetzung des Autors):

1 Einleitung (*Introduction*)

Gibt einen Überblick über die Anforderungsdefinition.

1.1 Zielsetzung (*Purpose*)

1.2 Produktziele (*Scope*)

1.3 Definitionen, Akronyme und Abkürzungen (*Definitions, Acronyms and Abbreviations*)

1.4 Referenzen (*References*)

1.5 Überblick (*Overview*)

2 Übersichtsbeschreibung (*Overall Description*)

Gibt einen Überblick über das Produkt und die allgemeinen Faktoren, die seine Konzeption beeinflussen.

2.1 Produkt-Umgebung (*Product Perspective*)

2.2 Produkt-Funktionen (*Product Functions*)

2.3 Benutzer-Eigenschaften (*User Characteristics*)

2.4 Restriktionen (*Constraints*)

2.5 Annahmen und Abhängigkeiten (*Assumptions and Dependencies*)

3 Spezifische Anforderungen (*Specific Requirements*)

Beschreibung aller Details, die für die Erstellung des System-Entwurfs benötigt werden. Das am besten geeignete Gliederungsschema dieses Kapitels hängt von der Anwendung und der zu spezifizierenden Software ab.

Die IEEE-Richtlinie enthält dazu acht Vorschläge.

Unabhängig von der Strukturierung sollte das Kapitel 3 folgende Informationen enthalten:

- Externe Schnittstellen-Anforderungen (*External Interface Requirements*)
- Funktionale Anforderungen (*Functional Requirements*)
- Leistungsanforderungen (*Performance Requirements*)
- Entwurfsrestriktionen (*Design Constraints*)
- Eigenschaften des Softwaresystems (*Software Systems Attributes*)
- Andere Anforderungen (*Other Requirements*)

¹Die IEEE (Institute of Electrical and Electronics Engineers) ist eine der zwei großen internationalen wissenschaftlichen Organisationen (die andere ist ACM), die für die Softwaretechnik wichtig sind. IEEE besitzt ein Standardisierungsprogramm für die Elektrotechnik und die Softwaretechnik.

Für eine **objektorientierte Entwicklung** kann Kapitel 3 folgendermaßen gegliedert werden:

3 Spezifische Anforderungen (Organisiert nach Objekten)

3.1 Externe Schnittstellen-Anforderungen

3.1.1 Benutzungsschnittstelle (*User Interfaces*)

3.1.2 Hardware-Schnittstellen (*Hardware Interfaces*)

3.1.3 Software-Schnittstellen (*Software Interfaces*)

3.1.4 Kommunikations-Schnittstellen (*Communications Interfaces*)

3.2 Klassen/Objekte (*Classes/Objects*)

3.2.1 Klasse/Objekt 1 (*Class/Object 1*)

3.2.1.1 Attribut (direkt oder geerbt) (*Attributes direct or inherited*)

3.2.1.1.1 Attribut 1 bis 3.2.1.1.n Attribut n

3.2.1.2 Funktionen (*Functions, Services, Methods, direct or inherited*)

3.2.1.2.1 Funktionale Anforderung 1.1 bis 3.2.1.2.m Funktionale Anforderung 1.m

3.2.1.3 Botschaften (*Messages, communications received or sent*)

3.2.2 Klasse/Objekt 2 bis 3.2.p Klasse/Objekt p

3.3 Leistungsanforderungen (*Performance Requirements*)

3.4 Entwurfsrestriktionen (*Design Constraints*)

3.5 Eigenschaften des Softwaresystems (*Software Systems Attributes*)

20.2 Anforderungsschablonen im V-Modell XT

Im V-Modell XT (siehe Lehrbuch der Softwaretechnik, Band 3) gibt es folgende Vorlagen für Anforderungsspezifikationen – im V-Modell **Produktvorlagen** genannt:

- Lastenheft Gesamtprojekt
- Anforderungen (Lastenheft)
- Gesamtsystemspezifikation (Pflichtenheft)
- Systemspezifikation

Im V-Modell XT wird immer ein System entwickelt – auch wenn es nur aus Software besteht.

Im »Lastenheft Gesamtprojekt« sollen alle an das zu entwickelnde System verbindlich gestellten Anforderungen vollständig und konsistent beschrieben werden. Es ist Basis für die Aufteilung in Teilprojekte.

»Alle relevanten Anforderungen an das System werden vom Auftraggeber ermittelt und dokumentiert. Kern des Lastenhefts Gesamtprojekt sind die funktionalen und nicht-funktionalen Anforderungen

Lastenheft
Gesamtprojekt

Zitat

IV 20 Anforderungsschablonen

an das System, sowie eine Skizze des Gesamtsystementwurfs. Der Entwurf berücksichtigt die zukünftige Umgebung und Infrastruktur, in der das System später betrieben wird, und gibt Richtlinien für Technologieentscheidungen. Die Skizze der Gesamtsystemarchitektur ist die bestimmende Grundlage für die Aufteilung des Gesamtprojektes in Teilprojekte. [...] Für die Erstellung des Lastenhefts Gesamtprojektes sowie für dessen Qualität ist der Auftraggeber alleine verantwortlich« [VXTLG08, S. 5].

Anforderungen (Lastenheft)

Zitat »Das Produkt Anforderungen (Lastenheft) enthält alle an das zu entwickelnde System verbindlich gestellten Anforderungen. Es ist Grundlage für Ausschreibung und Vertragsgestaltung und damit wichtigste Vorgabe für die Angebotserstellung. Das Lastenheft ist Bestandteil des Vertrags zwischen Auftraggeber und Auftragnehmer. Mit den Anforderungen werden die Rahmenbedingungen für die Entwicklung festgelegt, die dann vom Auftragnehmer in der Gesamtsystemspezifikation (Pflichtenheft) detailliert ausgestaltet werden.

Alle relevanten Anforderungen an das System werden vom Auftraggeber ermittelt und dokumentiert. Sie enthalten die für den Auftragnehmer notwendigen Informationen zur Entwicklung des geforderten Systems. Kern des Lastenhefts sind die funktionalen und nicht-funktionalen Anforderungen an das System, sowie eine Skizze des Gesamtsystementwurfs. Der Entwurf berücksichtigt die zukünftige Umgebung und Infrastruktur, in der das System später betrieben wird, und gibt Richtlinien für Technologieentscheidungen. Zusätzlich werden die zu unterstützenden Phasen im Lebenszyklus des Systems identifiziert und als logistische Anforderungen aufgenommen. Ebenfalls Teil der Anforderungen ist die Festlegung von Lieferbedingungen und Abnahmekriterien. [...]

Für die Erstellung des Lastenhefts sowie für dessen Qualität ist der Auftraggeber alleine verantwortlich« [VPAL08, S. 5].

Gliederung Das Lastenheft ist folgendermaßen gegliedert [VPAL08]:

1 Einleitung

Beschreibung der Aufgabe dieses Lastenhefts (siehe oben).

2 Ausgangssituation und Zielsetzung

Die Ausgangssituation und der Anlass zur Durchführung des Projektes werden beschrieben. Die Defizite bzw. Probleme existierender Systeme oder auch der aktuellen Situation werden dargestellt, die zur Entscheidung geführt haben, das Projekt durchzuführen. Die Vorteile, die durch den Einsatz des neuen Systems erwartet werden, werden aufgezählt. Alle relevanten *Stakeholder* des Projektes werden benannt und die technische und fachliche Einbet-

tung des zu entwickelnden Systems in seine Umgebung wird skizziert. Erste Rahmenbedingungen für die Entwicklung werden identifiziert und beschrieben.

3 Funktionale Anforderungen

»Funktionale Anforderungen beschreiben die Fähigkeiten eines Systems, die ein Anwender erwartet, um mit Hilfe des Systems ein fachliches Problem zu lösen. Die Anforderungen werden aus den zu unterstützenden Geschäftsprozessen und den Ablaufbeschreibungen zur Nutzung des Systems abgeleitet. Die Beschreibung der funktionalen Anforderungen erfolgt beispielsweise in Form von Anwendungsfällen (*Use Cases*). Ein Anwendungsfall beschreibt dabei einen konkreten, fachlich in sich geschlossenen Teilvorgang. Die Gesamtheit der Anwendungsfälle definiert das Systemverhalten. Ein Anwendungsfall kann in einfachem Textformat beschrieben werden, häufig stehen jedoch organisationsspezifische Muster zur Beschreibung zur Verfügung. Für datenzentrierte Systeme wird im Rahmen der funktionalen Anforderungen ein erstes fachliches Datenmodell erstellt, das als Grundlage des späteren Datenbankentwurfs dient. Das fachliche Datenmodell des Systems wird aus den Entitäten des Domänenmodells abgeleitet. Die funktionalen Anforderungen sind die zentralen Vorgaben für die Systementwicklung. Sie werden in der Gesamtsystemspezifikation (Pflichtenheft) übernommen und bei Bedarf konkretisiert.«

4 Nicht-funktionale Anforderungen

»Nicht-funktionale Anforderungen definieren grundlegende Eigenschaften eines Systems, die im Architekturentwurf berücksichtigt werden müssen.«

5 Skizze des Lebenszyklus und der Gesamtsystemarchitektur

»Das reine Aufstellen von Anwenderanforderungen ohne Überlegungen zu möglichen Lösungsräumen birgt die große Gefahr, unrealistische Anwenderanforderungen zu definieren. Für die Einordnung, Systematisierung, Kategorisierung und auch Priorisierung von Anwenderanforderungen ist ein Koordinierungsrahmen hilfreich, um die Visualisierung der Anwenderanforderungen zu erleichtern.

Diese Aufgabe kann eine Gesamtsystemarchitektur leisten, die die Sichtweise des Anwenders repräsentiert [...]. Das heißt, es ist eine funktionale Systemarchitektur mit Einbettung in die funktionalen Abläufe von Nachbarsystemen zu erstellen. Eine technische Systemarchitektur ist in dieser frühen Phase kaum möglich. [...]

Des Weiteren sind die Besonderheiten der Einsatzumgebung des neuen Systems zu beschreiben, um vor allem die Anforderungen an die Systemsicherheit berücksichtigen zu können.«

IV 20 Anforderungsschablonen

6 Lieferumfang

»Es sind alle Gegenstände und Dienstleistungen aufzulisten, die während des Projektverlaufs oder bei Abschluss des Projektes vom Auftragnehmer an den Auftraggeber zu liefern sind. Jede Lieferung erfordert eine Abnahmeprüfung.«

7 Abnahmekriterien

»Abnahmekriterien legen fest, welche Kriterien die Lieferung erfüllen muss, um den Anforderungen zu entsprechen. Sie sollen messbar dargestellt werden. Aus vertraglicher Sicht beschreiben die Abnahmekriterien die Bedingungen für die Entscheidung, ob das Endprodukt die gestellten Anforderungen erfüllt oder nicht. [...]

In der Phase bis zur Auftragsvergabe können die Abnahmekriterien nur in einer allgemeinen Form, zum Beispiel als K.-o.-Kriterien, angegeben werden. Darin wird beispielsweise definiert, dass mindestens 90 % aller Prüffälle für eine erfolgreiche Abnahme erfüllt sein müssen. Diese allgemeinen Abnahmekriterien sollten auch die Forderung nach einer Erstellung von Abnahmekriterien durch den Auftragnehmer enthalten. Dabei sind der Aufbau und die Anzahl der Abnahmekriterien durch den Auftraggeber zu skizzieren. Eine Strukturierung der Abnahmekriterien nach ihren drei wesentlichen Bestandteilen – Ausgangssituation, Aktion(en) und erwartetes Ergebnis – ist anzustreben. In jedem Fall müssen die erwarteten Ergebnisse der Abnahme pro Abnahmekriterium festgelegt werden.«

8 Abkürzungsverzeichnis

9 Literaturverzeichnis

10 Abbildungsverzeichnis

Gesamtsystemspezifikation (Pflichtenheft)

Zitat »Die Gesamtsystemspezifikation (Pflichtenheft) ist das Pendant zu dem Auftraggeberprodukt Anforderungen (Lastenheft) auf Auftragnehmerseite. Sie wird vom Auftragnehmer in Zusammenarbeit mit dem Auftraggeber erstellt und stellt das zentrale Ausgangsdokument der Systemerstellung dar.

Wesentliche Inhalte der Gesamtsystemspezifikation sind die funktionalen und nicht-funktionalen Anforderungen an das zu entwickelnde Gesamtsystem. Die Anforderungen werden aus den Anforderungen (Lastenheft) übernommen und geeignet aufbereitet. Eine erste Grobarchitektur des Systems wird entwickelt und in einer Schnittstellenübersicht beschrieben. Das zu entwickelnde System sowie weitere zu entwickelnde Unterstützungssysteme werden identifiziert und den Anforderungen zugeordnet. [...]. Abnahmekriterien

und Lieferumfang für das fertige Gesamtsystem werden aus den Anforderungen (Lastenheft) übernommen und konkretisiert« [VPGP08, S. 5].

Das Pflichtenheft ist folgendermaßen gegliedert [VPGP08]:

Gliederung

1 Einleitung

Beschreibung der Aufgabe dieses Pflichtenhefts (siehe oben).

2 Ausgangssituation und Zielsetzung

Analog wie beim Lastenheft (siehe oben).

3 Funktionale Anforderungen

»Die funktionalen Anforderungen sind die zentralen Vorgaben für die Systementwicklung. Sie werden in der Gesamtsystemspezifikation (Pflichtenheft) übernommen und bei Bedarf konkretisiert.«

4 Nicht-funktionale Anforderungen

Analog wie beim Lastenheft (siehe oben).

5 Risikoakzeptanz

»Für sicherheitskritische Systeme werden in diesem Thema Vorgaben für die Behandlung der Systemsicherheit festgelegt. Es wird aufgezeigt, welche Risiken im Rahmen des Systembetriebs bestehen, welche Schäden, oder auch welche Klassen von Schäden, mit welcher Wahrscheinlichkeit auftreten können und inwieweit das Eintreten eines Schadensfalls toleriert wird bzw. nicht mehr akzeptabel ist.

Die Risikoakzeptanz für die identifizierten möglichen Schadensfälle wird beispielsweise in Form einer Risikoakzeptanzmatrix dokumentiert. Die Matrix ist eine Vorgabe des Auftraggebers, in der er festlegt, bei welcher Schadensklasse und welcher Eintrittswahrscheinlichkeit er welche Risikoklasse akzeptiert.«

6 Lebenszyklusanalyse und Gesamtsystemarchitektur

»Ausgehend von den Anforderungen werden ein grober Entwurf des Gesamtsystems erstellt [...]. In der Gesamtsystemarchitektur wird das zentrale System mit den Unterstützungssystemen identifiziert und festgelegt, für welche Systeme ein Logistisches Unterstützungskonzept zu erstellen ist. Grundlage sind die funktionalen und nicht-funktionalen Anforderungen sowie die Skizze der Gesamtsystemarchitektur in den Anforderungen. [...]

Die Gesamtsystemarchitektur wird hinsichtlich der möglichen Verwendung von Fertigprodukten geprüft.«

7 Schnittstellenübersicht

»Zur Darstellung der Zusammenhänge zwischen dem System und seiner Umgebung wird eine Schnittstellenübersicht erstellt. Ausgehend vom System werden Schnittstellen zum Anwender, zu den Unterstützungssystemen, zur Logistik und zu Nachbarsystemen identifiziert und in geeigneter Form dokumentiert.«

IV 20 Anforderungsschablonen

8 Lieferumfang

»Es sind alle Gegenstände und Dienstleistungen aufzulisten, die während des Projektverlaufs oder bei Abschluss des Projektes vom Auftragnehmer an den Auftraggeber zu liefern sind. Jede Lieferung erfordert eine Abnahmeprüfung.«

9 Abnahmekriterien

»Die Abnahmekriterien legen fest, welche Kriterien die Lieferung zu erfüllen hat, um den Anforderungen im Lastenheft zu entsprechen. Die Beschreibung der Abnahmekriterien wird aus den Anforderungen (Lastenheft) übernommen.

Die Erfüllung der Abnahmekriterien wird im Rahmen der Eingangsprüfung beim Auftraggeber festgestellt.«

10 Anforderungsverfolgung zu den Anforderungen (Lastenheft)

»Im Rahmen der Anforderungsverfolgung zum Lastenheft wird zusammenfassend die Zuordnung der funktionalen und nicht-funktionalen Anforderungen aus dem Lastenheft zu Anforderungen im Pflichtenheft dargestellt. Die bidirektionale Verfolgbarkeit muss dabei sichergestellt werden. Die Darstellung kann beispielsweise anhand einer Matrix erfolgen.«

11 Anforderungsverfolgung

»Im Rahmen der Anforderungsverfolgung wird im Pflichtenheft zusammenfassend die Zuordnung der funktionalen und nicht-funktionalen Anforderungen zu Elementen der Gesamtsystemarchitektur (System, Unterstützungssystem, Segment oder Logistik) dargestellt. Die bidirektionale Verfolgbarkeit muss dabei sichergestellt werden. Die Darstellung kann beispielsweise anhand einer Matrix erfolgen.«

12 Abkürzungsverzeichnis

13 Literaturverzeichnis

14 Abbildungsverzeichnis

20.3 Schablonen für Lastenheft, Pflichtenheft und Glossar

Für kleinere Projekte werden hier zwei Anforderungsschablonen vorgestellt, die sich in der Praxis bewährt haben. Es gibt eine Schablone für ein **Lastenheft** und eine Schablone für ein **Pflichtenheft**. Das Lastenheft (= grobes Pflichtenheft) kann als Grundlage für das Pflichtenheft verwendet werden. Das Pflichtenheft kann aber auch als einziges Anforderungsdokument erstellt werden. Als weiteres Dokument ist auf jeden Fall ein **Glossar** zu erstellen.

Bewährt hat sich ein zweistufiges Vorgehen: Zuerst ein Lastenheft erstellen und dann darauf aufbauend als Detaillierung das Pflichtenheft schreiben. Wenn immer möglich, sollen Basiskonzepte zur Spezifikation verwendet werden, um natürlichsprachliche Anforderungen zu vermeiden.

Tipp

Die einzelnen Kapitel können in beliebiger Reihenfolge erstellt werden. Die Kapitelgliederung ist aber so aufgebaut, dass eine gute Lesbarkeit und Verständlichkeit sichergestellt ist, d. h., wenn das Dokument von oben nach unten gelesen wird, dann bekommt der Leser einen guten Eindruck von den Anforderungen an das zu entwickelnde System.

Es wird davon ausgegangen, dass die Erfassung der Anforderungen mit einem Textsystem erfolgt.

Jede Anforderung beginnt mit einem Buchstaben oder einer Buchstabenkombination, gefolgt von einer Zahl, eingeschlossen in Schrägstriche. Der Anforderungstyp wird durch einen Buchstaben gekennzeichnet, z. B. F für eine funktionale Anforderung. Für Anforderungen, die im Lastenheft stehen, wird als erster Buchstabe nach dem Schrägstrich ein L verwendet. Fehlt das L, dann handelt es sich um eine Anforderung im Pflichtenheft. Wird im Pflichtenheft auf das Lastenheft Bezug genommen, dann wird am Anfang der Anforderung die Anforderungsidentifikation des Lastenhefts in runden Klammern aufgeführt. Am Anfang werden die Zahlen in Zehnerschritten durchnummeriert, sodass spätere fachlich dazugehörige Anforderungen zwischengefügt werden können. Sind Anforderungen oder Attribute noch nicht definiert, dann wird dies durch die englische Abkürzung TBD (*to be defined*) gekennzeichnet.

Notation

/F10/ (/LF30/) Hier steht eine funktionale Anforderung (F) im Pflichtenheft, die auf eine Anforderung im Lastenheft referenziert.

Beispiel

Wird eine Anforderung in einem anderen Produkt referenziert, dann wird der Produktname davor gesetzt, z. B. SemOrg /F10/.

Die **Managementattribute** Priorität, Stabilität, Kritikalität und Entwicklungsrisiko einer Anwendung werden am Anfang des Dokuments mit Voreinstellungswerten (einschl. der Angabe des Wertebereichs) aufgeführt. Weicht eine Anforderung von diesen Voreinstellungswerten ab, dann wird das abweichende Attribut mit dem entsprechenden Wert bei dem betreffenden Attribut explizit aufgeführt. Aufwand und Konflikte sind bei der jeweiligen Anforderung aufzuführen (siehe »Anforderungsattribute«, S. 479).

Am Anfang des Dokuments:

Priorität aus Auftraggebersicht = {hoch, *mittel*, niedrig}

Priorität aus Auftragnehmersicht = {hoch, *mittel*, niedrig}

Stabilität = {fest, *gefestigt*, volatil}

Beispiel

IV 20 Anforderungsschablonen

Kritikalität = {hoch, mittel, *niedrig*, keine}

Entwicklungsrisiko = {hoch, mittel, *niedrig*}

In einer Anforderung:

/F30/ Das System muss die Kundendaten permanent speichern. Priorität aus Auftraggebersicht = hoch, Stabilität = fest.

Dokumentationsattribute wie Autor, Quelle, Version und Bearbeitungsstatus werden einmal am Anfang des Dokuments beschrieben. Ist es notwendig, dies pro Anforderung zu tun, dann kann diese Information auch hinter jede Anforderung geschrieben werden. Alternativ können auch nur die Abweichungen von den Angaben am Anfang des Dokuments an der jeweiligen Anforderung dokumentiert werden.

Die **Inhaltsattribute** sind bei jeder Anforderung anzugeben, soweit sie schon angebbbar sind.

Das Lastenheft

Das Lastenheft besitzt folgende Kapitelgliederung: **1 Visionen und Ziele**

Beschreibung der Visionen und Ziele, die durch den Einsatz des Systems erreicht werden sollen:

/LV10/ Vision 1 des Systems usw.

/LZ10/ Ziel 1 des Systems usw.

2 Rahmenbedingungen

Beschreibung der Anwendungsbereiche und Zielgruppen:

/LR10/ Anwendungsbereich 1 des Systems usw.

3 Kontext und Überblick

Festlegung der relevanten Systemumgebung (Kontext) und Überblick über das System:

/LK10/ Kontext 1 des Systems usw.

4 Funktionale Anforderungen

Die Kernfunktionalität des Systems ist aus Auftraggebersicht auf oberster Abstraktionsebene zu beschreiben. Auf Detailbeschreibungen ist zu verzichten:

/LF10/ Funktion 1 des Systems usw.

Wenn bereits möglich, sind die funktionalen Anforderungen nach Statik, Dynamik und Logik zu gruppieren. Erfolgt die Spezifikation natürlichsprachlich, dann sollten Sprachschablonen verwendet werden (siehe »Natürlichsprachliche Anforderungen«, S. 481).

5 Qualitätsanforderungen

Es sollte anhand einer Tabelle eine Qualitätszielbestimmung für das System anhand der Tab. 20.3-1 (hier mit Beispielangaben) vorgenommen werden.

Einzelne Qualitätsanforderungen können wie folgt festgelegt werden:

Systemqualität	sehr gut	gut	normal	nicht relevant
Funktionalität		X		
Zuverlässigkeit			X	
Benutzbarkeit		X		
Effizienz			X	
Wartbarkeit			X	
Portabilität			X	

Tab. 20.3-1:
Qualitätsziel-
bestimmung.

/LQB10/ Qualitätsanforderung zur Benutzbarkeit des Systems
/LQE10/ Qualitätsanforderung zur Effizienz des Systems usw.

Das Pflichtenheft

Das Pflichtenheft ist eine Verfeinerung und Präzisierung des Lastenhefts. Es besitzt folgende Kapitelgliederung: **1 Visionen und Ziele**
Beschreibung der Visionen und Ziele, die durch den Einsatz des Systems erreicht werden sollen:

/V10/ Vision 1 des Systems usw.

/Z10/ Ziel 1 des Systems usw.

2 Rahmenbedingungen

Beschreibung der organisatorischen Rahmenbedingungen: Anwendungsbereiche, Zielgruppen, Betriebsbedingungen

/R10/ Anwendungsbereich 1 des Systems usw.

/R20/ Zielgruppe 1 des Systems usw.

/R30/ Physikalische Umgebung des Systems

/R40/ Tägliche Betriebszeit des Systems

/R50/ Ständige Beobachtung des Systems durch Bediener oder unbeaufsichtigter Betrieb

Bei der technischen Produktumgebung sind folgende Festlegungen zu treffen:

/R60/ Eingesetzte Software auf der Zielmaschine

/R70/ Eingesetzte Hardware(komponenten) einschl. Konfiguration auf der Zielmaschine

/R80/ Organisatorische Randbedingungen und Voraussetzungen

Bei den Anforderungen an die Entwicklungsumgebung sind folgende Festlegungen zu treffen:

/R90/ Software auf dem Entwicklungssystem

/R100/ Hardware des Entwicklungssystems

/R110/ Orgware des Entwicklungssystems

/R120/ Entwicklungsschnittstellen

3 Kontext und Überblick

Festlegung der relevanten Systemumgebung (Kontext) und Überblick über das System:

/K10/ Kontext 1 des Systems usw.

IV 20 Anforderungsschablonen

4 Funktionale Anforderungen

Die Kernfunktionalität des Systems ist aus Auftraggebersicht auf oberster Abstraktionsebene zu beschreiben. Auf Detailbeschreibungen ist zu verzichten:

/F10/ Funktion 1 des Systems usw.

Wenn bereits möglich, sind die funktionalen Anforderungen nach Statik, Dynamik und Logik zu gruppieren. Erfolgt die Spezifikation natürlichsprachlich, dann sollten Sprachschablonen verwendet werden (siehe »Natürlichsprachliche Anforderungen«, S. 481).

5 Qualitätsanforderungen

Es sollte anhand einer Tabelle eine verfeinerte Qualitätszielbestimmung für das System vorgenommen werden. Es eignen sich dazu die Qualitätsmerkmale der ISO/IEC 9126-1 (siehe »Box: Qualitätsmerkmale nach ISO/IEC 9126-1«, S. 468). Ein Beispiel für die Anwendung zeigt die »Fallstudie: SemOrg – Die Spezifikation«, S. 107.

Einzelne Qualitätsanforderungen können unter Bezug auf die ISO/IEC 9126-1 wie folgt festgelegt werden:

/QFS10/ Qualitätsanforderung zur **Funktionalität (Sicherheit)** des Systems

/QBE10/ Qualitätsanforderung zur **Benutzbarkeit (Erlernbarkeit)** des Systems usw.

6 Abnahmekriterien

Abnahmekriterien legen fest, wie Anforderungen bei der Abnahme auf ihre Realisierung überprüft werden können (siehe »Abnahmekriterien«, S. 471).

7 Subsystemstruktur (optional) Hier wird eine Aufgliederung des zu entwickelnden Systems beschrieben, wenn die Entwicklung inkrementell (Kernsystem, Ausbaustufe 1, Aufbaustufe 2 usw.) erfolgen soll.

Das Glossar

Das Glossar besteht aus Glossarbegriffen mit anschließender Erklärung des jeweiligen Begriffs. Wird auf andere Glossarbegriffe verwiesen, dann wird dies durch einen Pfeil vor dem Begriff, auf den verwiesen wird, angegeben. Übersetzungen in andere Sprachen sowie Synonyme werden in Klammern hinter dem Glossarbegriff angegeben (siehe »Natürlichsprachliche Anforderungen«, S. 481).

Beispiele Die Anwendung dieser Schablonen zeigen folgende Fallstudien:

■ »Fallstudie: SemOrg – Die Spezifikation«, S. 107

■ »Fallstudie: Fensterheber – Die Spezifikation«, S. 117

20.4 Schablonen für agile Entwicklungen

In der agilen Softwareentwicklung (siehe »Lehrbuch der Softwaretechnik – Softwaremanagement«) ist es das Ziel mit möglichst wenig Dokumenten – außer dem Code – auszukommen. Ein Prinzip besteht darin, den Kunden oder einen Kundenrepräsentanten in das Entwicklungsteam zu integrieren. Außerdem ist der RE-Prozess entwicklungsbegleitend (siehe »Der Requirements Engineering-Prozess«, S. 449).

Anforderungen werden in Form von sogenannten *User Stories* (»Benutzergeschichten«) vom Kunden erhoben. Die Kunden sollen die Anforderungen selbst aufschreiben. Ein Entwickler kann dem Kunden aber Fragen stellen, um ihm beim Konzipieren der Anforderungen zu unterstützen.

Eine *User Story* soll Dinge beschreiben, die das System für den Kunden tun soll. *A User Story is »a concise, written description of a piece of functionality that will be valuable to a user (or owner) of the software.«* [Wate08]. Im einfachsten Fall besteht eine *User Story* aus 1 bis 2 umgangssprachlichen Sätzen in der Geschäftsterminologie des Kunden.

Interessenten und Kunden müssen sich über das Web über Seminare und Veranstaltungen informieren können.

Beispiel:
SemOrg

In der Regel wird jede *User Story* auf einer Karteikarte (z. B. im Format 8 x 13 cm oder im Postkartenformat) mit folgenden zusätzlichen Angaben notiert:

- Nr. oder Identifikation der *User Story* z. B. durch einen Bezeichner.
- Priorität (vergeben durch den Kunden) (von 1 bis 10, wobei 1 die höchste Priorität ist, oder hoch, mittel, niedrig).
- Aufwand zur Implementierung der Anforderung (z. B. in Stunden für ein 2er-Team). Die Schätzung erfolgt durch die Entwickler.

Die Abb. 20.4-1 zeigt eine *User Story* mit diesen Angaben auf einer »Postkarte«. Wie die Abb. 20.4-1 zeigt, können sich Prioritäten ändern.

Beispiel:
SemOrg

Ein weiteres Beispiel zeigt die Abb. 20.4-2 [Lang02].

Beispiel

Eine *User Story* sollte so genau formuliert sein, dass eine Aufwandsschätzung durch die Entwickler möglich ist. Wenn die Anforderung zu umfangreich ist, dann wird sie auf mehrere *User Stories* aufgeteilt. Eine *User Story* sollte in 1 bis 3 Wochen »idealer Entwicklungszeit« implementiert werden können. Wenn eine *User Story* implementiert werden soll, dann sollen die Entwickler die Möglichkeit

IV 20 Anforderungsschablonen

Abb. 20.4-1:
Beispiel für eine
User Story.

1001
Interessenten und Kunden müssen sich
über das Web über Seminare und
Veranstaltungen informieren können.

Priorität: 4
Aufwand: 20 Stunden

Story 26
05-Oct-01

Edit SP Details Screen

Add "Cancel" - Button to undo
changes and return to previous
page.

TC: Check that browser returns to correct
previous change

Estimated: 2h
Actual:

Abb. 20.4-2:
Beispiel für eine
User Story.

haben, mit dem Kunden zu sprechen, um Details festzulegen und Hintergrundwissen zu bekommen. Auch kann die *User Story* sich in-
zwischen geändert haben.

Abnahmetests Bei der Erstellung einer *User Story* oder zu einem späteren Zeit-
punkt sollten ein oder mehrere (automatisierte) Abnahmetests for-
muliert werden, die für den Kunden und die Entwickler eine Über-
prüfung erlauben, ob die *User Story* korrekt implementiert wurde.

20.4 Schablonen für agile Entwicklungen IV

In [Cohn04] wird folgende Textschablone für die Formulierung einer *User Story* vorgeschlagen: Mehr »formal«

»As a [role] I want [something] so that [benefit].«

In [Wate08b] wird eine ähnliche Textschablone vorgestellt:

»As a [user role], I want to [goal], so I can [reason].«

Dort wird auch vorgeschlagen, die Karteikarte wie folgt aufzuteilen (3 C's):

- **Card section:** Auf der Vorderseite der Karte steht oben der Name und die Beschreibung der Anforderung, irgendeine Referenznummer, der geschätzte Aufwand usw.
- **Conversation section:** Enthält mehr Informationen über die Anforderung.
- **Confirmation section:** Auf der Rückseite der Karte stehen Abnahmetests für die Anforderung.

Die Abb. 20.4-3 zeigt ein Beispiel für die Vorderseite der Karte und die Abb. 20.4-4 die Rückseite der Karte [Wate08b].

#0001	USER LOGIN	Fibonacci Size # 3
As a [registered user], I want to [log in], so I can [access subscriber content].		

The diagram shows a 'User Login' form with the following elements and annotations:

- Username:** A text input field. Annotation: 'User's email address. Validate format.'
- Password:** A text input field.
- Remember me:** A checkbox. Annotation: 'Store cookie if ticked and login successful.'
- Login:** A button. Annotation: 'Authenticate against SRS using new web service.'
- [message]:** A placeholder for a message. Annotation: 'Display message here if not successful. (see confirmation scenarios over)'
- Forgot password?:** A link. Annotation: 'Go to forgotten password page.'

Abb. 20.4-3:
Beispiel für eine
formalisierte User
Story (Vorderseite
der Karteikarte).

IV 20 Anforderungsschablonen

Abb. 20.4-4:
Beispiel für eine
formalisierte User
Story (Rückseite
der Karteikarte).

Confirmation

1. Success – valid user logged in and referred to home page.
 - a. 'Remember me' ticked – store cookie / automatic login next time.
 - b. 'Remember me' not ticked – force login next time.
2. Failure – display message:
 - a) "Email address in wrong format"
 - b) "Unrecognised user name, please try again"
 - c) "Incorrect password, please try again"
 - d) "Service unavailable, please try again"
 - e) Account has expired – refer to account renewal sales page.

Alternativ zu *User Stories* werden *Use Cases* (siehe »Geschäftsprozesse und *Use Cases*«, S. 250) und Szenarien (siehe »Szenarien«, S. 332) eingesetzt.

QS-Checkliste: Eine gute *User Story* sollte folgende Eigenschaften besitzen (INVEST-Modell, siehe z. B. [Wate08b]):



- **Independent:** Eine Anforderung sollte soweit wie möglich unabhängig von anderen Anforderungen sein.
- **Negotiable:** Eine *User Story* ist verhandelbar. Damit ist gemeint, dass sie keine Details enthält und auch nicht als Vertrag anzusehen ist. Sie stellt vielmehr eine Erinnerung an die Entwickler dar, zum Implementierungszeitpunkt mit dem Kunden die Details zu besprechen.
- **Valuable:** Eine *User Story* soll einen Wert für den Kunden darstellen. Sie beschreibt eine Funktionalität, keine Aufgaben.
- **Estimatable:** Der Aufwand zur Implementierung der Anforderung soll schätzbar sein.
- **Small:** Die Anforderung soll klein, aber nicht zu klein und nicht zu groß sein. Sie soll in 2–3 Mitarbeiterwochen implementierbar sein.
- **Testable:** Die Anforderung muss so beschrieben sein, dass sie testbar ist.

RE-Prozess Am Anfang einer Entwicklung werden *User Stories* aufgeschrieben, um den Umfang des Systems zu identifizieren. Während der Entwicklung werden neue Anforderungen aufgestellt, vorhandene *User Stories* unterteilt, Prioritäten neu gesetzt und *User Stories* entfernt, die nicht mehr zum Umfang des Systems gehören. Wird eine *User Story* implementiert, dann wird vorher je nach Anforderungsart oft eine Bildschirmskizze oder ein UML-Aktivitätsdiagramm erstellt. Zwischen 60 und 100 *User Stories* bilden ein Release.

20.4 Schablonen für agile Entwicklungen IV

- + Leicht zu erstellen.
- + Enger Kundenkontakt.
- Kunden sind keine Profis für das Schreiben von Anforderungen.
- Überblick über das gesamte zu entwickelnde System ist schwer zu erhalten.
- Starke Abhängigkeit von kompetenten Entwicklern.
- Schätzung des Aufwands schwierig.
- Nichtfunktionale Anforderungen und Restriktionen sind schwer zu berücksichtigen [Ambl08].

Bewertung

21 Anforderungen ermitteln und spezifizieren

Bevor mit der Ermittlung der Anforderungen begonnen werden kann, muss zunächst geklärt werden, wer die Anforderungen stellt. Es reicht *nicht* aus, nur den Auftraggeber, d. h. den der bezahlt, zu befragen. Die erste Aktivität muss daher darin bestehen, die *Stakeholder* zu identifizieren und einzubinden.

Anschließend oder parallel dazu muss das Projektumfeld bestimmt werden. Es muss nach zu berücksichtigenden Normen, Standards und Gesetzen recherchiert werden. Diese müssen gesammelt, kommentiert und abgelegt werden.

Nach diesen Vorarbeiten können die Anforderungen mit Hilfe verschiedener Ermittlungstechniken ermittelt werden. Es entstehen Notizen, Interviewprotokolle, Audioaufzeichnungen, Videos usw. Sie werden abgelegt und verwaltet. Danach oder parallel werden die Anforderungen spezifiziert, d.h die Anforderungsspezifikation wird schrittweise erstellt. Die Abb. 21.0-1 gibt einen Überblick über die Aktivitäten und die Artefakte.

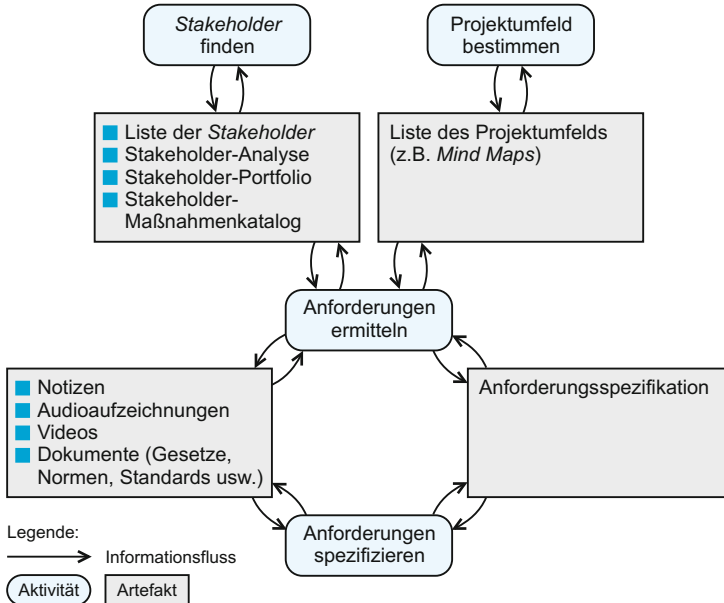
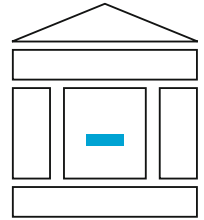


Abb. 21.0-1:
Übersicht über das
Ermitteln und
Spezifizieren der
Anforderungen.

IV 21 Anforderungen ermitteln und spezifizieren

Stakeholder finden

Eine der ersten und wichtigsten Aufgaben besteht darin, die relevanten Beteiligten, Betroffenen, Nutzer und Interessierten an der Software(entwicklung) – kurz **Stakeholder** genannt – zu identifizieren. Diese Personen oder Personengruppen beeinflussen die Softwareentwicklung, aber auch die Einführung und den Betrieb der fertiggestellten Software, mit unterschiedlicher Intensität, sowohl in positiver als auch in negativer Hinsicht. Es ist daher wichtig, diese Personen oder Personengruppen frühzeitig in den RE-Prozess einzubinden. Sie liefern direkt oder indirekt Informationen über die Ziele, die Rahmenbedingungen, den Kontext, die Anforderungen und auch die Risiken des Produkts bzw. Systems.

Permanente Aufgabe Beim Start des RE-Prozesses werden noch nicht alle *Stakeholder* gefunden – es muss daher während der Softwareentwicklung darauf geachtet werden, ob es weitere *Stakeholder* gibt. Werden wichtige *Stakeholder* vergessen, dann können z.B. bei der Inbetriebnahme neue Anforderungen gestellt werden, die aufwendig zu realisieren sind. Übersehene oder übergangene *Stackholder* können sich herabgesetzt fühlen und aus verletztem Stolz gegen das Produkt arbeiten (siehe auch [SLM07]).

Bewertung Um einen Überblick zu bekommen, ist es hilfreich, die Beziehung der identifizierten *Stakeholder* zum Produkt (positiv, neutral, negativ) und ihre Macht zu bewerten. Dass alle *Stakeholder* zielgerichtet und konstruktiv auf den Entwicklungserfolg hin arbeiten, ist eher die Ausnahme als die Regel.

Beispiel: Die Tab. 21.0-1 zeigt eine Stakeholder-Analyse.
SemOrg

Stakeholder	Erwartung	Klima/Stimmung	Bedeutung/Macht
Geschäftsführer	Hat als Auftraggeber & Geldgeber großes Interesse das Produkt zum Erfolg zu bringen.	+	5
Veranstaltungsbetreuer	Hat Angst zu viel am Computer arbeiten zu müssen.	–	2
Seminarsachbearbeiter	Kommt auch ohne die neue Software gut aus.	o	3
Legende		+ = positiv, o = neutral, – = negativ	5 = hoch, 1 = gering

Tab. 21.0-1: Eine Portfoliodarstellung ermöglicht es, die Stakeholder-Analyse grafisch zu veranschaulichen (Abb. 21.0-2).
Stakeholder-Analyse für die Fallstudie SemOrg.

Die Einordnung der *Stakeholder* in das Portfolio ist eine subjektive Abbildung, die die Einschätzung der Projektleitung widerspiegelt. Die *Stakeholder*, die im rechten oberen Quadranten eingetragen

21 Anforderungen ermitteln und spezifizieren IV

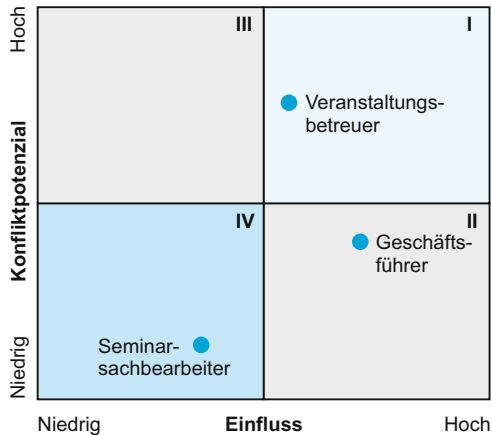


Abb. 21.0-2: Stakeholderportfolio.

sind (hoher Einfluss & hohes Konfliktpotenzial), müssen ständig im Blickfeld des Projektleiters sein. Für diese *Stakeholder* sollten Maßnahmen formuliert werden, um sie positiv zu beeinflussen.

Bei den *Stakeholdern* lassen sich verschiedene Persönlichkeitstypen unterscheiden, deren Kenntnis es erleichtert, Kommunikations- und Verständnisprobleme zu reduzieren [WSH07].

Persönlichkeitstypen

In [WMH07] wird beschrieben, welches Risiko der potenzielle negative Einfluss von *Stakeholdern* auf ein Projekt haben und wie man diese Risiken bewerten kann.

Risiken

Überlegen Sie, durch welche Maßnahmen man das Konfliktpotenzial senken könnte.



Auch die Beziehungen zwischen *Stakeholdern* können für das Projekt eine Rolle spielen. Beispielsweise können zwei *Stakeholder* entgegengesetzte Ziele anstreben. In solchen Fällen ist ebenfalls zu überlegen, wie die Konflikte gelöst werden können.

Die Projektleitung muss überlegen, wie sie die *Stakeholder* geeignet in das Projekt einbindet. Ziel muss es sein, aus Betroffenen Beteiligte zu machen.

Einbindung

Überlegen Sie, durch welche Maßnahmen *Stakeholder* eingebunden werden können.

Frage

Folgende Maßnahmen helfen, *Stakeholder* zu Beteiligten zu machen:

Antwort

- *Stakeholder* in die Kommunikation über das Projekt einbinden.
- *Stakeholder* Verantwortung und Aufgaben übertragen.
- *Stakeholder* zu wichtigen Besprechungen einladen.

Überlegen Sie, welche Informationen Sie über jeden *Stakeholder* dokumentieren sollten.

Frage

Das folgende Beispiel SemOrg zeigt, welche Informationen nützlich sind [Rupp07, S. 96].

Antwort

IV 21 Anforderungen ermitteln und spezifizieren

- Beispiel ■ **Rolle des Stakeholder:** Seminarsachbearbeiter
SemOrg ■ **Beschreibung:** Weiß, welche Anforderungen das Produkt für die Seminarplanung erfüllen muss.
■ **Konkrete Vertreter:** Frau Sommer, Tel. 0231/12345, E-Mail: Sommer@Teachware.de
■ **Verfügbarkeit:** 20 % verfügbar
■ **Wissensgebiet:** Kennt alle Vorgänge in der Teachware GmbH.
■ **Begründung:** Benutzerin des Produkts, Meinungsführerin in der Belegschaft

- Ergebnisse Als Ergebnisse der Stakeholder-Identifizierung sollten folgende Artefakte vorliegen:
■ Stakeholder-Analyse
■ Stakeholder-Portfolio
■ Stakeholder-Maßnahmenkatalog

Projektumfeld ermitteln

Definition Das **Projektumfeld** ist »das Umfeld, indem ein Projekt entsteht und durchgeführt wird, das das Projekt beeinflusst und von dem es beeinflusst wird« (DIN 69904).

Neben den *Stakeholdern* gehören zum Projektumfeld alle Elemente, die ein Projekt wesentlich beeinflussen, wie Vorgaben, Gesetze, Standards, Normen, ökologische, ökonomische, gesellschaftliche und kulturelle Einflüsse, Gesetzgebung von Ländern oder kulturelle Besonderheiten.

Kreativitäts- Um nichts im Projektumfeld zu übersehen, haben sich **Kreativitätstechniken** im Projektteam, wie z. B. **Brainstorming**, bewährt¹.

Mind Map Zur Darstellung des Projektumfelds eignen sich *Mind Maps*. In *Mind Maps* kann man Informationen sammeln und strukturieren. Begriffe und Halbsätze werden hier an Haupt- und Unteräste geschrieben. Die verschiedenen Gliederungsebenen und wichtige Inhalte kennzeichnet man durch besondere Schriftauszeichnungen, Farben, Bilder und Symbole. Eine *Mind Map* liefert eine Übersicht über ein Thema und seine Bestandteile und regt zudem zu neuen Einfällen an. Im Internet gibt es kostenlose Min Map-Programme, z. B. FreeMind (http://freemind.sourceforge.net/wiki/index.php/Main_Page).

- Ergebnis Als Ergebnis der Projektumfeld-Ermittlung sollte folgendes Artefakt vorliegen:
■ Liste von Elementen des Projektumfelds, alternativ als *Mind Map*.

¹Siehe z. B. das Buch [Schr05] mit E-Learning-Kurs.

Anforderungen ermitteln und dokumentieren

Sind die ersten *Stakeholder* identifiziert und ist das Projektumfeld ermittelt, dann können die Anforderungen, die die einzelnen *Stakeholder* haben, ermittelt und dokumentiert werden.

Zunächst eine Warnung: Erwarten Sie nicht, dass Ihnen die *Stakeholder* eindeutige und präzise Anforderungen auf dem richtigen Abstraktionsniveau nennen. Erwarten Sie vielmehr schwammige, unvollständige, widersprüchliche Anforderungen auf unterschiedlichem Abstraktionsniveau – von Beispielen bis hin zu globalen Aussagen.

Dies ist in der Regel kein böser Wille. Der *Stakeholder* hat keine fachliche Lösung im Kopf – er knüpft an seine Erfahrungen an und kennt oft nicht die Möglichkeiten, die eine heutige Software bieten kann. Oft orientiert er sich an vorhandenen Ist-Systemen und hat Schwierigkeiten, neue Möglichkeiten in Betracht zu ziehen.

»On half of the participating projects, senior management, project managers, and system analysts defined at least the initial requirements. [...] Involving stakeholders early also resulted in an increased understanding of the RE process being used« [HoLe01, S. 61].



Empirie
Zitat

Es gibt eine Vielzahl von **Ermittlungstechniken**, um die Anforderungen zu ermitteln. Welche Technik geeignet ist, hängt von den Randbedingungen des Projekts ab.

Ermittlungs-
techniken

Am meisten eingesetzt werden **Befragungstechniken**, die gut geeignet sind, das explizite Wissen der *Stakeholder* zu ermitteln.

Bei einem **strukturierten Interview** befragt der *Requirements Engineer* den *Stakeholder* anhand einer Anforderungsschablone, z.B. beginnend mit den Visionen und Zielen bis hin zu den Abnahmekriterien (siehe »Anforderungen und Anforderungsarten«, S. 455). Bei einem persönlichen Interview kann der *Requirements Engineer* den Gesprächsverlauf individuell anpassen, Rückfragen beantworten, Beispiele aufzählen und bei unklaren Aussagen nachfragen. Da *Stakeholder* in der Regel schneller sprechen, als der *Requirements Engineer* mitschreiben kann, sollte die Erlaubnis eingeholt werden, eine Audioaufnahme mitzuschneiden. Nachteilig ist, dass persönliche Interviews – zumindest wenn viele *Stakeholder* vorhanden sind – sehr zeitaufwendig sind.

Strukturierte
Interviews

Alternativ dazu ist es möglich, mehrere *Stakeholder* gleichzeitig zu interviewen. Dadurch spart man Zeit und eventuelle unterschiedliche Auffassungen werden sofort sichtbar. Der *Requirements Engineer* befindet sich dann in der Rolle eines **Moderators**, der zielgerichtet, aber nicht zu dominant das Gespräch leitet². Dazu ist jedoch viel Erfahrung erforderlich, um voranzukommen und nicht zwischen die Fronten unterschiedlicher Auffassungen zu geraten.

Moderatorrolle

²Siehe z. B. das Buch [Mott09] mit E-Learning-Kurs

IV 21 Anforderungen ermitteln und spezifizieren



Überlassen Sie niemals einem *Stakeholder* die Federführung bei der Ermittlung der Anforderungen. Er weiß nicht, welche Informationen Sie alle benötigen und er ist kein RE-Spezialist.

Weitere Techniken

Weitere mögliche **Befragungstechniken** sind:

- **Fragebögen:** Die *Stakeholder* erhalten einen gedruckten oder digitalen Fragebogen, den Sie ausfüllen.
- **Selbstaufschreibung:** Die *Stakeholder*, die das Ist-System nutzen, das abgelöst werden soll, können ihre Tätigkeiten und Arbeitsabläufe sowie ihre Wünsche an das neue System aufschreiben.
- **Stakeholder im Projektteam** (*On Site Customer*): Ein Repräsentant des Auftraggebers arbeitet im Projektteam vor Ort mit. Durch diese ständige Verfügbarkeit können Anforderungen und Fragen direkt geklärt werden. Bei agilen Entwicklungsmodellen ist dies in der Regel so (siehe »Schablonen für agile Entwicklungen«, S. 497).

Beobachtung

Nicht immer sind *Stakeholder* in der Lage, sprachlich Anforderungen zu formulieren, oder sie haben keine Zeit, bei der Anforderungsermittlung mitzuwirken. Bei Beobachtungstechniken werden *Stakeholder*, die Geschäftsprozesse ausführen, durch den *Requirements Engineer* beobachtet. Der *Stakeholder* ist dabei passiv und wird nur beobachtet oder ist aktiv und zeigt und erklärt dem *Requirements Engineer*, was er tut. Es besteht allerdings die Gefahr, dass das Ist-System dokumentiert wird. Der *Requirements Engineer* muss daher darauf achten, ineffiziente Prozesse zu erkennen und Verbesserungen vorzuschlagen. Bei allen Beobachtungstechniken ist zu berücksichtigen, dass sich das Verhalten des Beobachteten alleine durch die Beobachtungssituation ändert. Zwei Beobachtungstechniken werden oft eingesetzt:

- Die **Feldbeobachtung:** Die Arbeit des Benutzers wird bezogen auf seine Tätigkeiten, seine Handgriffe und seine Zeitabläufe – evtl. mit Videoaufzeichnung (siehe z. B. [JiLu06]) – beobachtet.
- Die **Lehrlingsrolle** (*apprenticing*): Der *Requirements Engineer* erlernt unter der Anleitung des Benutzers dessen Tätigkeiten. Auch schwer beobachtbare Arbeitsabläufe werden dadurch transparent.

Altsysteme
vorhanden

Soll durch eine Softwareentwicklung ein im Einsatz befindliches Altsystem (*legacy system*) abgelöst werden – was heute der Regelfall sein dürfte – dann kann eine Analyse des Altsystems wichtige Hinweise für die Anforderungen an das neue System geben. In der Regel reicht ein Verstehen des Altsystems als *black box* aus. Anhand der Bedienung und anhand von Benutzerhandbüchern können Informationen über Prozessabläufe, Ein- und Ausgabedaten und Funktionen ermittelt werden. Diese Daten können z. B. genutzt werden, um ein objektorientiertes Analysemodell zu erstellen (siehe »Beispiel: Objektorientierte Analyse«, S. 548). Arbeitet das Altsystem mit einer

relationalen Datenbank, dann kann es auch hilfreich sein, das Datenbankmodell in Form von SQL-Tabellen zu analysieren (*white box*). Daraus kann ebenfalls ein OOA-Modell abgeleitet werden.

Wichtig ist, dass man aus Aufwandsgründen keine detaillierte Analyse vornimmt und sich auf die wesentlichen Gesichtspunkte konzentriert. Ziel ist es in der Regel nicht, das Altsystem graduell zu verbessern, sondern ein neues System zu konzipieren, das die heutigen Techniken und Erkenntnisse nutzt.

Ist es vom Aufwand und vom Risiko her nicht sinnvoll, Teile eines Altsystems zu ersetzen, dann kann das Altsystem verpackt werden (*wrapping*), d.h. Teilsysteme oder Teilkomponenten werden durch eine Software-Schicht nach außen hin so verpackt, dass eine Nutzung durch das neue System möglich ist.

Jede Organisation, die heute Software entwickelt, sollte über ein Wiederverwendbarkeitsarchiv verfügen (siehe dazu »Das komponentenbasierte Modell« im »Lehrbuch der Softwaretechnik – Softwaremanagement«). Ein solches Archiv sollte nicht nur über fertig implementierte Komponenten verfügen, sondern auch über Anforderungsartefakte bisheriger Entwicklungen. Diese Artefakte können dann für Neuentwicklungen gesichtet und wiederverwendet werden.

Wieder-
verwendung

Gehört das zu entwickelnde Produkt zu einer Produktlinie, dann basieren die Anforderungen sowieso auf den Anforderungsartefakten der Produktlinie (siehe dazu »Das Modell für Produktfamilien/Produktlinien« im »Lehrbuch der Softwaretechnik – Softwaremanagement«).

Produktlinien

Um auf innovative Ideen zu kommen und um eine erste Vision von einem System zu entwickeln, eignen sich Kreativitätstechniken. Einen Überblick über solche Techniken finden Sie im »Lehrbuch der Softwaretechnik Softwaremanagement«.

Neue Ideen

Das Ergebnisse der Anforderungsermittlung können folgende Artefakte sein:

Ergebnisse

- Notizen
- Interviewprotokolle
- Audioaufzeichnungen
- Videos usw.

Anforderungen spezifizieren

Die Ergebnisse der Anforderungsermittlung können nun systematisch ausgewertet und schrittweise in die verwendete(n) Anforderungsschablone(n) unter Ergänzung von Anforderungsattributen (siehe »Anforderungsattribute«, S. 479) eingetragen werden. Bei einer agilen Softwareentwicklung entstehen *User Stories*. Fallen bei

IV 21 Anforderungen ermitteln und spezifizieren

der Anforderungsermittlung bereits Anforderungen an, die direkt in die Anforderungsspezifikation eingetragen werden können, z. B. Visionen und Ziele, dann kann dies direkt geschehen.

Definition

»Spezifikation: Ausführliche Beschreibung der Leistungen (z. B. technische, wirtschaftliche, organisatorische Leistungen), die erforderlich sind oder gefordert werden, damit die Ziele des Projekts erreicht werden. Anmerkung: Spezifikation kann auch als Pflichten- oder Lastenheft bezeichnet werden« (DIN 69901, S. 3).

Ergebnis

Das Ergebnis der Spezifikation ist folgendes Artefakt:

- Anforderungsspezifikation (evtl. aufgeteilt in Lasten- und/oder Pflichtenheft bzw. *User Stories* bei einer agilen Softwareentwicklung) gegliedert entsprechend einer Anforderungsschablone – mit integriertem oder separatem Glossar. Soweit möglich sollten Teile der Anforderungen bereits mit Hilfe von Basiskonzepten (siehe »Basiskonzepte«, S. 99) dokumentiert werden.

Individual- vs. Standardsoftware

Frage

Überlegen Sie, ob und wenn ja, wie sich das Ermitteln und das Spezifizieren von Individual- und Standardsoftware unterscheiden.

Antwort

Individualsoftware wird im Kundenauftrag entwickelt und besitzt folgende Eigenschaften [KMS05]:

- Die Anwender und die späteren Benutzer stehen in der Regel fest und können über Repräsentanten als *Stakeholder* eingebunden werden.
 - Anforderungen können zu Beginn weitgehend vollständig ermittelt werden. Aber: Anforderungen sind widersprüchlich.
 - Die Software ist langlebig und verfügt über vielfältige Schnittstellen zu anderen Systemen des Kunden.
 - Durch die Einbindung in vorhandene Prozesse und die Anbindung an Nachbarsysteme entstehen viele detaillierte Vorgaben.
 - Eine komplexe Ermittlung und Priorisierung von Anforderungen wird meist nicht durchgeführt.
 - Problematisch ist meistens die Konsolidierung der Anforderungen, bei der Inkonsistenzen und Anforderungslücken eliminiert werden.
 - Die Software ist oft nur einmal installiert und Updates sind selten.
- Standardsoftware** wird von der Marketingabteilung beauftragt und zeichnet sich durch folgende Eigenschaften aus:
- Die genaue Kenntnis des Absatzmarktes ist von ausschlaggebender Bedeutung.

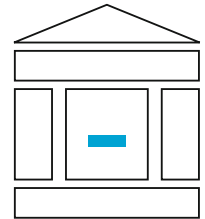
21 Anforderungen ermitteln und spezifizieren IV

- Die Anforderungen stammen aus zahlreichen Informationskanälen, z. B. durch Befragungen von Bestandskunden oder Partnern (*Voice of the Customer*), Marktrecherchen oder Wettbewerbsvergleichen. Intern liefern der Vertrieb, der Service und der Support Anforderungen.
- Die späteren Benutzer sind zum Entwicklungszeitpunkt nicht bekannt. Es stellt sich daher immer die Frage, ob Benutzer, von denen man Daten erhoben hat, repräsentativ für die Zielgruppe der Software sind. Eine aktive Minderheit kann eine schweigende Mehrheit überstimmen.
- Die spätere Art der Softwarenutzung und die Arbeitsabläufe beim Kunden können beliebig weit von den Ideen des Marketings abweichen. Daher müssen Möglichkeiten der Anpassung (*customizing*) und eine flexible Benutzerführung vorhanden sein.
- Die Software wird es in vielen verschiedenen Installationen betrieben, und es gibt unterschiedliche Versionen im Markt.

22 Anforderungen analysieren, validieren und abnehmen

Jedes Artefakt einer Softwareentwicklung ist einer Qualitätsüberprüfung zu unterziehen, bevor es für weitere Entwicklungsaktivitäten verwendet wird. Dies gilt insbesondere für die Anforderungsspezifikation, da sie die Basis für alle weiteren Entwicklungsaktivitäten bildet. Eine Überprüfung besteht aus zwei Schritten:

- Analyse der Anforderungsspezifikation
- Validation der Anforderungsspezifikation



Analysieren

Basis für die Überprüfung der Anforderungsspezifikation bzw. der Anforderungsdokumente sind die Qualitätskriterien, die die einzelnen Anforderungen und das jeweils gesamte Dokument erfüllen müssen (siehe »Anforderungen an Anforderungen«, S. 475).

Prüfkriterien

Alle natürlichsprachlichen Anforderungen müssen einer manuellen Überprüfung unterzogen werden. Gängige manuelle Prüfmethoden sind:

Manuelle Prüfung

- Inspektionen
- *Reviews*
- Walkthroughs

Weitere Methoden sind:

- Stellungnahme
- *Round Robin Review*
- *Peer Review*

Besonders zu empfehlen sind formale Inspektionen.

Alle diese Methoden sind im »Lehrbuch der Softwaretechnik – Softwaremanagement« beschrieben.

Ein wichtiges Hilfsmittel, das dafür sorgt, dass bei der Überprüfung nichts vergessen wird, sind **Checklisten**.

Checklisten

Sind Teile der Anforderungsspezifikation formal beschrieben, dann können auch automatisierte Hilfsmittel – zumindest ergänzend zu manuellen Prüfmethoden – eingesetzt werden.

Automatisierte
Überprüfung

IV 22 Anforderungen analysieren, validieren und abnehmen

Validieren

Validieren bedeutet, die Anforderungsspezifikation daraufhin zu überprüfen, ob sie das gewünschte Produkt richtig beschreibt. Das ist schwierig, da es *kein* Dokument gibt, gegen das die Prüfung durchgeführt werden kann. Es bieten sich zwei ergänzende Prüfverfahren an:

- Alle spezifizierten Anforderungen werden nochmals gegen die beschriebenen Visionen und Ziele geprüft. Trägt jede Anforderung dazu bei, die Visionen und Ziele zu verwirklichen? Wenn nein, dann ist sie zu entfernen.
- Alle *Stakeholder* bekommen die Anforderungsspezifikation zur Überprüfung – entweder jeder für sich oder im Rahmen eines gemeinsamen *Reviews*.

Nach Durchführung dieser Aktivitäten sollten die Anforderungen konsolidiert sein.

Abnehmen

Eine erfolgreiche Analyse und Validation sollte durch eine formelle Abnahme abgeschlossen werden.

Weitere Schritte Im Anschluss daran sollte eine Aufwandsschätzung und eine Priorisierung der Anforderungen stattfinden:

- »Schätzen des Aufwands«, S. 515
- »Anforderungen priorisieren«, S. 543

Beide Aktivitäten haben Rückwirkungen auf die Anforderungsspezifikation. Evtl. ist dann eine erneute Validation und Abnahme nötig. Alternativ kann eine Abnahme auch erst nach der Aufwandsschätzung und der Priorisierung erfolgen.

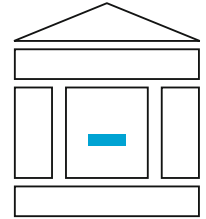
Weiterführende
Literatur [ElSc03]

23 Schätzen des Aufwands

Für jede Softwareentwicklung stellt sich die Frage, wie groß der Aufwand sein wird. In Abhängigkeit davon kann eine Entwicklung begonnen oder auch gestoppt werden – bevor weiterer Aufwand entsteht.

Eine Aufwandsschätzung ist möglich, wenn Erfahrungen aus in der Vergangenheit durchgeführten Softwareprojekten erfasst wurden und zur Verfügung stehen. Das Wissen darüber, wie viel Aufwand diese verursacht haben, kann genutzt werden – wenn das zu schätzende Projekt ähnlich ist:

- »Voraussetzungen und Einflussfaktoren«, S. 515
- »Warum ist das Schätzen des Aufwands wichtig?«, S. 517
- »Warum eine Aufwandsschätzung schwierig ist«, S. 518
- »Schätzverfahren«, S. 522
- »Die Function-Points-Methode«, S. 527
- »Object Points/Application Points«, S. 535
- »COCOMO II«, S. 536
- »Bewertung und weitere Aspekte«, S. 539
- »Zusammenfassung«, S. 542



i

23.1 Voraussetzungen und Einflussfaktoren

Die Voraussetzungen für die erfolgreiche Schätzung eines Softwareentwicklungsprojekts sind **Erfahrung** sowie eine gewisse **Ähnlichkeit** zu bereits durchgeführten Projekten. In aller Regel stellt ein Softwareunternehmen immer Software einer bestimmten Art in einer immer ähnlichen Weise her – die Projekte sollten also durchaus vergleichbar sein, und es ist daher möglich, aus den Erfahrungen mit vorigen Projekten zu lernen.

Relevante
Faktoren

Diese Ähnlichkeit hat sehr viele unterschiedliche Facetten, die gleichzeitig die für den zu erwartenden Aufwand relevanten Einflussfaktoren darstellen:

- **Quantität:** Anforderungen/Funktionalität sowie Projekteigenschaften, z. B. Art des Projekts (etwa Datenbankapplication gegenüber eingebetteter Software).
- **Qualität:** Nichtfunktionale Anforderungen wie z. B. Echtzeitanforderungen, Verfügbarkeit, beschränkte Ressourcen usw.
- **Kosten:** Für die Entwicklung zur Verfügung stehende Ressourcen.

IV 23 Schätzen des Aufwands

■ **Produktivität:** Eigenschaften der Entwicklungsorganisation (Prozesse, Personal, Erfahrung, verteilte Entwicklung, eingesetzte Techniken und Werkzeuge).

■ **Dauer:** Äußere Rahmenbedingungen, z. B. fester Liefertermin.

Teufels-
quadrat

Diese wesentlichen Einflussfaktoren sind stark voneinander abhängig und können im »Teufelsquadrat« gemäß Abb. 23.1-1 veranschaulicht werden [Snee87, S. 42]. Demnach ist die Produktivität (Fläche des Quadrats) konstant, und das Verbessern eines der vier übrigen Faktoren führt zwangsläufig zur Verschlechterung mindestens eines anderen Faktors.

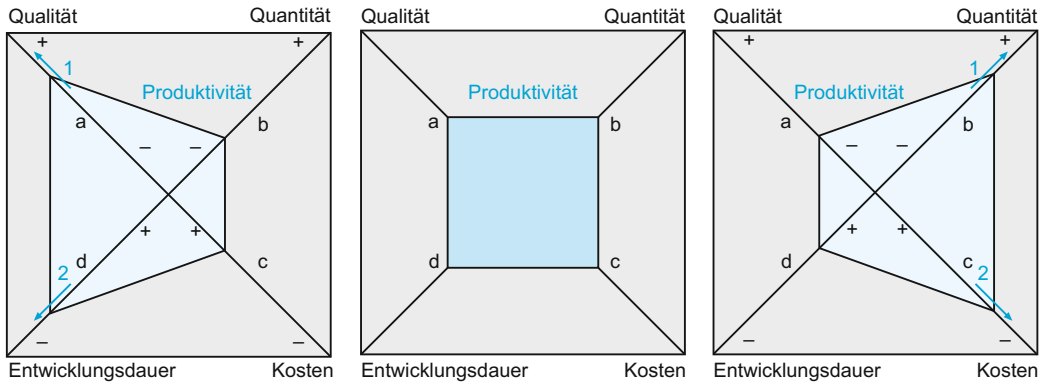


Abb. 23.1-1:
Beispiele zum
»Teufelsquadrat«.

Soll die Qualität eines zu entwickelnden Produkts erhöht (Pfeil 1) und gleichzeitig die Entwicklungsdauer verkürzt (Pfeil 2) werden, dann muss der Produktumfang reduziert werden. Gleichzeitig steigen die Entwicklungskosten (Abb. 23.1-1, linke Seite).

Soll die Quantität eines Produkts erhöht (Pfeil 1) und die Kosten verringert werden (Pfeil 2), dann muss die Qualität verringert und die Entwicklungsdauer verlängert werden (Abb. 23.1-1, rechte Seite).

Wenn Entwicklungsdauer und Kosten geschätzt werden sollen, hängen diese somit vor allem vom Umfang und von der geforderten Qualität des Zielprodukts sowie der Produktivität der Entwicklungsorganisation ab.

Es gibt eine ganze Reihe von Aufwandsschätzverfahren, die alle versuchen, die Erfahrung aus früheren Projekten zu nutzen. Je nachdem, welche Voraussetzungen vorliegen, kann die eine oder die andere Methode vorteilhaft angewandt werden. Idealerweise sollten mehrere verschiedene Methoden benutzt und miteinander verglichen oder kombiniert werden, um zu einem guten Ergebnis zu kommen.

23.2 Warum ist das Schätzen des Aufwands wichtig?

Überlegen Sie, warum es wichtig ist, den Aufwand zu schätzen.

Frage

Bei einem kommerziellen Softwareentwicklungsprojekt müssen zu einem **sehr frühen Zeitpunkt** bereits möglichst genaue Angaben über den für die Entwicklung anfallenden Aufwand, insbesondere die Kosten und die Dauer, angegeben werden. Diese Information ist sowohl für den Kunden, der ein konkretes Angebot samt Liefertermin erwartet, als auch für den Hersteller wichtig. Aus folgenden Gründen hat der Hersteller ein Interesse an einer möglichst guten Aufwandsschätzung:

Antwort

- **Angebotserstellung:** Der Hersteller muss seinen Aufwand kalkulieren, um einen Festpreis festlegen und einen Liefertermin nennen zu können. »Bezahlung nach Aufwand« ist in der Regel *nicht* möglich, denn damit sind die Kosten für den Kunden unkalkulierbar. Wenn also die tatsächlichen Kosten weit vom geschätzten Aufwand abweichen, kann dies zu Mehrkosten, Verlusten, Strafzahlungen und im schlimmsten Fall zur Insolvenz des Unternehmens führen. Liegen die Kosten im Vergleich zum Wettbewerber zu hoch, bekommt er den Auftrag nicht. Liegen sie zu niedrig, kann er seine Kosten nicht decken. Eine gute Aufwandschätzung kann somit mitentscheidend für den wirtschaftlichen Erfolg eines Unternehmens sein.
- **Mitarbeitereinplanung:** Bei Beauftragung müssen die benötigten Mitarbeiter eingeplant werden. Auch dafür ist die Aufwandschätzung Voraussetzung. Die Anzahl und Qualifikation der benötigten Mitarbeiter ist in verschiedenen Phasen des Projekts unterschiedlich. Sind die benötigten Mitarbeiter zum entsprechenden Zeitpunkt nicht verfügbar, so kann dies zu Verzögerungen führen. Dieser Aspekt muss daher schon bei der Schätzung der Dauer berücksichtigt werden (Liefertermin).
- **Projektplanung und -controlling:** Ferner wird eine detaillierte Schätzung für die spätere Projektplanung benötigt. Wenn die Dauern der einzelnen Aktivitäten nicht bekannt sind, können sie auch nicht zeitlich eingeplant werden. Und ohne Planung ist eine Überwachung des Projektverlaufs und ein rechtzeitiges Erkennen von Problemen nicht möglich.
- **make or buy:** Die zu erwartenden Kosten sind weiterhin für eine *make or buy*-Entscheidung wichtig. Eventuell ist es günstiger, benötigte Komponenten zuzukaufen oder von einem Drittanbieter herstellen zu lassen. Unter Umständen kann bei entsprechendem Kostenvorteil auch ein existierendes System zugekauft und angepasst oder erweitert werden.

IV 23 Schätzen des Aufwands

Es gibt also eine Vielzahl von Gründen, die eine möglichst genaue Aufwandsschätzung nötig machen. Aber auch der Kunde hat ein Interesse an einer guten Schätzung. Denn wenn sich der Hersteller verkalkuliert hat und in die Insolvenz geht, bekommt auch der Kunde sein gewünschtes Produkt nicht oder hat später eine Software, die niemand mehr warten kann.

Kosten Die zu erwartenden Kosten hängen direkt mit dem Aufwand zusammen. Der größte Kostenanteil einer Softwareentwicklung wird durch den Personalaufwand bestimmt. Über diesen werden in der Regel auch schon alle Nebenkosten eines Softwareunternehmens verrechnet, d.h. sie sind im Kostensatz eines Entwicklertages enthalten (Kosten für Büroräume, Ausstattung, Kommunikation etc.). Daneben können noch Hardware- oder Softwarekosten sowie Projektleitungs-, Reise-, Schulungs- und Wartungskosten anfallen.

Angebot Bei der Angebotserstellung spielen allerdings nicht nur die tatsächlich zu erwartenden Kosten eine Rolle. Unter Umständen sind hier auch **taktische** oder **strategische Überlegungen** des Unternehmens relevant. Wenn z.B. ein neuer Markt erschlossen werden soll oder ein Produkt (mit leichten Variationen) in Zukunft voraussichtlich noch mehrfach an andere Kunden verkauft werden kann und darf, muss der Preis, den der Erstkunde zahlt, nicht die gesamten Entwicklungskosten decken. Der Angebotspreis ergibt sich dann aus der Frage »mit welchem Preis gewinnen wir die Ausschreibung?« Diese Vorgehensweise wird auch *pricing to win* genannt. Ebenso können andere organisatorische, politische oder ökonomische Überlegungen zu einem Angebot führen, das mit den geschätzten Kosten nichts zu tun hat. Soll dagegen ein realistisches Angebot abgegeben werden, das die Kosten deckt und einen entsprechenden **Deckungsbeitrag** leistet, so ist eine gute Aufwandsschätzung als Basisinformation unerlässlich.

23.3 Warum eine Aufwandsschätzung schwierig ist

Neben der bereits genannten Vielzahl von Faktoren, die den Aufwand beeinflussen (siehe »Voraussetzungen und Einflussfaktoren«, S. 515), gibt es eine Reihe weiterer Umstände, die eine Aufwandsschätzung schwierig machen.

Anforderungen unklar So sind die Vorstellungen darüber, wie genau das Produkt aussehen soll, häufig zum Zeitpunkt der Angebotserstellung noch recht unklar. Erst im Laufe des Projekts wird zunehmend deutlich, *was* überhaupt gemacht werden soll, *wie* es realisiert wird, *wo* die Probleme

23.3 Warum eine Aufwandsschätzung schwierig ist IV

me stecken, und damit *wie hoch* der Aufwand tatsächlich ist. Auch der Kunde ist meist zu Beginn nicht in der Lage, seine Anforderungen klar zu definieren (*»I know it when I see it«*).

Mit der Unkenntnis der genauen Anforderungen gehen auch eine Reihe von Risiken einher. Eventuell sollen neue Techniken oder Werkzeuge zum Einsatz kommen, oder es gibt Abhängigkeiten von Dritten. Und vielleicht gibt es Anforderungen, die noch nicht bekannt sind, aber große technische Schwierigkeiten/Herausforderungen mit sich bringen könnten. Um derartige Unwägbarkeiten in die Kostenschätzung einzubeziehen, schlagen [MaLi03] vor, keinen fixen Auslieferungstermin zu nennen, sondern diesen als Wahrscheinlichkeitsverteilung anzugeben. Die Verteilung ergibt sich aus den Eintrittswahrscheinlichkeiten der einzelnen Risiken und dem entstehenden Schaden – in Form von Aufwand – bei deren Eintreten. Sie kann mittels Monte-Carlo-Simulation angenähert werden. Der Projektverlauf wird dabei sehr häufig zufallsgesteuert durchgespielt, wobei der Eintritt der verschiedenen Risiken mit der ihnen jeweils zugeordneten Wahrscheinlichkeit stattfindet. Entsprechend resultiert pro Simulationslauf ein Aufwandswert. Diese Werte ergeben dann entsprechend der Häufigkeit ihres Auftretens die Verteilung. Der Aufwand ist also abhängig von den Risiken, und die Risiken stellen somit einen schwer beurteilbaren Faktor für die Aufwandsschätzung dar.

Risiken

Die Dauer eines Projekts ist nicht mit dessen Entwicklungsaufwand identisch: Je mehr Personen an einem Projekt beteiligt sind, desto höher wird der Kommunikationsaufwand und -overhead [Broo75]. So kann ein Projekt, für das ein Entwickler allein 12 Monate brauchen würde, nicht von 12 Mitarbeitern in einem Monat fertiggestellt werden. Bei größeren Projekten ist die Zusammenarbeit vieler Mitarbeiter aber unumgänglich.

Dauer

Andererseits macht es bei größeren Projekten auch keinen Sinn, eine große Anzahl Mitarbeiter über die gesamte Projektlaufzeit dem Projekt zuzuordnen. Die Spezifikations- und Entwurfsphase wird von wenigen Experten schneller erledigt, als wenn hier die gesamte spätere Entwicklungsmannschaft mitarbeitet [Marc98]. Dagegen lassen sich Implementierung, Test und Dokumentation bei geeigneter Modularisierung des Systems effizient parallelisieren. Auch können je nach Aufgabe unterschiedliche Personen (Experten) zum Einsatz kommen. Allerdings müssen diese dann auch zum entsprechenden Zeitpunkt für das Projekt zur Verfügung stehen.

Mitarbeiter-
zuordnung

Die Bestimmung der Dauer ist also ein noch komplexeres Problem als die reine Aufwandsschätzung. Der Liefertermin ist aber für den Kunden oft von entscheidender Bedeutung, und eine Nichteinhaltung kann zu hohen Folgekosten führen.

IV 23 Schätzen des Aufwands

Produktivitäts-
unterschiede

Ein weiteres Problem bei der Aufwandsschätzung sind große Produktivitätsunterschiede bis zu Faktoren größer 10 zwischen einzelnen Entwicklern ([Brya94], [Boeh00]). Diese gleichen sich bei größeren Projekten gegenseitig aus, bei kleineren Projekten können sie die Projektdauer aber entscheidend beeinflussen. Ebenso spielt die Erfahrung eine Rolle. Wenn ein Entwickler schon viele ähnliche Projekte bearbeitet hat, wird er mit einem weiteren solchen Projekt wenig Probleme haben. Eine für ihn unbekannte Domäne (oder Sprache, Arbeitsumgebung, Werkzeuge, Technologie, Prozess) wird dagegen Einarbeitungsaufwand erfordern, und es muss mit Fehlversuchen oder zusätzlicher Zeit für die Prototyperstellung gerechnet werden.

Parkinsons Gesetz

Nach Parkinsons Gesetz [Park57] verbraucht ein Projekt immer die gesamte zur Verfügung stehende Zeit. Eine Überschätzung des Aufwands führt somit nicht zu einer kürzeren Projektlaufzeit – zumindest dann nicht, wenn allen Beteiligten der geplante Fertigstellungstermin bekannt ist. Überschätzungen führen (wenn Erfahrung aus diesem Projekt genutzt wird) in Folgeprojekten ebenfalls zur Überschätzung des Aufwands und damit unter Umständen zu Wettbewerbsnachteilen – wenn zu teuer angeboten wird und ein Konkurrent den Zuschlag erhält.

Einzel-
interessen

Ein verwandtes Problem ergibt sich aus den Einzelinteressen der schätzenden Personen. Entwickler schätzen eher zu hohem Aufwand, da sie nicht unter hohem Druck arbeiten wollen, »schöne« Lösungen bevorzugen, und eventuell andere Aufgaben mit im Aufwand unterbringen müssen (z. B. *Refactoring*). Das Management schätzt eher niedriger, um einen früheren Liefertermin nennen zu können oder Ressourcen zu sparen – oder es unterschätzt einfach die technische Komplexität.

Weitere Faktoren, die Probleme bereiten können, sind:

- **Aufwand:** Je genauer eine Aufwandsschätzung sein soll, desto aufwendiger ist sie. Erfolgt sie nur zur Erstellung eines Angebots, ist dies möglicherweise zu teuer. Die Kosten für die Erstellung eines Angebots, das den Zuschlag *nicht* erhält, müssen von anderen Projekten wieder erwirtschaftet werden.
- **Zeitdruck:** Die Aufwandsschätzung wird häufig unter Zeitdruck durchgeführt. Es gibt feste Abgabetermine für das Angebot und möglicherweise viele parallel laufende Projekte und Angebote.
- **Verfügbarkeit historischer Daten:** Informationen über vorige Projekte müssen erst einmal gesammelt werden, damit sie später zur Aufwandsschätzung genutzt werden können. Es gibt keinen unmittelbaren Nutzen, und somit ist der zusätzliche Aufwand für die Datenerhebung schwer zu rechtfertigen.
- **Erfahrung:** Sind diese Daten nicht verfügbar, ist man bei der Aufwandsschätzung auf die Erfahrung langjähriger Mitarbeiter angewiesen. Man ist von ihnen abhängig.

23.3 Warum eine Aufwandsschätzung schwierig ist IV

- **Beschränkte Ressourcen:** Häufig ist gar keine realistische Schätzung gefragt – sie muss sich an den vorhandenen Ressourcen und Rahmenbedingungen orientieren. Die Schätzung erfolgt dann gemäß Parkinsons Gesetz: »Wenn sowieso die gesamte *veranschlagte* Zeit gebraucht wird, dann wird auch die *verfügbare* Zeit schon irgendwie reichen.«

Nach Untersuchungen von Barry Boehm [Boeh08, S. 33] liegen die Schätzungen zu Beginn eines Projekts aufgrund der genannten Umstände um einen Faktor bis zu vier über oder unter dem wirklichen Aufwand (Abb. 23.3-1). Im Laufe des Projekts wird die Schätzung dann immer genauer, bis – zum Abschluss des Projekts – die genauen angefallenen Kosten und die Projektdauer schließlich bekannt sind. Für viele Fragestellungen kommt diese Information dann aber zu spät. Allerdings kann Boehms Erkenntnis genutzt werden, indem die Schätzung im Laufe des Projekts öfters angepasst bzw. verfeinert (siehe »COCOMO II«, S. 536) und entsprechend reagiert wird (z. B. Priorisierung von Anforderungen, Kommunikation/Absprache mit dem Kunden). So können böse Überraschungen für beide Seiten vermieden werden.

Faktor 4

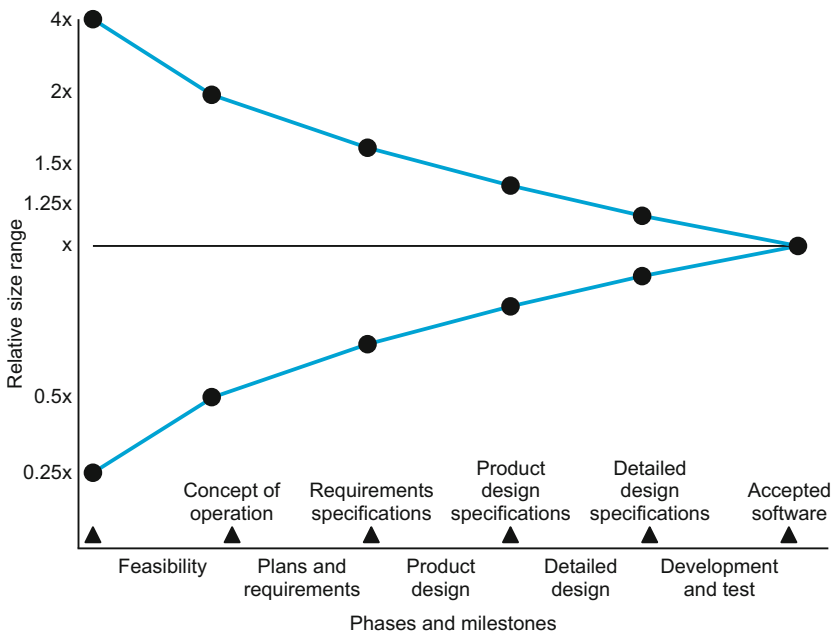


Abb. 23.3-1:
Entwicklung des Aufwandsschätzfehlers (in Anlehnung an [Boeh08, S. 33]). x ist der tatsächliche Aufwand.

Üblich ist es in der Praxis daher, den mit einem Schätzungsverfahren ermittelten Aufwand mit einem »Korrekturfaktor« zu multiplizieren. Dieser deckt dann unerwartete bzw. bei der Schätzung nicht berücksichtigte Faktoren pauschal ab. Dies können z. B. technische,

Korrekturfaktor

IV 23 Schätzen des Aufwands

aber auch organisatorische (z. B. Schwierigkeiten bei der Kommunikation mit dem Kunden) oder personelle Schwierigkeiten (Chefentwickler kündigt) sein.

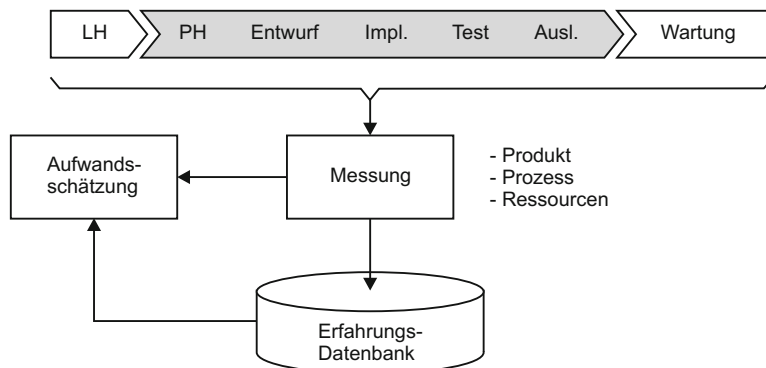
Dokumentation

In jedem Fall sollten die der Schätzung zugrunde liegenden Annahmen dokumentiert werden. So kann die Schätzung leichter an neue Entwicklungen angepasst werden, und es ist später nachvollziehbar, warum sie möglicherweise falsch war.

23.4 Schätzverfahren

Alle Schätzmethoden beruhen auf Erfahrung. Es müssen also Daten aus vergangenen Projekten erhoben worden sein und zur Verfügung stehen, aus denen dann auf den Aufwand eines neuen Projekts geschlossen werden kann. Die Abb. 23.4-1 zeigt den grundlegenden Prozess. Während des Entwicklungsprozesses werden Messungen durchgeführt. Deren Ergebnisse bilden zum einen die Basis für eine bestmögliche Aufwandsschätzung zum aktuellen Zeitpunkt, zum anderen werden sie nach Abschluss des Projekts in eine Erfahrungsdatenbank eingespeist. Diese Daten können dann für eine Aufwandsschätzung zusätzlich herangezogen werden. Da die Eigenschaften der Entwicklungsorganisation den Aufwand stark beeinflussen, bringen eigene Erfahrungswerte hier den größten Nutzen. Erfahrungswerte anderer Organisationen sind dagegen nur begrenzt einsetzbar.

Abb. 23.4-1:
Schätzen lernen
aus Erfahrung. LH
= Lastenheft, PH =
Pflichtenheft.



- i Es gibt verschiedene Arten von Aufwandsschätzmethoden:
- »Analogiemethode«, S. 523
 - »Expertenschätzung«, S. 523
 - »Bottom-up-Methode«, S. 524
 - »Prozentsatzmethode«, S. 525
 - »Algorithmische Schätzung«, S. 526
 - »Faustregeln«, S. 526

23.4.1 Analogiemethode

Eine recht einfache Kostenschätzungsmethode ist die **Analogiemethode**. Dabei wird ein bereits abgeschlossenes Projekt identifiziert, das dem abzuschätzenden Projekt möglichst ähnlich ist. Man kann davon ausgehen, dass dann auch der Aufwand für die Realisierung des neuen Projekts ähnlich sein wird. Eventuell hat man aus dem abgeschlossenen Projekt gelernt, z. B. Domänenwissen oder anderes Spezialwissen aufgebaut – dann ist der Aufwand möglicherweise geringer. Auch wenn Wiederverwendung möglich ist, reduziert sich der Aufwand.

Die Unterschiede zum gewählten Referenzprojekt müssen in jedem Fall genau betrachtet und für diese Teile der abweichende Aufwand dazuaddiert bzw. abgezogen werden. Das Verfahren arbeitet um so genauer, je ähnlicher die beiden Projekte sich sind (auch bezüglich der beteiligten Mitarbeiter). Es ist folglich nur anwendbar, wenn überhaupt ein vergleichbares Projekt existiert.

Eine eng verwandte Methode ist die **Relationsmethode**, bei der die Abweichungen vom Referenzprojekt über Faktorlisten und Richtlinien zur Anpassung der Schätzung führen. So kann es beispielsweise Faktoren für den Wechsel von einer bestimmten Programmiersprache zu einer anderen geben. Der Umgang mit Abweichungen ist bei dieser Methode also genau spezifiziert.

Ein Unternehmen soll ein Buchungssystem für einen Kreuzfahrtveranstalter erstellen. Es hat bereits mehrere ähnliche Systeme für andere Reiseveranstalter realisiert. Nur die für diesen Veranstalter spezifische Funktionalität (z. B. Kreuzfahrt statt Pauschalreisen) muss nun als Neuentwicklung geschätzt werden. Die übrige Funktionalität ist sehr ähnlich zu den vorigen Systemen, daher wird der zu deren Realisierung nötige Aufwand ebenfalls ähnlich sein. Werden Teile eines alten Systems wiederverwendet, sinkt der Aufwand für diese Teile, beispielsweise auf 25 % des Neuerstellungsaufwands. Beispiel

23.4.2 Expertenschätzung

Eine artverwandte und weit verbreitete Methode ist die **Expertenschätzung**. Hier beurteilt ein Experte das zu schätzende Projekt. Aufgrund seiner langjährigen Erfahrung mit vielen verschiedenen Softwareprojekten ist er in der Lage, den Aufwand intuitiv genauer anzugeben als jemand, der nicht über diese Erfahrung verfügt. Die Schätzung wird also nicht aus Messwerten früherer Projekte berechnet, sondern findet allein im Kopf des Experten statt. Sein Kopf bildet die Erfahrungsdatenbank, und die Messung erfolgte durch seine Mitarbeit in Projekten.

IV 23 Schätzen des Aufwands

Diese Methode kann weiter verfeinert werden, indem mehrere Experten befragt werden. Bei stark differierenden Schätzungen sollten die Experten dann zusammengebracht und ein Meinungsaustausch angestoßen werden. Da verschiedene Experten unterschiedliche Aspekte bei ihrer Schätzung berücksichtigen werden, sollten durch deren Kombination alle wesentlichen Aspekte einfließen und das Ergebnis entsprechend gut sein.

Experten können eigene erfahrene Entwickler oder Projektleiter sein, aber auch externe Berater mit entsprechendem Hintergrund. Der Vorteil des internen Experten ist dabei, dass er auch die firmenspezifischen Besonderheiten berücksichtigen kann.

Dreipunkt-
verfahren

Eine weitere Verfeinerung der Expertenschätzung ist das **Dreipunktverfahren**. Dabei werden drei Schätzungen abgegeben: Jeweils eine unter Annahme von optimistischen, pessimistischen und wahrscheinlichen Bedingungen. Daraus kann dann ein gewichteter Mittelwert (wie in PERT¹) berechnet werden:

$$T_E = (T_{\text{optimistisch}} + 4 \cdot T_{\text{wahrscheinlich}} + T_{\text{pessimistisch}}) / 6$$

Wideband-Delphi

Die **Wideband-Delphi-Methode** [Boeh81] ist ein Beispiel für eine systematisierte Expertenschätzung. Eine Gruppe von 3–7 Mitarbeitern trifft sich zwei Mal. Beim ersten Treffen wird das Projekt in ca. 10–20 Teilaufgaben zerlegt. Diese werden von jedem Mitarbeiter separat geschätzt, wobei die jeweils getroffenen Annahmen dokumentiert werden. Dann treffen sich die Mitarbeiter zum zweiten Mal, tauschen ihre Schätzungen aus und diskutieren diese unter Anleitung eines Moderators. Unterschiedliche Annahmen und Unklarheiten werden zusammengeführt und geklärt. So nähern sich die Schätzungen in mehreren Iterationen aneinander an, bis deren Spannweite für jede Teilaufgabe als akzeptabel angesehen wird.

23.4.3 Bottom-up-Methode

Eine Alternative zur direkten Schätzung der Gesamtkosten eines Projekts ist es, das Projekt in **kleine Einzelteile** zu zerlegen, die Kosten für die Entwicklung der Einzelteile zu schätzen und diese dann zusammenzuaddieren. Hinzu kommt der Integrationsaufwand. Dies ist die sogenannte **Bottom-up-Methode**. Diesem Verfahren liegen folgende Ideen zugrunde:

¹Die *Program Evaluation and Review Technique* ist eine ereignisorientierte Netzplantechnik, die für die Planung von Projekten, die mit hohen Unsicherheiten behaftet sind, entwickelt wurde. Der Zeitbedarf für jeden Vorgang wird nicht durch eine feste Größe, sondern über eine Beta-Wahrscheinlichkeitsverteilung geschätzt. Dabei kommt die Formel T zum Einsatz.

- 1 Bei den Schätzungen der Einzelteile kommen Abweichungen nach oben und nach unten in etwa gleich häufig vor und heben sich damit gegenseitig auf.
- 2 Kleinere Teile sind besser überschaubar und damit besser abschätzbar, d.h. die Genauigkeit dieser Teilschätzungen wird relativ gut sein.
- 3 Bei der Schätzung können die späteren Entwickler einbezogen werden, die ihren eigenen Aufwand vermutlich besser einschätzen können als Personen, die nicht an der Entwicklung beteiligt sind. Außerdem sind sie in der Regel Experten auf ihrem Spezialgebiet (z. B. Datenbanken oder Benutzungsoberflächen). Es handelt sich hier um eine Art **Expertenschätzung** im Kleinen, auf anderer Ebene.

Allerdings können gerade beim Zusammenbau und Zusammenspiel der Einzelkomponenten die größten Probleme auftreten, die bei diesem Verfahren leicht übersehen oder unterschätzt werden.

Watts Humphrey entwickelte für den persönlichen Softwareprozess **PROBE** (*Proxy Based Estimating*) – eine Methode zur Bestimmung des Entwicklungsaufwands einer einzelnen Komponente [Hump95]. Jeder Entwickler baut eine Datenbank mit Informationen über Komponenten auf, die er in der Vergangenheit entwickelt hat. Die Komponenten werden nach ihrem Typ (z. B. Berechnung, Daten, Logik) und ihrer Größe klassifiziert. Mittels linearer Regression werden diese Daten für die Schätzung des Entwicklungsaufwands zukünftiger Komponenten der gleichen Klasse benutzt. Da die Komponenten von einer einzelnen Person entwickelt werden, kann hier ein linearer Zusammenhang angenommen werden.

PROBE

23.4.4 Prozentsatzmethode

Die **Prozentsatzmethode** basiert auf der Annahme, dass die gleichen Phasen von Projekten (z. B. Anforderungsspezifikation, Entwurf, Implementierung, Test) immer einen ähnlichen Anteil am Gesamtaufwand haben. Dann kann aufgrund historischer Daten vom Aufwand einer einzelnen (z. B. der ersten) Phase auf den zu erwartenden Aufwand der anderen Phasen sowie den Gesamtaufwand geschlossen werden. Allerdings müssen für die Anwendbarkeit dieser Methode ebenfalls gewisse Anforderungen an die Ähnlichkeit der Projekte und Prozesse erfüllt sein. Die Verteilung ist unter anderem abhängig von der Größe des Projekts (Tab. 23.4-1).

Für ein System wurde bereits die Anforderungsspezifikation erstellt. Dafür wurden 10 **Personenmonate** (PM) benötigt, und aus der Spezifikation wurde ein Umfang von ca. 1.000 *Function Points* (siehe »Die Function-Points-Methode«, S. 527) ermittelt. Dann wird (Tab.

Beispiel

IV 23 Schätzen des Aufwands

Tab. 23.4-1:
Verteilung des
Aufwands nach
Projektgröße
(Angaben jeweils
in Prozent; FP =
Function Points)
[Jone07].

Aktivität	10 FP	100 FP	1.000 FP	10.000 FP	100.000 FP
Spezifikation	5	5	7	8	9
Entwurf	5	6	10	12	13
Implementierung	50	40	30	20	15
Test	27	29	30	33	34
Change Management	1	4	6	7	8
Dokumentation	4	6	7	8	9
Projektmanagement	8	10	10	12	12

23.4-1) ein Gesamtaufwand von ca. $10 \text{ PM} \cdot 100\% / 7\% = 143 \text{ PM}$ erwartet, von denen jeweils ca. 43 PM (=30%) für Implementierung und Test benötigt werden.

23.4.5 Algorithmische Schätzung

Bei der algorithmischen Schätzung (»Berechnung aus früh bekannten Größen«) werden eine Vielzahl von Fragen zum Projekt und zur Organisation gestellt und aus den Antworten auf den Aufwand geschlossen. Auch hier basiert die Schätzung auf Erfahrung aus abgeschlossenen Projekten: Entsprechende Zusammenhänge zwischen den Antworten (einer daraus resultierenden Punktzahl) und dem tatsächlichen Realisierungsaufwand müssen zuvor aus empirischen Daten ermittelt worden sein. Die zwei bekanntesten algorithmischen Verfahren sind:

■ »Die Function-Points-Methode«, S. 527

■ »COCOMO II«, S. 536

Die Kalibrierung dieser Verfahren erfolgt üblicherweise per Regressionsanalyse. Daneben wurden in den vergangenen 20 Jahren auch Ansätze aus der künstlichen Intelligenz für das Lernen des Verhältnisses zwischen Anforderungen und Aufwand benutzt. Dabei kommen unter anderem neuronale Netze, Bayes'sche Netze, unscharfe Logik oder Entscheidungsbäume zum Einsatz. COCOMO II wurde beispielsweise mit einem Bayes-Ansatz kalibriert, und das PROBE-Verfahren (siehe »Bottom-up-Methode«, S. 524) ist der unscharfen Logik zuzuordnen.

23.4.6 Faustregeln

Die meisten Schätzverfahren sind recht aufwendig. Alternativ gibt es auch einige sehr einfache Faustregeln, die ebenfalls auf Erfahrung basieren, jedoch aufgrund ihrer starken Vereinfachung nur zu sehr ungenauen Ergebnissen führen können. Sie basieren alle auf **DLOC** – der Anzahl der ausgelieferten produktiven Codezeilen.

DLOC schließt Code, der nur für Tests benötigt wird, nicht mit ein. In der Tab. 23.4-2 sind einige Faustregeln für prozedurale Sprachen in Abhängigkeit des Gesamtumfangs angegeben.

Größe [DLOC]	DLOC/PM	DLOC/PS
1.000	833	5,76
10.000	588	4,13
100.000	400	2,60
1.000.000	267	2,02

Tab. 23.4-2:
Faustregeln für
prozedurale
Sprachen
(PM=Personenmonat,
PS=Personenstunde)
[Jone07].

Ein System, das auf 200.000 DLOC geschätzt wird, kann (interpoliert) mit einer Produktivität von ca. 360 DLOC pro Personenmonat rechnen. Somit ergibt sich der erwartete Gesamtaufwand zu $200.000/360=556$ PM.

Beispiel

Entsprechende Regeln existieren auch für *Function Points* (siehe »Die Function-Points-Methode«, S. 527). In jedem Fall wird auch hier eine existierende Schätzung des Umfangs vorausgesetzt.
[Kras05]

Weiterführende
Literatur

23.5 Die Function-Points-Methode

Eine verbreitetes algorithmisches Schätzverfahren ist die im Jahr 1979 von Alan J. Albrecht (IBM) veröffentlichte **Function-Points-Methode** [Albr79]. Sie ist auch heute noch eines der wenigen Standardschätzverfahren für Software und hat daher eine entsprechende Bedeutung und Verbreitung. Seit 2003 ist sie standardisiert, unter anderem in der ISO/IEC-Norm 20926².



Alan J. Albrecht

Die Idee dieser Methode ist es, den funktionalen Umfang eines zu entwickelnden Softwaresystems auf Basis der funktionalen Anforderungen zu messen. Die Anforderungen (laut Lasten- oder Pflichtenheft) definieren, welche Funktionalität aus Sicht des Benutzers vom System erwartet wird. Dies wird in der Regel durch eine Beschreibung der nötigen Ein- und Ausgaben sowie der beteiligten Daten erreicht. Ist diese Information vorhanden, so kann sie systematisch ausgewertet werden und gibt ein relativ genaues Bild vom Umfang der Software – zumindest aus Benutzersicht. Aus dem Resultat kann dann unter Ausnutzung historischer Daten auf den personellen Aufwand für die Umsetzung geschlossen werden. Die Function-Points-Methode zielt dabei besonders auf klassische Informationssysteme ab (bzw. wurde dafür entwickelt), bei denen Informationen in ent-

²ISO/IEC 20926 ist die IFPUG-Variante (siehe unten) der Function-Points-Methode. Auch andere Versionen sind standardisiert, etwa COSMIC in der ISO/IEC-Norm 19761.

IV 23 Schätzen des Aufwands

sprechenden Masken eingegeben und über Berichte ausgewertet und ausgegeben werden. Sie ist jedoch prinzipiell unabhängig von der späteren technischen Realisierung des Systems.

Der Vorteil der Function-Points-Methode ist, dass sie zu einem relativ frühen Zeitpunkt im Projektverlauf schon relativ genaue und objektive Aussagen zum funktionalen Umfang der zu entwickelnden Software erlaubt. Voraussetzung dafür ist allerdings die Verfügbarkeit eines Lastenhefts, das die wesentlichen Anforderungen schon berücksichtigt und möglichst detailliert beschreibt. Daneben muss genau definiert sein, wie gezählt wird, um tatsächlich zu einem objektiven und vergleichbaren Ergebnis zu kommen. Entsprechende Zählregeln werden beispielsweise von der IFPUG (*International Function Point Users Group*) festgelegt. Deren Einhaltung kann dann zu einer besseren Austauschbarkeit beitragen.

Ablauf Der grobe Ablauf der Function-Points-Methode ist wie folgt:

- 1 **Zähltyp festlegen.** Hierbei wird zwischen einer Neuentwicklung, einer Weiterentwicklung und der Zählung eines bestehenden Systems unterschieden.
- 2 **Systemgrenzen festlegen.** In diesem Schritt wird genau definiert, welche Teile zum System gehören und welche außerhalb des Systems liegen. Hieraus ergeben sich die Schnittstellen (Ein-/Ausgaben) und die Unterscheidung zwischen internen und externen Daten/Datenbanken. Dies ist in der Abb. 23.5-1 am Beispiel von SemOrg veranschaulicht (siehe »Fallstudie: SemOrg – Die Spezifikation«, S. 107). Die Systemgrenze ist als Rechteck dargestellt, die Schnittstellen durch Pfeile.
- 3 **Identifizieren von Funktionstypen einschließlich Datenbeständen.** Hier erfolgt eine Zerlegung in Elementarprozesse und deren Zuordnung zu einem von fünf Funktionstypen. Dies wird im Folgenden genauer beschrieben.
- 4 **Bewertung der Komplexität** der einzelnen Funktionstypen. Je nach Funktionstyp und Umfang der jeweils beteiligten Daten wird die Anzahl zu zählender *Function Points* bestimmt.
- 5 **Ermittlung der gewichteten *Function Points*** (optional). Hierbei werden weitere Einflüsse berücksichtigt, die sich auf den Umfang des Systems auswirken, aber nicht in Ein-/Ausgaben und Datenbeständen zum Ausdruck kommen.

Elementarprozess Im dritten Schritt wird der Funktionsumfang der zu bewertenden Anwendung in Elementarprozesse zerlegt. Ein **Elementarprozess** ist eine atomare und einzigartige Aktivität des Systems aus Anwendersicht. **Atomar** bedeutet, dass der Elementarprozess die kleinste aus fachlicher Sicht sinnvolle in sich abgeschlossene Aktivität ist, die mit dem System durchführbar ist. **Einzigartig** bedeutet, dass der Elementarprozess durch die ein- oder ausgegebenen Daten oder

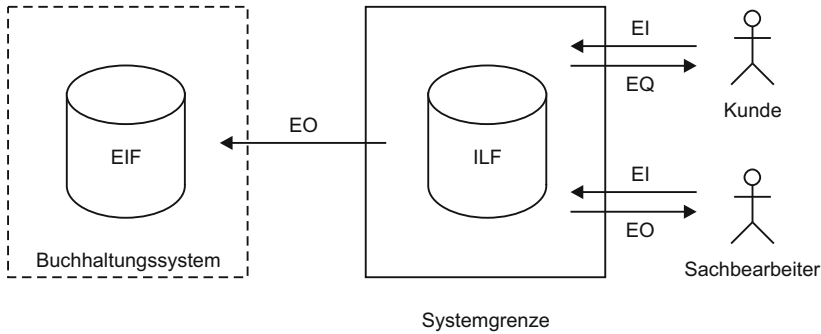


Abb. 23.5-1:
Systemgrenze am
Beispiel SemOrg.
Extern: Buchhal-
tungssystem (EIF),
Kunde, Kunden-
sachbearbeiter.
Intern:
Datenbestand (ILF)
Seminare,
Veranstaltungen,
Kunden, Dozenten,
Firmen,
Buchungen.
Transaktionen (EI,
EO, EQ) bewegen
Informationen
über die
Systemgrenze.

durch die Verarbeitungslogik aus Sicht des Anwenders unterscheidbar ist. Durch Anwendung dieser Prinzipien erreicht die Function-Points-Methode eine eindeutige und nachvollziehbare Zerlegung.

Beispiele für Elementarprozesse:

- a Die Erfassung eines neuen Kunden /LF20/.
- b Die Anzeige einer Veranstaltungsliste /LF10/.
- c Der Ausdruck von Versandpapieren /LF30/.
- d Die Übermittlung von Buchungsdaten an das Buchhaltungssystem /LF80/.

Jeder Elementarprozess lässt sich – auf Basis seines Hauptzwecks – einem **Funktionstyp** zuordnen:

1 Transaktions-Funktionstypen: Hierbei werden Daten über die Systemgrenze bewegt.

a Eingaben EI (External Input): Daten werden von außen ins System gebracht und im System gespeichert (als Datenbestand oder Systemzustandsänderung).

b Ausgaben EO (External Output): Daten werden vom System generiert und nach außen gegeben. Bei dieser Transaktion müssen entweder die Daten vor der Ausgabe transformiert (abgeleitet) oder der Systemzustand bzw. die internen Daten geändert werden.

c Abfragen EQ (External Inquiry): Eingabedaten gehen ins System, werden dort aber *nicht* gespeichert, und Ausgabedaten werden vom System nach außen gegeben. Hierbei handelt es sich im Gegensatz zur **Ausgabe** um eine reine Abfrage von Daten – ohne weitere Verarbeitung und ohne Seiteneffekte. Ein Beispiel für diesen Transaktionstyp ist eine Suchanfrage. Die Eingabedaten enthalten die Suchkriterien.

2 Daten-Funktionstypen: Dies sind Sammlungen von Daten, die sich entweder innerhalb oder außerhalb der Systemgrenze befinden. Sie werden durch die Transaktions-Funktionstypen gepflegt.

Funktions-
typen

IV 23 Schätzen des Aufwands

- a Interne Datenbestände** ILF (*Internal Logical File*): Eine Gruppierung von logisch zusammengehörigen Daten, die sich *innerhalb* der Systemgrenze befinden. Dies kann beispielsweise einer Datenbanktabelle oder Datei entsprechen.
- b Externe Datenbestände** EIF (*External Interface File*): Entsprechende Daten *außerhalb* der Systemgrenze, die von einem anderen System gepflegt werden.

- Beispiel 1b
- a** Die Erfassung eines neuen Kunden ist eine Eingabe, denn die Daten kommen von außen und werden im internen Datenbestand abgelegt.
 - b** Die Anzeige einer Veranstaltungsliste ist eine Abfrage. Die Daten werden über die Systemgrenze bewegt und unverändert angezeigt, und durch die Abfrage wird der Systemzustand nicht beeinflusst.
 - c** Der Ausdruck von Versandpapieren ist eine Ausgabe, da die Daten über die Systemgrenze zum Drucker geschickt werden und der erfolgte Druck in der Datenbank vermerkt wird (/F120/). Somit erfolgt eine dauerhafte Änderung des internen Datenbestands.
 - d** Die Kundendaten werden innerhalb des Systems abgelegt, also in einem ILF (*Internal Logical File*) (/F70/).

Komplexität bewerten Nachdem so die Elementarprozesse klassifiziert wurden, wird als nächstes ihre Komplexität bewertet. Dazu müssen jeweils zwei der folgenden Elementtypen gezählt werden:

- Ein **Datenelement** DET (*Data Element Type*) ist ein für den Benutzer erkennbares, eindeutig bestimmbares, nicht-rekursives Feld. Dazu gehören zum Beispiel Eingabefelder in einer GUI, aber auch Datenfelder einer Datei. DETs werden für jeden Funktionstyp gezählt.
- Ein **referenzierter Datenbestand** FTR (*File Type Referenced*) ist ein Datenbestand (ILF oder EIF), der von einer Transaktion (EI, EO, EQ) verwendet wird.
- Eine **Feldgruppe** RET (*Record Element Type*) ist eine vom Benutzer erkennbare, logisch zusammengehörige Gruppe von Datenelementen innerhalb eines Datenbestands. Die Feldgruppen werden für jeden internen oder externen Datenbestand (ILF oder EIF) gezählt.

Aus der Kombination von DETs mit FTRs (für Transaktionen) oder RETs (für Datenbestände) werden dann aus einer von fünf Komplexitätsmatrizen die *Function Points* abgelesen. Den Aufbau und den Inhalt der Matrizen zeigen die Abb. 23.5-2, die Abb. 23.5-3 und die Abb. 23.5-4.

23.5 Die Function-Points-Methode IV

		DETs			
		EI	1-5	5-15	≥16
FTRs	1	3	3	4	
	2	3	4	6	
	≥ 3	4	6	6	

Abb. 23.5-2: Komplexitätsmatrix Externe Eingaben (EI): Datenelemente (DETs) – referenzierte Datenbestände (FTRs).

		DETs			
		EO	1-5	6-19	≥20
FTRs	0-1	4	4	5	
	2-3	4	5	7	
	≥ 4	5	7	7	

		DETs			
		EQ	1-5	6-19	≥20
FTRs	0-1	3	3	4	
	2-3	3	4	6	
	≥ 4	4	6	6	

Abb. 23.5-3: Komplexitätsmatrixen Externe Ausgaben (EO), externe Abfragen (EQ): Datenelemente (DETs) – referenzierte Datenbestände (FTRs).

		DETs			
		ILF	1-19	20-50	≥51
RETs	1	7	7	10	
	2-5	7	10	15	
	≥ 6	10	15	15	

		DETs			
		ELF	1-19	20-50	≥51
RETs	1	5	5	7	
	2-5	5	7	10	
	≥ 6	7	10	10	

Abb. 23.5-4: Komplexitätsmatrixen Interne Datenbestände (ILF), externe Datenbestände (EIF): Datenelemente (DETs) – Feldgruppen (RETs).

a Bei der Erfassung des neuen Kunden sind die Datenfelder die einzelnen Eingabefelder der Kundendaten (/F70/: Kunden-Nr., Name, Adresse, Kommunikationsdaten, Geburtsdatum, Funktion, Umsatz, Kurzmitteilung, Notizen, Info-Material, Kunde seit). In diesem Fall würden daher 11 DETs und 1 FTR (die Kundendatentabelle) gezählt, was laut EI-Matrix (Abb. 23.5-2) 3 FPs ergibt (2. Spalte, 1. Zeile).

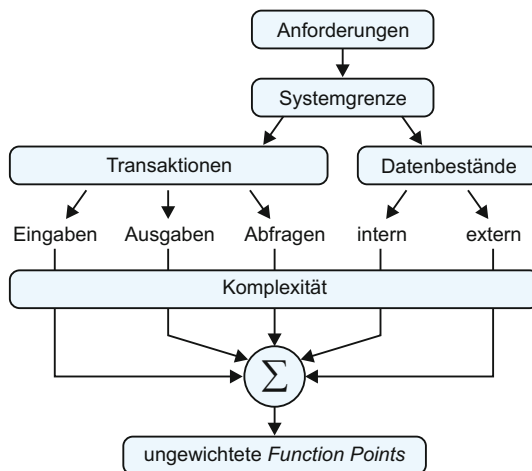
Beispiel 1c

IV 23 Schätzen des Aufwands

- b** Die Anzeige der Veranstaltungsliste betrifft die logischen Datencontainer Seminar und Veranstaltung (2 FTRs) und liest bis zu 17 DETs an Veranstaltungsdaten (/F100/) und 13 DETs an Seminar-daten (/F90/), was in der Summe 30 DETs ergibt. Laut EQ-Matrix (Abb. 23.5-3, rechts) ergibt dies 6 FP (3. Spalte, 2. Zeile).
- c** Beim Drucken ist die Anzahl der DETs und FTRs von den für den Druck zu lesenden Daten abhängig. Dies ist für SemOrg nicht genauer spezifiziert. Vermutlich werden die Daten über das gebuchte Seminar bzw. die Veranstaltung sowie die Daten des Teilnehmers bzw. Kunden und dessen Firma benötigt. Dies sind im Höchstfall 5 FTRs und 51 DETs, was laut EO-Matrix 7 FP ergibt – die höchstmögliche Wertung für Ausgabetransaktionen.
- d** Die Kundendaten (/F70/) bestehen aus 11 Feldern (DETs), die zu 2 logisch zusammengehörigen Feldern (RETs) zusammengefasst werden können: Stammdaten und Annotationen. Daraus ergeben sich laut ILF-Matrix (Abb. 23.5-4, links) insgesamt 7 FP.

Die so ermittelten *Function Points* werden über alle Elementarprozesse aufsummiert und ergeben so die **ungewichteten Function Points** (UFP). Diese stellen den Umfang der Anwendung dar. Der Prozess ist noch einmal in der Abb. 23.5-5 zusammengefasst. Sofern Zählregeln eingehalten werden, ist das Ergebnis einigermaßen eindeutig und vergleichbar. So stellte Kemerer bei Anwendung der IF-PUG 3.0-Zählregeln Abweichungen zwischen zwei Zählern von maximal 11 Prozent fest [Keme93].

Abb. 23.5-5:
Ermittlung der
ungewichteten
Function Points
auf Basis der
Anforderungen.



Beispiel 1d Für die Verwaltung der Kundendaten sind die Elementarprozesse gemäß der Tab. 23.5-1 zu berücksichtigen. Es ergibt sich für diesen Teil der Software ein Umfang von 26 *Function Points*.

Funktion/ Datenbestand	Typ	Anzahl RETs/FTRs	Anzahl DETs	FP
Kundendaten	ILF	3	11	7
Neuer Kunde	EI	2	7	4
Kundendaten aktualisieren	EI	2	7	4
Kundendaten löschen	EI	1	1	3
Kundenliste anzeigen	EQ	3	11	4
Kundendaten anzeigen	EQ	3	11	4
Summe				26

Tab. 23.5-1:
Function Points für
die Fallstudie
SemOrg.

In einem weiteren (optionalen) Schritt werden zusätzliche Einflüsse anhand von 14 Systemmerkmalen berücksichtigt. Diese umfassen vor allem technische Faktoren, aber auch nichtfunktionale Anforderungen. Sie können die UFP um $\pm 35\%$ verändern und ergeben die so genannten **gewichteten Function Points** (GFP). Die folgenden Systemmerkmale werden in der ursprünglichen Methode bei der Korrektur berücksichtigt und jeweils mit einem Wert zwischen 0 und 5 bewertet:

Gewichtete
Function Points

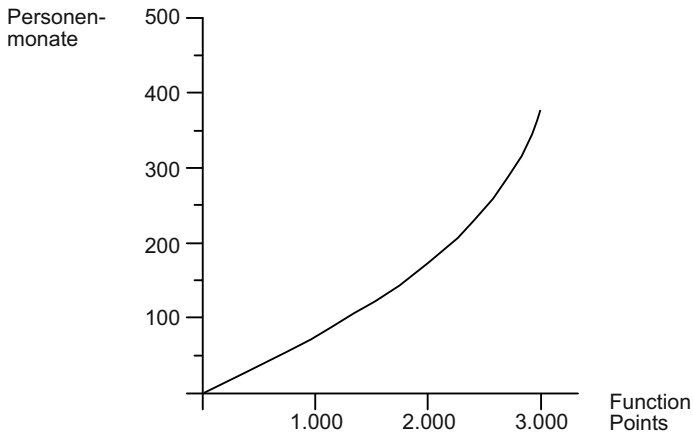
- **Datenkommunikation:** Das System kommuniziert mit anderen Systemen gemäß entsprechender Protokolle.
- **Verteilte Verarbeitung:** Das System ist auf mehrere Computersysteme verteilt. Dies kann die Verarbeitung und die Datenhaltung betreffen.
- **Performanceanforderungen:** Es werden besondere Anforderungen an schnelle Antwort- oder Durchlaufzeiten gestellt.
- **Ressourcennutzung:** Die Hardwareressourcen sind begrenzt, oder Teile der Anwendung stellen spezielle Anforderungen an den Prozessor.
- **Transaktionsrate:** Es muss mit einer so hohen maximalen Transaktionsrate gerechnet werden, dass spezielle Maßnahmen erforderlich sind, um diese verarbeiten zu können.
- **Online-Dateneingabe:** Das System soll interaktive Dateneingaben erlauben (im Gegensatz zum Batch-Betrieb).
- **Benutzungsfreundlichkeit:** Interaktive Funktionen sollen benutzerfreundlich realisiert werden, z.B. durch Einsatz von Navigationshilfen, Popup-Fenstern oder Mehrsprachigkeit.
- **Online-Update:** Die ILFs (*Internal Logical Files*) sollen zur Laufzeit interaktiv geändert werden können. Dabei sind unter Umständen Vorkehrungen gegen Datenverlust zu treffen.
- **Komplexe Verarbeitung:** Dazu gehören beispielsweise komplexe logische, mathematische oder Fehlerbehandlungs-Operationen.
- **Wiederverwendbarkeit:** Der Code soll in zukünftigen Systemen wiederverwendbar sein.

IV 23 Schätzen des Aufwands

- Migrations-/Installationshilfen: Bestehende Daten sollen ins neue System konvertiert werden können.
- Betriebshilfen: Vollautomatischer Betrieb mit entsprechenden Datensicherungs- und Wiederherstellungsfunktionen wird unterstützt.
- Mehrfachinstallationen: Das System soll in verschiedenen Hard- und Softwareumgebungen lauffähig sein.
- Änderungsfreundlichkeit: Das System unterstützt flexible Abfragen und Berichte. Steuerdaten können vom Benutzer konfiguriert werden.

Das Ergebnis kann schließlich als Eingabe für COCOMO II (siehe »COCOMO II«, S. 536) dienen (ungewichtete FP) oder per Erfahrungskurve in Aufwand umgerechnet werden (gewichtete FP). Die Abb. 23.5-6 zeigt die 1983 von IBM veröffentlichte Umrechnungskurve. Diese sollte keinesfalls direkt angewendet werden, veranschaulicht aber das typische Aussehen.

Abb. 23.5-6: Die »IBM-Kurve« zeigte 1983 erstmals den Zusammenhang zwischen Function Points und Personenmonaten.



Erst durch Erstellung und Verwendung eigener Erfahrungskurven lassen sich brauchbare Schätzungen erstellen. Alternativ dazu können die *Function Points* auch per Daumenregel direkt in Personenmonate oder Codezeilen umgerechnet werden. So entspricht 1 *Function Point* etwa 128 Zeilen C-Code oder 53 Zeilen Java-Code. Umrechnungsfaktoren für weitere Sprachen sind in der Tab. 23.5-2 angegeben [Jone95].

Weitere Faustregeln, basierend auf gewichteten *Function Points* (GFP) [Jone07]:

Entwicklungsdauer = $GFP^{0,4}$

Personen = $GFP / 150$ (aufgerundet)

Aufwand (PM) = Personen · Entwicklungsdauer

Wartungspersonal = $GFP / 750$

Produktivität: 10 FP pro PM

23.6 Object Points/Application Points IV

Sprache	LOC/UFP	Sprache	LOC/UFP
Access	38	HTML 3.0	15
Ada 95	49	Java	53
Assembler	320	Pascal	91
C	128	PERL	27
C++	55	Shell Script	107
Cobol	91	SQL	13
Tabellenkalkulation	6	Visual Basic 5	29
4GL	20	Visual C++	34

Tab. 23.5-2:
Faustregeln für
LOC pro Function
Point [Jone95].

Kosten: 1.200 \$ pro FP

Die Function-Points-Methode wird – trotz ihrer Abstammung aus der IT-Welt von vor 30 Jahren – auch heute noch verbreitet eingesetzt. Sie wird von der IFPUG weiterentwickelt und standardisiert. Daneben existieren knapp 40 weitere Varianten der Function-Points-Methode, die teilweise im Hinblick auf bestimmte Domänen (etwa Echtzeitsysteme) erweitert wurden. Ein Beispiel für eine solche Variante ist COSMIC (*Common Software Measurement International Consortium*). Die Notwendigkeit dieser Varianten ist allerdings umstritten [Jone07].

- + Die Erstellung eigener Erfahrungskurven und Einhaltung einheitlicher Zählregeln vorausgesetzt, lassen sich mit der Function-Points-Methode sehr gute Ergebnisse erzielen [Keme93].
- Nachteile sind der recht hohe Aufwand für die Zählung und die subjektive Bewertung der Systemmerkmale.

[ÖzMe06], [PoBo05]

Weiterführende
Literatur

23.6 Object Points/Application Points

Der hohe Aufwand für eine exakte Function-Points-Zählung führte im Rahmen von COCOMO II (siehe »COCOMO II«, S. 536) zur Einführung einer vereinfachten Methode – der Zählung von *Object Points*. Diese werden auch *Application Points* genannt, um Assoziationen zur objektorientierten Programmierung zu vermeiden. Diese Methode ist für 4GL-Projekte³ gedacht, das heißt für Projekte, in denen Bericht- und Eingabeformulargeneratoren eingesetzt werden, um die Menge des von Hand zu erstellenden Codes zu verringern. Die Zählung betrachtet im Wesentlichen die Anzahl verschiedener Eingabeformulare (*Screens*) und Berichte (*Reports*) sowie die Anzahl zusätzlich erforderlicher traditioneller – ausprogrammierter – 3GL-Module⁴. Für

³ 4GL: Programmiersprachen der 4. Generation, im Wesentlichen Sprachen zur Abfrage von Datenbanken, z. B. SQL.

⁴ 3GL: Programmiersprachen der 3. Generation, das sind prozedurale Programmiersprachen wie z. B. Pascal, Ada oder C.

IV 23 Schätzen des Aufwands

jedes Formular und jeden Bericht wird die Komplexität bewertet – zum Beispiel aufgrund der Menge und unterschiedlichen Zugehörigkeit oder Herkunft (Datenbanken bzw. Tabellen) der Daten. Aus der Tab. 23.6-1 können dann direkt die entsprechenden *Object Points* abgelesen werden. Die zusätzlichen 3GL-Module werden jeweils konstant mit 10 *Object Points* gezählt. Die Gesamtsumme ergibt ein Maß für den Umfang der Anwendung.

Tab. 23.6-1:
Bestimmung der
Object Points.

	Geringe Komplexität	Mittlere Komplexität	Hohe Komplexität
Formular	1	2	3
Bericht	2	5	8
3GL-Komponente			10

Beispiel Enthält das zu erstellende Produkt 7 Eingabeformulare mittlerer Komplexität, 2 Formulare hoher Komplexität, 8 Berichte mittlerer Komplexität sowie 4 3GL-Module, so ergeben sich:
 $7 \cdot 2 + 2 \cdot 3 + 8 \cdot 5 + 4 \cdot 10 = 100$ *Object Points*

Mit *Object Points* wird für 4GL-Projekte mit etwa halbem Aufwand eine zu *Function Points* vergleichbare Präzision erreicht [BKK91]. *Object Points* werden vor allem in der frühen Prototypenstufe von CO-COMO II eingesetzt.

23.7 COCOMO II

Das Kostenschätzungsmodell COCOMO (*Constructive Cost Model*) wurde 1981 von Barry Boehm vorgestellt [Boeh81]. 1995 entwickelte er eine überarbeitete Variante, die die Veränderungen in der Softwareentwicklungswelt berücksichtigt [BCH95], [Boeh+00]. Diese zweite Version wird hier vorgestellt.

COCOMO II bietet drei verschiedene Modelle an, die sich nach dem Zeitpunkt der Schätzung unterscheiden und für spätere Zeitpunkte zunehmend genauer werden:

- Frühe Prototypenstufe
- Frühe Entwurfsstufe
- Stufe nach Architekturentwurf

Die **frühe Prototypenstufe** (*Application Composition Model*) basiert auf einer Zählung der *Object Points* (siehe »Object Points/Application Points«, S. 535), einer Einschätzung der Produktivität (Entwickler und Werkzeuge) sowie dem aus früheren Projekten wiederverwendeten Anteil. Aus diesen Größen wird der Aufwand ermittelt, indem der Umfang zur Produktivität in Beziehung gesetzt wird:

$$PM = NOP / PROD$$

$$NOP = OP \cdot (100 - \%reuse) / 100$$

Dabei sind OP die gezählten *Object Points*, *%reuse* ist der Anteil, der durch Wiederverwendung abgedeckt werden kann, NOP sind die *New Object Points* (OP korrigiert bezüglich Wiederverwendung), PROD ist die Produktivität und PM ist der geschätzte Aufwand in Personenmonaten. Die angenommene Produktivität PROD kann zwischen 4 und 50 NOP pro Personenmonat variieren – also um bis zu einem Faktor 12. Sie wird anhand der Tab. 23.7-1 abgeschätzt.

	--	-	0	+	++
Erfahrung/Fähigkeiten der Entwickler	4	7	13	25	50
Reife/Fähigkeiten der CASE-Werkzeuge	4	7	13	25	50

Tab. 23.7-1: Ermittlung der Produktivität. Die angegebenen Werte sind NOP pro PM. PROD ergibt sich als Durchschnitt der beiden Einzelfaktoren.

Wird ein Projekt zur Erstellung eines Prototypen auf 100 *Object Points*, 25 % Wiederverwendungsanteil und eine mittlere Produktivität (PROD = 13) geschätzt, so ergibt sich:

$$\text{New Object Points } NOP = OP \cdot (100\% - \%reuse) = 75$$

$$PM = NOP / PROD = 5,8$$

Der erwartete Aufwand beläuft sich demnach auf knapp 6 Personenmonate.

Diese Stufe ist mit größter Vorsicht zu benutzen. Bei der Softwareentwicklung steigt mit zunehmender Entwicklerzahl der Kommunikationsaufwand überproportional an. Ein lineares Modell kann daher keine brauchbare Prognose liefern. Auch ist die Annahme, dass Wiederverwendung ohne jeglichen Aufwand erfolgen kann, unrealistisch.

In der **frühen Entwurfsstufe** wird von den *Function Points* bzw. daraus abgeleiteter Codegröße ausgegangen. Die Formel zur Berechnung der **Personenmonate** PM sieht so aus:

$$PM = A \cdot KLOC^E \cdot EM + PM_m$$

KLOC (*Kilo Lines of Code*) ist dabei der erwartete Umfang der Software (wie viele 1.000 Zeilen Code), PM_m ist ein Korrekturfaktor für generierten Code, und $A=2,94$ ein konstanter Erfahrungswert, den Boehm über eine große Anzahl von Projekten ermittelt hat (alle Werte aus der letzten Kalibrierung in 2000, siehe [Boeh00]). In den Exponent E gehen Informationen über die Erfahrung mit der Domäne, die Flexibilität des Entwicklungsprozesses, die Qualität des Risikomanagements und Entwurfs, den Teamzusammenhalt und die Prozessreife ein. Diese fünf Kriterien (*Cost Driver*) werden – mit Hilfe von weiter aufgegliederten Fragestellungen – jeweils mit einem von sechs Werten zwischen sehr gut/viel und sehr schlecht/wenig beurteilt, die Bewertung per Tabelle in einen Faktor w_i umgesetzt

Frühe Entwurfsstufe

IV 23 Schätzen des Aufwands

und alle Bewertungen aufsummiert. Die Tab. 23.7-2 gibt einen Überblick über die Faktoren und Wertebereiche. Die genauen Kriterien und Zwischenwerte sind dem »COCOMO II Model Definition Manual« zu entnehmen [Boeh00]. Der Exponent ergibt sich dann wie folgt:

$$E = B + \sum w_i / 100$$

E kann derzeit (Kalibrierung 2000) mit $B=0,91$ Werte bis 1,23 annehmen und setzt für Werte größer 1 den mit der Problemgröße superlinear steigenden Aufwand um. Für sehr gute Bewertungen und damit Werte kleiner 1 kann der Aufwand allerdings auch sublinear steigen.

Tab. 23.7-2: Cost Drivers: Die Summanden für den Exponenten E. Es sind jeweils der minimal und maximal mögliche Wert angegeben.

Kürzel	Beschreibung	Min.	Max.
PREC	Ähnlichkeit zu vorigen Projekten	0	6,20
FLEX	Flexibilität in der Entwicklung bzgl. Anforderungen, externen Schnittstellen, Zeitplan	0	5,07
RESL	(a) Risikomanagement (b) Aufwand für Architektur	0	7,07
TEAM	Teamzusammenhalt und Kooperationsbereitschaft	0	5,48
PMAT	CMM-Level oder mittlerer Erfüllungsgrad der 17 KPAs (Key Process Areas)	0	7,80

Beispiel Ein Projekt wird von einer Firma bearbeitet, die schon einige sehr ähnliche Projekte abgewickelt hat (PREC = 1,24). Es gibt in dem Projekt sehr genaue Vorgaben, die kaum Abweichungen erlauben (FLEX = 4,05). Risikomanagement wird praktiziert (RESL = 2,83). Das Projektteam arbeitet an einem Ort zusammen und kennt sich bereits (TEAM = 2,19). Die Abteilung hat einen geringen Entwicklungsprozessreifeegrad (PMAT = 6,24). Dann ergibt sich der Exponent E zu:

$$E = B + (\text{PREC} + \text{FLEX} + \text{RESL} + \text{TEAM} + \text{PMAT} / 100$$

$$= 0,91 + (1,24 + 4,05 + 2,83 + 2,19 + 6,24) / 100 = 1,0755$$

Im Multiplikator EM (*Effort Multiplier*) gehen Abschätzungen über Produktgüte und -komplexität, Plattformkomplexität, Fähigkeiten und Erfahrung des Personals, Zeitplan und Infrastruktur ein. Auch diese Kriterien sind zwecks besserer Bewertbarkeit weiter in Unterpunkte gegliedert. Die bis zu 6-stufige Bewertung wird in Tabellen nachgeschlagen. Der Normalfall wird jeweils mit 1,0 bewertet. Die Tab. 23.7-3 gibt auch für diese Faktoren einen Überblick. Die Faktoren für die frühe Entwurfsstufe sind jeweils fett gedruckt, darunter befinden sich die feiner aufgegliederten Faktoren, die in der Stufe nach Architekturentwurf zur Anwendung kommen. In der Tabelle sind auch jeweils die Mindest- und Höchstwerte angegeben. Sie geben einen Eindruck davon, wie groß der potenzielle Einflussgrad des jeweiligen Faktors auf den Aufwand ist. So haben die nichtfunktionalen Anforderungen insgesamt den größten Einfluss, während

23.8 Bewertung und weitere Aspekte IV

die geforderte Wiederverwendung vergleichsweise geringe Auswirkungen zeigt. Durch Multiplikation aller dieser Einzelfaktoren erhält man schließlich den Gesamtwert für EM.

Die Entwicklungszeit TDEV in Monaten ergibt sich aus folgender Formel:

$$\text{TDEV} = C \cdot \text{PM}^{(D + 0,2 \cdot (E-B))}$$

mit $C = 3,67$ und $D = 0,28$.

Die Anzahl benötigter Entwickler N ergibt sich damit zu:

$$N = \text{PM} / \text{TDEV}$$

Bei einem Projekt mit geschätzten 100.000 LOC ist der Aufwand im bestmöglichen Fall ($E = B$, $EM = 1,0$):

$$\text{PM} = 2,94 \cdot 100^E \cdot EM = 2,94 \cdot 100^{0,91} \cdot 1,0 = 194 \text{ PM}$$

$$\text{TDEV} = 3,67 \cdot \text{PM}^{(0,28 + 0)} = 16 \text{ Monate}$$

$$N = \text{PM} / \text{TDEV} = 12 \text{ Entwickler}$$

Steht nun weniger Zeit zur Verfügung als eigentlich gebraucht wird, so können die Auswirkungen über den SCED-Parameter analysiert werden. Angenommen es stehen nur 12 statt 16 Monate zur Verfügung, so bedeutet dies eine Straffung auf 75 %. Es ergibt sich (laut Tabelle) ein SCED-Faktor von 1,43, der zur Korrektur des nominalen Modells noch mit dem errechneten Aufwand multipliziert werden muss, und damit:

$$\text{PM} = 2,94 \cdot 100^{0,91} \cdot 1,43 = 278 \text{ PM}$$

$$N = \text{PM} / \text{TDEV} = 278 \text{ PM} / 12 \text{ Monate} = 23 \text{ Entwickler}$$

Die Verkürzung um 25 % erfordert also eine Verdoppelung der eingesetzten Ressourcen und bedeutet eine Steigerung des Aufwands um 43 %.

In der letzten Stufe von COCOMO II – der **Stufe nach Architekturentwurf** – werden zusätzlich die Auswirkungen erwarteter Änderungen von Anforderungen, das Ausmaß und der Aufwand für Wiederverwendung und Codegenerierung sowie weiter verfeinerte lineare Einflussfaktoren berücksichtigt. Die verfeinerten Faktoren finden sich in der Tab. 23.7-3 jeweils unter den zugehörigen zusammengefassten Faktoren der frühen Entwurfsstufe.

[BoVa08]

Stufe nach
Architektur

Weiterführende
Literatur

23.8 Bewertung und weitere Aspekte

Es wurden eine ganze Reihe von Schätzmethoden vorgestellt. Welches ist nun die Methode der Wahl, oder unter welchen Umständen ist welche Methode zu bevorzugen?

Eine allgemeingültige Empfehlung lässt sich dazu nicht aussprechen. Es hängt vom Zweck der Schätzung und von den für die Schätzung verfügbaren Ressourcen ab, welche Schätzung im konkreten

IV 23 Schätzen des Aufwands

Tab. 23.7-3:
Faktoren des
Effort Multiplier.
Fett gedruckt sind
die
Oberkategorien,
die in der frühen
Entwurfsstufe
benutzt werden.
Jeweils darunter
sind die
detaillierteren
Faktoren für die
Stufe nach Archi-
tekturentwurf
angegeben.

Kürzel	Beschreibung	Min.	Max.
RCPX	Produkteigenschaften	0,49	2,72
RELY	Erforderliche Zuverlässigkeit	0,82	1,26
DATA	Datenbankgröße für vollständigen Test	0,90	1,28
CPLX	Produktkomplexität	0,73	1,74
DOCU	Qualität der Dokumentation (gesamter Software-Lebenszyklus)	0,81	1,23
RUSE	Geforderte Wiederverwendbarkeit	0,95	1,24
PDIF	Plattform	0,87	2,61
TIME	Rechenzeitbeschränkungen	1,00	1,63
STOR	Hauptspeicherbeschränkungen	1,00	1,46
PVOL	Plattform-Volatilität	0,87	1,30
PERS	Personalfaktoren	0,50	2,12
ACAP	Fähigkeiten der Analysten (Spezifikation, Entwurf)	0,71	1,42
PCAP	Fähigkeiten der Programmierer	0,76	1,34
PCON	Personalfluktuations	0,81	1,29
PREX	Erfahrung der Entwickler	0,62	1,59
AEXP	Erfahrung mit Anwendungstyp	0,81	1,22
PLEX	Erfahrung mit Plattform	0,85	1,19
LTEX	Erfahrung mit Sprache/Werkzeugen	0,84	1,20
FCIL	Projektfaktoren	0,62	1,43
TOOL	Fähigkeiten/Fortschrittlichkeit der eingesetzten Werkzeuge	0,78	1,17
SITE	Verteilte Entwicklung (Kommunikationsmittel)	0,80	1,22
SCED	Straffheit des Zeitplans	1,00	1,43

Fall geeignet ist. Im Idealfall empfiehlt es sich, **mehrere der Verfahren** anzuwenden und so zu voneinander unabhängigen Ergebnissen zu kommen. Sind diese ähnlich, so steigt das Vertrauen in die Schätzung. In der Praxis werden die Kosten für ein derartiges Vorgehen aber zu hoch sein.

Die Verfahren unterscheiden sich durch unterschiedlichen Aufwand sowie verschiedene Voraussetzungen für ihren Einsatz voneinander. So kann die Experten- oder Analogie-Schätzung gut und günstig sein, wenn ein entsprechender Experte verfügbar ist bzw. Daten von sehr ähnlichen Projekten vorliegen. Die Bottom-up-Schätzung setzt die Zerlegung des Systems in Subsysteme oder Komponenten voraus, die ihrerseits sehr aufwendig oder zu spät verfügbar

sein kann. Die algorithmischen Verfahren sind mit Abstand am aufwendigsten, liefern aber auch die objektivsten Ergebnisse. Auf die Faustregeln sollte aufgrund ihrer Ungenauigkeit nur im Notfall zurückgegriffen werden.

Üblicherweise wird bei der Aufwandsschätzung nur an die Erstentwicklung der Software bis zur Auslieferung gedacht. Die Wartungsphase wird oft völlig außer acht gelassen, obwohl diese 60–80% des Aufwands im Softwarelebenszyklus ausmacht [Snee97]. Daher ist es angebracht, sich auch mit der Schätzung des Wartungsaufwands zu beschäftigen. Dieser muss zumindest teilweise in die Projektkosten einkalkuliert werden: Die Fehlerbehebung macht ca. 20% des Wartungsaufwands aus [a. a. O.]. Kosten für zusätzlich gewünschte Funktionen oder Anpassungen werden dagegen in der Regel separat beauftragt. Diese können mit einer der vorgestellten Methoden geschätzt werden. Diese Schätzung ist sogar einfacher als die eines neuen Projekts, denn es liegen in der Regel bereits umfangreiche Erfahrungen mit dem Kunden und der Domäne vor, es sind historische Daten über den Projektverlauf verfügbar, und die neuen Anforderungen sind meist recht konkret und von geringem Umfang. Andererseits schließen Wartungsarbeiten sehr viele unterschiedliche Aspekte ein, die unterschiedlich gehandhabt werden müssen [Jones07]. Und schließlich wird die Wartung um so aufwendiger, je älter das Produkt wird. Dies liegt vor allem daran, dass die ursprünglichen Entwickler nicht mehr verfügbar sind, die eingesetzten Techniken möglicherweise nicht mehr aktuell und damit bekannt sind, und sich die Qualität der Software durch ständige Wartungsarbeiten verschlechtert hat.

Wartung

Eine Schätzung ist im Allgemeinen nur dann realistisch möglich, wenn es sich nicht um etwas völlig Neues handelt. Wenn man etwas völlig Neues macht, von dem noch nicht einmal bekannt ist, ob es überhaupt realisierbar ist, handelt es sich um Forschung. Hier gelten andere Regeln. Eine Möglichkeit für solche Projekte ist es, zunächst nur den Rahmen für den Aufwand bzw. die Kosten zu stecken – was genau das Ergebnis ist, ergibt sich dann erst im Laufe des Projekts in Abstimmung der beteiligten Parteien. Es bietet sich bei diesem Modell eine enge Zusammenarbeit und ein inkrementeller Entwicklungsprozess an. Diese Vorgehensweise ist auf eine vertrauensvolle Zusammenarbeit zwischen Auftraggeber und Entwicklungsorganisation angewiesen. Sie wird häufig zwischen Forschungseinrichtungen und Unternehmen praktiziert. Ähnliche Ideen werden beispielsweise auch beim *Extreme Programming* angewandt.

Forschung

[Snee03], [BSB08]

Weiterführende
Literatur

23.9 Zusammenfassung

In den Beschreibungen der verschiedenen Schätzverfahren wurde deutlich, dass sie alle auf **Erfahrung** basieren. Es ist also notwendig, Erfahrungen zu sammeln. Die Schätzung hängt stark von der Entwicklungsorganisation ab, z. B. von ihrer Produktivität, ihren Prioritäten und Zielen. Konkret bedeutet dies, dass die Messung und Auswertung von Projekten der eigenen Organisation nötig ist, damit diese als Basis für zukünftige Schätzungen dienen können. Ist dies nicht der Fall, so ist man auf die undokumentierte Erfahrung von Experten angewiesen.

Gute Schätzungen sind *nicht* billig zu haben. Verfahren wie die Function-Points-Zählung sind teuer, und die Zählung bereits abgeschlossener Projekte ist zunächst einmal eine Investition in die Qualität zukünftiger Schätzungen. Und auch eine Expertenschätzung bindet die Produktivität hochbezahlter Mitarbeiter. Billige Alternativen wie die Faustregeln liefern entsprechend auch nur sehr vage Ergebnisse.

Eine Erkenntnis, die ausgenutzt werden sollte, ist die steigende Genauigkeit der Schätzung mit fortschreitendem Projektverlauf. Die initiale Schätzung bzw. die ihr zugrunde liegenden Annahmen sollten daher regelmäßig überprüft und angepasst werden, um Abweichungen vom Zeitplan und damit drohende Budgetüberschreitungen rechtzeitig erkennen zu können. Diese Maßnahme kann somit das Risikomanagement unterstützen.

Insgesamt lässt sich feststellen, dass systematische Aufwandschätzung sehr nützlich sein kann. Um sie zu ermöglichen, muss allerdings kontinuierlich ein gewisser Aufwand betrieben werden, um Projektdaten zu erfassen und eine Erfahrungsdatenbank aufzubauen. Mittels dieser Daten lassen sich dann neue Projekte recht zuverlässig abschätzen – wenn sie nicht völlig anders als alle bereits abgeschlossenen Projekte sind.

24 Anforderungen priorisieren

Bei der Anforderungsermittlung werden in der Regel *viele* Anforderungen aufgestellt. Aber nicht jede Anforderung ist gleich wichtig.

Nach welchen Kriterien können Prioritäten für Anforderungen vergeben werden?

Frage

Ein Kriterium ist sicher die **Wichtigkeit** einer Anforderung. Wichtigkeit kann aber Verschiedenes bedeuten: Wichtigkeit bezogen auf die Funktionalität des Systems, Wichtigkeit bezogen auf die Benutzungsfreundlichkeit, Wichtigkeit bezogen auf die frühzeitige Verfügbarkeit usw.

Antwort

Anforderungen können auch nach Kosten, Volatilität, Risiko hinsichtlich der Akzeptanz des Systems, Schaden bezogen auf die Nachteile, wenn die Anforderung nicht berücksichtigt wird, usw. gewichtet werden. Eine Priorisierung anhand eines Kriteriums ist in der Regel relativ einfach möglich. Werden mehrere Kriterien berücksichtigt, dann steigt der Aufwand mit jedem zusätzlichen Kriterium.

Klassifikation nach einem Kriterium

Häufig werden Anforderungen nach einem Kriterium priorisiert. In [IEEE830, S. 13] wird als Kriterium die **Notwendigkeit** (*necessity*) mit folgenden Ausprägungen vorgeschlagen:

- **Essenziell** (*essential*): Die Software wird nicht akzeptiert, wenn die Anforderung nicht in der geforderten Weise realisiert wird.
- **Bedingt notwendig** (*conditional*): Die Anforderung wertet die Software auf. Wird sie nicht realisiert, dann wird die Software aber nicht unakzeptabel.
- **Optional** (*optional*): Die Realisierung der Anforderung kann wertvoll sein, muss es aber nicht. Die Anforderung gibt dem Auftragnehmer die Möglichkeit, über die vorhandenen Anforderungen hinauszugehen.

Erfahrungen haben gezeigt, dass die Verwendung dieser Ausprägungen dazu führt, dass die meisten Anforderungen mit **essenziell** gekennzeichnet werden, während optionale Anforderungen nur selten vorkommen.

Die Kano-Klassifikation

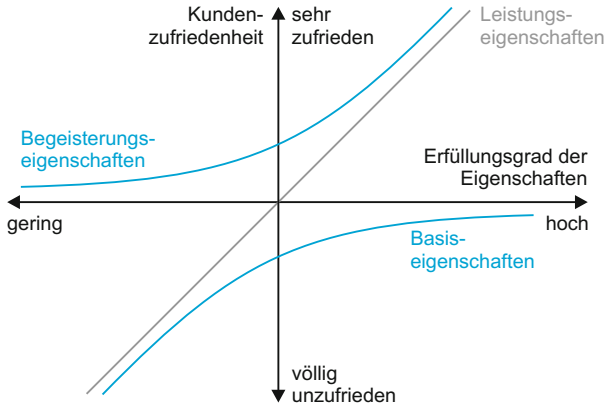
N. Kano hat festgestellt, wie Kundenwünsche und Produkteigenschaften zusammenhängen [KTS+84]. Die Produkteigenschaften werden in Kategorien eingeteilt:

IV 24 Anforderungen priorisieren

- **Basiseigenschaften:** Vom Kunden selbstverständlich vorausgesetzte Eigenschaften (implizite Erwartung). Fehlt eine Basiseigenschaft, dann entsteht Unzufriedenheit. Werden sie erfüllt, dann entsteht aber *keine* Zufriedenheit.
- **Leistungseigenschaften:** Vom Kunden bewusst geforderte Eigenschaften (Sonderausstattung!). Sie schaffen beim Kunden Zufriedenheit bzw. beseitigen Unzufriedenheit – je nach Ausmaß.
- **Begeisterungseigenschaften:** Eigenschaften, die der Kunde *nicht* erwartet hat. Die Kundenzufriedenheit wächst überproportional, wenn die Eigenschaften vorhanden sind.

Die Abb. 24.0-1 zeigt die Kundenzufriedenheit in Abhängigkeit von den erfüllten Produkteigenschaften.

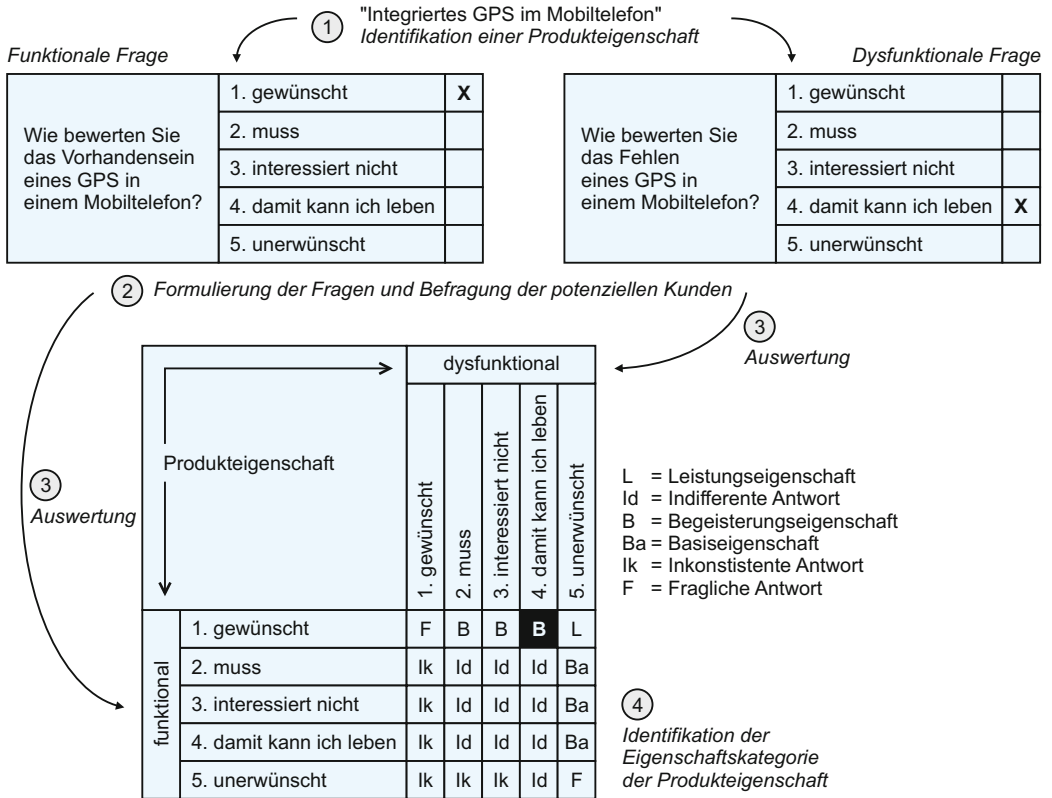
Abb. 24.0-1: Kundenzufriedenheit in Abhängigkeit von den Produkteigenschaften im Kano-Modell.



Wie die obere Kurve in der Abb. 24.0-1 zeigt, steigt die Kundenzufriedenheit überproportional mit der Anzahl realisierter Begeisterungseigenschaften. Im Laufe der Zeit werden aus Begeisterungseigenschaften Leistungseigenschaften und dann Basiseigenschaften. Etwas ursprünglich Besonderes wird mit der Zeit zu etwas Gewohntem – insbesondere wenn es nicht nur ein Anbieter, sondern im Laufe der Zeit alle Anbieter realisieren.

Beispiel Die Abb. 24.0-2 zeigt, wie eine Produkteigenschaft – hier »Integriertes GPS im Mobiltelefon« – kategorisiert wird. Zu der Produkteigenschaft werden im Schritt (1) funktionale Fragen (Wie bewertet der Kunde das Vorhandensein der Eigenschaft?) und dysfunktionale Fragen (Wie bewertet der Kunde das Nichtvorhandensein der Eigenschaft?) formuliert. Im Schritt (2) beantwortet der Kunde die Fragen. Die Auswertung im Schritt (3) führt zur Identifikation der Eigenschaftskategorie – hier handelt es sich um eine Begeisterungseigenschaft.

24 Anforderungen priorisieren IV



Das Kano-Modell kann auch effektiv zur Priorisierung von Anforderungen benutzt werden. Die Kategorisierung der Anforderungen kann durch jeden *Stakeholder* einzeln oder in einer Gruppensitzung gemeinsam erfolgen.

Abb. 24.0-2:
Beispiel für eine Eigenschaftsklassifikation nach dem Kano-Modell.

Weitere Priorisierungsmethoden

- **Ad-hoc-Anordnung** (*ad hoc ranking*): Die Anforderungen werden von einem *Stakeholder* oder einer Gruppe nach einem Kriterium in eine Rangfolge gebracht.
- **Top-Ten-Methode** (*top ten method*): n Anforderungen (z. B. 10) werden bezogen auf ein Kriterium ausgewählt und in eine Rangfolge gebracht.
- **Priorisierungsmethode nach Wiegers**: Es wird davon ausgegangen, dass eine Anforderung, wenn sie realisiert ist, proportional zu deren Nutzen und umgekehrt proportional zu den Kosten, dem Nachteil und dem Risiko der Anforderung ist. Mit Hilfe einer Priorisierungsmatrix werden die Anforderungsprioritäten berechnet [Wieg99].

IV 24 Anforderungen priorisieren

- **Kosten-Wert-Analyse** (*cost-value-analysis*): Durch paarweise Vergleiche der zu priorisierenden Anforderungen werden sowohl deren relativer Wert als auch deren relative Kosten ermittelt. Anschließend werden diese relativen Werte pro Anforderung in ein Koordinatensystem eingetragen und dann die Anforderungen einer von drei Prioritätsklassen zugeordnet [KaRy97].
- **VOP** (*Value-Oriented Framework*): Quantitative Methode mit Schwerpunkt auf dem Geschäftsnutzen [ASC07].

Die verschiedenen Priorisierungsmethoden können auch kombiniert eingesetzt werden.

[Pohl07], [BICI08]

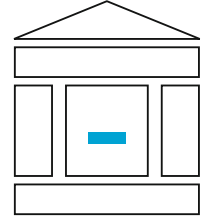
Weiterführende
Literatur

25 Anforderungen modellieren

Ausgehend von der abgenommenen Anforderungsspezifikation kann ein *Requirements Engineering*-Team die **fachliche Lösung** erarbeiten. Die fachliche Lösung muss folgende Anforderungen erfüllen:

- Korrekte Umsetzung der Anforderungen
- Vollständige Umsetzung der Anforderungen
- Referenzierung der Anforderungen
- Präzise, eindeutige und konsistente Formulierung der fachlichen Lösung: Die Präzision muss einen Grad an Formalität haben, so dass zumindest Teile davon automatisch mit Generatoren in eine technische Lösung und eine Implementierung umgesetzt werden können.

In der Regel wird eine fachliche Lösung – auch Produktmodell oder Anwendungsmodell genannt – erstellt.



»**Modell:** Idealisierte, vereinfachte, in gewisser Hinsicht ähnliche Darstellung eines Gegenstands, Systems oder sonstigen Weltausschnitts mit dem Ziel, daran bestimmte Eigenschaften des Vorbilds besser studieren zu können.

Anwendungsmodell: Modell, das sich auf einen Weltausschnitt bezieht, der Gegenstandsbereich einer Software-Anwendung ist« [HBB+94, S. 98].

Definitionen

Unter **Modellieren** versteht man dementsprechend das Erstellen und Modifizieren von Modellen.

Wird eine objektorientierte Analysemethode verwendet, dann ist es das Ziel ein objektorientiertes Modell zu erstellen. Als Beschreibungsmittel können die UML oder eine domänenspezifische Sprache eingesetzt werden. OOA

Beispielhaft wird eine OOA-Methode unter Verwendung der UML beschrieben: i

- »Beispiel: Objektorientierte Analyse«, S. 548

Auf die Vor- und Nachteile einer domänenspezifischen Sprache wird gesondert eingegangen:

- »Domänenspezifische Sprachen«, S. 563

Erfahrungsgemäß ergeben sich bei der Modellierung eine Vielzahl von Rückfragen bei den *Stakeholdern*, die beantwortet werden müssen und dann in die Modellierung einfließen.

IV 25 Anforderungen modellieren

Die Modellierung der fachlichen Lösung wird in der Regel durch weitere Artefakte ergänzt. Parallel oder zeitlich verzögert zur Modellierung wird oft das Konzept der Benutzungsoberfläche erstellt und bisweilen durch einen Prototyp ergänzt. Erfahrungsgemäß hat die Betrachtung der Benutzungsoberfläche Rückwirkungen auf das Produktmodell. Beispielsweise ergeben sich aus der Benutzungsoberfläche zusätzlich benötigte Attribute, die in das Produktmodell nachgetragen werden müssen. Wird für das Produkt ein Benutzerhandbuch benötigt, dann empfiehlt es sich ebenfalls, dieses Handbuch bereits zeitlich versetzt zur Konzeption der Benutzungsoberfläche in den Grundzügen zu erstellen.

Weitere Schätzung
des Aufwands

Auf der Basis der fachlichen Lösung kann eine verfeinerte Aufwandsschätzung durchgeführt werden, die teilweise sogar automatisch erfolgen kann, z. B. aus UML-Diagrammen (siehe »Schätzen des Aufwands«, S. 515).

Ergebnisse

Folgende Artefakte stehen nach der Modellierung zur Verfügung:

- Produktmodell bzw. fachliche Lösung
- evtl. Konzept der Benutzungsoberfläche oder ein Oberflächenprototyp
- evtl. Konzept des Benutzerhandbuchs

Agile Entwicklung

Bei einer agilen Softwareentwicklung wird in der Regel auf eine Modellierung verzichtet. Anhand der Spezifikation – d. h. anhand der *User Stories* – erfolgt eine direkte Programmierung (siehe »Schablonen für agile Entwicklungen«, S. 497).

Weiterführende
Literatur

[FHR08]

25.1 Beispiel: Objektorientierte Analyse

Eine heute weitverbreitete Methode, um von den Kundenanforderungen zu einer fachlichen Lösung zu gelangen, ist OOA – die objektorientierte Analyse.

Dabei muss unterschieden werden zwischen der **OOA-Methode**, um von den Anforderungen zur fachlichen Lösung zu gelangen, und dem Ergebnis der Analyse, dem **OOA-Modell**.

i Voraussetzung für eine Methode ist eine Strukturierung der OOA-Konzepte:

- »Strukturierung der OOA-Konzepte«, S. 549

Bei der Modellierung treten immer wieder bestimmte Muster auf, die eine Modellierung und eine Qualitätssicherung erleichtern:

- »OOA-Muster«, S. 550

Es gibt mehrere Methoden, ein OOA-Modell zu erstellen:

- »OOA-Methode«, S. 559

25.1.1 Strukturierung der OOA-Konzepte

1980 wurde mit der Programmiersprache Smalltalk-80 die Ära der objektorientierten Programmierung (OOP) eröffnet. Seitdem bilden Objekt, Klasse, Attribut, Operation, Botschaft und Vererbung die **Grundkonzepte** der objektorientierten Softwareentwicklung. Es dauerte jedoch weitere zehn Jahre, bis sich die Objektorientierung für die gesamte Softwareentwicklung durchsetzte. Ein Grund lag darin, dass die objektorientierten Grundkonzepte *nicht* ausreichten, um ein Fachkonzept problemnah zu modellieren.

Zur Historie

Erst die Autoren Coad und Yourdon ergänzten 1990 die objektorientierten Grundkonzepte um Konzepte aus dem Entity-Relationship-Modell (siehe »Entity-Relationship-Modell«, S. 199) und der semantischen Datenmodellierung. Sie fügten den Grundkonzepten Assoziationen (Beziehungstypen) und Aggregationen (Ist-Teil-von-Beziehungen) hinzu.

Umfangreiche Modelle können in Pakete gegliedert werden (siehe »Pakete«, S. 145). Der Objektlebenszyklus kann durch Zustandsautomaten spezifiziert werden (siehe »Zustandsautomaten«, S. 269). Andere Autoren integrierten *Use Cases* sowie Sequenz- und Objektdiagramme in die objektorientierte Welt. Die Ursprünge der objektorientierten Analyse sind in der Abb. 25.1-1 dargestellt.

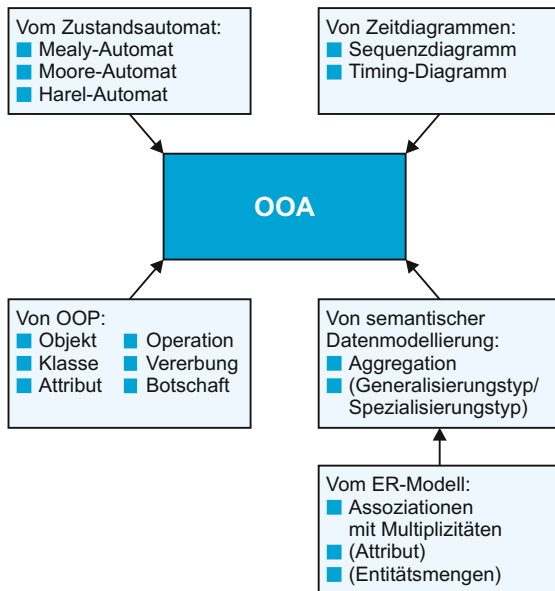


Abb. 25.1-1: Die Ursprünge der objektorientierten Analyse.

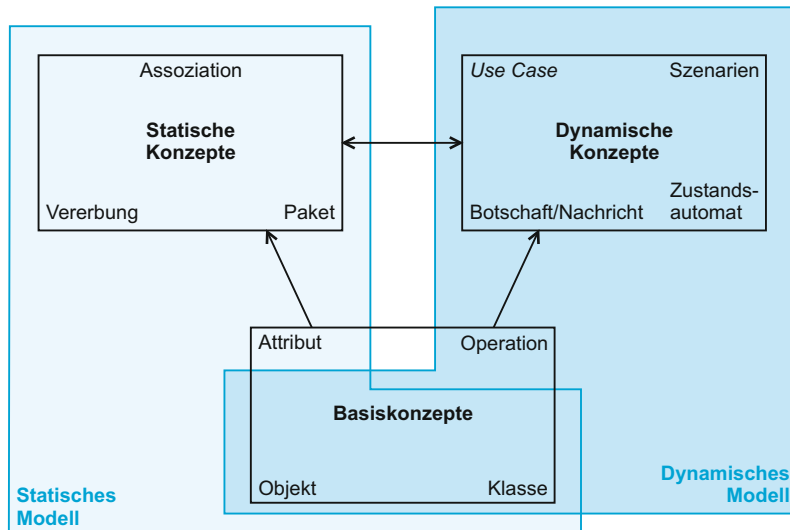
Die objektorientierten Grundkonzepte zusammen mit diesen Erweiterungen bilden die **objektorientierte Analyse** (OOA, *object oriented analysis*), die es dem *Requirements Engineer* erlaubt, die fachliche Lösung des zu realisierenden Systems zu modellieren.

IV 25 Anforderungen modellieren

Nach 1990 erlebte die objektorientierte Softwareentwicklung und insbesondere die objektorientierte Analyse eine stürmische Entwicklung. In mehreren Entwicklungsstufen entstand die **UML** (*unified modeling language*), die sich als Standard-Notation der modellbasierten Objektorientierung durchgesetzt hat.

Ziel Ziel der objektorientierten Analyse ist es, die fachliche Lösung eines neuen Softwareprodukts mit Hilfe objektorientierter Konzepte zu modellieren. Die entstehende fachliche Lösung besteht aus einem statischen und einem dynamischen Modell, ergänzt um logische Elemente (siehe »Constraints und die OCL in der UML«, S. 377). Welches dieser beiden Modelle im *Requirements Engineering* das größere Gewicht besitzt, hängt wesentlich von der jeweiligen Anwendung ab. Das statische Modell steht bei kaufmännisch/administrativen Anwendungen im Vordergrund. Das dynamische Modell ist bei technischen und softwareintensiven Systemen von besonderer Bedeutung. Die Abb. 25.1-2 zeigt, wie die objektorientierten Konzepte auf diese Modelle abgebildet werden.

Abb. 25.1-2:
Statisches und
dynamisches OOA-
Modell.



25.1.2 OOA-Muster

Die Analyse vorhandener OOA-Modelle zeigt, dass sich bestimmte Grundmuster in ähnlicher Form immer wiederholen.

Muster Vereinfacht ausgedrückt ist ein **Muster** (*pattern*) eine Idee, die sich in der Praxis bewährt hat. Ein **OOA-Muster** besteht aus einer Gruppe von Klassen mit feststehenden Verantwortlichkeiten und

25.1 Beispiel: Objektorientierte Analyse IV

Interaktionen [Coa95]. Es kann eine Gruppe von Klassen sein, die durch Beziehungen verknüpft ist, oder eine Gruppe von kommunizierenden Objekten.

Muster ermöglichen Softwareentwicklern eine effektive Kommunikation. Sie erlauben außerdem eine standardisierte Lösung bestimmter Probleme. Bei der Modellierung helfen sie z. B., die verschiedenen Arten der Assoziation zu unterscheiden.

Der Einsatz vorhandener Muster in neuen Softwareentwicklungen hängt stark vom Anwendungsbereich ab. Man unterscheidet allgemeine Muster und anwendungsspezifische Muster. Letztere bieten beispielsweise Problemlösungen für Planungssysteme oder Warenwirtschaftssysteme [Fowl97b]. Im Folgenden werden zehn allgemeine Muster vorgestellt [Balz05, S. 100 ff.].

Jedes Muster wird über einen eindeutigen Namen identifiziert. Es wird durch ein oder mehrere Beispiele erläutert, die skizzieren, für welche Problemstellung das Muster eine Lösung anbietet. Anschließend werden die typischen Eigenschaften dieses Musters aufgeführt.

Beschreibung von Mustern

Muster 1: Liste

In einer Lagerverwaltung besteht ein Lagermodul aus verschiedenen Lagerplätzen. Die Abb. 25.1-3 zeigt eine Lagerliste und die zugehörige OOA-Modellierung. Eine solche Problemstellung kommt in vielen Anwendungsbereichen vor und wird als Komposition modelliert.

Beispiel

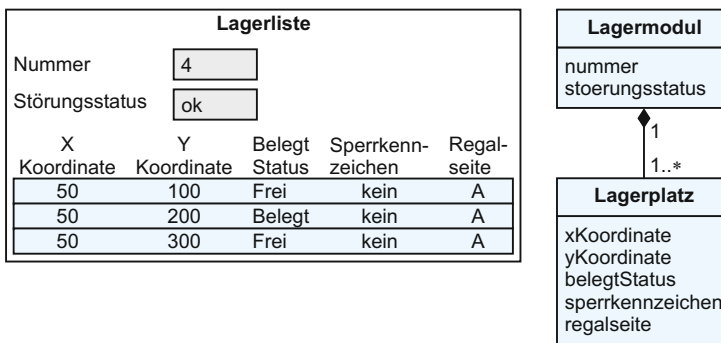


Abb. 25.1-3:
Beispiel für das
Muster Liste.

- Es liegt eine Komposition vor (siehe »Box: Komposition und Aggregation – Methode und Checkliste«, S. 175).
- Ein Ganzes besteht aus gleichartigen Teilen, d. h., es gibt nur eine Teil-Klasse.
- Teil-Objekte bleiben einem Aggregat-Objekt fest zugeordnet. Sie können jedoch gelöscht werden, bevor das Ganze gelöscht wird.
- Die Attributwerte des Aggregat-Objekts gelten auch für die zugehörigen Teil-Objekte.

Eigenschaften

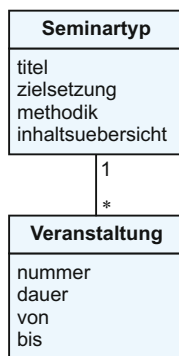
IV 25 Anforderungen modellieren

- Das Aggregat-Objekt enthält mindestens ein Teil-Objekt, d. h., die Multiplizität ist meist 1..*.

Muster 2: Exemplartyp

Beispiel: Von einem Seminartyp sind mehrere Veranstaltungen zu verwalten. SemOrg Würde diese Problemstellung durch eine einzige Klasse Veranstaltung modelliert, dann würden mehrere Objekte bei Titel, Zielsetzung, Methodik, Inhalt usw. identische Attributwerte besitzen. Eine bessere Modellierung ergibt sich, wenn die gemeinsamen Attributwerte mehrerer Veranstaltungen in einer neuen Klasse Seminartyp zusammengefasst werden (Abb. 25.1-4).

Abb. 25.1-4:
Beispiel für das
Muster
Exemplartyp.



- Eigenschaften
- Es liegt eine einfache Assoziation vor, denn es besteht keine Ist-Teil-von-Beziehung (siehe »Box: Assoziationen – Methode und Checkliste«, S. 166).
 - Einmal erstellte Objektverbindungen werden im Allgemeinen nicht verändert. Sie werden nur gelöscht, wenn das betreffende Exemplar entfernt wird.
 - Der Name der neuen Klasse enthält oft Begriffe wie Typ, Gruppe, Beschreibung, Spezifikation.
 - Eine Beschreibung kann – zeitweise – unabhängig von konkreten Exemplaren existieren. Daher ist die Multiplizität im Allgemeinen *many* (0..*).
 - Würde auf die neue Klasse verzichtet, so würde als Nachteil lediglich die redundante »Speicherung« von Attributwerten auftreten.

Muster 3: Baugruppe

Beispiel In der Abb. 25.1-5 soll ausgedrückt werden, dass jedes Auto exakt einen Motor und vier Räder haben soll. Da es sich hier um physische Objekte handelt, d. h. Objekte, die Gegenstände repräsentieren, liegt ein physisches Enthaltensein vor, das mit Hilfe der Komposition mo-

25.1 Beispiel: Objektorientierte Analyse IV

delliert wird. Wenn ein Auto verkauft wird, dann gehören Motor und Räder dazu. Die Zuordnung der Teile zu ihrem Ganzen bleibt normalerweise über einen längeren Zeitraum bestehen. Der Motor kann jedoch durch einen neuen Motor ersetzt werden und der alte Motor in ein anderes Objekt eingebaut werden.

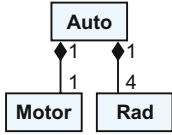


Abb. 25.1-5:
Beispiel für das
Muster Baugruppe.

- Es handelt sich um physische Objekte.
- Es liegt eine Komposition vor (siehe »Box: Komposition und Aggregation – Methode und Checkliste«, S. 175).
- Objektverbindungen bestehen meist über eine längere Zeit. Ein Teil-Objekt kann jedoch von seinem Aggregat-Objekt getrennt werden und einem anderen Ganzen zugeordnet werden.
- Ein Ganzes kann aus unterschiedlichen Teilen bestehen.

Eigenschaften

Muster 4: Stückliste

Eine Grafik besteht aus Grafikelementen, z. B. Rechteck, Text, Linie. Sie kann andere Grafiken enthalten (Abb. 25.1-6). Sowohl die Grafik als auch alle darin enthaltenen Elemente sollen als Einheit behandelt werden können. Jedes der Elemente soll auch einzeln gehandhabt werden können. Wird beispielsweise die Grafik kopiert, dann sollen alle darin enthaltenen Grafikelemente kopiert werden. Wird die Grafik gelöscht, dann werden auch alle seine Teile gelöscht. Ein Grafikelement kann vorher jedoch einer anderen Grafik zugeordnet werden.

Beispiel

Ein Sonderfall liegt vor, wenn sich diese Enthaltensein-Beziehung auf gleichartige Objekte bezieht. Beispielsweise setzt sich jede Komponente aus mehreren Komponenten zusammen. Ein Buchabschnitt enthält andere Buchabschnitte. Umgekehrt ist jede Komponente in einer oder keiner anderen Komponente enthalten.

Diese Problemstellung wird durch eine Komposition modelliert, wobei die verschiedenen Teil-Objekte durch eine Vererbung dargestellt werden. Bei der Klasse Grafik liegt eine 0..1-Multiplizität vor. Eine 1-Multiplizität würde bedeuten, dass jedes Grafikelement – also auch jede Grafik – in einer anderen Grafik enthalten sein müsste.

- Es liegt eine Komposition vor (siehe »Box: Komposition und Aggregation – Methode und Checkliste«, S. 175).
- Das Aggregat-Objekt und seine Teil-Objekte müssen sowohl als Einheit als auch einzeln behandelt werden können.

Eigenschaften

IV 25 Anforderungen modellieren

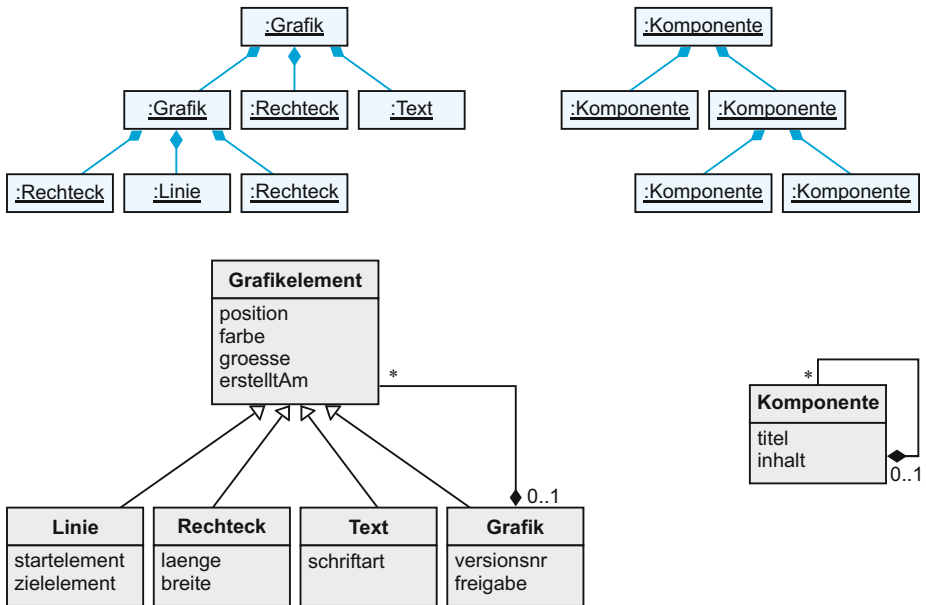


Abb. 25.1-6:
Beispiel für das
Muster Stückliste.

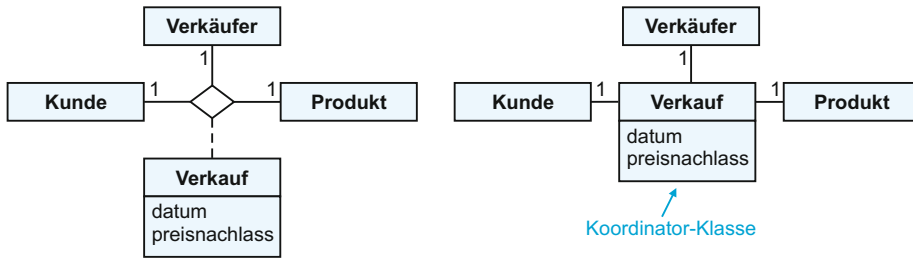
- Teil-Objekte können anderen Aggregat-Objekten zugeordnet werden.
- Die Multiplizität bei der Aggregat-Klasse ist 0..1.
- Ein Objekt der Art A kann sich aus mehreren Objekten der Arten A, B und C zusammensetzen.
- Der Sonderfall der Stückliste ist, dass ein Stück nicht aus Objekten unterschiedlicher Art, sondern nur aus einer einzigen Art besteht.

Muster 5: Koordinator

Beispiel In der Abb. 25.1-7 verbindet eine ternäre Assoziation Objekte der Klassen Kunde, Verkäufer und Produkt und »merkt« sich in der assoziativen Klasse Verkauf, welcher Verkäufer welchem Kunden welches Produkt verkauft hat. Diese ternäre Assoziation kann – wie abgebildet – in binäre Assoziationen und eine Koordinator-Klasse transformiert werden. Für eine Koordinator-Klasse ist typisch, dass sie selbst oft nur wenige Attribute und Operationen besitzt. Sie merkt sich vor allem, »wer wen kennt«. Als Sonderfall dieser Problemstellung kann eine binäre Assoziation mit einer assoziativen Klasse betrachtet werden.

- Eigenschaften
- Es liegen einfache Assoziationen vor (siehe »Box: Assoziationen – Methode und Checkliste«, S. 166).
 - Die Koordinator-Klasse ersetzt eine n-äre ($n \geq 2$) Assoziation durch binäre Assoziationen mit assoziativer Klasse.

25.1 Beispiel: Objektorientierte Analyse IV



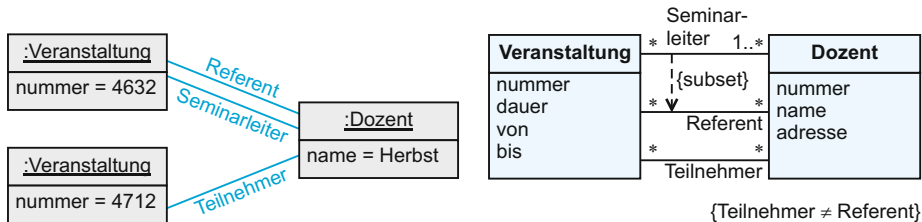
- Die Koordinator-Klasse besitzt kaum Attribute/Operationen, sondern mehrere Assoziationen zu anderen Klassen, im Allgemeinen zu genau einem Objekt jeder Klasse.

Abb. 25.1-7:
Beispiel für das
Muster
Koordinator.

Muster 6: Rollen

Ein Dozent kann auch Teilnehmer an einer Veranstaltung sein, um sich selbst weiterzubilden. Dabei kann der Dozent sowohl Referent, als auch Seminarleiter als auch Teilnehmer von Veranstaltungen sein. Der Dozent spielt – zur selben Zeit – in Bezug auf die Klasse Veranstaltung mehrere Rollen. Diese Problemstellung kommt relativ häufig vor und wird wie in der Abb. 25.1-8 modelliert. Würden anstelle der Klasse Dozent die Klassen Referent, Seminarleiter und Teilnehmer verwendet, dann hätten alle drei Klassen dieselben Attribute (und Operationen). Außerdem könnte nicht modelliert werden, dass ein bestimmtes Dozenten-Objekt sowohl Referent als auch Teilnehmer – bei anderen Veranstaltungen – ist.

Beispiel:
SemOrg



- Zwischen zwei Klassen existieren zwei oder mehrere einfache Assoziationen (siehe »Box: Assoziationen – Methode und Checkliste«, S. 166).
- Ein Objekt kann – zu einem Zeitpunkt – in Bezug auf die Objekte der anderen Klasse verschiedene Rollen spielen.
- Objekte, die verschiedene Rollen spielen können, besitzen unabhängig von der jeweiligen Rolle die gleichen Eigenschaften und ggf. gleiche Operationen.

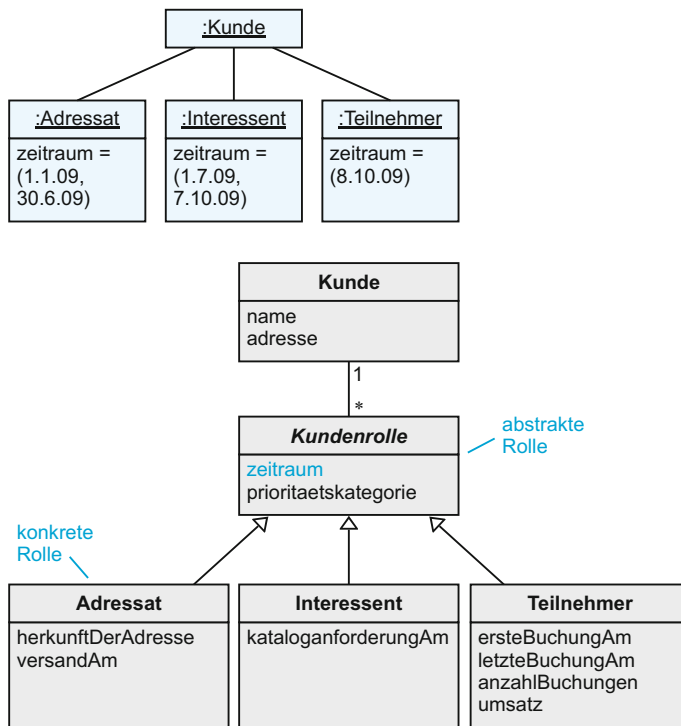
Abb. 25.1-8:
Beispiel für das
Muster Rollen.

IV 25 Anforderungen modellieren

Muster 7: Wechselnde Rollen

Beispiel: In der Abb. 25.1-9 wird modelliert, dass ein Kunde wechselnde Rollen in der Seminarorganisation einnehmen kann. Soll später nachvollzogen werden, welche Werte ein Objekt zu welchem Zeitpunkt besessen hat, dann ist eine Aufzeichnung der Historie notwendig. In der Klasse Kunde werden alle Daten gespeichert, die unabhängig von einem Zeitpunkt sind. Ein Kunde ist in der Regel zunächst ein Adressat von Werbematerial. Die Adresse wird bei einem Adressenhandel gekauft. Fordert ein Kunde einen Seminarkatalog an, dann wird er zum Interessenten. Bucht er eine Veranstaltung, dann ist er ein Teilnehmer. Um das Marketing zu optimieren, sollen alle drei Rollen gespeichert werden. Für alle Kunden wird zu bestimmten Zeitpunkten eine Prioritätskategorie ermittelt, die die Bedeutung der Kunden widerspiegelt.

Abb. 25.1-9:
Beispiel für das
Muster Wechselnde
Rollen.



- Eigenschaften
- Ein Objekt der realen Welt kann zu verschiedenen Zeiten verschiedene Rollen spielen. In jeder Rolle kann es unterschiedliche Eigenschaften (Attribute, Assoziationen) und Operationen besitzen.
 - Die unterschiedlichen Rollen werden mittels Vererbung modelliert (siehe »Box: Vererbung – Methode und Checkliste«, S. 155).

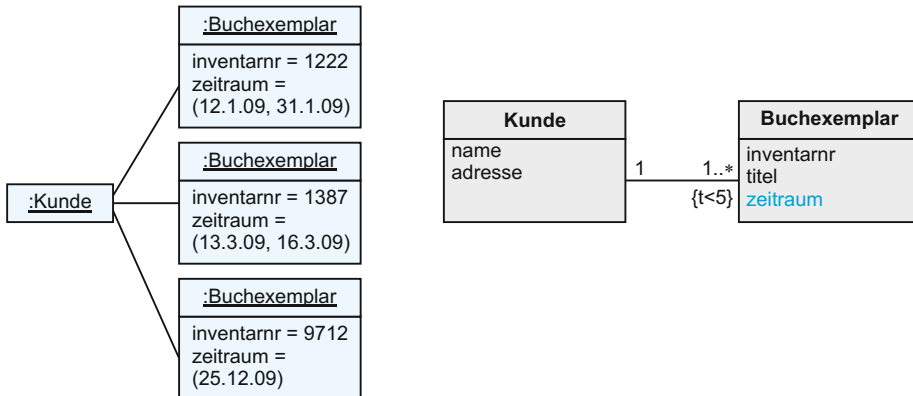
25.1 Beispiel: Objektorientierte Analyse IV

- Objektverbindungen zwischen dem Objekt und seinen Rollen werden nur erweitert, d. h. weder gelöscht noch zu anderen Objekten aufgebaut.

Muster 8: Historie

Eine Bibliothek muss speichern, welche Bücher welcher Kunde ausgeliehen hat. Dabei darf zu jedem Zeitpunkt t der Kunde nur fünf Buchexemplare ausgeliehen haben. Diese Problemstellung wird – wie in Abb. 25.1-10 gezeigt – mit einer Assoziation modelliert. Für jedes Buchexemplar wird der Zeitraum eingetragen. Die Restriktion $\{t \leq 5\}$ spezifiziert, dass ein Kunde zu einem Zeitpunkt maximal fünf Buchexemplare ausgeliehen hat. Wenn alle einmal ausgeliehenen Buchexemplare eines Kunden gespeichert sein sollen, dann bedeutet dies, dass die aufgebauten Verbindungen zu Buchexemplar bestehen bleiben, bis der Kunde nicht mehr Kunde der Bibliothek ist.

Beispiel



- Es liegt eine einfache Assoziation vor (siehe »Box: Assoziationen – Methode und Checkliste«, S. 166).
- Für ein Objekt sind mehrere Vorgänge bzw. Fakten zu dokumentieren.
- Für jeden Vorgang bzw. jedes Faktum ist der Zeitraum festzuhalten.
- Aufgebaute Objektverbindungen zu den Vorgängen bzw. Fakten werden nur erweitert.
- Die zeitliche Restriktion $\{t \# m\}$ (m = gültige Multiplizität) sagt aus, was zu einem Zeitpunkt gelten muss, wobei $\#$ für die Vergleichsoperationen $=$, $<$, $>$, \leq , \geq und \neq steht.

Abb. 25.1-10:
Beispiel für das
Muster Historie.

IV 25 Anforderungen modellieren

Muster 9: Gruppe

Beispiel In der Abb. 25.1-11 bildet sich eine Gruppe, wenn mehrere Mitarbeiter zu einem Projektteam gehören. Da das Projektteam auch ohne zugehörige Mitarbeiter existieren soll, wird die *many*-Multiplizität gewählt. Soll modelliert werden, dass beim Anlegen des Projektteams mindestens ein Mitarbeiter zugeordnet wird, dann wäre die Multiplizität 1..* zu wählen. Wenn ein Mitarbeiter aus einem Projektteam ausscheidet, dann wird die entsprechende Objektverbindung getrennt.

Abb. 25.1-11:
Beispiel für das
Muster Gruppe.

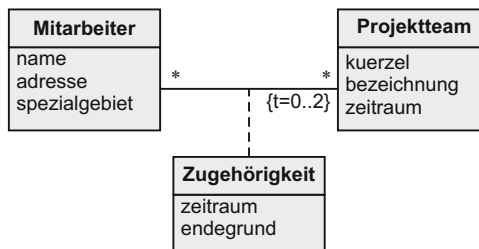


- Eigenschaften
- Es liegt eine einfache Assoziation vor (siehe »Box: Assoziationen – Methode und Checkliste«, S. 166).
 - Mehrere Einzel-Objekte gehören – zu einem Zeitpunkt – zum selben Gruppen-Objekt.
 - Es ist jeweils zu prüfen, ob die Gruppe – zeitweise – ohne Einzel-Objekte existieren kann oder ob sie immer eine Mindestanzahl von Einzel-Objekten enthalten muss.
 - Objektverbindungen können auf- und abgebaut werden.

Muster 10: Gruppenhistorie

Beispiel Soll die Zugehörigkeit zu einer Gruppe nicht nur zu einem Zeitpunkt, sondern über einen Zeitraum festgehalten werden, dann ist die Problemstellung wie in der Abb. 25.1-12 zu modellieren. Für jeden Mitarbeiter wird festgehalten, über welchen Zeitraum er zu einem Projektteam gehört hat. Die Restriktion {t=0..2} sagt aus, dass er zu einem Zeitpunkt in maximal zwei Projektteams tätig sein kann. Wenn ein Mitarbeiter ein Projektteam verlässt, dann wird dies durch die Attributwerte im entsprechenden Objekt von Zugehörigkeit beschrieben.

Abb. 25.1-12:
Beispiel für das
Muster
Gruppenhistorie.



25.1 Beispiel: Objektorientierte Analyse IV

- Ein Einzel-Objekt gehört – im Laufe der Zeit – zu unterschiedlichen Gruppen-Objekten. Eigenschaften
- Die Historie wird mittels einer assoziativen Klasse modelliert. Dadurch ist die Zuordnung zwischen Einzel-Objekten und Gruppen deutlich sichtbar (siehe »Assoziation«, S. 158).
- Die zeitliche Restriktion $\{t=m\}$ (m = gültige Multiplizität) sagt aus, was zu einem Zeitpunkt gelten muss.
- Da Informationen über einen Zeitraum festzuhalten sind, bleiben erstellte Objektverbindungen bestehen und es werden nur Verbindungen hinzugefügt.

25.1.3 OOA-Methode

Bei den objektorientierten Methoden lassen sich prinzipiell drei Vorgehensweisen unterscheiden:

- 1 Orientierung an den Informationen (Daten) des Systems (statisches Modell),
- 2 Orientierung an der Funktionalität (Verhalten) des Systems (dynamisches Modell),
- 3 Synthese von 1 und 2.

Die Praxis hat gezeigt, dass für eine erfolgreiche Modellierung das Zusammenwirken von statischem und dynamischem Modell unabdingbar ist, wobei der Ausgangspunkt der Modellierung und der jeweilige Schwerpunkt von dem zu modellierenden Anwendungsbeereich abhängt. Die UML selbst enthält *keine* Angaben zur methodischen Vorgehensweise.

Im Folgenden wird die Methode von [Balz05, S. 129 ff.] dargestellt, die anhand einer schrittweisen Vorgehensweise und durch Checklisten gestützt ein systematisches Erstellen eines OOA-Modells ermöglicht. Die methodische Vorgehensweise besteht dabei aus

- dem Makroprozess, der die methodischen Schritte festlegt und
- methodischen Regeln, die in Form von Checklisten zur Verfügung stehen.

Voraussetzung für die Erstellung eines OOA-Modells ist eine vorliegende Anforderungsspezifikation. Voraussetzung

Der Makroprozess beschreibt auf einem hohen Abstraktionsniveau, in welcher Reihenfolge die einzelnen Aufgaben zur Erstellung eines OOA-Modells auszuführen sind:

- Ermitteln der relevanten *Use Cases* (oft in der Anforderungsspezifikation bereits erfolgt). Makroprozess
- Ableitung von Klassen aus den *Use Cases*.
- Erstellen des statischen Modells.
- Parallel dazu Erstellen des dynamischen Modells.
- Berücksichtigung der Wechselwirkung beider Modelle.

IV 25 Anforderungen modellieren

Der Makroprozess berücksichtigt die Gleichgewichtigkeit (*balancing*) von statischem und dynamischem Modell (balancierter Makroprozess). Die Konzentration auf das statische Modell vor dem dynamischen sorgt für eine größere Stabilität des Modells und schafft durch die Bildung von Klassen eine wesentliche Abstraktionsebene. Wichtig ist, dass nach dem Erstellen des ersten statischen Modellkerns das dynamische und das statische Modell parallel weiter entwickelt werden, um deren Wechselwirkungen adäquat berücksichtigen zu können (Abb. 25.1-13).

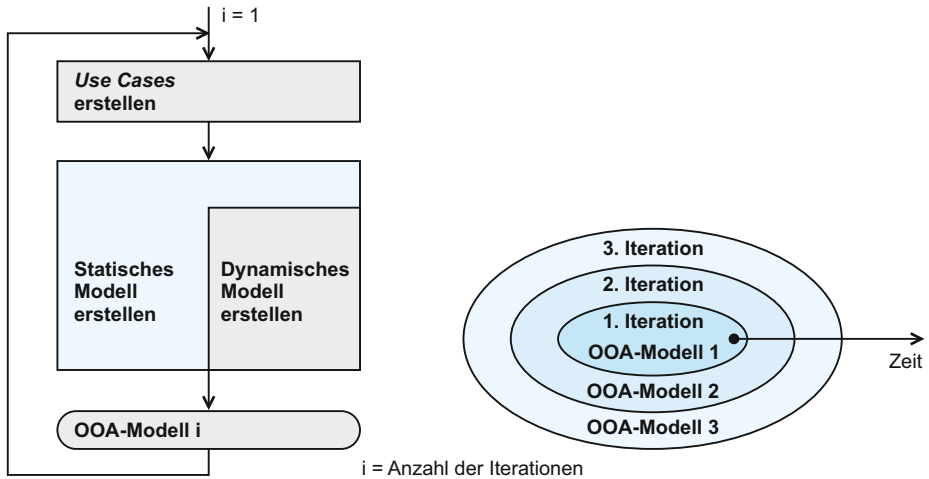


Abb. 25.1-13: Die hier beschriebene Methode realisiert einen evolutionären, iterativen Entwicklungsprozess. Zunächst wird eine objektorientierte Analyse für den Produktkern erstellt, der anschließend zu entwerfen und implementieren ist. Dieser Kern wird in weiteren Zyklen erweitert, bis ein auslieferbares System entsteht. Dabei wird die Arbeit früherer Zyklen *nicht* verworfen, sondern korrigiert und verbessert. Ein evolutionärer Prozess ist immer iterativ, weil er eine ständige Verfeinerung des Systems erfordert. Alle Erfahrungen und Ergebnisse eines Iterationsschrittes fließen in den nächsten Schritt ein. Für die Qualitätssicherung jedes Iterationsergebnisses hat sich die formale Inspektion bewährt.

Makroprozess Der Makroprozess umfasst folgende Aufgabenbereiche:

- Analyse im Großen,
- 7 Schritte zum statischen Modell und
- 3 Schritte zum dynamischen Modell.

Bei der **Analyse im Großen** handelt es sich um Aufgaben, die *nicht* spezifisch für eine objektorientierte Entwicklung sind, während die statische und dynamische Modellierung einen objektorientierten Charakter besitzen.

25.1 Beispiel: Objektorientierte Analyse IV

Die in den Boxen »Analyse im Großen«, »7 Schritte zum statischen Modell« und »3 Schritte zum dynamischen Modell« angegebenen Schritte können nicht immer sequenziell durchlaufen werden. Beispielsweise lassen sich oft gleichzeitig mit dem Identifizieren der Klassen auch Assoziationen finden.

1 Use Cases aufstellen

Erstellen der wesentlichen *Use Cases*.

→ *Beschreibung *Use Cases* (siehe »Box: Use Case – Methode und Checkliste«, S. 262)

→ *Use Case*-Diagramm (siehe »Geschäftsprozesse und *Use Cases*«, S. 250)

2 Pakete bilden

Bilden von Teilsystemen, d. h. zusammenfassen von Modellelementen zu Paketen. Bei großen Systemen, die im Allgemeinen durch mehrere Teams bearbeitet werden, muss die Bildung von Paketen am Anfang stehen.

→ Paketdiagramm (siehe »Pakete«, S. 145)

*Hinweis: Für jeden Schritt wird angegeben, welche Diagramme bzw. Modelle zu erstellen sind (mit »→« gekennzeichnet).

Box: Analyse im Großen

Alternative Methoden

Mögliche Alternativen zum beschriebenen balancierten Makroprozess sind der szenarien-basierte und der daten-basierte Makroprozess [IBM97].

Der **szenarien-basierte Makroprozess** ist empfehlenswert, wenn umfangreiche funktionale Anforderungen vorliegen und alte Datenbestände nicht existieren. Er besteht aus den Schritten:

- 1 *Use Cases* formulieren,
- 2 Szenarien aus den *Use Cases* ableiten,
- 3 Sequenzdiagramme aus den Szenarien ableiten,
- 4 Klassendiagramm erstellen und
- 5 Zustandsdiagramme erstellen.

szenarien-basiert

Der **daten-basierte Makroprozess** empfiehlt sich, wenn ein umfangreiches Datenmodell oder alte Datenbestände existieren und der Umfang der funktionalen Anforderungen zunächst unbekannt ist. Er ist auch dann zu wählen, wenn es sich um ein Auskunftssystem handelt, das später mehr oder weniger flexibel gestaltete Anfragen handhaben muss. Er umfasst die Schritte:

- 1 Klassendiagramm erstellen,
- 2 *Use Cases* formulieren,
- 3 Szenarien aus den *Use Cases* ableiten,
- 4 Sequenzdiagramme aus den Szenarien und dem Klassendiagramm ableiten und
- 5 Zustandsdiagramme erstellen.

daten-basiert

IV 25 Anforderungen modellieren

Box: 7 Schritte zum statischen Modell

1 Klassen identifizieren*

Für jede Klasse nur so viele Attribute und Operationen identifizieren, wie für das Problemverständnis und das einwandfreie Erkennen der Klasse notwendig sind.

→ Klassendiagramm (siehe »Funktions-Strukturen«, S. 142)

→ Kurzbeschreibung Klassen (siehe »Funktionalität«, S. 127)

2 Assoziationen identifizieren

Zunächst nur die reinen Verbindungen eintragen, d. h. noch keine genaueren Angaben, z. B. zur Multiplizität oder zur Art der Assoziation, machen.

→ Klassendiagramm (siehe »Pakete«, S. 145)

3 Attribute identifizieren

Identifizieren Sie alle Attribute des Fachkonzepts.

→ Klassendiagramm (siehe »Funktions-Strukturen«, S. 142)

4 Vererbungsstrukturen identifizieren

Aufgrund der identifizierten Attribute Vererbungsstrukturen erstellen.

→ Klassendiagramm (siehe »Funktions-Strukturen«, S. 142)

5 Assoziationen vervollständigen

Endgültig festlegen, ob eine »normale« Assoziation, Aggregation oder Komposition vorliegt sowie Festlegung der Multiplizitäten, Rollen, Namen und Restriktionen.

→ Klassendiagramm (siehe »Funktions-Strukturen«, S. 142)

→ Objektdiagramm (siehe »Funktionalität«, S. 127)

6 Attribute spezifizieren

Für alle identifizierten Attribute eine vollständige Spezifikation erstellen.

→ Attributspezifikation (siehe »Funktionalität«, S. 127)

7 Muster identifizieren

Das Klassendiagramm daraufhin überprüfen, ob Muster enthalten sind und ob diese richtig modelliert wurden (siehe »OOA-Muster«, S. 550).

*Hinweis: Für jeden Schritt wird angegeben, welche Diagramme bzw. Modelle zu erstellen sind (mit »→« gekennzeichnet).

Checklisten

Bei der praktischen Anwendung zeigt sich, dass eine grob beschriebene Vorgehensweise – wie der obige Makroprozess – nicht ausreicht. Auf der anderen Seite ist ein detailliertes Vorgehensmodell oft nur für bestimmte Anwendungsdomänen geeignet und kann *nicht* problemlos auf andere Bereiche übertragen werden. Der erfahrene *Requirements Engineer* wendet – meist mehr oder minder intuitiv – Hunderte von Regeln an, die er situationsspezifisch einsetzt. Für die Anwendung der Regeln gibt es *keine* festgelegte Reihenfolge. Derartige Regeln stehen hier in Form von Boxen mit Checklisten zur Verfügung.

Außerdem greift ein erfahrener *Requirements Engineer* in vielen Fällen auf bereits gelöste ähnliche Problemstellungen zurück (Muster). Durch die Verwendung von OOA-Mustern wird der Erstellungsaufwand reduziert und es wird eine höhere Standardisierung erreicht, was es anderen Lesern erleichtert, sich in das Modell einzuarbeiten (siehe »OOA-Muster«, S. 550).

1 Szenarien erstellen*

Jeden *Use Case* durch eine Menge von Szenarien präzisieren.

→ Sequenzdiagramm, Kommunikationsdiagramm (siehe »Sequenzdiagramm«, S. 333, »Kommunikationsdiagramm«, S. 343)

Alternativ oder ergänzend können auch Aktivitätsdiagramme verwendet werden (siehe »Kontrollstrukturen«, S. 227).

2 Zustandsautomat erstellen

Für jede Klasse prüfen, ob ein nicht-trivialer Lebenszyklus erstellt werden kann oder muss.

→ Zustandsdiagramm (siehe »Box: Zustandsautomat – Methode und Checkliste«, S. 295)

3 Operationen beschreiben

Überlegen, ob eine Beschreibung notwendig ist. Wenn ja, dann ist je nach Komplexitätsgrad die entsprechende Form zu wählen.

→ Klassendiagramm (siehe »Funktionalität«, S. 127)

→ fachliche Beschreibung der Operationen, Zustandsautomaten (siehe »Box: Zustandsautomat – Methode und Checkliste«, S. 295), Aktivitätsdiagramme (siehe »Kontrollstrukturen«, S. 227)

*Hinweis: Für jeden Schritt wird angegeben, welche Diagramme bzw. Modelle zu erstellen sind (mit »→« gekennzeichnet).

**Box: 3 Schritte
zum
dynamischen
Modell**

25.2 Domänenspezifische Sprachen

In der Programmiersprachenwelt unterscheidet man schon lange zwischen universell einsetzbaren Sprachen (*general-purpose languages*) – z. B. Java – und speziellen, auf einen Anwendungsbereich fokussierte Sprachen (*special-purpose languages*) – z. B. Sprachen zur Robotersteuerung. Eine ähnliche Entwicklung gibt es inzwischen bei Modellierungssprachen. Die UML ist eine universell einsetzbare Modellierungssprache. Als Alternative zur UML entstehen domänenspezifische Sprachen (*domain-specific languages*) – abgekürzt DSL. Werden DSLs für die Modellierung konzipiert, dann spricht man auch von DSM (*domain-specific modeling*).

Überlegen Sie, welche Vor- und Nachteile DSLs haben.

Gegenüber der UML hat eine DSL folgende Vorteile:

- + Die DSL orientiert sich an den Abstraktionen und der Semantik der jeweiligen Domäne, sodass Modellierer direkt mit den Domänenkonzepten arbeiten können.
- + Richtlinien und Regeln der Domäne können als Einschränkungen (*constraints*) in die DSL eingebunden werden, sodass es nicht möglich ist, ungültige oder unerwünschte Modelle zu modellieren.
- + Eine DSL ist schlank, d. h. der Umfang und die Komplexität sind beschränkt, sodass auch der Lernaufwand sinkt.
- + Es muss keine neue Semantik gelernt werden, da die Konzepte der Domäne in der Regel gut bekannt und bereits angewendet werden.

Frage
Antwort

IV 25 Anforderungen modellieren

- + Modelle, die in einer DSL modelliert sind, sind i. Allg. abstrakter, kompakter, besser lesbar, leichter verständlich und verifizierbar. Sie erlauben außerdem eine leichtere Kommunikation mit den *Stakeholdern*.

Dem stehen folgende Nachteile gegenüber:

- Die Anfangsinvestitionen sind hoch. Es muss eine DSL für die jeweilige Domäne entwickelt werden. Werkzeuge und Generatoren müssen bereitgestellt und gepflegt werden.
- Es muss genügend Projekte in der Anwendungsdomäne geben, damit sich die Entwicklung der DSL lohnt – in der Regel ist dies bei Produktlinien der Fall.
- Es ist schwierig, die richtigen Abstraktionen zu identifizieren.

Ziel Ziel ist es, im Rahmen einer modellgetriebenen Entwicklung (siehe »Modellgetriebene Entwicklung«, S. 79) aus den Modellen, die mit einer DSL modelliert wurden, weitgehend vollautomatisch mit Hilfe von Generatoren die fertige Anwendung zu generieren ([ToKe04], [KoHä05]).

Beispiel In der Literatur werden folgende Beispiele für DSLs aufgeführt:

- Modellierung von Konferenzverwaltungssystemen [KoHä05, S. 19].
- Mobile Telefonanwendungen [ToKe04, S. 32].
- Infotainment-Systeme für Autos [Tolv08, S. 38 f.]
- Point-of-Sale-Systeme für PDAs [Tolv08, S. 39]

Empirie Wird eine DSM zusammen mit Generatoren eingesetzt, dann sind große Produktivitäts- und Qualitätsgewinne zu verzeichnen [Tolv08, S. 39 ff.]:

- Bei Panasonic wurde eine Produktivitätssteigerung von 500 % verglichen mit einer manuellen Codierung erzielt.
- Bei Nokia wurde eine Verbesserung von 1000 % erreicht und die Entwicklungszeit reduzierte sich von 2 Wochen auf einen Tag.
- Bei Lucent wurden mehrere DSLs entwickelt und führten zu einer Produktivitätssteigerung von 300 % bis 1000 %.
- Bei der US Air Force sank die Fehlerquote um 50 %.

Best Practices Der erfolgreiche Einsatz einer DSL hängt von folgenden Faktoren ab:

- Erfahrene Entwickler müssen gute Abstraktionen finden und in die DSL integrieren.
 - Sich auf einen engen Anwendungsbereich konzentrieren.
- Die meisten DSL-Lösungen finden sich heute innerhalb einer einzigen Organisation.

Weiterführende Literatur [ESS08], [Siro08]

26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung

Unter Einsatz der Basiskonzepte (»Basiskonzepte«, S. 99) ergibt sich auf der Grundlage des Pflichtenheftes und des Glossars (siehe »Fallstudie: SemOrg – Die Spezifikation«, S. 107) folgende **fachliche Lösung**, gegliedert in die Bereiche:

- Statik
- Dynamik
- Logik

Visionen und Ziele werden *nicht* nochmals aufgeführt, da sie nicht weiter formalisierbar sind. Ein Konzept für die Benutzungsoberfläche ist noch *nicht* enthalten. Es ist nur die Ausbaustufe 1 (SemOrg V1.0 – Kern) modelliert.

Einschränkungen

SemOrg V1.0 – Fachliche Lösung

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Manfred Mustermann	Geschäftsführer Teachware	in Bearbeitung	11/09	Ausbaustufe 1 (SemOrg V1.0 – Kern), noch kein GUI

Statik

Die Aufteilung auf Pakete zeigt die Abb. 26.0-1.

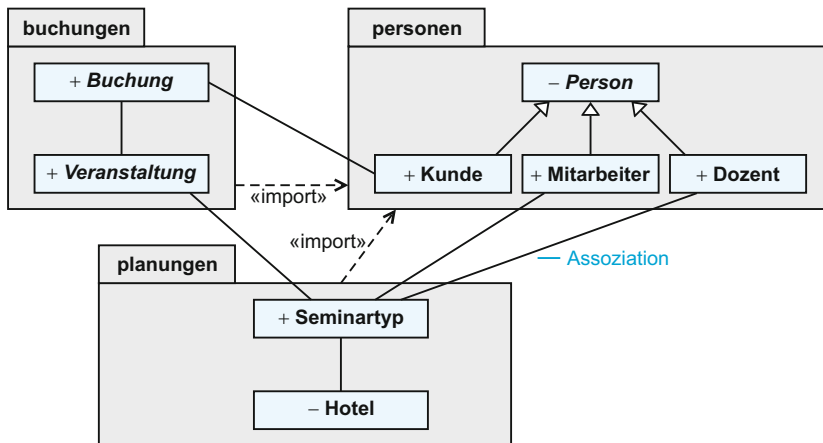


Abb. 26.0-1:
Paketdiagramm
zu SemOrg.

Das detaillierte Klassendiagramm zeigt die Abb. 26.0-2.

IV 26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung

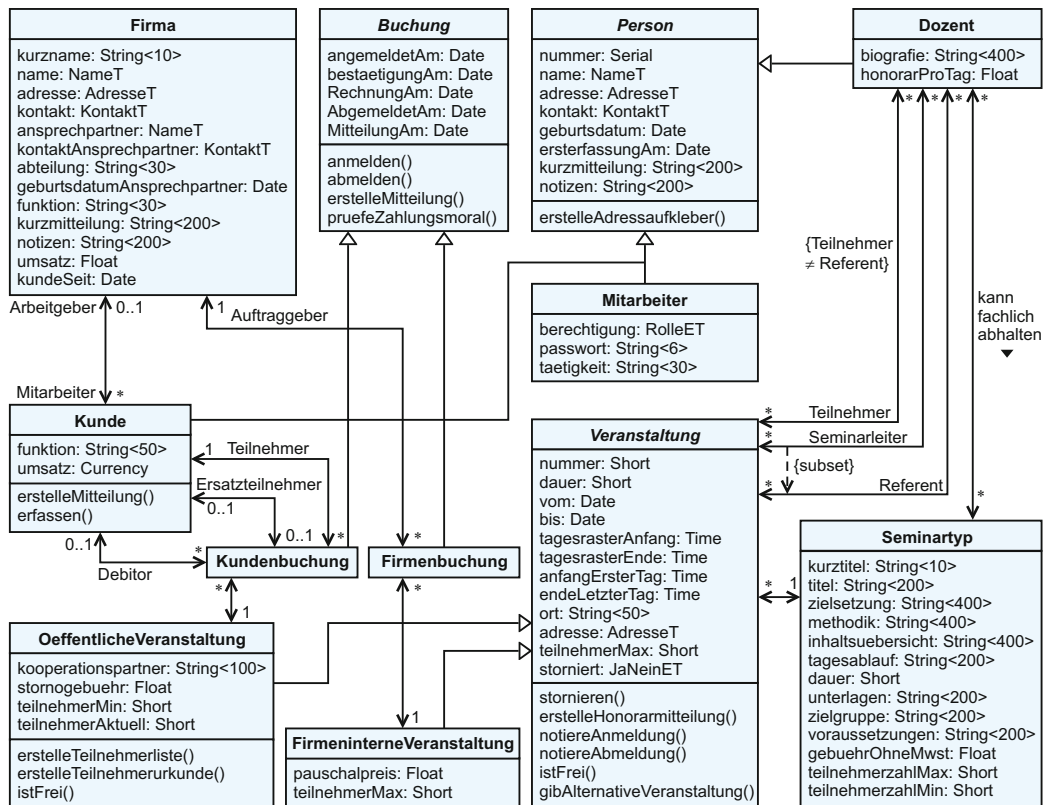
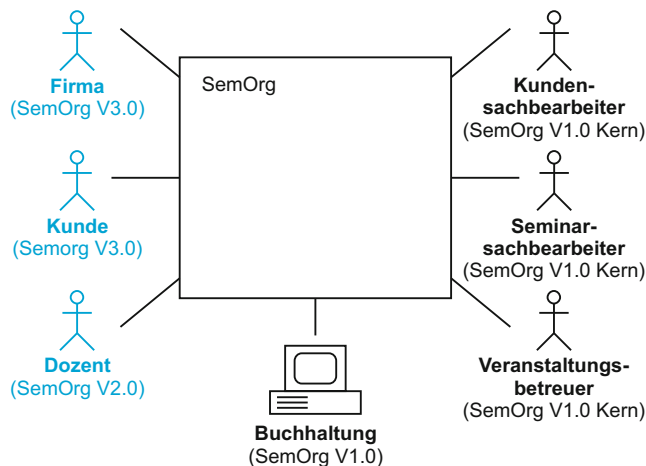


Abb. 26.0-2: Dynamik

Klassendiagramm
des Produkts
SemOrg.

Kontext und **Überblick** werden durch zwei *Use Case*-Diagramme (Abb. 26.0-3, Abb. 26.0-4) beschrieben.

*Abb. 26.0-3:
Umwelt des
Produkts SemOrg
(Umweltdia-
gramm) als UML-
Use-Case-
Diagramm.*



26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung IV

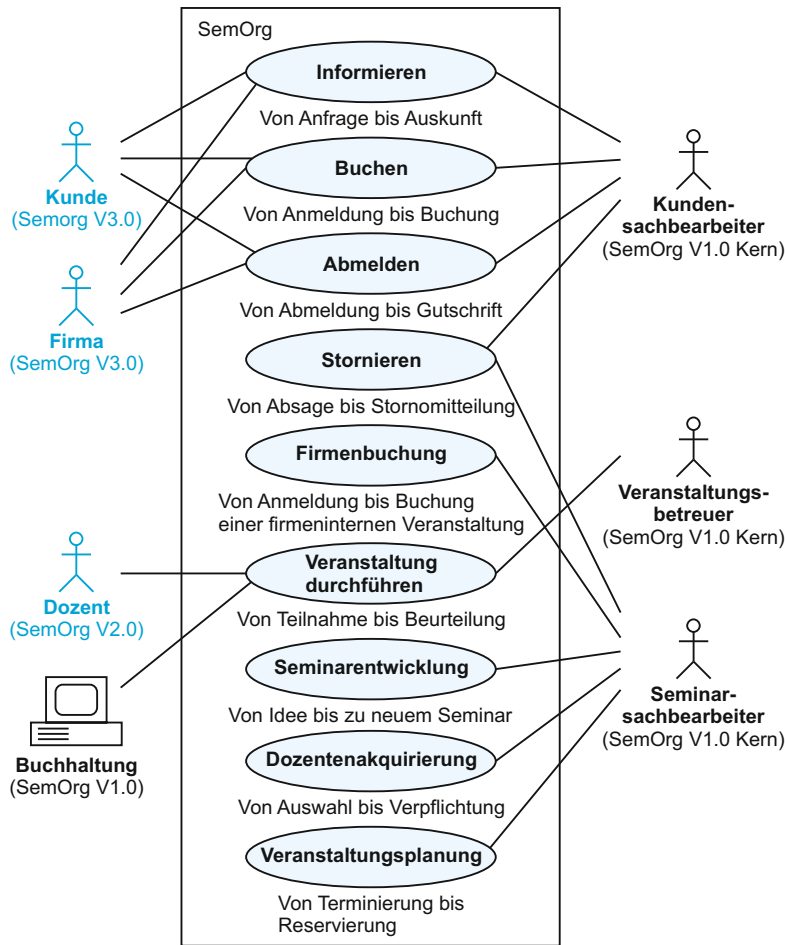


Abb. 26.0-4: UML-Use Cases des Produkts SemOrg (Übersichtsdiagramm).

Aus den Use Cases Buchen und Firmenbuchung wird der Use Case Zahlungsmoral prüfen »herausgezogen« (Abb. 26.0-5).



Abb. 26.0-5: Strukturierung von Use Cases mit der include-Beziehung.

Mit Hilfe einer Use Case-Schablone (»Use Case-Diagramme und -Schablonen«, S. 255) wurden die Use Cases präzisiert:

UC Informieren

Use Case: Informieren: Von Anfrage bis Auskunft

Ziel: Kunde erhält gewünschte Auskunft oder bekommt Infomaterial zugeschickt.

IV 26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung

Kategorie: primär

Vorbedingung: -

Nachbedingung Erfolg: Kunde hat gewünschte Information.

Nachbedingung Fehlschlag: Gewünschte Auskunft konnte nicht erteilt werden.

Akteure: Kundensachbearbeiter

Auslösendes Ereignis: Kunde schreibt (Brief, Fax, E-Mail) oder ruft an

Beschreibung:

1 Kundendaten abrufen

2 Information erteilen

Erweiterung:

1a Kundendaten aktualisieren

2a Adressaufkleber erstellen (für Versand von Info-Material)

Alternativen:

1a Neukunden erfassen

UC Buchen

Use Case: Buchen: Von Anmeldung bis Buchung

Ziel: Anmeldebestätigung und Rechnung an Kunden geschickt

Kategorie: primär

Vorbedingung: -

Nachbedingung Erfolg: Kunde ist angemeldet

Nachbedingung Fehlschlag: Mitteilung an Kunden, dass Veranstaltung ausgebucht oder ausfällt oder nicht existiert oder Kunde bereits angemeldet war. werden.

Akteure: Kundensachbearbeiter

Auslösendes Ereignis: Anmeldung des Kunden liegt vor.

Beschreibung:

1 Kundendaten abrufen

2 Veranstaltung prüfen

3 Buchung vornehmen

4 Anmeldebestätigung und Rechnung erstellen

5 Rechnungskopie an Buchhaltung

Erweiterung:

1a Kundendaten aktualisieren

1b Wenn Kunde Mitarbeiter einer Firma ist, dann Firmendaten anlegen bzw. wenn vorhanden, abrufen und aktualisieren.

1c Zahlungsmoral überprüfen

Alternativen:

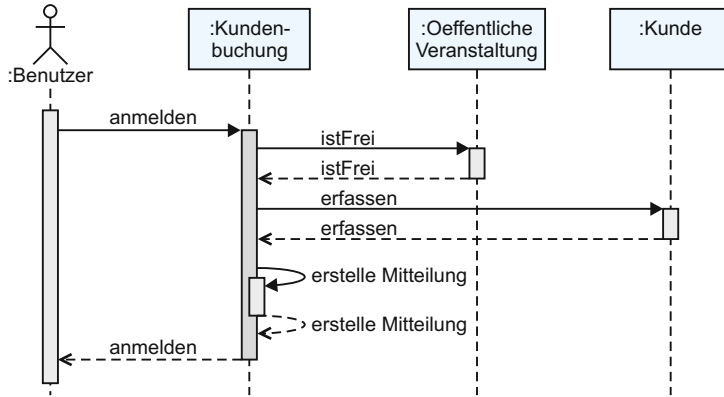
1a Neukunden erfassen

2a Auf alternative Veranstaltungen hinweisen, wenn ausgebucht.

2b Mitteilung »falsche Veranstaltung«, wenn nicht existierende Veranstaltung.

26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung IV

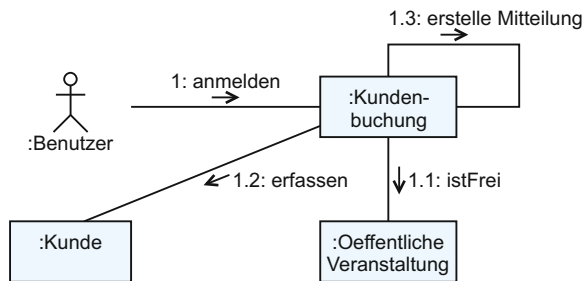
Die Anmeldung eines Neu-Kunden wird durch ein Sequenzdiagramm genauer spezifiziert (Abb. 26.0-6).



Sequenzdiagramm
Anmelden

Abb. 26.0-6:
Sequenzdiagramm
für eine
erfolgreiche
Anmeldung eines
neuen Kunden
zum Seminar.

Die Anmeldung eines Neu-Kunden wird zusätzlich durch ein Kommunikationsdiagramm definiert (Abb. 26.0-7).



Kommunikationsdiagramm
Anmelden

Abb. 26.0-7:
Kommunikationsdiagramm
für eine
erfolgreiche
Anmeldung eines
neuen Kunden
zum Seminar.

Die erlaubten Operationen in Abhängigkeit vom aktuellen Zustand bei einer Seminarveranstaltung zeigt die Abb. 26.0-8.

Zustands-
automat
Veranstaltung

UC Abmelden

Use Case: Abmelden: Von Abmeldung bis Gutschrift

Ziel: Abmeldebestätigung und Gutschrift an Kunden geschickt

Kategorie: primär

Vorbedingung: Kunde ist zu einer Veranstaltung angemeldet

Nachbedingung Erfolg: Kunde ist abgemeldet

Nachbedingung Fehlschlag: Kunde war nicht angemeldet

Akteure: Kundensachbearbeiter

Auslösendes Ereignis: Abmeldung des Kunden liegt vor

Beschreibung:

1 Kundendaten abrufen

2 Veranstaltung prüfen

3 Abmeldung vornehmen

4 Abmeldebestätigung und evtl. Gutschrift erstellen

IV 26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung

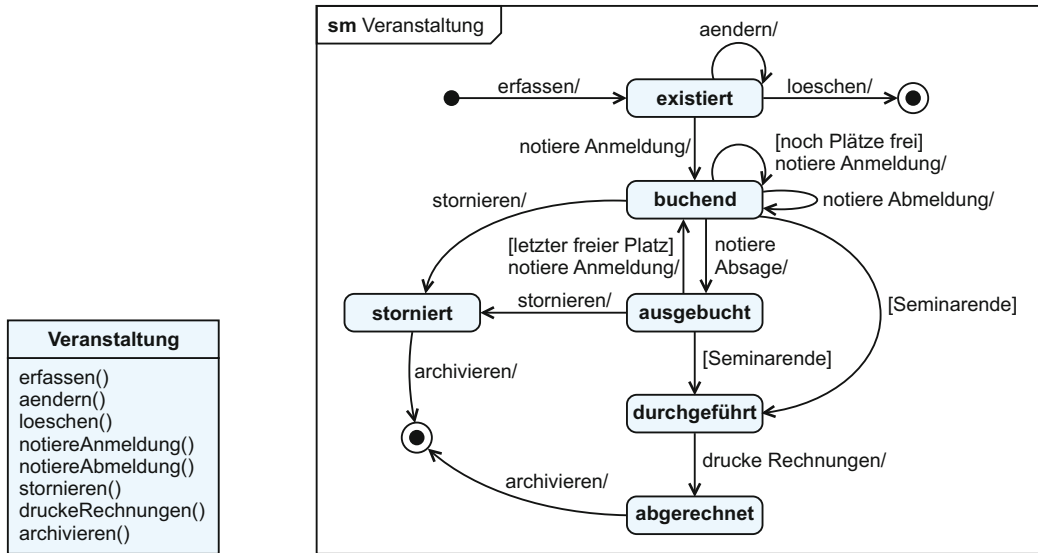


Abb. 26.0-8:
Zustandsautomat
der Klasse
Veranstaltung.

Erweiterung:

1a Kundendaten aktualisieren

3a Y Euro Stornogebühren oder Ersatzteilnehmer, wenn mehr als 4 Wochen vor der Veranstaltung abgemeldet.

3b 100 % Gebühren oder Ersatzteilnehmer, wenn später als X Wochen vor der Veranstaltung abgemeldet.

Alternativen: -

UC Stornieren

Use Case: Stornieren: Von Absage bis Stornomitteilung

Ziel: Stornomitteilung an alle Kunden, Dozenten und Veranstaltungsbetreuer der stornierten Veranstaltung geschickt

Kategorie: primär

Vorbedingung: Kunde ist zu der stornierten Veranstaltung angemeldet, Dozenten halten die stornierte Veranstaltung

Nachbedingung Erfolg: Kunden, Dozent(en), Veranstaltungsbetreuer und Seminarsachbearbeiter sind über Stornierung informiert

Nachbedingung Fehlschlag: -

Akteure: Kundensachbearbeiter, Seminarsachbearbeiter

Auslösendes Ereignis: Veranstaltung muss storniert werden, z. B. wegen Dozentenerkrankung

Beschreibung:

1 Betroffene Kunden, Dozenten und Veranstaltungsbetreuer abrufen

2 Veranstaltung stornieren

3 Stornomitteilung versenden

Erweiterung:

3a Gutschrift versenden

3b Gutschriftskopie an Buchhaltung

Alternativen:

1a Bei Dozentenausfall alternative Dozenten prüfen

3a Alternative Veranstaltungen anbieten

UC Firmenbuchung

Use Case: Firmenbuchung: Von Anmeldung bis Buchung einer firmeninternen Veranstaltung

Ziel: Anmeldebestätigung an Ansprechpartner der Firma geschickt

Kategorie: primär

Vorbedingung: -

Nachbedingung Erfolg: Firma hat Anmeldung und Rechnung erhalten

Nachbedingung Fehlschlag: Mitteilung an Kunden, dass firmeninterne Veranstaltung nicht möglich

Akteure: Seminarsachbearbeiter

Auslösendes Ereignis: Anmeldung der Firma liegt vor

Beschreibung:

1 Firmendaten abrufen

2 Veranstaltung erfassen

3 Buchung vornehmen

4 Anmeldebestätigung erstellen

Erweiterung:

1a Firmendaten aktualisieren

1b Zahlungsmoral überprüfen

Alternativen:

1a Neue Firma erfassen

2a Auf Firmenwünsche eingehen

2b Dozenten über Firmenwünsche informieren

UC Veranstaltung durchführen

Use Case: Veranstaltung durchführen: Von Teilnahme bis Beurteilung.

Ziel: Dozent(en) führen Veranstaltung durch

Kategorie: primär

Vorbedingung: Veranstaltung hat genügend Teilnehmer und ist nicht storniert.

Nachbedingung Erfolg: Veranstaltung ist durchgeführt

Nachbedingung Fehlschlag: -

Akteure: Veranstaltungsbetreuer

Auslösendes Ereignis: Anfangstermin der Veranstaltung

Beschreibung:

1 Teilnehmerliste und Beurteilungsbögen an Teilnehmer und Dozenten

2 Teilnehmerurkunden an Teilnehmer

3 Beurteilungen einsammeln

4 Durchführung bestätigen

IV 26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung

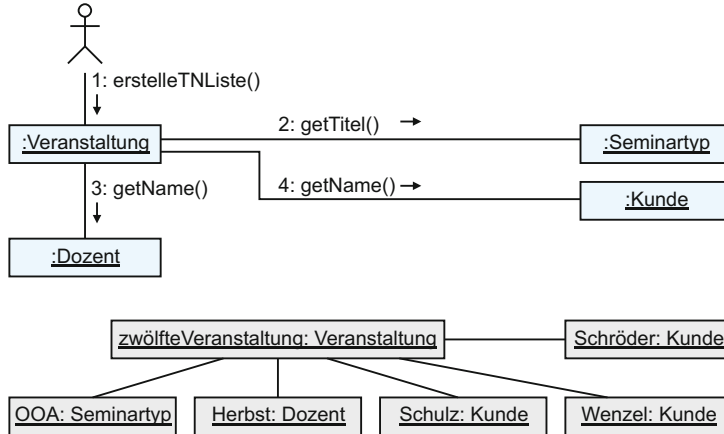
5 Honorarmitteilung an Buchhaltung

Erweiterung: -

Alternativen: -

Das Erstellen der Teilnehmerliste wird durch ein Kommunikationsdiagramm genauer modelliert (Abb. 26.0-9).

Abb. 26.0-9:
Kommunikations-
diagramm (oben)
im Vergleich zu
einem
Objektdiagramm
(unten).



UC Seminarentwicklung

Use Case: Seminarentwicklung: Von Idee zu neuem Seminar.

Ziel: Neues Seminar eingetragen

Kategorie: primär

Vorbedingung: Kunden-, Firmen-, Dozentenanregungen, Marktbeobachtungen.

Nachbedingung Erfolg: Neues Seminar erfasst

Nachbedingung Fehlschlag: -

Akteure: Seminarsachbearbeiter

Auslösendes Ereignis: Beginn der Planungsperiode

Beschreibung:

1 Seminar- und Veranstaltungsstatistik ansehen (Teilnehmerzahlen, betriebswirtschaftliches Ergebnis)

2 Seminar erfassen

Erweiterung:

2a Dozent(en) zuordnen

2b Veranstaltungen festlegen (*Use Case*: Veranstaltungsplanung).

Alternativen:

1a Seminare löschen

1b Seminare modifizieren

UC Dozentenakquirierung

Use Case: Dozentenakquirierung: Von Auswahl bis Verpflichtung

Ziel: Neuen Dozenten eingetragen.

Kategorie: sekundär

Vorbedingung: Marktbeobachtungen

Nachbedingung Erfolg: Neuer Dozent erfasst, Vertrag versandt

Nachbedingung Fehlschlag: -

Akteure: Seminarsachbearbeiter

Auslösendes Ereignis: Beginn der Planungsperiode oder sporadisch

Beschreibung: 1 Neue Seminare und Veranstaltungen ansehen

2 Dozent erfassen

Erweiterung:

2a Dozent Seminaren und Veranstaltungen zuordnen

Alternativen:

2a Dozent(en) löschen

2b Dozent(en) aktualisieren

UC Veranstaltungsplanung

Use Case: Veranstaltungsplanung: Von Terminierung bis Reservierung.

Ziel: Veranstaltung terminiert, Ort festgelegt, Räume reserviert

Kategorie: sekundär

Vorbedingung: -

Nachbedingung Erfolg: Veranstaltung geplant

Nachbedingung Fehlschlag: Veranstaltung nicht fertig geplant

Akteure: Seminarsachbearbeiter

Auslösendes Ereignis: Beginn der Planungsperiode oder sporadisch

Beschreibung:

1 Neue Seminare ansehen

2 Veranstaltung erfassen

Erweiterung:

1a Noch nicht fertig geplante Veranstaltungen ansehen

Alternativen:

2a Veranstaltung unvollständig planen

Logik

Ein Teil der logischen Abhängigkeiten ist im Klassendiagramm eingetragen.

Die Abhängigkeiten bezogen auf eine Teilnehmeranmeldung sind in einem Entscheidungstabellen-Verbund spezifiziert (Abb. 26.0-10).

Trifft die Regel R1 zu, dann ist der Kunde bereits bekannt, und es gibt die gewählte Veranstaltung.

In der Entscheidungstabelle ET1 müssen nun noch die Bedingungen B3, B4 und B5 geprüft werden (Abb. 26.0-11).

IV 26 Fallstudie: SemOrg V1.0 – Die fachliche Lösung

Abb. 26.0-10: ET
für eine Semi-
naranmeldung.

ET0: Erfasse Anmeldungen		R1	R2	R3	R4
B1	Personal-Nr. ok?	J	J	N	N
B2	Veranstaltungs-Nr. ok?	J	N	J	N
A6	Kunden neu eintragen			X	X
A7	Mitteilung »falsche Veranstaltung«		X		X
Weiter bei ET:		ET1	ET2	ET3	

Abb. 26.0-11: ET
für »Buchung
verarbeiten« und
»Zahlungsmoral
entscheiden«.

ET1: Buchung verarbeiten		R1	R2	R3	R4	E/se
B3	Bereits angemeldet?	N	N	N	N	
B4	Teilnehmerzahl aktuell < max?	J	J	N	N	
B5	Zahlungsverzug?	J	N	J	N	
A11	Mitteilung »bereits angemeldet«					X
A8	Mitteilung »ausgebucht«			X	X	
A9	Mitteilung »Zahlungsverzug«	X		X		
Buchung vornehmen (A1 bis A5)			X			
Weiter bei ET:		ET1.1				

ET1.1: Zahlungsmoral entscheiden		R1
Zahlungsverzug kritisch? (Kundensachbearbeiter Höhe anzeigen)		N
Buchung vornehmen (A1 bis A5)		X

Die Regel R2 der Entscheidungstabelle ET0 gibt die Situation wieder, dass der Kunde bereits bekannt ist, er sich aber zu einer nicht existierenden Veranstaltung anmelden will. Da er sowieso eine Mitteilung erhält, kann gleich noch überprüft werden, ob er im Zahlungsverzug ist (Abb. 26.0-12).

Abb. 26.0-12: ET
für
»Zahlungsverzug
prüfen«.

ET2: Zahlungsverzug prüfen		R1
B5	Zahlungsverzug?	J
A9	Mitteilung »Zahlungsverzug«	X

Die Regel R3 in der Entscheidungstabelle ET0 beschreibt die Situation, dass es sich um einen neuen Kunden handelt und dass die gewünschte Veranstaltung vorhanden ist. Es muss nur geprüft werden, ob die Veranstaltung ausgebucht ist (Abb. 26.0-13).

Abb. 26.0-13: ET
für
»Teilnehmerzahl
prüfen«.

ET3: Teilnehmerzahl prüfen		R1	R2
B4	Teilnehmerzahl aktuell < max?	J	N
Buchung vornehmen (A1 bis A5)		X	
A8	Mitteilung »ausgebucht«		X

27 Fallstudie: Fensterheber – Die fachliche Lösung

Auf der Grundlage des Pflichtenheftes der Fallstudie Fensterheber (»Fallstudie: Fensterheber – Die Spezifikation«, S. 117) wird hier die entsprechende fachliche Lösung beschrieben. Sie ist in die folgenden Bereiche gegliedert:

- Statik
- Dynamik und Logik

Die Softwarefunktion »Fensterheber« ist Teil des eingebetteten Systems »Türsteuergerät«. Die Funktionalität »Fensterheber« implementiert die elektrische Hebefunktion aller im Fahrzeug befindlichen Seitenfenster. Da sich die in der Software zu realisierenden Aufgaben für jedes einzelne Seitenfenster ähneln, wird im Folgenden repräsentativ die Software für das Fenster der Fahrertür betrachtet.

Fensterheber – Fachliche Lösung V1.0

Version	Autor	Quelle	Status	Datum	Kommentar
0.1	Hans Echt	Produktmanager Türsteuergerät	in Bear- beitung	11/09	

Statik

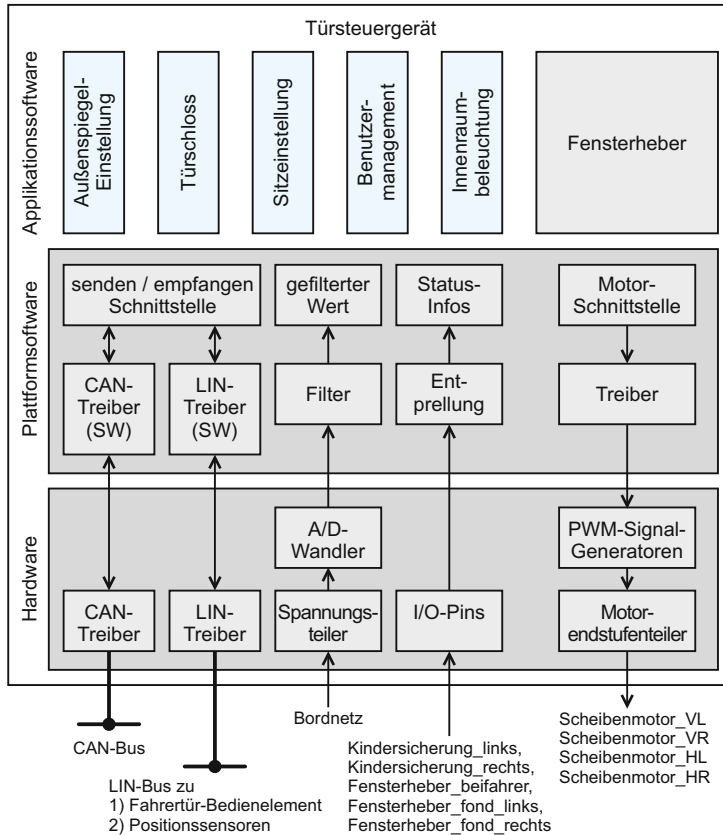
Da es sich bei dem Fensterheber um ein eingebettetes System handelt, ist es äußerst schwierig, die Software losgelöst von der Hardware zu betrachten. Die Abb. 27.0-1 beschreibt deswegen grob die eingesetzte Hardware im Hinblick auf die bereitgestellten Dienste. Des Weiteren erläutert das Schaubild im Ansatz, wie die Hardware, die Plattformsoftware (hardwarespezifische Software) und die Applikationssoftware zusammenarbeiten. Laut Spezifikation werden auf der Hardwareplattform neben dem Fensterheber noch weitere Applikationen ausgeführt. Diese sind im Schaubild der Abb. 27.0-1 angedeutet, werden hier jedoch nicht weiter beschrieben.

Nachdem in der Abb. 27.0-1 das Zusammenspiel des Gesamtsystems beschrieben wurde, präzisiert Abb. 27.0-2 die Softwarestruktur der Fensterheberapplikation und der von ihr verwendeten Plattformsoftware unter Zuhilfenahme eines Paketdiagramms.

Die eingesetzte Plattformsoftware besteht zum einen aus einem zugekauften, speziell für den verwendeten Prozessor entwickelten Betriebssystem. Dieses Betriebssystem kapselt die direkte Be-

IV 27 Fallstudie: Fensterheber – Die fachliche Lösung

Abb. 27.0-1: Das
System
Türsteuergerät
V1.0.



nutzung der Controllerhardware und stellt betriebssystemtypische Funktionen wie beispielsweise *Threads*, *Scheduling* oder Speicherzugriffsroutinen bereit. Um eine effiziente Realisierung der Applikation zu ermöglichen, müssen jedoch weitere applikationsspezifische Treiber und Schnittstellen entwickelt werden. So soll die Plattformsoftware des Türsteuergeräts unter anderem eine einfache Steuerung des Scheibenmotors (Motor_Schnittstelle) erlauben oder den schnellen Zugriff auf die Werte des Positionssensors (Positionsensor_Schnittstelle) ermöglichen.

Die Fensterheber-Applikation unterscheidet zwischen der Steuerung des Fahrerfensters (Fenstersteuerung_VL), des Beifahrerfensters (Fenstersteuerung_VR) und der Fondfenster (Fenstersteuerung_Hinten). Die Besonderheit der Fenstersteuerung_VL liegt im erweiterten Funktionsumfang der Fahrerbedienkonsole. Die getrennte Betrachtung der hinteren Fenster ist mit der Kindersicherungsfunktionalität begründet. Weiterhin existieren getrennte Klassen zur Realisierung des Einklemmschutzes und der Fehlererkennung.

27 Fallstudie: Fensterheber – Die fachliche Lösung IV

Da die Plattformsoftware abhängig von der eingesetzten Hardware ist und zum anderen abhängig vom zugekauften Betriebssystem, fokussiert die weitere Betrachtung ausschließlich auf die zu entwickelnde Applikationssoftware Fensterheber.

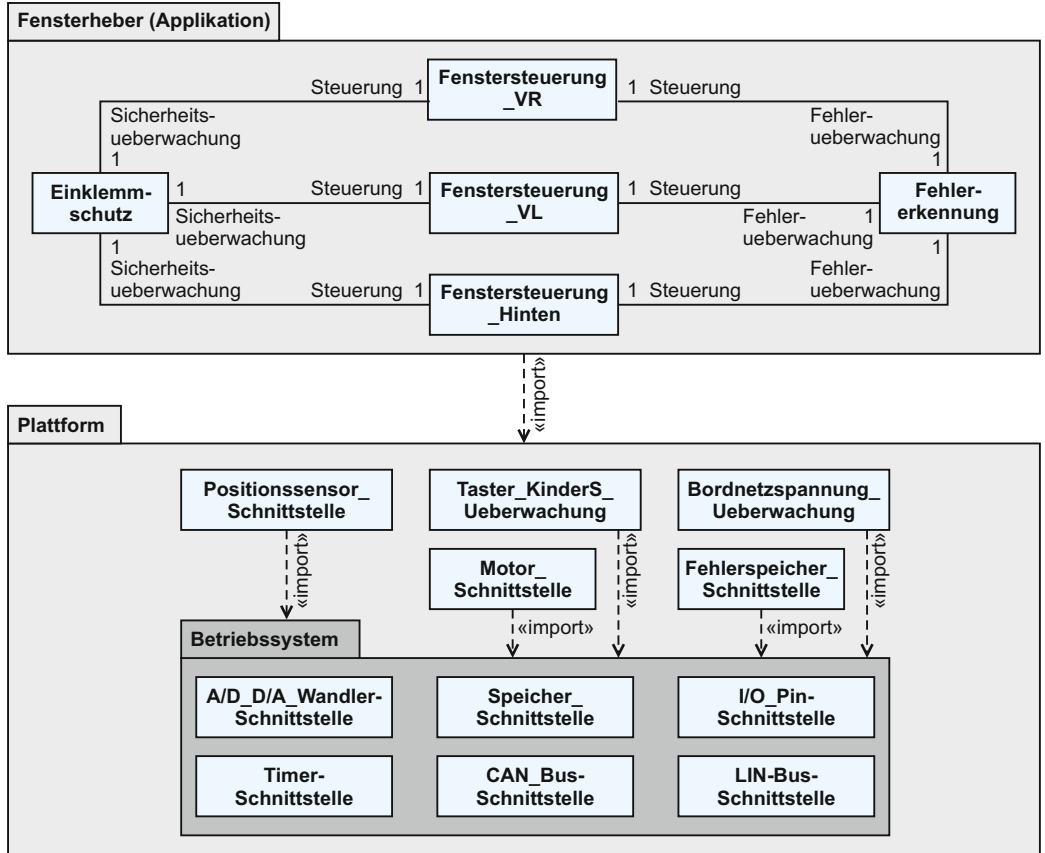


Abb. 27.0-2:
Paketdiagramm
der Fensterheber-
funktion.

Dynamik und Logik

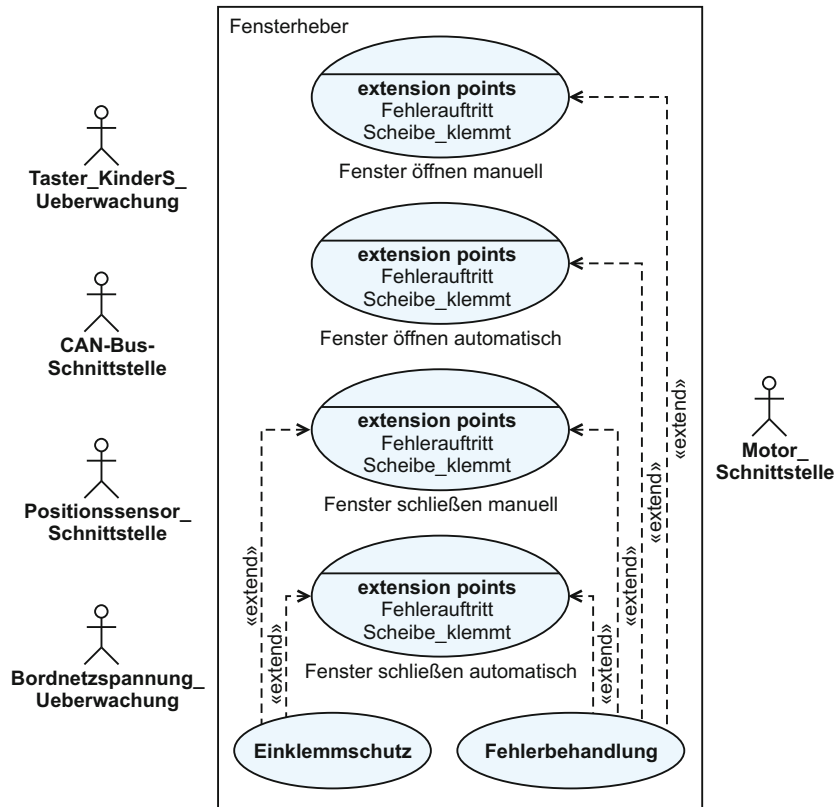
Das Use Case-Diagramm der Abb. 27.0-3 beschreibt sowohl den Kontext des Fensterhebers der Fahrtür als auch die vom System bereitgestellten Funktionen. Im Folgenden wird die Applikationssoftware als das zu entwickelnde System betrachtet. Aus diesem Grund handelt es sich bei den Akteuren des Use Case-Diagramms um die bereits aus der Abb. 27.0-2 bekannten Plattformkomponenten und *nicht* um die entsprechende Hardware.

IV 27 Fallstudie: Fensterheber – Die fachliche Lösung

Bei der Verwendung von *Use Case*-Diagrammen ist zu beachten, dass mit dem Diagramm die Systemgrenzen beschrieben werden. So sind laut der Abb. 27.0-3 die Motor-Schnittstellen, die Tasterüberwachung und die Positionssensoren *nicht* Teil des zu entwickelnden Systems.

Aus Gründen der Übersichtlichkeit wurden in der Abb. 27.0-3 auf die Assoziationen zwischen Akteuren und *Use Cases* verzichtet. Jeder Akteur nimmt an jedem *Use Case* teil. Eine Ausnahme bildet ausschließlich der *Use Case* »Einklemmschutz«. Der CAN-Bus und die Fensterhebertaster interagieren *nicht* mit diesem *Use Case*.

Abb. 27.0-3:
Use Case-
Diagramm zur
Fallstudie
»Fensterheber«.



Die *Use Cases* Fenster öffnen manuell (1), Fenster öffnen automatisch (2), Fenster schließen manuell (3) und Fenster schließen automatisch (4) beschreiben die Standardfunktionalität des elektrischen Fensterhebers. Die *Use Cases* beschreiben das Öffnen/Schließen der Seitenscheiben durch anhaltenden Tastendruck (1 und 3), sowie das vollautomatische Öffnen/Schließen des Seitenfensters über den kompletten Hub (2 und 4). Die Anforderungen zum Bewegen der Scheiben können zum einen von den entsprechenden

27 Fallstudie: Fensterheber – Die fachliche Lösung IV

Tastern oder durch CAN-Bus-Telegramme (z. B. vom Steuergerät des Cabriooverdecks oder von der Zentralverriegelung) gestellt werden. Der *Use Case* Einklemmschutz (5) beschreibt eine Sicherheitsfunktionalität des Fensterhebers. Sie soll verhindern, dass durch das Schließen des Fensters Beschädigungen oder gar Verletzungen verursacht werden.

UC Fenster öffnen manuell

Use Case: Fenster öffnen manuell

Ziel: Öffnen des Fensters

Kategorie: primär

Vorbedingungen: Fensterposition!= Unten && Bordnetzspannung > = 10V

Nachbedingung Erfolg: Fenster ist entsprechend geöffnet

Nachbedingungen Fehlschlag: Fehlerspeichereintrag vorhanden & Fehler-CAN-Telegramm wurde versendet

Akteure: CAN-Bus_Schnittstelle, Positionssensor_Schnittstelle, Taster_KinderS_Ueberwachung, Motor_Schnittstelle

Auslösendes Ereignis: Entsprechendes CAN-Telegramm oder Betätigung des Tasters

Beschreibung:

1 Empfang des entsprechenden Signals via CAN-Bus oder Taster

2 Motoren_Schnittstelle zum Öffnen des Fensters veranlassen

3 Motoren_Schnittstelle zum Ausschalten des Motors veranlassen, wenn das Öffnen-Signal nicht mehr anliegt oder wenn Fensterposition = Unten

Erweiterung: –

Alternativen:

2a Fehlerfall (für genauere Beschreibung siehe **UC Fehlerfall**)

UC Fenster öffnen automatisch

Use Case: Fenster öffnen automatisch

Ziel: Vollständiges Öffnen des Fensters

Kategorie: primär

Vorbedingungen: Fensterposition!= Unten && Bordnetzspannung > = 10V

Nachbedingung Erfolg: Fensterposition = Unten

Nachbedingungen Fehlschlag: Fehlerspeichereintrag vorhanden & Fehler-CAN-Telegramm wurde versendet

Akteure: CAN-Bus_Schnittstelle, Positionssensor_Schnittstelle, Taster_KinderS_Ueberwachung, Motor_Schnittstelle

Auslösendes Ereignis: Entsprechendes CAN-Telegramm oder Betätigung des Tasters

Beschreibung:

1 Empfang des entsprechenden Signals via CAN-Bus oder Taster

2 Motoren_Schnittstelle zum Öffnen des Fensters veranlassen

IV 27 Fallstudie: Fensterheber – Die fachliche Lösung

3 Motoren_Schnittstelle zum Ausschalten des Motors veranlassen, wenn Fensterposition = Unten

Erweiterung: -

Alternativen:

2a Fehlerfall (für genauere Beschreibung siehe **UC Fehlerfall**)

3a Fehlerfall (für genauere Beschreibung siehe **UC Fehlerfall**)

UC Fenster schließen manuell

Use Case: Fenster schließen manuell

Ziel: Schließen des Fensters

Kategorie: primär

Vorbedingungen: Fensterposition!= Oben && Bordnetzspannung > = 10V

Nachbedingung Erfolg: Fenster ist entsprechend geschlossen

Nachbedingungen Fehlschlag: Fehlerspeichereintrag vorhanden & Fehler-CAN-Telegramm wurde versendet

Akteure: CAN-Bus_Schnittstelle, Positionssensor_Schnittstelle, Taster_KinderS_Ueberwachung, Motor_Schnittstelle

Auslösendes Ereignis: Entsprechendes CAN-Telegramm oder Betätigung des Tasters

Beschreibung:

1 Empfang des entsprechenden Signals via CAN-Bus oder Taster

2 Motoren_Schnittstelle zum Schließen des Fensters veranlassen

3 Motoren_Schnittstelle zum Ausschalten des Motors veranlassen, wenn das Schließen-Signal nicht mehr anliegt oder Fensterposition = Oben

Erweiterung: -

Alternativen:

3a Fehlerfall (für genauere Beschreibung siehe **UC Fehlerfall**)

3b Einklemmschutz (für genauere Beschreibung siehe **UC Einklemmschutz**)

UC Fenster schließen automatisch

Use Case: Fenster schließen automatisch

Ziel: Vollständiges Schließen des Fensters

Kategorie: primär

Vorbedingungen: Fensterposition!= Oben && Bordnetzspannung > = 10V

Nachbedingungen Erfolg: Fensterposition = Oben

Nachbedingungen Fehlschlag: Fehlerspeichereintrag vorhanden & Fehler-CAN-Telegramm wurde versendet

Akteure: CAN-Bus_Schnittstelle, Positionssensor_Schnittstelle, Taster_KinderS_Ueberwachung, Motor_Schnittstelle

Auslösendes Ereignis: Entsprechendes CAN-Telegramm oder Betätigung des Tasters

Beschreibung:

- 1** Empfang des entsprechenden Signals via CAN-Bus oder Taster
- 2** Motoren_Schnittstelle zum Schließen des Fensters veranlassen
- 3** Motoren_Schnittstelle zum Ausschalten des Motors veranlassen, wenn Fensterposition = Oben

Erweiterung: -

Alternativen:

- 4a** Fehlerfall (für genauere Beschreibung siehe **UC Fehlerfall**)
- 4b** Einklemmschutz (für genauere Beschreibung siehe **UC Einklemmschutz**)

UC Einklemmschutz

Use Case: Einklemmschutz

Ziel: Verhindern, dass Sach- oder Personenschäden durch das schließende Fenster verursacht werden

Kategorie: primär

Vorbedingung: Schließvorgang aktiv

Nachbedingung Erfolg: Fensterposition = Unten

Nachbedingungen Fehlschlag: -

Akteure: Positionssensor_Schnittstelle, Motor_Schnittstelle

Auslösendes Ereignis: Fensterposition!= Oben && Motor ist angesteuert, um Fenster zu schließen && Fensterbewegung = keine

Beschreibung:

- 1** Überwachung der Fensterposition, der Fensterbewegung und der Motoransteuerung
- 2** Motoransteuerung soll Fenster schließen und der Stillstand des Fensters wird bemerkt bevor Fensterposition = Oben
- 3** Stoppen des Scheibenmotors
- 4** Motoren_Schnittstelle zum Öffnen des Fensters veranlassen

Erweiterung: -

Alternativen: -

Die Abb. 27.0-4 zeigt ein Zustands-Diagramm, in dem die *Use Cases* (1) bis (5) näher beschrieben werden. Das Zustandsdiagramm zeigt zwei parallel agierende Automaten. Die Aufgabe des oberen Automaten ist es, die Benutzereingaben in Anweisungen an den unteren Automaten umzusetzen. Der untere Automat implementiert die Steuerung des Scheibenmotors und den Einklemmschutz.

Zur Interaktion versendet der obere Automat die Nachrichten »hoch«, »runter« und »aus« an den unteren Automaten. Hierzu wurde die Notation »send <X>« verwendet um zu modellieren, dass der Automat die Nachricht X versendet. Im unteren Automaten findet die Notation »receive <X>« Anwendung um den Auftritt des entsprechenden Empfangsereignisses abzubilden. Hierbei handelt es sich um eine informelle Notation zur Modellierung der Kommunikation zwischen den Automaten.

IV 27 Fallstudie: Fensterheber – Die fachliche Lösung

Neben den receive-Ereignissen existieren noch die Ereignisse `MovChanged`, `PosChanged` sowie `OpChanged`. `MovChanged` ist ein Ereignis welches von der Plattformkomponente `Positionssensor_Schnittstelle` erzeugt wird, wenn sich die Bewegungsinformation des Sensors ändert. `PosChanged` wird von derselben Plattformkomponente erzeugt, zeigt jedoch eine Änderung der Positionsinformation an.

Die Anforderung zum Öffnen/Schließen einer Seitenscheibe kann entweder von der `CAN-Bus_Schnittstelle` oder von der `Taster_KinderS_Ueberwachung` gestellt werden. Die Variable `op` fasst den Inhalt der Anforderungen des Fenstertasters und derer vom CAN-Bus zusammen. Die Tab. 27.0-1 erklärt, unter welchen Umständen die Variable `op` welche Werte annimmt. Die Signale mit dem Präfix `CAN.` kommen vom CAN-Bus, die Signale mit dem Präfix `S1.` werden vom Tastenfeld in der Fahrertür erzeugt.

Tab. 27.0-1: Werte
für die Variable
`op`.

Signal	Wert	Führt zu
<code>CAN.WIN_VL_OP</code>	01	<code>op</code> = FensterHoch
<code>CAN.WIN_VL_OP</code>	10	<code>op</code> = FensterHochAuto
<code>CAN.WIN_VL_CL</code>	01	<code>op</code> = FensterRunter
<code>CAN.WIN_VL_CL</code>	10	<code>op</code> = FensterRunterAuto
<code>S1.FHB_VL</code>	hoch m.	<code>op</code> = FensterHoch
<code>S1.FHB_VL</code>	hoch a.	<code>op</code> = FensterHochAuto
<code>S1.FHB_VL</code>	runter m.	<code>op</code> = FensterRunter
<code>S1.FHB_VL</code>	runter a.	<code>op</code> = FensterRunterAuto
Sonst		<code>op</code> = keine

Die Befehle `Motor_aus`, `Motor_runter`, `Motor_hoch` werden von der Plattformapplikation »Motorschnittstelle« verstanden und über das Betriebssystem in die entsprechende Ansteuerung der Motorendstufe umgewandelt.

Hinweis

Mit Hilfe eines Zustandsautomaten können *Use Cases* präzisiert und formalisiert werden. Durch die Präzisierung ist es möglich, noch existierende aber unbemerkte Unklarheiten in den Anforderungen aufzudecken. Die Formalisierung erlaubt es oftmals, bestehende Widersprüche und Unregelmäßigkeiten aufzudecken.

Eine Kante, für die *kein* expliziter Trigger definiert wurde, wird durch das `Completion` Ereignis ausgelöst. Dieses Ereignis wird genau dann einmalig geworfen, wenn die *Entry*-Aktionen und *Do*-Aktivitäten des Zustands ausgeführt wurden. Sind keine *Entry*-Aktionen bzw. *Do*-Aktivitäten definiert, wird das Ereignis einmalig beim Betreten des Zustands geworfen. Möchte man ausdrücken, dass eine Transition gefeuert werden soll, sobald eine Bedingung erfüllt ist, muss das Schlüsselwort `when` verwendet werden, das ausdrückt, dass die

27 Fallstudie: Fensterheber – Die fachliche Lösung IV

Transition durch ein Änderungsereignis getriggert wird. Ein Änderungsereignis wird immer dann geworfen, wenn sich die dem Schlüsselwort nachfolgende boolesche Bedingung zu wahr auswertet.

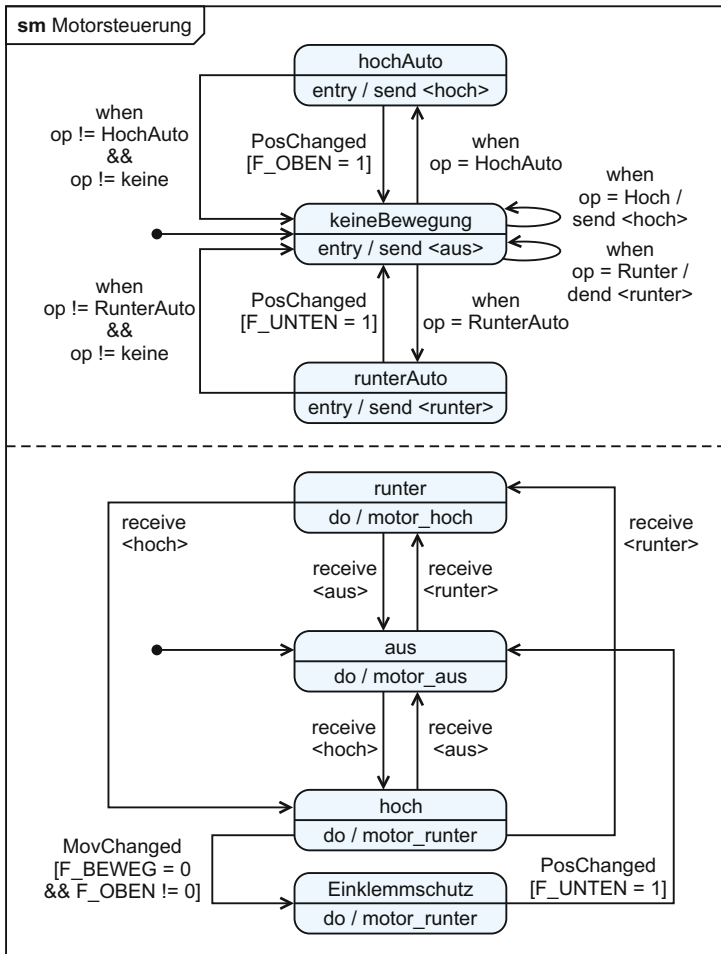


Abb. 27.0-4:
Zustandsautomat
zur
Standardfunktion
und zum
Einklemmschutz.

Der *Use Case* Fehlerbehandlung (6) beschreibt das Verhalten des Türsteuergeräts im Fehlerfall. Zur genaueren Beschreibung des Verhaltens im Fehlerfall wurden Sequenzdiagramme erstellt. Sie beschreiben, welche Fehler vom Steuergerät erkannt werden und wie es diese behandelt.

UC Fehlerbehandlung

Use Case: Fehlerfall

Ziel: Erkennung und Behandlung möglicher Systemfehler

Kategorie: primär

Vorbedingungen: -

IV 27 Fallstudie: Fensterheber – Die fachliche Lösung

Nachbedingungen Erfolg: Fehlerspeichereintrag vorgenommen & entsprechendes CAN-Telegramm gesendet

Nachbedingungen Fehlschlag: -

Akteure: CAN-Bus_Schnittstelle, Positionssensor_Schnittstelle, Taster_KinderS_Ueberwachung, Motor_Schnittstelle, Bordnetzspannung_Ueberwachung

Auslösendes Ereignis:

1 Bordnetzspannung < 10V && Anforderung zur Fensterbewegung

2 Fensterposition!= Unten && Fensterbewegung = Keine && Anforderung zum Öffnen des Fensters vorhanden

3 Fensterposition!= Unten && Fensterbewegung = Runter && Anforderung zum automatischen Öffnen des Fenster vorhanden && Fensterbewegung = Runter dauert bereits länger als 3 Sekunden an

4 Fensterposition!= Oben && Fensterbewegung = Hoch && Anforderung zum automatischen Schließen des Fenster vorhanden && Fensterbewegung = Hoch dauert länger bereits länger als 3 Sekunden an

Beschreibung:

1 Überwachung der Fenstersteuerung

2 Erkennung eines der auslösenden Ereignisse

3a Wenn Ereignis 1 erkannt wurde soll die Bewegung des Motors verhindert und eine Fehlernachricht auf dem CAN-Bus gesendet werden

3b Wenn eines der Ereignisse 2 bis 4 erkannt wurde, dann soll ein Eintrag im Fehlerspeicher vorgenommen werden und eine Fehlermeldung auf dem CAN-Bus gesendet werden

Erweiterung: -

Alternativen: -

Hinweis

Nutzt man Sequenzdiagramme zur Beschreibung von *Use Cases*, dann sollten die Lebenslinien des Sequenzdiagramms die Akteure aus dem *Use Case*-Diagramm oder das betrachtete System als Black-Box darstellen.

Die Abb. 27.0-5 zeigt das Fehlerszenario »Bordnetzspannung_Niedrig«. Liegt die Bordnetzspannung des Fahrzeuges zum Beginn einer Scheibenbewegung unter der 10V-Schwelle, so wird der Motor nicht in Bewegung versetzt und eine entsprechende Fehlermeldung wird über den CAN-Bus versendet.

Die Abb. 27.0-6 zeigt das Fehlerszenario »Keine_Bewegung«. Wenn nach eingehender Anforderung das Fenster zu öffnen zwar Spannung am Motor anliegt, aber der Positionssensor keine Bewegung erkennt, dann wird eine entsprechende Fehlermeldung auf dem CAN-Bus versendet und ein Eintrag im Fehlerspeicher vorgenommen.

27 Fallstudie: Fensterheber – Die fachliche Lösung IV

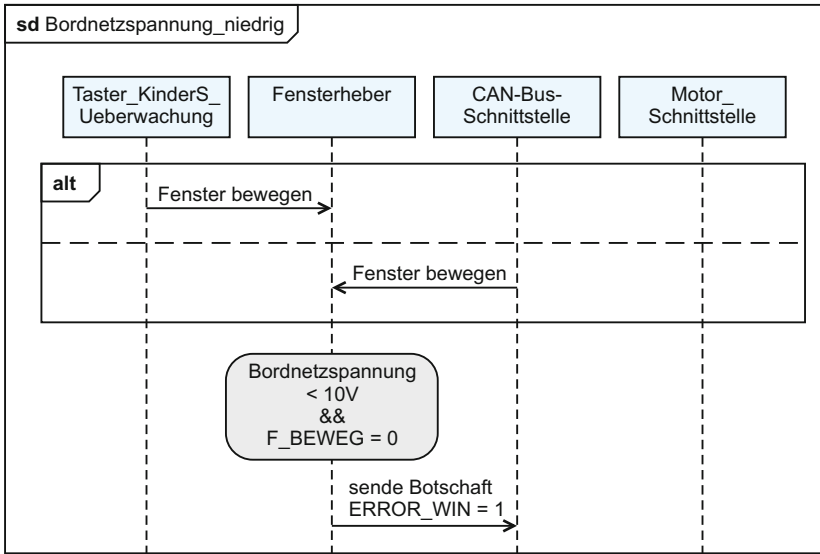


Abb. 27.0-5:
Fehlerbild Bord-
netzspannung.

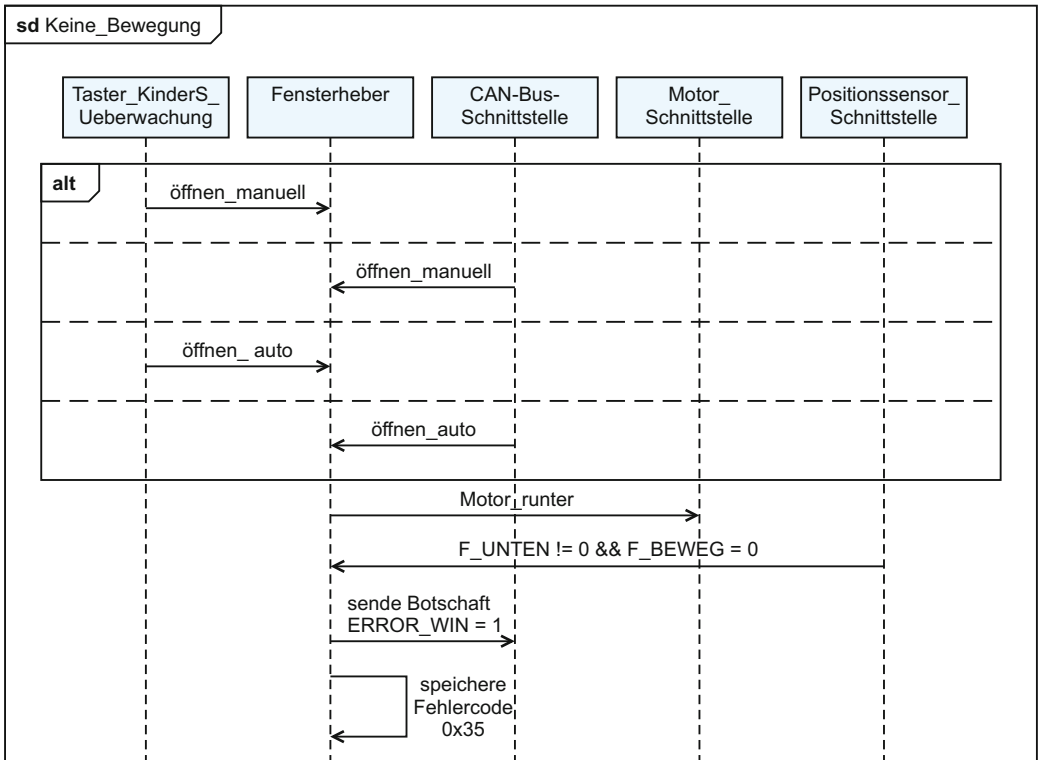


Abb. 27.0-6:
Fehlerbild
Verklemmung.

IV 27 Fallstudie: Fensterheber – Die fachliche Lösung

Die Abb. 27.0-7 beschreibt das Fehlerbild »Gestänge_Gebrochen«. Wird beim automatischen Schließen oder Öffnen der Fensterscheibe der Motor länger als 3 Sekunden angesprochen, ohne dass das Fenster seine Endstellung erreicht hat, wird ein Fehler angenommen. In diesem Fall muss wiederum eine Fehlermeldung auf dem CAN-Bus gesendet und eine Eintragung in den Fehlerspeicher geschrieben werden.

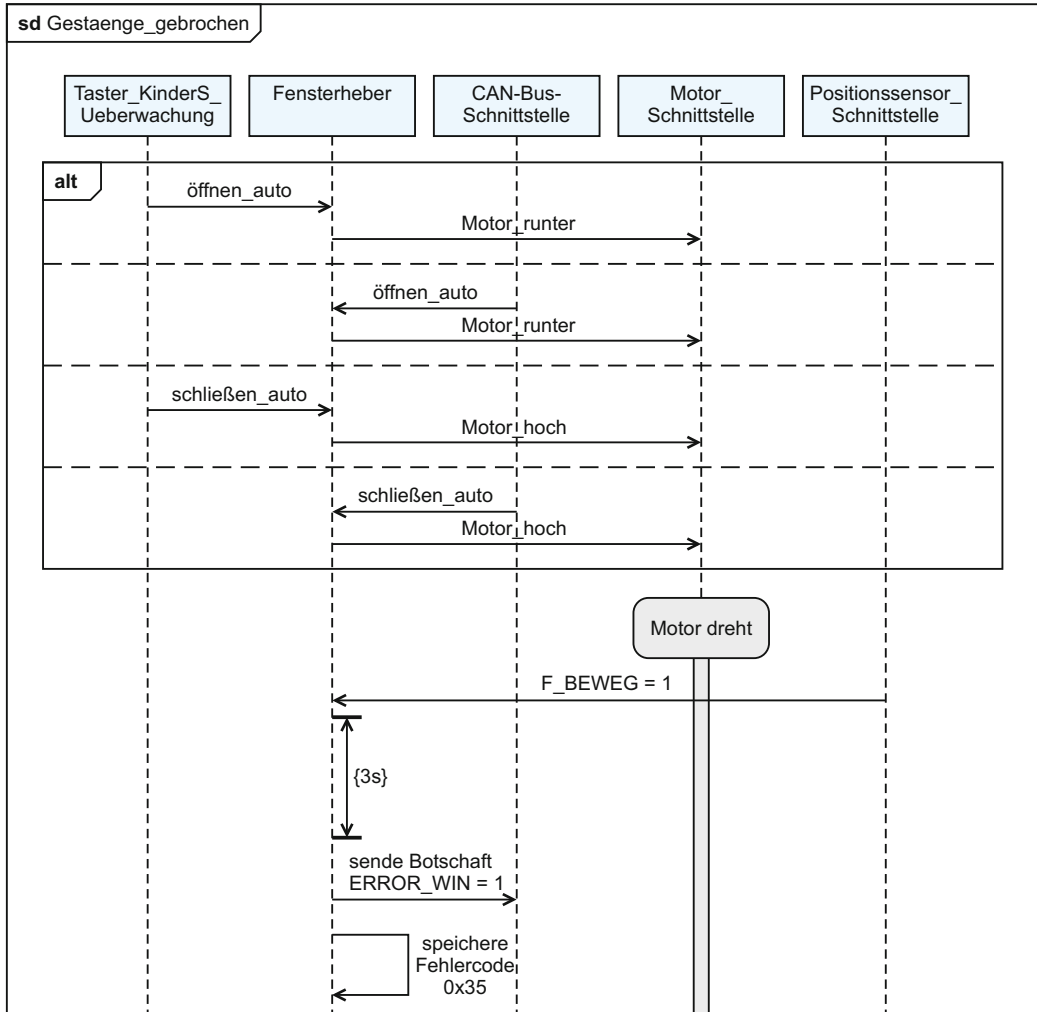
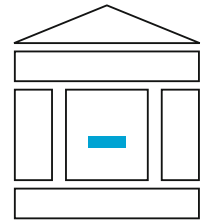


Abb. 27.0-7:
Fehlerbild Zeit-
überschreitung.

28 Modellierte Anforderungen analysieren, verifizieren und abnehmen

Die fachliche Lösung bzw. das Produktmodell bildet den Ausgangspunkt für die Erstellung der Softwarearchitektur. Jeder Fehler in der fachlichen Lösung muss mit überproportionalem Aufwand behoben werden, wenn er nicht schon im Produktmodell, sondern erst später gefunden wird. Die kostengünstigste Qualitätssicherung liegt also vor, wenn die fachliche Lösung mit allen begleitenden Artefakten überprüft wird. Eine Überprüfung besteht aus zwei Schritten:

- Analyse der fachlichen Lösung
- Verifikation der fachlichen Lösung



Analysieren

Bei der Analyse geht es darum festzustellen, ob die modellierten Anforderungen der Modellierungsmethode und -beschreibung entsprechen. Wird beispielsweise die UML verwendet, dann muss überprüft werden, ob die verwendeten Diagrammarten auch der UML-Notation entsprechen. Je nach eingesetztem UML-Werkzeug kann dies auch automatisch erfolgen. Die Analyse bezieht sich also darauf, ob das Modell den Konventionen und syntaktischen Regeln der gewählten Modellierungsnotation entsprechen. Außerdem muss geprüft werden, ob die Semantik der Anforderungen in sich konsistent ist.

Verifizieren

Im Gegensatz zur Überprüfung der Anforderungsspezifikation kann die fachliche Prüfung des Produktmodells gegen die Anforderungsspezifikation erfolgen. Es muss also überprüft werden, ob die modellierten Anforderungen semantisch identisch sind mit den spezifizierten Anforderungen. Ist dies nicht der Fall, dann würde ein anderes Produkt entwickelt, als spezifiziert wurde.

Prüfmethoden

Zur Analyse und Verifikation können z.B. folgende manuelle Prüfmethoden eingesetzt werden (siehe »Lehrbuch der Softwaretechnik – Softwaremanagement«):

IV 28 Modellerte Anforderungen analysieren, verifizieren und abnehmen

- Inspektion
- *Review*
- *Walkthrough*
- Stellungnahme
- *Round Robin Review*

Ergänzend können auch automatische Prüfmethode verwendet werden, z. B. für die Überprüfung eines UML-Modells.

Checklisten Ein wichtiges Hilfsmittel, das dafür sorgt, dass bei der Überprüfung nichts vergessen wird, sind **Checklisten**.

Abnehmen

Das *Requirements Engineering* endet mit der Qualitätsüberprüfung der fachlichen Lösung. Eine erfolgreiche Analyse und Verifikation sollte durch eine formelle Abnahme abgeschlossen werden.

Weiterführende
Literatur

[ElSc03]

Glossar

Abstrakte Klasse (*abstract class*)

Spiele eine wichtige Rolle in Generalisierungsstrukturen (Generalisierung), wo sie die Gemeinsamkeiten von einer Gruppe von Unterklassen definiert; im Gegensatz zu einer Klasse können von einer abstrakten Klasse *keine* Objekte erzeugt werden. Damit eine abstrakte Klasse verwendet werden kann, muss von ihr zunächst eine Unterklasse abgeleitet werden.

Abstraktion (*abstraction*) 1 Sicht auf ein Problem, bei dem die wesentlichen Informationen herausgezogen werden, die für einen speziellen Zweck relevant sind, und die restlichen Informationen ignoriert werden.

2 Der Prozess, um eine Abstraktion zu bilden.

Aggregation (*aggregation*) 1 Sonderfall der Assoziation. Liegt vor, wenn zwischen den Objekten der beteiligten Klassen eine »ist Teil von«-Beziehung bzw. »besteht aus«-Beziehung vorliegt. Teil-Klassen stehen in Beziehung zu einer Gesamtheits-Klasse bzw. Aggregat-Klasse. Der Komplexitätsgrad wird durch Kardinalitäten angegeben.

Kann aber auch eine Ist-Teil-Von-Hierarchie (is-part-of) zwischen Entitäten beschreiben; typisch für Stücklisten. Kennzeichnend ist, dass die Attribute der verknüpften Entitätstypen unterschiedlich sind.

2 Beschreibt eine Ist-Teil-Von-Hierarchie (is-part-of) zwischen Entitäten; typisch für Stücklisten. Kennzeichnend ist, dass die Attribute der verknüpften Entitätstypen unterschiedlich sind.

Akteur (*actor*) Ein Akteur ist in der UML ein Element, das mit einem System interagiert. Akteure befinden sich außerhalb des Systems. Akteure können Personen oder externe Systeme sein. Akteure sind Rollen, keine konkreten Personen oder Systeme.

Aktion (*action*) Kleinste ausführbare Einheit innerhalb einer Aktivität eines UML-Aktivitätsdiagramms. Eine Aktion kann ausgeführt werden, wenn die Vorgänger-Aktion beendet ist, wenn notwendige Daten zur Verfügung stehen oder wenn ein Ereignis auftritt. Eine Aktion kann auch ein Aktivitätsaufruf sein, d. h. von der Ausführung her gesehen, kann sich hinter einem Aktionsknoten eine sehr komplexe Verarbeitung verbergen.

Aktivität (*activity*) 1 Modelliert im UML-Aktivitätsdiagramm die Ausführung von Funktionalität bzw. Verhalten. Sie wird durch mehrere Knoten dargestellt, die durch gerichtete Kanten miteinander verbunden sind. Es lassen sich Aktionsknoten, Kontrollknoten und Objektknoten unterscheiden.

2 Spezifiziert in einem Zustandsdiagramm die durchzuführende Verarbeitung. Aktivitäten können an eine Transition angetragen oder mit einem Zustand verbunden sein.

Aktivitätsdiagramm (*activity diagram*) UML-Diagramm, das eine Aktivität durch ein großes Rechteck mit abgerundeten Ecken modelliert. Die Verarbeitungsschritte der Aktivität werden durch Graphen dargestellt, der aus Knoten und Pfeilen besteht. Die Knoten entsprechen im einfachsten Fall Aktionen. Die Pfeile (gerichtete Kanten) verbinden die Knoten und stellen im einfachsten Fall den Kontrollfluss der Aktivität dar. Viele Aktivitäten benötigen Eingaben und produzieren Ausgaben. Sie werden durch Parameterknoten beschrieben.

Anforderung (*requirement*) 1 Aussage über eine zu erfüllende qualitative und/oder quantitative Eigenschaft eines Produkts

2 eine vom Auftraggeber festgelegte Systemspezifikation, um ein System für den Entwickler zu definieren.

Glossar

Anwender (*user*) Mitglieder einer Institution oder Organisationseinheit, die zur Erfüllung ihrer fachlichen Aufgaben Computersysteme einsetzen (siehe auch Benutzer).

Anwendungssoftware (*application software*) Software, die Aufgaben des Anwenders mit Hilfe eines Computersystems löst. Setzt in der Regel auf der Systemsoftware der verwendeten Hardware auf bzw. benutzt sie zur Erfüllung der eigenen Aufgabe.

Artefakt (*artifact*) Ein greifbares Stück Information, das durch Mitarbeiter erzeugt, geändert und benutzt wird, wenn sie Aktivitäten ausführen. Kann ein Modell, ein Modellelement oder ein Dokument sein. Beispiele: Dokument, z. B. Lastenheft, Modell, z. B. objektorientiertes Analysemodell, Quellcode, z. B. C++-Programm. (Syn.: Arbeitsprodukt)

Artefakt-Schablone (*artifact template*) Legt die Struktur, den Inhalt und das Layout eines Artefakts fest; kann durch Richtlinien oder implizit durch Softwarewerkzeuge erfolgen. (Syn.: Artefakt-Muster)

Assoziation (*association*) Eine Assoziation modelliert Objektbeziehungen zwischen Objekten einer oder mehrerer Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwischen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch Angabe der Multiplizität an jedem Ende und einen optionalen Assoziationsnamen oder Rollennamen. Eine Assoziation kann bidirektional oder unidirektional sein.

Assoziationsklasse (*association class*) Besitzt eine Assoziation selbst wieder Attribute, Operationen und/oder Assoziationen zu anderen Klassen, dann wird sie zur Assoziationsklasse. Sie kann nach einem festen Schema in eine »normale« Klasse transformiert werden.

Assoziationsname (*association name*) Beschreibt die Bedeutung einer Assoziation. Oft handelt es sich um ein Verb, z. B. (Kunde) erteilt (Bestellung).

Attribut (*attribute*) Attribute beschreiben Daten, die von den Objekten der Klasse angenommen werden können.

Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch im Allgemeinen unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten Typ und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muss jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig. Abgeleitete Attribute lassen sich aus anderen Attributen berechnen. In Programmiersprachen heißen Attribute in der Regel Variable. (Syn.: Variable)

Aufruf (*call*) Beschreibt den Wechsel der Kontrolle von der aufrufenden Stelle zu dem aufgerufenen Algorithmus und die Rückkehr hinter die Aufrufstelle nach Beendigung des aufgerufenen Algorithmus. Ein Aufruf erfolgt normalerweise durch Angabe des Algorithmusnamens und der aktuellen Parameter.

Auswahl (*selection*) Ausführung von Anweisungen in Abhängigkeit von Bedingungen. Man unterscheidet die einseitige, die zweiseitige und die Mehrfachauswahl. (Syn.: Verzweigung, Fallunterscheidung)

Automat (*automat*) Mathematisches Modell eines Systems oder Gerätes, das auf Ereignisse oder Eingaben mit Aktionen oder Ausgaben reagiert.

Backtracking (*Backtracking*) Suchverfahren, bei dem im Falle einer Sackgasse zum letzten Entscheidungspunkt zurückgegangen und die nächste Alternative untersucht wird.

Bedingungs/Ereignis-Netz

(*Condition/Event Net, C/E Net*) Petri-netz, bei dem jede Stelle eine oder keine Marke enthält. Enthalten alle Eingabestellen einer Transition eine Marke (Vorbedingungen) und alle Ausgabestellen keine Marke (Nachbedingung), dann kann die Transition schalten. Die Stellen werden also als Bedingungen interpretiert, von denen die Ausführung eines Ereignisses (Transition) abhängt. (Abk.: B/E-Netz)

Begrenzte Entscheidungstabelle

(*limited-entry decision table*) Enthält als Bedingungsanzeiger nur die Elemente (J, N, -) und als Aktionsanzeiger nur das Element (X).

Benutzer (*user*) Personen, die ein Computersystem unmittelbar einsetzen und selbst bedienen (siehe auch Anwender).

Best Practice (*Best Practice*) Bestes Verfahren oder besser Erfolgsrezept. Ein Unternehmen, das *Best Practices* einsetzt, verwendet bewährte Verfahren, Techniken und Geschäftsprozesse, die für andere Unternehmen Vorbildcharakter haben können. Der Begriff stammt aus der angloamerikanischen Betriebswirtschaft.

Bidirektionale Assoziation (*bidirectional association*) Assoziation, deren Objektbeziehungen (*links*) in beiden Richtungen durchlaufen werden können.

Binder (*linker*) Softwareentwicklungswerkzeug, das mehrere separat übersetzte Programmeinheiten zu einer ausführbaren Einheit zusammenfasst.

Bindung (*cohesion*) Die Bindung gibt an, wie stark die Elemente einer Systemkomponente interagieren.

Botschaft (*message*) Aufforderung eines Senders (Objekt oder Klasse) an einen Empfänger (Objekt oder Klasse), eine Dienstleistung zu erbringen. Der Empfänger interpretiert die Botschaft und führt eine Operation aus.

Bottom-up-Methode (*bottom up method*) Vorgehensweise, bei der mit der hierarchisch tiefsten Systemkomponente begonnen und anschließend zu den höheren Ebenen fortgeschritten wird.

BPEL (Business Process Execution Language) Die *Business Process Execution Language* ist ein verbreiteter Standard für die XML-basierte Modellierung von Geschäftsprozessen z.B. im Rahmen von service-orientierten Architekturen (SOA).

Brainstorming (*brainstorming*) Kreativitätstechnik, um durch Sammeln und wechselseitiges Assoziieren von spontanen, verbal vorgetragenen Einfällen von Mitarbeitern in einer Gruppensitzung die beste Lösung eines Problems zu finden.

Breite-zuerst-Strategie (*Breath first search*) Die Breite-zuerst-Strategie ist eine uninformierte Suche, bei der der Suchbaum zuerst in die Breite entwi-

ckelt wird. Kürzere Suchpfade werden vor längeren Pfaden weiter bearbeitet. (Syn.: Breitensuche)

BRMS (Business Rule Management System) Software, die die Entwicklung regelbasierter Software-Lösungen ermöglicht. Sie enthält neben der Regel-Maschine, Mechanismen zur Verwaltung von Wissensbasen, Regel-Editoren und eine Benutzungsoberfläche.

Computer-Aided Software Engineering Entwicklung von Software mit Hilfe von computergestützten Softwareentwicklungswerkzeugen (Abk.: CASE)

Constraint (*constraint*) Logische Einschränkung, Bedingung oder Zusage, die immer wahr sein muss. *Constraints* können in umgangssprachlicher oder maschinenlesbarer Form spezifiziert werden. Ihr Zweck ist es, die Semantik eines Elements genauer zu modellieren. Bestimmte *Constraints* sind in der UML schon vordefiniert, weitere können durch die Modellierer hinzugefügt werden. (Syn.: Einschränkung, Restriktion, Bedingung, Zusage)

Data Mart (*data mart*) Ein *Data Mart* stellt einen Extrakt aus dem unternehmensweiten *Data Warehouse* für einen bestimmten Organisationsbereich, eine abgegrenzte Personengruppe oder eine spezifische Anwendung dar und wird häufig physisch separat gespeichert, um so schnelle Antwortzeiten für das jeweilige Einsatzgebiet gewährleisten zu können.

Data Mining (*data mining*) Erkennen unbekannter Zusammenhänge, Nutzen bekannter Zusammenhänge für Zukunftsprognosen, Automatisierung von Analysen und Klassifizierungen im Rahmen von Data Warehouses.

Data Warehouse (*data warehouse*) Kopie operativer Daten, speziell für Anfragen und Analysen strukturiert. Die Datenorganisation erfolgt in Hyperwürfeln.

Debugger (*debugger*) Ein Softwareentwicklungswerkzeug, mit dessen Hilfe man das Programm – meist mit direktem Bezug zum Quelltext – schrittweise ausführen, anhalten und steuern sowie Programmzustände abfragen und manipulieren kann.

Glossar

Deckungsbeitrag (*contribution margin*) Ein Deckungsbeitrag ist der Anteil aus den Erlösen von Verkäufen, der die proportionalen Kosten übersteigt und somit einen Beitrag zur Deckung der fixen Kosten leistet.

DLOC (Delivered Lines of Code) Anzahl der ausgelieferten produktiven Codezeilen. Dabei wird nur der Code gezählt, der beim Kunden zum Einsatz kommt. Testcode zählt somit nicht dazu.

DTD (*Document Type Definition*) Definiert den strukturellen Aufbau einer Klasse von XML-Dokumenten. Mit Hilfe von DTDs kann die strukturelle Korrektheit von XML-Dokumenten automatisch geprüft werden.

Durchführbarkeitsstudie (*feasibility study*) Studie zur Erarbeitung einer Empfehlung, ob ein geplantes Softwareprodukt nach Prüfung der fachlichen, personellen und ökonomischen Durchführbarkeit realisiert werden soll.

Dynamische Programmanalyse (*dynamic analysis*) Eine Analyse eines Programms durch Beobachtung der Ausführung des Programms. (Syn.: Dynamische Analyse)

Dynamische Testverfahren (*dynamic testing methods*) Führen ein ausführbares, zu testendes Programm auf einem Computersystem aus.

EBNF (Extended Backus-Naur-Form) Formalismus zur Beschreibung der Syntax von Programmiersprachen (Syntaxdiagramm).

Editor (*editor*) essenzielles Softwareentwicklungswerkzeug, mit dessen Hilfe der Entwickler sein Programm verfasst

Eintreffer-Entscheidungstabelle (*single-hit decision table*) Zu einem Zeitpunkt ist höchstens eine der vorhandenen Regeln anwendbar. Ist eine gültige Regel gefunden, dann ist die Auswertung beendet.

Element (*element*) Informationseinheit in einem XML-Dokument. Ein Element besteht aus der Anfangsmarkierung, dem Elementinhalt und der Endmarkierung. Die Anfangsmarkierung kann zusätzlich XML-Attribute enthalten. Ein Sonderfall sind leere Elemente, die keinen Inhalt besitzen.

Empathie (*empathy*) Empathie (Einfühlungsvermögen) ist die Fähigkeit, sich auf Bedürfnisse und Stimmungen anderer Personen in besonderem Maße einstimmen zu können.

Entity-Relationship-Modell (*ER-model*) Datenmodellierung durch Identifikation von Entitätstypen (mit ihren Attributen) und Beziehungstypen sowie Beschreibung der Kardinalitäten. (Abk.: ER-Modell)

Entscheidungsbaum (*decision tree*) Grafische Darstellung einer vertikalen Entscheidungstabelle, bei der alle Alternativen und Aktionen explizit ausformuliert werden.

Entscheidungstabelle (*decision table*) Erlaubt die Beschreibung auszuführender Aktionen in Abhängigkeit von durch »und« verknüpften Bedingungen. Man unterscheidet Eintreffer- und Mehrtreffer-Entscheidungs-Tabellen, begrenzte und erweiterte Entscheidungstabellen sowie vollständige Entscheidungstabellen. Entscheidungstabellen können horizontal, vertikal oder als Entscheidungsbaum dargestellt werden. Mehrere Entscheidungstabellen können zu einem Entscheidungstabelle-Verbund zusammengefasst werden.

Entscheidungstabellen-Verbund (*decision table combination*) Verknüpfung mehrerer Entscheidungstabellen mittels Sequenz, Verzweigung, Schleife oder Schachtelung, um eine problemadäquate Beschreibung zu ermöglichen.

Ereignis (*event*) Tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer. Es kann sein: Eine wahr werdende Bedingung, ein Signal, eine Botschaft (Aufruf einer Operation), eine verstrichene Zeitspanne oder das Eintreten eines bestimmten Zeitpunkts. In den beiden letzten Fällen spricht man von zeitlichen Ereignissen.

Erweiterte Entscheidungstabelle (*augmented decision table*) Bedingungsanzeiger und Aktionsanzeiger enthalten beliebigen Text.

Essenzielles Softwareentwicklungswerkzeug (*essential software tool*) Softwareentwicklungswerkzeuge, die für die Softwareentwicklung unabdingbar sind. Dazu zählen: Editor, Übersetzer und Binder oder Interpreter.

Existenzquantifizierung (*existential quantification*) Für eine Variable wird definiert, dass es (mindestens) eine Wertebelegung für x gibt, so dass die Aussage p gilt: $\exists x p(x)$.

Expertenschätzung (*expert judgment*) Eine Schätzung, die ein Experte nur aufgrund seiner Erfahrung abgibt. Es wird daher keine spezielle Methodik oder Datenbank benötigt.

Expertensystem (*expert system*) Ein Expertensystem ist ein meist ein regelbasiertes System, welches auf der Basis von formalisiertem menschlichen Wissen, ähnlich einem Experten, agieren kann.

Faktenbasis (*fact base*) Eine Faktenbasis enthält die Fakten, auf deren Grundlage die Regeln auf Anwendbarkeit geprüft werden. Die Faktenbasis kann sich nach Ausführung einer Regel ändern.

Fremdschlüssel (*foreign key*) Ein Attributwert in einem Datensatz eines relationalen Datenbanksystems, der in einer anderen Datenbanktabelle Wert eines Primärschlüssels ist.

Function-Points-Methode (*Function Points Analysis*) Algorithmische Methode zum Schätzen des Umfangs einer Software auf Grund der Anforderungen. Der Umfang wird in *Function Points* angegeben.

Funktion (*function*) Transformiert Eingabedaten in Ausgabedaten und/oder verändert interne Zustände; beschreibt eine Tätigkeit.

Funktionale Äquivalenzklassenbildung (*functional equivalence (class) partitioning*) Funktionales Testverfahren, das Testdaten aus gebildeten Äquivalenzklassen der Ein- und Ausgabebereiche der Programme ableitet. Eine Äquivalenzklasse ist eine Menge von Werten, die nach der funktionalen Spezifikation des Programms vom Programm wahrscheinlich gleichartig behandelt werden. Es werden gültige und ungültige Äquivalenzklassen unterschieden.

Funktionale Bindung (*functional cohesion*) Alle Elemente einer Methode tragen dazu bei, eine einzige, in sich abgeschlossene Aufgabe zu erledigen.

Funktionale Testverfahren (*functional testing*) Dynamische Testverfahren, bei denen die Testfälle aus der funktionalen Spezifikation des Testlings abgeleitet werden. Beispiele sind die funktionale Äquivalenzklassenbildung und die Grenzwertanalyse

Funktionsbaum (*function tree*) Mit Hilfe eines Funktionsbaums wird eine hierarchische Zerlegung von Funktionen in detailliertere Funktionen grafisch dargestellt.

Geheimnisprinzip (*information hiding*) Die Interna einer Systemkomponente bzw. eines Subsystems sind von außerhalb nicht sichtbar, d.h. nur explizit über Schnittstellen bereitgestellte Informationen sind sichtbar.

Generierung (*generation*) Eine automatische Transformation eines Artefakts einer höheren Abstraktionsebene in ein Artefakt einer niederen Abstraktionsebene; zum Beispiel die Code-Generierung aus einem UML-Diagramm heraus.

Generischer Typ (*generic type*) Steht als Typ-Stellvertreter – auch Typparameter oder Typvariable bezeichnet – in einer Klassen-, Schnittstellen- oder Methoden- bzw. Operationsdeklaration. Wird bei der Anwendung durch einen konkreten Typ (aktuelles Typ-Argument) ersetzt. Generische Typen können geschachtelt und eingeschränkt werden (Typeinschränkung). (Syn.: parametrisierter Typ, Typparameter, Typvariable)

Geschäftsprozess (*business process*) Ein Geschäftsprozess (oft abgekürzt nur »Prozess« genannt) ist eine Abfolge von Funktionen (auch als →Aktivitäten bezeichnet) zur Erfüllung einer betrieblichen Aufgabe, wobei eine Leistung in Form von Informations- und/oder Materialtransformation erbracht wird. (Syn.: Prozess, Ablauf)

Geschäftsregel (*business rule*) Eine Geschäftsregel ist eine Regel, die einen wirtschaftlichen Sachverhalt beschreibt.

Glossar (*glossary*) Definiert und erläutert Begriffe, um eine einheitliche Terminologie sicherzustellen.

Grenzwertanalyse (*boundary value analysis*) Funktionales Testverfahren und fehlerorientiertes Testverfahren, da es auf einer konkreten Fehlererwartungshaltung basiert. Die Testfälle werden in der Regel so gewählt, dass sie auf den Randbereichen von Äquivalenzklassen liegen (funktionale Äquivalenzklassenbildung).

Harel-Automat (*Harel statechart*) Zustandsautomat mit geschachtelten Zuständen (hierarchischer Zustandsautomat), bedingten Zustandsübergängen, Aktionen (Mealy-Automat) und Aktivitäten (Moore-Automat), Zuständen mit Gedächtnis und Nebenläufigkeit (*concurrency*).

Heuristische Suche (*heuristic search*) Die heuristische Suche nutzt zum Aufbau des Suchbaums Informationen über das Problem wie Abstandsmaße zum Ziel oder Kosten der Regelanwendung. (Syn.: Informierte Suche)

Hierarchie (*hierarchy*) Eine Hierarchie ist eine Struktur, deren Systemkomponenten in einer Rangfolge entsprechend festgelegten Regeln angeordnet sind.

Hierarchischer Zustandsautomat (*hierarchical finite state machine*) Erlaubt eine Strukturierung von Mealy-Automaten durch eine Baumhierarchie. Die Hierarchieebenen werden geschachtelt dargestellt.

Hierarchisches Petrinetz (*hierarchical Petri net*) Stellen und Transitionen eines Petrinetzes können sowohl verfeinert als auch zu einer höheren Abstraktionsebene zusammengefasst werden (Kanal-Instanzen-Netz).

Hyperwürfel (*hyper cube*) Speicherung von Daten in mehrdimensionalen Strukturen. Jede Zelle innerhalb eines Würfels wird durch die Elemente aller Dimensionen bestimmt und kann direkt angesprochen werden (Data Warehouse).

Implikation (*implication*) Die Implikation ($A \rightarrow B$) ist eine logische Formel, die als wenn A dann B gelesen wird.

Inside-out-Methode (*inside out method*) Vorgehensweise, bei der zunächst die Interna eines Systems betrachtet

und modelliert werden und erst anschließend die Umwelt bzw. der Kontext des Systems.

Integrierte Entwicklungsumgebung (*Integrated Development Environment*) Eine Plattform, die eine Reihe von Softwareentwicklungswerkzeugen in einer einheitlichen Benutzungsoberfläche integriert. Im Englischen wird hierfür der Begriff *Integrated Development Environment* verwendet, dessen Abkürzung IDE auch im Deutschen gebräuchlich ist. (Abk.: IDE)

Interpreter (*interpreter*) Softwareentwicklungswerkzeug, das für interpretierte Sprachen die Aufgaben des Übersetzers und Binders zur Laufzeit übernimmt. Der Interpreter implementiert eine virtuelle Maschine, die die Programmstrukturen zur Laufzeit analysiert und ausführt, ohne dass die Instruktionen in den Maschinencode des Prozessors übersetzt werden, auf dem das Programm ausgeführt wird.

Kanal-Instanzen-Netz (*channel agency net*) Obere Netzebenen bei hierarchischen Petrinetzen. Eine Stelle wird als Kanal, eine Transition als Instanz interpretiert.

Klasse (*class*) Definiert für eine Kollektion von Objekten deren Struktur (Attribute), Verhalten (Operationen, Methoden) und Beziehungen (Assoziationen, Vererbungsstrukturen). Aus Klassen können – mit Ausnahme von abstrakten Klassen – neue Objekte erzeugt werden. Der Klassenname muss mindestens im Paket, besser im gesamten System eindeutig sein.

Klassendiagramm (*class diagram*) Stellt die objektorientierten Konzepte Klasse, Attribute, Operationen und Beziehungen (Vererbung, Assoziation) zwischen Klassen in grafischer Form dar (UML). Zusätzlich können Pakete modelliert werden.

Klausel (*clause*) Eine Klausel ist eine prädikatenlogische Formel, deren Variablen alle allquantifiziert sind. Die Teilformeln bestehen nur aus Literalen, die mittels ODER verknüpft sind.

Kommunikationsdiagramm (*communication diagram*) Ein Kommunikationsdiagramm in der UML beschreibt die Objekte und die Objektbe-

ziehungen zwischen diesen Objekten. An jede Objektbeziehung (*link*) kann eine Operation mit einem Pfeil angetragen werden. Die Reihenfolge und Verschachtelung der Operationen wird durch eine hierarchische Nummerierung angegeben.

Komponente (*component*) Binärer Softwarebaustein, der Dritten Funktionalität über Schnittstellen zur Verfügung stellt und unabhängig ausgeliefert werden kann. Benötigt eine Komponentenplattform und besitzt die Fähigkeit zur Selbstbeschreibung. Weist nur explizite Kontextabhängigkeiten auf und verfügt über die Fähigkeit zur Anpassung, um eine möglichst hohe Kompositionsfähigkeit zu erreichen.

Komposition (*composition*) Besondere Form der Aggregation. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch einem anderen Ganzen zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile.

Konfliktmenge (*conflict set*) Die Konfliktmenge ist die Menge der in einem Zustand anwendbaren Regeln. Mittels einer Konfliktlösungsstrategie wird dann eine Regelauswahl vorgenommen.

Kontrollstrukturen (*control structures*) Geben an, in welcher Reihenfolge (Sequenz) und ob (Auswahl) bzw. wie oft (Wiederholung) Anweisungen ausgeführt werden sollen (lineare Kontrollstrukturen), bzw. ob andere Programme aufgerufen werden (Aufruf).

Konzept (*concept*) Beschreibt einen definierten Sachverhalt unter einem oder mehreren Gesichtspunkten (Notation).

Konzeptionelles Modell (*conceptual model*) Integration aller Entity-Relationship-Diagramme (siehe Entity-Relationship-Modell) eines Systems zu einem Gesamtschema.

Kopplung (*coupling*) Die Kopplung gibt an, wie die Schnittstellen zwischen Systemkomponenten aussehen. Es werden der Kopplungsmechanismus, die Schnittstellenbreite und die Kommunikationsart betrachtet.

Kreativitätstechniken (*creativity methods*) Formalisierte Anwendung der heuristischen Prinzipien Assoziation, Strukturtransformation, Abstraktion, Kombination, Variation zur Erzeugung von Kreativität.

Lastenheft (*target specification, contract specification*) Fachliches Ergebnisdokument (Artefakt) der Planungsphase, auch grobes Pflichtenheft genannt; Teil einer Durchführbarkeitsstudie.

Lebendigkeit (*liveliness*) Stellt in einem Petrinetz sicher, dass ausgehend von einer Anfangsmarkierung die Transitionen immer so schalten, dass eine vorgegebene Transition im weiteren Verlauf nochmals schalten kann. (Abk.: liveliness)

Lineare Kontrollstrukturen (*linear control structures*) Besitzen genau einen Eingang und einen Ausgang; Sequenz, Auswahl, Wiederholung, Aufruf, strukturierte Programmierung

Literal (*literal*) Ein Literal ist eine atomare logische Formel oder deren Negation.

Lokalität (*locality*) Alle für einen Gesichtspunkt relevanten Informationen befinden sich räumlich zusammenhängend angeordnet auf einer oder wenigen Seiten. Für den Gesichtspunkt irrelevante Informationen sind *nicht* vorhanden.

Marke (*token*) Dient zur Darstellung der dynamischen Vorgänge in einem Petrinetz. Eine Marke repräsentiert ein Objekt, das durch die Transitionen weitergegeben wird. Eine Marke selbst wird durch einen kleinen schwarzen Kreis in einer Stelle dargestellt.

MDA (*Model Driven Architecture*; Modellgetriebene Architektur) Ein von der OMG (*Object Management Group*) standardisiertes Vorgehen, das mit den Ideen der modellgetriebenen Entwicklung plattformunabhängige und plattform-spezifische Entwicklung trennt.

Mealy-Automat (*Mealy automaton*) Zustandsautomat, bei dem die Ausgaben bzw. Aktionen an den Zustandsübergang gebunden sind.

Mehrfachvererbung (*multiple inheritance*) Jede Klasse kann mehr als eine direkte Oberklasse besitzen. Werden gleichnamige Attribute oder Opera-

Glossar

tionen von verschiedenen Oberklassen geerbt, dann muss der Namenskonflikt gelöst werden (siehe auch Einfachvererbung).

Mehltreffer-Entscheidungstabelle (*multi-hit decision table*) Zu einem Zeitpunkt können mehrere der vorhandenen Regeln anwendbar sein. Bei der Anwendung müssen alle Regeln überprüft werden.

Meta-Modell (*Meta Model*) Ein Modell, das die Spezifikationssprache von Modellen festlegt.

Meta-Information (*meta information*) Information über Artefakte selbst, wie zum Beispiel die Autorenschaft, die Änderungsgeschichte und Ähnliches.

Meta-Meta-Modell (*Meta Meta Model*) Ein Modell, das die Spezifikationssprache von Meta-Modellen definiert

Metaregel (*meta rule*) Eine Metaregel ist eine Regel, die die Regelauswahl steuert. Metaregeln definieren eine Konfliktlösungsstrategie.

Methode (*method*) Planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. (Syn.: Vorgehensweise)

Modellgetriebene Entwicklung (*Model-Driven Software Development*) Softwareentwicklung, bei der aus einer formalen Modellbeschreibung lauffähige Programme automatisiert generiert werden. (Abk.: MDSD; Syn.: Modellbasierte Entwicklung)

Modul im weiteren Sinne (*module*) Ein Modul i.w.S. ist eine weitgehend kontextunabhängige, prinzipiell austauschbare funktionale Einheit oder semantisch zusammengehörende Funktionsgruppe innerhalb eines Gesamtsystems. (Syn.: Modul i.w.s.)

Moore-Automat (*Moore state machine*) Zustandsautomat, bei dem die Ausgaben bzw. Aktionen an den Zustand gebunden sind.

Multiplizität (*multiplicity*) Die Multiplizität eines Attributs spezifiziert die Anzahl der Werte, die ein Attribut enthalten kann oder muss. Bei einer Assoziation gibt die Multiplizität am gegenüberliegenden Assoziationsende an, wie viele Objektbeziehungen von einem

Objekt zu den Objekten der assoziierten Klasse für diese eine Assoziation ausgehen können.

Muster (*pattern*) Beschreibt in abstrakter Form eine bewährte Lösung und setzt sie in Bezug zur Problemstellung und zur Systemumgebung. In der OO-Welt sind Muster Strukturen von Klassen bzw. Objekten. Man unterscheidet Analysemuster (OOA-Muster) und Entwurfsmuster.

Nassi-Shneiderman-Diagramm (*Nassi-Shneiderman diagram*) siehe Struktogramm-Notation.

Navigierbarkeit (*navigability*) Legt fest, ob eine Assoziation unidirektional oder bidirektional realisiert wird.

Oberklasse (*super class*) In einer Generalisierungsstruktur heißt jede Klasse, von der eine andere Klasse abgeleitet wird, Oberklasse dieser Klasse. Die Oberklasse vererbt ihre Eigenschaften und Verhalten an ihre Unterklassen.

Objekt (*object*) Besitzt einen Zustand (Attributwerte und Verbindungen zu anderen Objekten), reagiert mit einem definierten Verhalten (Operationen) auf seine Umgebung und besitzt eine Objektidentität, die es von allen anderen Objekten unterscheidet. Jedes Objekt ist Exemplar einer Klasse. (Syn.: Exemplar, Instanz)

Objektdiagramm (*object diagram*) Stellt in der UML Objekte und ihre Verbindungen (*links*) untereinander dar. Objektdiagramme werden im Allgemeinen verwendet, um einen Ausschnitt des Systems zu einem bestimmten Zeitpunkt zu modellieren (Momentaufnahme). Objekte können einen – im jeweiligen Objektdiagramm – eindeutigen Namen besitzen oder es können anonyme Objekte sein. In verschiedenen Objektdiagrammen kann der gleiche Name unterschiedliche Objekte kennzeichnen.

Objektorientierte Analyse (*object oriented analysis*) Ermittlung und Beschreibung der Anforderungen an ein Software-System mittels objektorientierter Konzepte und Notationen. Das Ergebnis ist ein OOA-Modell. (Abk.: OOA)

OLAP (*On-line Analytical Processing*) Auswertung von multidimensionalen Datenstrukturen. Wichtige Operationen sind slice & dice, roll up und drill down.

OMG (Object Management Group) Konsortium von über 800 Mitgliedern zur Schaffung von Industriestandards für objektorientierte Anwendungen.

OOA (*object oriented analysis*; Objektorientierte Analyse) Kurzform für Objektorientierte Analyse

OOA-Modell (*oo analysis model*) Fachliche Lösung des zu realisierenden Systems, die in einer objektorientierten Notation modelliert wird. Das OOA-Modell ist das wichtigste Ergebnis des *Requirements Engineering*.

OOA-Muster (*analysis pattern*) Eine Gruppe von Klassen mit feststehenden Verantwortlichkeiten und Interaktionen, die eine bestimmte – wiederkehrende – Problemlösung beschreiben (siehe auch Muster).

OOD-Modell (*oo design model*) Technische Lösung des zu realisierenden Systems, die in einer objektorientierten Notation modelliert wird. Das OOD-Modell basiert in der Regel auf einem OOA-Modell.

Outside-in-Methode (*outside in method*) Vorgehensweise, bei der zunächst die Umwelt bzw. der Kontext eines Systems betrachtet und modelliert wird und erst anschließend die Interna des Systems.

Paket (*package*) Hierarchischer Strukturierungsmechanismus, der es erlaubt Komponenten zu einer größeren Einheit zusammenzufassen. Ein Paket kann selbst Pakete enthalten. Eine Komponente kann z.B. ein Programm oder in der Objektorientierung eine Klasse sein. In der UML gruppiert ein Paket Modellelemente, z.B. Klassen. Pakete können in einem Paketdiagramm dargestellt werden. In der UML wird ein Paket als ein Rechteck mit einem Reiter dargestellt. In Java können Klassen und Schnittstellen zu Paketen zusammengefasst werden. (Syn.: Subsystem, subject, category)

Personenmonate (*person months*) Gibt bei der Aufwandsschätzung die Summe aller Arbeitsmonate aller betei-

ligten Personen in einem (Teil-) Projekt an. Ein Personenjahr wird dabei meist mit 10 Personenmonaten gleichgesetzt.

Petrinetz (*Petri net*) Petrinetze sind eine stark formalisierte Notation zur Modellierung und Untersuchung ereignisdiskreter Systeme mit Hilfe von Marken, die in sogenannten »Stellen« abgelegt werden und »Transitionen«, also Übergängen. Bei der Ausführung einer Transition werden Marken aus den vorangehenden Stellen entnommen und den folgenden Stellen hinzugefügt.

Pflichtenheft (*requirements specification, detailed specification*) Anforderungsdokument, das die Anforderungen an ein neues Produkt aus Auftraggeber- und Auftragnehmer-Sicht festlegt. Oft eine Detaillierung eines Lastenheftes. Meist nur verbal beschrieben.

Planungsphase (*planning phase*) Umfasst alle Aktivitäten, die nötig sind, um die fachliche, ökonomische und personelle Durchführbarkeit einer Produktentwicklung zu prüfen. Die Ergebnisse werden in einer Durchführbarkeitsstudie zusammengefasst. (Syn.: Voruntersuchung, Durchführbarkeitsuntersuchung)

Plattform (*platform*) Hardware-Architektur eines bestimmten Modells oder einer bestimmten Familie von Computersystemen, z.B. Windows-Plattform. Eine Plattform ist in der Regel eine Kombination aus Betriebssystem und Prozessortyp (Prozessor). Software wird in der Regel für eine Plattform entwickelt. (Syn.: Computer-Plattform, computer platform)

Plug-in (*plug-in*) Kleinste Softwareeinheit, die separat entwickelt, ausgeliefert und in eine Plattform integriert werden kann.

Prädikats/Transitions-Netz (*PrT-net*) Höheres Petrinetz, bei dem individuelle Marken (»gefärbte Marken«) verwendet werden. Die Pfeile werden mit Variablen versehen, die Transitionen mit Schaltbedingungen und Schaltwirkungen. (Abk.: Pr/T-Netz)

Primärschlüssel (*primary key*) Ein Primärschlüssel legt eine eindeutige Ordnung der Datensätze in einer Datenbanktabelle fest. Damit ist jeder Datensatz der Tabelle eindeutig identifizier-

Glossar

bar. In einer Tabelle kann nur höchstens ein Primärschlüssel angelegt werden. Ein Primärschlüssel besteht aus einer einzigen oder einer Kombination von mehreren Spalten der betreffenden Tabelle. (Syn.: Primärindex)

Prinzip (*principle*) Ein Prinzip ist ein Grundsatz, den man seinem Handeln zugrunde legt.

Produktionsregel (*production rule*) Eine Produktionsregel ist eine Regel, die im Ergebnis eine Aktion ausführt aber keine Änderung in der Faktenbasis bewirkt.

Profiler (*profiler*) Ein Softwareentwicklungswerkzeug, mit dessen Hilfe man das Laufzeitverhalten (Rechenzeit und Speicherbedarf) vermessen kann.

Programmablaufplan-Notation (*control flow notation*) In DIN 66001 genormte, grafische Darstellungsform für Kontrollstrukturen von Algorithmen. (Abk.: PAP; Syn.: Flussdiagramm-Notation)

Prolog (*Programming in Logic*) Eine Programmiersprache, die auf der Beschreibung eines Problems mittels Fakten und Regeln basiert. Sie kann als eine regelbasierte Software angesehen werden.

Prozessmodell (*process model*) Allgemeiner Entwicklungsplan, der das generelle Vorgehen beim Entwickeln eines Produkts festlegt. (Syn.: Vorgehensmodell)

Refactoring (*refactoring*) Eine Restrukturierung von Quelltext, um die Struktur zu verbessern, ohne dabei das Verhalten zu ändern.

Regel (*rule*) Bedingte Aussage bestehend aus zwei Teilen. Der erste Teil besteht aus einem oder mehreren Wenn-Ausdrücken und legt die Bedingungen fest, die eingehalten werden müssen, wenn der zweite Teil, bestehend aus einem oder mehreren Dann-Ausdrücken, eintreten soll.

Regel-Interpreter (*rule interpreter*) Die Software-Komponente, die die Verarbeitung der Regeln unter Nutzung der Fakten- und Regelbasis steuert. (Syn.: Regel-Maschine, Inferenzmaschine)

Regelauswahl (*rule selection*) Eine Regel ist notwendig, falls die Konfliktmenge mehrere Regeln enthält. Die Konflikt-

lösungsstrategie steuert die Regelauswahl. Die ausgewählte Regel wird dann ausgeführt.

Regelbasierte Software (*rule-based software*) Regelbasierte Software nutzt für die Lösung eines Problems eine formale Beschreibung des Wissens in wenn-dann-Form (Regeln).

Regelbasis (*rule base*) Eine Regelbasis enthält die Regeln der Problembeschreibung. Der Regel-Interpreter arbeitet auf dieser Menge von Regeln.

Regressionstest (*regression testing*) Wiederholung der bereits durchgeführten Tests nach Änderung des Programms. Er dient zur Überprüfung der korrekten Funktion eines Programms nach Modifikationen z. B. Fehlerkorrekturen.

Requirements Engineering (*requirements engineering*) Teilgebiet der Softwaretechnik; Aufgabe ist die systematische Ermittlung, Beschreibung, Modellierung und Analyse von Anforderungen im Dialog mit dem Auftraggeber unter Einsatz geeigneter Methoden und Werkzeuge. (Abk.: RE; Syn.: Systemanalyse)

Restrukturierung (*restructuring*) Eine (manuelle oder automatische) Transformation eines Artefakts auf derselben relativen Abstraktionsebene.

Rete-Algorithmus (*Rete algorithm*) Der Rete-Algorithmus ist ein effizienter Algorithmus zur Bestimmung anwendbarer Regeln bei regelbasierter Software. Er basiert auf einer Netz-Darstellung aller Regeln.

Reverse Engineering (*reverse engineering*) Eine (manuelle oder automatische) Transformation eines Artefakts von einer niederen in eine höhere Abstraktionsebene.

Rolle (*role*) Eine Rolle stellt eine Zusammenfassung von Aufgaben dar, die in einem bestimmten Kontext (z. B. bei der Durchführung eines Geschäftsprozesses) i.d.R. von einer Person durchgeführt werden. Rollen müssen Mitarbeitern nicht fest zugeordnet sein. Ein Mitarbeiter kann je nach Situation unterschiedliche Rollen einnehmen – auch mehrere Rollen gleichzeitig.

Rollenname (*role name*) Beschreibt, welche Bedeutung ein Objekt in einer Assoziation besitzt. Eine binäre Assoziation besitzt maximal zwei Rollen.

Rückwärtsverkettung (*backward chaining*) Die Rückwärtsverkettung interpretiert Regeln (wenn A dann B) in der Rückrichtung (B falls A). Die Verarbeitung beginnt immer mit einer Hypothese, die dann auf bekannte Fakten zurückgeführt (bewiesen) wird.

Schlüssel (*key*) Minimale Attribut-Kombination, um eine Entität in einer Entitätsmenge eindeutig zu identifizieren.

Schlussfolgerungsregeln (*inference rule*) Schlussfolgerungsregeln sind Regeln, die zu neuen Fakten führen und somit das Wissen über das Problem erweitern. (Syn.: Schlussregeln)

Schnittstelle (*interface*) In der UML besteht eine Schnittstelle nur aus Operationen, die keine Implementierung besitzen. Sie ist äquivalent zu einer Klasse, die keine Attribute, Zustände oder Assoziationen und ausschließlich abstrakte Operationen besitzt. Die Menge aller Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die Schnittstelle der Klasse bzw. des Objekts. Eine Schnittstelle kann in Java aus Konstanten und abstrakten Operationen bestehen. Schnittstellen können in Java dazu verwendet werden, eine ähnliche Struktur wie die Mehrfachvererbung zu realisieren.

Semantische Datenmodellierung (*semantic data modeling*) Erweiterung des Entity-Relationship-Modells (ER-Modells) um Vererbung (*is-a*) und Aggregation (*is-part-of*).

Sequenz (*sequence*) Mehrere Anweisungen werden hintereinander, von links nach rechts und von oben nach unten, ausgeführt (Aneinanderreihung). Die Anweisungen werden in der Regel durch ein Semikolon voneinander getrennt.

Sequenzdiagramm (*sequence diagram*) Grafische, zeitbasierte Darstellung mit vertikaler Zeitachse, Botschaften zwischen Objekten und Klassen. Botschaften werden durch horizonta-

le Linien, Objekte und Klassen durch gestrichelte, vertikale Linien repräsentiert.

Sicherheitsfaktor (*certainty factor*) Ein Sicherheitsfaktor ist eine Zahl aus dem Bereich $[-1..+1]$ und stellt eine Bewertung eines Fakts oder einer Regel dar. Es ist *keine* Angabe einer Wahrscheinlichkeit.

Skolemisierung (*skolemisation*) Skolemisierung ist ein Algorithmus zur Beseitigung von existenzquantifizierten Variablen aus einer logischen Formel. Wird bei der Umwandlung einer Formel in die Klausel-Form eingesetzt.

Software (*software*) Programme, zugehörige Informationen und notwendige Dokumentation, die es zusammengefasst erlauben, mit Hilfe eines Computersystems Aufgaben zu erledigen. (Abk.: SW)

Softwareentwicklungswerkzeug (*CASE Tool*) Programme, die die Entwicklung von Software unterstützen. (Syn.: Entwicklungswerkzeuge, Werkzeug)

Softwaresystem (*software system*) System, dessen Systemkomponenten und Systemelemente aus Software bestehen.

Softwaretechnik (*software engineering*) Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Herstellung, Anwendung und Pflege von umfangreichen Softwaresystemen. (Abk.: SWT, SE)

Spezialisierung (*specialization*) Von einer Spezialisierung spricht man, wenn zu einer Klasse ein oder mehrere Unterklassen gebildet werden, die alle Eigenschaften dieser Oberklasse aufweisen, und darüber hinaus noch zusätzliche spezielle Eigenschaften. Die Oberklasse ist dabei die Generalisierung der Unterklassen.

Stakeholder (*stakeholder*) Person oder Organisation, die eigene Interessen bei einer Softwareentwicklung oder beim späteren Einsatz vertritt. (Syn.: Akteure, Anspruchsberechtigte, Interessenvertreter)

Glossar

Statische Programmanalyse (*Static Analysis*) Eine Analyse eines Programms nur aufgrund des Quelltextes oder anderer statischer Systembeschreibungen, ohne es jedoch auszuführen. (Abk.: Statische Analyse)

Stelle (*place*) Ein Knotentyp in einem Petrinetz. Passive Systemkomponente, in der Informationen oder Material abgelegt werden können. Der aktuelle Belegungszustand einer Stelle wird durch eine Marke angegeben.

Stellen/Transitions-Netz (*Place/Transition Net, P/T Net*) Petrinetz, bei dem jede Stelle eine definierte Markenzapazität und jeder Pfeil ein Pfeilgewicht zugeordnet bekommt. Entsprechend den Gewichten entfernen Transitionen Marken von den Eingabestellen und verteilen sie auf die Ausgabestellen. (Abk.: S/T-Netz)

Stereotyp (*stereotype*) Erweiterungsmechanismus der UML. Führt ein neues Modellierungskonstrukt ein, das auf einer existierenden Klasse bzw. einem existierenden Typ des UML-Metamodells basiert und damit eine neue Untermetaklasse bzw. einen neuen Untertyp beschreibt. Ein Stereotyp gibt einem UML-Konstrukt eine besondere Semantik, darf jedoch dessen Struktur nicht verändern. Die UML enthält einige vordefinierte Stereotypen, erlaubt aber auch die Definition weiterer.

Struktogramm-Notation (*Nassi-Shneiderman diagram*) Grafische Darstellung linearer Kontrollstrukturen. (Syn.: Nassi-Shneiderman-Diagramm)

Struktur (*structure*) Eine Struktur ist eine reduzierte Darstellung eines Systems, die den Charakter des Ganzen offenbart unter Verzicht auf untergeordnete Details (siehe auch Abstraktion).

Strukturiertes Programmieren i.e.S. (*Structured Programming*) Beschreibung eines Algorithmus durch ausschließliche Verwendung von linearen Kontrollstrukturen (keine goto- bzw. Sprungkonstrukte).

Syntaxdiagramm (*syntax diagram*) Grafische Darstellung der Backus-Naur-Form (EBNF); üblich zur Beschreibung der Syntax von Programmiersprachen.

SysML (*Systems Modeling Language*) SysML ist eine universelle grafische Modellierungssprache zur Spezifikation, zur Analyse, zum Entwurf und zur Verifikation komplexer Systeme, die u.a. aus Hardware und Software bestehen können. Sie basiert auf der UML 2.

System (*system*) Ein System ist ein Ausschnitt aus der realen oder gedanklichen Welt, bestehend aus Systemkomponenten bzw. Subsystemen, die untereinander in verschiedenen Beziehungen stehen können.

Systemsoftware (*system software*) Software, die für eine spezielle Hardware oder Hardwarefamilie entwickelt ist, um den Betrieb und die Wartung dieser Hardware zu ermöglichen sowie ihre funktionellen Fähigkeiten zu ergänzen.

Tabelle (*table*) Tabellarische Darstellung eines Relationenschemas (Tabelnenkopf) mit Relationen (Tabelleninhalt) bei relationalen Datenbanken.

Tagged Value (*tagged value*) UML-Erweiterungsmechanismus, der eine Eigenschaft (*property*) eines UML-Elements repräsentiert. Ein *tagged value* besteht aus dem Schlüsselwort (*tag*), das die Elementeigenschaft benennt, und einem zugehörigen Datenwert (*value*) der Eigenschaft. Eine Eigenschaftsspezifikation besitzt die Form *tag* = *value*. (Syn.: Eigenschaftswerte)

Test spezieller Werte (*special values testing*) Testverfahren, die für die Eingabedaten selbst oder für von den Eingabedaten abhängige Aspekte bestimmte Eigenschaften fordern. Ziel dieser Testansätze ist es, aus der Erfahrung heraus fehlersensitive Testfälle aufzustellen. Die Grundidee ist, eine Liste möglicher Fehler oder Fehlersituationen aufzustellen und daraus Testfälle abzuleiten. Meist werden Spezialfälle zusammengestellt, die unter Umständen auch bei der Spezifikation übersehen wurden, z. B. Null.

Thread (*thread*) Logische Ausführungseinheit innerhalb eines Prozesses. Dient zur parallelen Ausführung von Code innerhalb eines Prozesses. Alle Threads eines Programms haben Zugriff auf die globalen Daten des Programms, was den Datenaustausch zwi-

schen den Threads sehr erleichtert. Jeder Thread hat zusätzlich einen eigenen Datensatz.

Tiefe-zuerst-Strategie (*depth first strategy*) Die Tiefe-zuerst-Strategie ist eine uninformierte Suche, bei der der Suchbaum zuerst in die Tiefe entwickelt wird. Längere Suchpfade werden vor kürzeren Pfaden weiter bearbeitet. (Syn.: Tiefensuche)

Timing-Diagramm (*timing diagram*) Das UML-Timing-Diagramm ist ein zweidimensionales Diagramm. Auf der x-Achse wird die Zeit, auf der y-Achse werden die Zustände von Objekten aufgetragen. Es erlaubt eine präzise zeitliche Spezifikation von dynamischen Abläufen. (Syn.: Zeitverlaufsdiagramm)

Top-down-Methode (*top down method*) Vorgehensweise, bei der mit der hierarchisch höchsten Systemkomponente begonnen und anschließend zu den tieferen Ebenen fortgeschritten wird.

Transformation (*transformation*) Eine (meist automatische) Überführung eines Artefakts in ein anderes.

Transition (*transition*) Eine Transition verbindet in Petrinetzen zwei oder mehrere Stellen miteinander. Transitionen schalten unter bestimmten Bedingungen, d. h. sie transportieren Marken zwischen Stellen.

Überschreiben (*overriding*) Von Überschreiben bzw. Redefinition spricht man, wenn eine Unterklasse eine geerbte Operation der Oberklasse – unter dem gleichen Namen – neu implementiert. Beim Überschreiben müssen die Anzahl und Typen der Ein-/Ausgabeparameter gleichbleiben. Bei der Implementierung der überschriebenen Operation wird im Allgemeinen die entsprechende Operation der Oberklasse aufgerufen. Ein Konstruktor sollte mindestens einen Konstruktor der Oberklasse aufrufen!

Übersetzer (*compiler*) Softwareentwicklungswerkzeug, das den Quelltext des Programms in eine ausführbare Form transformiert.

UML (*Unified Modeling Language*) Notation zur grafischen Darstellung objektorientierter Konzepte (objektorientierte Softwareentwicklung). Zur grafi-

schen Darstellung gehören unter anderem Klassendiagramme und Objektdiagramme. Die UML wurde von den »Methodenpäpsten« Booch, Rumbaugh und Jacobson bei der Rational Software Corporation entwickelt und 1997 von der OMG (*Object Management Group*) als Standard verabschiedet. Seitdem werden unter www.omg.org regelmäßig neue UML-Releases veröffentlicht.

UML-Profil (*uml profile*) Ergänzung des UML-Metamodells. Ein UML-Profil ist ein vordefinierter Satz von Stereotypen, *Tagged Values* und *Constraints*, die das UML-Modell für eine spezielle Umgebung, einen Anwendungsbereich oder einen Prozess anpassen. Profile sind erst seit der UML 2 im Metamodell verankert.

Unidirektionale Assoziation (*unidirectional association*) Assoziation, deren Objektbeziehungen (*links*) nur in einer Richtung durchlaufen werden können.

Unterklasse (*sub class*) Klasse, die in einer Generalisierungsstruktur Eigenschaften und Verhalten von einer anderen Klasse erbt. Man sagt auch: die Unterklasse spezialisiert ihre Oberklasse.

Unternehmensdatenmodell (*company data model*) Ganzheitliche Darstellung der Entitätstypen und ihrer Verknüpfungen aller Bereiche eines Unternehmens unter Berücksichtigung der Schnittstelle zueinander in einheitlicher Form.

URL (*uniform resource locator*; uniform resource locator) Im Web verwendete standardisierte Darstellung von Internetadressen; Aufbau: `protokoll://domain-Name/Dokumentpfad`.

Use Case (*use case*) Sequenz von zusammengehörenden Transaktionen, die von einem Akteur im Dialog mit einem System ausgeführt werden, um ein Ergebnis von messbarem Wert zu erstellen. Messbarer Wert bedeutet, dass die durchgeführte Aufgabe einen sichtbaren, quantifizierbaren und/oder qualifizierbaren Einfluss auf die Systemumgebung hat. Eine Transaktion ist eine Menge von Verarbeitungsschritten, von denen entweder alle oder keiner ausgeführt werden. (Syn.: Anwendungsfall)

Glossar

Use Case-Diagramm (*use case diagram*) UML-Diagramm, das die Beziehungen zwischen Akteuren und *Use Cases* in einem Softwaresystem modelliert. Auch Beziehungen zwischen *Use Cases* («extend» und include) können eingetragen werden.

Use Case-Schablone (*use case template*) Schablone zur semiformalen Spezifikation von *Use Cases*. Sie enthält folgende Informationen: Name, Ziel, Kategorie, Vorbedingung, Nachbedingung im Erfolgsfall, Nachbedingung bei einem Fehlschlag, Akteure, auslösendes Ereignis, Beschreibung des Standardfalls sowie Erweiterungen und Alternativen zum Standardfall.

Verbalisierung (*verbalization*) Gedanken und Vorstellungen in Worten ausdrücken und damit ins Bewusstsein bringen. In der Softwaretechnik bedeutet dies, aussagekräftige Namen und geeignete Kommentare zu wählen und selbstdokumentierende Konzepte, Methoden und Sprachen einzusetzen.

Vererbung (*inheritance*) Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse (Basisklasse) und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und Assoziationen der Basisklasse. Operationen der Basisklasse dürfen – kompatibel – redefiniert werden. Es entsteht eine Klassenhierarchie oder Vererbungsstruktur. Man unterscheidet die Einfachvererbung und die Mehrfachvererbung.

Verklemmung (*deadlock*) Situation in einem Petrinetz, in der keine Transition schalten kann, die aber bei einer anderen Schaltreihenfolge hätte vermieden werden können.

Vollständige Entscheidungstabelle (*complete decision table*) Alle möglichen Bedingungskombinationen sind als Regeln vorhanden (formal vollständig; 2^n , n =Anzahl der Bedingungen) bzw. alle fachlich möglichen Bedingungskombinationen sind vorhanden (inhaltlich vollständig).

Vorwärtsverkettung (*forward chaining*) Die Vorwärtsverkettung von Regeln (wenn A dann B) beginnt mit der

Menge der Fakten und gewinnt durch die Regelanwendung neue Fakten hinzu.

Wiederholung (*iterative construct*) Wiederholte Ausführung von Anweisungen in Abhängigkeit von einer Bedingung oder für eine gegebene Wiederholungszahl. Man unterscheidet Wiederholungen mit Abfrage der Wiederholungsbedingung vor jedem Wiederholungsdurchlauf, nach jedem Wiederholungsdurchlauf und Wiederholungen mit fester Wiederholungsanzahl. (Syn.: Schleife)

Wurzelelement (*root element*) Jedes XML-Dokument besitzt ein Wurzelelement. Das ist dasjenige Element, das alle anderen Elemente des XML-Dokuments enthält.

XMI (*XML Metadata Interchange*) Auf XML basierender Standard der OMG, der es ermöglicht Modelle (z.B. UML-Modelle) zwischen Software-Entwicklungswerkzeugen auszutauschen.

XML (eXtensible Markup Language)
1 Universell einsetzbare Sprache zum Austausch strukturierter Informationen. Basiert – wie die *Standard Generalized Markup Language* (SGML) – auf der Trennung von Inhalt und Struktur.

2 Eine Sprache (oder Meta-Sprache) zur Beschreibung der inhaltlichen Struktur von Dokumenten. XML ist ein W3C-Standard und in der Industrie inzwischen weit verbreitet.

XML-Attribut (*XML attribute*) Informationseinheit, die ein Element in einem XML-Dokument ergänzt. Attribute bestehen aus Name-Wert-Paaren und werden in die Anfangsmarkierung eines Elements eingetragen. Attributwerte sind Zeichenketten, die in Anführungszeichen stehen müssen.

XML-Schema (*XML schema*) Sprache, mit der sich die Struktur und die Semantik von XML-Dokumenten beschreiben lassen. XML-Schemata sind selbst XML-Dokumente und stellen eine Verbesserung gegenüber DTDs dar.

XP (eXtreme Programming) Agiles Prozessmodell, das auf vier Werten beruht und aus 15 Prinzipien besteht. Ein kleines Team entwickelt in einem Raum jeweils paarweise und iterativ Program-

me, wobei die Kommunikation informell ist und die Dokumentation sich auf den Code beschränkt.

Zeitbehaftetes Petrinetz (*timing Petri net*) Höheres Petrinetz, bei dem Stellen oder Transitionen mit einem deterministischen oder stochastischen Zeitverbrauch versehen werden können.

Zustand (*state*) 1 Der Zustand eines Objekts wird bestimmt durch seine Attributwerte und seine Verbindungen (*links*) zu anderen Objekten, die zu einem bestimmten Zeitpunkt existieren.

2 Interne Konfiguration eines Zustandsautomaten. Ein Zustand besteht solange, bis ein Ereignis eintritt, das einen Zustandsübergang auslöst.

Zustandsautomat (*finite state machine, finite automaton*) Besteht aus einer endlichen Anzahl interner Zustände. Zwischen den Zuständen gibt es Zustandsübergänge, die in Abhängigkeit von Eingaben oder Ereignissen durchgeführt werden. Eine Ausgabe oder Aktion kann beim Zustandsübergang erfolgen (Mealy-Automat) oder in einem Zustand (Moore-Automat). Hat einen

Anfangszustand und kann einen Endzustand besitzen. (Syn.: endlicher Automat, sequential machine)

Zustandsdiagramm (*state chart diagram*) Grafische Darstellungsform eines Zustandsautomaten. Die Zustände werden in der UML als Rechtecke mit abgerundeten Ecken dargestellt. Die Zustandsübergänge werden durch beschriftete Pfeile angegeben (Eingabe/Ausgabe).

Zustandsmatrix (*statechart matrix*) In Matrixform werden entweder die Zustände oder die Ereignisse oder der Ausgangszustand und der Zielzustand als Matrixdimensionen eines Zustandsautomaten aufgetragen.

Zustandstabelle (*statechart table*) In Spaltenform werden der aktuelle Zustand, das Ereignis, die Aktion und der Folgezustand eines Zustandsautomaten dargestellt.

Zustandsübergang (*transition*) Verbindet zwei Zustände. Kann nicht unterbrochen werden und wird stets durch ein Ereignis ausgelöst. Ein Zustandsübergang kann auch auf den eigenen Zustand erfolgen. (Syn.: Transition)

Literatur

[Albr79]

Albrecht, A. J.; *Measuring Application Development Productivity*, in: Proc. of the Joint SHARE, GUIDE, and IBM Application Development Symposium, 1979, S. 83–92.

[Allw05]

Allweyer, Thomas; *Geschäftsprozessmanagement – Strategie, Entwurf, Implementierung, Controlling*, 1. Auflage, Herdecke-Bochum, W3L-Verlag, 2005.

[Ambl08]

Ambler, Scott W.; *Beyond Functional Requirements On Agile Projects*, in: Dr. Dobb's Journal, October 2008, 2008, S. 64–66.

[ANSI83]

ANSI/IEEE Std. 729–1983, *IEEE Standard Glossary of Software Engineering Terminology*, New York, IEEE Inc., 1983.

[ASC07]

Azar, Jim; Smith, Randy K.; Cordes, David; *Value-Oriented Requirements Prioritization in a Small Development Organization*, in: IEEE Software, January/February 2007, 2007, S. 32–37.

[Bake72]

Baker, F. T.; *Chief programmer team management of production programming*, in: IBM Systems Journal, No.1, 1972, 1972, S. 56–73.

[Balz05]

Balzert, Heide; *Lehrbuch der Objektmodellierung – Analyse und Entwurf mit der UML 2*, 2. Auflage, Heidelberg, Spektrum Akademischer Verlag, 2005.

[Balz08]

Balzert, Helmut; *Lehrbuch der Softwaretechnik – Softwaremanagement*, 2. Auflage, Heidelberg, Spektrum Akademischer Verlag, 2008.

[BCH95]

Boehm, B. W.; Clark, B.; Horowitz, E.; *Cost Models for Future Software Lifecycle Processes*, in: Annals of Software Engineering, 1995.

[BeMa93]

Bertino, A.; Martino, L.; *Object-Oriented Database Systems – Concepts and Architectures*, Wokingham, Addison-Wesley, 1993.

[BFH+07]

Biskup, Hubert; Fischer, Thomas; Hesse, Wolfgang; Müller-Luschnat, Günther; Scheschonk, Gert; *Ein Begriffsnetz für die Software-Entwicklung*, in: Informatik-Spektrum, 30.3.2007, 2007, S. 217–224.

[BiRo90]

Bietham, J.; Rohrig, N.; *Datenmanagement*, in: Handbuch Wirtschaftsinformatik, Stuttgart, Poeschel-Verlag, 1990, S. 737–755.

[BJN+06]

Broy, Manfred; Jarke, Matthias; Nagl, Manfred; Rombach, Dieter; *Manifest: Strategische Bedeutung des Software Engineering in Deutschland*, in: Informatik-Spektrum, 29.3.2006, 2006, S. 210–220.

Literatur

- [BKK91]
Banker, R.; Kauffmann, R.; Kumar, R.; *An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment*, in: Journal of Management Information Systems, Vol. 8, No. 3, 1991, S. 127–150.
- [BICl08]
Blaine, J. David; Cleland-Huang, Jane; *Software Quality Requirements: How to Balance Competing Priorities*, in: IEEE Software, March/April 2008, 2008, S. 22–24.
- [Bloc05]
Bloch, Joshua; *Effective Java – Programming Language Guide*, 11. Auflage, Boston, Addison-Wesley, 2005.
Ausgezeichnetes Buch, das zeigt, bei welchen Java-Sprachkonstrukten welche Alternativen welche Vor- und Nachteile haben.
- [BLW05]
Baker, P.; Loh, P.; Weil, F.; *Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, in: LNCS (Lecture Notes in Computer Science), 3713, 2005, S. 476–491.
- [Boeh+00]
Boehm, B. W.; *Software Cost Estimation with COCOMO II*, Upper Saddle River, Prentice Hall, 2000.
Insgesamt neun Autoren.
- [Boeh00]
Boehm, B. W.; *COCOMO II Model Definition Manual – Version 2.1*, University of Southern California, 2000.
- [Boeh08]
Boehm, B. W.; *Making a Difference in the Software Century*, in: IEEE Computer, March, 2008, S. 32–38.
- [Boeh81]
Boehm B. W.; *Software Engineering Economics*, Englewood Cliffs, Prentice Hall, 1981.
- [BoHu03]
Boehm, Barry; Huang, Li Guo; *Value-Based Software Engineering: A Case Study*, in: Computer, March, 2003, S. 33–41.
- [Booc94]
Booch, Grady; *Object-Oriented Analysis and Design with Applications*, Second Edition, Redwood City, The Benjamin/Cummings Publishing Company, 1994.
- [Booc96]
Booch, G.; *Object Solutions – Managing the Object-Oriented Project*, Menlo Park, Addison-Wesley, 1996.
- [BoVa08]
Boehm, Barry W.; Valerdi, Ricardo; *Achievements and Challenges in Cocomo-Based Software Resource Estimation*, in: IEEE Software, September/October, 2008, S. 74–83.
- [BRJ99]
Booch, Grady; Rumbaugh, James; Jacobson, Ivar; *The Unified Modeling Language User Guide*, Reading, Addison-Wesley, 1999.
- [Broc08]
Der Brockhaus in 15 Bänden. Permanent aktualisierte Online-Auflage, Leipzig, Mannheim, F.A. Brockhaus, 2008, <http://www.xipolis.net/login/login.php>.

- [Broc88]
Brockhaus Enzyklopädie in 24 Bänden, Mannheim, F.A. Brockhaus, 1988.
- [Broo75]
 Brooks, F. P.; *The Mythical Man-Month*, Reading, Addison-Wesley, 1975.
 Eines der ersten Bücher, das sich mit Fragen des Softwaremanagements befasste.
- [Broy06]
 Broy, Manfred; *The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems*, in: Computer, October, 2006, S. 72–80.
- [Brya94]
 Bryan, G.E.; *Not all Programmers are created equal*, in: Proc. of Aerospace Applications Conference, 1994.
- [BSB08]
 Brommer, Christoph; Spindler, Markus; Barr, Volkert; *Softwarewartung – Grundlagen, Management und Wartungstechniken*, dpunkt-Verlag, 2008.
- [BuFo98]
 Bulos, Dan; Forsman, Sarah; *Getting Started with ADAPT – OLAP Database Design*, San Rafael, Symmetry, 1998, http://www.symcorp.com/downloads/ADAPT_white_paper.pdf.
 Abgerufen am 15.4.2009.
- [Chen76]
 Chen, P.; *The Entity-Relationship Model – Towards a Unified View of Data*, in: ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, 1976, S. 9–36.
 Originalartikel von Chen, der mit diesem Artikel das ER-Modell schuf.
- [Coa95]
 Coad, P.; mit North, D.; Mayfield, M.; *Object Models, Strategies, Patterns, and Applications*, Englewood Cliffs, Yourdon Press, Prentice Hall, 1995.
- [Cock03]
 Cockburn, Alistair; *Use Cases effektiv erstellen*, Heidelberg, Redline GmbH, 2003.
- [Cock97]
 Cockburn, A.; *Structuring Use Cases with Goals*, 1997, <http://members.aol.com/acockburn/papers/usecases.htm>.
- [Cohn04]
 Cohn, Mike; *User Stories Applied: For Agile Software Development*, Amsterdam, Addison-Wesley Longman, 2004.
- [CoYo91]
 Coad, P.; Yourdon, E.; *Object-Oriented Analysis*, 2. Auflage, Englewood Cliffs, Yourdon Press, Prentice Hall, 1991.
- [cz06]
UML-Modelle können Rahmen für Fahrzeugarchitektur bilden, in: Computer Zeitung, 23.10.06, 2006, S. 13.
- [Dahm00]
 Dahme, Christian; *Wissenschaftstheoretische Positionen in Bezug auf die Gestaltung von Software*, in: Wissenschaftsforschung 2000, Berlin, Gesellschaft für Wissenschaftsforschung, 2000, S. 167–177.
- [Dene08]
 Denert, Ernst; *Software-Engineering als Mittel zum Zweck*, in: Objekt-Spektrum, 6, 2008, S. 28–33.
- [DiHu91]
 Dillmann, R.; Huck, M.; *Informationsverarbeitung in der Robotik*, Berlin, Springer-Verlag, 1991.

Literatur

- [DIN EN28631]
Programmkonstrukte und Regeln für ihre Anwendung, Berlin, Beuth-Verlag, 1994.
- [DIN66001]
Sinnbilder und ihre Anwendung, Berlin, Beuth-Verlag, 1983.
- [DIN66241]
Entscheidungstabelle, Berlin, Beuth-Verlag, 1979.
- [DIN66261]
Sinnbilder für Struktogramme nach Nassi-Shneiderman, Berlin, Beuth-Verlag, 1985.
- [DIN69905]
DIN 69905 – Projektabwicklung – Begriffe, Berlin, Beuth Verlag GmbH, 1997.
- [DKJ05]
Dyba, Tore; Kitchenham, Barbara A.; Jorgensen, Magne; *Evidence-Based Software Engineering*, in: IEEE Software, January/February, 2005, S. 58–65.
- [Dold03]
Doldi, Laurent; *UML 2 Illustrated – Developing Real-Time & Communication Systems*, Toulouse, TMSO, 2003.
- [DoLe05]
Dombrowski, Eric; Lechtenböcker, Jens; *Evaluation objektorientierter Ansätze zur Data-Warehouse-Modellierung*, in: Datenbank-Spektrum, 15, 2005, S. 18–25.
- [Doug04]
Douglass, Bruce Powel; *Real Time UML: Advances in the UML for Real-Time Systems*, 3. Auflage, Boston, Pearson, 2004.
- [DPF05]
Denger, Christian; Paech, Barbara; Freimut, Bernd; *Achieving high quality of use-case-based requirements*, in: Informatik Forschung und Entwicklung, 20, 2005, S. 11–23.
- [Eber08]
Ebert, Christof; *Systematisches Requirements Engineering und Management*, Heidelberg, dpunkt.verlag, 2008.
- [Ehri01]
Hartmut Ehrig; Bernd Mahr; Felix Cornelius; Martin Groß-Rhode; Philip Zeitze; *Mathematisch-theoretische Grundlagen der Informatik*, Berlin, Springer-Verlag, 2001.
- [ElSc03]
van Elsuwe, Heiko; Schmedding, Doris; *Metriken für UML-Modelle*, in: Informatik Forschung und Entwicklung, 18, 2003, S. 22–31.
- [EmKo08]
Eman, Khaled El; Koru, A. Günes; *A Replicated Survey of IT Software Project Failures*, in: IEEE Software, September/October, 2008, S. 84–90.
- [Engl93]
Englisch, Joachim; *Ergonomie von Software-Produkten – Methodische Entwicklung von Evaluationsverfahren*, Mannheim, B.I.-Wissenschaftsverlag, 1993.
- [EnRo03]
Endres, Albert; Rombach, Dieter; *A Handbook of Software and Systems Engineering*, Amsterdam, Addison Wesley, 2003.
- [Erk02]
Katrin Erk; Lutz Priesse; *Theoretische Informatik*, Berlin, Springer-Verlag, 2002.

- [ESS08]
Engels, Gregor; Sauer, Stefan; Soltenborn, Christian; *Unternehmensweit verstehen – unternehmensweit entwickeln: Von der Modellierungssprache zur Softwareentwicklungsmethode*, in: Informatik-Spektrum, 31.5.2008, 2008, S. 451–459.
- [FHR08]
Fieber, Florian; Huhn, Michaela; Rumpe, Bernhard; *Modellqualität als Indikator für Softwarequalität: eine Taxonomie*, in: Informatik-Spektrum, 31.5.2008, 2008, S. 408–424.
- [FMP+87]
Firth, R.; Mosley, V.; Pethia, R.; Roberts, L.; Wood, W.; *A Guide to the Classification and Assessment of Software Engineering Tools*, Pittsburgh, Software Engineering Institute (SEI), 1987.
CMU/SEI-87-TR-10; ESD-TR-87-111.
- [FMS08]
Friedenthal, Sanford; Moore, Alan; Steiner, Rick; *OMG Systems Modeling Language (OMG SysML) Tutorial*, 2008, <http://www.omgsysml.org/INCOSE-2008-OMGSysML-Tutorial-Final-revb.pdf>.
Abgerufen am 8.4.2009.
- [Fowl05]
Martin Fowler; *Refactoring – Improving the Design of Existing Code*, 17. Auflage, Boston, Addison Wesley, 2005.
Ausgezeichnetes Buch, das zeigt, wie vorhandene Programme systematisch überarbeitet und verbessert werden können.
- [Fowl97a]
Fowler, M.; *UML Distilled – Applying the Standard Object Modeling Language*, Reading, Addison Wesley, 1997.
- [Fowl97b]
Fowler, M.; *Reusable Object Models*, Menlo Park, Addison Wesley, 1997.
- [Freg1879]
Frege, Gottlob; *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, in: G. Frege: Begriffsschrift und andere Aufsätze, Hildesheim u. a., Georg-Olms-Verlag, 2007.
6. Nachdruck der 2. Auflage von 1964.
- [Frem07]
Duden – Das Fremdwörterbuch, 9. Auflage, Hrsg. Duden-Redaktion, Mannheim, Bibliographisches Institut & F. A. Brockhaus AG und Duden Paetec GmbH, 2007.
auf CD-ROM.
- [GaGl98]
Gabriel, R.; Gluchowski, P.; *Grafische Notationen für die semantische Modellierung multidimensionaler Datenstrukturen in Management Support Systemen*, in: Wirtschaftsinformatik, Heft 6, 1998, S. 493–502.
- [GGP09]
Gabriel, Roland; Gluchowski, Peter; Pastwa, Alexander; *Datawarehouse & Data Mining*, Herdecke, Witten, W3L-Verlag, 2009.
- [GI08]
Was ist Softwaretechnik?, Gesellschaft für Informatik, Fachgruppe Softwaretechnik, 2008, http://pi.informatik.uni-siegen.de/gi/fg211/fg211_st_defs.html#teilgebiete.
Abgerufen am 4.10.2008.
- [Glin08]
Glinz, Martin; *A Risk-Based, Value-Oriented Approach to Quality Requirements*, in: IEEE Software, March/April, 2008, S. 34–41.

Literatur

- [GIWi07]
Glinz, Martin; Wieringa, Roel J.; *Stakeholders in Requirements Engineering*, in: IEEE Software, March/April, 2007, S. 18–20.
- [GRS03]
Görz, Günther; Rollinger, Claus-Rainer; *Handbuch der Künstlichen Intelligenz*, 4. Auflage, Hrsg. Schneeberger, Josef, München, Wien, Oldenbourg, 2003.
- [HaHe99]
Harren, Arne; Herden, Olaf; *MML und mUML – Sprache und Werkzeug zur Unterstützung des konzeptionellen Data Warehouse Design*, in: Proceedings 2.GI-Workshop Data Mining und Data Warehousing als Grundlage moderner entscheidungsunterstützender Systeme (DMDW99), Magdeburg, 1999, S. 57–68.
- [Hare87]
Harel, D.; *Statecharts: A Visual Formalism for Complex Systems*, in: Science of Computer Programming, 1987, 1987, S. 231–274.
Beschreibt hierarchische und nebenläufige Zustandsautomaten.
- [HBB+94]
Hesse, W.; Barkow, G.; Braun, H.; Kitthaus, H.-B.; Scheschonk, G.; *Terminologie der Softwaretechnik? Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 1: Begriffssystematik und Grundbegriffe*, in: Informatik-Spektrum, 17, 1994, S. 39–47.
- [HBB+94]
Hesse, W.; Barkow, G.; von Braun, H.; Kittlaus, H.-B.; Scheschonk, G.; *Terminologie der Softwaretechnik*, in: Informatik-Spektrum, Nr. 17, 1994, S. 96–105.
- [HBK+94]
Hesse, W.; Barkow, G.; Braun, H.; Kitthaus, H.-B.; Scheschonk, G.; *Terminologie der Softwaretechnik – Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 1: Begriffssystematik und Grundbegriffe*, in: Informatik-Spektrum, 17, 1994, S. 39–47.
- [Herd01]
Herden, Olaf; *Eine Entwurfsmethodik für Data Warehouses. Dissertation*, Oldenburg, Universität Oldenburg, 2001.
- [HKL+84]
Hesse, W.; Keutgen, H.; Luft, A. L.; Rombach, H.D.; *Ein Begriffssystem für die Softwaretechnik*, in: Informatik-Spektrum, Juli 1984, 1984, S. 200?213.
- [Hohn07]
Hohn, Bernhard; *Aktuelle Arbeitmarkchancen für IT-Fachleute*, in: Informatik-Spektrum, 30.3.2007, 2007, S. 211–217.
- [HoLe01]
Hofmann, Hubert F.; Lehner, Franz; *Requirements Engineering as a Success Factor in Software Projects*, in: IEEE Software, July/August, 2001, S. 58–66.
- [Holt99]
Holthuis, J.; *Der Aufbau von Data Warehouse-Systemen*, Wiesbaden, Deutscher Universitäts-Verlag, 1999.
- [HoPa02]
Houdek, F.; Paech, B.; *Das Türsteuergerät – eine Beispielspezifikation*, Kaiserslautern, Fraunhofer IESE, 2002.
Fraunhofer IESE-Report Nr. 002.02/D, 31. Januar, 2002.
- [HoU188]
Hopcroft, J. E.; Ullmann, J. D.; *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Bonn, Addison-Wesley, 1988.
Englische Originalausgabe: Introduction to automata theory, languages and computation, Reading: Addison-Wesley, 1979. Das Standardwerk zur Automatentheorie.

- [Hrus98]
Hruschka, P.; *Ein pragmatisches Vorgehensmodell für die UML*, in: Objekt Spektrum, 2/98, 1998, S. 34–54.
- [HRV 84]
Herzog, O.; Reisig, W.; Valk, R.; *Petri-Netze: ein Abriß ihrer Grundlagen und Anwendungen*, in: Informatik-Spektrum, Band 7, 1984, 1984, S. 20–27.
Kurzgefasster Überblick über B/E- und S/T-Netze einschließlich der Analysemöglichkeiten.
- [HSB07]
Heinsohn, J.; Socher-Ambrosius, R.; Boersch, I.; *Wissensverarbeitung*, Heidelberg, Berlin, Spektrum Akademischer Verlag, 2007.
- [Hump95]
Humphrey, W. S.; *A Discipline For Software Engineering*, Reading, Addison-Wesley, 1995.
- [Huth09]
Huth, Stefan; *Probleme und Fehler im Requirements-Engineering: Ergebnisse einer aktuellen Studie*, in: Objektspektrum, 1/2009, 2009, S. 11–13.
- [IBM97]
Developing Object-Oriented Software – An Experience-Based Approach, Hrsg. IBM Object-Oriented Technology Center, New Jersey, Prentice Hall, 1997.
- [IEEE1233]
IEEE Guide for Developing System Requirements Specifications – Description, Hrsg. IEEE, New York, IEEE, Inc., 1998.
IEEE Std 1233–1998.
- [IEEE1362]
IEEE Guide for Information Technology – System Definition – Concept of Operations (ConOps) Document, Hrsg. IEEE, New York, IEEE, Inc., 1998.
IEEE Std 1362–1998.
- [IEEE830]
IEEE Standards Board; *IEEE recommended practice for software requirements specifications*, IEEE Press, 1998.
- [IEEE90]
IEEE Standard Glossary of Software Engineering Terminology – IEEE Std 610.12–1990, IEEE, 1990.
- [InLu83]
Inhetveen, R.; Luft, A. L.; *Abstraktion, Idealisierung und Modellierung bei der Spezifikation, Konstruktion und Verifikation von Software-Systemen*, in: Angewandte Informatik, 12, 1983, S. 541–548.
- [Inmo96]
Inmon, W. H.; *Building the Data Warehouse*, 2. Auflage, New York et al., John Wiley & Sons, 1996.
- [IREB07]
Lehrplan IREB Certified Professional for Requirements Engineering – Foundation Level Version: 1.3, Hrsg. International Requirements Engineering Board, 2007, http://certified-re.de/de/files/cpre-fl-lehrplan_20080110.pdf.
Abgerufen am 2.4.2009.
- [Jack76]
Jackson, M.A.; *Data Structure as a Basis for Program Design*, in: Structured Programming – Infotech State of the Art Report, Maidenhead/Berkshire, Infotech International, 1976.

Literatur

- [JaMa02]
Jarke, Matthias; Mayr, Heinrich C.; *Mediengestütztes Anforderungsmanagement*, in: Informatik-Spektrum, 16. Dezember, 2002, S. 452–464.
- [JCJ+92]
Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G.; *Object-Oriented Software Engineering – A Use Case Driven Approach*, Wokingham, Addison Wesley, 1992.
- [JiLu06]
Jirotka, Marina; Luff, Paul; *Supporting Requirements with Video-Based Analysis*, in: IEEE Software, May/June, 2006, S. 42–44.
- [Jone07]
Jones, C.; *Estimating Software Costs: Bringing Realism to Estimating*, 2nd Edition, McGraw-Hill, 2007.
- [Jone95]
Jones, C.; *Backfiring: Converting Lines of Code to Function Points*, in: IEEE Computer, Vol. 28, No. 11, 1995, S. 87–88.
- [JRH+04]
Jeckle M.; Rupp C.; Hahn J.; Zengler B.; Queins S.; *UML 2 glasklar*, München, Hanser Verlag, 2004.
- [KaRy97]
Karlsson, K.; Ryan, K.; *A Cost-Value Approach for Prioritizing Requirements*, in: IEEE Software, Vol. 14, No. 5, 1997, S. 67–74.
- [Keme93]
Kemerer, C. F.; *Reliability of Function Points Measurement: A Field Experiment*, in: Communications of the ACM, Vol. 36, No. 2, 1993, S. 85–97.
- [KMS05]
Kowallik, Petra; Müller-Prove, Matthias; Strauß, Friedrich; *Requirements-Engineering im Spannungsfeld von Individual- und Produktsoftware*, in: i-com, Nr. 3, 2005, S. 41–46.
- [KoHä05]
Kovse, Jernej; Härder, Theo; *Domänenspezifische Modellierungssprachen*, in: IT focus, 11/12, 2005, S. 18–21.
- [Kope76]
Kopetz, H.; *Software-Zuverlässigkeit*, München-Wien, 1976.
- [Kram07]
Kramer, Jeff; *Is Abstraction the Key to Computing?*, in: Communications of the ACM, April, 2007, S. 36–42.
- [Kras05]
Krasemann, Hartmut; *Makro-Schätzen von Projekten*, in: Objekt-Spektrum, 2005, S. 66–73.
- [KTS+84]
Kano, N.; Tsuij, S.; Seraku, N.; Takahashi, F.; *Attractive Quality and Must-be-Quality*, in: Quality – The Journal of the Japanese Society for quality Control, Vol. 14, No. 2, 1984, S. 39–44.
- [LäCl08]
Lämmel, Uwe; Cleve, Jürgen; *Künstliche Intelligenz*, 3. Auflage, Leipzig, Hanser-Verlag, 2008.
- [Lang02]
Benutzergeschichte, Hrsg. Lange, Manfred, 2002, <http://www.xpexchange.net/german/intro/userStory.html>.
Abgerufen am 10.11.2008.

- [Luft84]
Luft, A. L.; *Zur Bedeutung von Modellen und Modellierungs-Schritten in der Softwaretechnik*, in: Angewandte Informatik, 5, 1984, S. 189–196.
- [LWE01]
Lawrence, Brian; Wiegers, Karl; Ebert, Christof; *The Top Risks of Requirements Engineering*, in: IEEE Software, Vol. 18, No. 6, Nov/Dec, 2001, S. 62–63.
- [MaLi03]
DeMarco, Tom; Lister, Timothy; *Bärentango – Mit Risikomanagement Projekte zum Erfolg führen*, München Wien, Carl Hanser Verlag, 2003.
- [Marc78]
DeMarco, Tom; *Structured Analysis and System Specification*, Englewood Cliffs, Yourdon Press, 1978.
- [Marc98]
DeMarco, T.; *Der Termin – Ein Roman über das Projektmanagement*, München, Hanser-Verlag, 1998.
- [MCDy93]
McGregor, J.; Dyer, D.; *A Note on Inheritance and State Machine*, in: ACM SIGSOFT, Oct., 1993, S. 61–69.
- [MoDe08]
Mohagheghi, P.; Dehlen, V.; *Where is the Proof? – A Review of Experiences from Applying MDE in Industry. Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, in: Lecture Notes In Computer Science, 5095, 2008, S. 432–443.
- [Mott09]
Motte, Petra; *Moderieren, Präsentieren, Faszinieren*, Herdecke, Witten, W3L-Verlag, 2009.
- [NaMa01]
Naiburg, Eric J.; Maksimchuk, Robert A.; *UML for Database Design*, Boston, Addison-Wesley, 2001.
- [NaSh73]
Nassi, Ike; Shneiderman, Ben; *Flowchart Techniques for Structured Programming*, in: SIGPLAN, August, 1973, S. 12–26.
- [NuEa07]
Nui, Nan; Easterbrook, Steve; *So, You Think You Know Others' Goals? A Repertory Grid Study*, in: IEEE Software, March/April 2007, 2007, S. 53–61.
- [OBN+08]
Ozkaya, Ipek; Bass, Len; Nord, Robert L.; Sangwan, Raghvinder S.; *Making Practical Use of Quality Attribute Information*, in: IEEE Software, March/April, 2008, S. 25–33.
- [OCL06]
OMG; *Object Constraint Language, OMG Available Specification, Version 2.0*, 2006, <http://www.omg.org/docs/formal/06-05-01.pdf>.
- [Odel94]
Odell, J.; *Six different kinds of composition*, in: Journal of Object-Oriented Programming, January 1994, 1994, S. 10–15.
- [OMG09a]
Object Management Group; *OMG Unified Modeling Language (OMG UML), Superstructure, V2.2*, 2009, <http://www.omg.org/spec/UML/2.2/Superstructure>.
- [OMG09b]
Object Management Group; *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.2*, 2009, <http://www.omg.org/spec/UML/2.2/Infrastructure>.

Literatur

[OMR+92]

Oppermann, R.; Murchner, B.; Reiterer, H.; Koch, M.; *Software-ergonomische Evaluation – Der Leitfaden EVADIS II*, Berlin, Walter de Gruyter, 1992.

[Ott08]

Ott, Devamani; *Requirements Engineering Barometer – Schlussfolgerungen*, 2008, <http://www.omis.ch/up/13.pdf>.

Auszug aus der Masterarbeit an der FHS St. Gallen Hochschule für Angewandte Wissenschaften, abgerufen am 2.4.2009.

[ÖzMe06]

Öztürk, Muhammet; Meyn, Albert; *Function-Point-Messungen und standardisierte Aufwandsschätzverfahren*, in: Objekt-Spektrum, 5, 2006, S. 76–82.

[Paec08]

Paech, Barbara; *What Is a Requirements Engineer?*, in: IEEE Software, July/August, 2008, S. 16–17.

[Park57]

Parkinson, G.N.; *Parkinson's Law and Other Studies in Administration*, Boston, Houghton-Mifflin, 1957.

[Parn71]

Parnas, D. L.; *Information Distribution Aspects of Design Methodology*, in: Proceedings Information Processing 71, Amsterdam, North-Holland, 1971, S. 339–344.

[PoBo05]

Poensgen, Benjamin; Bock, Bertram; *Die Function-Point-Analyse: Ein Praxis-handbuch*, dpunkt-Verlag, 2005.

[Pohl07]

Pohl, Klaus; *Requirements Engineering*, Heidelberg, dpunkt.verlag, 2007.

[Prec01]

Prechelt, Lutz; *Kontrollierte Experimente in der Softwaretechnik – Potenzial und Methodik*, Berlin, Springer, 2001.

[Prec99]

Prechelt, Lutz; *Vorlesung "Ausgewählte Kapitel der Softwaretechnik"*, 1999, <http://page.mi.fu-berlin.de/prechelt/swt2/node24.html>.
Abgerufen am 26.3.2009.

[Reis85]

Reisig, W.; *Systementwurf mit Netzen*, Springer-Verlag, 1985.

[RuNo03]

Russell, Stuart; Norvig, Peter; *Artificial Intelligence*, New Jersey, Prentice, 2003.

[Rupp07]

Rupp, Chris; *Requirements-Engineering und -Management*, 4. Auflage, München, Hanser, 2007.

[ScGr06]

Schacher, M.; Grässler, P.; *Agile Unternehmen durch Business Rules*, Berlin, Heidelberg, Springer, 2006.

[Sche90a]

Scheer, A.-W.; *Wirtschaftsinformatik – Informationssysteme im Industriebetrieb*, 3. Auflage, Berlin, Springer-Verlag, 1990.

Enthält die systematische Entwicklung eines Unternehmensdatenmodells für einen Industriebetrieb. Auf einem DIN A1-Faltblatt ist das vollständige ER-Modell dargestellt.

[Sche90b]

Scheer, A.-W.; *Unternehmensdatenmodell*, in: Lexikon der Wirtschaftsinformatik, Berlin, Springer-Verlag, 1990, S. 438–440.

- [Schm06]
Schmidt, J.; *Social Software: Onlinegestütztes Informations-, Identitäts- und Beziehungsmanagement*, in: Forschungsjournal Neue Soziale Bewegungen, Nr. 2, 2006.
- [Schö00]
Uwe Schöning; *Logik für Informatiker*, Heidelberg, Spektrum Akademischer Verlag, 2000.
- [Schr05]
Schröder, Marion; *Heureka, ich hab's gefunden – Methoden für die kreative Problemlösung, Ideenfindung & Alternativenauswahl*, Herdecke, W3L-Verlag, 2005.
- [ScWi98]
Schneider, Geri; Winters, Jason P.; *Applying Use Cases – A Practical Guide*, Reading, Addison Wesley Longman, 1998.
- [Simo62]
Simon, H. A.; *The Architecture of Complexity*, in: Proceedings of the American Philosophical Society, Vol. 106, 1962, S. 467–482.
- [Siro08]
Sirotnin, Victor; *UML vs. DSL: Stärken, Schwächen, Unterschiede und Mischformen*, in: Objekt-Spektrum, 6/2008, 2008.
- [SLM07]
Straube, R.; Leuschner, H. D.; Müller, P.; *Konfliktmanagement für Projektleiter*, Freiburg, Haufe-Verlag, 2007.
- [Snee03]
Sneed, Harry M.; *Aufwandsschätzung von Software-Reengineering-Projekten*, in: Wirtschaftsinformatik, 45 (2003) 6, 2003, S. 599–610.
- [Snee87]
Sneed, H. M.; *Software-Management*, Köln, Müller GmbH, 1987.
- [Snee97]
Sneed, H. M.; *Measuring the Performance of a Software Maintenance Department*, in: Proc. of Conference on Software Maintenance and Reengineering, 1997, S. 119–128.
- [Stan94]
Chaos Report, Hrsg. Standish Group, 1994.
- [Star07a]
Starke, Gernot; *Regelbasierte Systeme*, in: Java-Spektrum, 6/2007, 2007, S. 36–38.
- [SVE+07]
Stahl, Thomas; Völter, Markus; Efftinge, Sven; Haase, Arno; *Modellgetriebene Softwareentwicklung – Techniken, Engineering, Management*, 2. Auflage, Heidelberg, Dpunkt-Verlag, 2007.
- [ToKe04]
Tolvanen, Juha-Pekka; Kelly, Steven; *Domänenspezifische Modellierung*, in: Objektspektrum, Nr. 4, 2004, S. 30–35.
- [Tolv08]
Tolvanen, Juha-Pekka; *Domänenspezifische Modellierung in der Praxis*, in: Objekt-Spektrum, Nr. 4, 2008, S. 38–40.
- [UML07]
OMG; *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, 2007, <http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF>.
OMG Document Number: formal/2007-11-04.
- [UmMe06]
Umbach, Hartmut; Metz, Pierre; *Use Cases vs. Geschäftsprozesse*, in: Informatik-Spektrum, 29.6.2006, 2006, S. 424–432.

Literatur

- [V-Modell XT 06]
V-Modell XT – Version 1.2.0, Hrsg. Bundesrepublik Deutschland, 2006, <http://www.v-MODELL-XT.de>.
- [VDI2519]
Lasten-/Pflichtenheft für den Einsatz von Förder- und Lagersystemen, in: VDI-Handbuch Materialfluss und Fördertechnik, Band 8, Berlin, Beuth Verlag GmbH, 2001.
Obwohl diese VDI-Richtlinie nicht für die Entwicklung von Softwaresystemen vorgesehen ist, definiert sie die Begriffe Lasten- und Pflichtenheft auch für die Softwareentwicklung treffend.
- [VPAL08]
V-Modell XT Produktvorlage Anforderungen (Lastenheft), 2008, <http://www.v-modell-xt.de/>.
- [VPGP08]
V-Modell XT Produktvorlage Gesamtsystemspezifikation (Pflichtenheft), 2008, <http://www.v-modell-xt.de/>.
- [VXTLG08]
V-Modell XT Produktvorlage Lastenheft Gesamtprojekt, 2008, <http://www.v-modell-xt.de/>.
- [WaK199]
Warmer, J.; Kleppe, A.; *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.
- [Wate08]
Waters, Kelly; *Introducing User Stories*, <http://groups.google.com/group/allaboutagile/files>.
- [Wate08b]
Waters, Kelly; *Writing Good User Stories*, 2008, <http://www.agile-software-development.com/2008/04/writing-good-user-stories.html>.
Abgerufen am 10.11.2008.
- [Weil06]
Weilliens, Tim; *Requirements-Engineering mit der UML*, in: Objekt-Spektrum, 1, 2006, S. 42–44.
- [Wein71]
Weinberg, G. U.; *The Psychology of Computer Programming*, New York, Van Nostrand Reinhold Company, 1971.
- [Wend93]
Wendt, S.; *Defizite im Software Engineering*, in: Informatik-Spektrum, Februar, 1993, S. 34–38.
- [Wieg99]
Wiegers, K. E.; *Software Requirements*, Redmond, Microsoft Press, 1999.
- [Wiek99]
Wieken, J.-H.; *Der Weg zum Data Warehouse*, München, Addison-Wesley, 1999.
- [Wies98]
Wiese, J.; *Ein Entscheidungsmodell für die Auswahl von Standardanwendungssoftware am Beispiel von Warenwirtschaftssystemen*, Münster, Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster, 1998.
Arbeitsbericht Nr. 62: Arbeitsberichte des Instituts für Wirtschaftsinformatik.
- [WiMc02]
Wirfs-Brock, Rebecca; McKean, Alan; *Object Design. Roles, Responsibilities and Collaborations.*, Addison-Wesley, 2002.

- [WMH07]
Woolridge, Richard W.; McManus, Denise J.; Hale, Joanne E.; *Stakeholder Risk Assessment: An Outcome-Based Approach*, in: IEEE Software, March/April, 2007, S. 36–45.
- [WSH07]
Wentzel, Paul-Roux; Seckinger, Otmar; Hindel, Bernd; *Requirements-Engineering aus der Sicht der Persönlichkeitsanalyse*, in: Objekt-Spektrum, 5/2007, 2007, S. 34–41.
- [WWW90]
Wirfs-Brock, R.; Wilkerson, B.; Wiener, L.; *Designing Object-Oriented Software*, Englewood Cliffs, Prentice Hall, 1990.
- [Yin03]
Yin, Robert K.; *Case Study Research. Design and Methods – Applied Social Research Methods Series, Vol. 5*, 3. Auflage, Thousand Oaks, London, New Delhi, Sage Publications, 2003.
- [Your89]
Yourdon, E.; *Modern Structured Analysis*, Englewood Cliffs, Prentice-Hall, 1989.
- [ZBD+06]
Zaha, JM.; Barros A.; Dumas M.; ter Hofstede, A.; *A Language for Service Behavior Modeling*, in: Proceedings 14th International Conference on Cooperative Information Systems (CoopIS 2006), Berlin, Heidelberg, Springer, 2006.
- [Zies04]
Ziesche, Peter; *Nebenläufige & verteilte Programmierung – Konzepte, UML 2-Modellierung, Realisierung mit Java 1.4 & Java 5*, Herdecke, W3L-Verlag, 2004.

Sachindex

A

- Abnahmekriterien 471, 490
 - Abnahmeszenario 473
 - Abnahmetest 498
 - Abnehmen 514, 588
 - Abstrakte Klasse **151**
 - Abstraktion **26**
 - Abstraktionsebenen 30
 - Ad-hoc-Anordnung 545
 - Aggregation **171**
 - Akteur **251**, 255
 - Aktion **237**
 - Aktionsknoten 237
 - Aktivität **236**, **275**
 - Aktivitätsdiagramm 228, **237**, 254, 317
 - Petrinetz 317
 - Algorithmische Schätzung 526
 - Altsysteme 508
 - Analogiemethode 523
 - Analysierbarkeit 470
 - Analysieren 513, 587
 - Änderbarkeit 470
 - Anforderung 475
 - Abnahmekriterien 471
 - abnehmen 514, 588
 - analysieren 513, 587
 - ANSI/IEEE Std 830-1998 485
 - Attribut 479
 - Attributierungsschema 480
 - ermitteln 503
 - INVEST 500
 - Kano-Priorisierung 543
 - Kriterien 475
 - modellieren 547
 - natürlichsprachlich 481
 - nichtfunktional 463
 - Priorisierung 543
 - Priorisierungsmethoden 545
 - Schablone 485, 487, 492
 - spezifizieren 503
 - Textschablone 499
 - User Story 497
 - validieren 514
 - verifizieren 587
- Anforderungsschablonen 482, 485, 492
 - ANSI/IEEE Std 830-1998 485
 - natürlichsprachlich 482
 - V-Modell XT 487
 - Anforderungsspezifikation 510
 - Angemessenheit 468
 - Anpassbarkeit 470
 - ANSI/IEEE Std 830-1998 486
 - Anwender **5**
 - Anwendungsmodell 547
 - Anwendungssoftware **5**
 - Application Points 535
 - Applikationssoftware 5
 - Artefakt **62**, **443**
 - Audioaufzeichnungen 509
 - Interviewprotokolle 509
 - Notizen 509
 - Produktspezifikation 510
 - Spezifikation 510
 - Videos 509
 - Artefakt-Schablone **443**
 - Assoziation **158**
 - bidirektionale 165
 - höherwertige 163
 - n-äre 163
 - Name der 159
 - reflexive 161
 - ternäre 163
 - unidirektionale 165
 - Assoziationsklasse **162**
 - Assoziationsname **159**
 - Attraktivität 469
 - Attribut **181**
 - XML 187
 - XML-Schema 197
 - Attributtyp
 - XML-Schema 198
 - Audioaufzeichnungen 509
 - Aufruf 235
 - Aufwand 515
 - Algorithmische Schätzung 526
 - Analogiemethode 523
 - Application Points 535
 - Bewertung 539
 - Bottom-up-Methode 524
 - COCOMO 536
 - Dreipunktverfahren 524
 - Expertenschätzung 523
 - Faktor 4 521
 - Faktoren 515
 - Faustregeln 526, 534
 - Forschung 541
 - Function-Point-Methode 527
 - Gründe für eine Schätzung 517
 - Object Points 535
 - PROBE 525
 - Prozentsatzmethode 525
 - Schätzmethoden 522
 - Schwierigkeiten 518
 - Wartung 541
 - Wideband-Delphi-Methode 524
- Aufwandsschätzmethoden 522
 - Aussagenlogik 358
 - Austauschbarkeit 470
 - Auswahl **229**
 - ein- und zweiseitig 229
 - Automat **269**
- ## B
- Backtracking* **417**
 - Basisklasse 150
 - Basiskonzepte 100
 - Basissoftware 4
 - Basistechniken 23
 - Baugruppe 552
 - Bedienbarkeit 469
 - Bedingungs/Ereignis-Netz **305**
 - Befragungstechniken 507
 - Begrenzte
 - Entscheidungstabelle **399**
 - Benutzbarkeit 469
 - Benutzer **5**
 - Beobachtung 508
 - Bestensuche 425
 - Best Practice* **18**, **24**, **442**
 - Betriebsbedingungen 460
 - Beziehungsmanagement 74
 - Bidirektionale Assoziation **165**
 - Binder **60**
 - Bindung **37**
 - Bottom-up-Methode **53**, 524
 - Box
 - Aggregation 175
 - Aktivität 245
 - Assoziationen 166

Sachindex

Attribute 187
Klassen 137
Kommunikationsdiagramm 346
Komposition 175
Multiplizitäten 169
Pakete 148, 155
Petrinetz 322
Sequenzdiagramm 346
Use Case 262
Zustandsautomat 295
BPEL **199**, 253
BPMN 252
Brainstorming **298**, **506**
Breite-zuerst-Strategie **417**, **422**
Breitensuche 418, 422, 425
BRMS **410**, 412
C
Change Management 450
Checkliste
 Funktionale Bindung 129
 Funktionsname 129
 Java-Methodenname 129
 Schnittstellenname 134
COCOMO 536
code duplication 141
complexType (XML-Schema) 194
Computer-Aided Software Engineering **60**
Constraint **378**
control structures 227
D
Data Mart **216**
Data Mining **217**
Data Warehouse **216**
Debugger **62**
Deckungsbeitrag **518**
Diagnose 429
Disassembler 65
DLOC **526**
DNF 363
Dokumentanalyse 138
Dokumentieren 68
Dreipunktverfahren 524
DSL 563
DSM 563
DTD **192**
Dynamische Programmanalyse **61**, **67**, **69**

E
EBNF **181**, **185**
Eclipse 77
Editieren 63
Editor **60**
eEPK 252
Effizienz 469
Einfacher Typ (XML-Schema) 196
Einfaches Element
 XML 186
 XML-Schema 193
Eintreffer-Entscheidungstabelle **400**
Element **186**
 Häufigkeit (XML-Schema) 194
 XML 186
 XML-Schema 193
Element (XML)
 einfaches 186
 leeres 186
 strukturiertes 190
Elementname (XML) 186
Empathie **444**
Entity-Relationship-Modell **199**
Entscheidungsbaum **393**
Entscheidungstabelle **386**
Entscheidungstabellen-Verbund **395**
EPK 252, 253
Ereignis **291**
Erlernbarkeit 469
Ermittlungstechniken 507
Erweiterte
 Entscheidungstabelle **399**
Essenzielles
 Softwareentwicklungswerkzeug **60**
ET 386
 Anwendung 389
 Darstellungsformen 393
 Eintreffer 400
 Else-Regel 392
 Entscheidungsbaum 393
 Erstellung 387
 erweitert 399
 horizontal 393
 Konsolidierung 391
 Mehrtreffer 401
 Optimierung 391
 Überblick 402
 Überprüfung 391

 Verbund 395
 vollständig 391
Exemplar-Ebene 30
Exemplartyp 552
Existenzquantifizierung 406, **407**
Expertenschätzung **523**, **525**
Expertensystem **409**, 412, 427
Expertenverortung 74

F
Facette 197
Fachliche Lösung 547
Faktenbasis **409**, 419
Fallstudie
 Fensterheber 117, 575
 Seminarorganisation 107, 565
Fallunterscheidung 229
Fehlertoleranz 468
Feldbeobachtung 508
Fensterheber
 Fachliche Lösung 575
Flussdiagramme 228
Formale Logik 357
Fragebogen 508
Fremdschlüssel **205**
Function-Points-Methode **527**
Function Points 525
Funktion **127**
Funktionale
 Äquivalenzklassenbildung **473**
Funktionale Bindung **129**
Funktionale Testverfahren **473**
Funktionalität 127, 468
Funktionsbaum **143**

G
Geheimnisprinzip **42**
Genauigkeit 468
Generalisieren 150
Generierung **62**, **64**
Generischer Typ **31**
Geschäftsprozess 250, **251**
Geschäftsregel **428**
Glossar **107**, **117**, 482, **492**
 Struktur 482
Grenzwertanalyse **473**
Gruppe 558
Gruppenhistorie 558

H

Häufigkeit von Elementen
(XML-Schema) 194
Harel-Automat **278**
Heuristische Suche 417, 418,
425
Hierarchie **38**
Hierarchischer
Zustandsautomat **279**
Hierarchisches Petrinetz **312**
Hierarchisierung 38
Historie 557
Hyperwürfel **218**

I

IDE 76
Trends 78
Identitätsmanagement 74
Implikation **406**
Individualsoftware 510
Informationsmanagement 74
Inside-out-Methode **54**
Installierbarkeit 470
Integrierte
Entwicklungsumgebung **59**,
76
Interoperabilität 468
Interpreter **60**
Interviewprotokolle 509
ISO/IEC 9126-1 465, 468

J

Junktoren 359

K

Kanal-Instanzen-Netz **312**
Kann-Assoziation 158
Kano-Klassifikation 543
Kapselung 43
Kardinalitäten 200
Klasse **131**
abgeleitete 151
abstrakte 151
als Typ 152
Klassendiagramm **155**, **158**
Klausel **407**
KNF 363
Koexistenz 470
Kommentare 47
Kommunikationsdiagramm
343
Kommunikationssoftware 74
Komplexer Datentyp
(XML-Schema) 194

Komplexes Element
(XML-Schema) 194
Komplexitätsarten 6
Komponente **135**
Komposition **171**
Konfigurationsmanagement 67
Konfliktmenge **407**, **412**
Konkretisierung 28
Kontext 462
Kontrollstrukturen **227**
Konzept **18**, **56**, **100**
Klassifikation 104
Notationen 100
Konzeptionelles Modell **200**
Koordinator 554
Kopplung **37**
Kosten-Wert-Analyse 546
Krähenfuß-Notation 202
Kreativitätstechniken **506**

L

Lösungsraum 437
Lastenheft **107**, **117**, 487, **492**,
510
Laufanweisung 234
Lebendigkeit **327**
Leeres Element
XML 186
XML-Schema 195
Lehrlingsrolle 508
Let's Dance 253
Lieferumfang 490
Liste 551
Literal **409**
Logik
Aussagen- 358
formale 357
Prädikaten- 367
Lokalität **45**

M

Marke **303**
Markov-Kette 292
MC-Notation 201
MDA 80, **85**
MDD 80
MDS 79
Mealy-Automat **275**
Mehrfachauswahl 231
Mehrfachvererbung **153**
Mehrtreffer-
Entscheidungstabelle
400
Messen 68
Meta-Ebene 30

Meta-Information **62**, **68**
Meta-Meta-Modell **83**
Meta-Modell **83**
Metaregel **408**
Methode **53**
Bottom-up 53
Inside-out 53
Outside-in 53
Top-down 53
Modellbildung 28
Modellgetriebene Entwicklung
59, **61**, **79**
Modellierungssprache
domänenspezifisch 563
universell 563
Moderatorrolle 507
Modul 40
Modularität 40
Modul im weiteren Sinne **41**
Moore-Automat **275**
Multiplizität **158**
Muss-Assoziation 158
Muster **550**

N

Nachvollziehen 67
Namensgebung 47
Nassi-Shneiderman-Diagramm
228
Navigierbarkeit **163**
Normalformen 363
Notationen 100
Notizen 509
n plus 1/2-Schleife 233
Numerische Notation 202

O

Oberklasse **150**
Object Constraint Language
380
Object Points 535
Objekt **132**
Objektdiagramm **159**
OCL 380
OLAP **216**
OMG **106**
OO-Grundkonzepte 549
OOA **47**, 549
OOA-Methode 548, 559
OOA-Modell **62**, 548
OOA-Muster **550**
Baugruppe 552
Exemplartyp 552
Gruppe 558
Gruppenhistorie 558

Sachindex

Historie 557
Kordinator 554
Liste 551
Rollen 555
Stückliste 553
Wechselnde Rollen 556

OOD-Modell **62**
OOP 549
Outside-in-Methode **54**

P

Paket **145**
 Schreibweise 148
PAP 228
Personenmonate **525, 537**
Petrinetz **303**
 Analyse 327
 B/E-Netz 305
 hierarchisch 312
 Kanal-Instanzen-Netz 312
 Klassifikation 330
 Marken 303
 Methode 322
 Pr/T-Netz 311
 S/T-Netz 309
 Schaltregel 304
 Simulation 327
 Stellen 303
 stochastisch 316
 Strukturelemente 320
 Transitionen 303
 zeitbehaftet 314
Pflichtenheft **107, 117, 487, 492, 510**
PIM 85
Planung 430
Plattform **4**
Plug-in **78**
Portabilität 470
Prädikatenlogik 367
Prädikats/Transitions-Netz **311**
Prüfen 71
Primärschlüssel **204**
Prinzip **25**
 der Abstraktion 28
 der Bindung 37
 der Hierarchisierung 38
 der Kopplung 37
 der Lokalität 45
 der Modularisierung 40
 der Strukturierung 35
 der Verbalisierung 47
 Geheimnis- 42
Prinzipien
 Abhängigkeiten zwischen 50

 Wirkungsbereich 25
Priorisierung 543
Priorisierungsmethoden 545
PROBE 525
Problem vs. Lösung 437
Produktionsregel **406, 414**
Produktlinie 509
Produktmodell 547
Produktvorlagen 487
Profil 103
Profiler **62, 69**
Program-Slicing 67
Programmablaufplan-Notation **228**
Projektfeld 506
Prolog **410, 417**
Prozentsatzmethode 525
Prozessmodell **75**
PSM 85

Q

Qualitätsmerkmale 465, 468
Qualitätsstufen 466
Qualitätszielbestimmung 466

R

Rückwärtsverkettung 414, 416, **416, 417**
Rahmenbedingungen 459, 461
Redefinition 152
Refactoring **65**
Regel 404, **406**, 412, 414, 419, 420, 429
 Auswahl einer 407
 Bestensuche 425
 bewertete 426
 Breite-zuerst 417
 Rückwärtsverkettung 414
 Tiefensuche 420
 Verkettung 414
 Vorwärtsverkettung 414
Regel-Interpreter **409**
Regelauswahl 407, **407**
Regelbasierte Software **409**
Regelbasis **409**
Regressionstest **471**
Reife 468
Requirements Engineering
 Anforderungen an
 Anforderungen 475
 Aufwand 440, 441
 Ausbildung 436
 Best Practices 442
 Empirie 439
 Rahmenbedingungen 459

Risiken 441
Systemkontext 461
Visionen 457
Ziele 457
Requirements Engineering **434**
Restrukturierung **62, 65**
Rete-Algorithmus **412**
Reverse Engineering **65**
Rolle **443, 555**
Rollenname **159**

S

Schätzen des Aufwands 515
Schätzmethoden 522
Schlüssel **204**
Schlussfolgerungsregeln **414**
Schnittstelle **132**
sd 335
Selbstaufschreibung 508
Selbstdokumentation 48
Semantische
 Datenmodellierung **200**
SemOrg
 Fachliche Lösung 565
 Glossar 109
 Lastenheft 107
 Pflichtenheft 110
Sequenz **229**
Sequenzdiagramm **333**
Sicherheit 468
Sicherheitsfaktor **427**
simpleType (XML-Schema) 196
Skolemisierung **407**
Software 3, **3**
 Charakteristika 9
 Komplexitätsarten 6
 Produkt 3
 System 3
 Veränderungen 11
 Werkzeuge 59
Software-Krise 9
Softwareentwicklung 5
 globale 73
Softwareentwicklungswerkzeug **59, 60**
Softwaresystem **4**
Softwaretechnik **17**
 evidenzbasiert 21
 wertbasiert 21
Soziale Software 74
Spezialisierung **152**
Spezifikation
 nichtfunktionale
 Anforderungen 466
Stückliste 553

Stabilität 470
Stakeholder **440, 455, 504**
 Standardsoftware 510
 Standardtyp
 XML-Schema 193
 Statik 127
 Statische Programmanalyse **67, 69**
 Stelle **303**
 Stellen/Transitions-Netz **309**
 Stereotyp **84, 104, 182, 206, 223**
 Struktogramm-Notation **228**
 Struktur **35, 127**
 statische 127
 Strukturiertes Element (XML) 190
 Strukturiertes Interview 507
 Strukturiertes Programmieren i.e.S. **227**
Subject 256
 Suchbaum 423
 Suchen 65
 Syntax
 Konstantendeklaration 185
 Variablendeklaration 185
 Syntaxdiagramm **181**
 SysML **101, 466**
 System **25**
 Systementwicklung 5
 Systemgrenze 462
 Systemkontext 462
 Systemsoftware **4**
 Systemumgebung 461

T

Tabelle **204**
Tagged Value **206, 223**
 TBD 107
 Technisches System 5
 Testbarkeit 470
 Test spezieller Werte **473**
 Testverfahren 473
Thread **236**
 Tiefe-zuerst-Strategie **417, 421**
 Tiefensuche 418, 420, 425
 Timing-Diagramm **352**
 Top-down-Methode **53**
 Transformation **64**
 Transformieren 64
 Transition 271, **303**
 Typ-Ebene 30

U

Übergang 271
 Überschreiben **152**
 Übersetzer **60**
 UML **31, 101, 228, 228**
 abstrakte Klasse 151
 Kommunikationsdiagramm 343
 Lollipop 134
 Navigierbarkeit 165
 OCL 380
 Profil 103
 Schnittstelle 133
 Stereotyp 104
 Strukturdiagramme 103
 Timing-Diagramm 353
 Vererbung 151
 Verhaltensdiagramme 103
 Zeitverlaufsdiagramm 353
 UML-Profil **103, 206**
 Unidirektionale Assoziation **165**
 Unterklasse **150**
 Unternehmensdatenmodell **209**
 URL **192**
Use Case **48, 250, 251, 255**
 extend 257
 Generalisierung 257
 include 257
Use Case-Diagramm **256**
Use Case-Schablone **260**
User Story 497

V

Validieren 514
 Varianten 67
 VBSE 21
 Verbalisierung **46**
 Verbinden 66
 Verbrauchsverhalten 469
 Vererbung **151**
 Verifizieren 587
 Verklemmung **328**
 Versionen 67
 Verständlichkeit 469
 Verzweigung 229
 Video 509
 Vollständige
 Entscheidungstabelle **391**
 Voreinstellung (XML-Schema) 196, 198
 Vorwärtsverkettung 414, 415, **415**

W

Wahrheitstabelle 362
 Wartbarkeit 470
 Wechselnde Rollen 556
 Werkzeuge
 Abdeckungsarten 62
 Anforderungen 89
 Auswahlkriterien 87
 behandelte Artefakte 62
 Capture&Replay 67
 CASE 61
 Eclipse 77
 Editoren 63
 essenzielle 60
 Evaluationsverfahren 90
 interprojektfähig 76
 Klassifikation 60
 Komplementarität 63
 methodenneutral 75
 methodenunterstützend 75
 multiprojektfähig 75
 Trends 78
 zum Dokumentieren 68
 zum Messen 68
 zum Nachvollziehen 67
 zum Prüfen 71
 zum Suchen 65
 zum Transformieren 64
 zum Verbinden 66
 zum Verfolgen und Überwachen 70
 zum Verwalten & Versionieren 67
 zur Kollaboration 73
 zur Kommunikation 73
 zur Spezifikation 64
 zur Visualisierung 66
 Wertigkeit 158
 Wideband-Delphi-Methode 524
 Wiederherstellbarkeit 468
 Wiederholung **231**
 Wiederverwendung 509
 Wissenschaftsdisziplin 2
 Wissensmanagement 74
 Wurzelement **191**

X

XMI **199**
 XML **186, 190**
 XML-Attribut 187, **187**
 XML-Schema **192**
 referenzieren 193
 XML-Schema vs. DTD 199
 XP **55**

Sachindex

Z

Zählschleife 234
Zählvariable 235
Zeitbehaftetes Petrinetz **316**
Zeitverhalten 469
Zielgruppen 459
Zustand **269**
 Übergang 271
 Pseudo- 280
 Transition 271

Zustandsübergang **272**
Zustandsautomat **269**
 Erstellung 270
 Harel 277
 hierarchisch 279
 hybrid 278
 Markov-Kette 292
 Mealy 275
 mit Endzuständen 274
 mit Gedächtnis 285

Moore 275
 nebenläufig 286
Notationen 272
Protokoll- 289
Verhaltens- 289
Wächter 278
Zustandsdiagramm **272**
Zustandsmatrix **272**
Zustandstabelle **272**
Zuverlässigkeit 468