ADORA-L – EINE MODELLIERUNGSSPRACHE ZUR SPEZIFIKATION VON SOFTWARE-ANFORDERUNGEN

DISSERTATION

DER WIRTSCHAFTSWISSENSCHAFTLICHEN
FAKULTÄT
DER UNIVERSITÄT ZÜRICH

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von
STEFAN JOOS
von
Deutschland

genehmigt auf Antrag von
PROF. DR. MARTIN GLINZ
PROF. DR. HELMUT SCHAUER

Mai 2000

Als ich überlegte, wieviel verschiedene Ansichten über die gleiche Sache es geben kann, deren jede einzelne ihren Verteidiger unter den Gelehrten findet, und wie doch nur eine einzige davon wahr sein kann, da stand es für mich fest: Alles, was lediglich wahrscheinlich ist, ist wahrscheinlich falsch.

Descartes (Abhandlung über die Methode)

Danksagung

Die vorliegende Dissertation ist während meiner Tätigkeit am Institut für Informatik der Universität Zürich entstanden. Ich danke an dieser Stelle allen Personen, die in irgendeiner Form zum Gelingen dieser Arbeit beigetragen haben.

Mein besonderer Dank gilt Herrn Prof. Dr. Martin Glinz für die äußerst sorgfältige und konstruktive Betreuung meiner Arbeit. Herrn Prof. Dr. Helmut Schauer danke ich für die freundliche Zusammenarbeit.

Bedanken möchte ich mich vor allem bei meinen Kolleginnen und Kollegen am Institut für die hilfreichen Diskussionen und die vielen guten Ratschläge. Besonders hervorheben möchte ich die produktive Zusammenarbeit mit Stefan Berner, Martin Arnold, Johannes Ryser, Reto Schmid sowie die zwischenmenschliche Unterstützung von Stefan Berner, Norbert E. Fuchs, Uta Schwertel, Rolf Schwitter und Anca Vaduva. Juliane Berner danke ich für die unermüdliche Hilfe beim Korrekturlesen.

Dankbar für alles, was sie für mich getan haben, bin ich meiner Mutter Rita und meinem Vater Dr. Josef Joos, die den Beginn und den Abschluß dieser Arbeit nicht mehr erleben durften.

Stuttgart, im April 2000

Stefan Joos

Zusammenfassung

Im Rahmen dieser Arbeit wird die Spezifikationssprache ADORA-L zur Beschreibung von Software-Anforderungen entwickelt. Dazu werden die Problematik der Dokumentation von Anforderungen analysiert und der Stand aktueller Lösungsansätze erörtert. Aus den Defiziten vorhandener Ansätze und grundlegenden Erfordernissen heutiger Software-Spezifikationen wird eine Sprache abgeleitet, die aufzeigt, wie Software-Anforderungen anschaulich und strukturiert dokumentiert und wie somit auch umfangreiche und komplexe Problemstellungen verständlich beschrieben werden können. Die grundlegende Idee der hier entwickelten Spezifikationssprache ADORA-L ist die gemeinsame Beschreibung von Daten, Verhalten und Funktionalität in einem einzigen integrierten Systemmodell. Die Modellierung erfolgt grundsätzlich auf Ebene abstrakter Objekte und nicht durch eine Klassenmodellierung. Die abstrakten Objekte vermeiden die Modellierungsanomalie bei der Dekomposition von Klassenmodellen und sind einfacher, anschaulicher und vor allem präziser einsetzbar als die häufig verwendeten Klassen. Zentraler Bestandteil von ADORA-L ist die Teil/Ganzes-Hierarchie. Diese strukturiert ein Systemmodell durch Objekte, die wiederum Komponenten übergeordneter Objekte sind. Alle Aspektbeschreibungen (wie beispielsweise Funktions-, Struktur- und Verhaltensbeschreibungen) orientieren sich an dieser primären Struktur, indem entsprechende Sachverhalte dort integriert dargestellt werden. Besonders hervorzuheben ist hierbei die hierarchische und integrierte Beschreibung von Verhalten, die im wesentlichen auf den Statecharts-Ansatz nach [Harel87] aufsetzt. Die Teil/Ganzes-Hierarchie in ADORA-L ist ein mächtiges Abstraktionsinstrument, welches sowohl von Detailstrukturen als auch von Detailverhalten abstrahiert. Struktur und Verhalten lassen sich so hierarchisch auf verschiedenen Abstraktionsebenen modellieren. Abstraktion stellt hierbei eine grundlegende Voraussetzung für die Bewältigung umfangreicher und komplexer Problemstellungen dar.

ADORA-L ist primär eine grafische Spezifikationssprache, d.h. die grundlegende Struktur wird mit Hilfe von Grafik, Detailbeschreibungen werden textuell beschrieben. Von besonderer Bedeutung ist eine variabel formale Darstellung in ADORA-L, die es erlaubt, Anforderungen angepaßt an vorhandene Risiken und Kosten entweder präzise und ausführlich oder grob und schnell zu modellieren.

Abstract

The scope of this work is the development of a specification language (ADORA-L) intended to describe software requirements and architecture in a single object-oriented framework. This work is motivated in two ways. First, by the severe weaknesses of existing methods in terms of system decomposition. Second, by general ideas about specifications like object-orientation and the usage of hierarchical models. The general goal is to get a comprehensive specification which describes requirements and architecture in an understandable, clear and structured way - even for large-scale specification. As already mentioned the basic idea of the specification language ADORA-L is to model the aspects of data, functionality and behaviour in a single hierarchical object framework. Modeling is based on objects (so called abstract objects) instead of classes. Thus, we resolve modeling anomalies that occur in class models. Additionally modeling with abstract objects is more easier, more understandable and more precise than modeling with classes. Whole-part-hierarchies are a key feature of ADORA-L. Systems are decomposed by objects, which are components of other first class objects with full object semantics. All aspect descriptions (like descriptions of behaviour, structure or functions) use this primary structure. All aspects are integrated and represented in this single integrated structure. Particularly the behaviour description is based on the statechart mechanism [Harel87] and therefore, it supports an integrated behaviour modeling.

To provide powerful abstraction mechanisms is crucial to manage and understand especially large-scale specifications. System decomposition through whole-part hierarchies has proven to be a convenient and powerful abstraction mechanism. It allows for the description of aspects like system structure or behaviour on different levels of abstraction. The usage of abstractions is a fundamental precondition to manage complex problem descriptions. Primary ADORA-L is a graphical language: A graphical notation is used to represent the basic structure of a system. Descriptions on a detailed level will be represented textually. Another key feature of ADORA-L is to model requirements with a variable degree of formalism. This enables the developer to adjust the description of requirements to cost and risk factors. So, its up to the developer/ to model different aspects or parts of the system with an arbitrary degree of detail.

Inhaltsverzeichnis

Kapitel 1

Einführung

Dieses Kapitel steckt den Rahmen der vorliegenden Arbeit ab. Zunächst werden die Bereiche Anforderungsspezifikation und Spezifikationssprachen als Kontext der Arbeit vorgestellt. Anschließend werden grundsätzliche Probleme bei der Anforderungsspezifikation genannt. Nach der Aufgabenstellung werden die Ergebnisse dieser Arbeit, nämlich die Entwicklung der Spezifikationssprache ADORA-L, skizziert. Das Kapitelende beinhaltet den weiteren Aufbau der Arbeit.

Um die Einführung möglichst überschaubar zu halten, werden viele der hier verwendeten Begriffe nicht oder nur kurz erläutert. Eine ausführliche Erläuterung findet sich in den nachfolgenden Kapiteln.

1.1 Motivation

In der heutigen Zeit findet sich Software in fast allen Lebensbereichen wieder. Nahezu alle Systeme in Wirtschaft und Industrie wären ohne sie nicht vorstellbar. Für die Zukunft ist darüber hinaus eine weitere Entwicklung hin zu hochgradig softwarebasierten Systemen zu erwarten. Durch diese Systeme ergeben sich neue Anwendungsgebiete, zudem sind sie in der Regel billiger und flexibler als entsprechende herkömmliche softwarefreie Systeme. Die Erstellung von Software auf der anderen Seite steckt noch in den Anfängen und wird nach wie vor nur ungenügend beherscht. Die entwickelte Software ist häufig fehlerhaft und deckt sich nicht mit den Bedürfnissen der Benutzer. Software-Projekte überschreiten häufig den gesetzten Termin- und Kostenrahmen oder sie werden vorzeitig abgebrochen. Besonders problematisch in Software-Projekten ist der *Spezifikationsprozeß*, d.h. das *Analysieren des Problembereichs* und das *Spezifizieren der Anforderungen* an das zu entwickelnde System.

Die Entwicklung von Software-Systemen ist als Auftragsleistung eines Anbieters gegenüber einem Kunden zu verstehen. Sie erfordert daher die Zusammenarbeit und Koordination vieler Personen. Daher müssen sich sowohl Kunden als auch Anbieter über Umfang und Inhalt der zu erstellenden Software verbindlich absprechen und diese *Anforderungen* an die Software im Sinne eines Produktvertrages niederlegen. Dieser Produktvertrag wird in der Informatik als *Anforderungsspezifikation* bezeichnet. Für die Entwicklung von Software ist die Anforde-

rungsspezifikation von zentraler Bedeutung: Sie ist sowohl Vorgabe für die Konzeption, die Architektur und die Implementierung der Software als auch die Referenz für deren Abnahme durch den Kunden. Für den *Spezifikationsprozeß* ist von großer Bedeutung, wie die Anforderungen in geeigneter Weise strukturiert und dokumentiert werden und welche sprachlichen Mittel hierfür zu verwenden sind. Die Konzeption und der Aufbau der *Spezifikationssprache* sind maßgeblich für die Verständlichkeit und die Prüfbarkeit der Anforderungsspezifikation verantwortlich. Eine schlecht gewählte *Spezifikationssprache* vermindert häufig die Qualität der *Anforderungsspezifikation*; dies hat eine hohe Fehleranfälligkeit der Software zur Folge.

Derzeit wird als Spezifikationssprache hauptsächlich die natürliche Sprache verwendet. Dennoch erscheint sie für die *Spezifikation von Anforderungen* ungeeignet. Natürliche Sprache ist hochgradig mißverständlich und benötigt ständige Rückkopplungsschritte, um Sachverhalte präziser zu klären. Im *Software-Entwicklungsprozeß* kann dies allein von der Menge der zu klärenden Anforderungen nicht mehr gewährleistet werden. Immer mehr kommen Spezifikationssprachen auf, die Anforderungsspezifikationen formal mathematisch oder durch konstruktive Modelle beschreiben. Diese Sprachen sind zwar gegenüber der natürlichen Sprache deutlich präziser und weniger leicht interpretierbar, jedoch weisen sie allesamt Schwächen bei der Spezifikation sehr umfangreicher und komplexer Problemstellungen auf.

1.2 Probleme

Die bei der Erstellung von Anforderungsspezifikationen gemachten Fehler, die *Spezifikationsfehler*, haben die unangenehme Eigenschaft, sich in nachfolgenden Entwicklungsaktivitäten fortzupflanzen. Konsequenzen sind Folgefehler in der

- Lösungskonzeption und in der Architektur der Software, die aufgrund von Vorgaben aus der Spezifikation entwickelt wurden
- Implementierung, welche die Entscheidungen der Lösungskonzeption umsetzen und realisieren

Spezifikationsfehler werden typischerweise spät aufgedeckt, meistens bei der Abnahme durch den Kunden oder gar erst beim Betrieb. Die meisten Spezifikationsfehler lassen sich nur unter Mithilfe des Kunden finden, da nur er zwischen gewollter oder unerwünschter Anforderung unterscheiden kann. Dem Kunden selber sind jedoch mangels Fachverständnis die Dokumente und die Tätigkeiten der Lösungskonzeption und der Implementierung fremd und unverständlich. Spezifikationsfehler sind wegen den auftretenden Folgefehlern und ihrer schweren Auffindbarkeit die teuersten Fehler innerhalb der Software-Entwicklung. Da sie zudem statistisch gesehen einen hohen Anteil an der Gesamtzahl der gemachten Fehler einnehmen [Davis93, S. 27f], wirken sie sich entsprechend signifikant auf die Gesamtkosten aus. Worin liegt aber der

Grund für die Fehleranfälligkeit von Anforderungsspezifikationen? Hierfür sind hauptsächlich zwei Aspekte verantwortlich:

- Die Software-Evolution (Eigenschaft von Software-Anforderungen, sich durch Umwelteinflüsse zu verändern) und
- das Kommunikationsloch zwischen Kunden und Entwicklern

Die Software-Evolution – Die Anforderungen an eine Software sind einer ständigen Evolution unterworfen. Von besonderer Bedeutung sind Veränderungen, die in der Software nachgeführt werden müssen, also bei der Entwicklung der Software oder beim Betrieb derselben. Ursache für die Evolution sind

- Veränderungen im Unternehmensumfeld. Die Problematik liegt in der Tatsache, daß in der Spezifikation eine zukünftige Situation beschrieben werden muß, nämlich die Situation zum Zeitpunkt der Einführung des zu entwickelnden Software-Systems [Macaulay96, S. 2f].
- Veränderungen, die sich im Laufe von Entwicklungsaktivitäten ergeben. Hierbei sind hauptsächlich Anforderungen betroffen, die sich nicht bzw. unter nicht akzeptablen Bedingungen (z. B. zu hohe Kosten, zu schlechte Leistungsdaten) umsetzen lassen.
- Veränderungen, die sich durch die Inbetriebnahme der Software ergeben. Durch die Inbetriebnahme können sich Betriebsabläufe und Aufgaben von Mitarbeitern ändern [Macaulay96, S. 3f]. Das neue Umfeld und die Benutzung des Systems ändern häufig die ursprünglichen Anforderungen [Lehman85].

Das Kommunikationsloch zwischen Kunde und Entwickler – Eine prinzipielle Schwierigkeit bei der Erstellung von Spezifikationen liegt in der Verständigung zwischen Kunde und Entwickler. Beide haben ein eigene Sicht auf den Problembereich, bewerten und gewichten Anforderungen unterschiedlich und haben in der Regel verschiedenes Vokabular und unterschiedlichen Fachsprachen.

- Was für den Kunden selbstverständlich wirkt und daher unter Umständen nicht oder nur nebenbei erwähnt wird, kann für den Entwickler essentiell sein. Umgekehrt werden für den Kunden sichtbare, aber für die Softwarelösung nebensächliche Sachverhalte (z. B. Benutzeroberflächen) überbetont und somit für den Entwickler zu stark gewichtet.
- Anforderungen werden aufgrund unterschiedlicher Begriffsvorstellungen unterschiedlich interpretiert und führen zu falscher, zu unvollständigen bzw. zu teuren oder ungewünschten Funktionalitäten.
- Verständigungsprobleme finden sich auch zwischen verschiedenen Kundenvertretern und innerhalb des Entwicklungsteams, da auch hier jeweils nicht von einer eindeutigen Begriffsbasis ausgegangen werden kann.

Solche Diskrepanzen in der Verständigung sind nur schwer aufzudecken, da durch die unterschiedliche Interpretation von keiner Seite her die Fehler offensichtlich sind. Deshalb sind in

solchen Fällen ein disziplinierter *Spezifikations- und Prüfprozeß* und vor allem eine aussagekräftige und verständliche *Spezifikationssprache* unbedingt notwendig. Eine solche Sprache vermindert durch eine definierte Begriffsbasis die Verständigungsprobleme zwischen beteiligten Personen und unterstützt zudem die Gliederung eines komplexen Problembereichs. Zudem unterstützt die Spezifikationssprache den Prüfprozeß durch geeignete Konsistenzprüfungen und durch eine anschauliche und übersichtliche Präsentation.

In den letzten Jahren wurde eine Vielzahl unterschiedlicher Spezifikationssprachen entwickelt, mit dem Ziel, den oben beschriebenen Anforderungen gerecht zu werden. Im wesentlichen werden hier *informale*, *formale* und *teilformale* Sprachansätze unterschieden:

- *Informale Ansätze* beschreiben Anforderungen textuell natürlichsprachlich. Informale Ansätze werden in der Praxis sehr häufig eingesetzt, da zum einen die Sprache allgemein beherrscht wird und zum anderen Spezifikationen verständlich wirken und einfach erstellt werden können. Solche *Textspezifikationen* sind jedoch sehr fehleranfällig und aufgrund schlechter Gliederungsmechanismen nicht für umfangreiche Spezifikationen geeignet.
- *Formale Ansätze* beschreiben Anforderungen durch ein mathematisches Kalkül mit präzise definierten Sprachkonstrukten. Formale Ansätze werden selten und nur für kleinere Problemstellungen verwendet. Formale Spezifikationen sind in der Regel in ihrer Erstellung zu aufwendig und häufig schwer verständlich.
- *Teilformale Ansätze* beschreiben Anforderungen konstruktiv durch Bildung von *Systemmodellen*. Die Modellierung von Systemen erfolgt durch Sprachkonstrukte, die nur zum Teil in ihrer Bedeutung definiert sind. Teilformale Ansätze sind im allgemeinen präziser und strukturierter als informale Ansätze und verständlicher und einfacher als formale Ansätze.

Das größte Potential liegt in Ansätzen der teilformalen Sprachklasse und hier im besonderen die *objektorientierten Modellierungsansätze* (siehe Kapitel 4.5 und 5.1). Diese Ansätze bilden den goldenen Mittelweg zwischen formalen und informalen Ansätzen, indem sie die jeweiligen Vorzüge sich zu eigen machen. Sowohl teilformale Spezifikationsvertreter als auch die Vertreter der anderen Sprachklassen weisen zum Teil jedoch erhebliche Schwächen auf und eignen sich daher nur eingeschränkt zur Spezifikation von Anforderungen (siehe Kapitel 4). Die Defizite liegen allgemein

- in fehlenden oder unausgereiften Mechanismen zur Strukturierung der Spezifikation. Für umfangreichere Probleme bedeutet dies einen Mangel an Verständlichkeit und Prüfbarkeit.
- im Unvermögen, Anforderungen in variabler Präzision und flexiblem Formalitätsgrad beschreiben zu können. Dies ist jedoch notwendig, um den Spezifikationsprozeß an Entwicklungskosten und Risiko anzupassen. Es muß einerseits möglich sein, risikoarme und beherrschbare Teile der Spezifikation grob und mit weniger Aufwand und andrerseits riskante Teile sehr präzise und formal zu beschreiben.

Hier stellt sich die Frage, welche Sprachkonzepte notwendig und sinnvoll sind, damit die oben aufgeführten Forderungen und Defizite überwunden werden können.

1.3 Aufgabenstellung

Zielsetzung dieser Arbeit ist es, das im vorigen Kapitel beschriebene Defizit vorhandener Spezifikationssprachen zu beheben. Hierzu soll eine *teilformale* und *objektorientierte Spezifikationssprache* konzipiert und entwickelt werden. Die *Teilformalität* und damit verbunden die Verwendung von konstruktiven Modellen erscheint derzeit das Mittel der Wahl, um Spezifikationsprobleme im «Großen» zu bewältigen (siehe Kapitel 1.2). Die *Objektorientierung* (siehe Kapitel 4.5 und 5.1) ist darüber hinaus ein mächtiges Paradigma, um Anforderungen intuitiv und realitätsnahe auf semantischer Ebene zu gliedern und zu strukturieren.

Bis Mitte der 90er Jahre wurden eine Vielzahl konzeptionell ähnlicher objektorientierter Ansätze entwickelt (Bsp.: [Coad91], [Booch94], [Rumaugh91], [Embley94]). Alle diese Ansätze orientierten sich im wesentlichen an der klassischen Datenmodellierung und modellieren auf Ebene der *Klassen*. Systemmodelle werden gebildet, indem *Fragmente* davon durch diverse Diagrammarten beschrieben werden (beispielsweise durch Klassendiagramme). Das Systemmodell selber ist nicht sichtbar, es ergibt sich aussschließlich indirekt durch *Überlappung* von Diagrammen. Diese *diagrammbasierte Modellierung* beschreibt ein Systemmodell nur punktuell ohne es in seiner Ganzheit darzustellen. Entsprechend schwierig ist es, solche Systemmodelle einer Prüfung zu unterziehen. Vollständigkeit und Widerspruchsfreiheit sind im speziellen zwei Eigenschaften, die nicht befriedigend geprüft werden können. Ursache für diese *diagrammbasierte Modellierung* ist jedoch nicht die Objektorientierung selber, sondern vielmehr eine unausgereifte Sprachkonzeption verbunden mit dem irrigen Glauben, daß Entwicklungswerkzeuge dieses Problem lösen könnten.

Bis heute hat sich diese Form der objektorientierten Modellierung kaum verändert. Konzeptionell entwickelten sich diese Sprachansätze kaum weiter. Jüngere Sprachansätze wie beispielsweise die UML [Booch98] oder die OML [Firesmith96] versuchen lediglich, durch spezielle Sprachkonstrukte die Sprache selber erweiterbar zu machen und dadurch ähnliche objektorientierte Sprachdialekte zu *emulieren*. Es steht zu erwarten, daß sich eine zukünftige Version der UML zu einem Quasi-*Sprachstandard* entwickelt und somit zahlreiche ähnliche Sprachen überflüssig macht.

Die hier zu entwickelnde Spezifikationssprache distanziert sich von der *punktuellen* und *diagrammbasierten* Modellierung. Ansatzpunkt ist eine Präzisierung des zugrundeliegenden Objektgedankens und eine entsprechend geeignetere Umsetzung des objektorientierten Paradigmas. Der Schwerpunkt dabei liegt in der Klärung, wie Anforderungsspezifikationen im

Großen beschrieben werden, wie also umfangreiche und komplexe Software-Systeme spezifiziert werden.

1.4 Ergebnisse

Diese Arbeit stellt die Grundkonzeption und den Aufbau der Sprache ADORA-L vor, mit deren Hilfe Software-Anforderungen konstruktiv durch ein objektorientiertes Modell erfaßt werden. ADORA-L hebt sich von der UML und von vielen anderen bekannten, aber mäßig erfolgreichen objektorientieren Modellierungssprachen ab. Die vorgestellte Sprache zeichnet sich durch intuitivere und präzisere Sprachkonzepte aus und stellt eine besser verwendbare Notation zur Verfügung. Statt der wenig praktikablen diagrammbasierten Konzeption werden Systemmodelle in ADORA-L zusammenhängend, abstrahierend und in variablem Präzisionsgrad beschrieben. Die Sprache selber wird durch die folgenden drei Eigenschaften charakterisiert:

- *Hierarchische Dekomposition der Modellstruktur* – Systeme werden zerlegt in Kompositionen und in darin enthaltene Komponenten. Dies erlaubt eine einfache und leichte Übernahme von physischen Strukturen der Realität in ein Systemmodell.
- *Integrierte Modellierung von Modelleigenschaften* – zu modellierende Eigenschaften werden nicht separat, sondern zusammenhängend und gemeinsam in einem gemeinsamen Modell beschrieben.
- *Variabilität in der Modellierungspräzision* – Zu modellierende Sachverhalte können im Systemmodell variabel präzise, d.h. variabel formal modelliert werden. Zu modellierende Sachverhalte können sowohl grob und mit wenig Aufwand, als auch detailliert und mit hohem Aufwand beschrieben werden.

ADORA-L ist Teil des Projekts ADORA, welches die Entwicklung einer fundierten und praxisbezogenen Methode zur Gewinnung, Dokumentation und Prüfung von Anforderungsspezifikationen zum Ziel hat.

Die Grundidee von ADORA-L ist die Beschreibung von Daten-, Funktions- und Verhaltensaspekten eines Systems in einer einzigen *integrierten hierarchischen Modellstruktur.* Im Gegensatz zu bekannten objektorientierten Sprachen erfolgt die Modellierung auf Ebene von *abstrakten Objekten* und nicht auf Klassenebene. Ein *abstraktes Objekt* ist dabei nicht als konkretes Individuum der Wirklichkeit zu verstehen. Es ist vielmehr ein abstrakter Repräsentant, der in einer bestimmten Rolle verwendet wird. Die prinzipielle Modellierung auf Objektebene erscheint uns natürlicher und intuitiver als die bisher gebräuchliche klassenorientierte Denkweise. Sie ist Grundlage für die Bildung einer aussagekräftigen und gut gegliederten Modellstruktur. *Die hierarchische Modellstruktur* wird realisiert durch die Verwendung einer *Dekompositions-* oder *Teil/Ganzes-Hierarchie.* Objekte des Modells werden als Komponenten anderer Objekte, den *Kompositionen* dargestellt. *Kompositionen* sind im Gegensatz zu beste-

henden objektorientierten Modellierungssprachen nicht mehr nur bedeutungsarme Objektbehälter. Sie sind vielmehr vollwertige Objekte, deren Eigenschaften, Funktionalitäten und Verhaltensweisen mit denen ihrer Komponenten abgestimmt sind. Das Verhalten wird in ADORA-L zustandsbasiert in Anlehnung an den Statechart-Ansatz [Harel87] modelliert. Durch diese *Objekt*-Modellierung wird die Verhaltensbeschreibung präziser und verständlicher. Spezifikationen werden somit ausführbar und dadurch besser prüfbar.

ADORA-L überwindet die Defizite vorhandener Ansätze, Anforderungsspezifikationen im Großen zu beschreiben. Möglich wird dies durch die Bereitstellung wirkungsvoller Mechanismen zur systematischen Strukturierung von Spezifikationen. Anforderungen sind dadurch variabel präzise und formal beschreibbar. Diese Flexibilität ist notwendig, um die Spezifikation an Kosten und Risiko anzupassen. ADORA-L ist in einen Spezifikationsprozeß eingebettet und bietet Verknüpfungspunkte zur Erfassung und zur Prüfung von Anforderungen. Durch die Verwendung eines objektorientierten Paradigmas lassen sich die Anforderungsspezifikationen zudem einfach in Architekturentwürfe überführen.

1.5 Aufbau der Arbeit

Die Arbeit ist folgendermaßen aufgebaut: *Teil I* legt die Grundlagen für ADORA-L dar. *Kapitel 2* untersucht, welche Rolle und welche Bedeutung die Anforderungsspezifikation in der Software-Entwicklung spielt und welche Konsequenzen sich daraus für die Verwendung einer Spezifikationssprache ergeben. Darauf aufbauend werden in *Kapitel 3* Anforderungen an eine Spezifikationssprache erarbeitet, wobei besonderes Augenmerk auf die Entwicklung umfangreicher und komplexer Spezifikationen gelegt wird. Auf Grundlage dieser Anforderungskriterien untersucht *Kapitel 4* existierende Spezifikationssprachen und arbeitet jeweils deren Stärken und Schwächen heraus.

Der *Teil II* stellt die Spezifikationssprache ADORA-L vor. *Kapitel 5* führt die grundlegenden Sprachkonzepte und Prinzipien auf. Es klärt und motiviert die Konzepte der hierarchischen Modellstruktur, das Kommunikationskonzept sowie die Konzepte zur Verhaltens- und zur Funktionsbeschreibung. *Kapitel 6* liefert einen Überblick über die Sprache ADORA-L, indem die einzelnen Komponenten der Sprache skizzen- und beispielhaft vorgestellt und erläutert werden.

Teil III schließlich liefert die eigentliche Sprachdefinition von ADORA-L. *Kapitel 7* erläutert die einzelnen Komponenten der Sprache durch Definition der ADORA-L-Sprachkonstrukte in Syntax und Semantik. Um einen besseren Eindruck von der Sprache zu erhalten, werden die einzelnen Sprachkonstrukte anhand eines durchgängigen Beispiels illustriert. *Kapitel 8* schließlich faßt die Ergebnisse der Arbeit zusammen und bewertet die entwickelte Sprache

ADORA-L. Es liefert weiterhin einen Ausblick, indem es weitere laufende und zukünftige Aktivitäten des Projekts ADORA skizziert.

Kapitel 2

Kontext der Arbeit

Dieses Kapitel erläutert die Rolle und Bedeutung der Anforderungsspezifikation und des Spezifikationsprozesses innerhalb der Software-Entwicklung. Zu Beginn wird die Software-Entwicklung selber als eine Ingenieurdisziplin vorgestellt. Anschließend wird geklärt, wie sich die Anforderungsspezifikation in den Software-Entwicklungsprozeß einbettet und welche Bedeutung dabei die geeignete Wahl der Spezifikationssprache einnimmt.

2.1 Software-Entwicklung als Ingenieurdisziplin

Die Computerindustrie ist im Vergleich zu anderen technischen Bereichen eine vergleichsweise junge Industrie. Mittlerweile finden sich Hard- und Software in nahezu allen Bereichen in Handel und Industrie wieder; große technische und kommerzielle Systeme sind ohne sie nicht mehr vorstellbar. Möglich wurde dieser Boom durch die Entwicklung immer leistungsfähigerer und billigerer Hardware. Der hohe volkswirtschaftliche Nutzen und die Verfügbarkeit leistungsfähiger Hardware führt zur Entwicklung immer komplexerer und umfangreicherer Software. Im Gegensatz zu den bahnbrechenden Entwicklungen im Hardware-Bereich wurde die Software trotz ihrer Komplexität ad-hoc, individuell und wenig geplant entwickelt. War dieses Vorgehen gerade noch ausreichend, um Bedürfnisse von überschaubaren Lösungen für den Eigengebrauch zu befriedigen, so versagte es bei den neuen Ansprüchen. Die Entwicklung begann zumeist mit der ersten Programmzeile, eine detaillierte Klärung der Problemstellung, der Software-Architektur fand in der Regel genauso wenig statt wie Qualitätsprüfungen und Prozeß- und Projektplanungen.

Dieses Dilemma führte Ende der 60er Jahren im Rahmen zweier NATO-Tagungen [Naur76] zur Forderung nach Software Engineering, also zur Software-Entwicklung als ingenieurmäßiger Disziplin, wie sie in anderen technischen Bereichen üblich ist. Seit der Einführung des Begriffs hat sich Software Engineering von der Provokation zur allgemein anerkannten Disziplin entwickelt. Ohne sie erscheint die wirtschaftliche und risikoarme Entwicklung von Software nicht mehr möglich.

2.2 Spezifikation von Anforderungen

Die Anforderungsspezifikation übernimmt in der Entwicklung von Software eine herausragende Rolle. In ihr werden verbindliche produktspezifische Absprachen zwischen Kunden und Software-Entwicklern festgelegt. Werden die Anforderungen an ein Software-System fehlerhaft oder unklar erfaßt, weist auch das realisierte Software-System entsprechende Defizite und Schwachstellen auf. Das folgende Kapitel geht daher speziell auf den Prozeß der Spezifikation ein. Es klärt, welche Aktivitäten in einem Spezifikationsprozeß unterschieden werden und wie diese Aktivitäten charakterisiert sind. Abschließend wird die Bedeutung einer geeignet gewählten Spezifikationssprache im Spezifikationsprozeß erläutert.

Eine *Anforderung* ist eine Aussage oder eine Fähigkeit, die eine Software erfüllen oder besitzen muß, um einen Vertrag, eine Norm oder ein anderes formell bestimmtes Dokument zu erfüllen [IEEE90]. Der Begriff der Anforderungsspezifikation wird dabei in zwei Bedeutungen verwendet: Als das resultierende *Dokument*, welches die ermittelten Anforderungen enthält und als *Prozeß* der Entwicklung dieses Dokuments. Zur Vermeidung einer Mehrdeutigkeit werden hier die beiden Bedeutungen sprachlich wie folgt unterschieden:

- Das Anforderungsdokument wird als *Spezifikation*, *Anforderungsspezifikation* bezeichnet. In der Literatur finden sich hierfür auch die Ausdrücke *Pflichtenheft*, *Anforderungsdokument* und *Software Requirements Specification*.
- Die Vorgehensweise zur Ermittlung des Anforderungsdokuments wird als *Spezifikationsprozeß* bezeichnet [Glinz98a].

Der *Anforderungsspezifikation* und dem *Spezifikationsprozeß* kommen im Rahmen einer Software-Entwicklung eine zentrale Bedeutung zu [Hsia88, S. 77f]. Ohne das Vorhandensein einer guten Anforderungsspezifikation

- wissen die Entwickler nicht, was sie erstellen sollen,
- wissen die Kunden nicht, was sie geliefert bekommen
- gibt es keine Möglichkeit, das entwickelte System gegen die gestellten Anforderungen zu prüfen

Für den Spezifikationsprozeß muß also gleiches gelten wie für den gesamten Entwicklungsprozeß. Für eine systematische und disziplinierte Software-Entwicklung ist im Sinne des Software Engineerings ebenfalls eine ingenieurmäßige Herangehensweise notwendig.

2.2.1 Anforderungstechnik

Das systematische, disziplinierte und überprüfbare Vorgehen zur Entwicklung von Anforderungen wird in der Informatik als *Anforderungstechnik* (engl.: *Requirements Engineering*) bezeichnet. Die *Anforderungstechnik* unterscheidet die drei folgenden grundlegenden Entwick-

lungsaktivitäten: Das *Erfassen*, das *Dokumentieren* und das *Prüfen* von Anforderungen [Pohl93].

- Die Aktivität der *Erfassung von Anforderungen* beschäftigt sich mit der Anforderungsgewinnung und damit vorwiegend mit der Analyse des gestellten Problems.
- Die Aktivität der *Dokumentation von Anforderungen* beschäftigt sich damit, wie Anforderungen auf geeignete Art und Weise strukturiert und niedergeschrieben werden können.
- Die Aktivität der *Prüfung von Anforderungen* gleicht die dokumentierten Anforderungen mit den Vorstellungen des Kunden ab bzw. deckt Unstimmigkeiten zwischen den dokumentierten Anforderungen auf.

Die drei Aktivitäten sind nicht sequentiell, sondern als iterativer Spezifikationsprozeß in Zusammenarbeit mit dem Kunden zu verstehen [Pohl93]. Ein typischer Ablauf eines Spezifikationsprozesses in Abbildung 1 soll den iterativen Charakter und die Zusammenarbeit zwischen Entwickler und Kunden verdeutlichen.

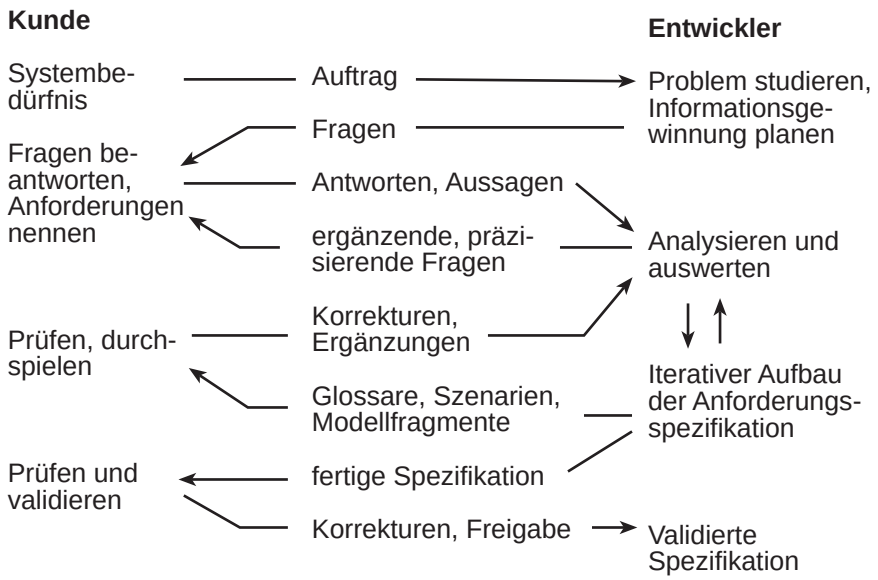


Abb. 1: Beispielhafter Ablauf eines Spezifikationsprozesses nach [Glinz98a]

Ein systematischer Spezifikationsprozess ist notwendig, um die Kosten der Gesamtentwicklung zu senken und das Risiko einer fehlgeschlagenen Entwicklung zu minimieren. [Macaulay96] identifiziert drei Vorteile eines systematischen und geplanten Spezifikationsprozesses:

- Die Führung und Lenkung eines Projekts aufgrund von definierten Resultaten und Arbeitsergebnissen. Spezifikationsprozesse werden daher sinnvollerweise durch zu erbringende Arbeitsergebnisse spezifiziert.

- Die Meß- und Prüfbarkeit der Effektivität des gewählten Prozesses und damit verbunden die Möglichkeit der Prozeßverbesserung.
- Die Möglichkeit, einen systematischen Prozess mit Hilfe von computerunterstützten Werkzeugen zu automatisieren.

Die folgenden Kapitel beschäftigen sich im einzelnen mit den Aktivitäten der Erfassung, der Dokumentation und der Prüfung von Anforderungen.

2.2.2 Erfassen von Anforderungen

Bei der Erfassung von Anforderungen werden die Bedürfnisse des Kunden geklärt mit dem Ziel, ein kollektives Verständnis des Problems aller Beteiligten zu erhalten [Crow95]. In der Literatur wird diese Aktivität auch häufig als Problemanalyse bezeichnet [Macaulay96]. Beteiligt sind dabei die folgenden Rollen:

- Die Auftraggeber der Software-Entwickler
- die Benutzer des zukünftigen Systems
- Personen, die zwar das Software-System nicht direkt benutzen werden, die jedoch indirekt durch Prozesse des IST-Systems involviert sind.

Für die Gewinnung von Anforderungen sind sowohl die vorhandenen als auch die zukünftigen Technologien zu berücksichtigen. Dies gilt insbesondere dann, wenn Handlungsabläufe in bisherigen Überlegungen zwar sinnvoll automatisiert werden könnten, die Technologien dafür aber nicht oder nur eingeschränkt vorhanden sind. Und schließlich ergeben sich Anforderungen aufgrund gültiger Standards und Gesetze, die mit dem Problembereich in Zusammenhang stehen. Bei der Erfassung von Anforderungen sind sämtliche Aspekte des Problembereichs zu untersuchen. Gegenstand der Untersuchung sind nach [Davis93, S.23f]

- Personen und externe Systeme, die mit dem Problembereich in Zusammenhang stehen bzw. stehen werden
- Dokumente, die von diesen Personen oder Systemen zur Zeit oder zukünftig erstellt, bearbeitet oder in Anspruch genommen werden bzw. zukünftig verwendet werden sollen
- Tätigkeiten, die von diesen Personen oder Systemen zur Zeit oder zukünftig durchgeführt werden.
- Verfahren und Methoden, wann und wo diese Tätigkeiten zur Zeit oder zukünftig durchgeführt werden.

Vereinfacht ausgedrückt ist die Gewinnung von Anforderungen eine Aktivität, in der Analytiker (Entwickler) die Zeit damit verbringen, Personen als potentielle Informationsträger zu befragen und die gewonnenen Informationen und Randbedingungen gedanklich zu durchdrin-

gen [Davis93, S. 20]. Bei dieser vereinfachten Sicht sollten folgende prinzipiellen Probleme mitberücksichtigt werden ([Glinz98a, Kapitel 7], [Davis93, S. 43], [Macaulay96, S. 2]).

- Die beteiligten Personen auf Kundenseite haben unterschiedliche Beweggründe und Sichtweisen auf das zu entwickelnde System. Unter Umständen ergeben sich daher sich widersprechende Anforderungen. Widersprüche ergeben sich beispielsweise zwischen den Vorstellungen der Benutzer, die auf komfortable Bedienbarkeit Wert legen, und den Auftraggebern, die den finanziellen Aspekt hoch bewerten und deshalb ein gutes Nutzen-Kosten-Verhältnis anstreben.
- Kundenvertreter haben zwar konkrete Vorstellungen, was sie wollen, können diese aber nicht richtig formulieren und daher nur unzulänglich den Entwicklern mitteilen. Dieses Problem wird durch die Tatsache verstärkt, daß Kundenvertreter und Entwickler in der Regel unterschiedliche Fachsprachen verwenden und daß zudem unterschiedliche Denkweisen aufeinanderprallen.
- Kundenvertreter haben nur eine vage Vorstellung, was das zu erstellende System leisten soll. Dieses Problem tritt vorwiegend dann auf, wenn das zu entwickelnden System betriebsspezifische Handlungsabläufe in hohem Maße verändert und automatisiert und eine starke Restrukturierung des System-Umfelds zur Folge hat.

Die Probleme bei der Anforderungsgewinnung erfordern fundierte Methoden, welche die von [Davis90] aufgeführten Aktivitäten «Interviews führen», «Problem gedanklich durchdringen» und das «Finden von Randbedingungen» unterstützen und systematisieren. Die Methoden zur Anforderungsgewinnung lassen sich grundsätzlich in zwei Kategorien einteilen: In Methoden,

- die den Umgang und die Kommunikation mit Kundenvertretern beschreiben (i) und
- die eine Beschreibung problemspezifischer Aspekte zum Ziel haben (ii).

Methoden der Kategorie (i) betonen den sozialen und psychologischen Aspekt im Umgang mit dem Kunden und beschreiben vornehmlich die Form und die Gestaltung der Kunden-Entwickler-Kommunikation mittels Interviews, Fragebogen, Workshops usw. Beispiele hierfür sind etwa die Ansätze ETHICS [Mumford84], des partizipativen Designs ([Floyd89], [Greenbaum91], [Kyng91]) oder auch JAD [Martin91].

Methoden der Kategorie (ii) beschäftigen sich primär mir konkret fachlichen und problemspezifischen Fragestellungen, indem für den Kunden relevante Aspekte betrachtet werden. Beispiele für solche Methoden sind etwa die Ansätze zur Erfassung von Anwendungsfällen (siehe [Anderson93], [Jacobson94] und [Glinz95]), die Erstellung von Glossaren zur Festlegung der Begriffe im Problembereich (siehe [McDavid96], [Ortner98], [Sowa92]), oder die Domänenmodellierung etwa durch Entity-Relationship-Modelle [Chen76].

2.2.3 Dokumentation und Darstellung von Anforderungen

Ein zentraler Bestandteil des Requirements Engineerings beschäftigt sich mit dem Inhalt und dem Aufbau des Anforderungsdokumentes. Geklärt werden soll dabei, welche Aspekte zu beschreiben sind, welche Sprachmittel hierfür verwendet werden und wie das Dokument organisiert und strukturiert sein sollte. Für die Entwicklung von Software ist sie aus folgenden drei Gründen essentiell und unverzichtbar [Davis93, S. 177]:

- Die Anforderungsspezifikation ist Kommunikationsmittel zwischen Auftraggeber, Benutzer und Entwickler. Eine Anforderungsspezifikation verringert die Wahrscheinlichkeit, daß Differenzen nicht erst bei der Abnahme, sondern bereits im Spezifikationsprozeß aufgedeckt werden.
- Sie ist schriftliche Vorgabe für Test-, Prüf- und Abnahmepläne im Rahmen des Software-Entwicklungsprozesses.
- Die Anforderungsspezifikation ist Grundlage für Führung und Lenkung der Evolution des zu entwickelnden Systems. Sie allein definiert, was die Software leisten soll und welche Leistungen nicht bereitgestellt werden. Die Veränderungen von Anforderungen ist daher nur dann beherrschbar, wenn diese in der Anforderungsspezifikation mit einfließen und die Evolution und ihre verschiedenen Konfigurationen systematisch dokumentiert werden.

Eine Anforderungsspezifikation sollte eine vollständige Beschreibung der Schnittstelle des Systems mit seiner Umgebung, d.h. anderen Software-Systemen, Kommunikationskanälen, Hardware und menschlichen Benutzern beinhalten [Davis, S. 179]. Beim Inhalt einer Anforderungsspezifikation unterscheidet man *funktionale* und *nichtfunktionale* Anforderungen. *Funktionale* Anforderungen beschreiben, welche Aufgaben das Software-System erfüllen soll, d.h. welche Resultate aufgrund welcher Eingaben zu liefern sind. Die Gesamtheit aller funktionaler Anforderungen definiert ein System als eine komplexe funktionale Transformation von Eingangsgrößen zur Ausgabe der gewünschten Resultate, abhängig von den möglichen Zuständen des Systems. Konkret sind die funktionalen Aspekte zu klären,

- welche Daten in welcher Form erzeugt, verwendet, verändert und abgelegt werden,
- welche Funktionen an der Transformation von Eingangs- in Ausgangsgrößen beteiligt sind und
- welche Daten durch sie verarbeitet werden.

Nichtfunktionale Anforderungen legen die Art und Weise fest, in der die Aufgaben des Software-Systems zu erbringen sind. Nichtfunktionale Anforderungen machen Aussagen über Leistungsfähigkeit, besondere Qualitäten und Randbedingungen des Systems. Leistungsfähigkeiten sind demnach Aussagen über Zeiten, Geschwindigkeiten, Raten, Mengen usw. Besondere Qualitäten sind z. B. Zuverlässigkeit, Benutzerfreundlichkeit, Wartbar- und

Portierbarkeit. Randbedingungen sind alle Forderungen, welche die Menge der möglichen zusätzlich beschränken, z. B. Gesetze und Normen [Glinz98a, Kapitel 7].

Funktionale und *nichtfunktionale* Anforderungen legen die produktspezifische Eigenschaften von Software fest. Projektspezifische Anforderungen spezifizieren das Software-Projekt an sich, machen also etwa Aussagen über Zeit- und Kostenaufwendungen, Meilensteine, Aktivitäten und Phasen des Projekts. Nach Davis [Davis90, S. 180] sollten jedoch projektspezifische Anforderungen nicht in die Anforderungsspezifikation selber aufgenommen, sondern separat und entkoppelt beschrieben werden. Gleiches gilt für Dokumente, die im Rahmen des Qualitäts- und Konfigurationsmanagements entstehen. Die Lebensdauer einer Anforderungsspezifikation ist eng an die Lebensdauer des Software-Systems gekoppelt, während Projektpläne vorwiegend in der Entwicklung Verwendung finden. Zudem werden Projektpläne und Spezifikationen in unterschiedlichen Kontexten erstellt und angewendet. Von zentraler Bedeutung sind die Fragen,

- wie das Anforderungsdokument aufgebaut und gegliedert ist und
- mit welchen sprachlichen Mitteln die Anforderungen dokumentiert werden.

Für den Aufbau und die Gliederung des Anforderungsdokuments existieren eine Reihe von Standards, die im wesentlichen Vorschläge sind zur organisatorischen Gliederung des Anforderungsdokumentes. Beispiele hierfür sind die Standards des amerikanischen Verteidigungsministeriums [DoD85], der [NASA76] oder etwa des IEEE [IEEE93]. Solche Standards skizzieren nur ein grobes Gerüst eines Anforderungsdokuments durch Vorgabe eines Inhaltsverzeichnisses.

Generell sind detailliertere Fragen über den Aufbau stark abhängig von den verwendeten Spezifikationssprachen und den zugrundeliegenden Sprachprinzipien. Die Verwendung objektorientierter Sprachen und Vorgehensweisen legt beispielsweise nahe, auch das Anforderungsdokument selber «objektorientiert» zu gliedern, d.h. nach Klassen, Objekten oder Teilsystemen geordnet (siehe auch Kapitel 4.5, S. 54). Funktionsorientierte Ansätze (z. B. die strukturierte Analyse (siehe auch Kapitel 4.4, S. 47) legen hingegen eine Gliederung nach Funktionen nahe (beispielsweise nach ([Fairley85]).

2.2.4 Prüfung von Anforderungen

Innerhalb des Spezifikationsprozesses stellt die Prüfung von Anforderungen die dritte zentrale Aktivität dar. Die primäre Zielsetzung dabei ist eine möglichst frühzeitige Aufdeckung von Entwicklungsfehlern um somit Fehlerkosten minimal zu halten. Als Entwicklungsfehler werden angesehen:

- *falsche* (d.h. nicht zutreffende) Anforderungen

- *fehlende* Anforderungen
- *sich widersprechende* Anforderungen
- *mehrdeutige* Anforderungen

Diese Fehlerarten sind gefährlich, weil sie nur durch Domänenwissen und dadurch nur durch Mithilfe des Kunden offensichtlich werden können. In Entwicklungsphasen ohne Kundenbeteiligung verschleppen sie sich, führen zu Folgefehlern und damit zu hohen Fehlerbehebungskosten. Darüber hinaus gibt es eine Reihe weiterer Faktoren, die zu prüfen sind. [Davis94, S. 181] führt folgende Faktoren an: Die *Verständlichkeit* des Dokumentes, die *Prüfbarkeit* der aufgeführten Anforderungen, die *Rückverfolgbarkeit* der aufgeführten Anforderungen auf früher erstellte Dokumente (z. B. Gesprächesnotizen, Vorabstudien, Standards) und die Entwurfsunabhängigkeit. Die Faktoren *Verständlichkeit* und *Prüfbarkeit* sind hierbei von besonderer Bedeutung. Unverständliche und nicht prüfbare Anforderungen erschweren nachfolgende Entwicklungsaktivitäten, da die getroffenen Vorgaben im Anforderungsdokument schlechter interpretiert und umgesetzt werden können. Unverständliche Anforderungsdokumente erschweren zudem das Auffinden der oben genannten «harten» Fehler, da der Abgleich mit den Kundenvorstellungen erschwert wird.

Für eine systematische Prüfung von Anforderungsdokumenten stehen eine Reihe von Verfahren zur Verfügung: *Reviews und verwandte Inspektionsansätze, analytische und mathematische Prüfmaßnahmen* sowie *Prototypenentwicklungen*.

- Ein Review ist eine formell organisierte Zusammenkunft von Personen zur inhaltlichen oder formellen Überprüfung eines Produktteils (in diesem Fall eines Anforderungsdokumentes) nach vorgegebenen Prüfkriterien [Frühauf91]. Inhaltliche Prüfungen gleichen die Vorstellungen des Kunden mit den Inhalt der Anforderungsspezifikation ab. Für eine inhaltliche Prüfung ist daher eine Beteiligung von Kundenvertretern Voraussetzung. Formale Prüfungen gleichen das Anforderungsdokument gegen Referenzunterlagen wie Richtlinien und Standards ab oder decken Widersprüche und Inkonsistenzen innerhalb des Anforderungsdokumentes auf.
- *Analytische und mathematische* Prüfmaßnahmen untersuchen Anforderungsdokumente unter Zuhilfenahme von computergestützten Werkzeugen und sind vorwiegend in der Lage, Widersprüche, Anomalien und Lücken aufzudecken. Mathematische Prüfmaßnahmen erlauben das Beweisen von definierten Sachverhalten, indem sie auf mathematische Theorien Bezug nehmen [Sommerville96, S. 465]. Voraussetzung hierfür ist eine Spezifikationssprache, die auf einer hinreichend fundierten mathematischen Basis aufbaut.
- *Prototypen* ermitteln und validieren Anforderungen, indem Teile des Systems vorab anhand lauffähiger Software realisiert und dem zukünftigen Benutzer nahegebracht werden können. Weiterhin erlauben es Prototypen, die Machbarkeit kritischer Anforderungen zu prüfen [Floyd84]. Prototypen sind besonders geeignet, um Adäquatheit und Vollständigkeit der Anforderungen zu klären, erfordern jedoch einen hohen Entwicklungsaufwand. Prototypen

schließlich setzen zwar prinzipiell keine Ansprüche an den Aufbau der Spezifikation voraus. Soll jedoch ein Prototyp automatisch aufgrund der Spezifikation erzeugt werden, so ist dort neben einer definierten Syntax eine Ausführungssemantik notwendig.

Die Effektivität der verschiedenen Prüfverfahren hängt stark von der verwendeten Spezifikationssprache ab. Reviews sind besonders dann effektiv, wenn die Anforderungen verständlich beschrieben sind, d.h. sowohl die verwendete Notation bekannt und einfach verständlich ist und das Dokument klar und übersichtlich gegliedert ist. Analytische Prüfmaßnahmen bzw. automatische Prototypen-Generierungen setzen eine syntaktisch ausreichend formale Spezifikationssprache voraus bzw. erfordern Informationen zur Ablaufsemantik der Sprachkonstrukte.

2.3 Bedeutung von Beschreibungssprachen im Spezifikationsprozeß

Die Bedeutung von Spezifikationssprachen sollte bei der Spezifikation nicht unterschätzt werden. Die Wahl einer geeigneten Spezifikationssprache hat erhebliche Auswirkungen auf die Qualität der Anforderungsspezifikation. Eine gut gewählte Spezifikationssprache

- unterstützt durch geeignete Sprachkonzepte und -konstrukte die Bildung einer klaren und abstrahierenden Struktur der Anforderungsspezifikation.
- unterstützt den Spezifikationsprozeß, indem sie über Sprachkonzepte methodische Hilfestellungen liefert. Die Verwendung einer objektorientierten Spezifikationssprache beispielsweise unterstützt eine objektorientierte Vorgehensweise [Embley92, S. 5].
- vereinfacht die Umsetzung von Spezifikationen in Entwürfe und in Implementierungen mit «kompatiblen» Konzepten. Eine Spezifikation, die auf Basis einer objektorientierten Sprache entwickelt wurde, läßt sich etwa ohne Paradigmenwechsel in einen objektorientierten Entwurf überführen.
- trägt konstruktiv zur Fehlervermeidung bei und ermöglicht analytische Konsistenzprüfungen, indem die Sprache Informationen in geeigneter Weise redundant beschreibt. Durch Restriktionen werden so Inkonsistenzen in der Spezifikation offensichtlich.
- ermöglicht die Entwicklung von Prototypen, die aus der Anforderungsspezifikation generiert werden. Eine Prototypengenerierung ist dann möglich, wenn für Sprachkonstrukte eine Ausführungssemantik vorhanden ist. Gegenüber herkömmlich programmiertem Prototypen sind generierte Prototypen einfacher erstell- und änderbar. Durch die eindeutige Zuordenbarkeit zu Teilen der Spezifikationen ist einfach nachweisbar, welche Anforderungen durch den Prototypen realisiert werden. Bei «hard«-codierten Prototypen hingegen koexistieren Prototyp und Spezifikation nebeneinander und müssen gegenseitig konsistent gehalten werden. [Glinz93, S. 175 f.]

Kapitel 3

Anforderungen an eine Spezifikationssprache

Die vorhergehenden Kapitel diskutierten die Bedeutung von Anforderungsspezifikationen. Dieses Kapitel beschäftigt sich mit der Frage, welche Eigenschaften eine Spezifikationssprache aufweisen sollte, um die Güte des Anforderungsdokuments zu erhöhen und gleichzeitig positive Auswirkungen auf den Spezifikationsprozeß zu haben.

Zunächst wird diskutiert, welche Qualitätsmerkmale im allgemeinen für eine Spezifikation und für eine geeignete Spezifikationssprache von Bedeutung sind. Im speziellen hervorgehoben werden hier die zentrale Bedeutung der Merkmale *Verständlichkeit* und *Prüfbarkeit*. Anschließend werden die beiden zentralen Sprachanforderungen aus Sicht der Praxis erläutert: Der adäquate Umgang mit *Größe und Komplexität von Problemstellungen* und mit einer *Unterstützung von inkrementellen Entwicklungen*. Aufbauend auf die generellen als auch auf die praktischen Anforderungen werden konstruktive Eigenschaften erarbeitet, die von Anforderungssprachen umzusetzen sind. Als wesentliche Eigenschaften werden hier unterschieden:

- die konstruktive Anforderungsbeschreibung durch Modelle
- der Einsatz von aussagekräftigen und von abstrahierenden Strukturierungsmechanismen
- die Unterstützung eines geeigneten Formalisierungskonzepts

3.1 Qualitätsmerkmale in der Spezifikation

Eine *gute* Anforderungsspezifikation muß eine Reihe von Qualitäten erfüllen. [Davis90, 93] sieht in den folgenden Qualitätsmerkmalen die wichtigsten Merkmale, um die Güte einer Anforderungsspezifikation zu gewährleisten:

- Adäquatheit – dokumentiert die Spezifikation das, was vom Kunden gefordert und gewünscht wird?

- *Vollständigkeit* – dokumentiert die Spezifikation alle Forderungen des Kunden (inhaltliche Vollständigkeit [Davis93, S. 187])? Ist die Spezifikation darüber hinaus auch formal vollständig, d.h. werden Standards, Reglements oder Formatierungen eingehalten?
- *Widerspruchsfreiheit* – widersprechen sich Anforderungen in der Spezifikation? Werden in der Spezifikation unterschiedliche Aussagen gemacht, die nicht in Einklang zu bringen sind (inhaltliche Widersprüche)? Werden unterschiedliche Begriffe für ein und denselben Sachverhalt verwendet oder existieren falsche Verweise und Bezugnahmen, beispielsweise in Indizes und Verzeichnissen (formelle Widersprüche)?
- *Eindeutigkeit* – ist die Spezifikation unterschiedlich auslegbar? Lassen alle dokumentierten Anforderungen genau eine Interpretation zu?
- *Prüfbarkeit* – lassen sich die dokumentierten Anforderungen überprüfen? Existieren Verfahren, um die gestellten Anforderung zu prüfen?
- *Verständlichkeit* – liegen die Anforderungen der Spezifikation in einer verständlichen und überschaubaren Form vor? Sind die Anforderungen prägnant formuliert, sind sie lesbar? Ist die gesamte Spezifikation so organisiert, daß unterschiedliche Abstraktionsebenen unterstützt werden und eine zufriedenstellende Orientierung und Manövrierbarkeit gewährleistet ist?

[Davis90] nennt darüber hinaus weitere Qualitätsmerkmale, die in zweiter Priorität zu berücksichtigen sind:

- *Änderbarkeit* – läßt sich die Spezifikation einfach erweitern oder modifizieren? Existieren im Anforderungsdokument geeignete Redundanzen, um konsistente und vollständige Änderungen zu gewährleisten?
- *Rückverfolgbarkeit* – Ist der Quelle ersichtlich, aufgrund der eine Anforderung dokumentiert wird, d.h. sind Personen, Dokumente oder Systeme, aufgrund denen eine Anforderung erhoben wurde, dokumentiert? Können diese Quellen entsprechend durch Verweise annotiert werden?
- *Entwurfsunabhängigkeit* – Ist die Spezifikation frei von Entwurfsentscheidungen?
- *Umsetzbarkeit* – Läßt sich die Anforderungsspezifikation einfach in späteren Entwicklungsphasen weiterverwenden (z. B. in der Entwurfs-, Test- oder Wartungsphase)? Werden ähnliche oder verwandte Paradigmen verwendet?

Die Qualitätsmerkmale *Verständlichkeit* und *Prüfbarkeit* sind bei der Wahl der Spezifikationssprache von besonderer Bedeutung: Die Sprache selber trägt in erheblichem Maße dazu bei, Spezifikationen verständlicher und prüfbarer zu gestalten. Die beiden Qualitätsmerkmale werden daher im folgenden im Detail diskutiert.

3.1.1 Verständlichkeit der Spezifikation und der Sprache

In der Literatur wird die *Verständlichkeit* einer Spezifikation häufig mit der Verständlich- und Handhabbarkeit der Spezifikationssprache gleichgesetzt (beispielsweise in [Davis90] oder in [Stein94]). Eine Spezifikation ist demnach *verständlich*, wenn sie auch in einer verständlichen Sprache dokumentiert ist. Natürlichsprachliche Spezifikationen sind nach dieser Vorstellung verständlicher als formale Sprachen, da natürliche Sprachen von den beteiligten Personen bereits beherrscht und verstanden werden. Die Verständlichkeit von Spezifikationen läßt sich aber nicht nur auf die einfache und intuitive Handhabbarkeit der Spezifikationssprache reduzieren: Umfangreiche natürlichsprachliche Spezifikationen sind in der Regel mangels geeigneter Strukturierungsmechanismen schlecht verständlich und daher hochgradig fehleranfällig ([Kotonya98, S. 44f], [Davis93, S. 212ff]). Die Verständlichkeit von Spezifikationen muß daher weiter gefaßt werden:

Eine Anforderungsspezifikation ist *verständlich*, wenn für alle beteiligten Personen (Auftraggeber, Systembenutzer, Entwickler)

- die einzelnen Anforderungen in klarer und verständlicher Form dokumentiert sind
- Zusammenhänge und Abhängigkeiten zwischen Anforderungen offensichtlich sind
- die gesamte Anforderungsspezifikation so organisiert und gegliedert ist, daß unterschiedliche Abstraktionsebenen unterstützt werden und eine zufriedenstellende Orientierung und Manövrierbarkeit gewährleistet ist.

Die Verständlichkeit wird in Bezug auf Anforderungsspezifikationen häufig unterschätzt. Sie unterstützt alle Tätigkeiten, bei denen Menschen mit der Spezifikation umgehen. Insbesondere unterstützt eine klar, strukturiert und verständlich geschriebene Spezifikation die Qualitätsziele *Adäquatheit*, *Vollständigkeit*, *Widerspruchsfreiheit* und *Eindeutigkeit*, da Fehler und Schwachstellen offensichtlicher und daher besser erkennbar gemacht werden.

3.1.2 Prüfbarkeit von Spezifikationen

Eine Anforderungsspezifikation ist *prüfbar*, wenn alle dokumentierten Anforderungen prüfbar sind. Eine einzelne Anforderung ist *prüfbar*, wenn endliche und kosteneffektive Verfahren existieren, mit deren Hilfe eine Person oder eine Maschine prüfen kann, ob ein Software-System die gestellte Anforderung erfüllt [IEEE84]. Neben der direkten Prüfung des Anforderungsdokuments sind explizit auch Verfahren vorstellbar, um die Spezifikation oder Teile davon automatisch oder halbautomatisch in eine andere, besser prüfbare Form umzuwandeln.

Die Wahl einer geeigneten Spezifikationssprache spielt bezüglich der Kosteneffektivität eines Prüfverfahrens eine große Rolle: Je nach deren Konzeption lassen sich *Prototypen* oder *Simu-*

lationen erzeugen. Zudem kann die Prüfbarkeit einer Spezifikation durch Hinzufügen geeigneter *Redundanzmechanismen* verbessert werden.

> **Definition** *Redundanzmechanismus – Redundanzmechanismen* sind Mechanismen, welche die Dokumentation von redundanten Informationen ermöglichen bzw. fordern. Durch Redundanzen werden Inkonsistenzen in der Spezifikation offensichtlich. Dies ist im speziellen nützlich für das Auffinden von Inkonsistenzen, die sich aufgrund von Änderungen der Spezifikation ergeben. Beispiel für Redundanzmechanismen sind die Typsysteme in Programmiersprachen. Redundanzmechanismen sind vorwiegend nützlich, um Spezifikationen auf *Widersprüche, Mehrdeutigkeiten* und bedingt auch auf *Unvollständigkeiten* zu prüfen.

Häufig werden im Zusammenhang mit der *Prüfbarkeit* nur die mathematischen und automatisierbaren Verfahren betrachtet. Zielsetzung ist hier primär eine mathematische Beweisführung, indem Invarianten über eine Spezifikation bewiesen werden. Häufig unterschlagen wird die Prüfung durch beteiligte Personen, d.h. durch Kundenvertreter, durch Endbenutzer oder durch Entwickler selber (beispielsweise in Form eines *Reviews* oder eines *Walkthroughs*). Diese Form der Prüfung hängt wiederum eng mit der Frage zusammen, wie verständlich die Spezifikation als Ganzes und im Detail ist. Eine umfangreiche, vollständig in Prädikatenlogik gehaltene formale Spezifikation beispielsweise erlaubt zwar die Prüfung anhand gewisser Beweisführungsarten, ein Abgleich mit Kundenwünschen ist wegen der schweren Verständlichkeit jedoch fast unmöglich.

3.1.3 Bewältigung der Komplexität von Anforderungen

Wie bereits in Kapitel 1.1 und 2.1 angesprochen, steigt die Software-Komplexität, also auch die Menge an Anforderungen an Software immer mehr an. Entsprechend muß eine Spezifikationssprache in der Lage sein, auch umfangreiche Spezifikationsdokumente in geeigneter Weise zu beschreiben. Geeignet bedeutet im besonderen,

- einfach in einer großen Menge von Informationen zu navigieren, also sowohl einen Überblick im Groben, als auch von gewünschten Teilen Detailbeschreibungen zu bekommen.
- unterschiedlichen Aufwand in der Präzision und in der Detaillierung zu betreiben. Nebensächliche oder beherrschbare Teilbereiche nur unscharf und grob zu beschreiben, andererseits aber auch kritische und risikoreiche Bereiche präzise und detailliert zu beschreiben.
- Sachverhalte in frühen Entwicklungsschritten grob und nur rudimentär zu skizzieren und in weiteren Entwicklungsschritten diese durch Detailinformationen anzureichen, ohne daß die Eigenschaft der vergröberten Vorstellung von Sachverhalten verloren geht.

- Umgekehrt eine detaillierte und dadurch sehr komplexe Beschreibung eines Sachverhaltes durch einen vergröberten Überblick anzureichern, ohne daß die Detailbeschreibung verloren geht.

Ziel ist die Problembeschreibung auf detaillierten und auf vergröberten Ebenen. Zudem müssen Detail- und die Grobbeschreibungen konsistent und widerspruchsfrei sein. Hierfür muß die Spezifikationssprache geeignete *Strukturierungskonzepte* und *-mechanismen* zur Verfügung stellen.

3.1.4 Unterstützung von inkrementellen Entwicklungen

Die Anforderungen an Software sind einer ständigen Veränderung unterworfen (siehe Kapitel 1.2). Von besonderer Bedeutung sind daher *inkrementelle Prozeßmodelle*, welche mit der Problematik der Anforderungs-Evolution und der Fehlerfortpflanzung auf geeignete Weise umgehen (siehe [Basili75]). Grundidee der inkrementellen Prozeßmodellierung ist es, ein System nicht als Ganzes zu entwickeln, sondern die Software inkrementell «wachsen» zu lassen. Anhand der Klärung des groben Leistungsumfangs werden ein *Kernsystem* sowie darauf inkrementell aufsetzende Erweiterungen identifiziert (Abbildung 2). Die Erweiterungen werden jeweils als Teillieferungen an den Kunden ausgeliefert, wobei jede Teillieferung ein betriebsfähiges und einsetzbares Teilsystem darstellt. Durch die schrittweise Entwicklung und Einführung eines Systems können Erfahrungen aus dem Betrieb in die laufende Entwicklung eingebracht werden. Da der Kunde fortlaufend mit den funktionsfähigen Erweiterungen des Systems konfrontiert wird, sind zudem Spezifikationsfehler der ausgelieferten Systeme schneller aufzufinden und kosteneffizienter zu beheben.

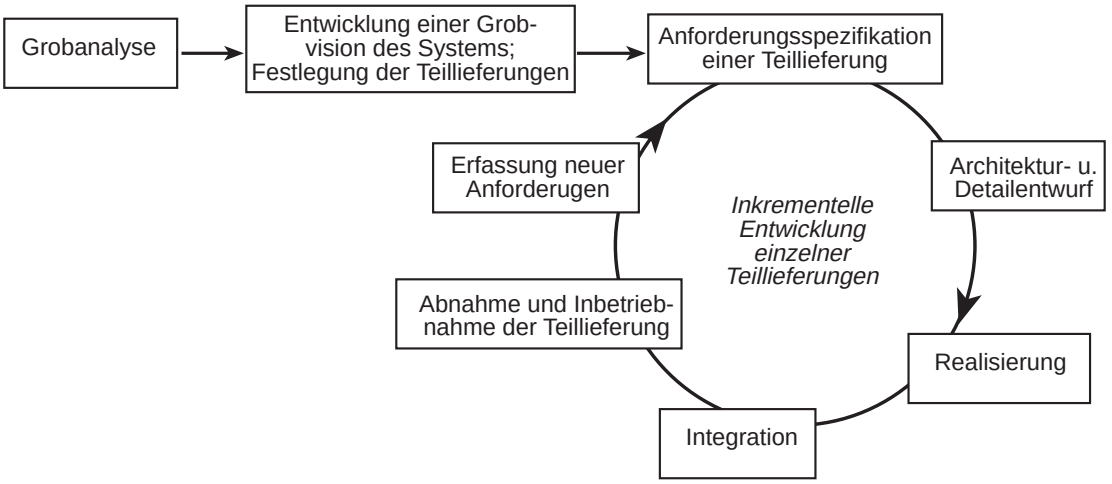


Abb. 2: Die zu durchlaufenden Phasen eines inkrementellen Prozesses. Durch die Teillieferungen wird ein real einsetzbares Basissystem inkrementell erweitert. Die Anforderungen können somit fortlaufend vom Kunden validiert werden.

Inkrementelle Prozeßmodelle stellen derzeit das Mittel der Wahl dar, um Software-Systeme im Großen zu entwickeln. Der Hauptvorteil liegt in der erwähnten Berücksichtigung der Software-Evolution, und im frühen Vorhandensein von vorzeigbaren Resultaten. Dadurch ist der Projektverlauf leichter prüfbar. Zudem lassen sich teure Spezifikations- und Entwurfsfehler frühzeitig aufdecken. Voraussetzung ist jedoch eine ausreichende Abklärung einer *Grobvision* des zu entwickelnden Systems und der entsprechenden Teilsysteme sowie eine flexible und erweiterbare Anforderungsspezifikation. Die *Grobvision* ist notwendig, um das gesamte System sinnvoll in Kernsystem und in Inkremente aufzuteilen. Hierfür ist ein flexibles Konzept der Anforderungsspezifikation notwendig, da sie im Verlauf der Iterationen ständig erweitert, modifiziert und geprüft werden muß. Im speziellen muß das Spezifikationsdokument

- variabel detailliert und formalisiert gehalten sein, um Teilprobleme sowohl präzise (für die momentane Iteration) als auch grob (für spätere inkrementelle Schritte) zu spezifizieren.
- klar strukturiert und einfach zerlegbar sein, um ein Gesamtsystem einfach durch seine Teilsysteme zu beschreiben.

Andernfalls besteht die Gefahr eines unsystematischen Entwicklungsprozesses. Dies führt zwangsläufig zu qualitativ schlecht entworfenen und hochgradig fehleranfälligen Gesamtsystemen.

3.2 Konstruktive Modellierung von Anforderungen

Modelle stellen eine attraktive Variante dar, Anforderungen konstruktiv als eine Grobvision eines zukünftigen Systems zu beschreibe. Das folgende Kapitel zeigt, warum ein solcher *konstruktiver Ansatz* besonders geeignet ist, Spezifikationen «im Großen» zu entwickeln. Es erläutert hierzu die Vorteile gegenüber einem *deskriptiven Ansatz*. Darauf aufbauend wird skizziert, welche Möglichkeiten sich durch einen solchen *konstruktiven Ansatz* ergeben.

3.2.1 Deskriptive und konstruktive Beschreibungen

Grundsätzlich unterscheidet man bei der Formulierung von Anforderungen zwischen *deskriptiven* und *konstruktiven Beschreibungen*. *Deskriptive Beschreibungen* definieren ein System durch die geforderten Ausgabedaten und die benötigten Eingabedaten [Davis90]. Das zu entwickelnde System wird als eine funktionale Transformation von Eingabe- in Ausgabedaten spezifiziert (siehe Abbildung 3). Beispiele hierfür sind die *Prädikaten-*, die *Aussagenlogik* sowie Sprachen, die auf ähnlichen mathematischen Kalkülen aufbauen. Eine *konstruktive Beschreibung* spezifiziert ein System, in dem zukünftige Komponenten und Interaktionen zwischen den Komponenten in idealisierter Weise als eine Vision des zu entwickelnden Systems beschrieben werden [Davis93, S. 177f]. Eine *konstruktive Beschreibung* ist also eine Art

abstrakter Bauplan eines zu entwickelnden Systems (Abbildung 3). Beispiele für *konstruktive Beschreibungen* sind *Datenfluß-* oder *Entity-Relationship-Modelle* (siehe auch Kapitel 4).

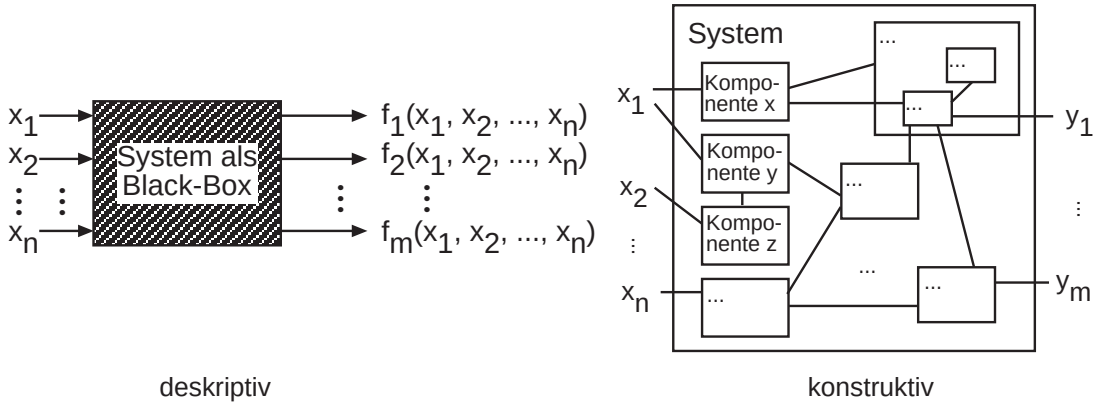


Abb. 3: Deskriptive und konstruktive Beschreibung von Anforderungen

Deskriptive Beschreibungen erscheinen zunächst geeigneter und ehrlicher, um Anforderungen zu spezifizieren: Ein System als Black-Box anzusehen unterstützt die Forderung nach Lösungsneutralität (siehe Kapitel 3.1), da die Systemarchitektur nicht durch Vorgaben und Restriktionen eingeschränkt werden kann. Sie sind jedoch bei umfangreichen Problemstellungen sehr aufwendig und häufig nur durch Spezialistenhand zu erstellen. Deskriptive Beschreibungen sind zudem häufig umständlich und daher schwer verständlich. Dies gilt im besonderen dann, wenn zustandssensitive Problembereiche spezifiziert werden sollen, da Systemzustände deskriptiv nur indirekt beschrieben werden können und lösungsneutrale deskriptive «Black-Box»-Spezifikationen dadurch in der Regel auf einer sehr abstrakten und unverständlichen Ebene liegen.

Konstruktive Beschreibungen hingegen sind lösungsabhängig, da sie Bezug auf eine *Systemvision* nehmen. Sie erschweren dadurch eine klare Trennung zwischen Spezifikation und Entwurf. Dabei besteht die Gefahr, suboptimale Lösungskonzepte durch die Spezifikation vorzugeben [Glinz98a, Kapitel 7]. Bei der konkreten Verwendung von Methoden zeigt sich jedoch, daß konstruktive Ansätze im allgemeinen und im besonderen für umfangreiche Problemstellungen die geeignetere Variante darstellen ([Davis93, S. 213f], [Glinz,98, Kapitel 7]):

- Die konkrete und direkte Systembeschreibung ist bei konstruktiven Darstellungen einfach erstellbar und in der Regel besser verständlich
- Die Zerlegung von Spezifikationen ist leichter möglich als bei deskriptiven Beschreibungen. Die direkte Beschreibung eines zu erstellenden Systems unterstützt die Identifikation von Teilsystemen und deren Separierung
- Zudem erlaubt eine konstruktive Beschreibung als eine idealisierte Lösung eine vereinfachte Realisierung, da Strukturen übernommen oder leicht modifiziert werden können.

3.2.2 Konstruktive Anforderungsmodelle

Die Vorteile eines *konstruktiven Ansatzes* führen dazu, die Spezifikation als eine Modellbildung und die Anforderungsspezifikation selber als ein *Systemmodell* zu verstehen. Der dabei verwendete Modellbegriff geht ursprünglich auf den von Stachowiak zurück [Stachowiak73]. Modelle sind danach ein Ab- oder Vorbild eines bereits vorhandenen oder noch zu erschaffenden Urbildes oder Originals. Das Modell selber repräsentiert dieses Original. Modelle im stachowiakschen Sinne sind im wesentlichen durch drei Merkmale charakterisiert, durch das Abbildungs-, das Verkürzungs- und das pragmatische Merkmal ([Schneider93], [Stachowiak73]):

- *Abbildungsmerkmal* – Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale (die selbst wieder Modelle sein können).
- *Verkürzungsmerkmal* – Modelle erfassen nicht alle Attribute bzw. Eigenschaften des durch sie repräsentierten Originals, sondern nur solche, die dem jeweiligen Modellersteller oder -benutzer relevant erscheinen.
- *Pragmatisches Merkmal* – Modelle sind nicht eindeutige Abbilder eines Originals. Sie sind stets Modelle für jemanden, zu einer bestimmten Zeit und unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen (zu einem bestimmten Zweck).

Modelle werden in Ingenieurwissenschaften häufig verwendet, um zu entwickelnde Systeme *konstruktiv* zu spezifizieren und die Eigenschaften und Besonderheiten abzuklären, bevor das eigentliche System entwickelt wird bzw. in Produktion geht. Dies gilt insbesondere dann, wenn das direkte Arbeiten mit dem Original zu *gefährlich*, zu *teuer*, *verboten* oder *unmöglich* ist. Beispiele finden sich etwa im Bauingenieurwesen oder in der Luft- und Raumfahrt in Form von verschiedenartigen Bauplänen, Konstruktionsplänen oder Simulationsmodellen. Modelle sind aber ebenso geeignet, um die Anforderungen an ein zu entwickelndes Software-System konstruktiv zu beschreiben. Dabei geht man von der Grundidee aus, ein zu entwickelndes System durch eine Vision eines möglichen Systems in abstrakter Form zu beschreiben. Im folgenden wird diese konstruktive Form der Modellierung von Anforderungen als *konstruktive Anforderungsmodellierung* und die Modelle als *konstruktive Anforderungsmodelle* bezeichnet.

> **Definition** *Konstruktives Anforderungsmodell (kurz: Anforderungsmodell)* – Ein Anforderungsmodell im Sinne des Requirements Engineerings ist eine Anforderungsspezifikation, welche auf Grundlage eines konstruktiven Paradigmas die Anforderungen an ein zu spezifizierendes Software-System durch ein Modell beschreibt. Diese Modelle beschreiben Anforderungen, indem Strukturen und Eigenschaften der Realität übernommen werden und darauf aufbauend das zukünftige System *konstruktiv* durch ein *Systemmodell* beschrieben wird.

Die Pragmatik von Anforderungsmodellen ergibt sich aus der Tatsache, daß Anforderungsspezifikationen im allgemeinen einfacher und besser durch eine konstruktive und idealisierte Modellierung als deskriptiv durch die Beschreibung von Ein-/Ausgabetransformationen (siehe Kapitel 3.2.1) zu erfassen sind. Das *Anforderungsmodell* verkürzt dieses Original, indem nur ein abstraktes und idealisiertes System beschrieben wird, welches sich auf die Spezifikation der zu erbringenden Leistungen beschränkt und nicht oder nur in Ansätzen auf die mögliche Realisierung eingeht. Anzumerken ist noch, daß prinzipiell alle Anforderungsspezifikationen Modelle und damit Vorbild eines noch zu erstellenden Systems sind, im besonderen also auch deskriptive Beschreibungsformen. Durch den Begriff Anforderungsmodellierung soll hier jedoch ausschließlich der Aspekt der *konstruktiven Modellierung* abgedeckt werden.

Bei Anforderungsmodellen sind folgende zwei Faktoren von großer Bedeutung:

- *Granularität* – Wie weit, d.h. bis zu welchem Granularitätsgrad soll das Anforderungsmodell verfeinert werden und wie können feingranulare Sachverhalte in vergröberter Form modelliert werden?
- *Formalisierung* – Mit welcher Präzision, d.h. mit welchem Formalisierungsgrad sollen Anforderungen dargestellt werden?

Die Frage nach dem erforderlichen Grad der *Granularität* und der *Formalisierung* läßt sich nicht allgemeingültig beantworten. Grundsätzlich muß abgewogen werden zwischen:

- *Kosten*, die für die Erstellung und Weiterverarbeitung des Anforderungsmodells notwendig sind. Feingranulare und stärker formalisierte Anforderungsmodelle erfordern einen erhöhten Erstellungsaufwand und sind daher mit höheren Kosten verbunden und
- *Risiko* bei der Umsetzung und der Realisierung des jeweiligen Anforderungsmodelles: Teile des Anforderungsmodells, bei denen Probleme bei der Realisierung absehbar sind oder hohe Qualitätsanforderungen vorhanden sind, sollten feingranular und/oder formalisiert dargestellt werden.

Für die Spezifikationssprache, in der das Anforderungsmodell geschrieben wird, ergeben sich daraus eine Reihe von Forderungen:

- Die Spezifikationssprache muß *Gliederungskonstrukte* enthalten, um das Anforderungsmodell durch verschiedene, aufeinander aufbauende Granularitätsstufen zu modellieren (siehe nächstes Kapitel).
- Die Spezifikationssprache sollte ebenfalls Konstrukte zur Verfügung stellen, um bestimmte Anforderungen deskriptiv zu beschreiben. Im speziellen ist die Verwendung des deskriptiven Paradigmas sinnvoll, um detaillierte und feingranulare Teile der Spezifikation abstrakt durch Ein- und Ausgabegrößen zu beschreiben. Darüber hinaus ist es von Vorteil, wenn Teile des Anforderungsmodells sowohl konstruktiv als auch deskriptiv beschrieben werden können. Diese Teile des Anforderungsmodells können sowohl detailliert durch die kon-

struktive, als auch abstrakt durch die deskriptive Beschreibung als «Black-Box» dargestellt werden. Stimmen entsprechende deskriptive und die konstruktive Beschreibungen überein, läßt sich ein komplexes Anforderungsmodell in anschauliche, aufeinander aufbauende Abstraktionsebenen gliedern.

- Die Spezifikationssprache sollte einen *variablen Formalisierungsgrad* zur Verfügung stellen. Anforderungen können angepaßt an eine Kosten-Risiko-Abwägung in einen entsprechenden Formalisierungsgrad modelliert werden, entkoppelt vom Formalisierungsgrad des restlichen Anforderungsmodells.

3.3 Strukturierung von Anforderungsmodellen

Das letzte Kapitel stellte die Modellierung von Anforderungen als eine Möglichkeit vor, um auch umfangreiche und komplexe Spezifikationen verständlich zu beschreiben. Allein die Verwendung von Modellen und von Modellierungssprachen garantiert jedoch noch nicht die gewünschten Eigenschaften aus Kapitel 3.1. Zusätzlich müssen Paradigmen, Konzepte, Mechanismen vorhanden sein, welche ein solches Modell strukturieren und gliedern. Eine von der Sprache unterstützte aussagekräftige und starke Struktur ist zudem für inkrementelle Entwicklungsprozesse wichtig: Nur so wird eine Spezifikation änderungsrobust und kann durch inkrementelle Ergänzungen problemlos modifiziert werden.

Im wesentlichen existieren zwei Prinzipien zur Strukturierung von Anforderungsmodellen, die *Abstraktion* und die *Projektion* [Yeh80]. Die Prinzipien von *Abstraktion* und *Projektion* und deren Bedeutung für die Konzeption einer Spezifikationssprache werden in den folgenden beiden Kapiteln erläutert.

3.3.1 Strukturierung nach dem Abstraktionsprinzip

Die Abstraktion ist im Rahmen der Modellbildung das wichtigste Prinzip, um komplexe und vielschichtige Sachverhalte zu modellieren und zu strukturieren [Stachowiak73, S. 144f].

> **Definition** *Abstraktion* – Eine Abstraktion eines Originals ist eine Abbildung, welche *alle* Merkmale des Originals entweder direkt oder in zusammengefaßter Form abbildet. Durch eine solche zusammenfassende Abbildung wird das Original in vergröberter Form beschrieben, ohne den Anspruch auf eine lückenlose Beschreibung zu verlieren (in Anlehnung an [Brockhaus94]).

Der Mechanismus zur Umsetzung eines Abstraktionsprinzips wird entsprechend als *Abstraktionsmechanismus* bezeichnet.

Definition *Abstraktionsmechanismus* – Ein Abstraktionsmechanismus einer (Spezifikations-) Sprache ist Mechanismus, der Sprachkonstrukte zur Verfügung stellt, um eine entsprechende Abstraktion durch eine Notation umzusetzen.

Bei der Modellierung im allgemeinen und bei der Anforderungsmodellierung im besonderen lassen sich im wesentlichen die folgende Abstraktionsarten unterscheiden: Die *Klassifikation*, die *Generalisierung*, die *Komposition* und die *Benutzung* ([Stachowiak73], [Glinz98a, Kapitel 7).

- *Klassifikation* – Eine Klassifikation faßt die Gemeinsamkeiten einer Menge von konkreten Individuen zu einem Typ zusammen, indem ausschließlich die Gemeinsamkeiten, nicht aber die spezifischen kontextabhängigen Ausprägungen zusammengefaßt beschrieben werden.
- *Generalisierung* – Eine Generalisierung faßt die Gemeinsamkeiten einer Menge ähnlicher Komponenten durch eine allgemeinere Komponente zusammen. Diese Zusammenfassung vom Spezialfall zum allgemeinen Fall abstrahiert von individuellen Eigenschaften und hebt die grundlegenden und gemeinsamen Eigenschaften hervor.
- *Komposition* – Eine Komposition interpretiert eine Menge von Komponenten als Teile einer übergeordneten Komponente. Diese übergeordnete Komponente ist somit eine strukturelle Zusammenfassung einzelner Teile. Die Komposition abstrahiert somit von strukturellen Detailbeschreibungen und hebt eine vergröberte Struktur im Sinne einer Teile- oder Komponentenschachtelung hervor. Die Zerlegung von Kompositionen in Komponenten stellt eine hierarchische Zerlegung des Ganzen in Teile dar.
- *Benutzung* – Die Benutzungsabstraktion (kurz: Benutzung) teilt Komponenten in eine aufeinander aufbauende Hierarchie ein. Eine Komponente nimmt darin Leistungen tiefer liegender Komponenten in Anspruch und stellt Leistungen höherliegenden Komponenten zu Verfügung. Die benutzende Komponente muß dabei nur Kenntnis von den Leistungen von den tieferliegenden Komponenten haben, ohne Kenntnis der Realisierung dieser Leistungen zu haben. Insbesondere ist es für die benutzende Komponente ohne Bedeutung, ob der Leistungserbringer die Leistung selber realisiert oder ob er andere tieferliegende Komponenten in Anspruch nimmt. Benutzungsabstraktionen können darüber hinaus verwendet werden, um Systeme in aufeinander und sich benutzende Schichten zu gliedern.

Abstraktionen stellen das mächtigste Strukturierungsmittel zur Gliederung von Anforderungsmodellen dar. Für eine Spezifikationssprache ist eine intelligente Umsetzung der obigen Abstraktionen durch entsprechende Abstraktionsmechanismen von zentraler Bedeutung. Erstaunlicherweise zeigt es sich, daß viele existierende Spezifikationssprachen nur wenig Gebrauch von dem Potential von Abstraktionsmechanismen machen (siehe auch Kapitel 4, S. 35).

3.3.2 Strukturierung durch Projektionen

Projektionen stellen neben dem Abstraktionsprinzip eine weitere Möglichkeit dar, komplexe Sachverhalte zu strukturieren und Spezifikationen dadurch zu veranschaulichen ([Yeh80], [Davis90]). Das zugrundeliegende Prinzip geht davon aus, nicht die Gesamtheit, sondern nur Perspektiven und Sichten auf diese Gesamtheit zu modellieren. Im Gegensatz zur Abstraktion werden also nicht Merkmale zusammengefaßt, sondern Merkmale nach bestimmten festzulegenden Kriterien ausgeblendet. In der Literatur werden Projektionen häufig auch als Sichten bezeichnet.

Definition *Projektion* (Synonym: Sicht) – Eine Projektion eines Originals ist eine Abbildung, welche einen Teil der Merkmale des Originals abbildet. Im Gegensatz zur Abstraktion existieren keine expliziten Kriterien, nach denen diese Merkmale abgebildet werden. Eine Projektion bildet also Merkmale eines Originals willkürlich ab, d.h. allein der Modellentwickler, nicht aber ein zugrundeliegendes Konzept entscheidet, welche Merkmale abgebildet werden.

Definition *Projektionsmechanismus* (Synonym: Sichtenmechanismus) – Ein Projektionsmechanismus einer (Spezifikations-)sprache ist Mechanismus, der es erlaubt, Modelle mit Hilfe von verschiedenen Sichten oder Perspektiven zu modellieren. Der Mechanismus muß darüber hinaus klären, wie sich das Gesamtmodell aus den verschiedenen Sichten und Perspektiven zusammensetzt und welche Konsistenzbedingungen dafür gelten.

Im Bereich der Anforderungsspezifikation sind grundsätzlich *Aspekt-*, *Teilsystem-*, und *Akteurprojektionen* sinnvoll und gebräuchlich:

- *Aspektprojektionen* (Aspektsichten) – Eine Aspektprojektion ist eine Projektion, welche Merkmale heraushebt, die durch Aspekte der Spezifikationssprache und ihrer Sprachkonzepte festgelegt sind. Eine Aspektprojektion ist festgelegt durch eine Menge von Sprachkonstrukten, welche hervorgehoben werden, unabhängig davon, wie diese Sprachkonstrukte ausgeprägt werden. (Beispiel: In einer Spezifikationssprache werden alle Elemente in einer Sicht dargestellt, deren Sprachkonstrukte Verhalten modellieren.)
- *Teilsystemprojektionen* (Teilsystemsichten) – Eine Teilsystemprojektion bildet Komponenten aus dem Gesamtsystem in Sichten ab, die zusammen ein Teilsystem modellieren. Die Teilsystemprojektion hat Ähnlichkeiten mit der Kompositionsabstraktion, sie ist jedoch weniger mächtig und weniger aussagekräftig. Dies liegt in erster Linie daran, daß Teilsystemprojektionen sich überschneiden können, d.h. Komponenten des Gesamtmodells in mehreren Projektionen aufgeführt werden. Im Gegensatz zu Kompositionsabstraktionen bilden Teilsystemprojektionen keine abgeschlossenen Teilsysteme. Teilsystemprojektionen

eignen sich auch nur eingeschränkt zur Bildung hierarchischer Strukturen: Projektionen über Projektionen liefern nur wenig Hinweise auf die tatsächlichen Aufbau des Gesamtsystems, im speziellen, welche Komponenten enthalten und welche nicht enthalten sind.

- *Akteurprojektionen* (Akteursichten) – Eine Akteurprojektion bildet alle Komponenten eines Gesamtsystems ab, die für einen *Akteur* relevant und sichtbar sind. Ein *Akteur* ist eine Person oder ein anderes externes System, welche(s) mit dem Gesamtsystem kooperieren, d.h. Leistungen des Gesamtsystems erbringen oder in Anspruch nehmen.

Projektionen und *Projektionsmechanismen* sind nur bis zu einem bestimmten Grad für die Strukturierung von Modellen geeignet. Gegenüber Abstraktionen haben sie den Nachteil, Sachverhalte eben nicht in abstrakter Form zusammenzufassen, sondern nur bestimmte Elemente nach festgelegten Kriterien abzubilden und dadurch hervorzuheben. Werden diese Kriterien unzureichend gewählt, sind Projektionsüberschneidungen und schlecht strukturierte Systemmodelle die Folge. Projektionen sollten daher in Spezifikationssprachen nicht als alleiniger *Strukturierungsmechanismus* sondern nur in Verbindung mit *Abstraktionsmechanismen* eingesetzt werden.

3.4 Präzision einer Spezifikationssprache

Vollständigkeit, Widerspruchsfreiheit und Eindeutigkeit gehören zu den wichtigsten Qualitätsmerkmalen einer Anforderungsspezifikation (siehe auch Kapitel 3.1, S. 19). Zentrale Aufgabe der Anforderungstechnik ist die Entwicklung von präzisen, d.h. von möglichst adäquaten, vollständigen, widerspruchsfreien und eindeutigen Spezifikationen. Eine Spezifikationssprache unterstützt die obigen Qualitätsziele vorwiegend durch das Vorhandensein eines hohen *Formalitätsgrads.* Solche *formalen Spezifikationen* sind jedoch zumeist unverständlich und nur sehr aufwendig zu erstellen. In folgendem Kapitel wird erläutert, warum *teilformale Spezifikationen* sowohl kosteneffektiv einsetzbar sind als auch ausreichend geprüft werden können. Werden zudem geeignete Strukturierungsmechanismen gewählt, sind *teilformale Sprachen* zudem gerade für Entwicklungen im «Großen» besser verständlich und dadurch geeigneter als formale oder informale Spezifikationen. Eine wichtige und notwendige Variante teilformaler Spezifikationen sind Sprachen mit *variablem Formalitätsgrad*. In diesen Sprachen können je nach geforderter Präzision verschiedene Spezifikationsteile variabel formal spezifiziert werden.

3.4.1 Formale versus informale Spezifikationssprachen

Die Sprachkonstrukte einer Spezifikationssprache mit *hohem Formalitätsgrad* sind sowohl syntaktisch als auch semantisch nahezu vollständig definiert. Diese Sprachen (im folgenden werden diese als *formale Sprachen* bezeichnet) besitzen entsprechend einen hohen Anteil an Konstrukten, die auf einem geeigneten mathematischen Kalkül basieren. Zumeist werden

Techniken und Werkzeuge zur Verfügung gestellt, welche für das Beweisen von Eigenschaften von formalen Spezifikationen sowie für Programmverifikationen eingesetzt werden können [Hall90, S. 13].

Formale Sprachen erfordern einen erhöhten Aufwand bei der Formulierung und der Dokumentation von Anforderungen [Hoare87, S. 372]. Idealerweise verbessert dieser erhöhte Aufwand die Qualität der Spezifikation und fördert konstruktiv die Klärung von kritischen Sachverhalten, d.h. die Klärung von *Widersprüchen* und *Mehrdeutigkeiten*. Weiterhin vereinfachen einige formale Sprachen den Einsatz von Software-Werkzeugen und die Generierung von Prototypen und fördern daher auch die Prüfung von *Adäquatheit* und *Vollständigkeit* (in Zusammenarbeit mit dem Kunden).

Die bisher aufgeführten Qualitätsmerkmale, Vollständigkeit, Eindeutigkeit und Widerspruchsfreiheit legen den Schluß nahe, daß Spezifikationen mit formalen Sprachen beschrieben werden sollten. Dies steht jedoch klar im Widerspruch zu den Qualitätsmerkmal *Verständlichkeit* und *Adäquatheit*, da diese formale Sprachen in der Regel wenig verständlich sind [Davis93, S. 189f]. Zudem eignen sich formale Spezifikationssprachen in der Praxis nicht für vollständige Anforderungsspezifikationen, da der Aufwand für die Formalisierung *aller* Anforderungen in der Regel zu groß ist [Davis, S. 215f]. Ähnlich verhält es sich mit der Prüfbarkeit einer Anforderungsspezifikation: Einerseits lassen sich formale Sprachen durch die Verwendung von mathematischer und analytischer Methoden besser prüfen, anderseits behindert das schlechtere Verständnis die eine befriedigende Anforderungsprüfung, da der Abgleich mit den Kundenvorstellungen dadurch erschwert wird.

3.4.2 Teilformale Spezifikationssprachen

Damit sowohl Verständlichkeit und Prüfbarkeit als auch Adäquatheit, Vollständigkeit, Eindeutigkeit und Widerspruchsfreiheit unterstützt werden, ist der Einsatz *teilformaler* Sprachen sinnvoll.

> **Definition** *Teilformale Sprache* – Eine teilformale Sprache ist eine Sprache, deren Sprachkonstrukte nur zum Teil formal definiert sind. Formal definierte Sprachkonstrukte sind sowohl syntaktisch als auch in ihrer Bedeutung präzise festgelegt. Informale Sprachkonstrukte hingegen sind nur syntaktisch festgelegt, ihre Interpretation ergibt sich implizit aufgrund der bilateralen Verständigung der beteiligten Personen.

Die Grundidee *teilformaler Sprachen* besteht darin, die Grobstruktur, also die zugrundeliegende Organisation formal und Detailinformationen weniger formal oder informal zu beschreiben. Die präzise und ausführliche Beschreibung der Grobstruktur ist sinnvoll, da dort Fehler in der Regel teure Folgefehler verursachen. Auf Detailebene hingegen ist eine unschärfere Beschreibung aktzeptabler, da Folgefehler eher lokal entstehen (im speziellen, wenn die Grob-

struktur präzise gehalten ist). Voraussetzung für den Erfolg einer teilformalen Sprache ist jedoch das Vorhandensein von geeigneten Konstrukten und Konzepten, um eine klare Struktur der Anforderungsspezifikation zu unterstützen. (siehe auch Kapitel 3.3, S. 28). Der größte Vorteil einer *teilformalen Sprache* liegt in einem günstigen Kosten/Nutzen-Verhältnis. Wird die Sprache geeignet gewählt, sind Spezifikationen

- ausreichend formal, um die Entstehung von Widersprüchen und Mehrdeutigkeiten zu erschweren
- verständlicher als informale und formale Spezifikationen, da die präzise Struktur die Orientierung und die Manövrierbarkeit erhöht und Informationen auf Detailebene schnell und einfach zu erfassen sind.
- prüfbarer als formale und informale Spezifikationen, da sie durch die gute Verständlichkeit besser validierbar sind und in eingeschränktem Maße auch mathematisch/analytische Verfahren möglich sind.

Beispiele für *teilformale Spezifikationssprachen* sind Entity-Relationship-Modellierungssprachen [Chen76], Zustandsautomaten [Hopcroft90] und Petrinetze [Petri62] (siehe auch Kapitel 4, S. 35).

3.4.3 Teilformale Sprachen mit variablem Formalisierungsgrad

Ein Schritt weiter geht die Forderung nach teilformalen Spezifikationssprachen, welche einen *variablen Formalisierungsgrad* unterstützen.

> **Definition** *Variabler Formalisierungsgrad* (in teilformalen Sprachen) – Eine teilformale Sprache unterstützt einen variablen Formalisierungsgrad, wenn Teile der Spezifikation in verschiedenen Graden der Formalisierung beschrieben werden können. Teile der Spezifikation, die logisch gesehen auf gleicher Abstraktionsebene liegen, können dadurch verschieden formal beschrieben werden.

Der Vorteil eines variablen Formalisierungsgrades liegt generell in einer besseren Anpaßbarkeit der Software-Entwicklung an Kosten und Entwicklungsrisiko einzelner Komponenten [Glinz98a, Kapitel 7].

Variable Formalitätsgrade erfordern einen erhöhten Aufwand bei der Sprachdefinition, da für einen Aspekt verschieden formale Notationen bereitgestellt werden müssen. Solche Sprachen sind in der Regel auch komplexer und erfordern einen erhöhten Lernaufwand. Die verwendeten unterschiedlich formalen Notationen sollten daher möglichst einfach gewählt werden:

- Unterschiedlich formale Notationen, welche einen Aspekt beschreiben, sollten möglichst ähnlich aufgebaut sein und unter Umständen notationelle Überschneidungen aufweisen.

- Es sollten bekannte und nicht gänzlich neue Notationen verwendet werden. Beispielsweise bieten sich hier natürliche Sprachen (als informale Notation), Programmiersprachen, Z oder SDL (als formale Sprache) an.
- Die unterschiedlichen Notationen sollten sich wenn möglich überschneiden, d.h. dieselben teilweise dieselben Sprachkonstrukte verwenden. Diese Forderung vereinfacht die Orientierung und die Integration der verschiedenen Notationen. Beispiel hierfür ist der Pseudocode einer Programmiersprache und die Programmiersprache selber.

Kapitel 4

Überblick über existierende Spezifikationssprachen

In der Vergangenheit wurden zahlreiche Sprachen zur Spezifikation von Anforderungen entwickelt. Das eigentliche Problem einer klar strukturierten Anforderungsmodellierung im Großen bleibt jedoch nach wie vor ungelöst: Zwar finden sich in einigen Ansätzen sinnvolle und notwendige Ideen und Konzepte wieder, die zentrale Problematik jedoch, wie komplexe und umfangreiche Spezifikationen beschrieben und wie bzw. welche Strukturierungsmechanismen hierfür praktikabel eingesetzt werden können, bleibt nach wie vor unbeantwortet.

Spezifikationssprachen lassen sich in die Kategorien *verhaltensorientierte, funktionsorientierte und objektorientierte Ansätze* unterscheiden. Diese Ansätze orientieren sich an präzisen vorgegebenen Paradigmen, welche Regeln, Integritäten und Prinzipien zur semantischen Strukturierung von Spezifikationen vorgeben.

- *Verhaltensorientierte Sprachansätze* strukturieren Systemmodelle konstruktiv anhand des zeitlichen Verhaltens eines Systems. Das System wird durch eine Menge diskreter Systemzustände und einer Übergangsfunktion zwischen diesen Systemzuständen modelliert. Darüber hinaus finden sich Ansätze, welche die explizite Modellierung von kontinuierlichem Verhalten erlauben.
- *Funktionsorientierte Sprachansätze* strukturieren Systemmodelle konstruktiv anhand der Funktionalitäten und Leistungen des Systems, indem Aktivitäten (oder Prozesse) identifiziert werden und die Kommunikation zwischen diesen Aktivitäten beschrieben wird.
- *Objektorientierte Sprachansätze* strukturieren Systemmodelle konstruktiv anhand von Objekten. Jedes Objekt umfaßt einen in sich geschlossenen Teil von Daten, von Funktionalität und von Verhalten des Systems. Eine spezielle Form objektorientierter Ansätze sind die datenorientierten Ansätze, welche größtenteils auf die explizite Modellierung von Verhalten und Funktionalität verzichten.

Es existieren darüber hinaus weitere Ansätze, die sich im wesentlichen nicht an speziellen Paradigmen orientieren und es freistellen, wie Spezifikationen prinzipiell strukturiert werden. Besonders hervorzuheben sind hier die *natürlichsprachlichen Ansätze*:

- *Natürlichsprachliche Ansätze* beschreiben Anforderungen mit Hilfe einer natürlichen Sprache bzw. mit einer Sprache, die auf einer natürlichen Sprache aufbaut.
- Unter den *paradigmenneutralen Ansätzen* werden alle übrigen Ansätze zusammengefaßt, die sich nicht auf vorgegebene Paradigmen festlegen. Diese Ansätze basieren häufig auf einem mathematischen Kalkül und setzen diese durch formale Sprachen um.
- Im folgenden werden zunächst die paradigmenneutralen Ansätze diskutiert. Im Anschluß daran werden die drei Sprachkategorien diskutiert, die definierte Beschreibungsparadigmen vorgeben.

4.1 Natürlichsprachliche Ansätze

Natürlichsprachliche Ansätze umfassen entweder natürliche Sprachen oder Sprachen, die auf diesen aufbauen. Anforderungsspezifikationen werden in der Praxis sehr häufig mit natürlichsprachlichen Ansätzen beschrieben [Sommerville96, S. 122f], da

- eine natürliche Sprache nicht erlernt werden muß
- natürlichsprachliche Spezifikationen in der Regel einfach, direkt und schnell erstellt werden können
- natürliche Sprache von allen beteiligten Personen (Entwickler, Benutzer, Auftraggeber) gleichermaßen beherrscht wird und eine natürlichsprachliche Spezifikation für alle verstehbar ist.

Diese Vorzüge können jedoch nicht darüber hinwegtäuschen, daß die Verwendung von natürlichen Sprachen zur Anforderungsspezifikation sehr problematisch ist. Gründe hierfür liegen zum einen im informalen Charakter einer natürlichen Sprache, zum anderen im Mangel an ausreichenden Gliederungskonzepten:

- Natürliche Sprachen sind *informal*, d.h. die Sprachsemantik bleibt undefiniert. Diese undefinierte Semantik mag zunächst von Vorteil sein, wenn einfache Spezifikationen zu erstellen sind. Die entwickelten «semantikfreien» Spezifikationen sind jedoch hochgradig *interpretierbar* und *mehrdeutig*. Natürlichsprachliche Spezifikationen sind zudem häufig vage und unscharf, da klärungsbedürftige Sachverhalte unpräzise beschrieben und somit ungeklärte Teile nicht offensichtlich werden [Pullman94]. Für die Anforderungsspezifikation sind sie daher als Kommunikationsmittel ungeeignet, da in den meisten Fällen eine Rückkopplung zwischen Ersteller und Leser einer Spezifikation nicht gewünscht ist, da diese ja gerade im Sinne eines Vertrages zu erbringende Leistungen festhalten soll.
- Natürliche Sprachen stellen nur mangelhafte *Mechanismen zur Gliederung* einer Spezifikation zur Verfügung. Mechanismen wie etwa *Absätze*, *Seiten* oder *Kapitel* sind rein syntaktisch und dadurch wenig hilfreich [Hall90]. Natürliche Sprachen sind daher auch wenig

geeignet, Probleme konstruktiv mit Hilfe von Modellen zu beschreiben, da die textuelle Notation und der Mangel an Gliederungskonstrukten keine anschauliche Modellierung zulassen [Glinz98a].

Aufgrund einer einfachen und verständlichen Spezifikationssprache täuschen natürlichsprachliche Spezifikationen den Eindruck einer ebenfalls einfachen und verständlichen Spezifikation vor. Tatsächlich neigen diese Spezifikationen jedoch zu *Widersprüchen, Inkonsistenzen, Vagheiten* und *Mehrdeutigkeiten* und sind zudem schwer prüfbar [Hall90]. Eine Reihe von natürlichsprachlichen Ansätzen versucht, diese Schwachstellen durch geeignete Erweiterungen abzumildern, und zwar im wesentlichen durch *Formalisierung* und durch *Strukturierung* der natürlichen Sprache:

- *Formalisierung* – Natürliche Sprachen werden formalisiert, indem sie restringiert und reglementiert werden mit dem Ziel, Spezifikationen durch eine erhöhte Präzision und durch Minimierung von Mehrdeutigkeiten robuster zu machen. Erreicht werden soll dies durch die Reglementierung von «gefährlichen» Wörtern oder Satzstrukturen (beispielsweise das Verbot von Wörtern/Satzstrukturen mit optionalem oder relativistischem Charakter wie «normal», «effektiv», «kann», «falls gewünscht») [Wilson97]. Einen Schritt weiter gehen Sprachen wie ACE [Schwitter98] oder CLARE [Alshawi92], in denen bestimmten Schlüsselwörtern eine definierte *Ausführungssemantik* zugeordnet wird und die somit in eingeschränktem Maße formal prüfbar oder sogar ausführbar sind.
- *Strukturierung* – Natürliche Sprachen werden durch Richtlinien bezüglich der Dokumentenstruktur ergänzt. Hierbei werden Vorgaben über den strukturellen Aufbau des Dokumentes gemacht sowie reglementiert, auf welche Weise natürlichsprachliche Texte formatiert werden. Beispiele sind etwa Kapitel- und Inhaltsstrukturen nach [IEEE93], [DOD85], [NASA76], [Fairley85, S. 85], das Anlegen von Glossaren [Ortner98], die Zerlegung und Klassifikation von komplexen textuellen Formulierungen in kurze, überschaubare Sätze oder Satzfragmente [Heninger80], [Wilson97] oder der Aufbau von Hypertext-Vernetzungen [McGrew89].

Die Restriktion von natürlicher Sprache ist für die Verwendung als Spezifikationssprache eine Gratwanderung zwischen *Verständlich-* und *Vagheit* auf der einen und *Präzision* und *Unverständlichkeit* auf der anderen Seite: Je stärker natürliche Sprachen restringiert werden, desto weiter entfernen sie sich von einer ursprünglich einfach verständlichen und handhabbaren Sprache. Einfache Sachverhalte können häufig nur umständlich und schwer verständlich formuliert werden. Eine weitere Gefahr verbirgt sich in der Zuweisung von Bedeutungen (etwa von Ablaufsemantik) zu bestimmten Schlüsselwörtern oder Schlüsselstrukturen, wie beispielsweise bei der Sprache ACE. Dort wird eine «und»- Verknüpfung zweier Sätze als sequentielle Abfolge zweier Aktivitäten interpretiert [Schwitter98, S.170f]. Für Ausdrücke der Art «wenn … dann», «falls … dann», «… oder …» wird eine entsprechende *Ablaufsemantik* zugeordnet. Nach außen hin erscheint die Spezifikation natürlichsprachlich, tatsächlich jedoch haben einzelne Sprachelemente definierte formale Bedeutungen. Diese sind dem Entwickler oder Leser

nicht offensichtlich, da er im Glauben gelassen wird, er arbeite mit einer natürlicher Sprache. Formale Sprachkonstrukte sollten daher, wie es in teilformalen und formalen Sprachen üblich ist, als solche auch notationell kenntlich gemacht werden [Hall90]. Sprachrestriktionen der einfacheren Art sind in eingeschränktem Maße [Wilson98] durchaus sinnvoll. Ein Teil der Probleme natürlichsprachlicher Spezifikationen läßt sich in der Tat durch Beschränkung bestimmter Wörter oder Strukturen vermeiden. Insbesondere wird durch diese Reglementierung der Entwickler genötigt, Sachverhalte genauer abzuklären und detailgetreuer zu formulieren. Sie sind jedoch weniger geeignet, Widersprüche oder Spezifikationslücken zu vermeiden oder das Problem der Mehrdeutigkeit in den Griff zu bekommen. Strukturierungsvorschläge der vorgestellten Art sind generell eine nützliche und sinnvolle Hilfestellung für die Entwicklung von Spezifikationen. Sie beheben jedoch nicht die inhärenten Mängel natürlicher Sprachen, keine ausreichend sprachbasierten Mechanismen zur Strukturierung von Anforderungen anzubieten und die Bildung einer sinnvollen Gliederung der Spezifikation konstruktiv zu unterstützen. Die Wahl, die Umsetzung und die Wartung der Gliederung einer natürlichsprachlichen Spezifikation liegt daher vollständig auf Seiten des Entwicklers.

4.2 Paradigmenneutrale Sprachansätze

Es existiert eine Vielzahl unterschiedlicher Ansätze, die sich in die *paradigmenneutrale* Sprachklasse einordnen lassen. Die meisten dieser Ansätze sind hochgradig deskriptiv, indem sie auf Grundlage eines mathematischen Kalküls Anforderungen ausschließlich durch eine formale Sprache beschreiben. Beispiele für solche paradigmenneutrale Ansätze sind Z [Woodcock96], VDM [Bjorner78] bzw. diverse axiomatische Ansätze [Sommerville96]. Diese Ansätze sind jedoch aus folgenden Gründen zur Anforderungsspezifikation im Großen nicht geeignet:

- Die Sprachansätze sind zumeist formal und mathematisch basiert und daher wenig flexibel, wenn Anforderungen einfach und wenig präzise beschrieben werden sollen (siehe auch die Forderung nach einem variablen Formalisierungsgrad, Seite 31)
- Die Notationen der Ansätze sind häufig nur schlecht erlernbar und auch schlecht verständlich und kryptisch gehalten. Entsprechend leidet darunter die Verständlichkeit und die Prüfbarkeit.
- Durch das Fehlen von vorgegebenen Sprachparadigmen existiert auch gleichzeitig keine klare Vorgabe, wie die Spezifikationen strukturiert und gegliedert sind. Umfangreiche und komplexe Problembeschreibungen erfordern jedoch gerade eine solche klare Strukturierung. Zwar ist es möglich, für die Beschreibung ein geeignetes Paradigma hinzuzunehmen (beispielsweise können Anforderungen objektorientiert mit Hilfe von Z modelliert werden). Diese Sprachen unterstützen jedoch diese Paradigmen nicht und kontrollieren im speziellen nicht die entsprechende Paradigmeneinhaltung.

Paradigmenneutrale Ansätze werden daher an dieser Stelle nicht weiter ausgeführt. Ein Überblick über paradigmenneutrale und formale Sprachansätze findet sich beispielsweise in [Sommerville96] oder [Hall90]. In den nächsten Kapiteln werden die wichtigsten Vertreter der verbleibenden Ansätze vorgestellt sowie deren Stärken und Schwachstellen diskutiert.

4.3 Verhaltensorientierte Sprachansätze

Man unterscheidet zwischen *diskreten* und *kontinuierlichen* Ansätzen zur Verhaltensmodellierung. *Diskrete Ansätze* modellieren das zeitliche Verhalten eines Systems, indem aufzählbare, diskrete Zustände des Systems modelliert werden. Das Systemverhalten wird durch Zustandsübergänge spezifiziert, d.h. wann und unter welchen Bedingungen ein Zustandsübergang erfolgt und welche funktionalen Auswirkungen mit diesem Zustandsübergang verbunden sind. *Kontinuierliche Ansätze* modellieren das zeitliche Verhalten eines Systems, indem skalare Größen des Systems und die Wirkungen zwischen diesen Systemgrößen durch stetige Funktionen modelliert werden. Üblicherweise werden in der Anforderungsspezifikation diskrete Modellierungsansätze verwendet. Prinzipiell können Verhaltensprobleme jederzeit auch kontinuierlich modelliert werden. Eine diskrete Modellierung ist jedoch häufig sinnvoller, da durch die Verwendung digitaler Systeme Informationen zwangsläufig diskretisiert werden müssen.

In diesem Kapitel werden im Detail die folgenden Ansätze zur Verhaltensmodellierung vorgestellt: *Zustandsautomaten*, *Statecharts*, *Petrinetze*, *SREM*, *ASTRAL* und *SDL*. Kontinuierliche verhaltensorientierte Ansätze werden hier nur am Rande durch Nennung der *System-Dynamics-Modelle* vorgestellt. Die Modellierung kontinuierlichen Verhaltens ist für die Anforderungsspezifikation im allgemeinen nur von geringer Bedeutung und wird üblicherweise nur in speziellen technischen Disziplinen eingesetzt.

4.3.1 Zustandsautomaten

Modellierungsart • *Integrierte Modellierung* • *Grafisch/textuelle Notation*	***Primäre Struktur*** • *Zustands/Übergangsgraph*	***Formalitätsgrad*** • *Graphstruktur formal* • *Übergänge variabel formal*
Datenmodellierung • *keine explizite Datenmodellierung*	***Verhaltensmodellierung*** • *automatenbasiert* • *Grundlage: Automatentheorie* • *Kommunikation über Nachrichten oder gemeinsame Variablen*	***Funktionale Modellierung*** • *wird beschrieben durch natürliche Sprache, Pseudocode, formale Sprachen (Bsp: Programmiersprachen), ….*

Ein Zustandsautomat ist definiert durch eine endliche Menge von elementaren Zuständen, eine Menge von *Zustandsübergängen*, von *Ereignissen* und von *Aktionen*. Sie basiert auf einer ausgereiften mathematischen Theorie, der *Automatentheorie* [Hopcroft90]:

- Ein *elementarer Zustand* repräsentiert einen definierten Zeitabschnitt, in dem das System bestimmte Eigenschaften und Verhaltensweisen zeigt und auf bestimmte Eingaben reagiert.
- Ein *Zustandsübergang* spezifiziert einen zeitlosen Übergang eines Zustandes in einen anderen.
- Ein *Ereignis* ist ein Zeitpunkt, in dem ein spezieller Sachverhalt eintritt.
- Eine *Aktion* ist eine Funktionalität, die zum Zeitpunkt des Zustandsüberganges angestoßen und daraufhin ausgeführt wird.

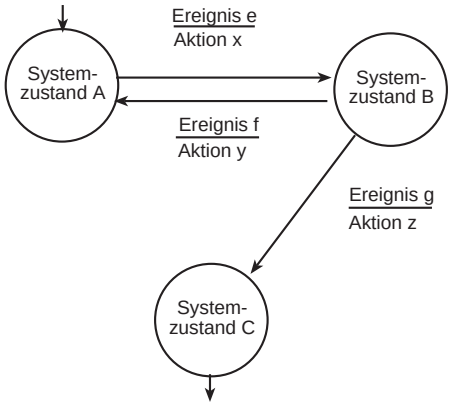


Abb. 4: Beispiel eines Zustandsautomaten

Abbildung 4 zeigt skizzenhaft ein durch einen Zustandsautomaten modelliertes System mit den Systemzuständen A, B und C (dargestellt durch Kreise). Die Zustandsübergänge werden als Pfeile zwischen den jeweiligen Zuständen gezeichnet, die Ereignisse e, f, g und die Aktionen x, y und z werden an die entsprechenden Zustandsübergänge gebunden. Der Systemzustand A ist Anfangszustand, d.h. der zu Beginn gültige Zustand. Der Systemzustand C ist der Endzustand, d.h. der Zustand, an dem das Systemverhalten terminiert. Ein Zustandsautomat befindet sich zu jeder Zeit in genau einem Zustand. Die Zustandsübergänge sind somit zeitfrei. Ein Übergang erfolgt, wenn das entsprechende Ereignis eingetreten ist. Ist dies der Fall, wird die entsprechende Aktion angestoßen und ein neuer Zustand erreicht. Im folgenden wird beispielhaft für den Zustandsautomaten in Abbildung 4 eine Ereignisfolge erläutert: Zu Beginn befindet sich der Zustandsautomat in Zustand A. In dem Moment, wo das Ereignis e eintritt, wechselt das System von Zustand A in den Zustand B. Gleichzeitig wird die Aktion x ausgeführt. Tritt e zu einem späteren Zeitpunkt erneut ein, so wird es ignoriert (und nicht für einen späteren Zeitpunkt gespeichert). B wird erst wieder verlassen, wenn entweder das Ereignis f oder g ausgelöst wird.

4.3.2 Statecharts

Modellierungsart • *Integrierte Modellierung* • *Grafisch/textuelle Notation*	***Primäre Struktur*** • *Zustandshierarchie*	***Formalitätsgrad*** • *Graphstruktur formal* • *Übergänge variabel formal*
Datenmodellierung • *keine explizite Datenmodellierung*	***Verhaltensmodellierung*** • *automatenbasiert* • *Grundlage: Automatentheorie* • *Kommunikation: Broadcast-Nachrichten*	***Funktionale Modellierung*** • *variabel formal durch natürliche Sprache, Pseudocode, formale Sprache (Bsp: Programmiersprachen), ….*

Statecharts sind spezielle Zustandsautomaten, deren Zustände hierarchisch und parallel zerlegbar sind. Sie wurden erstmals von [Harel87] vorgestellt mit der Zielsetzung, durch *hierarchische* und *parallele* Verhaltensbeschreibungen auch umfangreiche Problemstellungen verständlich beschreiben zu können. Neben *elementaren Zuständen* existieren zwei weitere komplexe Zustandsarten:

- Das *hierarchisch geschachtelte Statechart* enthält eine eingeschachtelte Zustandsbeschreibung
- Das *parallele Statechart* setzt sich zusammen aus mehreren vollständigen Zustandsautomaten, die quasi parallel zueinander stehen (die parallelen Zustandsautomaten haben zwar keine expliziten Zustandsübergänge untereinander, sie können sich jedoch durch Ereignisse synchronisieren).

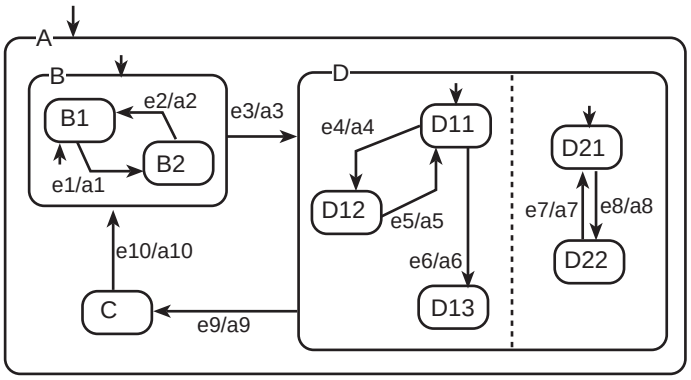


Abb. 5: : Beispiel eines hierarchisch verschachtelten Statecharts

In Abbildung 5 sind zu Beginn die Zustände A, B und B1 aktiv. Wird das Ereignis e1 oder e2 ausgelöst, verhält sich ein *Statechart* analog zu einem Zustandsautomaten: Bei Auslösung des Ereignisses e3 wird der Zustand B und B1 bzw. B2 verlassen und der Zustand D, D11 und D21 aktiv. Sobald das Ereignis a9 ausgelöst wird, werden die lokalen Zustände in D inaktiv und der Zustand C aktiv. Statecharts haben dieselbe Ausdrucksmächtigkeit wie Zustandsautomaten. Sowohl die hierarchische Verschachtelung als auch die parallelen Beschreibungen lassen sich äquivalent auch mit einem Zustandsautomaten modellieren (jedoch entsprechend umständlich und ausführlich).

4.3.3 Petrinetze

Modellierungsart • *Integrierte Modellierung* • *Grafisch/textuelle Notation*	***Primäre Struktur*** •*Petrinetz-Graph (Stellen evtl. hierarchisch geschachtelt)*	***Formalitätsgrad*** • *Graphstruktur formal* • *Transitionenspezifikation variabel formal*
Datenmodellierung • *keine explizite Datenmodellierung*	***Verhaltensmodellierung*** • *Grundlage: Graphen- und Petrinetztheorie* • *Kommunikation über Nachrichten oder gemeinsame Variablen*	***Funktionale Modellierung*** • *variabel formal durch natürliche Sprache, Pseudocode, formale Sprache (Bsp: Programmiersprachen), ….*

Petrinetze [Petri62] basieren ebenso wie Zustandsautomaten auf einer fundierten mathematischen Theorie, der Graphentheorie und der Petrinetztheorie. Die Grundidee ähnelt derer von Zustandsautomaten, Petrinetze sind jedoch komplexer und ermöglichen eine bessere Modellierung paralleler Abläufe. Ein Petrinetz ist ein gerichteter Graph mit zwei sich abwechselnden Knotenarten:

- Eine *Stelle* ist Träger eines Systemzustandes und speichert Informationen mit Hilfe von *Marken*. Die *Marken* und ihre Verteilung über die *Stellen* des Modells hinweg modellieren den aktuellen Systemzustand.
- Eine *Transition* legt fest, wie sich Marken im Netz fortbewegen können. Sie entnehmen *Marken* aus *Stellen* und transportieren sie zeitlos zu einer neuen *Stelle*. Einer *Transition* wird analog zu Zustandsautomaten durch ein *Ereignis* und durch diverse *Aktionen* näher spezifiziert.

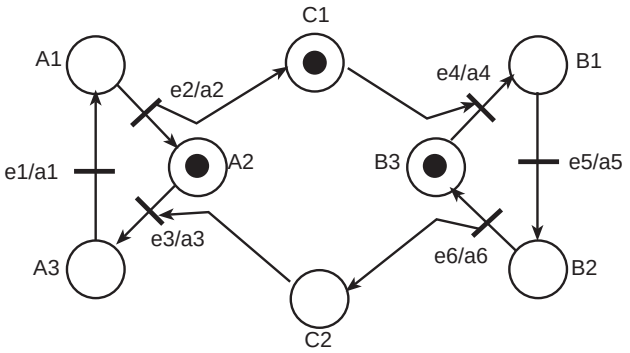


Abb. 6: Beispiel eines Petrinetzes

Die Initialbelegung der *Marken* beschreibt den Startzustand des Petrinetzes. Eine *Transition* feuert, wenn alle Eingangsstellen mit *Marken* belegt, alle Ausgangsstellen markenfrei sind und das der *Transition* zugeordnete Ereignis ausgelöst wurde. Wenn eine *Transition* feuert, werden alle Marken aus den Eingangsstellen entnommen und alle Ausgangsstellen mit Marken belegt. Verschiedene *Transition*en können gleichzeitig feuern, haben sie gemeinsame Eingangsstellen, wird die feuernde *Transition* nichtdeterministisch ausgewählt. In Abbildung 6 hat das Petrinetz *Marken* in den *Stellen* C1, B3 und A2. Von der Markenverteilung kann nur die *Transition* feuern, welche das *Ereignis* e4 erwartet. Wird dieses *Ereignis* ausgelöst, werden zwei *Marken* aus

B3 und C1 entfernt und eine *Marke* in B1 gelegt. Nachdem die *Ereignisse* e5 und e6 ebenfalls ausgelöst wurden, geht eine *Marke* von B1 auf B2 und von B2 auf B3 und C2 über.

Aufbauend auf den klassischen Petrinetzen existieren Erweiterungen, die *Stellen* mit mehreren *Marken* zulassen. Die ist beispielsweise dann von Vorteil, wenn *Marken* mit erzeugbaren und konsumierbaren Entitäten oder Leistungen assoziiert werden. Weiterhin ist es möglich, Petrinetz-Transitionen zusätzlich durch Prädikate zu attributieren. Diese Prädikate haben eine ähnliche Bedeutung wie Übergangsbedingungen bei Zustandsautomaten. Analog dazu kann ein Petrinetz-Übergang nur erfolgen, wenn das Prädikat erfüllt ist (und entsprechende Marken vorhanden sind). Darüber hinaus existiert eine Reihe von Erweiterungsvorschlägen, um Petrinetz-Systeme durch Stellenzerlegungen hierarchisch zu strukturieren. Problematisch ist hier jedoch das weitestgehend ungeklärte Zusammenspiel zwischen den Marken des Detail- und des Grobpetrinetzes und die Frage, welche Marken des Detailpetrinetzes wann und wie zu entfernen sind.

4.3.4 RSML

Modellierungsart • *Integrierte Modellierung* • *Grafisch/textuelle Notation*	***Primäre Struktur*** • *Zustandshierarchie (ähnlich einer Statechart-Hierarchie)*	***Formalitätsgrad*** • *formal*
Datenmodellierung • *durch globale Variablen*	***Verhaltensmodellierung*** • *automatenbasiert* • *ähnlich der Statechart-Semantik*	***Funktionale Modellierung*** • *formal durch die Modellierung von Zustandsübergängen durch Wahrheitstabellen*

Die *Requirements State Machine Language* (RSML) baut auf den Statecharts auf und erweitert diese durch syntaktische und semantische Konstrukte. Diese haben das Ziel, gegenüber den *Statecharts* eine ausführlichere und anschaulichere funktionale Modellierung zu ermöglichen. Zentrales Sprachelement ist hierbei die *Und/Oder-Tabelle*, eine Wahrheitstabelle, mit deren Hilfe Zustandsübergänge formal spezifiziert werden (siehe [Heimdahl96], [Leveson94]). Weiterhin ist es in RSML möglich, *Blöcke* zu definieren. Innerhalb der Blöcke ist eine synchrone Kommunikation möglich, Kommunikation zwischen Blöcken ist jedoch stets asynchron (bei den ursprünglichen Statecharts wird ausschließlich ein gemeinsames Broadcasting-Konzept zur Verfügung gestellt).

4.3.5 SREM

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *Textuelle Notation*	• *Erweiterter Zustands/Übergangs-Graph*	• *Graphstruktur formal* • *ALPHA-Funktionen formal* • *Übergänge und diverse Annotationen variabel formal*
Datenmodellierung	***Verhaltensmodellierung***	***Funktionale Modellierung***
• *durch globale Variablen*	• *automatenbasiert* • *Grundlage: Automatentheorie* • *Kommunikation über Nachrichten oder über gemeinsame Variablen*	• *textuell formal durch ALPHA-Funktionen*

Die *Software Requirements Engineering Methodology* (SREM) wurde in den siebziger Jahren entwickelt (siehe [Alford82], [Alford85]). Sie basiert im wesentlichen auf eine Erweiterung von Zustandsautomaten zur Modellierung von Nebenläufigkeiten. Zudem wurde die Notation von SREM durch zahlreiche Elemente erweitert, um Eingaben, Ausgaben und Aktivitäten besser zu visualisieren. Aktivitäten werden in SREM-Modellen durch ALPHA-Funktionen modelliert. Diese ALPHA-Funktionen können weiter bis auf die Stufe von Prozeduren und Prozesse zerlegt werden.

4.3.6 ASTRAL

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *Textuelle Notation*	• *textuell durch eine hierarchische Prozeßstruktur* • *Prozeßbeschreibung durch Zustands/Übergangsgraphen*	• *formal*
Datenmodellierung	***Verhaltensmodellierung***	***Funktionale Modellierung***
• *durch lokale und globale Prozeßvariablen*	• *automatenbasiert* • *Grundlage: Automatentheorie* • *Implizite Kommunikation über Nachrichten oder über exportierte Variablen*	• *textuell durch eine formale Prozeßbeschreibung*

ASTRAL ist eine formale Spezifikationssprache, die vorwiegend für die Spezifikation von technischen Systemen und von Echtzeitsystemen entwickelt wurde [Coen93/95/97]. Ein ASTRAL-Systemmodell besteht aus einer Menge von Prozeßbeschreibungen (in Astral: process specification) und einer globalen prozeßübergreifenden Beschreibung (in Astral: global specification). Ein Prozeß wird auf verschiedenen Granularitätsstufen textuell durch Zustandsautomaten modelliert. Für die Beschreibung der Zustandsautomaten wird eine spezielle, formal getypte Beschreibungssprache verwendet. Die Kommunikation basiert auf einem impliziten Nachrichtenparadigma, bei dem von Prozessen gesendete Nachrichten für alle Prozesse sichtbar und prinzipiell verwertbar sind (*Broadcast*-Konzept). Die Kommunikation von Prozessen wird weiterhin durch *Export*- und *Importschnittstellen* spezifiziert.

4.3.7 SDL

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *wahlweise textuelle oder grafische Notation*	• *textuell durch eine hierarchische Prozesstruktur* • *Prozessbeschreibung durch Zustands/Übergangsgraphen*	• *textuelle Beschreibung formal, grafische Notation teilformal*
Datenmodellierung	***Verhaltensmodellierung***	***Funktionale Modellierung***
• *durch lokale Prozeßvariablen, realisiert durch abstrakte Datentypen*	• *automatenbasiert* • *Grundlage: Automatentheorie* • *explizite Kommunikation zwischen Blöcken über Nachrichten oder über exportierte Variablen* • *Prozeßkommunikation innerhalb von Blöcken asynchron*	• *textuell durch eine formale axiomatische Beschreibung*

Die *Specification and Description Language* (SDL) [Belina91] ist ähnlich konzipiert und aufgebaut wie die zuvor beschriebene ASTRAL-Sprache. SDL verwendet jedoch in der Grobbeschreibung ein explizites Nachrichtenkonzept: Die Interaktion von Nachrichten (in SDL: signals) wird durch *Kanäle* modelliert, d.h. durch Angabe konkreter Nachrichtenpfade. Aktivitäten werden hier durch *Blöcke* modelliert. Die Blöcke selber werden durch *Prozesse* spezifiziert, die sich wiederum asynchron verständigen können. SDL stellt weiterhin neben einer textuellen eine semantisch äquivalente grafische Notation zur Verfügung, welche sich ineinander überführen lassen.

4.3.8 Kontinuierliche verhaltensorientierte Ansätze

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *grafische Notation*	• *grafisch durch Attribut/Flußgraphen*	• *formal*
Datenmodellierung	***Verhaltensmodellierung***	***Funktionale Modellierung***
• *durch globale Variablen/Attribute*	• *Modellierung einer expliziten Attribut/Variablenveränderungen durch kontinuierliche Datenflüsse*	• *variabel formal durch Berechnungsfunktionen*

Kontinuierliche verhaltensorientierte Ansätze basieren im wesentlichen auf einer Problembeschreibung mit Hilfe mathematischer Funktionen und auf *linearen* und *nichtlinearen Differentialgleichungssystemen*. Kontinuierliche Verhaltensmodelle werden an dieser Stelle nur am Rande erwähnt, da sie in der Regel deutlich weniger komplex als zustandsorientierte Verhaltensmodelle sind und die Probleme daher weniger im Bereich Anschaulichkeit und Strukturierung und mehr im Bereich mathematische Analyse und von Simulation liegen. Beispiel eines kontinuierlichen Modellierungsansatzes ist die *System-Dynamics*-Modellierung [Forrester71], bei dem skalare Systemgrößen, Quellen, Senken, Konstanten sowie Berechnungsfunktionen durch einen Graph modelliert werden.

4.3.9 Bewertung

Verhaltensorientierte Ansätze eignen sich vorwiegend für Problemstellungen, bei denen Verhaltensaspekte im Vordergrund stehen. Dies ist häufig der Fall bei Systemen mit hohen Echtzeitanforderungen, bei verteilten und parallelen Systemen sowie bei eingebetteten Systemen [Sommerville96, S.239f]. Allgemein jedoch reicht das Verhaltensparadigma nicht aus, um große Systemmodelle anschaulich und strukturiert zu modellieren. Häufig werden daher Zustandsautomaten, Statecharts oder Petrinetze mit anderen Sprachparadigmen kombiniert. In der folgenden Tabelle werden Vorzüge und Defizite aufgeführt sowie die einzelnen Ansätze abschließend bewertet.

Ansatz	**Vorzüge**	**Nachteile**	**Bemerkungen**
Zustands-automaten	*einfache, leicht erlernbare Notation, im kleinen verständlich*	*keine Abstraktionsmechanismen und keine Hierarchisierung (Dekomposition) möglich. Keine Unterstützung von Parallelität*	*nur geeignet für kleine und kurze Problemstellungen mit hohem Verhaltensanteil und wenig Daten*
Statecharts	*erlaubt die Modellierung von Parallelität, hierarchische Zustandsmodellierung*	*fehleranfälliges Broadcast-Nachrichtenkonzept, unzureichende Datenmodellierung*	*Geeignet für einfach und komplexe Systeme mit geringen Datenbeständen und diskreten Verhaltensweisen. Statecharts sind jedoch nicht zur Modellierung von großen Datenbeständen geeignet.*
Petrinetze	*sehr gute Unterstützung paralleler Prozesse*	*Hierarchische Zustandszerlegung nicht möglich bzw. problematisch umzusetzen. Komplexe und teilweise schwer verständliche Notation (im Vergleich zu automatenbasierten Ansätzen)*	*Geeignet zur Modellierung hochgradig paralleler Prozesse. Ansonsten sind Petrinetze im Großen ungeeignet, da sie schlecht zerlegbar und häufig schwer verständlich sind.*
RSML	*siehe Statecharts. Präzise und anschauliche Modellierung von Zustandsübergängen*	*siehe Statecharts. Notation vollständig formal. Keine Unterstützung eines variablen Formalitätsgrades. Besseres und verständlicheres Kommunikationskonzept als Statecharts.*	*RSML ähneln den Statecharts. Sie erlauben jedoch nur eine formale und ausführliche Spezifikation von Zustandsübergängen.*
SREM	*einfache, leicht erlernbare Notation*	*Nur eingeschränkte Modellierung von Nebenläufigkeiten möglich. Hierarchische Zerlegungen werden nicht unterstützt.*	*SREM bietet keine Unterstützung für eine hierarchische Zustandszerlegung an und ist daher nur eingeschränkt für umfangreiche Problemstellungen geeignet.*

ASTRAL	*Klare und präzise Zerlegung von Prozessen in Teilprozesse. Präzise Spezifikation von Export- und Importschnittstellen von Prozessen. Echtzeitunterstützung*	*ASTRAL-Modelle sind textuell Die grafische Visualisierung von Grobstrukturen wäre jedoch sinnvoll. Fehleranfälliges Broadcast-Nachrichtenkonzept. Keine Unterstützung eines variablen Formalitätsgrades*	*ASTRAL ist eine formale Spezifikationssprache, die im besonderen für die Modellierung von technischen Systemen und von Echtzeitsystemen geeignet ist. Durch den formalen Charakter ist sie jedoch für Systeme im Großen weniger geeignet.*
SDL	*Aufwendige aber fehlerrobuste Modellierung von Kommunikation durch Nachrichtenkanäle.*	*Verwendung einer schwer verständlichen formalen axiomatischen Notation.*	*Ebenso wie Astral ist SDL eine «technische» und echtzeitbezogene Sprache, die ebenso Schwächen bei umfangreichen Problemstellungen aufweist.*

Statecharts und Statecharts-ähnliche Ansätze erscheinen am besten geeignet, um große Systemmodelle zustandsbasiert zu zerlegen. Ausnahme ist eine Systemmodellierung durch Petrinetze, in denen hochgradig parallele Prozesse anschaulich modelliert werden können.

Die Ansätze RSML, ASTRAL und SDL sind formale Spezifikationsansätze. ASTRAL zeichnet sich durch eine präzise Modellierung der Export- und Importschnittstellen von Prozessen aus. Dadurch werden unerwünschte und nicht sichtbare Seiteneffekte bei der Prozeßkommunikation verhindert. Die präzise formale Beschreibung erlaubt zudem Konsistenzprüfungen der hierarchischen Beschreibungen und erschwert dadurch Modellinkonsistenzen. SDL verwendet ein explizites Nachrichtenkonzept, welches zwar einen erhöhten Aufwand bei Modellerstellung und -pflege erfordert, dafür aber fehlerrobuster ist. Bei beiden Ansätzen wird jedoch klar, welche Nachteile formale Sprachen generell haben: Die Erstellung und Pflege von Modellen ist sehr aufwendig. Als alleiniges Beschreibungsmittel sind sie für Systemmodellierungen im Großen daher nicht geeignet.

4.4 Funktionsorientierte Sprachen

Funktionsorientierte Sprachen zerlegen ein Problem in eine Menge interagierender Komponenten mit jeweils klar definierter Funktionalität. Zustand und Daten eines Systems werden im wesentlichen zentralisiert modelliert und sind für alle Komponenten sicht- und änderbar. Darüber hinaus haben Komponenten einen lokalen, für andere Komponenten nicht sichtbaren Zustandsraum, der jedoch nur innerhalb der Ausführung der Komponente gültig ist und daher Informationen nur zeitlich begrenzt halten kann (siehe Abbildung 7). Die Zerlegung eines funktionsorientierten Modells ergibt sich aus den Benutzungszusammenhängen zwischen den

funktionalen Komponenten und führt zu einer Benutzungshierarchie (siehe auch Kapitel 3.3, S. 28).

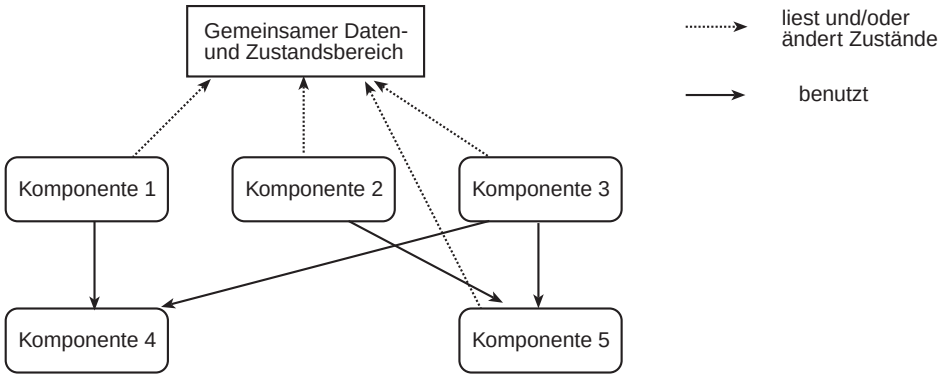


Abb. 7: Beispiel eines funktional zerlegten Systemmodells

Im folgenden Kapitel werden mit der *Strukturierten Analyse*, SADT, STATEMATE und JSD die wichtigsten Vertreter des funktionalen Paradigmas vorgestellt.

4.4.1 Strukturierte Analyse

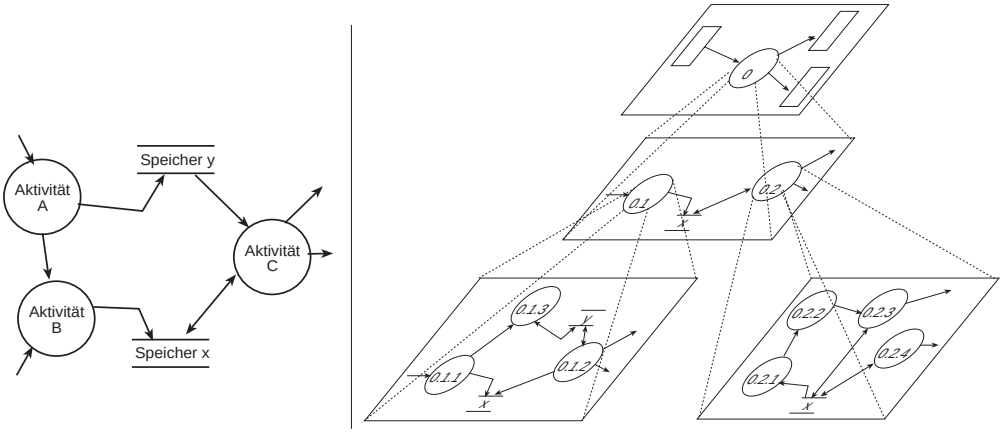
Modellierungsart • *Integrierte Modellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Funktionale Dekomposition durch hierarchische Datenfluß-diagramme*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch teilglobale Variablen (sog. Speicher, die von allen Aktivitäten sowie deren Subaktivitäten sichtbar sind)*	***Verhaltensmodellierung*** • *indirekt durch Modellierung von Datenflüssen* • *auf Detailebene durch Pseudo-code*	***Funktionale Modellierung*** • *durch Aktivitäten und Subaktivitäten sowie auf Detailebene durch Pseudocode (Mini-Spezifikationen)*

Die *Strukturierte Analyse* (SA) nach [DeMarco78] und [Gane79] baut auf dem Grundprinzip von hierarchisch zerlegbaren Datenfluß-Modellen auf. Ein SA-Modell besteht aus einem zentral stehenden hierarchischen Datenflußmodell, welches durch eine Reihe von Detailspezifikationen durch ein *Datenlexikon* (beschreibt die Strukturen der verwendeten *Datenflüsse* und *Speicher*) und *Mini-Spezifikationen* angereichert und präzisiert wird.

- Ein Datenfluß-Modell beschreibt, wie Eingabedaten durch eine Folge von funktionalen Transformationen, den *Aktivitäten*, in Ausgabedaten transformiert werden.
- *Datenflüsse* modellieren die Übergabe von Datensätzen zwischen *Aktivitäten*. Eine *Aktivität* arbeitet dann, wenn die von ihr benötigten Datenflüsse eintreffen. Die Resultate einer *Aktivität* sind wiederum Datensätze, die an andere *Aktivitäten* über Datenflüsse weitergegeben werden.

Daten können zudem innerhalb des Systems in *Speicher* geschrieben und von denen gelesen werden. Abbildung 8 zeigt ein Beispiel eines elementaren Datenflußdiagramms (a) und eines hierarchisch geschachtelten Datenflußmodells (b): *Aktivitäten* werden durch Kreise, *Speicher*

durch zwei waagrechte Balken und Datenflüsse durch Einfach- oder Doppelpfeile dargestellt. In Abbildung 8 (a) sind drei Aktivitäten A, B, und C und Speicher X und Y aufgeführt. A und B haben Schreibzugriff auf X bzw. Y, C Lesezugriff auf Y und Lese- und Schreibzugriff auf X. Weiterhin existiert ein Datenfluß von Aktivität A zu B.



(a) (b)

Abb. 8: Beispiel eines Datenfluß-Diagrammes (a) und eines hierarchischen Datenflußmodells (b)

Datenflußmodelle in ihrer ursprünglichen Form beschreiben ausschließlich die Systemfunktionen und die jeweils benötigten Daten. Der Steuerfluß selber, d.h. in welcher zeitlichen Abfolge die Aktivitäten ausgeführt werden, wird nicht explizit modelliert. [Hatley87] und [Ward85] entwickelten jedoch auf die ursprüngliche *Strukturierte Analyse* aufbauende Echtzeit-Erweiterungen, welche die Modellierung zeitlicher Abfolgen erlauben.

Die *Aktivitäten* eines Datenflußmodells werden durch weitere eingeschachtelte Datenflußmodelle präzisiert. Abbildung 8 (b) zeigt ein Beispiel eines hierarchischen Datenflußmodells: Auf oberster Ebene steht ein Kontextdiagramm, welches Kommunikation und Datenaustausch zwischen System (als Kreis dargestellt) und Repräsentanten der Umgebung (als Rechtecke) modelliert. In einer Datenflußmodell-Hierarchie müssen eine Reihe von Konsistenzbedingungen erfüllt werden. Beispielsweise müssen sich Datenflüsse auf höherer Ebene tieferliegenden Ebenen wiederfinden. Ein *Speicher,* der in tieferliegenden verschiedenen Diagrammen gemeinsam benutzt wird, muß auch in einer höheren Hierarchieebene aufgenommen werden (im Beispiel der *Speicher* X). Elementare *Aktivitäten,* die nicht weiter zerlegbar sind, werden in *Mini-Spezifikationen* durch einen programmiersprachenähnlichen Pseudocode modelliert. Das *Datenlexikon* schließlich spezifiziert die Struktur der *Speicher* und der *Datenflüsse* mit Hilfe regulärer Ausdrücke.

In der von [Yourdon89] vorgeschlagenen Erweiterung werden parallel zum Datenflußmodell die Daten und Datenstrukturen durch Entity-Relationship-Modelle modelliert. Da jedoch die Zusammenhänge zwischen Datenfluß- und Entity-Relationship-Modell nicht ausreichend

geklärt sind, handelt es sich nicht um ein Modell, sondern eher um zwei potentiell widersprüchliche Aspektmodelle.

4.4.2 SADT

Modellierungsart • *Aspektmodellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Hierarchische Datenflußdiagramme* • *Hierarchische Datendiagramme*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch hierarchische Datendiagramme (sog. Datagramme)*	***Verhaltensmodellierung*** • *indirekt durch Modellierung von Datenflüssen* • *auf Detailebene durch Pseudocode*	***Funktionale Modellierung*** • *durch Aktivitäten und Subaktivitäten* • *auf Detailebene durch Berechnungsvorschriften*

Die S*tructured Analysis and Design Technique* (SADT) wurde Mitte der siebziger Jahre von [Ross77] vorgestellt. Die Konzepte von SADT ähneln denen der *strukturierten Analyse*. Zusätzlich werden hier Aktivitäten näher charakterisiert und Daten nicht innerhalb des Datenflußmodells sondern separat modelliert. Eine *Aktivität* wird zusätzlich zu den ein- und ausgehenden Datenflüssen durch *Steuervariablen* (engl. control parameters) und durch *Berechnungsvorschriften* (engl. mechanisms) spezifiziert.

Das hierarchische Datenflußmodell wird in SADT als *Aktigramm* bezeichnet. Die Daten des Systems werden in *Datagrammen* modelliert, die optisch den Aktigrammen ähneln. Die funktionalen Transformationen zwischen Daten werden als Verbindungen zwischen Daten (dargestellt durch Rechtecke) modelliert. *Akti-* und *Datagramme* sind zwei vollständig redundante Modellarten. Die zugrundeliegende Idee ist eine doppelte, redundante Modellierung von Daten und Aktivitäten, um dadurch Widersprüche und Schwachstellen zwischen den beiden Sichtweisen aufzudecken.

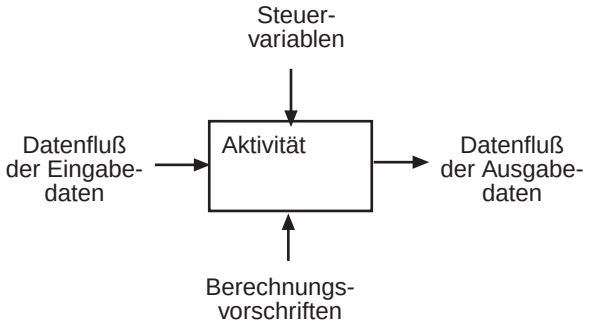


Abb. 9: Beschreibung einer SADT-Aktivität

Beispiel: Für die Aktivität zur Berechnung der Funktion Y=(A*X + B) ist X die Eingangsgröße, Y die Ausgangsgröße, A und B sind Steuervariablen, die Berechnungsvorschrift sind die mathematischen Grundoperationen (*, +, =).

4.4.3 STATEMATE

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *textuelle und grafische Notation*	• *Hierarchische Datenflußdiagramme* • *Sekundäre Struktur: Hierarchische Zustandsautomaten*	• *teilformal*
Datenmodellierung	***Verhaltensmodellierung***	***Funktionale Modellierung***
• *durch teilglobale Variablen (siehe Strukturierte Analyse)*	• *durch Statecharts*	• *durch Aktivitäten und Subaktivitäten sowie auf Detailebene durch Pseudocode (Mini-Spezifikationen)*

STATEMATE wurde Anfang der achtziger Jahre entwickelt [Harel96a]. Der Ansatz baut ebenso wie SA auf dem Konzept der hierarchischen Datenflußmodellierung auf. Mittelpunkt der Modellierung ist ebenfalls ein hierarchisches Datenflußmodell (in Statemate als *activity chart* bezeichnet), in welchem *Aktivitäten*, *Speicher* und *Datenflüsse* beschrieben werden. Das Verhalten von *Aktivitäten* kann jedoch zusätzlich durch *Statecharts* (siehe Kapitel 4.3.2) hierarchisch modelliert werden. Die Ereignisse, welche die Zustandsübergänge in den Statecharts auslösen, werden als spezielle Datenflüsse mit Steuerinformationen modelliert. Statemate stellt darüber hinaus ein Moduldiagramm zur Verfügung, indem aufbauend auf dem Datenflußmodell entwurfsspezifische Aussagen modelliert werden. Das Datenflußmodell selber wird analog zur *Strukturierten Analyse* durch *Mini-Spezifikationen* und *Datenlexika* weiter verfeinert. Abbildung 10 zeigt eine Übersicht über die Modellierungssprache von *STATEMATE* und die Zusammenhänge zwischen den verwendeten Modellen. Das *Activity-Chart* beinhaltet Aktivitäten, die analog zur Strukturierten Analyse hierarchisch durch weitere Datenflußdiagramme (dargestellt durch eckige Rechtecke) und Aktivitäten, die durch ein verschachteltes Statechart (dargestellt durch abgerundete Rechtecke) spezifiziert werden. Das *Modul-Chart* ordnet die Aktivitäten aus dem *Activity-Chart* entsprechenden Modulen der Realisierung zu.

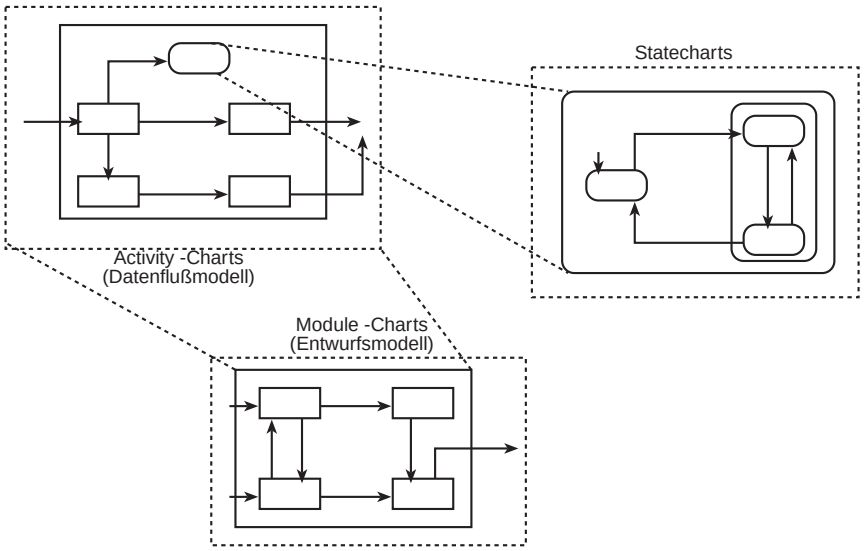


Abb. 10: Überblick über die verschiedenen Diagrammarten in STATEMATE (alle drei Charts können für sich genommen wiederum hierarchisch strukturiert werden)

4.4.4 JSD

Modellierungsart • *Integrierte Modellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Funktionale Dekomposition durch hierarchische Prozeßzerlegung*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch globale Variablen*	***Verhaltensmodellierung*** • *durch Verknüpfungen von Prozessen miteinander*	***Funktionale Modellierung*** • *durch Prozesse und Subprozesse sowie auf Detailebene durch Pseudocode*

Jackson Systems Design (JSD) wurde Mitte der achtziger Jahre von M. Jackson [Jackson83] entwickelt und geht auf dessen früheren Ansatz *Jackson Structured Programming* (JSP) zurück. Ein Systemmodell wird hier durch *Prozeßnetze* (engl. process network) modelliert. Ein *Prozeßnetz* besteht aus einer Menge von sequentiellen Prozessen, die über *Datenflüsse* (JSD: data stream) und *gemeinsame Speicher* (JSD: status vector) miteinander kommunizieren. Die sequentiellen Prozesse im Prozeßnetz werden mit Hilfe von *Jackson-Diagrammen* modelliert. Ein *Jackson-Diagramm* beschreibt den Steuerfluß eines sequentiellen Prozesses, indem dieser in detailliertere Prozesse zerlegt und diese wiederum durch *Sequenz, Iteration* und *Selektion* verknüpft werden. Abbildung 11 zeigt die grundlegenden Notationselemente eines Prozeßnetzes. Sequentielle Prozesse werden durch Rechtecke, gemeinsame Speicher durch Kreise und Datenflüsse durch Rauten dargestellt.

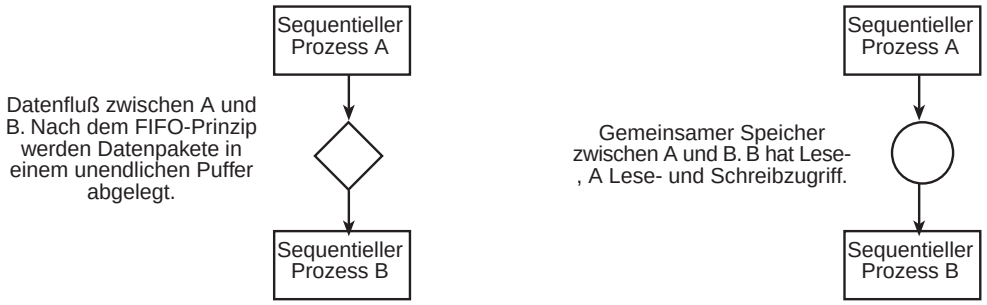


Abb. 11: Grundlegende Notationselemente in JSD

Abbildung 12 zeigt die Verknüpfungselemente *Sequenz, Iteration* und *Selektion* in *Jackson-Diagrammen*. Ein *Jackson-Diagramm* wird durch eine Baumhierarchie mit den Verknüpfungen gebildet.

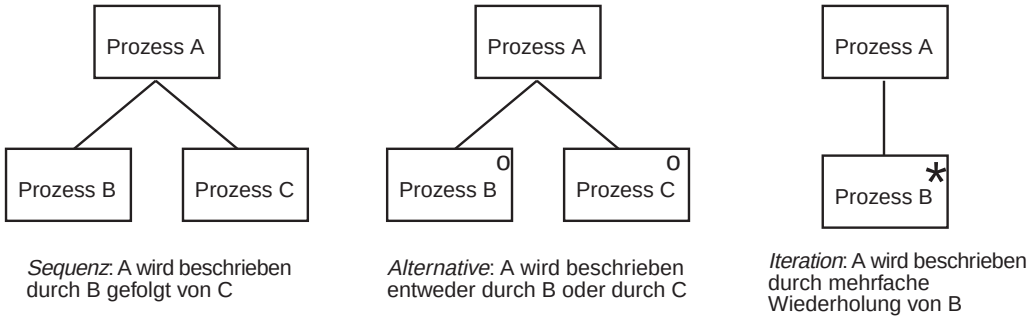


Abb. 12: Verknüpfungen von Aktivitäten in JSD

4.4.5 Bewertung

Datenflußmodelle und verwandte Modellierungsansätze (im folgenden als *strukturierte Ansätze* bezeichnet) waren früher ein recht verbreitetes Mittel zur konstruktiven Systemmodellierung. Der Vorteil *strukturierter Ansätze* liegt zum einen in der einfachen Notation und der leichten Erlernbarkeit der Sprache. Zudem bietet die Sprache durch die hierarchische Datenflußmodellierung Konzepte zur funktionalen Systemdekomposition an: Hierarchische Datenflußmodelle ermöglichen die Beschreibung von Systemmodellen in unterschiedlicher und anpaßbarer Spezifikationstiefe (siehe Kapitel 3.3.1). In folgender Tabelle werden die vorgestellten funktionsorientierten Ansätze zusammenfassend bewertet:

Ansatz	Vorzüge	Nachteile	Bemerkungen
SA	*Einfache und verständliche Notation. Funktionale Beschreibung auf unterschiedlichen Abstraktionsebenen.*	*Daten werden nicht gekapselt und können nur global oder teilglobal beschrieben werden.*	*Durch die funktionale Systemdekomposition können auch umfangreiche Problemstellungen verständlich beschrieben werden. Problematisch ist jedoch die nichtlokale Datenmodellierung*
SADT	*siehe Vorteile SA. Durch Modellierung zusätzlicher Eigenschaften ist gegenüber SA eine präzisere Modellierung möglich.*	*Keine integrierte Modellierung. Durch die redundante Modellierung ist die Entwicklung, sehr aufwendig und wenig änderungsrobust. Datagramme erlauben nur eine grobe Datenmodellierung*	*Das zusätzliche Aufwand einer Aspektmodellierung rechtfertigt nicht den Nutzen der Redundanz und damit verbunden der Fehleraufdekkung. Stattdessen erscheint eine integrierte strukturierte Modellierung sinnvoller.*
STATEMATE	*Siehe Vorteile SA. Aussagekräftige Verhaltensmodellierung durch die Integration von Statecharts.*	*Das Problem der Datenmodellierung und der Datenkapselung ist nach wie vor ungelöst.*	*Die Integration von Statecharts ist vor allem für technische Systeme sinnvoll, bei denen die Verhaltensbeschreibung von großer Bedeutung sind.*
JSD	*Unterstützt eine entwurfs- und implementierungsnahe Modellierung*	*Modellierung von entwurfs- und implementierungsnahen Steuerflüssen. Schwächen bei der Systemkomposition: Die von Jackson vorgeschlagenen Prozeßnetze sind von Grund aus flach, Zerlegungsmechanismen sind nicht vorgesehen*	*JSD ist weniger gut für Anforderungsmodellierung, und eher für entwurf- und implementierungsnahe Beschreibungssprache geeignet. JSD-Modelle sind nur eingeschränkt für die Modellierung von komplexen und umfangreichen Systemmodellen geeignet*

Die strukturierten Ansätze weisen jedoch auch eine Reihe von Defiziten auf, welche die Modellierung komplexer Problemstellungen erheblich einschränken:

- Eine Schwachstelle ist die mangelhafte Umsetzung der Systemmodelle in entsprechende Architekturmodelle. Die Struktur von Architekturmodellen ist in der Regel anders aufgebaut als im Datenflußmodell aufgeführt. Dieses Phänomen wird häufig als *Strukturbruch* bezeichnet [Glinz91, S. 24]. Ursache hierfür ist das Paradigma des Datenflusses und dessen

Modellierung: Datenflüsse sind wenig geeignet, um reale Systemarchitekturen, d.h. um Aufruf- und Dienstleistungsstrukturen (engl. Client-Server) darzustellen. Der *Strukturbruch* ist ebenso Ursache für die schlechte Integrierbarkeit wiederverwendbarer Software und allgemein für die Modellierung von Systemschnittstellen [Glinz91, S. 25f].

- Ein weiteres Defizit ist die mangelhafte Integration von Datenbeschreibungen: Diese sind entweder separat (bei Yourdon[78]) oder global (bei [DeMarco78] und [Gane79]) im Modell gehalten. Zwar werden bei den strukturierten Ansätzen die Aktivitäten (oder Funktionen, Prozesse) zerlegt und dadurch abstrahiert, die Dekomposition von Daten wird jedoch nicht oder nur in geringem Maße unterstützt. Dies hat zur Folge, daß lokale Änderungen der Systemdaten globale Auswirkungen haben können. Zudem wird das Geheimnisprinzip [Parnas72] nicht unterstützt: Wird die Struktur von einzelnen Datenbeständen geändert, so hat dies unter Umständen langwierige Änderungen im ganzen Systemmodell zur Folge.

4.5 Objektorientierte Modellierungsansätze

Objektorientierte Modellierungsansätze beschreiben ein System durch eine Menge von Objekten, die mit Hilfe von Nachrichten miteinander kooperieren. Ein Objekt ist ein individuell erkennbares, von anderen Objekten unterscheidbares Element des Realität [Glinz98b]. Es beschreibt ein zusammenhängendes Teilsystem, d.h.

- einen Teil logisch zusammengehöriger Daten,
- die darüber gültigen Funktionen und Leistungen
- sowie dessen Verhaltensweisen.

Gleichartige Objekte werden durch eine gemeinsame Einheit, die *Klasse* zusammengefaßt. Objekte ermöglichen die Umsetzung des von [Parnas72| vorgestellten Geheimnisprinzips:

- Durch das Zusammenführen von logisch zusammengehörigen Daten und deren Funktionen sind Objekte Teilsysteme mit sehr hoher Kohäsion. Dies ist vorteilhaft, weil das Lokalitätsprinzip unterstützt wird, indem lokale Änderungen im Modell sich nur auf die lokale Umgebung auswirken.
- Daten werden gekapselt und können über definierte Schnittstellen abgefragt bzw. manipuliert werden (durch Funktionen und Leistungen, die über diese Daten definiert sind).
- Durch die Kapselung der Attribute wird die Robustheit bei Änderungen erhöht sowie unerwünschte Seiteneffekte durch eine unbedachte Änderung der internen Objektstruktur erschwert.

Das Konzept der Objektorientierung ist prinzipiell geeignet, um umfangreiche Systemmodelle zu zerlegen und zu strukturieren, und zwar gleichermaßen im Sinne einer Kompositions-, einer

Benutzungs- und einer Generalisierungsabstraktion (siehe Kapitel 3.3). Die *Kompositionsabstraktion* ergibt sich durch Zusammenfassung zusammenhängender Objekte zu komplexeren Einheiten, die selbst wieder Objekteigenschaften haben. Für die *Benutzungsabstraktion* werden Objekte als Dienstleistungseinheiten interpretiert, welche definierte Leistungen zur Verfügung stellen. Ein Objekt kann diese Leistungen vollständig selber erbringen oder die Hilfe anderer Objekte beanspruchen. Diese Delegation bleibt dem Auftraggeber, der eine Leistung in Anspruch nehmen will, verborgen, von der Realisierung der Leistung wird also abstrahiert. Die *Generalisierungsabstraktion* ergibt sich durch Unterteilung von Klassen in generalisierte und spezialisierte Klassen, also in *Ober-* und *Unterklassen* (siehe Kapitel 3.3.1).

Es existiert eine Vielzahl von Ansätzen, die auf dem objektorientierten Paradigma aufsetzen. Diese *objektorientierten Ansätze* sind gegenüber anderen Modellierungsansätzen vergleichsweise jung, die ersten Ansätze entstanden Anfang der achtziger Jahre. Mittlerweise existiert eine fast unüberschaubare Vielzahl an objektorientierten Ansätzen. Die wichtigsten und verbreitetsten Vertreter dieses Sprachparadigmas, die OOA, OOAD, OMT, UML, UML/O-Chart, OOSA und ROOM werden im folgenden vorgestellt und bewertet. Diese Ansätze stehen stellvertretend für eine Vielzahl weiterer konzeptionell ähnlicher Ansätze (siehe [Stein94]).

4.5.1 Entity-Relationship nach Chen

Modellierungsart • *Strukturelle Aspektmodellierung*	***Primäre Struktur*** • *Entitätsmodell inklusive Assoziationszusammenhängen*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch Entitäten und Entitätsattribute*	***Verhaltensmodellierung*** • *-*	***Funktionale Modellierung*** • *-*

Die Entity-Relationship-Modellierung ERM [Chen76] ist die Wurzel aller daten- und objektorientierter Modellierungsansätze. ERM beschreibt ein Systemmodell mit Hilfe von *Entitäten* und *Beziehungen*. Entitäten modellieren Sachverhalte, Personen, Ereignisse, Größen der realen Welt. *Beziehungen* modellieren die Zusammenhänge zwischen *Entitäten. Die* ERM modelliert ausschließlich den strukturelle Aspekt eines Systemmodells, Verhaltens- und Funktionsbeschreibungen sind nicht vorgesehen.

4.5.2 OOA nach Coad/Yourdon

Modellierungsart • *Aspektmodellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Klassenmodell inklusive Assoziations-, Aggregations- und Spezialisierungszusammenhängen*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch Objektattribute und Objekte*	***Verhaltensmodellierung*** • *durch Zustandsautomaten* • *Kommunikation durch Nachrichtenversendung*	***Funktionale Modellierung*** • *durch Methoden (in der Elementarbeschreibung) pseudocodeartig*

Die *Object Oriented Analysis Methode* (OOA) [Coad91] ist einer der ersten objektorientierten Spezifikationsansätze. Ein Systemmodell wird dort grundsätzlich auf Ebene von Klassen modelliert. Im Zentrum steht ein Klassenmodell, bestehend aus einer Menge von Klassen, die durch Attribute und Methoden spezifiziert werden. Zusammenhänge zwischen Klassen werden dort durch folgende Beziehungsarten modelliert (siehe auch Abbildung 13):

- *Assoziation* – Macht eine Aussage darüber, ob Objekte von den in Beziehung stehenden Klassen kommunizieren, kooperieren und/oder ob strukturelle Abhängigkeiten existieren. Die Art der Zusammenarbeit wird durch eine geeignete Wahl einer Bezeichners näher beschrieben. Strukturelle Abhängigkeiten werden weiterhin durch Kardinalitäten spezifiziert.
- *Aggregation* (OOA: Whole-Part Structure) – Spezielle Form der Assoziation. Schwerpunkt ist eine stärkere Formulierung von strukturellen Abhängigkeiten, indem Objekte einer Klasse als Teil oder Mitglied eines anderen Objekts modelliert werden.
- *Spezialisierung* (OOA: Gen-Spec Structure) – Strukturiert Klassen nach dem Prinzip der Generalisierungsabstraktion. Eine Klasse B ist Spezialisierung einer Klasse A, wenn B neben den eigenen Eigenschaften auch alle Eigenschaften von A besitzt.
- *Botschaftsverbindung* (OOA: service connection) – Zeigt eine Botschaft an, die von einer Klasse A zu einer Klasse B geschickt werden kann.

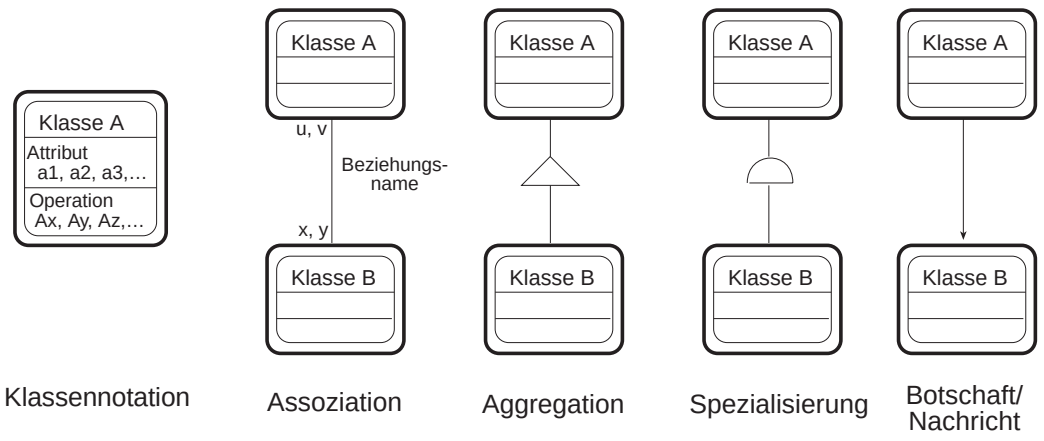


Abb. 13: Notationeller Überblick über OOA nach [Coad91]

Die Klassenmodelle in OOA lehnen sich stark an die klassische Datenmodellierung und im speziellen an die Entity-Relationship-Modellierung [Chen76] an. Aufbauend auf dem Klassenmodell können Klassen nach bestimmten Aspekten näher spezifiziert werden. Die Zustände sowie die Zustandsänderungen einer Klasse werden durch ein *Zustandsdiagramm* modelliert. Die *Elementarbeschreibung* einer Klasse (OOA: class definition), d.h. die Spezifikation der Attribute, Attributtypen, der Operationen mit Schnittstellen usw. wird textuell mit Hilfe eines programmiersprachenähnlichen Pseudocode beschrieben. Operationen können durch *Ablaufdiagramme* (OOA: service chart) spezifiziert werden. Das *Klassenmodell* an sich ist flach, eine Zerlegung in Teilsysteme ist eingeschränkt durch die Unterteilung in *Sachgebiete* (OOA: subject layers) möglich. Abbildung 14 zeigt in schematisches Beispiel einer Einteilung eines Klassenmodells mit den Klassen A, A1, A2, B, C, D in die Sachgebiete *1*, *2* und *3*. Die *Sachgebiete* machen keine Aussage darüber, in welcher Form die Zusammenarbeit oder die strukturellen Abhängigkeiten zwischen Klassen verschiedener Sachgebiete zu spezifizieren sind. Desweiteren gibt es keine Möglichkeit, Zusammenhänge zwischen *Sachgebieten* zu modellieren.

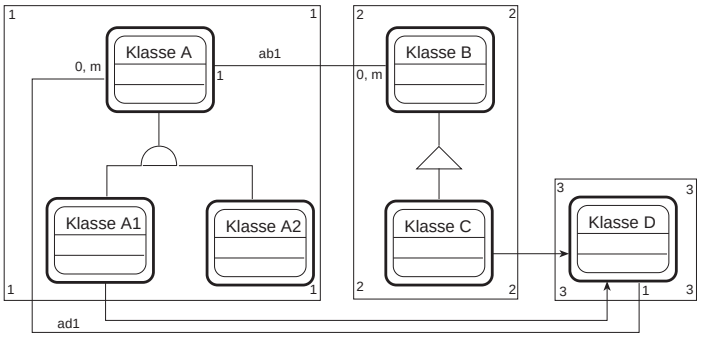


Abb. 14: Subject layers in OOA: Zerlegung eines Klassenmodells in Teilsysteme

4.5.3 OOAD nach Booch

Modellierungsart • *Aspektmodellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Klassenmodell*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch Objektattribute und Objekte*	***Verhaltensmodellierung*** • *durch Statecharts* • *Kommunikation durch Nachrichtenversendung*	***Funktionale Modellierung*** • *durch Methoden (in der Elementarbeschreibung) in einer Programmiersprache*

Die *Object Oriented Analysis and Design* Methode (OOAD) [Booch94] baut ein Systemmodell ähnlich wie OOA [Coad91] auf ein zentral stehendes Klassenmodell auf. OOAD betont jedoch stärker die Aspektmodellierung, indem das zentrale Klassenmodell durch eine Reihe von aspektbezogenen Modellarten, den *Aspektmodellen*, angereichert und ergänzt wird. OOAD unterscheidet dabei folgende Aspektmodelle:

- *Zustandsübergangsmodelle* (OOAD: state diagrams) – Analog zu OOA. Mit Hilfe von Zustandsautomaten werden die möglichen Zustände sowie Zustandsübergänge modelliert.

- *Instanzmodelle* (OOAD: object diagrams) – Beschreiben die Nachrichteninteraktion einer Menge konkreter Klasseninstanzen punktuell, d.h. für einen begrenzten Zeitraum. Die einzelnen Nachrichten werden in zeitlich chronologischer Form festgehalten. Instanzmodelle werden verwendet, um die wichtigsten Abläufe eines Systemmodells explizit zu modellieren.
- *Interaktionsmodelle* (OOAD: interaction diagrams) – Beschreiben eine zu den Instanzmodellen alternative Modellierung der Nachrichteninteraktion. Schwerpunkt bei den Interaktionsmodellen ist jedoch die Modellierung der Interaktion mit der Systemumgebung [Booch94, S. 217 f] und die Beschreibung von typischen Szenarien von externen Akteuren mit dem System.
- *Entwurfsspezifische Modelle* (OOAD: module & process diagrams) – Beschreiben die physikalische Struktur der im Klassenmodell vorhandenen Klassen in Form von Entwurfs- und Implementierungsmodulen und teilen vorhandene physikalische Ressourcen zu.

Das zentrale Klassenmodell in OOAD ist ähnlich wie in OOA prinzipiell flach. Jedoch lassen sie sich in eingeschränktem Maße durch *Klassenkategorien* [Booch94, S. 178f] hierarchisch in Teilsysteme gliedern. Das Konstrukt *Klassenkategorie* ist eine Art Klassenbehälter, dessen Funktionalität durch eine Auswahl «bedeutender» Komponentenklassen beschrieben wird. Beziehungen zwischen *Klassenkategorien* lassen sich nicht explizit modellieren, sondern ergeben sich aufgrund der enthaltenen Klassen: Existiert zwischen der Klasse A und der Klasse B eine Beziehung und ist A in der *Klassenkategorie* CA und B in der *Klassenkategorie* CB, so existiert zwischen CA und CB eine semantisch verarmte Beziehung, eine *Benutzungsbeziehung* (OOAD: Use, S. 179 f.).

4.5.4 OMT nach Rumbaugh et al.

Modellierungsart • *Aspektmodellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Klassenmodell* • *DFD-Hierarchie (in einem separatem Aspektmodell)*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch Objektattribute und Objekte* • *und redundant durch DFD-Speicher*	***Verhaltensmodellierung*** • *durch Statecharts* • *Kommunikation durch Nachrichtenversendung*	***Funktionale Modellierung*** • *durch Methoden (welche in einer Programmiersprache notiert sind)*

Die *Object Modeling Technique* (OMT) nach [Rumbaugh91] unterscheidet sich von den Ansätzen OOA, OOAD und UML in der grundsätzlichen Konzeption: Neben dem Klassenmodell wird ein nicht zum Objektparadigma konformes *Funktionsmodell* eingeführt. Das *Funktionsmodell* ist ein Datenflußmodell (siehe auch Kapitel 4.4, S. 47) und modelliert in Anlehnung an die *strukturierten Ansätze* die Systemfunktionalität. Das Klassenmodell ähnelt denen in den bereits vorgestellten Ansätzen und orientiert sich ebenfalls stark an der klassischen Datenmo-

dellierung. Verhaltensweisen von Klassen werden aufbauend auf der Struktur des Klassenmodells in einem Dynamikmodell durch Zustandsautomaten beschrieben.

Für die Systemdekomposition wird im Klassenmodell ein Konstrukt namens *Subsystem* verwendet, welches in seiner Bedeutung ähnlich schwach ist wie die *Pakete* der UML (siehe Kapitel 4.5.5).

4.5.5 UML nach Booch et al.

Modellierungsart • *Aspektmodellierung* • *textuelle und grafische Notation*	***Primäre Struktur*** • *Klassenmodell*	***Formalitätsgrad*** • *teilformal*
Datenmodellierung • *durch Objektattribute und Objekte*	***Verhaltensmodellierung*** • *durch Statecharts im Zustandsübergangsdiagramm* • *sowie redundant in Interaktions- und Aktivitätsdiagrammen* • *Kommunikation durch Nachrichtenversendung*	***Funktionale Modellierung*** • *durch Methoden in einer Programmiersprache*

Die *Unified Modeling Language* (UML) nach [Booch98] ist ein Sprachansatz neueren Datums, der sich im wesentlichen an die Aspektmodellierung der OOAD anlehnt. Zielsetzung der UML war die Entwicklung einer Einheitssprache, welche schließlich zu einem Industriestandard wird. Die UML entstand ursprünglich aus einer Sprachfusion von OOAD und OMT. Später wurden nach und nach die Ansätze von [Jacobson94], Anregungen aus Industrie und Wirtschaft und schließlich die Sprache OML [Firesmith96] ganz oder teilweise integriert.

Neben der aspektorientierten Modellierung wurden in der UML Konzepte hinzugefügt, mit deren Hilfe sich ein vorgegebener Sprachkern erweitern und auf spezielle Bedürfnisse oder Problembereiche anpassen läßt. Besonderer Augenmerk wurde vor allem darauf gelegt, zahlreiche konzeptionell ähnliche objektorientierte Sprachen wie OOA, OOAD, OMT usw. zu standardisieren und alle durch *eine* erweiterbare objektorientierte Sprache abzudecken. Erwähnt werden sollte an dieser Stelle auch die OML [Firesmith96], ein zur UML ähnlicher Standardisierungsansatz, der allerdings im Laufe der Zeit in die UML «integriert» wurde. Die UML ist momentan noch einem starken Wandel unterzogen. Zudem ist die Beschreibung der Sprache noch sehr unvollständig und widersprüchlich. In der Version 1.1 werden diverse Aspektmodelle unterschieden: Instanzen-, Szenarien-, Zustandsübergangs-, Interaktions-, Aktivitäten- (UML: activity), Prozeß- und Modulmodelle. Bis auf die Szenarienmodelle stimmen die Aspektmodelle in ihrer Bedeutung im wesentlichen mit denen von OOAD überein und bringen nur geringe Neuerungen. Szenarienmodelle beschreiben *Anwendungsfälle* zwischen externen Benutzern und dem System (siehe Kapitel 2.2.2).

Für die Strukturierung der Systemmodelle in Teilsysteme wird ähnlich wie in OOAD ein Behälterkonstrukt angeboten, das *Paket*, dessen Bedeutung und Interpretation jedoch weitest-

gehend offen gelassen wird. Die Erweiterbarkeit der UML wird über *Stereotypen* realisiert (siehe auch [Joos98], [Österreich98]). Sie bieten die Möglichkeit, die UML-Basissprache syntaktisch und semantisch zu erweitern, indem zu einem beliebigen Modellelement *Stereotypen* zugeordnet werden. Diese Stereotypen spezifizieren Prädikate, welche die Eigenschaften dieses Modellelements modifizieren.

Im Bereich der objektorientierten Modellierungssprachen wird die UML zukünftig aufgrund ihrer Erweiterbarkeit und einer umfaßenden Vermarktung voraussichtlich eine tragende und zentrale Rolle einnehmen. Diese Entwicklung wird durch zahlreiche Werkzeug-Entwickler unterstützt, die spezielle UML-CASE-Werkzeuge anbieten.

4.5.6 UML/O-Chart nach Harel

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Aspektmodellierung* • *textuelle und grafische Notation*	• *O-Charts (eine präzisierte Form eines Klassenmodells)*	• *teilformal*
Datenmodellierung • *durch Objektattribute und Objekte*	***Verhaltensmodellierung*** • *durch Statecharts* • *Formale und präzise Spezifikation der Zustandsübergänge* • *Kommunikation durch Nachrichtenversendung*	***Funktionale Modellierung*** • *durch eine eigene C++-ähnliche Sprache*

Die *UML/O-Chart* von [Harel96b, 97] ist eine Erweiterung der UML. Die Erweiterung umfaßt im wesentlichen die *O-Charts* zur präziseren Klassenmodellierung und eine detaillierte formale Spezifikation der Zustandsübergänge. In den *O-Charts* werden Klassen hierarchisch ineinander geschachtelt spezifiziert. Die Quantität einer Klasse (d.h. die Anzahl der zugeordneten Objekte) wird durch eine *Klassenkardinalität* modelliert. Die Zustandsübergänge und die Klassenmethoden werden durch eine eigene operationale formale Sprache (ähnlich der Programmiersprache C++) spezifiziert.

4.5.7 OOSA nach Embley et al.

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *grafische Notation*	• *Klassenmodell und eine hierarchische High-Level-Modellierung* • *Sekundärstruktur: Statecharts*	• *teilformal*
Datenmodellierung • *durch Objektattribute und Objekte*	***Verhaltensmodellierung*** • *durch Statecharts* • *Kommunikation durch Nachrichtenversendung*	***Funktionale Modellierung*** • *durch Methoden in einer Programmiersprache*

Die *Object Oriented Systems Analysis* Methode (OOSA) nach [Embley92] verwendet zur Systemmodellierung ausschließlich ein Klassenmodell und ein integriertes Zustandsübergangsmodell (ähnlich den Statecharts). Im Klassenmodell wird versucht, mit Hilfe einer *High-*

Level-Modellierung Modelle hierarchisch zu strukturieren. Hierfür werden durch *unabhängige* und *dominante Implosionen* zwei Konzepte zur Zusammenfassung von Klassengruppen zu einer einzigen übergeordneten Klasse zur Verfügung gestellt.

- Die *unabhängige Implosion* faßt n Klassen zu einer neuen *High-Level-Klasse* zusammen.
- Bei der *dominanten Implosion* wird eine der n Klassen als *High-Level-Klasse* gewählt, welche die restlichen n-1 Klassen in abstrakter Form zusammenfaßt.

Detailbeziehungen der n Klassen (bzw. der n-1 Klassen bei der *dominanten Implosion*) zu den Klassen der Umgebung werden durch rein syntaktische Beziehungslinien ohne Beziehungseigenschaften repräsentiert. Durch die zwei Implosionsarten können Klassenmodelle sukzessive vergröbert oder verfeinert werden. Ansonsten stimmen die Klassenmodelle und die Zustandsübergangsmodelle im groben mit denen der OOAD oder der UML überein: Jeder Klasse wird eine entsprechende zustandsbasierte Verhaltensbeschreibung zugeordnet, die jedoch lokal, ohne Integration anderer Zustandsbeschreibungen, das Verhalten dieser einen Klasse modelliert.

4.5.8 ROOM nach Selic et al.

Modellierungsart	***Primäre Struktur***	***Formalitätsgrad***
• *Integrierte Modellierung* • *wahlweise textuelle oder grafische Notation*	• *Klassenmodell (in welches lokales Klassenverhalten eingeblendet wird)* • *Teil/Ganzes-Klassenhierarchie durch Verwendung von Klassenreferenzen*	• *teilformal*
Datenmodellierung	***Verhaltensmodellierung***	***Funktionale Modellierung***
• *durch Objektattribute und Objekte*	• *durch Statecharts* • *Kommunikation durch explizite Nachrichtenversendung (durch Kanäle)*	• *durch Methoden (in der Elementarbeschreibung) in einer Programmiersprache* • *Schnittstellendefinition durch Klassenprotokolle*

Der *Real-Time Object Oriented Modeling* Ansatz (ROOM) nach [Selic94] ist ein Modellierungsansatz, der speziell für technische und für Echtzeit-Problemstellungen entwickelt wurde. Im Mittelpunkt steht eine Teil/Ganzes-Klassenhierarchie, in welcher eine Komponente einer Klasse durch eine Referenz auf eine andere Klasse modelliert wird. In diese Hierarchie wird das lokale Verhalten einzelner Klassen eingeblendet (ähnlich der Verhaltensintegration bei OOSA). Besonderheit der ROOM-Methode ist die präzise Modellierung der Schnittstelle einer Klasse (durch ein Schnittstellenprotokoll) und darauf aufbauend die explizite Nachrichtenmodellierung durch Nachrichtenkanäle.

4.5.9 Bewertung

Das objektorientierte Konzept weist gegenüber den anderen hier vorgestellten Modellierungskonzepten eine Reihe von Vorteilen auf. Objektorientierte Systemmodelle sind in ihrer Grob-

struktur in der Regel stabiler und weniger änderungsanfällig als funktional oder verhaltensorientiert zerlegte Systemmodelle. Die Praxis zeigt, daß im Laufe von Software-Entwicklungen Funktionen und mögliche Systemzustände deutlich stärker der Änderung unterworfen sind und mutieren als feststehende Entitäten (also als Objekte) der realen oder der Vorstellungswelt [Davis90], [Macaulay96]. Ein weiterer Vorteil des objektorientierten Konzeptes ist die konzeptionell gute Unterstützung der Abstraktionsmechanismen *Generalisierung, Komposition* und *Benutzung* und nicht zuletzt die Umsetzung des *Geheimnisprinzips* [Parnas72]. Das objektorientierte Konzept ist aus diesem Grund prädestiniert, komplexe und umfangreiche Systemmodelle zu strukturieren und klar und verständlich zu halten. Mag das objektorientierte Konzept jedoch noch so vielversprechend klingen, praktisch wird es nur sehr unzureichend in objektorientierten Modellierungssprachen umgesetzt. Die zentralen Schwachstellen liegen vor allem in der mangelhaften Realisierung von Dekompositionskonzepten und in unzureichenden Konzepten zur Beschreibung globalen Verhaltens.

Mangelhafte Realisierung von Dekompositionskonzepten – In nahezu allen objektorientierten Modellierungsansätzen finden sich Sprachkonstrukte, die es erlauben, eine Menge von Klassen in vergröberter Form zu einem komplexeren Teilsystem zusammenfassen. Die Teilsystemkonstrukte gehen jedoch nicht oder nur sehr unwesentlich über die Bedeutung einer Menge oder eines Behälters hinaus. Notwendig wäre jedoch die präzise Klärung der Bedeutung eines solchen *Teilsystemkonstruktes* und daraus resultierend die Erfassung von Teilsystemeigenschaften. Geklärt werden müßte,

(i) in welcher Form die Funktionalität eines Teilsystems beschrieben wird, welcher Zusammenhang zwischen der Teilsystem-Funktionalität und der bereitgestellten Funktionalität der einzelnen Komponenten des Teilsystems (Klassen, Objekte oder wiederum Teilsysteme) besteht

(ii) ob ein Teilsystem selbst wiederum die Charakteristika eines Objektes bzw. einer Klasse aufweist, d.h. ob es eigene Attribute und Verhaltensweisen besitzt oder ob ein Teilsystem allein über seine Komponenten definiert ist

(iii) wie die Zusammenarbeit und die Kommunikation eines Teilsystems mit seiner Umgebung (potentiell anderen Teilsystemen oder Klassen) beschrieben wird und ob und wie ein Teilsystem mit seinen Komponenten kommunizieren kann

(iv) wie generell eine Klasse als Komponente eines Teilsystems auf Ausprägungsebene, d.h. auf Ebene der Klasseninstanzen zu interpretieren ist: Ist ein Teilsystem, welches Klassen enthält, ein Typ oder eine Ausprägung? Müssen alle Instanzen einer Klasse im übergeordneten Teilsystem liegen oder ist es möglich, daß Instanzen an anderer Stelle des Modells vorkommen können?

Unzureichende Konzepte zur Beschreibung globalen Verhaltens – Der zweite Kritikpunkt besagt, daß vorhandene Ansätze Verhalten entweder lokal für einzelne Klassen (beispielsweise

durch Zustandsautomaten) oder nur punktuell und beispielhaft (beispielsweise durch Instanz- oder Interaktionsmodelle) beschreiben. Das gemeinsame Verhalten mehrerer unterschiedlicher Objekte (etwa diejenigen eines Teilsystems) wird jedoch nur unzureichend unterstützt. Die Zerlegung von Systemmodellen in überschaubare Teilsystem erfordert jedoch eine Verhaltensbeschreibung auf abstrakterer Ebene als nur für relativ feingranulare Klassen.

Im folgenden werden die einzelnen objektorientierten Ansätze im Überblick bewertet:

Ansatz	**Vorzüge**	**Nachteile**	**Bemerkungen**
ERM	*Erster bekannter Ansatz zur anschaulichen Datenmodellierung*	*Keine Funktions- und Verhaltensbeschreibung. Keine Möglichkeit der Systemdekomposition vorgesehen*	*Nur geeignet für datenintensive Modellierungen, bei denen die Funktionalität anderweitig beschrieben wird*
OOA	*Einfache, überschaubare und leicht zu erlernende Notation*	*Die OOA-Sachgebiete sind nur ein unzureichendes Strukturierungsmittel, da keine Sachgebiets-Eigenschaften (Verhalten oder Funktionalität) modelliert werden können*	*Geeignet nur als «oo-Einstiegssprache». Für komplexere Problemstellungen ist sie jedoch ungeeignet.*
OOAD	*Die OOAD-Klassenkategorien erlauben eine deutlich präzisere hierarchische Modellierung (geg. OOA).*	*Die Klassenkategorien sind sehr restriktiv in ihrer Anwendung. Zudem sind die Benutzungsbeziehungen zwischen Klassenkategorien semantisch sehr verarmt.* *Notation etwas zu barock (Geschmacksache)*	*Defizite bei der Modellstrukturierung*
OMT	-	*Die OMT-Subsysteme sind ähnlich eingeschränkt wie die UML-Pakete.* *Klassen- und Funktionsmodell sind strenggenommen zwei unabhängige Modelle, die ein hochgradig redundantes und fehleranfälliges Vorgehen provozieren, da Probleme sowohl objektorientiert als auch funktionsorientiert modelliert werden. Wenig intuitive Notation (hier speziell die Beziehungskardinalitäten)*	*Große Defizite bei der Modellstrukturierung. Durch eine parallel funktionale Modellierung ist OMT praktisch zu aufwendig und nur bedingt mit dem Objektparadigma konform.*
UML	*Anpassung und Präzision der UML-Basissprache durch Stereotypen.* *Hoher Bekanntheitsgrad, gute Werkzeugunterstützung*	*Notation durch das Vorhandensein der vielen Aspektmodelle umfangreich und sehr unübersichtlich.* *Die UML-Pakete sind ähnlich semantisch schwach wie die OOA-Sachgebiete. Zwar können Beziehungen zwischen Paketen und der Paketumgebung modelliert werden, weitere Teilsystem-Eigenschaften (Attribute, Funktionalität, Verhalten) lassen sich jedoch nicht beschreiben.*	*Defizite bei der Modellstrukturierung, die auch nicht durch den Einsatz von Stereotypen behoben werden können.* *Durch die Vielzahl an Aspektbeschreibungen wandelt sich die UML weg von einer Sprache hin zu einer Diagrammartenbibliothek.*
UML/O-Charts	*Siehe Vorteile UML.* *Präzise und anschauliche lokale Verhaltensbeschreibung.* *Eine gegenüber OOA, OMT, OOAD, UML deutlich aussagekräftigere Modellstrukturierung.*	*Siehe Nachteile UML. Statecharts beschreiben nach wie vor nur lokales Verhalten und sind nicht in die Objekthierarchie integriert*	*Die Integration der Statecharts in ein OO-Paradigma erscheint nicht so gelungen wie deren Integration in STATEMATE. Die prinzipiellen Schwachstellen der UML werden dadurch nicht behoben.*

OOSA	*Eine gegenüber OOA, OMT, OOAD, UML deutlich aussagekräftigere Modellstrukturierung.*	*Durch das OOSA-Implosionskonzept werden zwar Klassen zu High-Level-Klassen zusammengefaßt, die Beziehungen zur Umgebung werden jedoch nicht richtig berücksichtigt und nur schlecht modelliert.*	*Eine integrierte Modellierung setzt das Vorhandensein aussagekräftiger Abstraktionsmechanismen voraus, die jedoch durch die High-Level-Modellierung nur zum Teil bereitgestellt werden.*
ROOM	*Präzise und anschauliche lokale Verhaltensbeschreibung. Detaillierte und präzise explizite Nachrichtenmodellierung*	*Semantik der Klassenreferenzen ungeklärt: Auf welche Klasseneigenschaften nimmt eine Referenz Bezug, welche bleiben verborgen. Durch die Referenzierung entsteht nur scheinbar eine abstrahierende Klassenhierarchie (verborgen bleiben unsichtbare Seiteneffekte).*	*Unterstützt die Modellierung technischer und echtzeitproblematischer Problemstellungen. Die Verwendung von Klassenreferenzen und der Klassenhierarchien stellt ebenfalls kein ausreichendes Strukturierungskonzept dar.*

Die Ansätze OOA, OOAD, OMT, UML uns UML/O-Chart gehen von derselben Grundüberlegung aus, ein Systemmodell durch ein zentrales Klassenmodell zu beschreiben, welches gegebenenfalls durch weitere *Aspektmodelle* ergänzt wird. Speziell in diesen Ansätzen werden verstärkt *Projektionsmechanismen* zur Modellstrukturierung eingesetzt, Abstraktionsmechanismen aber eher vernachlässigt[1] (siehe Kapitel 3.3). *Projektionen* werden in zwei verschiedenen Varianten eingesetzt:

- Zum einen als eine Projektion von bestimmten Aspekten, realisiert durch verschiedene *Aspektmodelle* (etwa durch Szenarien-, Zustandsübergangs-, Interaktions-, Aktivitäten- (UML: activity), Prozeß- und Modulmodelle).
- Zum anderen innerhalb eines bestimmten Aspektmodells durch *Überlappung* von Einzeldiagrammen: Ein Fragment eines Aspektmodells wird ohne Bezugnahme auf seinen Kontext eigenständig aufgeführt.

Die zweite Variante der Projektion ist ein höchst zweifelhafter Strukturierungsmechanismus: Ein durch Überlappung gebildetes *Aspektmodell* wird leicht unvollständig, weil nicht alle zu modellierenden Zusammenhänge sich in den Einzeldiagrammen wiederfinden. Zudem vermittelt die Überlappung nur Einzelsichten, jedoch kein Eindruck des Gesamtmodells oder größerer Teile davon. Projektionen und die Bildung von Sichten allein reichen jedoch nicht aus, um komplexe Probleme zu strukturieren. Insbesondere reichen sie nicht aus, um fehlende und schlecht implementierte Abstraktionsmechanismen zu ersetzen.

Integrierte Modellierungsansätze wie beispielsweise OOSA oder ROOM besitzen gegenüber den aspektorientierten Ansätzen deutlich aussagekräftigere Strukturierungsmechanismen. Auch dort jedoch ist ein durchgängiger und brauchbarer Zerlegungsmechanismus im Sinne einer Komposition oder einer Benutzung nicht vorhanden.

---

[1] Ausnahme bilden hier die Generalisierungshierarchien, welche durchgängig in allen Ansätzen unterstützt wird.

Durch die Sprache ADORA-L soll gezeigt werden, daß sich die meisten der oben aufgeführten Probleme umgehen lassen, wenn Systemmodelle nicht stur auf Ebene der Klassen, sondern auf Ebene von *abstrakten Objekten* beschrieben werden. Diese abstrakten Objekte erlauben einerseits eine präzise Modellierung des Objektkontextes, was für eine sinnvolle globale Verhaltensmodellierung notwendig ist. Zum anderen lösen sie das Problem der Klassenmodelle, indem sie den Dualismus zwischen intensionaler und extensionaler Bedeutung von Klassen auflösen und das Systemmodell extensional beschreiben (während die Klassen einzig als Objekttypen verwendet werden).

Kapitel 5

Sprachkonzepte von ADORA-L

Die Sprachkonzepte legen die grundsätzlichen Prinzipien einer Sprache fest. Sie bilden ihr Grundgerüst und sind dafür verantwortlich, wie die verschiedenen Komponenten einer Sprache zusammenhängen und wie deren Sprachkonstrukte zusammenspielen. Die Qualität einer Sprachdefinition ist daher hochgradig abhängig von der sorgfältigen Wahl von widerspruchsfreien und sich ergänzenden Sprachkonzepten. Für die Spezifikationssprache ADORA-L wurden folgende Sprachkonzepte gewählt:

- Die *Objektorientierung* als das grundsätzliche Sprachparadigma und damit verbunden die objektorientierte Modellierung auf Ebene *abstrakter Objekte* (siehe Kapitel 5.1).
- Die Verwendung eines *hierarchischen* und *integrierten* Gesamtmodells zur Anforderungsmodellierung (siehe Kapitel 5.2).
- Die Objektkommunikation durch die *Versendung von Nachrichten*. Die Kommunikation zwischen Objekten wird *explizit* durch *Nachrichtenkanäle* modelliert, die gleichzeitig Basis für die statische Beziehungsmodellierung zwischen Objekten bilden (siehe Kapitel 5.3).
- Eine in das Teil/Ganzes-Objektmodell *integrierte Verhaltensbeschreibung.* Das Verhalten wird beschrieben durch hierarchisch geschachtelte Zustandsautomaten und darauf abgestimmte Zustandsübergänge (siehe Kapitel 5.4).
- Die Beschreibung der *Objektfunktionalität in variablem Formalisierungsgrad*. Die Operationen werden durch die Angabe von Voraussetzungen und Nachbedingungen spezifiziert. Wahlweise kann hierfür die formale Sprache ADORA-FSL, eine natürliche Sprache oder eine Mischform dieser Sprachen verwendet werden (siehe Kapitel 5.5).
- Eine zur Objekthierarchie *orthogonale Klassenhierarchie* im Sinne einer Generalisierungsabstraktion (siehe Kapitel 5.6), welche die im Objektmodell verwendeten Typen sowie deren Zusammenhänge deklariert.

Das folgende Kapitel diskutiert diese Sprachkonzepte im Detail und klärt deren Zusammenhänge.

5.1 Objektorientierung und abstrakte Objekte

Das folgende Kapitel klärt, welche Problematik sich durch die *Bedeutungsdualität* von Klassen in Klassenmodellen ergibt. Im Anschluß daran wird das Konzept der *abstrakten Objekte als* Alternative zum Konzept der Modellierung auf Klassenebene vorgestellt. Es wird gezeigt, weshalb die Modellierung auf Ebene der abstrakten Objekte die genannten Probleme im wesentlichen überwindet und deshalb einem klassenorientierten Konzept vorzuziehen ist. Zum Abschluß wird gezeigt, in welcher Weise das Konzept der *abstrakten Objekte* in ADORA-L umgesetzt wird.

5.1.1 Die Bedeutungsdualität von Klassen

Die Modellierungssprache ADORA-L zählt zu den objektorientierten Modellierungsansätzen, d.h. sie orientiert sich am objektorientierten Paradigma: Ein Systemmodell wird anhand einer Menge interagierender Objekte beschrieben, gleichartige Objekte werden wiederum durch Klassen zusammengefaßt (siehe auch Kapitel 4.5, S. 54). Umgesetzt wird dieses objektorientierte Paradigma in objektorientierten Modellierungssprachen üblicherweise in Form von *Klassenmodellen*. In Anlehnung an Chen´s Entity-Relationship-Modelle [Chen76] ist die Klasse das zentrale Beschreibungsmittel und wird hier in einer problematischen dualistischen Bedeutung benutzt:

- Eine Klasse repräsentiert einerseits ein Objekt oder eine Menge gleichgearteter Objekte. In dieser *extensionalen* Bedeutung muß der Objektkontext (d.h. die Kommunikation, die Interaktion und die Beziehungen des Objekts) von der Klasse beschrieben werden.
- Eine Klasse ist jedoch gleichzeitig *intensional* ein Objekttyp. Sie beschreibt kontextunabhängige Eigenschaften von Objekten, d.h. die gemeinsamen Eigenschaften aller Objekte desselben Typs.

Diese Doppeldeutigkeit führt zur *Modellierungsanomalie*: Der Kontext, in dem ein Objekt steht, läßt sich nicht mehr präzise modellieren. Die Ursache liegt im intensionalen Charakter einer Klasse: Alle Instanzen einer Klasse werden alle gleichzeitig und gemeinsam beschrieben. Die gemeinsame Beschreibung aber hat zur Konsequenz, daß der individuelle Kontext einer Klasseninstanz nicht mehr im einzelnen zu erfassen ist. Abbildung 15 zeigt die Schwäche der Klassenmodellierung. Instanzen der Klasse `B` werden in zwei unterschiedlichen Kontexten gebraucht, in Objektgruppe I und II. Die Instanzen `B'` und `B''` haben in ihrem jeweiligen Kontext unterschiedlichartige Beziehungen und Instanzen. Auf Ebene der Klassen ist jedoch die Unterschiedlichkeit der zugehörigen Instanzen nicht mehr darstellbar.

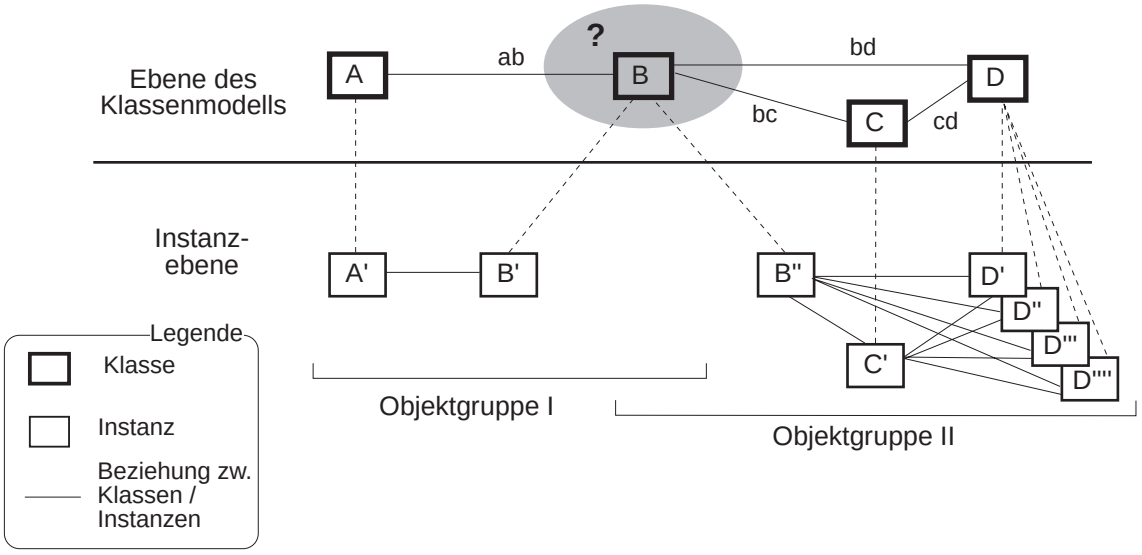


Abb. 15: Die Modellierungsanomalie bei der Klassenmodellierung: Der Kontext der Instanzen von B sind nicht adäquat modellierbar.

Die *Modellierungsanomalie* soll nun anhand eines einfachen Beispiels erläutert werden. In Abbildung 16 beschreibt ein Klassenmodell (in einer UML-ähnlichen Notation) den groben Aufbau eines Institutes, indem sowohl das Sekretariat als auch die einzelnen Forschungsgruppen auf baugleichen Druckern diverse Dokumente ausdrucken. Die einzelnen Institutionen werden durch Klassen, die Zusammenhänge zwischen den Klassen durch Beziehungen modelliert.

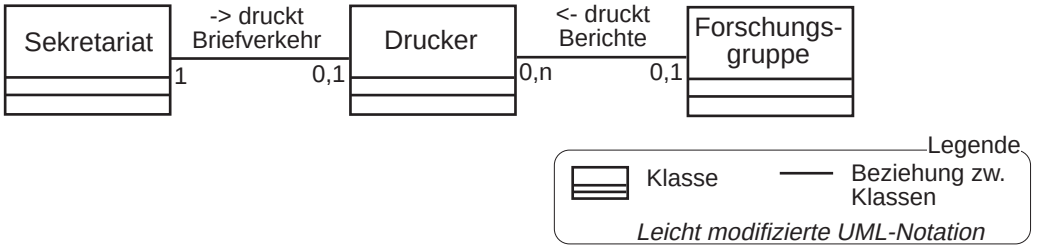


Abb. 16: Beispiel für die Schwächen einer Modellierung auf Klassenebene[1]

Durch die Klassenmodellierung sind eine Reihe von Sachverhalten nicht oder nur unklar dargestellt:

(i) Kann ein bestimmter Drucker (also eine Instanz der Klasse Drucker) nur vom Sekretariat oder von den Forschungsgruppen benutzt werden oder werden alle Drucker gemeinsam von allen Institutionen gleichermaßen benutzt?

[1] Die Beziehungskardinalitäten werden in der Art von «Eine Forschungsgruppe druckt Berichte auf 0 bis n vielen Druckern» *notiert.*

(ii) Ist ein Drucker einer Forschungsgruppe zugeordnet oder können mehrere Forschungsgruppen einen Drucker gemeinsam benutzen?

(iii) Gibt es Drucker, die weder von Sekretariaten noch von irgendeiner Forschungsgruppe verwendet werden?

Ursache für diese Ausdrucksschwäche ist das Fehlen einer geeigneten Kontextmodellierung. Die Modellierung eines Druckertyps ist sinnvoll, da alle Drucker baugleich sind, also auch dieselben Leistungen erbringen. Jedoch kann nicht differenziert werden, in welchen Rollen die einzelnen Drucker-Instanzen verwendet werden (siehe Fall (i) und (ii)). Auf Ebene der Klassen können diese Rollen nur gemeinsam durch die ausdrucksschwache Optionalität («0,1» bzw. «0,n») modelliert werden. Diese erzwungene Optionalität hat weiterhin zur Folge, daß die Frage in Fall (iii) im Klassenmodell nicht ersichtlich ist.

Diese Problematik kann durch Verwendung von Klassenspezialisierungen theoretisch umgangen werden. Eine generelle Klasse (Beispiel: Die Klasse Drucker) wird durch rollenspezifische Klassenspezialisierungen (Beispiel: Die Klassen Sekretariatsdrucker und Gruppendrucker) ergänzt. Diese Lösung ist jedoch sehr unflexibel und führt zu unnötig komplexen Modellen.

5.1.2 Modellierung durch abstrakte Objekte

In ADORA-L wird diese Bedeutungsdualität vermieden, indem ein Systemmodell auf zwei Abstraktionsebenen modelliert wird, auf einer extensionalen *Objektebene* und auf einer intensionalen *Typebene*. Die *Objektebene* beschreibt das System auf Ebene von *abstrakten Objekten*. *Abstrakte Objekte* sind Platzhalter für konkrete Objektinstanzen und beschreiben diese somit in abstrakter Form. Ein *abstraktes Objekt* besitzt einen eigenständigen Kontext, hat jedoch keine konkret bekannte Identität. Es unterscheidet sich von der konkreten *Objektinstanz*, weil die Identität durch die Abstraktion noch unbestimmt ist. Es unterscheidet sich von der *Klasse*, weil es einen definierten Kontext besitzt. Zur Verdeutlichung des Objektkonzeptes werden im folgenden die Begriffe *Objektinstanz*, *Objekt*, *Objektmenge* und *Klasse* voneinander abgegrenzt.

Definition *Objektinstanz* (Synonym: Instanz) – Eine Objektinstanz ist eine im laufenden System konkret vorhandene und individuelle Einheit. Jede Objektinstanz ist einem abstrakten Objekt bzw. einer abstrakten Objektmenge zugeordnet. Sie enthält durch Attribute und Komponenten repräsentierte Information, deren Struktur im abstrakten Objekt (Objektmenge) definiert ist. Eine Objektinstanz kann die dort definierten Nachrichten empfangen, d.h. es besitzt für jede definierte Nachricht eine entsprechende Leistungsbeschreibung und entsprechende Verhaltensweisen.

Definition *Abstraktes Objekt (Synonym: Objekt)* – Ein abstraktes Objekt ist Platzhalter für und Repräsentant von einer Instanz und definiert deren Attribute, deren Funk-

tionalität und deren Verhalten auf abstrakter Ebene. Abstrahiert wird von der Individualität und der Konkretheit. Die genaue Identität des Objekts sowie die konkreten Attributwerte bleiben unbekannt, deren Eigenschaften, im speziellen Informationen über den speziellen Objektkontext, finden sich in der Objektdefinition wieder.

Definition *abstrakte Objektmenge* – Eine abstrakte Objektmenge ist eine Menge von abstrakten Objekten derselben *Klasse* (siehe Definition Klasse). Die Elemente der abstrakten Objektmenge stehen im selben Kontext und werden durch eine gemeinsame Definition beschrieben. Aus einer abstrakten Objektmenge können Objekte entfernt und neue Objekte erzeugt und hinzugefügt werden. Eine abstrakte Objektmenge besitzt also grundsätzlich eine variable Größe.

Die *Typebene* beschreibt ausschließlich die intensionalen Aspekte von Objekten, also den Objekttyp. Der Typ eines Objektes legt für eine Reihe gleichartiger Objekte die Gemeinsamkeiten fest, etwa das gemeinsame Verhalten oder den Leistungsumfang der Objekte. Im Gegensatz zu der verbreiteten dualistischen Klassenbedeutung werden Klassen in ADORA-L ausschließlich intensional als Typen von Objekten verwendet.

Definition *Klasse* – Eine Klasse ist ein Typ, der die gemeinsamen Eigenschaften einer Menge von gleichartigen Objekten intensional definiert. Alle Objekte und Objektmengen der Klasse sind also nach dieser Typdefinition aufgebaut.

Abbildung 17 zeigt das in ADORA-L verwendete Konzept der Modellierung auf zwei Ebenen, einer intensionalen Typ- und einer extensionalen abstrakten Objektebene.

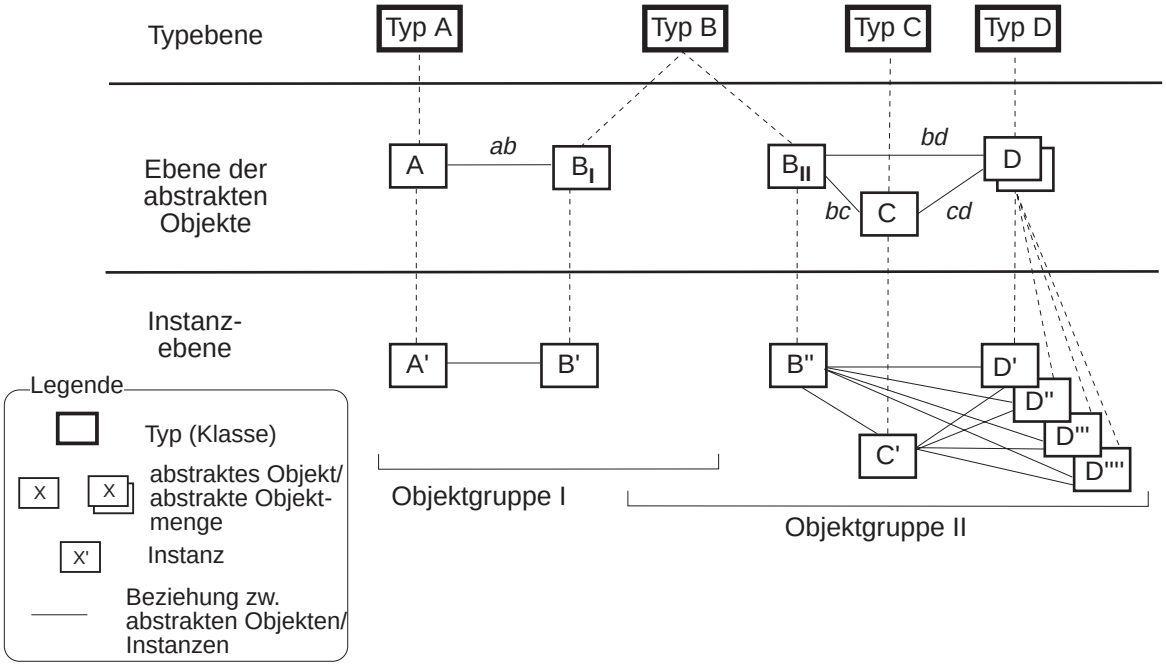


Abb. 17: Modellierung auf Ebene der abstrakten Objekte: Der Kontext von B' läßt sich unabhängig von dem von B" modellieren. Folge ist Überwindung der Modellierunganomalie.

Die Ebene der abstrakten Objekte stellt eine Abstraktion der Instanzebene dar, welche selber nach wie vor nicht modelliert wird. Zwar wird von der konkreten Identität der Instanzen abstrahiert, nicht aber vom Kontext, in dem die Instanzen stehen. Die Abbildung zeigt auf Instanzebene zwei Teilsysteme (als Objektgruppen) von Instanzen, in denen mit B' und B'' jeweils eine Instanz des Typs B enthalten ist. In der Objektgruppe II befindet sich zudem eine beliebig große Anhäufung von Instanzen des Typs D, die mit den Instanzen B'' und C' kommunizieren. Auf Ebene der abstrakten Objekte können durch B_I und B_{II} zwei abstrakte Objekte des Typs B modelliert werden, die aber in unterschiedlichen Kontexten stehen (dieser Sachverhalt kann in einem Klassenmodell in dieser Form nicht dargestellt werden). Die Instanzen D', D'',... werden durch die Objektmenge D dargestellt. Auf Typebene wird das kontextunabhängige Verhalten von gleichartigen Objekten gemeinsam beschrieben. Die Klasse B etwa beschreibt die gemeinsamen Eigenschaften der Objektinstanzen B' und B'' bzw. die Gemeinsamkeiten ihrer abstrakten Objekte B_I und B_{II}.

In Abbildung 18 wird das Beispiel aus Abbildung 16 erneut modelliert, diesesmal jedoch durch eine intensionale Typebene und eine extensionale Objektebene.

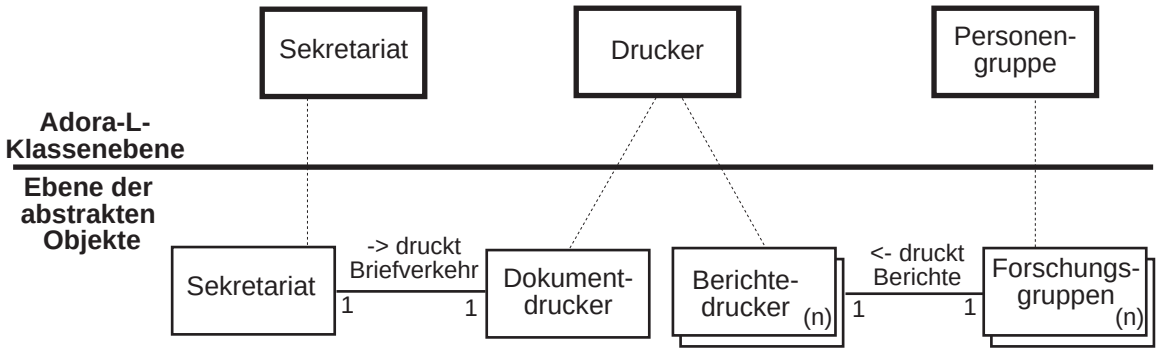


Abb. 18: Beispiel aus Abbildung 16, modelliert durch abstrakte Objekte bzw. durch Objektmengen.

Unterschieden werden die Klassen (Objekttypen) Sekretariat, Drucker und Personengruppe. Zwischen den Klassen existieren keine typbezogenen Zusammenhänge (wie beispielsweise Spezialisierungen usw.). Der Klasse Sekretariat ist ein abstraktes Objekt zugeordnet, der Klasse Personengruppe eine Menge von Forschungsgruppen (dargestellt durch die Objektmenge). Die Bezeichnung der Klasse muß nicht notwendigerweise mit der des Objektes bzw. der Objektmenge übereinstimmen. Im Falle einer Abweichung bezeichnet der Objekt/Objektmengenname die jeweilige Rolle, indem es/sie im Kontext verwendet wird[1]. Der Klasse Drucker sind ein abstraktes Objekt und eine abstrakte Objektmenge zugeordnet, welche jeweils im Kontext Sekretariat bzw. Forschungsgruppe eingesetzt werden. Hier wird offensichtlich, daß insgesamt nur ein Sekretariat existiert und diesem genau ein Drucker zugeordnet ist. Ebenso ist ersichtlich, daß jede Forschungsgruppe einen eigenen Drucker besitzt, der von niemandem anders benutzt

[1] Weitere Aussagen über Objektnamen siehe Abbildung 5.2.3

werden kann (modelliert durch die Beziehungskardinalitäten). Sämtliche der drei Fragen, welche im Beispiel von Abbildung 16 unbeantwortet blieben, können hier modelliert werden.

Anzumerken ist noch, daß einige Ansätze (z. B. [Booch94], [Rumba91] oder [Booch98]) *Objektdiagramme* zur Beschreibung des Objektkontextes verwenden. Ein *Objektdiagramm* beschreibt jeweils einen speziellen Handlungsablauf. Auch dort werden Sprachkonstrukte eingesetzt, die den abstrakten Objekten bzw. Objektmengen entsprechen. Objektdiagramme sind jedoch nicht geeignet, um obige Probleme zu beheben, da sie nur punktuelle und beispielhafte Aussagen machen. Ein zu modellierendes System kann so nur partiell beschrieben werden, da das Klassenmodell durch die Objektdiagramme nicht ersetzt wird.

Das Konzept der abstrakten Objekte erinnert an prototypenbasierte Sprachansätze wie beispielsweise SELF oder BETA (siehe [Ungar94] bzw. [Krogdahl93]). In diesen Ansätzen werden Modelle allein auf Ebene *prototypischer Objekte* spezifiziert, die den hier vorgestellten abstrakten Objekten sehr ähnlich sind. Neue Objekte werden jedoch nicht durch eine Typschablone definiert, sondern werden durch *Klonung* und *Mutation* eines existierenden prototypischen Objektes gebildet. Die Klonung übernimmt alle Eigenschaften und Wertebelegungen des Originals in das neue Objekt, welches durch die Mutation modifiziert und ergänzt werden kann. Werden Leistungen nicht direkt von einem Objekt umgesetzt, wird es an das übergeordnete Objekt *delegiert*. Eine so gebildete *Klonhierarchie* unterstützt die Modellierung einer Generalisierungsabstraktion. Im Gegensatz dazu ist ADORA-L nach wie vor zweischichtig und beschreibt sowohl auf intensionaler und extensionaler Ebene, wobei Extension und Intension klar voneinander abgegrenzt sind. Die Trennung ist von zentraler Bedeutung, um Modelle hierarchisch zu strukturieren und um andererseits eine separate Generalisierungsabstraktion zu ermöglichen.

Für die Umsetzung der Idee der abstrakten Objekte und der zweischichtigen Modellierung ist eine wichtige Rahmenbedingung zu beachten. Die Modellierung auf Ebene der abstrakten Objekte erfordert eine eindeutige Zuordnung der Instanzen zu ihren Platzhaltern, also zu den jeweiligen abstrakten Objekten:

> **Integritätsbedingung** [1] *Eindeutigkeit der Instanz/Objekt-Zuordnung*: Jede Objektinstanz, die auf Instanzebene aufgeführt wird, ist genau einem Objekt auf der abstrakten Objektebene zugeordnet.

Wird eine Instanz mehreren Objekten zugeordnet, wird an unterschiedlichen Stellen des Objektmodells ein und dieselbe Instanz beschrieben, was dann zu unsichtbaren Seiteneffekten und damit zum Widerspruch zum objektorientierten Paradigma führt.

---

[1] Integritätsbedingungen werden im Sinne von [Zehnder89, S. 175f] als eine Invariante, welche vom Modell selber zu erfüllen ist.

Zusammenfassend läßt sich sagen, daß ADORA-L-Systemmodelle durch die Verwendung von abstrakten Objekten die Bedeutungsdualität von Klassen und damit die inhärente Ausdrucksschwäche der Klassenmodellierung überwinden. Dieses Objekt-Konzept ist Grundlage für eine effektive und präzise Modellierung der Struktur und des Verhaltens von Objektgruppierungen. Diese Fähigkeit ist unentbehrlich, wenn große und umfangreiche Systemmodelle entwickelt werden, die durch die Beschreibung kooperierender Teilsysteme (als Anwendung einer Objektgruppierung) beschrieben werden.

5.2 Verwendung eines hierarchischen und integrierten Gesamtmodells

Die Modellierung abstrakter Objekte ist Grundlage für die ADORA-L-Systemmodellierung. In diesem Kapitel wird darauf aufbauend erläutert, wie ein mit ADORA-L beschriebenes Systemmodell prinzipiell aufgebaut und gegliedert ist und welche Konzepte hierfür eingesetzt werden.

Die Besonderheit in ADORA-L ist die Verwendung eines integrierten Gesamtmodells, in dem alle relevanten Aspekte gemeinsam in *einer* Struktur modelliert werden. Es wird gezeigt, warum dieser integrierte Ansatz dem verbreiteten Ansatz der Aspektmodellierung vorzuziehen ist (Kapitel 5.2.1). Die *Teil/Ganzes-Objekthierarchie* wird hierbei als grundsätzliches Modellskelett verwendet. In dieses Modellskelett werden grundlegende Modellaspekte im Sinne einer Projektion eingeblendet. Unterschieden wird dabei die *strukturelle*, die *verhaltensorientierte* und die *funktionale Einblendung* (siehe Kapitel 5.2.2). Abschließend wird erläutert, wie Klassen in ADORA-L beschrieben werden und wie die redundante Modellierung von Klassen und Objekten aufgelöst wird.

5.2.1 Integrierte Modellierung versus Aspektmodellierung

Eine Modellierungssprache muß in der Lage sein, eine Vielzahl unterschiedlicher Aspekte gemeinsam durch ein Systemmodell zu beschreiben. Bei der Konzeption einer solchen Sprache muß geklärt werden, durch welche Sprachmittel diese Aspekte beschrieben werden und an welcher Stelle sich im Systemmodell diese Aspekte wiederfinden. Im Bereich der Spezifikationssprachen existieren zwei grundsätzliche Lösungsansätze: Die *Aspektmodellierung* und die *integrierte Modellierung*.

In der *Aspektmodellierung* besteht ein Systemmodell aus einer Menge nebeneinanderstehender Aspektmodelle, die jeweils spezielle Aspekte herausgreifen und diese separat modellieren.

Zusammenhänge und Überschneidungen zwischen den Aspektmodellen sind auf Metaebene durch Integritätsbedingungen zu spezifizieren (siehe Abbildung 19).

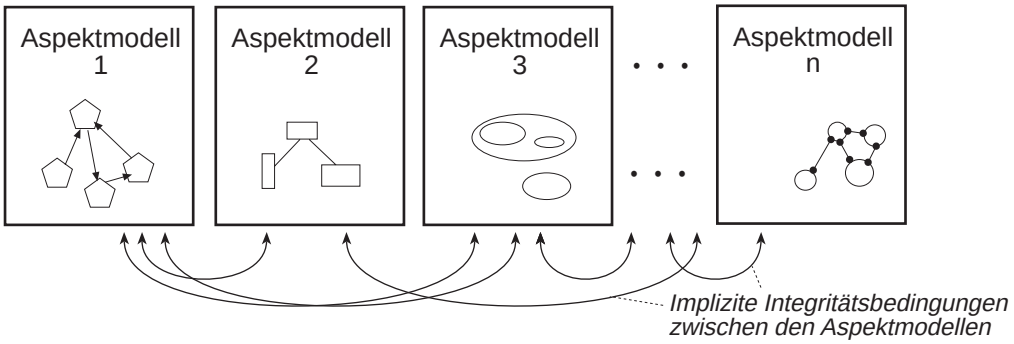


Abb. 19: Schema einer Aspektmodellierung: Zwischen den einzelnen Aspektmodellen existieren nicht sichtbare Integritätsbedingungen, die vom Modellierer jedoch berücksichtigt werden müssen

Beispiele für Aspektmodellierungen sind die objektorientierten Ansätze nach [Booch94], [Booch98], [Firesmith96], [Yourdon89] und [Rumbaugh91].

Eine *integrierte Modellierung* beschreibt ein Systemmodell durch ein zentrales Gesamtmodell. Alle zu modellierenden Aspekte werden in diesem Gesamtmodell integriert dargestellt. Notwendig hierfür ist das Vorhandensein einer aussagekräftigen skelettartigen Struktur, der *Basisstruktur*. Diese Basisstruktur wird dann durch alle notwendigen Aspektbeschreibungen kompatibel ergänzt. Jede Aspektbeschreibung ist so zu wählen, daß sie unabhängig von den anderen Aspektbeschreibungen in die Basisstruktur ein- und ausgeblendet werden kann. Die integrierte Modellierung erfolgt durch eine gemeinsame zusammenhängende Notation (Abbildung 20). Beispiele für die integrierte Modellierung sind die strukturierten Ansätze (SA) nach [Gane79], [DeMarco78] und [Harel96a].

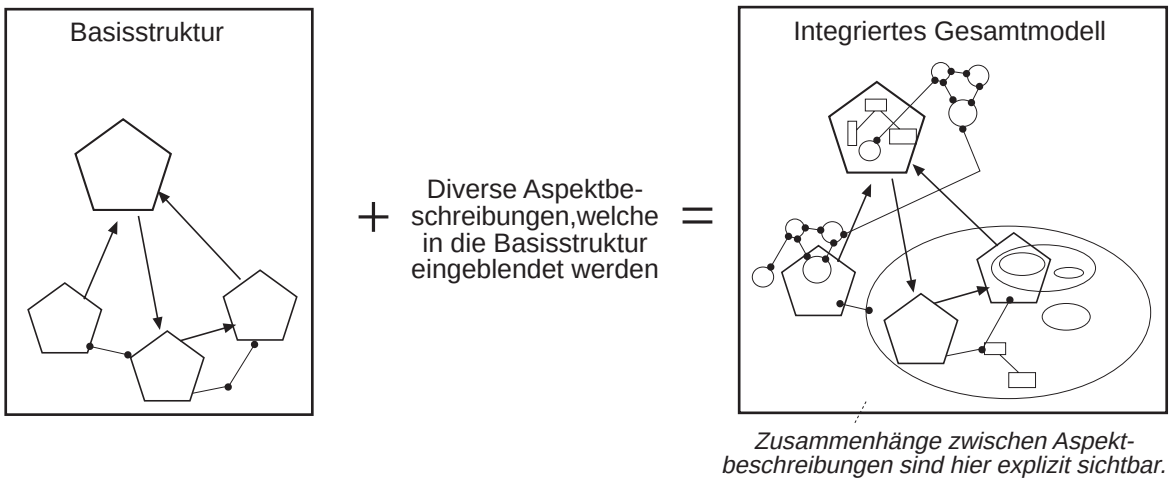


Abb. 20: Schema einer integrierten Modellierung: In eine gemeinsame Basisstruktur werden Aspekteinblendungen eingeblendet. Implizite Integritätsbedingungen sind nicht

vorhanden, das sie im Gesamtmodell explizit visualisiert werden.

Für ADORA-L wurde das Konzept der integrierten Modellierung gewählt. Zwar erfordert es mehr Sorgfalt in der Wahl der Basisstruktur und der einblendbaren Aspektbeschreibungen. Die integrierte Modellierung wurde jedoch der Aspektmodellierung aus zweierlei Gründen vorgezogen:

- Die gemeinsame Basisstruktur führt in der Regel zu besser strukturierten und dadurch einfacher verständlichen Modellen. Dies gilt insbesondere dann, wenn Systemmodelle für komplexe und umfangreiche Problemstellungen entwickelt werden. Integrierte Gesamtmodelle sind zudem durch ihre klarere Struktur besser verständlich.
- Sie sind in ihrer Konzeption robuster und weniger fehleranfällig. Die Systemmodelle enthalten im allgemeinen weniger Widersprüche und Unstimmigkeiten, da durch die zentrale Basisstruktur die zu modellierenden Informationen weniger redundant dargestellt werden. Bei Aspektmodellierungen hingegen wird häufig die gleiche Information in unterschiedlichen Aspektmodellen aufgeführt, ohne daß diese Redundanz dem Entwickler sichtbar ist.

5.2.2 Die Teil/Ganzes-Hierarchie kombiniert mit aspektbezogenen Einblendungen

Das folgende Kapitel stellt für ADORA-L die Basisstruktur *Teil/Ganzes-Objekthierarchie* sowie die Aspektbeschreibungen durch *aspektbezogene Einblendungen* vor. Als Basisstruktur wird in ADORA-L die *Teil/Ganzes-Objekthierarchie* verwendet. Sie wird wie folgt definiert:

> **Definition** *Teil/Ganze-Objekthierarchie*: Eine Teil/Ganzes-Objekthierarchie teilt Objekte in eine baumartige Hierarchie ein, in der einem übergeordneten Objekt endlich viele Objekte untergeordnet werden. Die Zuordnung von über- zu untergeordnetem Objekt hat die Bedeutung einer Teil/Ganzes-Struktur. Die *Komponente* ist physisch Teil der *Komposition* und nur in dieser aufgeführt. Logisch ist sie Bestandteil der Beschreibung der Funktionalität, des Verhaltens und der Struktur der Komposition. [1]

Die Teil/Ganzes-Objekthierarchie stellt einen Abstraktionsmechanismus im Sinne der Kompositionsabstraktion (siehe auch Kapitel 3.3.1) dar. Sie ist daher sehr gut geeignet, auch umfangreiche Modelle verständlich und überschaubar zu strukturieren. Ein weiterer Vorteil dieser Basisstruktur ist die einfache Integration der unten aufgeführten Aspektbeschreibungen. Abbildung 21 zeigt, wie das Beispiel aus Abbildung 16 mit Hilfe einer Teil/Ganzes-Objekthierarchie anschaulich modelliert werden kann. Das Objekt Drucker bzw. die Objektmenge Berichtedrucker (beide gehören der Klasse Drucker an) werden hier als Komponenten,

[1] Die Begriffe *Komposition* und *Komponente* wurden bereits in Kapitel 3.3.1 angesprochen

also als Teile andere Objekte bzw. Objektmengen beschrieben. Die relativ umständliche Modellierung des Verhältnisses zwischen Forschungsgruppen und Berichtedrucker (pro Forschungsgruppe existiert genau ein Drucker) wird hier nur durch eine Objektmenge mit dem Berichtedrucker als Komponente modelliert.

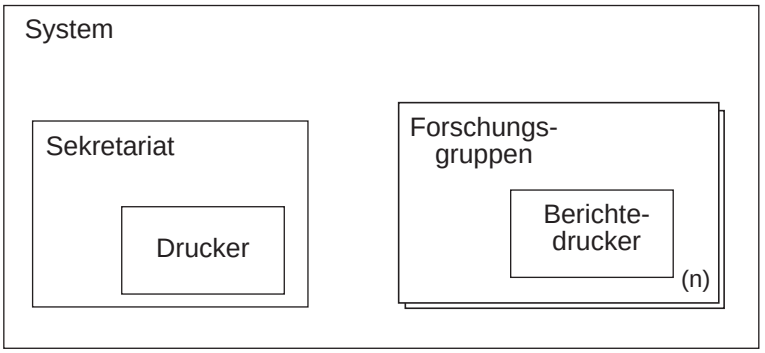


Abb. 21: Beispiel aus Abbildung 16, modelliert durch eine Teil/Ganzes-Objekthierarchie

Die Aspektbeschreibungen werden in ADORA-L durch *aspektbezogene Einblendungen* realisiert:

> **Definition** *Aspektbezogene Einblendung* – Eine aspektbezogene Einblendung definiert eine Teilmenge von Sprachkonstrukten einer Modellierungssprache, welche gemeinsam und überlappend mit den Sprachkonstrukten der Basisstruktur einen bestimmten Aspekt des Systemmodells heraushebt. Explizit ist es möglich, unterschiedliche Einblendungen gemeinsam in eine Basisstruktur einzublenden.

ADORA-L stellt in der jetzigen Fassung *strukturelle, verhaltensorientierte und funktionale Einblendungen* zur Verfügung. Geplant und angedacht ist zudem die *interaktionsorientierte Einblendung*:

- Die *strukturelle Einblendung* beschreibt die Struktur- und Benutzungsbeziehungen zwischen Objekten sowie zwischen Objekten des Systems und des Systemkontextes. Die Beziehungen werden als Nachrichtenflüsse sowie als strukturelle Prädikate zwischen den beteiligten Objekten interpretiert.
- Die *verhaltensorientierte Einblendung* beschreibt das Systemverhalten mit Hilfe von hierarchischen und parallelen Zustandsautomaten analog zu den Harelschen Statecharts [Harel87]. Die Komponenten eines Objektes werden hierbei als komplexe und eingeschachtelte Zustände interpretiert.
- Die *funktionale Einblendung* beschreibt die Funktionalität von Objekten im Sinne von Dienstleistungen. Diese Dienstleistungen können von anderen Objekten mit Hilfe von Nachrichten in Anspruch genommen werden.

- Die *interaktionsorientierte Einblendung* beschreibt die Anwendungsfälle (engl: Use Cases), welche vom Systemmodell unterstützt und bereitgestellt werden. Ein Anwendungsfall beschreibt eine Interaktionssequenz zwischen dem System und einem Benutzer. Beschrieben werden Anwendungsfälle unter Zuhilfenahme der Objekte/Objektmengen der Basisstruktur sowie der Systemumgebung.

Diese aspektbezogenen Einblendungen wurden gewählt, da sie relativ entkoppelt voneinander sind und dadurch nur wenige Überschneidungen in den verwendeten Sprachkonstrukten entstehen. Zudem ist diese Unterteilung überschaubar und intuitiv verständlich. Die einzelnen Einblendungen können unabhängig voneinander in die Teil/Ganzes-Objekthierarchie (also in die Basisstruktur) im Sinne der obigen Definition «eingeblendet» werden.

Das Konzept der *interaktionsorientierten Einblendung* wurde in der jetzigen ADORA-L-Sprachbeschreibung nicht aufgenommen und wird vermutlich erst in einer zukünftigen Version integriert. Grund hierfür ist die bisher ungeklärte Frage, ob Anwendungsfälle überhaupt in ein Systemmodell integriert werden sollen und wie Anwendungsfälle in die Teil/Ganzes-Objekthierarchie integriert werden können. Die Klärung dieser Fragen würde jedoch den Rahmen dieser Arbeit sprengen. Die Konzeption der aspektbezogenen Einblendungen wird in den Kapiteln 5.3 bis 5.6 vertieft und im Detail diskutiert.

5.2.3 Namensräume innerhalb der Teil/Ganzes-Hierarchie

Die Teil/Ganzes-Hierarchie in ADORA-L legt die grundlegende Struktur eines Objektmodells fest. Darüber hinaus klärt sie, auf welche Art und Weise Objekte und Objektmengen in ihrer Umgebung identifiziert werden. Grundsätzlich werden Modellelemente durch eine Bezeichnung, also einen eindeutigen Namen identifiziert. Für umfangreiche Systemmodelle ist die alleinige Identifizierung über die Modellelementnamen jedoch wenig sinnvoll, da jeder Objektname (Objektmengenname, usw.) im Objektmodell nur einmal vorkommen dürfte. Dies führt in der Regel zu einer wenig intuitiven und schlecht verständlichen Namensgebung. Zu diesem Zweck wird in ADORA-L das Konzept der *Namensraumbegrenzung* eingeführt:

> **Definition** *Namensraumbegrenzung* – Eine Namensraumbegrenzung eines Objektes (einer Objektmenge) gibt an, in welchen Bereichen eines Modells das Modellelement durch seinen Namen direkt identifiziert werden kann. Außerhalb dieser Begrenzung kann das Objekt (die Objektmenge) nur indirekt durch einen *Namenspfad* referenziert werden.

Prinzipiell kann das Konzept der Namensraumbegrenzung in einer Modellierungssprache implizit oder explizit umgesetzt werden. Eine implizite Namensraumbegrenzung verwendet ein geeignetes Strukturierungskonzept der Sprache, welches den Namensraum automatisch begrenzt (Beispiele hierfür sind etwa Teilsystemkonstrukte wie das Package [Booch98] oder

die Klassenkategorien [Booch94]). Die explizite Namensraumbegrenzung verwendet ein eigenständiges Sprachkonstrukt, mit dessen Hilfe Namensräume variabel vom Modellierer selber definiert werden können.

In ADORA-L wird eine implizite Namensraumbegrenzung in Form der Teil/Ganzes-Hierarchie verwendet. Die Kompositionen dieser Hierarchie dienen hierbei als feste Namensgrenzen. Wird auf ein Objekt (eine Objektmenge) innerhalb der Komposition, in der sie enthalten ist, referenziert, so geschieht dies direkt über dessen Name. Außerhalb dieser Begrenzung wird ein Objekt (eine Objektmenge) durch einen Namenspfad identifiziert. Der Namenspfad enthält, neben dem eigentlichen Objektnamen (Objektmengennamen) die übergeordneten Kompositionen des Objektes (der Objektmenge) in der Teil/Ganzes-Hierarchie. Referenziert eine Komponente A eine Komponente B und liegen beide nicht in derselben Komposition, so muß der entsprechende Namenspfad alle übergeordneten Kompositionen von B enthalten, bis eine gemeinsame übergeordnete Komposition von A und B erreicht ist.

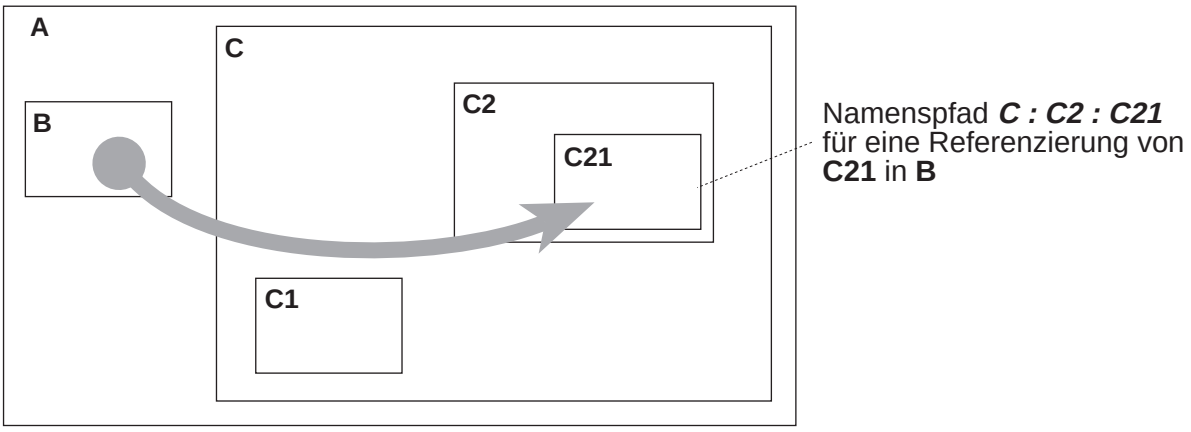


Abb. 22: Beispiel einer Objektreferenzierung (außerhalb der Namensgrenzen): Referenzierung der Komponente C21 durch B

In Abbildung 22 wird ein Beispiel einer Objektreferenzierung außerhalb des Namensraumeserläutert (der Namensraum der Komponente C21 ist durch die Komposition C2 festgelegt). Die einzelnen Elemente des Namenspfades werden syntaktisch durch einen Doppelpunkt getrennt. In ADORA-L wurde aus folgenden Gründen ein implizites Namensraumbegrenzung gewählt:

- Durch die Teil/Ganzes-Hierarchie ist eine klare und präzise Struktur vorgegeben, die ein Objektmodell in Teilbereiche gliedert. Eine solche Hierarchie eignet sich daher hochgradig für die Einführung von Namensräumen
- Durch die implizite Namensraumbegrenzung sind keine zusätzlichen Notationen notwendig. Die Modellierungssprache selber wird daher nicht zusätzlich komplexer.

Ein implizite Namenskonzept ist zwar weniger flexibel und erhöht unter Umständen den Modellierungsaufwand. Da jedoch voraussichtlich die Namenspfad-Referenzierung selten ein-

gesetzt wird, erscheint es nicht sinnvoll, ADORA-L durch eine wenig aussagekräftige Namensraumnotation zu ergänzen (Anmerkung: In den meisten Fällen werden in ADORA-L Objekte nicht direkt, sondern über eine Objekt-Objektbeziehung referenziert, siehe Kapitel 5.3).

Innerhalb eines Namensraumes, d.h. im Fall von Objekten und Objektmengen, müssen folgerichtig die Objektnamen eindeutig sein:

> **Integritätsbedingung** *Eindeutigkeit von Komponentennamen* – Innerhalb einer Komposition muß jeder Name einer Komponente eindeutig sein. Dies gilt gleichermaßen für Objekte als auch für Objektmengen als Komponenten.

Weiterhin gelten auch für andere Sprachkonstrukte Integritätsbedingungen bezüglich der Namensgebung. So darf es beispielsweise keine Namenskonflikte von Attributnamen oder Operationsnamen innerhalb eines Objektes (einer Objektmenge) geben.

5.2.4 Modellierung des Systemkontextes

Die Beschreibung des Zusammenhangs eines Systems zu seinem Kontext und damit die Festlegung der Systemgrenzen ist eine wichtige Aufgabe der Systemmodellierung. Von zentraler Bedeutung ist die Modellierung der Systeminteraktion mit der Umgebung, da hier das eigentliche Leistungsangebot aufgeführt ist, welches zur Verfügung gestellt wird. Dieses Leistungsangebot ist einerseits Grundlage für den Abgleich mit den Vorstellungen der Auftraggeber und der Systembenutzer, andererseits ist es Vorgabe für die zu erbringende Funktionalität des gesamten Systemmodells.

Der Systemkontext wird in ADORA-L durch *Kontextentitäten* modelliert (ähnlich der Kontextmodellierung der *strukturierten Analyse*). Bezüglich der Interaktion werden diese *Kontextentitäten* gleich behandelt wie die «normalen» Objekte des Systemmodells, d.h. statisch durch eine Beziehungs- und dynamisch durch eine Nachrichtenmodellierung (siehe Kapitel 5.3). Nicht modelliert wird jedoch die innere Struktur und das Verhalten der Kontextentitäten. ADORA-L unterscheidet zwei Arten von *Kontextentitäten*:

- *Externe Akteure* – Ein *externer Akteur* ist ein eigenständiges externes System, eine Person, ein Sensor o. ä., welches mit dem Systemmodell bzw. mit den dortigen Objekten (bzw. Objektmengen) interagiert. Außer der statischen und der dynamischen Kommunikation werden keine weiteren Eigenschaften des externen Akteurs im Systemmodell erfaßt.
- *Externe Objekte* – Ein *externes Objekt* ist ein System, welches mit dem Systemmodell bzw. mit den dortigen Objekten interagiert. Im Gegensatz zum *externen Akteur* ist es jedoch Bestandteil und damit Objektkomponente des Systemmodells. Es verhält sich daher wie ein gewöhnliches Objekt, dessen Interna nicht modelliert werden. Ein externes Objekt wird ver-

wendet, um Drittkomponenten in das Systemmodell aufzunehmen und diese dort als Black-Box zu verwenden (zwar sind die Leistungen eines externen Objektes bekannt, nicht aber dessen Struktur und dessen Verhalten).

Analog zu den Objektmengen existieren entsprechende *externe Objektmengen* und *externe Akteurmengen*. Prinzipiell wird es dem Modellierer überlassen, ob er eine *Kontextentität* als *externen Akteur* oder als *externes Objekt* modelliert. *Externe Akteure* betonen die Eigenständigkeit eines Systemkontextes, während externe Objekte in das Systemmodell integriert sind und eher den Charakter einer tatsächlichen Systemkomponente haben, dessen Beschreibung in nur stark vergröberter Form vorliegt. Externe Akteure sind beispielsweise gut geeignet, um menschliche Systembenutzer darzustellen. In eingebetteten Systemen finden sich andererseits häufig Kandidaten für *externe Objekte*, nämlich genau die Teile, welche entweder bereits als Realisierung vorliegen oder in anderer Form dokumentiert werden.

5.3 Kommunikation zwischen Objekten

Die Verwendung des objektorientierten Paradigmas erfordert eine Klärung, in welcher Weise Objekte miteinander paradigmenkonform kommunizieren. Weiterhin muß durch die Spezifikationssprache festgelegt werden, wie diese Kommunikation sprachlich festgehalten wird. Im folgenden wird zunächst die Grundidee des in ADORA-L verwendeten Nachrichtenkonzeptes vorgestellt. Es wird geklärt, wie die Objektkommunikation statisch durch *Beziehungen* modelliert wird und wie Beziehungen hierarchisch in Objekthierarchien integriert werden. Speziell für die Modellierung der Kommunikation in hierarchischen Objektmodellen wird das Konzept der hierarchischen Beziehungen eingeführt. Im zweiten Teil wird erläutert, wie Nachrichten in ADORA-L dynamisch modelliert werden.

5.3.1 Grundidee

In ADORA-L erfolgt die Kommunikation zwischen Objekten ausschließlich über Nachrichten. Die Kommunikation über gemeinsame Speicher oder Datenbereiche ist nicht vorgesehen, da sie nur schwer mit dem objektorientierten Paradigma in Einklang zu bringen ist. Das Nachrichtenkonzept in ADORA-L ist *explizit*: Der Empfänger einer Nachricht wird vom Versender direkt angesprochen, indem zwischen Versender und Empfänger ein Nachrichtenkanal definiert wird. Ein *implizites*[1] Nachrichtenkonzept wurde in ADORA-L bewußt vermieden. Zwar vereinfacht es die Modellierung von Nachrichtenflüssen, jedoch sind die Systemmodelle im allgemeinen bei Modifikationen fehleranfälliger und neigen zu unsichtbaren Seiteneffekten [Selic94].

In ADORA-L wird die Kommunikation zwischen Objekten sowohl statisch als auch dynamisch modelliert:

- Statisch in der *strukturellen Einblendung*, indem zwischen den Objekten (Objektmengen) Beziehungen modelliert werden, welche wiederum Beziehungen zwischen Objektinstanzen auf der Ausprägungsebene repräsentieren.
- Dynamisch in der *verhaltensorientierten Einblendung*, indem die Nachrichtenversendung und der Nachrichtenempfang in den Objekten und Objektmengen spezifiziert wird. Die Nachrichten wiederum sind Auslöser für Objektoperationen und/oder führen zu Veränderungen des Objektzustandes.

[1] Ein implizites Nachrichtenkonzept spezifiziert beim Auslösen einer Nachricht ohne explizite Nennung des Zielobjektes. Die Nachricht wird an alle Objekte des Modells verschickt. Jedes Objekt entscheidet für sich selber, ob es auf die jeweilige Nachricht reagiert oder nicht.

5.3.2 Statische Modellierung von Zusammenhängen

ADORA-L modelliert Nachrichten und Referenzen statisch in der *strukturellen Einblendung* durch Beziehungen:

> **Definition** *Beziehung* – Eine Beziehung beschreibt die gegenseitige Referenzierbarkeit der in Beziehung gesetzten Objekte (Objektmengen) und damit potentielle Informationsflüsse. Sie stellt eine Abstraktion einer Menge von Beziehungen auf Ausprägungsebene dar, welche zwischen den entsprechenden Objektinstanzen gültig sind. Ein Informationsfluß zwischen zwei Objekten (Objektmengen) repräsentiert ein oder mehrere Nachrichten, welche zwischen ihnen wechselseitig versendet und empfangen werden. Die Referenzierbarkeit macht eine Aussage darüber, ob ein Objekt (Objektmenge) die Existenz und die Identität eines anderen Objektes (Objektmenge) kennt und entsprechende Zugriffsrechte besitzt. Explizit ist es möglich, daß zwischen zwei Objekten (Objektmengen) mehrere Beziehungen existieren.

Die Referenzierbarkeit zwischen zwei Objekten (oder Objektmengen) impliziert nicht zwangsläufig einen Informationsfluß. Ein struktureller Zusammenhang liegt beispielsweise vor, wenn ein Objekt (speziell öffentlich zugreifbare) Attribute eines anderen Objektes abfragt, ohne jedoch Nachrichten mit diesem auszutauschen. Beziehungen können auch dann verwendet werden, wenn allein ein logischer Zusammenhang der Art «*Abteilung besteht aus n Mitarbeitern*» modelliert werden soll (siehe auch *strukturelle Beziehung).* Eine Beziehung kann wahlweise einen Zusammenhang zwischen *zwei Objekten*, zwischen *einem Objekt und einer Objektmenge* oder zwischen *zwei Objektmengen* ausdrücken. Auf Ebene der Objektinstanzen werden diese drei Varianten unterschiedlich interpretiert (siehe Abbildung 23).

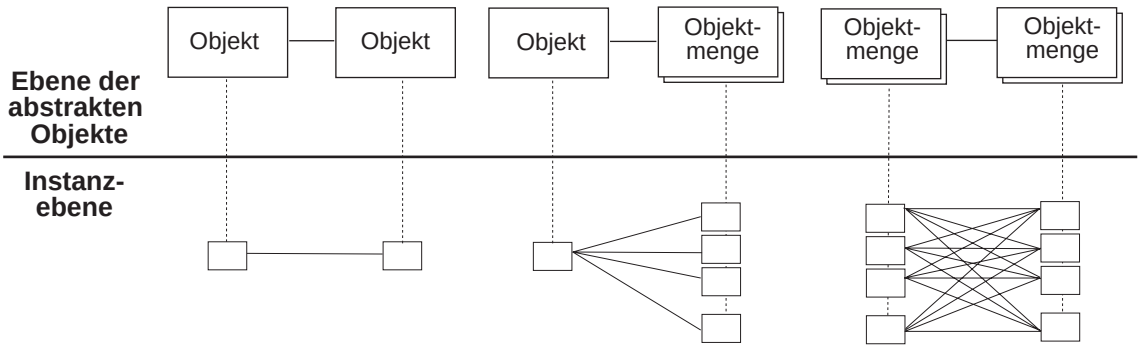


Abb. 23: Die verschiedenen Interpretationen einer Beziehung auf Instanzebene: Zwei Objekte stehen in Beziehung 1:1, Objekt und Objektmenge 1:n und zwei Objektmengen 1:n oder m:n

Eine *Beziehung* zwischen zwei Objekten repräsentiert *eine* Beziehung zwischen den entsprechenden Instanzen auf Ausprägungsebene. Eine Beziehung zwischen Objekt und Objektmenge bzw. zwischen zwei Objektmengen beschreibt eine Menge von *Beziehungen* zwischen Objektinstanzen, nämlich zwischen einer Objektinstanz und der Menge *aller* Objektinstanzen

(beschrieben durch eine Objektmenge). Eine Beziehung auf Schemaebene repräsentiert also *alle möglichen Kombinationen* von Beziehungen auf Ausprägungsebene. Vorstellbar wäre auch eine individuelle und explizite Modellierung der Beziehungen auf Ausprägungsebene und damit verbunden ein explizites Erzeugen und Löschen von Beziehungen, falls entsprechende Instanzen erzeugt oder gelöscht werden. Problematisch hierbei ist jedoch ein sehr hoher Modellierungsaufwand und die Frage, welche zusätzlichen sprachlichen Konzepte und Konstrukte zur Verfügung gestellt werden müßten. Eine solche Erweiterung würde den Rahmen dieser Arbeit sprengen und wird daher lediglich als eine potentielle Erweiterung vorgestellt (siehe Kapitel 8).

Die Modellierung der Zusammenhänge zwischen Objekten macht prinzipiell nur Sinn, wenn *alle* Nachrichten abstrakt durch *Beziehungen* beschrieben werden (andernfalls wäre die statische Modellierung unvollständig und inkonsistent). Es muß also folgende Modellintegrität gewährleistet sein:

> **Integritätsbedingung** *Vollständigkeit bzgl. Beziehungen* – Können zwei Objektinstanzen Nachrichten austauschen, d.h. wird in den entsprechenden Objekten (Objektmengen) eine Nachrichtenversendung modelliert, so muß hier auch eine Beziehung modelliert werden.

Eine modellierte Beziehung impliziert aber nicht zwangsläufig eine direkte Verbindung der beiden Objekte auf Instanzebene, da Nachrichten auch über dritte Objekte (welche mit den beiden Objekten/Objektmengen in Beziehung stehen) weitergeleitet werden. Im umgekehrten Fall muß nach obiger Integritätsbedingung jedoch stets eine Beziehung existieren.

In ADORA-L werden zwei Arten von Beziehungen unterschieden: Die *strukturelle Beziehung* und die *Benutzung*:

> **Definition** *Strukturelle Beziehung* – Eine *strukturelle Beziehung* ist eine Beziehung, welche einen strukturellen Zusammenhang zwischen zwei Objekten (Objektmengen) des Objektmodells beschreibt. Die strukturelle Beziehung drückt aus, daß zwei Objekte (Objektmengen) zusammenarbeiten und kooperieren, ohne auf die Art der Kommunikation näher einzugehen. Soll die Zusammenarbeit nur für eine der beiden Komponenten sichtbar sein, ist die Beziehung in diese Richtung *gerichtet*. Andernfalls ist sie für beide Komponenten sichtbar und *ungerichtet*.

> **Definition** *Benutzung* (synonym: *Benutzungsbeziehung*) – Eine Benutzung ist eine gerichtete Beziehung zwischen zwei Objekten/Objektmengen A und B, in der A eine von B bereitgestellte Leistung in Anspruch nimmt. Benutzungen sind stets gerichtete Beziehungen.

Eine *strukturelle Beziehung* spezifiziert einen logischen Zusammenhang in der Art einer Zusammenarbeit oder einer Kooperation und damit einen beliebigen Austausch von Nachrichten und Nachrichtendaten. Eine *Benutzung* beschreibt im Gegensatz zur *strukturellen Beziehung* einen gerichteten Zusammenhang. Sie stellt einen Spezialfall einer Kooperation in Form einer Auftraggeber/Auftragnehmer-Struktur (engl. client-server) dar.

Die Beziehungen werden generell durch Namen und durch Kardinalitätsangaben näher spezifiziert. *Ungerichtete strukturelle Beziehungen* werden durch zwei Bezeichnungen näher beschrieben, die jeweils das Verhältnis von A nach B und umgekehrt ausdrücken. *Gerichtete strukturelle Beziehungen* und *Benutzungen* hingegen sind gerichtet und werden nur in eine Richtung benannt (die Richtung geht von der Komponente aus, welche Zugriff auf die andere Komponente hat). Die Kardinalitäten geben an, wieviele Beziehungen auf Ebene der Instanzen erlaubt bzw. gefordert sind. Sie machen also eine Aussage über konkrete Beziehungen über die Quantität der Beziehungen auf Ausprägungsebene, in dem spezifiziert wird,

- ob eine Beziehung in allen oder nur in manchen Modellausprägungen vorhanden ist (*Optionalität*).
- wieviele Instanzen einer Objektmenge mit einer Objektinstanz in Verbindung stehen. Spezifiziert wird, wieviele Beziehungen minimal und maximal in allen Modellausprägungen vorhanden sein dürfen (*Multiplizität zwischen Objekt und Objektmenge*)
- wieviele Instanzen einer Objektmenge mit wievielen Instanzen einer anderen Objektmenge durch Beziehungen verbunden sind. Spezifiziert wird in beide Richtungen, wieviele Beziehungen minimal und maximal in allen Ausprägungen vorhanden sein dürfen (*Multiplizität zwischen Objektmengen*), siehe Abbildung 24.

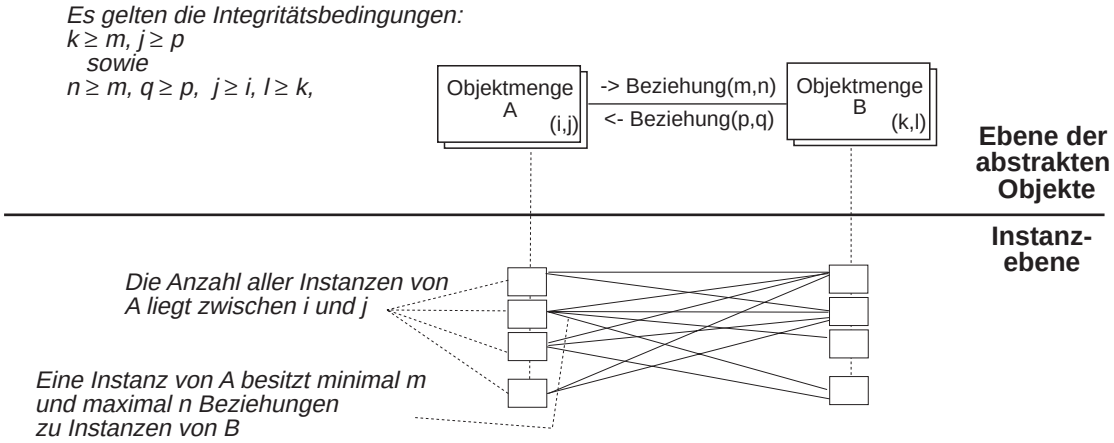


Abb. 24: Die Bedeutung der Kardinalitäten einer Objektmengen-Beziehung. Objekte als Sonderfall einer Objektmenge besitzt die Kardinalitäten i = j = 1

Die Kardinalitätsmodellierung in ADORA-L erinnert stark an die der klassischen Datenmodellierung (beispielsweise an [Chen76]). Im Gegensatz zur Datenmodellierung werden hier Beziehungen jedoch über *abstrakte Objekte* und nicht über *Klassen* definiert. Spezifiziert wird

also nur die Optionalität und die Multiplizität eines speziellen Objekt/Objektmengen-Zusammenhangs, nicht aber alle möglichen Instanzbeziehungen einer Klasse. Sie sind also deutlich präziser einsetzbar (siehe Kapitel 5.1.1).

5.3.3 Hierarchische Modellierung von Beziehungen

Die bisher vorgestellten Konzepte allein reichen nicht aus, um innerhalb einer hierarchischen Basisstruktur Beziehungen sinnvoll statisch zu modellieren. Zwar ist die Basisstruktur in der Lage, von den eingekapselten Komponenten eines Objektes bzw. einer Objektmenge zu abstrahieren. Nicht möglich ist jedoch eine Abstraktion der Objektkommunikation. Notwendig ist hier ein abstrahierendes Konzept, welches Beziehungen auf Detailebene (d.h. zwischen tief eingekapselten Objekten/Objektmenge) auf Grobebene (d.h. auf Ebene der Kompositionen) in abstrakter Form sichtbar macht.

Um dem gerecht zu werden, wird in ADORA-L das *Konzept der hierarchischen Beziehungen* eingeführt. Es erlaubt die Zuordnung einer Beziehung einer tieferliegenden Ebene zu einer auf höher liegenden Ebene, einer *Oberbeziehung*. Wird die Zuordnung von Beziehung zu *Oberbeziehung* so gewählt, daß die *Oberbeziehung* eine vergröberte Beschreibung der Beziehung auf Detailebene darstellt, so handelt es sich um eine Abstraktion von konkreter zu vergröberter Objektkommunikation.

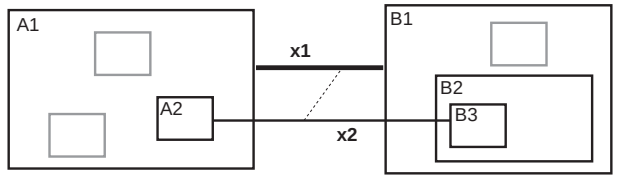
> **Definition** *Oberbeziehung* – Eine *Oberbeziehung* ist eine *Beziehung*, die eine oder mehrere Beziehungen auf Detailebene in vergröberter Form zusammenfaßt. Explizit ist es möglich, weitere Oberbeziehungen wiederum durch eine darüberliegende *Oberbeziehung* zusammenzufassen.

Eine Oberbeziehung ist für sich selbst genommen wiederum eine Beziehung, kann also sowohl eine *strukturelle Beziehung* als auch eine *Benutzung* sein. Explizit ist die Wahl der Beziehungsart der Oberbeziehung unabhängig von der/den Beziehungsarten der Beziehungen, welche sie zusammenfaßt. Es ist also auch beispielsweise möglich, eine Reihe von Beziehungen auf Detailebene durch eine Benutzungs-Oberbeziehung zusammenzufassen. eine solche Restriktion wäre zwar in bestimmten Fällen sinnvoll, es finden sich jedoch auch Ausnahmen, in der eine «frei wählbare» Zusammenfassung Sinn macht.

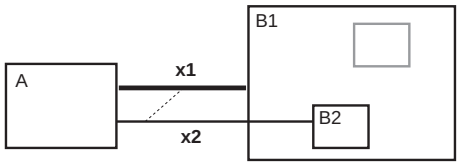
Eine Beziehung ist dann Oberbeziehung einer anderen Beziehung, wenn folgende Integritätsbedingung gilt:

> **Integritätsbedingung** *Zuordnung von Oberbeziehungen* – Eine Beziehung A ist dann und nur dann Oberbeziehung einer Beziehung B, wenn

- die durch B zusammenhängenden Objekte jeweils Komponenten (oder Komponenten von Komponenten, etc.) der Objekte sind, welche durch A miteinander in Beziehung stehen (siehe Abbildung 25a)
- ein Objekt gleichzeitig mit einem zweiten Objekt durch A, mit einem dritten Objekt durch B zusammenhängt und das dritte Objekt Komponente (oder Komponente einer Komponente, etc.) des zweiten Objektes ist (siehe Abbildung 25b).



a) *Die Oberbeziehung x1 verbindet übergeordnete Kompositionen von A2 und B3*



b) *Die Oberbeziehung x1 verbindet nur eine übergeordnete Komposition (B1) mit A, also der ursprünglichen Beziehung x2*

Abb. 25: Die Integritätsbedingung *Zuordnung von Oberbeziehungen* – Klärung der möglichen Zuordnungen einer Beziehung zu einer Oberbeziehung[1]

Die Integritätsbedingung *Zuordnung von Oberbeziehungen* ist jedoch nicht ausreichend für eine abstrahierende Beziehungshierarchie. Zusätzlich muß die Eigenschaft, daß bestimmte Beziehungen auf Detailebene tatsächlich auch geeigneten Oberbeziehungen zugeordnet werden, garantiert werden. Hierzu ist folgende Integrität zu erfüllen:

Integritätsbedingung *Vollständigkeit der Oberbeziehungen* – Eine Beziehungshierarchie ist dann vollständig, wenn jede Beziehung, die zwei in unterschiedlichen Kompositionen liegende Objekte (Objektmengen) verbindet, der *nächstmöglichen* Oberbeziehung zugeordnet ist. Existieren mehrere nächstmögliche Oberbeziehungen, muß eine davon explizit ausgewählt werden.

Definition *Nächstmögliche Oberbeziehung* – Eine Oberbeziehung ist für eine Beziehung *nächstmöglich*, wenn keine weitere Oberbeziehung existiert, welche «näher an der Beziehung liegt». Das Maß der Distanz zwischen einer Beziehung und einer Oberbeziehung ist die Summe der Verschachtelungstiefen, in der die Objekte/Objektmengen der Beziehung in die beiden Kompositionen der Oberbeziehung eingekapselt sind.

[1] Oberbeziehungen werden hier durch fette, normale Beziehungen durch einfache Linien, der Zusammenhang Beziehung-Oberbeziehung durch eine gestrichelte Linie dargestellt.

Die Integrität der Vollständigkeit einer Beziehungshierarchie und die Zuordnung einer Beziehung zu einer nächstmöglichen Oberbeziehung wird in Abbildung 26 näher erläutert.

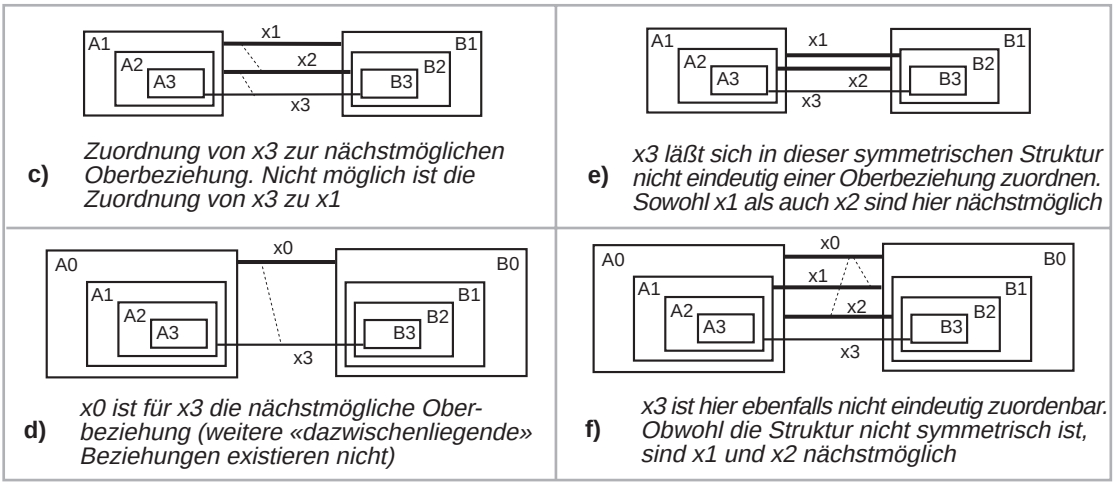


Abb. 26: Verdeutlichung der Vollständigkeit einer Beziehungshierarchie. In c) und d) können Beziehungen zu *nächstmöglichen* Oberbeziehungen zugeordnet werden, in e) und f) muß eine Oberbeziehung explizit ausgewählt werden.

Anmerkungen zu Abbildung 26: In c) kann die Beziehung x3 nicht x1 zugeordnet werden. Die Summe der Verschachtelungstiefen ist für x1/x3 4, für x2/x3 ist sie 3. In d) ist x3 die Verschachtelungstiefe zwar sehr hoch, da jedoch keine «naheliegendere» Beziehung existiert, ist die Zuordnung legitim. In e) und f) schließlich ist x3 keiner Oberbeziehung eindeutig zuordenbar. Die Summe der Verschachtelungstiefen ist in e) 3, in f) 4.

Das Konzept der Zuordnung einer Beziehung zur einer *nächstmöglichen* Oberbeziehung wurde aus folgenden Gründen eingeführt:

- jeder Beziehung sollte nicht mehr als einer Oberbeziehung zugeordnet werden – Durch eine baumartige Beziehungshierarchie wird die Verständlichkeit gegenüber einer freien Zuordnung erheblich verbessert. Zudem erhält eine nächstmögliche Oberbeziehung die Bedeutung einer Bündelung von Beziehungen, indem die zugrundeliegenden Beziehungen vereinigt werden.
- Die nächstmögliche Zuordnung erlaubt es, die Beziehungshierarchie in den meisten Fällen automatisch (etwa durch ein entsprechendes Modellierungswerkzeug) zu erzeugen. Diese Eigenschaft ist wichtig, da eine manuelle Zuordnung von Beziehungen zu Oberbeziehungen sehr aufwendig sein kann.

Die Teil/Ganzes-Objekthierarchie stellt eine Abstraktion im Sinne des Teil/Ganzes-Prinzips dar und damit eine sinnvolle Strukturierung der ADORA-L-Basisstruktur. Die Beziehungshierarchie baut auf diese Hierarchie auf und modelliert die Kommunikation in der strukturellen Einblendung ebenfalls auf unterschiedlichen Abstraktionsebenen. Dies ermöglicht eine klare

Verständlichkeit der Struktur des Objektmodells und bietet zudem vergröberte Sichten darauf, wie Teilsysteme des Systemmodells miteinander kommunizieren.

5.3.4 Dynamische Modellierung von Nachrichten

In ADORA-L wird ein explizites Nachrichtenkonzept umgesetzt. Prinzipiell sind zwei Varianten vorstellbar, um in ADORA-L ein explizites Nachrichtenkonzept zu integrieren:

(i) Durch direkte Benennung der Zielkomponente – In einem Objekt (einer Objektmenge) wird eine Nachricht versendet, indem dort die Zielkomponente direkt, d.h. durch explizite Nennung spezifiziert wird.

(ii) Durch *Pipelining* (das Nachrichtenkanal-Prinzip) – Die Versendung von Nachrichten erfolgt über Nachrichtenkanäle, die hierfür spezifiziert werden. Das Zielobjekt wird in der Objektspezifikation also nicht explizit benannt, sondern ergibt sich aufgrund der Spezifikation des Nachrichtenkanals.

Die Variante (i) ermöglicht eine einfache und schnelle Entwicklung von Objektmodellen. Die «Hartverdrahtung» der Kommunikationsverbindungen ist es jedoch problematisch, wenn Objektmodelle verändert oder erweitert werden. Im ungünstigsten Fall müssen beim Austausch eines Objektes gleichzeitig auch die Beschreibungen der umgebenden Objekte (Objektmengen) modifiziert werden [Belina91]. Längerfristig gesehen erscheint die Variante (ii) angebrachter. Zwar müssen hier zusätzlich Nachrichtenkanäle modelliert werden, dafür werden die Modelle deutlich robuster. Die Beschreibung des Objektverhaltens wird entkoppelt von seiner Umgebung modelliert, da nur noch durch Beziehungen die Zielkomponenten spezifiziert werden. Diese definierten Beziehungen können relativ einfach in ein neues Umfeld eingebettet werden. In ADORA-L werden daher Nachrichten mit Hilfe des Nachrichtenkanal-Prinzips modelliert, wie sie beispielsweise auch in SDL [Belina91] eingesetzt wird.

Nachrichtenkanäle werden in ADORA-L durch Beziehungen der *strukturellen Einblendung* spezifiziert. Soll also innerhalb einer Verhaltensbeschreibung eine Nachricht von einem Objekt zu einem anderen versendet werden, muß zwischen diesen beiden Objekten eine Beziehung existieren (siehe auch Integritätsbedingung, S. 84).

Nachrichten werden in ADORA-L folgendermaßen spezifiziert:

(i) Durch die Wahl einer aussagekräftigen Bezeichnung, um sie von anderen Nachrichten unterscheidbar zu machen.

(ii) Durch die Art der Kommunikation. Als Kommunikationsarten werden *synchrone* und *asynchrone* Nachrichten sowie *asynchrone Multicasting*-Nachrichten zur Verfügung gestellt

(iii) Durch ein Objekt, welches die Nachricht auslöst und versendet (innerhalb eines Zustandsautomaten oder als Objektoperation (siehe nächstes Kapitel)

(iv) Durch eine Beziehung aus der *strukturellen Einblendung*, über welche die Nachricht versendet wird (Benutzung, strukturelle Beziehung)

(v) Durch Daten in Form von Parametern, die mit der Nachricht optional mitversendet werden

(vi) Durch einen optionalen Rückgabewert, der bei synchronen Nachrichten direkt an den Versender der Nachricht zurückgegeben wird

Zu (ii): ADORA-L unterscheidet grundsätzlich zwischen *synchroner* und *asynchroner Kommunikation*. *Synchrone Kommunikation* geht davon aus, daß Verhaltensweisen und Abläufe beider Objekte aufeinander abgestimmt sind, und beide zeitgleich, d.h. in gemeinsamen Zeitschritten ablaufen. Bei *asynchroner Kommunikation* sind Verhaltensweisen und Abläufe der beteiligten Objekte (Objektmengen) nicht aufeinander abgestimmt, insbesondere können die Zeitschritte unterschiedlich sein sowie nichtdeterministische Verzögerungen beim Austausch von Nachrichten entstehen [Engesser88]. Synchrone und asynchrone Kommunikation wird durch die Modellierung von *synchronen* und *asynchronen* Nachrichten umgesetzt. Welche Art der Kommunikation vorliegt, ist also Eigenschaft jeder spezifischen Nachricht, nicht jedoch Eigenschaft von Beziehungen oder von Objekten (Objektmengen).

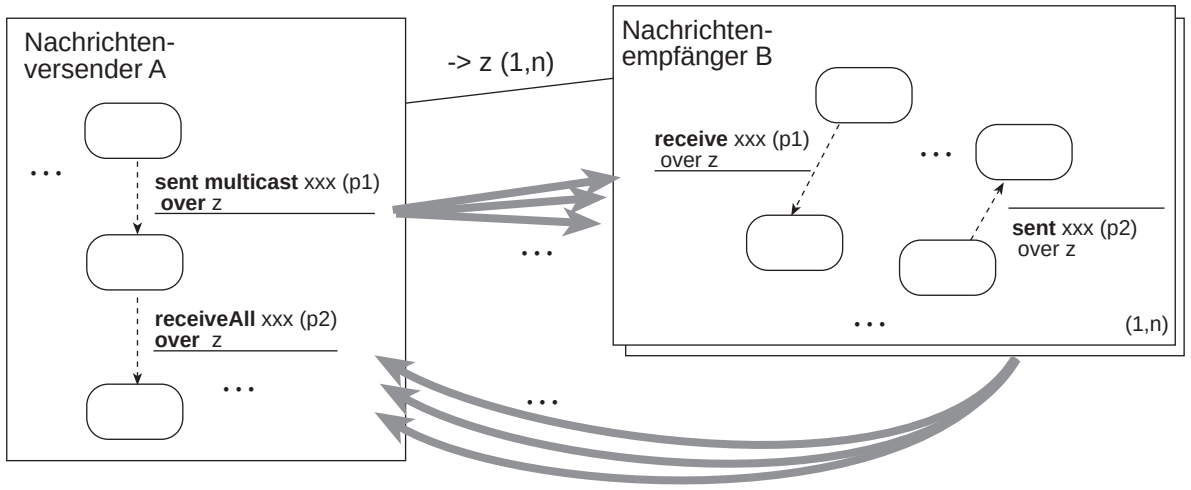


Abb. 27: Beispiel einer multicasting-Kommunikation zwischen einer Objektinstanz (in A) und einer Menge von Instanzen (in B).

Besondere Bedeutung haben Beziehungen, über die Nachrichten an Objektmengen versendet werden. Die versendete Nachricht ist dort an alle Objektinstanzen der Objektmenge adressiert. Die Selektion, welche Objektinstanzen auf diese Nachricht reagieren, erfolgt in der Funktions- und Verhaltensbeschreibung der Objektmenge. Besonders behandelt werden asynchrone Kommunikationen, bei denen Objektmengen beteiligt sind und bei denen der Nachrichtenversender Rückgabewerte erwartet. ADORA-L stellt hierfür die *Multicast*-Nachrichten zur Verfügung. Mit

Multicast-Nachrichten ist es möglich, *alle* Rückgabewerte einer asynchron versendeten Nachricht, welche an eine Objektmenge versendet wurde, zu erfassen. Abbildung 27 zeigt beispielhaft eine Multicasting-Kommunikation. In der Verhaltensbeschreibung von Objekt A wird einen Nachrichtenversendung zur Objektmenge B spezifiziert (also Nachrichten an alle Instanzen in B). Durch das Schlüsselwort receiveAll wird bei einer folgenden Übergangsbedingung auf alle Antworten der Instanzen von B gewartet, erst dann erfogt ein Zustandsübergang.

Zu (v) und (vi): Eine Nachricht hat in ADORA-L den Charakter eines Ereignisses, d.h. es ist ein Geschehen, welches in einem gegebenen Kontext eine Bedeutung hat und sich räumlich und zeitlich lokalisieren läßt [Oestereich98]. In erweiterter Form ist es aber auch möglich, *Nachrichten als Informationsträger* zu verwenden. S*ynchrone Nachrichten* können einen expliziten und direkten *Rückgabewert* besitzen, der unmittelbar nach Versenden der Nachricht dem Versender als berechneter Wert vorliegt und zeitlos vom Nachrichtenempfänger berechnet wurde (genau genommen handelt es sich hierbei um eine zusätzliche implizite synchrone Nachricht, welche nach Abschluß von Berechnungen an den Versender zurückversandt wird).

Durch die ausdrucksstarke und präzise Modellierung von Beziehungen in der strukturellen Einblendung wird auch die dynamische Modellierung von Nachrichten stark vereinfacht. Im Gegensatz zu den klassischen objektorientierten Ansätzen ([Booch94], [Rumbaugh98] etc.) werden auf struktureller Ebene deutlich mehr verbindliche Aussagen bzgl. Kommunikation getroffen.

5.4 Integration einer zustandsbasierten Verhaltensbeschreibung

In objektorientierten Modellierungsansätzen ist es üblich, die Verhaltensbeschreibung als eigenständige Aspektmodelle zu erfassen, indem pro Klasse ein Zustandsautomat (bzw. ein Statechart) entwickelt wird (siehe Kapitel 4.5). Diese Form der Verhaltensbeschreibung hat jedoch einen entscheidenden Nachteil:

> *Das Verhalten kann nur lokal für jeweils eine Klasse modelliert werden. Eine zusammengehörige Verhaltensbeschreibung einer Gruppe von Objekten ist nicht bzw. nur sehr unanschaulich möglich. Im besonderen ist die gemeinsame Modellierung eines Teilsystems mit seinen Komponenten nicht vorgesehen.*

Ursache hierfür ist die bereits beschriebene Problematik der Modellierung auf Ebene der Klassen[1]. Zwar ist es prinzipiell möglich, Verhalten global durch ein entsprechend umfangreiche Verhaltensbeschreibung (beispielsweise durch ein Statechart) zu modellieren. Dies steht

[1] Der Begriff *Klasse* wird hier in der dualistischen Bedeutung der objektorientierten Ansätze aus Kapitel 4.5 verwendet.

jedoch im Widerspruch zum Objektparadigma, da das Objektverhalten und im speziellen die Objektzustände nicht mehr durch Objekte gekapselt sind.

Durch die Modellierung auf Ebene der abstrakten Objekte wird in ADORA-L das Fundament für eine globale Verhaltensbeschreibung gelegt. Möglich wird dies durch die Unterstützung einer sinnvollen und präzisen Modellierung des Objektkontextes. Zudem läßt sich eine solche Verhaltensbeschreibung einfach in ein entsprechendes Gesamtmodell integrieren.

5.4.1 Konzept der hierarchischen Zustandsautomaten

Das Konzept der Zustandsautomaten ist prinzipiell gut geeignet, um das diskrete Verhalten in objektorientierten Modellen zu beschreiben. Das Prinzip der Modellierung von Zuständen und Zustandsübergängen kann einfach auf Objekte und Objektzustände übertragen werden. Eine Alternative zur zustandsbasierten Modellierung ist die Verhaltensbeschreibung mit Petrinetzen. Diese erlauben eine präzisere und anschaulichere Modellierung von Nebenläufigkeiten. Petrinetze sind jedoch schlechter hierarchisch zerlegbar und sind durch eine komplexere Sprache schlechter verständlich als Zustandsautomaten (siehe Kapitel 6.4). Für ADORA-L werden daher Zustandsautomaten zur Verhaltensmodellierung verwendet.

Grundlage für die zustandsbasierte Verhaltensbeschreibung sind die hierarchisch zerlegbaren Statecharts von [Harel87] (siehe Kapitel 5.2). Diese Zustandshierarchie folgt dem Paradigma der Zustandsautomaten: Zu einem Zeitpunkt ist (auf einer Abstraktionsebene) genau ein Zustand aktiv. Tiefer eingeschachtelte Zustände spezialisieren die Verhaltensbeschreibung eines abstrakten Zustandes. Ist der komplexe Zustand aktiv, so gilt dies auch für genau einen speziellen Zustand.

Die Objekte der Basisstruktur werden als komplexe Zustände interpretiert, die ihrerseits wieder ein eingekapseltes Verhalten besitzen. Die Basisstruktur selber liefert also die grundlegende Struktur der Zustandshierarchie [Glinz93]. Für die Verhaltensbeschreibung werden die Objekte durch weitere Zustände ergänzt. Es werden insgesamt drei Arten von Zuständen unterschieden:

- *Komponentenzustände* – Die Objekte der Teil/Ganzes-Objekthierarchie bilden in der Interpretation verschachtelter Zustände den grundlegenden Aufbau des verhaltensorientierten Objektmodells. Objektmengen haben dabei die Bedeutung eine Menge nebenläufiger Zustände (siehe unten).
- *Elementare Zustände* – Zustände, welche einen Objektzustand konkret identifizieren. Es werden also zusätzliche Zustände erfaßt, die nicht bereits durch *Komponentenzustände* erfaßt sind.

- *Komplexe Zustände* – Zustände, welche ähnlich den elementaren Zuständen einen konkreten Objektzustand beschreiben. Die komplexen Zustände schachteln jedoch weitere Zustände ein (siehe auch die Zustandsverschachtelung der Statecharts).

In der Verhaltensbeschreibung werden Zustandsübergänge zwischen diesen Zuständen modelliert. Ein Zustandsübergang beschreibt einen zeitlosen Übergang von einem Zustand zu einem anderen. Sie werden näher spezifiziert durch eine *Übergangsbedingung* und eine *Übergangsaktion* (siehe Kapitel 4.3.1).

- Die *Übergangsbedingung* einer Zustandsbedingung gibt an, unter welchen Umständen ein Zustandsübergang erfolgt. Unter der Voraussetzung, daß der Ursprungszustand aktiv ist und die Übergangsbedingung zutrifft, erfolgt ein Zustandsübergang zum Zielzustand. Eine Zustandsbedingung kann Aussagen machen über zu empfangene Nachrichten, über Objektattribute oder über Zeitattribute.
- Die *Übergangsaktionen* eines Zustandsüberganges geben an, welche Tätigkeiten im Falle eines Zustandsüberganges ausgeführt werden. Eine Übergangsaktion ist entweder das Versenden einer Nachricht, die Manipulation eines Objektattributes oder die Erzeugung bzw. die Löschung von Objektinstanzen.

Damit die Zeitlosigkeit von Zustandsübergängen gewahrt bleibt, sind bestimmte Einschränkungen notwendig. Beispielsweise kann eine Übergangsaktion, welche eine asynchrone Nachricht versendet, nicht im Rahmen dieses Überganges auf eine Antwort warten (da beliebig viel Zeit zwischen versenden und empfangen verstreicht).

Sind zwei Zustände des Objektmodells durch einen Zustandsübergang miteinander direkt oder indirekt verbunden, so sind sie miteinander zeitlich gekoppelt. Dies hat zur Folge, daß nur einer der beiden Zustände zu einem Zeitpunkt gültig sein kann. Handelt es sich im speziellen um zwei *Objekte als Zustände*, so ist also mindestens ein Objekt «ungültig». Das bedeutet, die konkreten Wertebelegungen des Objektes sind über die Dauer der Ungültigkeit hinweg undefiniert. Sobald das Objekt wieder gültig ist, nimmt es seine Initialwerte an.

Alle Zustände, die nicht direkt oder indirekt über Zustandsübergänge verbunden sind, gelten als nebenläufig. Nebenläufige Zustände sind jedoch nicht automatisch synchronisationsfrei, da sie sich gegenseitig durch Nachrichten synchronisieren können.

5.4.2 Integritätsbedingungen der Verhaltensbeschreibung

Es existiert eine Reihe von Restriktionen bezüglich der Zustände des Objektmodells und bezüglich der Vernetzung dieser Zustände durch Zustandsübergänge. Im folgenden Teil werden diese in Form von Integritätsbedingungen vorgestellt. Für die Formulierung der Integritätsbedingungen sind eine Reihe von Definitionen notwendig:

Definition *Verbindung zweier Zustände* – Zwei Zustände A und B sind verbunden, wenn ein Zustandsübergang von A nach B und/oder von B nach A existiert.

Definition *Verbindung einer Mengen von Zuständen* – Eine Menge von Zuständen ist verbunden, wenn für beliebige Zustände A und B dieser Menge gilt: A ist mit B verbunden oder es existiert ein Zustand C der Menge, welcher mit A und mit B verbunden ist.

Definition *Nebenläufigkeit zweier Zustände* – Zwei Zustände sind nebenläufig, wenn diese durch keinen Zustandsübergang verbunden sind und kein dritter Zustand existiert, der mit beiden Zuständen durch einen Zustandsübergang verbunden ist.

Definition *Nebenläufigkeit von Zustandsmengen* – Zwei Mengen von Zuständen sind nebenläufig, wenn kein Zustand der einen Menge mit einem Zustand der anderen Menge verbunden ist.

Die erste Integritätsbedingung beschäftigt sich mit den Startzuständen der Zustandsbeschreibung. Der Startzustand eines Objektes ist der Zustand, der zum Zeitpunkt der Objektinitialisierung gültig ist. Die Integritätsbedingung beschreibt, in welchen Fällen ein Startzustand vorhanden sein muß.

Integritätsbedingung *Vollständigkeit und Eindeutigkeit der Anfangszustände* – Ein Objektmodell ist vollständig und eindeutig bezüglich den Startzuständen, wenn für jedes Objekt des Objektmodells gilt: Sei x_1, x_2, … x_n die Mengen von Zuständen des Objektes, welche nebenläufig zueinander sind. Jede Menge x_i besitzt genau einen Startzustand.

Die nachfolgende Integritätsbedingung klärt, welche Zustandsübergänge in einem Objektmodell aufgenommen werden dürfen.

Integritätsbedingung *Konsistenz der Zustandsübergänge* – Die Zustandsübergänge eines Objektmodells sind konsistent, wenn alle Zustandsübergänge zwei Zustände eines Objektes oder einem Zustand einer Objektkomponente und einen Zustand des Objektes verbinden.

Die letzte Integritätsbedingung schließt eine Anomalie aus, in der Zustände im Objektmodell zwar aufgeführt, aber niemals erreicht werden können.

Integritätsbedingung *Verbindung von Zuständen* – Jeder Zustand eines Objektmodells ist *verbunden*, wenn er entweder Startzustand ist oder es eine Sequenz von Zustandsübergängen gibt, die von einem Startzustand zu diesem Zustand führt.

5.5 Beschreibung der Objektfunktionalität

Im folgenden Kapitel werden die Konzepte und Prinzipien erläutert, um in ADORA-L die Funktionalität von Objekten zu beschreiben. Zunächst wird durch die Operation das prinzipielle Konzept der Modellierung von Funktionalitäten vorgestellt. Anschließend werden die grundlegenden Charakteristika einer funktionalen Einblendung in ADORA-L gezeigt.

5.5.1 Operationen in ADORA-L

Die Funktionalität von Objekten wird mit Hilfe einer Menge von *Operationen* beschrieben. Die Operationen eines Objektes werden ausgelöst durch eine Nachricht gleichen Namens, welche der Operation potentielle Parameter weiter- und potentielle Rückgabewert an das zu versendende Objekt zurückleitet (siehe auch Kapitel 5.3, S. 82).

> **Definition** *Operation (Synonym: Objektoperation)* – *Operationen* sind Dienstleistungen eines Objektes (einer Objektmenge), welche anderen Objekten (Objektmengen) zur Verfügung gestellt werden oder intern von anderen *Operationen* verwendet werden. *Operationen* verändern den Zustandsraum einer Objektinstanz, können weitere Nachrichten an Objekte (Objektmengen) versenden oder von anderen Objekten empfangen. Eine *Operation* ist grundsätzlich zeitbehaftet (Ausnahme: *Synchrone Operation*). Angestoßen und aufgerufen wird sie durch eine entsprechende Nachricht, welche auch die für die Ausführung nötigen Parameter überbringt.

Operationen können in ADORA-L sowohl durch synchrone als auch durch asynchrone Nachrichten ausgelöst werden (siehe Kapitel 5.3.2). *Operationen* sind in der Regel in ihrem Ablauf *zeitbehaftet*, daher muß eine Rückmeldung an die auslösende Instanz explizit durch eine eigenständige asynchrone Nachricht modelliert werden. Zeitlose Operationen, die also logisch keinen Zeitraum beanspruchen, werden als *synchrone Operationen* bezeichnet:

> **Definition** *Synchrone Operation* (*Synonym: synchrone Objektoperation)* – Eine *synchrone Operation* ist eine Operation, welche durch eine synchrone Nachricht ausgelöst wird. Die durch eine *synchrone Operation* spezifizierten Berechnungen sind aus Sicht des Nachrichtensenders (der Versender der auslösenden synchronen Nachricht) zeitlos. *Synchrone Operationen* können daher unmittelbar Rückgabewerte an die auslösende Objektinstanz zur Verfügung stellen.

Synchrone Operationen sind in ihrer Verwendung eingeschränkt. So ist es beispielsweise nicht möglich, daß eine ausgelöste *synchrone Operation* asynchron Nachrichten absetzt und auf deren Beantwortung wartet. In jedem Fall muß hier garantiert sein, daß die Berechnung zeitlos erfolgt. Dadurch können *synchrone Operationen* vollständig im Sinne einer mathematischen Funktion mit Seiteneffekten eingesetzt werden, d.h. sie können unproblematisch für die Spezi-

fikation der ebenfalls zeitlosen Zustandsübergänge sowie beliebig innerhalb anderer *Operationen* eingesetzt werden. Zudem reduzieren sie bei Ausführung der Verhaltensbeschreibung die Gefahr von *Verklemmungen*.

Die Operationen eines Objekts (einer Objektmenge) stellen Leistungen für andere Objekte zur Verfügung (und können mit entsprechend gleichnamigen Nachrichten angeregt werden) oder sie sind lokal, d.h. nur innerhalb des Objektes sicht- und zugreifbar. Lokale Operationen dienen beispielsweise zur Spezifikation von Übergangsaktionen oder werden von anderen Operationen des Objekts (der Objektmenge) verwendet.

5.5.2 Die funktionale Modellierung in ADORA-L

In ADORA-L wird die Funktionalität von Objekten in der funktionalen und in der verhaltensorientierten Einblendung modelliert. In der funktionalen Einblendung werden vorwiegend zustandsunabhängige Funktionalitäten wie Objektoperationen spezifiziert. In der verhaltensorientierten Einblendung wird Funktionalität abhängig vom Verhalten aufgeführt, d.h. Operationen, die in der zustandsbasierten Verhaltensbeschreibung Verwendung finden. Im Detail werden in ADORA-L folgende funktionale Eigenschaften dargestellt:

In der funktionalen Einblendung werden dargestellt:

- Typinformationen – Welcher Klasse gehört das Objekt an, welche Stereotypen sind für das Objekt gültig?
- Zustandsinformationen – Welche Attribute besitzt das Objekt, welchen Datentyp haben diese Attribute? Wie werden diese Attribute initialisiert? Sind diese Attribute nur innerhalb des Objekts (der Objektmenge) sicht- und zugreifbar, oder kann auch von außerhalb darauf zugegriffen werden?
- Spezifikationen der Objektleistungen – Welche Funktionalitäten sind dem Objekt zugeordnet und wie sind diese spezifiziert?
- Metainformationen des Objektes – Mit welchen Operationen kann ein Objekt erzeugt oder gelöscht werden, welche Restriktionen existieren bzgl. der Anzahl von Objekten (im Falle von Objektmengen)?

In der verhaltensorientierten Einblendung werden dargestellt:

- Spezifikation von Übergangsbedingungen (Empfang von Nachrichten, logische Aussagen über Objektattribute oder über Nachrichtenparameter)
- Spezifikation von Aktionen (Versenden von Nachrichten, Aufrufen von objekteigenen Operationen, Spezifikation von elementaren Operationen)

Die funktionale Spezifikation erfolgt axiomatisch, d.h. durch Angabe der Voraussetzung und der Zusicherung der Operation. Das Beschreibungsmittel hierzu ist eine textuelle Sprache, und erfolgt wahlweise informal, teilformal oder formal. Damit wird ein wesentlicher Teil der Forderung nach variablem Formalitätsgrad (siehe auch Kapitel 3.4, S. 31) umgesetzt. Die informale Beschreibung ist in natürlicher Sprache gehalten, die formale Sprache ADORA-FSL lehnt sich an die Sprache ASTRAL [Coen93] an. Die teilformale Sprache schließlich enthält Elemente der formalen Sprache, kombiniert mit Elementen der natürlichen Sprache. Die Sprache ASTRAL wurde deshalb als Vorbild der formalen Sprache gewählt, da sie sehr einfach gehalten und einfach verständlich ist und gleichzeitig vollständig ausreichend ist für die Spezifikation der wenig komplexen funktionalen Problembeschreibungen.

5.6 Die Modellierung von Typeigenschaften

Die Extension eines Systemmodells wird in ADORA-L mit Hilfe eines Objektmodells und der Modellierung auf Ebene der abstrakten Objekte beschrieben. Zusätzlich ist es jedoch auch notwendig, intensionale Aussagen über Objekte in Form von Objekttypen zu treffen. Die intensionale Beschreibung macht Aussagen über Gemeinsamkeiten oder Ähnlichkeiten zwischen abstrakten Objekten. Nachfolgend wird erläutert, wie in ADORA-L Typeigenschaften modelliert werden und wie diese Beschreibungen in das integrierte Gesamtmodell eingebettet werden. Das zentrale Beschreibungsmittel zur intensionalen Beschreibung ist eine *Klassenhierarchie*, welche Objekttypen (also Klassen) in Ober- und Unterklassen einteilt.

5.6.1 Das Typverzeichnis

Die intensionalen Eigenschaften der Objekte könnten prinzipiell in Form eines intensionalen Aspektmodells, also in einem separaten Klassenmodell beschrieben werden. Diese Art der Modellierung ist beispielsweise bei Ansätzen üblich, welche die Aspektmodellierung unterstützen. Objekt- und Klassenmodell wären jedoch hochgradig redundant: Viele im Objektmodell getroffenen Aussagen sind gleichzeitig auch charakteristisch für den Objekttyp. Diese Form der Redundanz soll in ADORA-L durch den Ansatz eines integrierten Gesamtmodells entschärft werden.

Damit die Eigenschaft der integrierten Modellierung gewahrt bleibt, werden die intensionalen Eigenschaften in ADORA-L mit Hilfe eines *Typverzeichnisses* spezifiziert. Das *Typverzeichnis* ist kein eigenständiges Modell, sondern vielmehr als eine orthogonale Erweiterung des Objektmodells zu verstehen. Im Typverzeichnis werden intensionale Zusammenhänge zwischen Klassen modelliert, die Eigenschaften der Klasse selber werden jedoch nicht im Typverzeichnis, sondern indirekt in den zugehörigen Objekten des Objektmodell mitspezifiziert. Die Klassenspezifikation ergibt sich hier durch die Betrachtung eines zugehörigen Objektes, in

welchem alle kontextabhängigen Informationen ausgeblendet werden. Im *Typverzeichnis* werden die im Objektmodell spezifizierten Objekteigenschaften durch weitere Eigenschaften ergänzt. Im Typverzeichnis selber werden folgende Sachverhalte modelliert:

- *Die Klassenhierarchie* – Die Hierarchie von Klassen im Sinne einer Generalisierungsabstraktion. Die Klassenhierarchie wird detailliert im Kapitel 5.6.2 behandelt.
- Die *Stereotypen* des Objektmodells – Stereotypen werden verwendet, um gemeinsame Eigenschaften von Modellelementen zum Ausdruck zu bringen. Die Stereotypen werden im Typverzeichnis deklariert. Ein so deklarierter Stereotyp kann im Objektmodell einem Modellelement zugewiesen werden.
- Die *Datentypen* des Objektmodells – Datentypen werden verwendet, um den Typ von Attributen und Nachrichtenparametern im Objektmodell zu deklarieren. Die zentrale Deklaration von Datentypen ist dann notwendig, wenn derselbe Datentyp von mehreren Objekten in Anspruch genommen werden muß.

Durch das Konzept des Typverzeichnisses wird das Problem der redundanten extensionalen und intensionalen Beschreibung vermieden. Durch die Aufführung von Klassen als Verweise auf die zugeordneten Objekte des Objektmodells bleibt zudem das Konzept des integrierten Gesamtmodells gewahrt. Notwendig ist jedoch eine Abklärung, welche Teile der Objektspezifikation kontextfrei sind und somit für alle weiteren Objekte desselben Typs bindend sind.

5.6.2 Die Klassenhierarchie

Häufig weisen die im Objektmodell aufgeführten Objekte und Objektmengen typspezifische Ähnlichkeiten auf, d.h. weist die Klasse eines Objekts (einer Objektmenge) x Ähnlichkeiten zur Klasse des Objekts (der Objektmenge) y auf. Die Modellierung solcher Zusammenhänge auf Klassenebene kann dazu verwendet werden, die Gesamtheit aller verwendeten Klassen zu strukturieren und damit die typspezifischen Objekteigenschaften verständlicher darzustellen. Aus diesem Grund wird in ADORA-L im Typverzeichnis eine *Klassenhierarchie* aufgeführt. Die Klassenhierarchie strukturiert eine Menge von Klassen nach dem Prinzip der Generalisierungsabstraktion.

> **Definition** *Klassenhierarchie* – Die Klassenhierarchie teilt eine Menge von Klassen in eine Hierarchie von über- und untergeordneten Klassen ein. Eine übergeordnete Klasse wird hierbei als Oberklasse, eine untergeordnete Klasse als Unterklasse bezeichnet. Für alle Klassen der Klassenhierarchie muß gewährleistet sein, daß alle gültigen Aussagen der Oberklasse auch für die Unterklasse gültig sind.

Die Klassenhierarchie ist ein Abstraktionsmechanismus, da die Oberklasse die Eigenschaften einer oder mehrerer Unterklassen zusammengefaßt beschreibt (siehe Kapitel 3.3.1). Sie kann durch zwei Arten in einem objektorientierten Ansatz umgesetzt werden. In der Literatur wer-

den diese zwei Arten als *Spezialisierung* und als *Subtyping* bezeichnet [Eckert94]. Die *Spezialisierung* setzt die Generalisierungsabstraktion in einer abgeschwächten Form, das *Subtyping* setzt sie in einer verschärfte Form um. In ADORA-L wird das Konzept der Spezialisierung eingesetzt. Im folgenden werden beide Arten der Umsetzung der *Generalisierungsabstraktion* miteinander verglichen.

5.6.3 Subtyping versus Spezialisierung

Subtypen und Spezialisierungen werden wie folgt definiert ([Blair91], [Eckert94]):

> **Definition** *Subtyp* – Eine Klasse S ist Subtyp einer anderen Klasse K, wenn S mindestens die Funktionalität und das Verhalten von K zur Verfügung stellt [Blair91].
>
> **Definition** *Spezialfall* (Spezialisierung) – Eine Klasse S ist Spezialfall einer Oberklasse K (im Sinne einer Generalisierungsabstraktion), wenn jede Funktionalität und jedes Verhalten von K *in spezieller Form* von S zur Verfügung gestellt wird [Eckert94].

Ein Objekt einer Subtyp-Unterklasse ist vollständig «aufwärtskompatibel», d.h. in jedem Kontext der Oberklasse verhält es sich gleich wie Objekte dieser Oberklasse. Jede Funktionalität einer Oberklasse muß sich also in ihren Subtypen wiederfinden. Die Subtyping-Restriktionen sind sehr scharf und in der Regel nur schwer oder umständlich umsetzbar [Eckert94, S. 4 f.]. Zudem ist der Nachweis einer korrekten Typ/Subtyp-Beziehung nur sehr schwer und umständlich zu führen, da alle möglichen Sequenzen von Operationen einer Unterklasse im Kontext der Oberklasse geprüft werden müssen.

Die Spezialisierung stellt eine abgeschwächte Form des Subtypings dar. Statt einer garantierten Aufwärtskompatibilität zur Oberklasse wird hier lediglich eine abgeschwächte «Abwärtskompatibilität» gefordert. Alle Operationen bzw. alle möglichen Operationssequenzen[1] der Oberklasse müssen in spezieller Form durch die spezialisierte Klasse ermöglicht werden. Genauer formuliert lassen sich die Klassenoperationen Op_K, Op_S zwischen einer Klasse K und einer Klasse S als Spezialisierung so einander zuordnen, daß gilt:

(i) Für jede Operation von K gibt es eine Spezialoperation in S mit
$((pre(Op_K) \rightarrow pre(Op_S)) \wedge ((post(Op_S) \rightarrow post(Op_K))$

(ii) Für jede mögliche Operationssequenz $OpSeq_K$ von K gibt es eine *spezialisierte* Operationssequenz in S, für die gilt:

---

[1] Eine Operationssequenz einer Klasse ist eine legale und definierte Sequenz von Klassenoperationen, bei denen jeweils die Nachbedingung jeder Operation die Vorbedingung der Nachfolgeoperation impliziert.

$((pre(OpSeq_K) \rightarrow pre(OpSeq_S)) \wedge ((post(OpSeq_S) \rightarrow post(OpSeq_K))$

$OpSeq_S$ ist eine spezielle Operationssequenz zu $OpSeq_K$. Alle Operationen in $OpSeq_K$ finden sich in derselben Reihenfolge auch in $OpSeq_S$ wieder, jedoch können in der speziellen Operationssequenz zusätzliche Operationen vorhanden sein, die kein Pendant in K besitzen.

Im Gegensatz zum *Subtyping* sind die Objekte einer spezialisierten Klasse nicht automatisch kompatibel mit dem Kontext der Objekte der Oberklasse. Die oben beschriebene spezialisierte Operationssequenz reicht noch nicht aus, um ein spezielleres Objekt in einem generelleren Kontext korrekt einzusetzen (siehe auch Beispiel unten). Im praktischen Einsatz ist diese schwächere Bedingung jedoch ausreichend, in der Regel müssen nur überschaubar wenige Operationssequenzen im Kontext der Oberklasse gültig sein, nicht aber alle möglichen Sequenzen der Spezialisierung. Diese relevanten Sequenzen sind im Einzelfall zu prüfen. Die Spezialisierung von Klassen ist zudem deutlich besser handhabbar und praktisch einsetzbar im Gegensatz zum Subtyping-Konzept. In ADORA-L wird daher das Konzept der Spezialisierung verwendet.

Beispiel: Gegeben seien die beiden Klassen *Rechteck* und *Quadrat*

Rechteck	**Quadrat**
Erzeuge : Nat → Rechteck	Erzeuge : Nat → Quadrat
Verschiebe : Rechteck × Nat → Rechteck	Verschiebe : Quadrat × Nat → Quadrat
Skaliere_H : Rechteck × Nat → Rechteck	Skaliere : Quadrat × Nat → Quadrat
Skaliere_B : Rechteck × Nat → Rechteck	
Drehe : Rechteck × Nat → Rechteck	Drehe : Quadrat × Nat → Quadrat
Zeichne : Rechteck → ε	Zeichne : Quadrat → ε

Auszug Spezifikation für Skalieroperationen

pre(Skaliere_H) = Nat ≥ 0	pre(Skaliere) = Nat ≥ 0
post(Skaliere_H) = (Höhe'==Nat*Höhe)	post(Skaliere_B) = (Breite'==Nat*Breite) ∧
pre(Skaliere_B) = Nat ≥ 0	(Höhe'==Nat*Höhe)
post(Skaliere_B) = (Breite'==Nat*Breite)	

Intuitiv und auch im Sinne einer Spezialisierung ist die Klasse Quadrat eine Spezialisierung der Klasse Rechteck. Es läßt sich zeigen, daß jede Funktionalität von Rechteck sich in spezialisierter Form in Quadrat wiederfindet. So werden beispielsweise die Operationen Skaliere_H und Skaliere_B durch die Operation Skaliere spezialisiert (im Sinne der Implikation von Voraussetzungen und Zusicherungen). Ebenso läßt sich zeigen, daß jede Operationssequenz von Rechteck in spezieller Form auch für das Quadrat gilt.

Das obige Verhältnis zwischen Rechteck und Quadrat ist jedoch **kein** Subtypenverhältnis, da sich das Quadrat im Kontext des Rechtecks nicht wie ein Rechteck verhält. So lassen sich die im Rechteck spezifizierten Operationen Skaliere_B und Skaliere_H nicht

auf Quadrat anwenden. Hierfür wäre eine Ergänzung der Operationsspezifikation folgendermaßen notwendig (was wenig intuitiv ist und praktisch zu aufwendig wäre):

Skaliere_H : Quadrat $\times$ Nat $\rightarrow$ Rechteck
Skaliere_B : Quadrat $\times$ Nat $\rightarrow$ Rechteck

5.6.4 Spezialisierung/Generalisierung von Instanzen

In Adora-L ist es möglich, dynamisch den Typ (also die Klasse) einer Instanz während der Laufzeit zu verändern. Die Veränderung ist jedoch ausschließlich im Sinne einer Spezialisierung oder einer Generalisierung möglich, indem eine Instanz einem spezielleren oder einem allgemeineren Klasse (als Objekttyp) zugeordnet wird. Andere Mutationen sind nicht möglich. Grundlage für die Spezialisierung und Generalisierung von Instanzen ist die Klassenhierarchie, in welcher entsprechende Ober- und Unterklassen definiert sind.

Dieser dynamische Spezialisierungs/Generalisierung ist ein mächtiger Mechanismus. Er erlaubt es, die Eigenschaften der Instanzen des Systemmodells flexibel an neue Begebenheiten anzupassen und Sachverhalte anschaulicher und einfacher zu modellieren. Explizi ist es so möglich, eine Objektmenge variabler Größe mit Instanzen mit unterschiedlich speziellen Typen (entsprechend der Klassenhierarchie) zu füllen. Eine Objektmenge GeometrischeFiguren beispielsweise wäre in der Lage, Sowohl Kreise, Rechtecke, Quadrate usw. als Instanzen zu beschreiben, welche alle dieselben Basiseigenscchaften besitzen (und alle entsprechende Operationen ausführen könnten).

Die dynamische Typveränderung wird durch die Bereitstellung spezieller Metaoperationen ermöglicht, welche vom Modellierer für alle relevanten Instanztypveränderungen definiert werden muß. Diese Metaoperationen transformieren die Typeigenschaften einer Instanz in die entsprechenden Eigenschaften des neuen Typs. Prinzipiell exisiteren zwei Klassen solcher Metaoperationen:

- *Generalisierungsoperationen* – Die Generalisierungsoperation wandelt eine Instanz einer Klasse in eine Instanz einer Oberklasse zu.
- *Spezialisierungsoperationen* – Die Spezialisierungsoperation wandelt eine Instanz einer Klasse in eine Instanz einer Unterklasse um.

Bei Typmanipulationen gibt es eine Randbedingung:

Eine Instanz darf nie einem allgemeineren Typ angehören als das Objekt oder die Objektmenge, welcher sie angehört.

Diese Randbedingung ist notwendig, da andernfalls die grundlegenden Eigenschaften des Objektes (der Objektmenge) im allgemeinen nicht mehr garantiert werden können (durch eine Generalisierung gehen einer Instanz Eigenschaften verloren!).

- Eine Generalisierungs/Spezialisierungsoperationen wird durch folgende Attribute definiert:
- Welchem Typ (Klasse) gehört die Instanz momentan an?
- Welchen Typ (Klasse) soll die Instanz neu annehmen (muß eine entsprechende Oberklasse bzw. Unterklasse sein)?
- Wie werden die Wertebelegungen der Instanzattribute umgewandelt?
- In welchem Zustand (im Sinne der Verhaltensbeschreibung) befindet sich die transformierte Instanz)?

5.6.5 Heuristiken für die Entwicklung einer Klassenhierarchie

Im bisherigen Teil wurde die Spezialisierung als grundsätzliches Prinzip beim Aufbau der Klassenhierarchie vorgestellt. Ungeklärt ist jedoch bisher die Frage, wie das Prinzip der Spezialisierung auf Modellierungsebene in einer Klassenhierarchie umgesetzt wird. Im Detail ist zu klären, welche Bedingungen die Oberklasse und die Unterklasse eines Spezialisierungszusammenhangs zu erfüllen haben. Die Festlegung solcher Regeln ist aus zweierlei Gründen problematisch:

(i) Die direkte Umsetzung der beiden Spezialfall-Charakteristika (siehe S. 69) ist praktisch zu aufwendig und in der Regel auch nicht sinnvoll, da nur eine sehr eingeschränkte Menge an Operationssequenzen einer Klasse überhaupt durchlaufen werden.

(ii) Die grundlegenden Charakteristika und Invarianten einer Klasse werden häufig nicht explizit modelliert, sondern ergeben sich indirekt durch die Gesamtheit aller Invarianten der Klassenoperationen und Klassenattribute. Die Übernahme von Klasseninvarianten ist somit nur schwer nachvollzieh- und prüfbar.

Bemerkung zu (ii): Im Beispiel *Quadrat ist Spezialfall von Rechteck* (siehe Kapitel 5.6.3) etwa die wichtigste Invariante von Quadraten gegenüber Rechtecken, nämlich zwei identisch lange Seiten zu haben, nicht explizit modelliert. Verletzungen dieser Invariante (beispielsweise, wenn ein Rechteck als Spezialfall eines Quadrates modelliert wird) sind schwer zu bestimmen.

Bisher wurde das Problem, hinreichend und notwendige Integritätsbedingungen für Klassenspezialisierungen zu finden, nicht oder nur unzureichend gelöst. In [Paech94] beispielsweise werden funktionale Regeln definiert, ohne jedoch Objektattribute zu berücksichtigen. In [Weber98] dürfen einer spezialisierten Klasse nur «gutartige» Klassenoperationen hinzugefügt werden, welche die von der Oberklasse übernommenen Attribute nicht manipulieren.

In ADORA-L werden daher keine explizit zu erfüllenden Regeln und Bedingungen formuliert. Ziel vielmehr ist die Formulierung von Heuristiken, welche Hilfestellungen für den Aufbau einer Klassenhierarchie geben und auf potentielle Fehlerquellen hindeuten. Diese *Spezialisierungsheuristiken* sind jedoch weder notwendiges noch hinreichendes Kriterium für die Entwicklung einer korrekten Klassenhierarchie (im Sinne des Prinzips der Spezialisierung). Aus der Nichteinhaltung einer Heuristik kann demzufolge genauso wenig auf eine fehlerhafte Klassenhierarchie geschlossen werden wie aus einer vollständigen Umsetzung der Heuristiken die Korrektheit gezeigt werden kann.

ADORA-L unterscheidet folgende *Spezialisierungsheuristiken*:

- *Übernahme der Klassenattribute*
- *Übernahme der Komponenten*
- *Übernahme der Operationen*
- *Übernahme der Objektzustände*
- *Hinzufügen nebenläufiger Zustände*
- *Übernahme von Zustandsübergängen*
- *Hinzufügen neuer Zustandsübergänge*

Im folgenden werden diese *Spezialisierungsheuristiken* im Detail vorgestellt.

Die beiden folgenden Heuristiken beschreiben die Übernahme der wesentlichen Struktur der Oberklasse in die Unterklasse. Grundlage hierfür ist die Idee, daß die grundlegende Struktur, gebildet durch die Klassenattribute und Klassenkomponenten im wesentlichen erhalten bleibt und nur durch neue, zusätzliche Elemente ergänzt wird.

Heuristik *Übernahme der Klassenattribute* – Alle Attribute der Oberklasse finden sich auch in der Unterklasse wieder. Sowohl die Attributbezeichnung als auch der Attributtyp stimmt in der Ober- und der Unterklasse überein.

Heuristik *Übernahme der Komponenten* – Alle Komponenten der Oberklasse finden sich auch in gleicher oder in spezieller Form in der Unterklasse wieder. Der Objekttyp der Komponenten der Oberklasse stimmt dabei mit den entsprechenden Komponenten der Unterklasse überein bzw. die Komponente der Unterklasse besitzt einen Spezialfall des Typs der Oberklasse. Die Kardinalität der Komponente sowie die Beziehungen zwischen den Komponenten innerhalb einer Klasse können in der Ober- und der Unterklasse unterschiedlich sein (die Bezeichnungen der Komponenten müssen jedoch übereinstimmen).

Besonders hervorzuheben ist dabei die Umsetzung von Klassenkomponenten durch speziellere Komponenten in der Unterklasse. Eine Klasse wird dadurch Spezialfall einer Oberklasse, da eine Komponente spezialisiert wird. Diese Heuristiken sind als konstruktive Bauanleitungen für die Entwicklung spezieller Klassen gedacht. Es ist durchaus vorstellbar, daß eine Klasse Unterklasse einer anderen Klasse ist, aber vollständig andere Attribute und Komponenten aufweist. Die Klassenhierarchie ist aber bei Einhaltung dieser Heuristiken wesentlich besser verständlich, da sich die Klassenbeschreibungen in gewisser Weise wiederholen.

Die folgende Heuristik setzt im wesentlichen die erste Eigenschaft eines Spezialfalls (siehe Kapitel 5.6.3) um. Die Restriktion über die gleiche Namensgebung der Operationen zwischen Ober- und Unterklasse wurde ebenfalls aus Gründen der Übereinstimmung der Struktur so gewählt.

Heuristik *Übernahme der Operationen* – Alle Operationen der Oberklasse finden sich auch in gleicher oder in spezieller Form in der Unterklasse wieder. Die Operationen in der Unterklasse besitzen dieselbe Bezeichnung. Die Voraussetzung der Oberklassenoperation impliziert die Voraussetzung der entsprechenden Operationen der Unterklassen. Die Zusicherung der Unterklassenoperation impliziert die Zusicherung der entsprechenden Oberklassenoperation.

Die folgenden vier Heuristiken beschreiben den Zusammenhang zwischen Ober- und Unterklasse in der Verhaltensbeschreibung. Die erste Heuristik beschreibt die Übernahme der Zustände der Oberklasse in die Unterklasse, also im speziellen die elementaren Zustände, die komplexen Zustände und die Objektkomponenten in der Rolle eines Zustandes (siehe Kapitel Verhaltensbeschreibung). In der zweiten Heuristik wird erläutert, welche Art von Zuständen zu diesen übernommenen Zuständen hinzugefügt werden dürfen.

Heuristik *Übernahme der Objektzustände* – Alle Zustände der Oberklasse finden sich auch in gleicher oder in detaillierterer Form in der Unterklasse wieder. Ein elementarer Zustand der Oberklasse kann entweder

- ebenfalls als elementarer Zustand gleichen Namens
- oder durch einen komplexen Zustand gleichen Namens

in die Unterklasse übernommen werden. Komplexe Zustände werden in derselben Struktur und derselben Namensgebung in die Unterklasse übernommen. Die Anfangszustände der Oberklasse werden dabei in die entsprechenden Zustände der Unterklasse übernommen.

Heuristik *Hinzufügen nebenläufiger Zustände* – Die Zustandsbeschreibung einer Klasse sowie der komplexen Zustände dieser Klasse können durch eine nebenläufige Zustandsmenge ergänzt werden (siehe Kapitel 5.5.2).

Die nächsten zwei Heuristiken beschreiben den Zusammenhang zwischen den Zustandsübergängen der Ober- und der Unterklasse.

Heuristik *Übernahme von Zustandsübergängen* – Sämtliche Zustandsübergänge der Oberklasse werden auf die entsprechenden Zustände der Unterklasse übernommen. Für die Zustandsbedingungen und Zustandsaktionen gilt dabei folgendes:

- Die Bedingung eines Zustandsüberganges der Unterklasse impliziert die Bedingung des entsprechenden Zustandsübergangs der Oberklasse
- Die Zusicherung der Aktionen eines Zustandsüberganges der Unterklasse impliziert die Zusicherung der Aktionen des Zustandsübergangs der Oberklasse

Heuristik *Hinzufügen neuer Zustandsübergänge* – In der Unterklasse dürfen neue Zustandsübergänge nur zwischen neu hinzugefügten Zuständen und zwischen neu hinzugefügten und übernommenen Zuständen hinzugefügt werden (hierbei ist die Heuristik *Hinzufügen nebenläufiger Zustände* zu beachten).

Durch die Verwendung von Spezialisierungsheuristiken in ADORA-L werden Hilfestellungen für den Bau und die Prüfung von Klassenhierarchien vorgestellt. In den meisten Fällen läßt sich selbst unter Anwendung entsprechender Heuristiken eine fehlerhafte Klassenhierarchie konstruieren. Im «Normalfall» sollten sie jedoch ausreichen, konsistente Hierarchien in ADORA-L zu modellieren.

Kapitel 6

Überblick über ADORA-L

Das vorige Kapitel führte die grundlegenden Konzepte der Sprache ADORA-L ein. Diese Sprachkonzepte sind Grundlage für den groben Aufbau der Sprache sowie für die Sprachkonstrukte selber. Im folgenden Kapitel wird die Sprache ADORA-L im Überblick vorgestellt. Schwerpunktmäßig wird aufgeführt, durch welche Modellkomponenten ein ADORA-L-Systemmodell gebildet wird und wie diese Komponenten miteinander zusammenhängen. Zur besseren Veranschaulichung wird ein durchgängiges Beispiel mit Hilfe von ADORA-L modelliert. Zunächst erfolgt eine kurze Vorstellung des *Ticketing-Systems* als durchgängiges Illustrationsbeispiel, eine ausführlichere Beschreibung findet sich in Anhang C.

> **Illustrationsbeispiel** Ticketing-System – Ein Ticketing-System soll den Verkauf von Tikkets für beliebige Veranstaltungen an mehreren, geographisch verteilten Verkaufsstellen ermöglichen. Eine Verkaufsstelle gestattet den Verkauf von Tickets einer wählbaren Veranstaltung. Die Verkaufsstelle soll Informationsanzeigen, Benutzereingaben, Sitzplatzreservationen, Buchungen, Bezahlungen und Ticketerstellungen (Druck) handhaben. Die Verkaufsstelle soll ein Selbstbedienungsautomat sein. Die angebotenen Veranstaltungen sollen angezeigt werden, sofern keine Verkaufstransaktion im Gange ist. Ein Benutzer kann einen noch nicht abgeschlossenen Ticketkauf jederzeit abbrechen. Die Verkaufsstellen sind durch ein Netz mit sämtlichen Veranstaltungsservern verbunden. Die von einem Veranstaltungsserver angebotenen Veranstaltungstickets werden exklusiv nur von diesem angeboten und allen Verkaufsstellen gleichermaßen zugänglich gemacht.
>
> Bei den folgenden Betrachtungen wird schwerpunktmäßig der Systemteil Verkaufsstelle betrachtet.

6.1 Aufbau eines ADORA-L-Systemmodells

Ein Systemmodell wird in ADORA-L grundsätzlich durch ein *Objektmodell* und durch ein *Typverzeichnis* beschrieben. Im *Objektmodell* wird der Problembereich auf Ebene der *abstrakten Objekte* modelliert. Die grundlegende Modellstruktur des Objektmodells wird durch die *Basis-*

struktur beschrieben, eine Objekthierarchie, die nach dem Prinzip der Teil/Ganzes-Strukturierung aufgebaut ist. Diese *Basisstruktur* wird durch weitergehende Aspektbeschreibungen in Form von aspektbezogenen Einblendungen ergänzt, indem diese Beschreibungen additiv in die bestehende Basisstruktur eingeblendet werden. ADORA-L unterscheidet hierbei die Aspektbeschreibungen *strukturelle* Einblendung, *verhaltensorientierte* Einblendung und *funktionale* Einblendung[1]. In Abbildung 28 wird der Aufbau eines Objektmodells veranschaulicht. Das Objektmodell als Ganzes umfaßt die *Basisstruktur* sowie die drei Aspektbeschreibungen. Aspektbeschreibungen können unabhängig voneinander zur Basisstruktur ein- oder ausgeblendet werden (und analog dazu können sie getrennt voneinander erstellt, modifiziert und geprüft werden). Das Objektmodell ist das zentrale Beschreibungsmittel in ADORA-L. Die grundlegende Systemmodellierung erfolgt durch Modellierung einer Teil/Ganzes-Objekthierarchie sowie durch die integrierte Modellierung zentraler Aspekte.

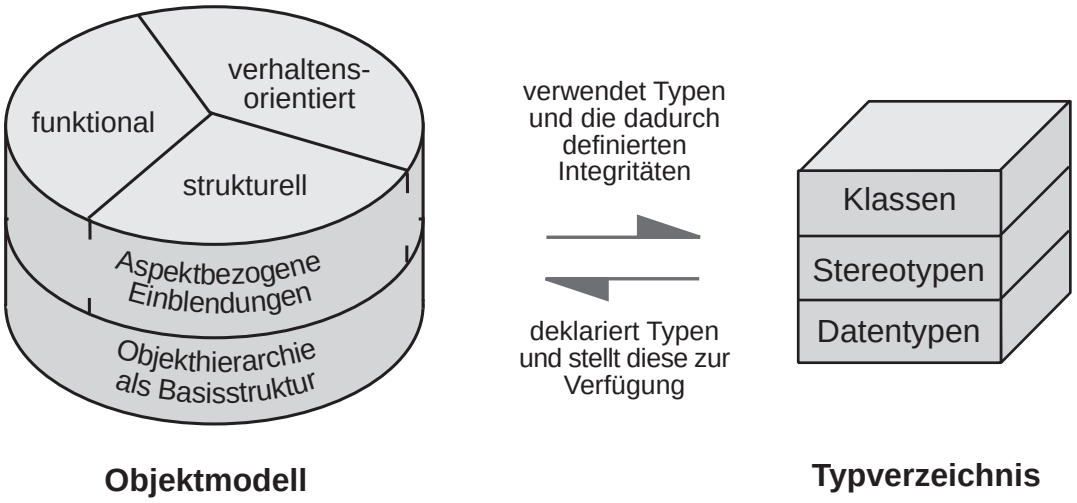


Abb. 28: Überblick über die Spezifikationssprache ADORA-L: Objektmodell und Typverzeichnis

Das *Typverzeichnis* steht orthogonal zur im Objektmodell aufgeführten Teil/Ganzes-Objekthierarchie. In ihm werden diverse Typen deklariert, die im Objektmodell zugewiesen und verwendet werden. Das *Typverzeichnis* ist also kein eigenständiges Modell, sondern ist vielmehr als ein Deklarationsbereich für das Objektmodell zu verstehen. Beschrieben werden die Typarten *Klasse*, *Stereotyp* und *Datentyp* (siehe Abbildung 28). *Klassen* werden durch *Objektschablonen* und durch eine *Klassenhierarchie* spezifiziert. Die *Objektschablonen* sind im wesentlichen Abbildungen der typspezifischen Eigenschaften der Objekte und der Objektmengen, welche einer Klasse zugeordnet sind[2]. Die *Klassenhierarchie* spezifiziert eine Klasse zusätzlich durch Angabe von spezielleren und generelleren Klassen. Die Klassenbeschreibung

[1] Es ist geplant, in ADORA-L in einer zukünftigen Fassung zusätzlich eine *interaktionsorientierte Einblendung* zu integrieren (siehe Kapitel 5.2).

[2] typspezifische Eigenschaften sind beispielsweise die Objektattribute, die Komponenten der Objekte oder die Objektzustände

ist also nicht als eigenständige Beschreibung, sondern vielmehr als eine Referenz auf entsprechende Objektbeschreibungen zu verstehen. In der Metapher erinnert der Zusammenhang zwischen Typverzeichnis und Objektmodell an den Typ-Variablen-Zusammenhang in prozeduralen Programmiersprachen.

Stereotypen spezifizieren Ergänzungen und Erweiterungen vorhandener Modellelemente durch ergänzende oder restringierende Eigenschaften.

Datentypen schließlich spezifizieren Datenstrukturen, welche den Objektattributen und den Nachrichtenparametern zugewiesen werden.

Das vollständige Objektmodell besteht aus einer *Basisstruktur* und den drei aspektbezogenen Einblendungen. Die Aus- und Einblendung von zusammengehörigen Aspekten soll die Erstellung, die Änderung und die Navigation innerhalb des Modells vereinfachen. Die Vereinfachung ergibt sich im wesentlichen durch eine Reduktion der sichtbaren Sprachkonstrukte. In den folgenden Kapiteln werden die einzelnen Komponenten des Objektmodells sowie das Typverzeichnis im Überblick erläutert.

6.2 Die Basisstruktur des Objektmodells

Die ADORA-L-*Basisstruktur* bildet das Skelett und damit die grundlegende Gliederungsstruktur des Systemmodells. Sie teilt die Objekte und die Objektmengen des Objektmodells nach dem Prinzip der Teil/Ganzes-Strukturierung in *Komponenten* und *Kompositionen* ein und bildet dadurch eine Objekthierarchie. Diese Teil/Ganzes-Objekthierarchie ist ein Abstraktionsmechanismus im Sinne der Kompositionsabstraktion (siehe Kapitel 5.2). In der *Basisstruktur* erfolgt zudem die Spezifikation des Systemkontextes: Innerhalb der Objekthierarchie wird dieser durch *externe Objekte* beschrieben, außerhalb der Hierarchie durch *externe Akteure* (siehe Kapitel 5.2.4). Abbildung 29 zeigt einen Überblick über die Basisstruktur eines ADORA-L-Systemmodells. Die Teil/Ganzes-Objekthierarchie wird durch ineinander geschachtelte Objekte bzw. Objektmengen beschrieben (dargestellt durch einfache Rechtecke bzw. durch Rechteckstapel). Die "…"-Notation hinter dem Objektnamen gibt an, ob das Objekt bzw. die Objektmenge selber eingeschachtelte Objektkomponenten besitzt. Der Systemkontext wird in der Basisstruktur wie folgt beschrieben: Die *externen Akteure* werden außerhalb der Basisstruktur aufgeführt (dargestellt durch eine Sechseck-Notation). Externe Objekte (dargestellt durch Rechtecke, ergänzt durch die zusätzliche Bezeichnung «*:extern*») werden integriert beschrieben: Für die Umgebung des *externen Objektes* besitzt es alle Eigenschaften eines normalen Objektes.

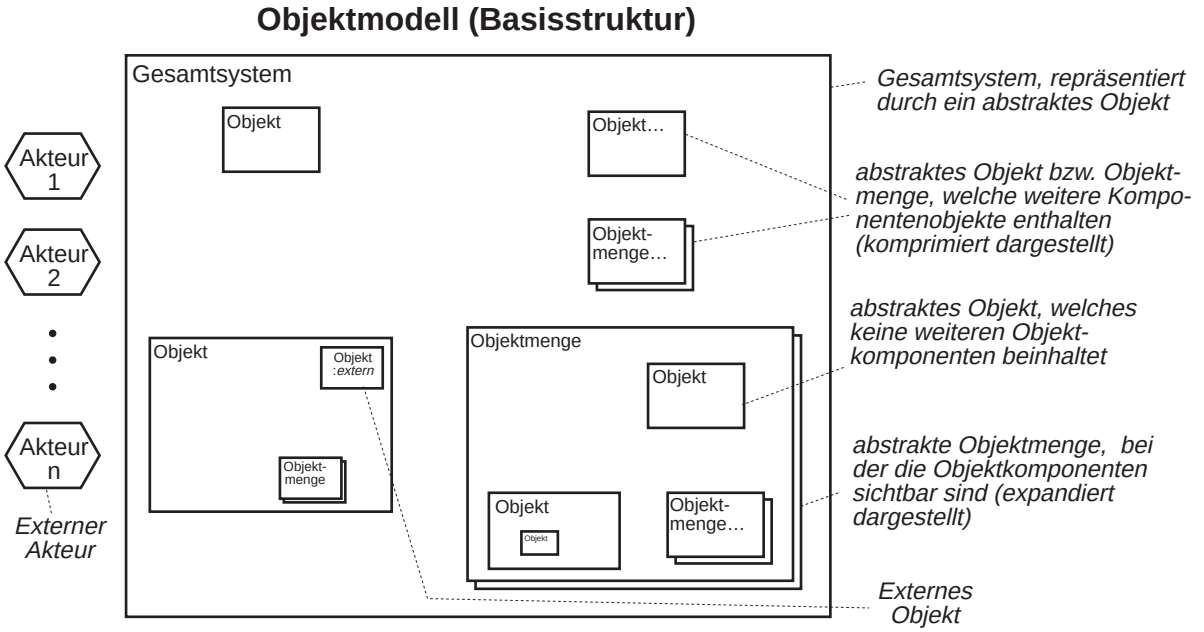


Abb. 29: Die Basisstruktur von ADORA-L im Überblick

Für die Visualisierung der Teil/Ganzes-Hierarchie wird der Mechanismus der *Komprimierung* und der *Expansion* verwendet. Dieser Darstellungsmechanismus erlaubt es, auf Grundlage der Kompositionsabstraktion Objekte (Objektmengen) nur teilweise sichtbar (in komprimierter Form) sowie komprimiert dargestellte Objekte (Objektmengen) an anderer Stelle detaillierter (in expandierter Form) zu visualisieren.

Im folgenden soll die Modellierung der Basisstruktur durch das Beispiel «Ticketing-System» verdeutlicht werden (siehe Abbildung 30). Die Verkaufsstellen reservieren bzw. buchen Tickets ab, indem sie mit entsprechenden Veranstaltungsservern in Kontakt treten. An- und Abmeldung der Verkaufsstellen und der Veranstaltungsserver sowie die physische Zugriffskontrolle wird von der Netzverwaltung übernommen. Die Verkaufsstellen und die Veranstaltungsserver besitzen zusätzlich eine Beschreibung auf Detailebene (symbolisiert durch die «…»-Ergänzung). Die Netzverwaltung soll im Anforderungsmodell nur am Rande und deshalb ohne eine Detailbeschreibung spezifiziert werden. Insgesamt kommunizieren drei unterschiedliche Institutionen mit dem System (modelliert durch externe Akteure):

- Eine beliebige Menge von Kunden (nehmen Ticketing-Dienste in Anspruch)
- Eine beliebige Menge von Serverbetreibern (unterhalten die Veranstaltungsserver)
- sowie ein Operateur (wartet das System).

Die *externen Akteure* werden nur als Referenz aufgeführt, nicht jedoch selber modelliert. Beispielsweise wird keine Aussage darüber gemacht, ob der Operateur eine bestimmte Person dar-

stellt oder dadurch eine Rolle bezeichnet wird, die von unterschiedlichen Personen ausgefüllt werden kann.

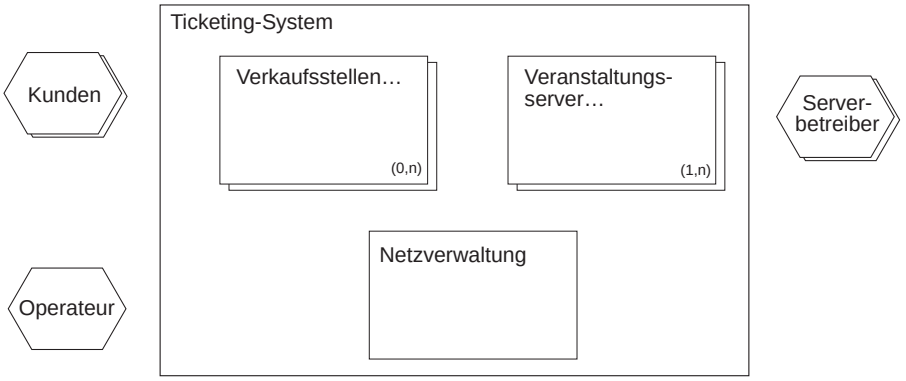


Abb. 30: Die Basisstruktur des Ticketing-Systems im Groben sowie dessen Systemkontext.

In Abbildung 31 wird das Ticketing-System detaillierter durch die Expansion der Komponente Verkaufsstellen dargestellt (ohne aspektbezogene Einblendungen oder den Systemkontext zu berücksichtigen). Die fünf Objektkomponenten von Verkaufsstellen sind in der expandierten Form sichtbar. Von diesen enthalten das Terminal, die Reservation und die Auskunft wiederum weitere eingeschachtelte Objekte. Die Veranstaltungen werden durch eine Objektmenge, d.h. durch eine variabel große Menge von Objektinstanzen modelliert. Anzumerken ist noch, daß der Zusammenhang zwischen den Objektmengen Verkaufsstellen und Veranstaltungen multiplikativ ist, d.h. jede Instanz von Verkaufsstellen besitzt eine eigenständige Menge von Veranstaltungsinstanzen.

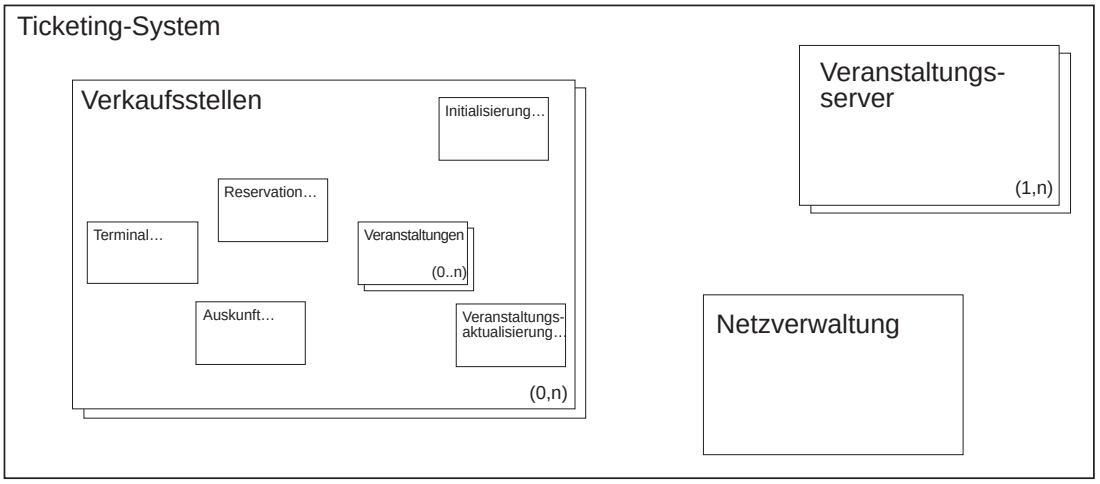


Abb. 31: Das *Ticket-System* mit der Komponente *Verkaufsstellen* in expandierter Darstellung.

Die hier beschriebenen Verkaufsstellen enthalten eine Art Puffer (beschrieben durch Veranstaltungen), in dem Informationen zu den verfügbaren Veranstaltungen lokal in jeder Verkaufsstelle gehalten wird. Die Komponente Veranstaltungsaktualisierung hat die Aufgabe, diesen Puffer regelmäßig mit den tatsächlich existierenden Veranstaltungen abzugleichen. Die Komponenten

Auskunft und Reservation modellieren die beiden zentralen Leistungen einer Verkaufsstelle. Die Initialisierung beschreibt den relativ umfangreichen Vorgang, eine Verkaufsstelle zu initialisieren und im Netz anzumelden.

Die Objektkomponente Reservation schließlich wird in Abbildung 32 ebenfalls in expandierter Form dargestellt. Sie enthält mit dem Ticketdrucker ein externes Objekt, also ein externes System, welches jedoch als Objekt in das Systemmodell integriert ist. Es besitzt einzig eine funktionale Einblendung (siehe Kapitel 7.5), in der die Objektfunktionalität deklariert wird. Auf Detailebene existieren weder eine *strukturelle* noch eine *verhaltensorientierte Einblendung*.

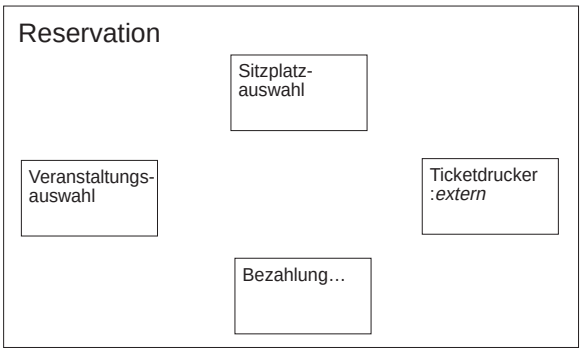


Abb. 32: Die Reservation (Komponente von Verkaufsstellen) mit Ticketdrucker als externem Objekt

6.3 Die strukturelle Einblendung des Objektmodells

Die *strukturelle Einblendung* fügt der ADORA-L-Basisstruktur Sprachkonstrukte hinzu, welche im wesentlichen die Zusammenarbeit und die Kommunikation zwischen Objekten und Objektmengen modellieren. Diese Zusammenhänge werden in der *strukturellen Einblendung* statisch in Form von *Beziehungen* modelliert. *Beziehungen* sind in ADORA-L Grundlage für die eigentliche Modellierung von Nachrichten und von Objektbezügen. In der Verhaltensbeschreibung (siehe Kapitel 6.4) werden diese Beziehungen schließlich verwendet, um konkret Nachrichten von einem Objekt (einer Objektmenge) zu einem anderen zu verschicken. ADORA-L unterscheidet hierbei die beiden Beziehungsarten *Beziehung* und *Benutzung:*

- Die *strukturelle Beziehung* modelliert einen Zusammenhang zwischen zwei Objekten (Objektmengen) als eine Kooperation. Beide Objekte (Objektmengen) sind gleichberechtigte Partner, die beliebig Nachrichten und Informationen miteinander austauschen bzw. sich gegenseitig referenzieren können. Eine *strukturelle Beziehung* kann sowohl gerichtet als auch ungerichtet verwendet werden. Ungerichtet wird sie in beide Richtungen durch Kardinalitäten und durch Beziehungsbezeichnungen näher spezifiziert. Ist sie gerichtet, erfolgt die Attributierung nur in eine Richtung. Notiert werden strukturelle Beziehungen durch eine einfache Linie zwischen den beteiligten Komponenten.

- Die *Benutzung* modelliert die Kommunikationsverbindung zwischen zwei Objekten (Objektmengen) als eine Delegierung. Eine der beiden Komponenten stellt Dienstleistungen zur Verfügung, welche von der anderen Komponente in Anspruch genommen werden. Die Benutzung ist ein gerichteter Zusammenhang. Sie wird in Auftragnehmer/Auftraggeber-Richtung durch Kardinalitäten näher spezifiziert. Notiert wird die Benutzung durch eine einfache bepfeilte Linie zwischen den beteiligten Komponenten.

Abbildung 33 zeigt einen schematischen Überblick über eine ADORA-L-Basisstruktur, ergänzt durch eine strukturelle Einblendung.

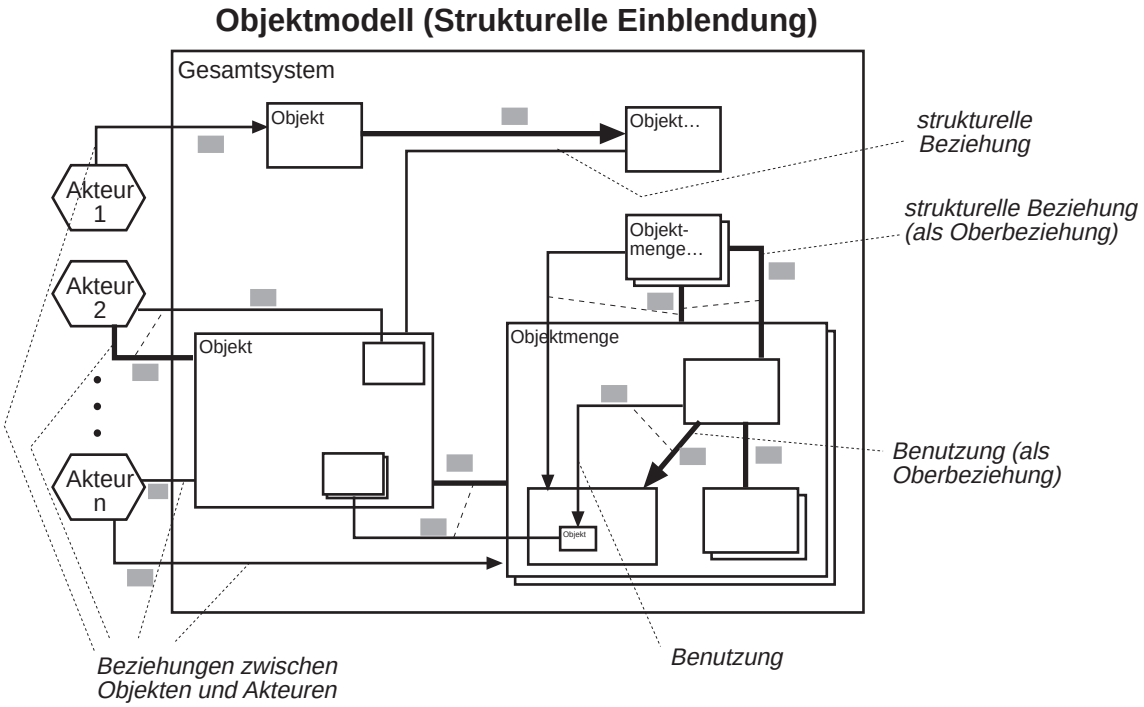


Abb. 33: Skizze einer strukturellen Einblendung in die Basisstruktur. Die grauen Kästchen sind Platzhalter für die Bezeichnung sowie die Spezifikation der Kardinalitätsspezifikation der jeweiligen Beziehung.

Beziehungen werden ebenfalls verwendet, um den Zusammenhang des Systems mit seinem Kontext zu modellieren. Die *externen Akteure* und die *externen Objekte* werden hierfür wie Objekte (bzw. Objektmengen) behandelt. Im Unterschied zu den obigen Beziehungen beschreiben sie hier jedoch nicht Strukturen innerhalb des Objektmodells, sondern statische Zusammenhänge und Kommunikationskanäle zwischen dem System und seinem Kontext. Desweiteren gelten für sie alle Eigenschaften von Beziehungen zwischen Objekten (insbesondere gelten auch die Integritätsbedingungen für Beziehungen aus Kapitel 5.3.3).

Die Besonderheit des ADORA-L-Objektmodells ist die Verwendung einer Teil/Ganzes-Objekthierarchie, in der Objekte auf mehreren Abstraktionsebenen beschrieben werden. Diese Form der hierarchischen Modellierung erfordert jedoch ein entsprechendes Konzept zur

Modellierung von verschieden abstrakten Beziehungen: Beziehungen auf Detailebene müssen sich in sinnvoller Form auf abstrakter Ebene wiederfinden (siehe Kapitel 5.3.3). Hierfür wird in ADORA-L das Konzept der *hierarchischen Beziehungen* eingesetzt. Dieses Konzept unterscheidet zwischen einfachen Beziehungen und *Oberbeziehungen*, welche Beziehungen auf Detailebene zusammenfassen. *Oberbeziehungen* werden in ADORA-L durch dicke Linien (im Falle von strukturellen Beziehungen) oder durch dick bepfeilte Linien (im Falle von Benutzungen) dargestellt. Die Zusammenhänge zwischen einer Beziehung und ihrer Oberbeziehung, also die *Beziehungsstruktur*, werden durch gestrichelte Verbindungslinien beschrieben (siehe Abbildung 33).

Eine Beziehung wird nur dann dargestellt, wenn beide beteiligten Objekte (Objektmengen) in der Visualisierung sichtbar sind. Eine Beziehung ist nicht sichtbar, wenn eines der beiden beteiligten Objekte (Objektmengen) Komponente einer Komposition ist und diese Komposition wiederum (in der aktuellen Visualisierung) nur in komprimierter Form vorliegt. In diesem Fall wird die Beziehung durch eine geeignete Oberbeziehung vertreten und dadurch in abstrakter Form visualisiert. Unbedingte Voraussetzung hierfür ist jedoch die Gewährleistung der Integritätsbedingungen *Vollständigkeit der Oberbeziehungen* und *Zuordnung von Oberbeziehungen* (siehe Kapitel 5.3.3).

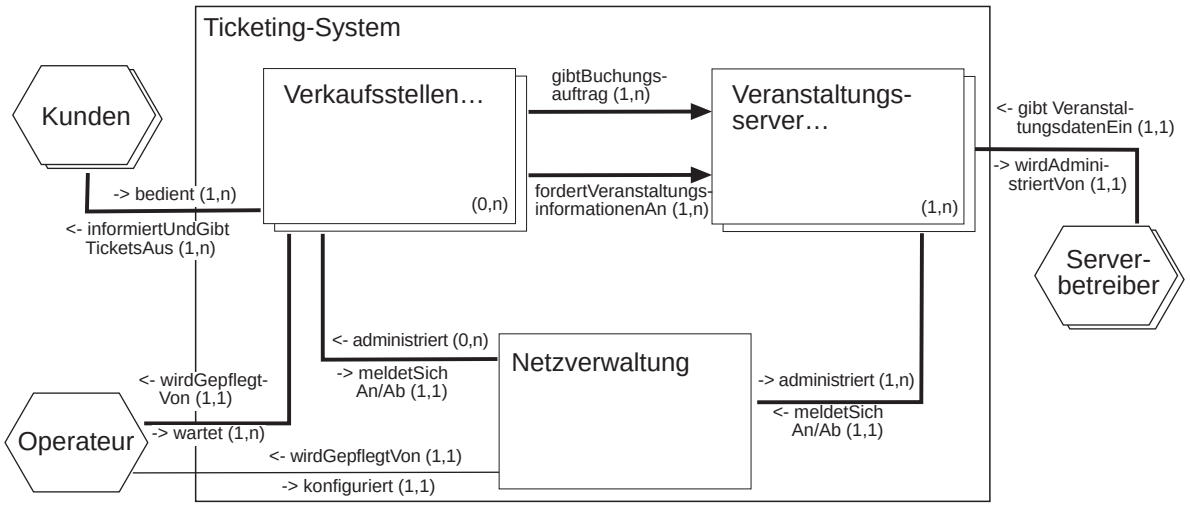


Abb. 34: Ein Überblick über das Ticketing-System in seiner Basisstruktur mit der strukturellen Einblendung

Abbildung 34 zeigt das Ticketing-System in der Basisstruktur aus Abbildung 31 zusammen mit der *strukturellen Einblendung*: Die Komponenten Verkaufsstellen und Veranstaltungsserver liegen in komprimierter Form vor. Die *strukturellen Beziehungen* und die *Benutzungen* zwischen den Komponenten des Ticketing-Systems werden als Oberbeziehungen modelliert. Sie abstrahieren also von tieferliegenden Beziehungen zwischen Komponenten der drei Verkaufsstellen-Komponenten. Die externen Akteure Kunden, Operateur und Serverbetreiber werden in der strukturellen Einblendung wie Objekte behandelt (sie können im Unterschied zu Objekten jedoch keine Komponenten enthalten). Eine Menge von Kunden sowie ein Operateur interagie-

ren mit den Verkaufsstellen. Da hier ebenfalls Oberbeziehungen verwendet werden, existieren weitere tiefergeschachtelte Komponenten, zu denen diese *externen Akteure* Beziehungen haben. Die Beziehung *wirdGepflegtVon-konfiguriert* zwischen dem Operateur und der Netzverwaltung muß hingegen eine einfache Beziehung sein, da die Netzverwaltung keine eingeschachtelten Objekte enthält.

In der folgenden Abbildung 35 werden die Verkaufsstellen in detaillierter Form dargestellt. Sichtbar sind nun Beziehungen auf Detailebene, welche durch die Oberbeziehungen aus Abbildung 34 in abstrakter Form beschrieben werden. Die Beziehung zwischen Kunden und Verkaufsstellen wurde verfeinert, indem die Beziehung gibtAnfrageEin als Unterbeziehung zu bedient eingeführt wurde.

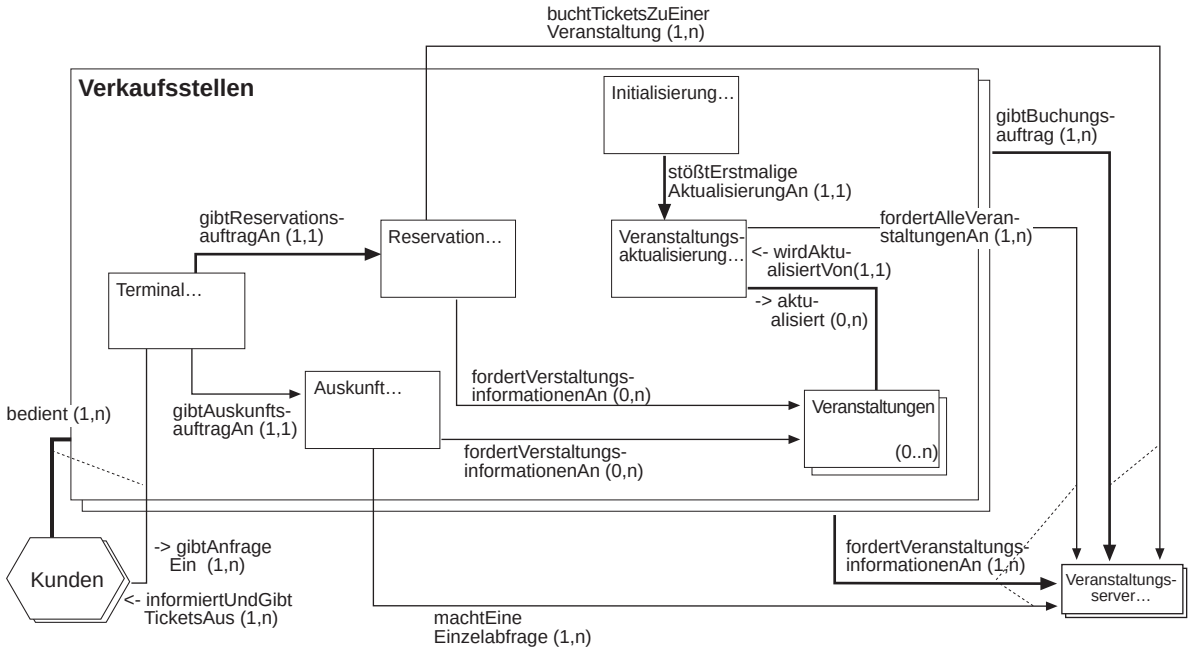


Abb. 35: Die Objektmenge Verkaufsstellen (Komponente des Ticketing-Systems) in expandierter Form dargestellt.

Zusätzlich werden Beziehungen zwischen den Komponenten der Objektmenge Verkaufsstellen spezifiziert. Die Komponenten Reservation und Auskunft verwenden den Informationspuffer Veranstaltungen, um ihre Leistungen anzubieten. Tatsächliche Buchungen oder Sitzplatzabfragen werden jedoch direkt mit dem jeweiligen Verkaufsserver abgeklärt. In Abbildung 36 wird das Objekt Reservation modelliert. Wiederum zu sehen sind die weiter verfeinerten Beziehungen zwischen der Komponente einer Komponente einer Verkaufsstelle und seiner Umgebung. Die Beziehung fragtNachFreienPlätzen ist entsprechend eine Unterbeziehung der Unterbeziehung gibtBuchungsauftrag. Die Beziehung holtAlleVeranstaltungen hingegen ist eine Beziehung innerhalb einer Verkaufsstelle, sie geht zur Komponente Veranstaltungen.

Anzumerken ist noch, daß hierarchisch gegliederte Beziehungen nicht notwendigerweise unterschiedlich bezeichnet werden müssen. Die Beziehung fragtNachFreienPlätzen beispiels-

weise könnte weitere Unterbeziehungen besitzen, welche das Objekt Platzauswahl mit Komponenten des Veranstaltungsservers verbinden. Damit das Objektmodell nicht unnötig mit unterschiedlich bezeichneten Beziehungen überladen wird, ist häufig eine Gleichbenennung sinnvoll (siehe die beiden namensgleichen Beziehungen fordertVeranstaltungsinformationenAn).

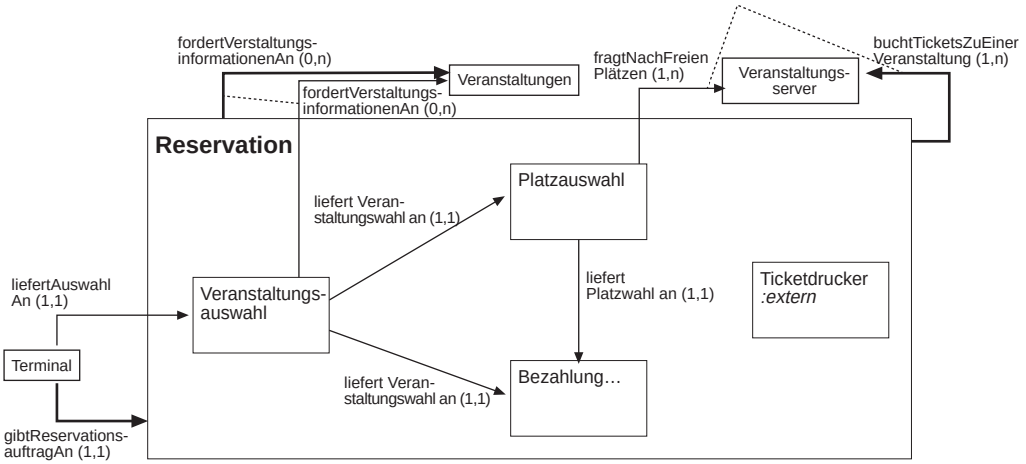


Abb. 36: Das Objekt Reservation (Komponente der Verkaufsstellen) in expandierter Form dargestellt.

6.4 Die verhaltensorientierte Einblendung des Objektmodells

Die *verhaltensorientierte Einblendung* ergänzt die ADORA-L-Basisstruktur durch Sprachkonstrukte, welche das Verhalten von Objekten und Objektmengen modellieren. Die Verhaltensbeschreibung ist in die Teil/Ganzes-Objekthierarchie integriert und orientiert sich an der dort verwendeten Hierarchie. Verhalten wird grundsätzlich auf Grundlage von Zuständen und Zustandsübergängen modelliert. ADORA-L beschreibt das Verhalten eines Objektes (einer Objektmenge) als eine Abstraktion seines Detailverhaltens, also des Verhaltens der eingeschachtelten Objektkomponenten. Die Komponenten werden in die Verhaltensbeschreibung eines Objektes (bzw. einer Objektmenge) in Form von *Komponentenzuständen* integriert. Diese *Komponentenzustände* enthalten wiederum die Verhaltensbeschreibung der jeweiligen Komponente. Dadurch findet sich die Teil/Ganzes-Hierarchie in der hierarchischen Verhaltensmodellierung wieder, die verhaltensorientierte Einblendung ist also mit der ADORA-L-Basisstruktur kompatibel. In ihr werden neben den *Komponentenzuständen* die beiden Zustandsarten *elementare* und *komplexe Zustände* (siehe Kapitel 5.2) verwendet. Diese Zustandsarten werden verwendet, um neben den *Komponentenzuständen* noch weitere Objektzustände zu modellieren.

Die Notation der Komponentenzustände ist identisch mit der entsprechenden Objekt- bzw. Objektmengen-Notation. *Elementare Zustände* werden durch ovale, *komplexe Zustände* durch abgerundete Rechtecke dargestellt. *Komplexe Zustände* sind analog zum Mechanismus der

Objektdarstellung in *expandierter* und *komprimierter* Form visualisierbar. Abbildung 37 zeigt eine *verhaltensorientierte Einblendung* im Überblick.

Alle Zustände eines Objektmodells sind prinzipiell nebenläufig, falls sie nicht direkt oder indirekt durch einen Zustandsübergang miteinander verbunden sind. Ein Zustandsübergang spezifiziert anhand einer *Übergangsbedingung* und anhand von *Übergangsaktionen*, unter welchen Umständen ein Zustand gewechselt wird und welche Auswirkung dieser Zustandswechsel auf die Umgebung hat. Ein *Zustandsübergang* wird in ADORA-L durch eine bepfeilte gestrichelte Linie zwischen dem Quell- und dem Zielzustand notiert. *Zustandsübergänge* und *Übergangsbedingungen* können wahlweise in Prosa (durch natürliche Sprache), formal (durch die formale Sprache ADORA-FSL, siehe Kapitel 6.7) oder teilformal (durch eine Mischung von Prosa mit formalen Schüsselworten) spezifiziert werden. Ausdrücklich ist es möglich, einen Zustandsübergang multipel durch unterschiedliche Formalitätsgrade zu beschreiben.

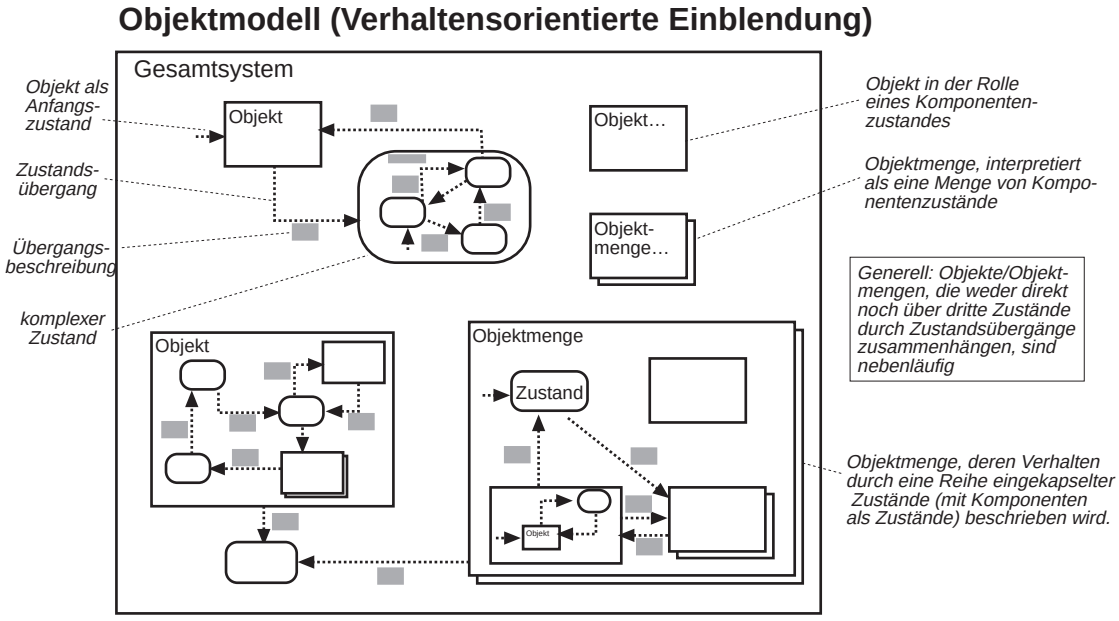


Abb. 37: Skizze einer Basisstruktur und einer darin integrierten verhaltensorientierten Einblendung

Von besonderer Bedeutung ist die Spezifikation von *Übergangsbedingungen* und *-aktionen* mit Hilfe von *Objektoperationen* (siehe Kapitel 6.5). Sinnvoll ist dies vor allem dann, wenn komplexe Sachverhalte spezifiziert werden müssen (und die Verhaltensbeschreibung dadurch unübersichtlich wird) oder wenn häufig dieselben Sachverhalte an verschiedenen Stellen der Verhaltensbeschreibung aufgeführt werden.

In Abbildung 38 wird als Beispiel für eine verhaltensorientierte Einblendung die Objektmenge Verkaufsstellen modelliert. Die Komponenten aus der Basisstruktur werden hier integriert als *Komponentenzustand* (im Falle von Objekten) bzw. als Menge nebenläufiger *Komponentenzustände* (im Falle von Objektmengen) interpretiert. Eine Zustandskomponente kann demzufolge aktiv oder auch passiv (also temporär nicht existent) sein. Die Komponente Initialisierung bei-

spielsweise ist nur zu Beginn bzw. beim Neuanfahren der Verkaufsstelle aktiv und nur dann durch eine Objektinstanz konkret vertreten. Bei Aktivierung der Initialisierung wird diese neu auf den internen Anfangszustand zurückgesetzt. Die Komponenten Reservation, Auskunft und Veranstaltungsaktualisierung sind dann aktiv, wenn auch der komplexe Zustand aktiviert aktiv ist. Die Komponenten Terminal und Veranstaltungen hingegen sind permanent aktiv, d.h. nebenläufig zur restlichen Verhaltensbeschreibung.

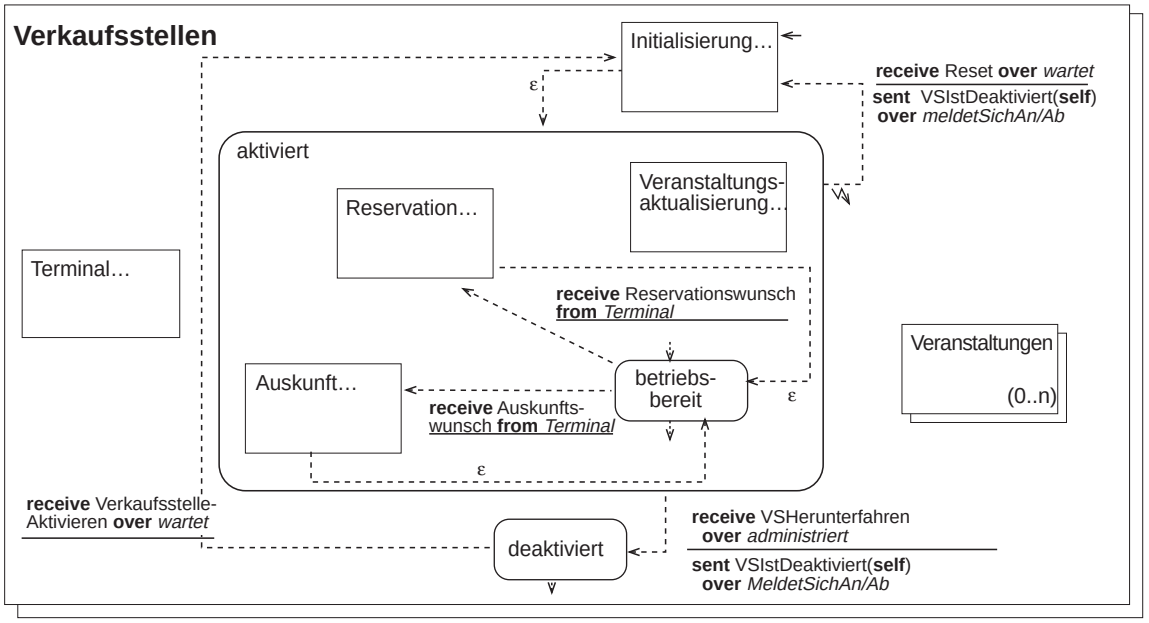


Abb. 38: Die Objektmenge Verkaufsstelle (Komponente des Ticketing-Systems) in der verhaltensorientierten Einblendung.

Die Übergänge werden im Beispiel formal durch ADORA-FSL spezifiziert. Die verwendeten ε-Übergänge beschreiben einen unbedingten Übergang, d.h. ihre Übergangsbedingung ist stets erfüllt. Nachrichten werden durch Nennung des verwendeten Nachrichtenkanals, d.h. der Beziehung versendet und empfangen (siehe Zustandsübergänge zum Zustand deaktiviert hin und von ihm weg). Ausnahme hierbei sind Nachrichten, die von einer Komponente der Verhaltensbeschreibung kommen bzw. an eine solche Komponente verschickt werden (siehe Zustandsübergänge innerhalb des komplexen Zustandes aktiviert). In diesem Fall werden die jeweiligen Komponentenbeziehungen direkt aufgeführt.

In Abbildung 39 wird die Verhaltenbeschreibung der Komponente Reservation modelliert. Zunächst erfolgt die Auswahl der gewünschten Veranstaltung. Diese ist hier als eigenständige Objektkomponente beschrieben[1]. Nach Beendigung von Veranstaltungsauswahl werden wiederum in einer eigenständigen Komponente Sitzplatzauswahl die zu reservierenden Sitzplätze

[1] Alternativ könnte Veranstaltungsauswahl auch als komplexer Zustand modelliert werden, welcher keine lokale Daten halten kann. Die notwendigen Daten müßten also nicht als Nachrichtenparameter übergeben, sondern könnten direkt als Zugriffe auf die Attribute des Objekts Reservation modelliert werden.

ausgewählt (hierfür muß Sitzplatzauswahl Kontakt mit dem Veranstaltungsserver aufnehmen). Mit der gewählte Veranstaltung sowie den gewünschten Sitzplätzen wird nun ein Buchungsversuch beim Veranstaltungsserver unternommen. Der Veranstaltungsserver bestätigt diesen Versuch (d.h. er bucht die gewünschten Plätze) oder er verwirft ihn (Bemerkung: Auf einen Veranstaltungsserver können zwischen der Auswahl der Sitzplätze und dem Reservierungswunsch andere Verkaufsstellen asynchron zugreifen).

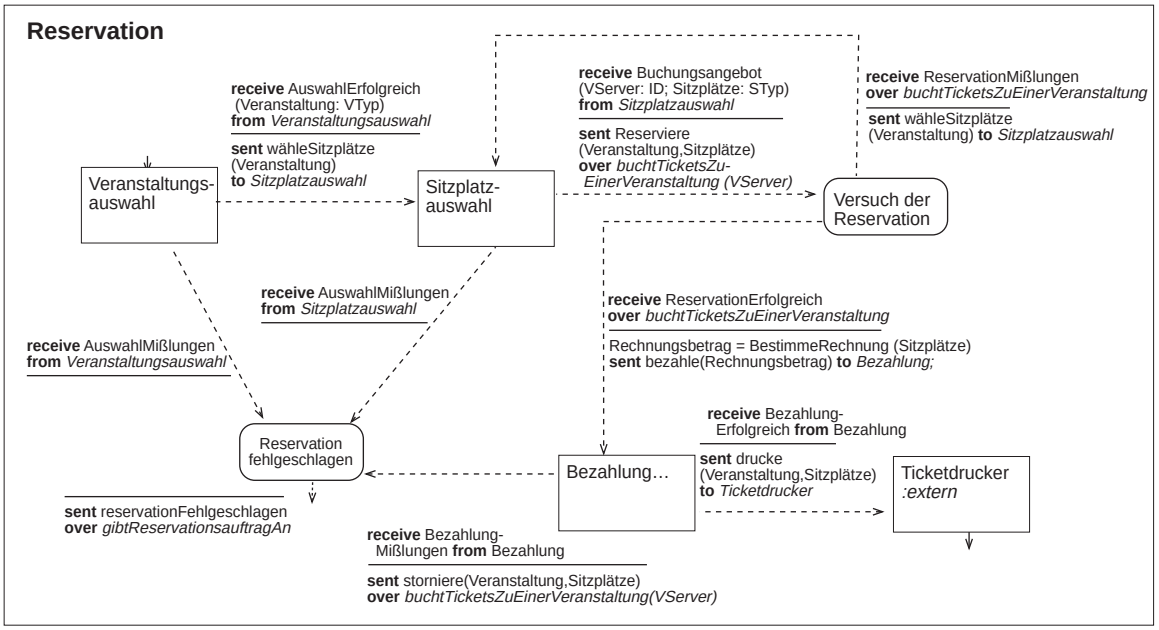


Abb. 39: Skizze einer verhaltensorientierten Einblendung des Objekts Reservation (Komponente der Verkaufsstellen)

Ein Besonderheit in diesem Beispiel sind die Versendungen der Art

sent reserviere (Veranstaltung, Sitzplätze) **over** buchtTicketsZuEinerVeranstaltung(VServer)

und

sent storniere (Veranstaltung, Sitzplätze) **over** buchtTicketsZuEinerVeranstaltung(VServer)

Hierbei ist zu beachten, daß Veranstaltungsserver und Verkaufsstellen Objektmengen sind, also jeweils mehrere Instanzen existieren. Beim Versenden der beiden Nachricht wird die genaue ID der Zielinstanz, welche sich innerhalb einer Objektmenge befindet, mitgeliefert. Auf Seiten des Veranstaltungsservers wird hier die Nachricht *nur an diese* eine Objektinstanz weitergeleitet. Wurde die Veranstaltung erfolgreich gebucht, erfolgt entsprechend die Bezahlung (im Objekt Bezahlung) und der Druck der Tickets (im Objekt Ticketdrucker). Mißlingt der Bezahlungsvorgang, wird eine entsprechende Rücksetznachricht an den Veranstaltungsserver gesendet. (Bemerkung: Eine Reaktion auf den Ausfall des Ticketdruckers wurde in diesem Systemmodell nicht berücksichtigt.)

6.5 Die funktionale Einblendung des Objektmodells

Die *funktionale Einblendung* fügt der Objektmodell-Basisstruktur Sprachkonstrukte hinzu, welche die funktionalen Eigenschaften von Objekten bzw. Objektmengen spezifizieren. Die Einblendung ist im Gegensatz zur *strukturellen* und zur *verhaltensorientierten Einblendung* rein textuell. Sie kann somit nicht überlappend mit den anderen Einblendungen dargestellt werden, sondern wird als eine zusätzliche eigenständige Textbeschreibung von Objekten (bzw. von Objektmengen) annotiert. Abbildung 40 zeigt einen Überblick einer *funktionalen Einblendung* in eine ADORA-L-Basisstruktur. Jedem Objekt, jeder Objektmenge sowie jedem externen Objekt wird eine *Elementarbeschreibung* zugeordnet.

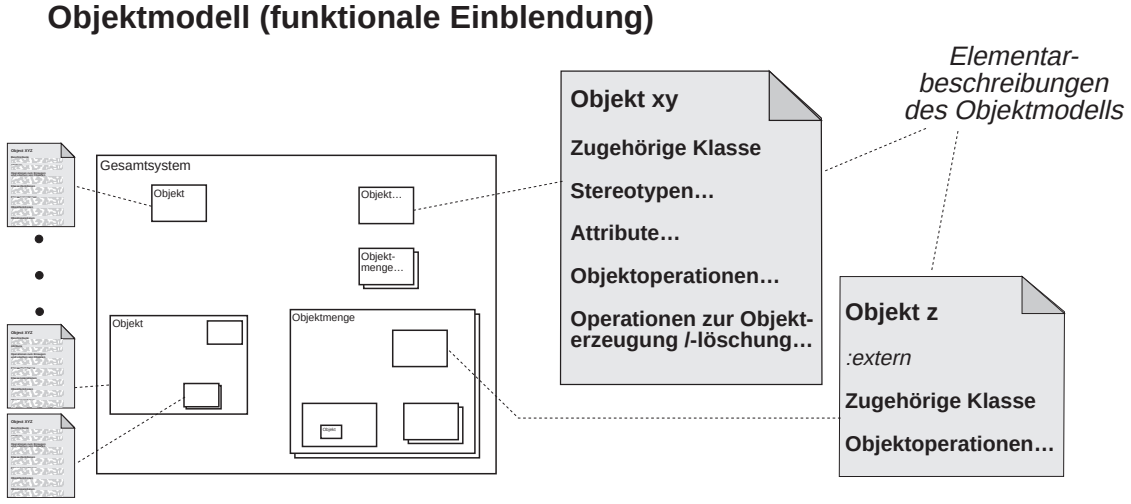


Abb. 40: Skizze der funktionalen Einblendung in die Basisstruktur

Die *Elementarbeschreibung* eines Objekts macht Aussagen darüber,

- welcher Klasse das Objekt zugeordnet ist
- welche Stereotypen ihr zugewiesen sind
- welche Attribute und welche Attributtypen das Objekt besitzt
- welche Initialisierungswerte diese Attribute haben
- wie die zeitbehafteten *Objektoperationen* und die zeitlosen *Objektoperationen* axiomatisch spezifiziert werden (durch die Angabe einer Voraussetzung und einer Zusicherung der Objektoperation), siehe Kapitel 5.5

Die Elementarbeschreibung einer Objektmenge enthält neben diesen Sachverhalten noch

- eine axiomatische Spezifikation der Metaoperationen der Objektmenge, d.h. die Erzeugungs- oder Löschoperationen, um neue Instanzen in die Objektmenge einzufügen und vorhandene Instanzen zu löschen.

Die *Elementarbeschreibung* eines externen Objekts beschreibt ausschließlich die Leistungsdaten des externen Objektes, tiefergehende Spezifikationen entfallen dabei. Diese Elementarbeschreibung enthält

- die Zuordnung zu einer Klasse (diese Klasse ist ebenfalls extern, d.h. nur externe Objekte können ihr zugeordnet werden) sowie
- die axiomatische Spezifikation der Objektoperationen (ebenfalls durch die Angabe einer Operationsvoraussetzung und einer Operationszusicherung)

Die Elementarbeschreibungen eines Objektmodells sind analog zu der textuellen Spezifikation von Zustandsübergängen variabel formal. Je nach gewünschter Präzision sind Elementarbeschreibungen oder auch einzelne Abschnitte sowohl als Prosa, formal oder teilformal formulierbar (Grundlage der formalen und teilformalen Sprache ist die formale Sprache ADORA-FSL). Kapitel 6.7 geht im Detail auf die Varianten der textuellen Funktionsspezifikation in ADORA-L ein.

6.6 Das Typverzeichnis des Objektmodells

Das ADORA-L-Typverzeichnis ist eine zur Teil/Ganzes-Objekthierarchie orthogonale Beschreibung. In ihm werden *Klassen*, *Stereotypen* und *Datentypen* spezifiziert, welche im Objektmodell ausgeprägt oder angewandt werden (siehe Abbildung 41).

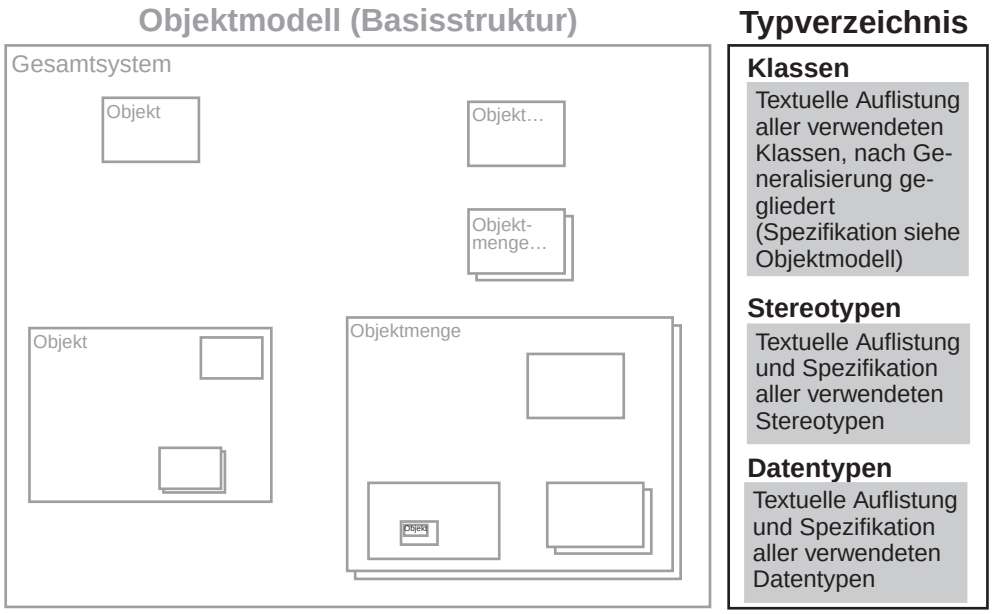


Abb. 41: Übersicht über das Typverzeichnis in ADORA-L

Klassen werden in ADORA-L grundsätzlich und ausschließlich in der Bedeutung eines Typs, d.h. eines Objekttyps verwendet (siehe Kapitel 5.1). Sie erfassen Gemeinsamkeiten und Ähn-

lichkeiten zwischen Objekten eines Objektmodells. Die Gemeinsamkeiten werden durch *Objektschablonen*, die Ähnlichkeiten durch *Spezialisierungen* und durch *Klassenhierarchien* spezifiziert (siehe Abbildung 42). Im Typverzeichnis sind nur Klassen aufgeführt, deren Objekte und Objektmengen

(i) in mehr als einem Kontext des Objektmodells, d.h. an mehr als einer Stelle verwendet werden oder

(ii) als Spezialisierungen anderer Objekte (Objektmengen) modelliert werden sollen.

Andernfalls ist eine Nennung des Objekttyps nicht notwendig, da nur ein einziges Objekt (bzw. eine Objektmenge) diesen Objekttyp besitzt und dieser Typ somit nicht zentral aufgeführt werden muß.

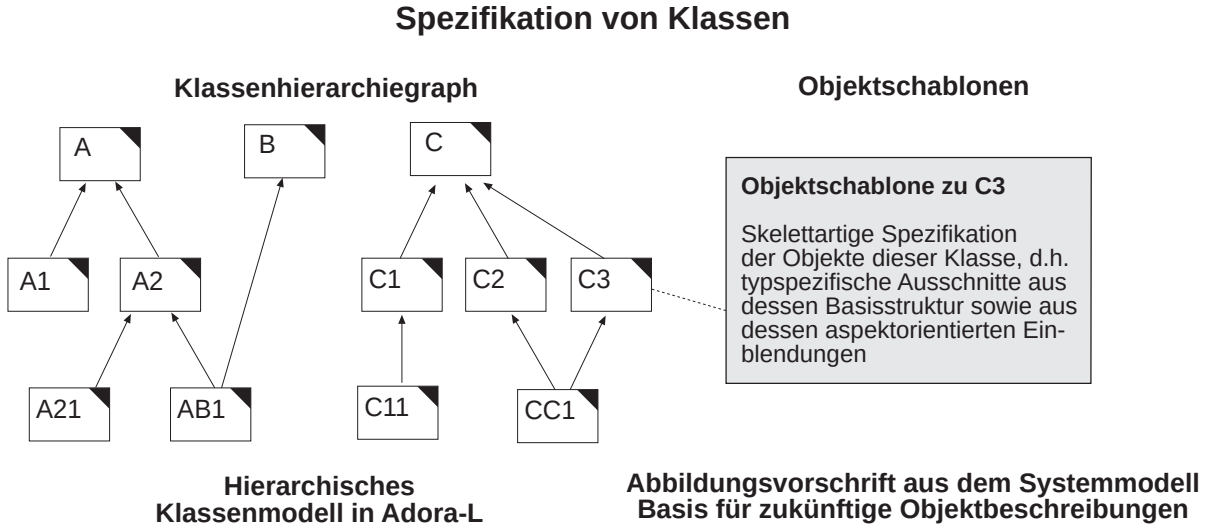


Abb. 42: Skizze eines Klassenhierarchiebaums und die Klassendefinition als Objektschablone. Die Objektschablone ist durch ein zugehöriges Objekt des Objektmodells definiert und kann automatisch generiert werden. Es ist Grundlage für neue typgleiche Objekte (Objektmengen), welche dem Modell hinzugefügt werden.

Die *Klassenhierarchie* klärt Generalisierungs/Spezialisierungszusammenhänge zwischen Klassen (siehe Kapitel 5.6). Eine *Objektschablone* ist eine eindeutig definierte Abbildung der typspezifischen Eigenschaften aus den zugehörigen Objekten des Objektmodells. Sie werden primär verwendet, um neu zu beschreibende Objekte (bzw. Objektmengen) zunächst kontextunabhängig zu spezifizieren, und diese nach und nach durch kontextbezogene Informationen zu ergänzen.

Stereotypen klassifizieren die Elemente eines Modells zusätzlich zu bereits vorhandenen Klassifikationsarten. Ein Stereotyp einer Modellierungssprache ist ein Beschreibungsmechanismus, der zu bestehenden Sprachkonstrukten präzisierende, ergänzende oder verändernde Aussagen über Modellelemente macht (nach [Joos98]). *Stereotypen* sind also als ein zur Klassenhierar-

chie orthogonaler Klassifikationsmechanismus zu verstehen. Außer Klassen können aber auch andere Modellelemente wie beispielsweise Objekte oder *Beziehungen* (siehe Kapitel 5.6) klassifiziert werden. *Stereotypen* werden in ADORA-L mit Hilfe eines formatierten *Prosatextes* spezifiziert (siehe Abbildung 43). Bei der Beschreibung von Stereotypen wird spezifiziert,

- welchen Sprachkonstrukten der Stereotyp zugewiesen werden kann. Explizit ist es auch möglich, einen konkreten Stereotyp mehreren Modellelementen (also einer Gruppierung von Modellelementen) zuzuweisen
- welche Bedeutung der Stereotyp hat und welche Eigenschaften einem Modellelement zugewiesen werden
- welche Variablen verwendet werden, um das Modellelement näher zu beschreiben
- welche Restriktionen an das Modellelement gestellt werden

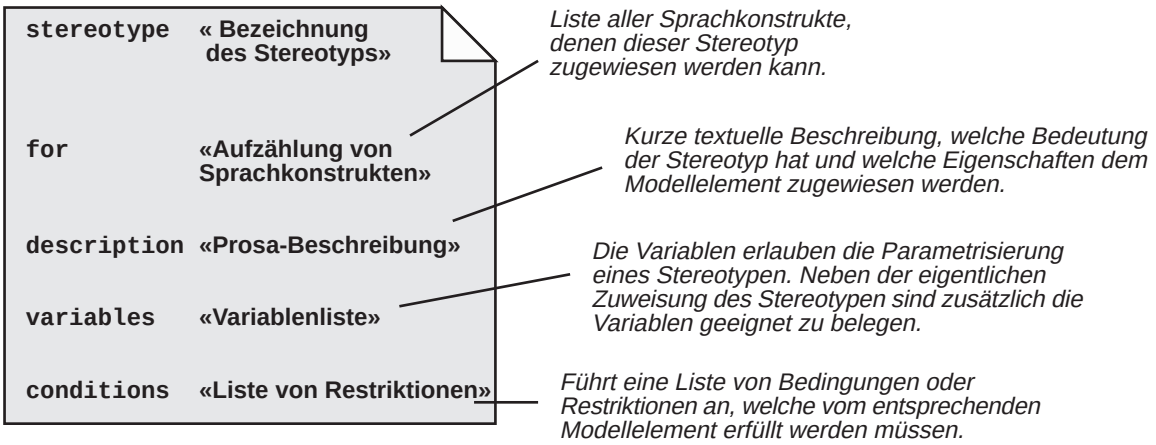


Abb. 43: Beschreibung von Stereotypen durch formatierten Prosatext

> Beispiel: Der Stereotyp Version kann dem Sprachkonstrukt *Objekt* und *Klasse* zugewiesen werden. Er beschreibt, wann das Modellelement erstellt, wann es zuletzt geändert wurde und welche Personen für die Erstellung und die Änderung verantwortlich sind. Die Stereotypenvariablen sind Erstellungsdatum, Änderungsdatum, Autoren, jeweils vom Typ String. Version besitzt genau eine Restriktion: Das Erstellungsdatum muß kleiner sein als das Änderungsdatum.

Die im Typverzeichnis deklarierten *Datentypen* schließlich werden verwendet, um Typen von Objektattributen oder von Nachrichtenparametern an zentraler Stelle und somit in allen Objektbeschreibungen zur Verfügung zu stellen. Zwar können Datentypen auch lokal in Objekten (bzw. Objektmengen) spezifiziert werden. Probleme ergeben sich jedoch dann, wenn Daten eines bestimmten Typs von Objekt zu Objekt weitergereicht werden und lokal definierte Datentypen für andere Objekte nicht sichtbar sind. Für die Notation der Datentypen wird die Sprache ADORA-FSL verwendet (siehe folgendes Kapitel).

6.7 Formale, teilformale und informale ADORA-L-Detailsprachen

ADORA-L spezifiziert ein Systemmodell durch textuelle und durch grafische Sprachkonstrukte. Grafik wird im wesentlichen verwendet, um den groben Aufbau und die Struktur zu modellieren. Dies zeigt sich besonders deutlich bei der ADORA-L-*Basisstruktur* und bei der *strukturellen* und der *verhaltensorientierten Einblendung*. Textuelle Beschreibungen werden eher dann verwendet, um Sachverhalte im Detail zu erfassen. Sie finden sich in den *Elementarbeschreibungen* (der *funktionalen Einblendung*) und der Spezifikation der Zustandsübergänge (der *verhaltensorientierten Einblendung*). Hierbei werden verschieden formale Varianten der textuellen Spezifikation unterstützt. Generell kann innerhalb eines Systemmodells der Grad der textuellen Formalisierung frei gewählt werden, selbst die multiple Spezifikation eines Sachverhaltes in unterschiedlichen Formalitätsgraden wird unterstützt. Unterschieden werden folgende Formalitätsgrade:

- *informal* – durch die Verwendung einer natürlichen Sprache
- *formal* – durch ADORA-FSL (ADORA Formal Specification Language), eine formale Teilsprache von ADORA-L
- *teilformal* – durch Vermischung von ADORA-FSL-Schlüsselwörtern und natürlicher Sprache
- Mehrfachspezifikation eines Sachverhaltes durch eine Kombination der obigen Varianten (Bsp: Informale und gleichzeitig formale Beschreibung einer Zustandsübergangsbedingung). **Wichtig:** Maßgebend ist jedoch stets die formellere Beschreibung.

Informale textuelle Beschreibungen werden durch einfache oder doppelte Anführungszeichen am Beginn und am Ende kenntlich gemacht («"» und «'»).

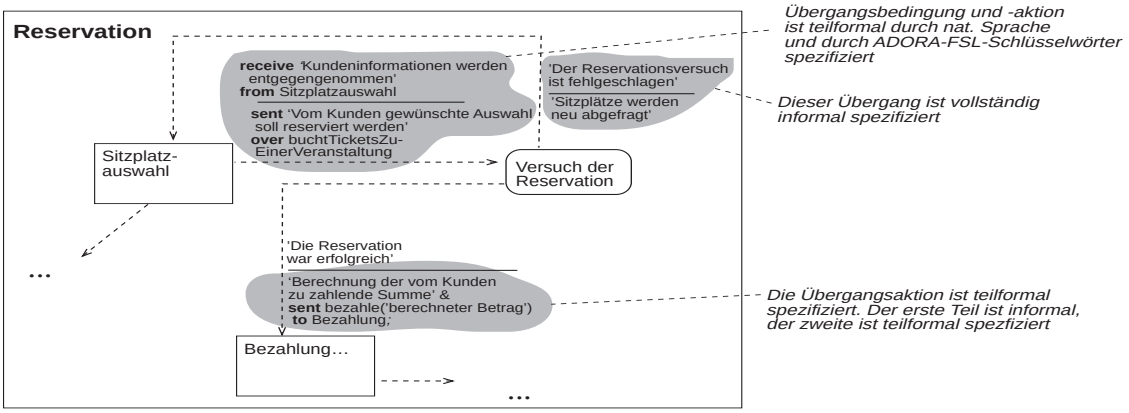


Abb. 44: Beispiel aus Abbildung 39, jedoch mit informalen und teilformalen textuellen Beschreibungen

In Abbildung 44 wird anhand der Verhaltensbeschreibung aus Abbildung 39 der Ansatz der informalen und der teilformalen Textspezifikation erläutert (in Abbildung 39 findet sich die entsprechende formale Variante durch ADORA-FSL). Formale Teilbeschreibungen können entweder teilweise oder vollständig mit Hilfe natürlicher Sprache vergröbert beschrieben werden. Werden hierfür noch ADORA-FSL-Schlüsselwörter verwendet (beispielsweise beim Übergang Sitzplatzauswahl -> VersuchderReservation), so handelt es sich um eine teilformale Beschreibung, andernfalls ist die Beschreibung rein informal (siehe Übergang VersuchderReservation -> Sitzplatzauswahl). Der Übergang vom Zustand VersuchderReservation zu Bezahlung ist informal (in der Übergangsbedingung) und teilformal (in der Übergangsbedingung). Zu beachten ist hierbei, daß zwar in der Übergangsaktion eine Aktivität vollständig informal beschrieben ist, diese aber in einen teilformalen Teil eingebettet ist.

6.7.1 Die formale Detailsprache ADORA-FSL

Im folgenden wird die Sprache ADORA-FSL im Überblick vorgestellt. Sie lehnt sich grundsätzlich an die Sprachkonzepte und -konstrukte der Sprache ASTRAL an [Coen91, 93, 95, 97]. Die Sprache ASTRAL ist eine axiomatische formale Sprache zur Spezifikation von Problemen im Echtzeitbereich. Sie eignet sich als Basis für ADORA-L, da sie relativ einfach aufgebaut und durch den geringen Sprachumfang schnell erlern- und beherrschbar ist. ADORA-L benötigt jedoch nicht die ganze Mächtigkeit von ASTRAL, da nur einzelne funktionale Teilaspekte spezifiziert werden. ADORA-FSL wird im wesentlichen für vier Einsatzgebiete verwendet:

- zur Spezifikation von Datentypen (im Typverzeichnis oder lokal in funktionalen Objektspezifikationen)
- zur Deklaration von Objektattributen, Metaattributen von Objekten/Klassen sowie von Nachrichtenparametern durch die spezifizierten Datentypen.
- zur Beschreibung von Nachrichten durch Spezifikation des Versendes (als eine Aktion) und durch Spezifikations des Empfangs (als eine Bedingung für einen Zustandsübergang oder als Anstoß einer Objektoperation)
- zur axiomatischen Spezifikation von Objektoperationen und Zustandsübergängen mit Hilfe von Voraussetzungen und Zusicherungen [nach Hoare87]

6.7.2 Datentypen in ADORA-FSL

Es werden folgende primitive Datentypen unterschieden: integer, real, boolean, id, mid, time, char und string. Alle anderen einfachen oder konstruierten Datentypen leiten sich aus diesen und/ oder den Konstruktionsregeln is typedef, is, is list of oder is structure of ab.

Beispiel:

Ziffer **is typedef** d: Integer (d ≥ 0 & d ≤ 9)
(Subtyp: Einschränkung eines bekannten Typs)

Ziffernfarbe **is** (rot, grün, blau, gelb)
(Aufzählungstyp)

Ziffernfolge **is list of** Ziffer
(Liste. Enthält eine geordnete Menge von Elementen gleichen Typs)

Colorierte_Ziffernfolge **is structure of** (l : Ziffernfolge
c : Ziffernfarbe)
(Struktur: Enthält verschiedene Elemente, dessen Typen frei wählbar sind)

Von spezieller Bedeutung sind die Typen id und mid. Jede Objektinstanz besitzt eine bzgl. des Systemmodells eindeutige Identifikation, die id. Die id einer Objektinstanz kann innerhalb der Objektbeschreibung mit Hilfe von self abgefragt werden. Der Objekttyp eines Objektes i:id kann über die Metaoperation idtype(i), dessen Oberklasse kann über die Metaoperation supertype(i) abgefragt werden. Der Typ MID wird verwendet, um die Modellelemente selber zu identifizieren. Die Metaoperation idtype(i:id) beispielsweise hat den Ergebnistyp MID. Sie identifiziert den Typ der Objektinstanz i. Weitere *Metaoperationen*[1] sind in siehe Kapitel 7.5.4 aufgeführt.

6.7.3 Variablen in ADORA-FSL

Variablen werden in ADORA-FSL verwendet, um Objektattribute, Objekt-Metaattribute und Nachrichtenparameter zu spezifizieren:

- Objektattribute beschreiben die lokalen Daten eines Objektes. ADORA-L unterscheidet hierbei zwischen *öffentlichen* und *privaten* Objektattributen. *Öffentliche Attribute* (notiert durch das Schlüsselwort «public») sind auch für andere Objekte (Objektmengen) sicht- und lesbar. *Private Attribute* (als Standardfall ohne zusätzliche Schlüsselwörter kenntlich gemacht) hingegen sind ausschließlich vom Objekt (von der Objektmenge) sicht- und manipulierbar.
- Objektmetaattribute sind Informationen der Schemaebene, welche für die Objektspezifikation verwendet werden dürfen (Bsp: Die minimale, maximale oder aktuelle Anzahl der Objekte einer Objektmenge, self)
- Nachrichtenparameter sind die Daten, welche im Zusammenhang mit einer Nachricht verschickt werden.

Variablen werden über einen *Datentyp* deklariert. Die Deklaration kann auf zwei verschiedene Arten erfolgen: Lokal innerhalb der *funktionalen Einblendung* des entsprechenden Objektes

[1] Metaoperationen sind in ADORA-L Operationen, die Eigenschaften der Schemaebene liefern bzw. manipulieren

und global im Typverzeichnis des Systemmodells. Im ersten Fall ist die Variable nur im lokalen Umfeld sicht- und verwendbar. Im letzteren Fall ist der entsprechende Datentyp allen Objekten (Objektmengen) bekannt. Dies ist dann von Bedeutung, wenn verschiedene Objekte (Objektmengen) miteinander kommunizieren und Daten eines Datentyps miteinander austauschen.

Beispiel einer lokalen Objektattribut-Deklaration (Schlüsselwörter sind hier fett geschrieben)

object Sonderangebot
type

Ziffer **is typedef** d: Integer (d ≥ 0 & d ≤ 9),
Ziffernfarbe **is** (rot, grün, blau, gelb),

Ziffernfolge **is list of** Ziffer,
Colorierte_Ziffernfolge **is structure of** (l : Ziffern
folge, c : Ziffernfarbe)

…

var

Aktionspreis, Standardpreis: Colorierte_Ziffernfolge

Minimalpreis: Ziffernfolge

…

6.7.4 Spezifikation von Nachrichten

ADORA-L unterstützt ein explizites Nachrichtenkonzept. Nachrichten werden von der versendenden Objektinstanz durch explizite Nennung der Beziehung zu einer empfangenden Objektinstanz weitergeleitet. Die Spezifikation einer Objektkommunikation umfaßt die Spezifikation des Versendens und die Spezifikation des Empfangs von Nachrichten.

Wie und wo werden Nachrichten versendet?

Nachrichten werden im Rahmen von Zustandsübergängen oder innerhalb von Objektoperationen versendet. Hierfür unterscheidet ADORA-L folgende Arten des Nachrichtenversendens (Anmerkung: Alle Aussagen über das Versenden und das Empfangen von Nachrichten gelten gleichermaßen für Objekte *und* für Objektmengen):

- Das Versenden von asynchronen Nachrichten – Der Zusatz **over** <Beziehungsname> gibt an, über welche Beziehung die Nachricht verschickt werden soll (das Zielobjekt muß mit dem zu versendenden Objekt durch eine Beziehung in Zusammenhang stehen). Rückgabewerte sind bei asynchronen Nachrichten nicht möglich.

  sent <Nachrichtenname> (**in:** Parameterliste) **over** <Beziehungsname>

Wird die Nachricht zu einer Objektkomponente des versendenden Objektes geschickt, wird das Zielobjekt direkt spezifiziert (ein Objekt hat zu seinen Komponenten keine explizit modellierte Beziehung):

sent <Nachrichtenname> (**in:** Parameterliste) **to** <Komponentenname>

Asynchrone Nachrichten können ebenfalls als ein Multicast-sent gebündelt versendet werden (siehe Kapitel 7.4.1):

sent multicast <Mulitcastname> (**in:** Parameterliste) **to** <Komponentenname>

- Das Versenden einer synchronen Nachricht mit Rückgabewert (der Rückgabewert wird zeitlos berechnet, indem eine *Objektoperation* angestoßen wird, welche die Ergebniswerte ermittelt).

  <Nachrichtenname> (**in:** Parameterliste; **out:** Parameterliste ;
  over: Beziehungsname) (ohne impliziten Rückgabewert)

  oder alternativ bei genau einem Rückgabeparameter:

  <Nachrichtenname> (**in:** Parameterliste; **over:** Beziehungsname; Parametertyp)
  (Funktionswert ist ein Parameter vom Typ Parametertyp)

  Ist das Zielobjekt eine Komponente des versendenden Objektes, wird entsprechend zu oben der Ausdruck «over: Beziehungsname» durch «to: Komponentenname» ersetzt.

Wie und wo werden Nachrichten empfangen?

Eine Nachricht ist entweder der Anstoß einer gleichnamigen Objektoperation oder ein zeitgebundenes Bedingungsargument eines Zustandsüberganges. Entsprechend zur Versendung von Nachrichten werden hierbei folgende Varianten unterschieden:

- Zustandsübergangsbedingung für asynchrone Nachrichten und für synchrone Nachrichten ohne Rückgabewert

  receive <Nachrichtenname> (**in:** Parameterliste) **over** <Beziehungsname> bzw.

  receive <Nachrichtenname> (**in:** Parameterliste) **from** <Objektkomponente>

  Die so beschriebenen Zustandsübergangsbedingungen werden als logische Ausdrücke interpretiert, die bei Empfang der entsprechenden Nachricht wahr und ansonsten falsch sind. Multicast-Nachrichten werden statt receive durch receiveall und der Angabe eines Multicastnamen-Identifikators notiert (siehe auch Kapitel 7.4.1):

  receiveAll <Mulitcastname> (**in:** Parameterliste) **from** ...

- Das Empfangen von synchronen Nachrichten mit Ergebniswert, welche also eine unmittelbare Antwort erwarten, sind in der verhaltensorientierten Einblendung **nicht** zulässig, da ein zeitloses Verhalten nicht garantiert ist (siehe Kapitel 6.5).

- Durch eine Nachricht wird eine entsprechende Objektoperation angestoßen. Möglich ist dies für asynchrone und synchrone Nachrichten und synchrone Nachrichten ohne Rückgabewert:

operation <Nachrichtenname> (**in:** Parameterliste) *(asynchrone Operation)*

syncoperation <Nachrichtenname> (**in:** Parameterliste)

syncoperation <Nachrichtenname> (**in:** Parameterliste, **out:** Parameterliste)

syncoperation <Nachrichtenname> (**in:** Parameterliste) : Parametertyp

(synchrone Operationen ohne Rückgabewert, mit Rückgabewerten bzw. mit genau einem Rückgabewert)

In allen Fällen müssen sowohl Nachrichtenname mit dem Operationsname als auch die entsprechenden Parameterlisten übereinstimmen.

6.7.5 Axiomatische Spezifikation von Objektoperationen

Die formale Spezifikation von Objektoperationen erfolgt in ADORA-FSL axiomatisch durch die Angabe von Voraussetzungen und Zusicherungen (nach [Hoare87]). Abbildung 45 soll den prinzipiellen Aufbau der Spezifikation verdeutlichen:

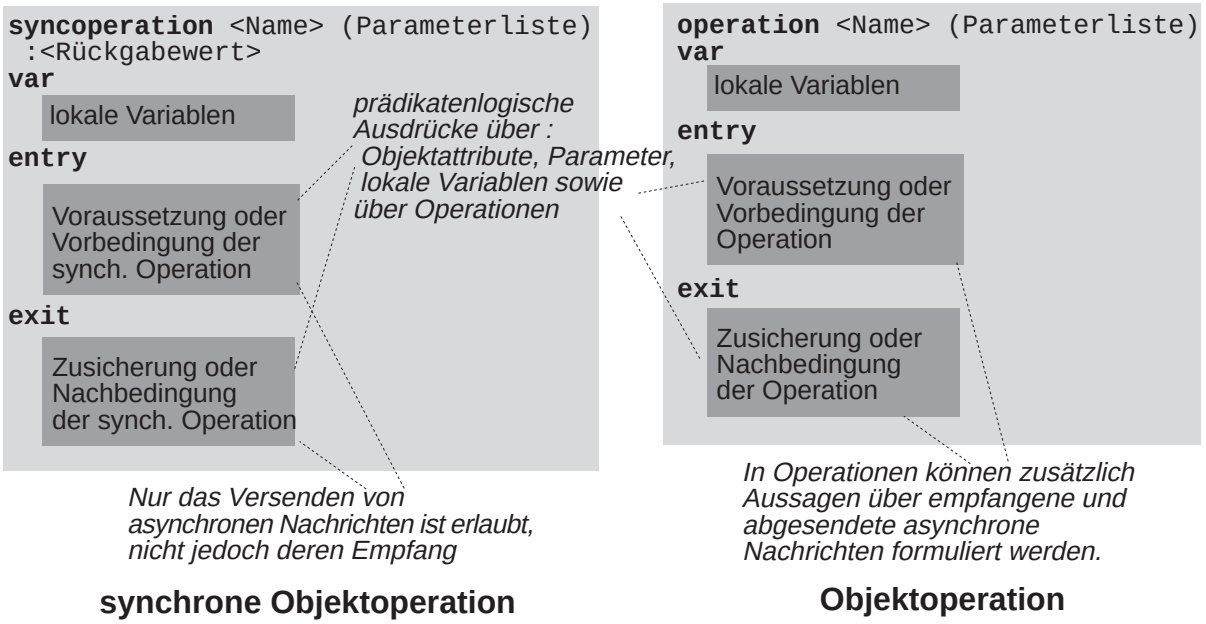


Abb. 45: Überblick über den Aufbau einer Spezifikation einer Objektoperation bzw. einer synchronen Objektoperation

Die *Voraussetzung* (Schlüsselwort entry) der Operation gibt die Vorbedingung an, die notwendig für deren Ausführung ist. Die *Zusicherung* (Schlüsselwort exit) gibt die Nachbedingung der Operation an, die nach Ausführung der Operation gültig ist. Sowohl die Voraussetzung als auch die Zusicherung sind formale prädikatenlogische Ausdrücke.

- Zur Spezifikation von *synchronen Objektoperationen* – Synchrone Objektoperationen sind zeitlos: Zwischen dem Zeitpunkt vor der Operationsausführung (also wenn die Voraussetzung gilt) und dem Zeitpunkt der Beendigung (wenn also die Zusicherung gültig ist) darf

logisch gesehen keine Zeit vergehen. *Voraussetzung* und *Zusicherung* dürfen daher nur zeitlose Aussagen modellieren. Dies sind im wesentlichen Aussagen über die Objektattribute und über die Nachrichtenparameter (der Nachricht, welche die Operation ausgelöst hat). Zusätzlich können weitere synchrone Objektoperationen integriert werden, da diese ebenfalls zeitlos durch das synchrone Versenden einer Nachricht Resultate liefert.

- Zur Spezifikation von *Objektoperationen* – Im Gegensatz zur synchronen Objektoperation ist die normale Objektoperation zeitbehaftet, d.h. logisch gesehen kann zwischen der Auslösung und der Beendigung Zeit vergehen. Diese Objektoperationen können somit nur durch asynchrone Nachrichten ausgelöst werden (andernfalls würde das Prinzip der zustandbasierten Modellierung und die Verwendung zeitloser Zustandsübergänge verletzt werden). Dadurch ist es aber explizit möglich, in Objektoperationen das Warten auf ein Nachricht zu modellieren. Neben den oben beschriebenen prädikatenlogischen Aussagen kann also auch das Versenden, das Empfangen asynchroner Nachrichten und das expliziten Warten modelliert werden.

Das folgende Beispiel zeigt die axiomatische Spezifikation der beiden Objektoperationen skalieren und zeichnen des Objektes Rechteck:

```
Object Rechteck

attribute
    Höhe, Breite: real;
…


syncoperation skalieren (s: real);
entry
    s>0
exit
    Höhe' = Höhe * s &
    Breite' = Breite * s

end skalieren;
```

```
syncoperation zeichnen (p: point);
var
    p1, p2, p3: point
entry

exit
    p1.x=p.x & p1.y=p.y+Breite &
    p2.x=p.x + Höhe & p2.y=p.y &
    p3.x=p.x + Höhe & p3.y=p.y+Breite &
    sent zeichneLinie (p ,p1) over v &
    sent zeichneLinie (p ,p2) over v &
    sent zeichneLinie (p1,p3) over v &
    sent zeichneLinie (p2,p3) over v
end skalieren;
```

Bemerkung: point ist im Typverzeichnis deklariert.
v ist eine Beziehung zum Objekt, welches zeichneLinie realisiert.

Die synchrone Operation zeichnen verwendet zur Realisierung eine synchrone Operation eines weiteren Objektes (gekennzeichnet durch v, der Beziehung zwischen Rechteck und diesem Objekt). Ein weiteres Beispiel einer synchronen Operation ist BestimmeRechnung (siehe Abbildung 36, Kapitel 7.4). Sie bestimmt den Rechnungsbetrag, der insgesamt für eine Kartenbestellung zu zahlen ist:

```
STyp is list of SpTyp;

SpTyp is structure of (Position: PTyp; Preis: real)  (PTyp hält Positionsdaten des Sitzplatzes
fest)


syncoperation BestimmeRechnung (Sitzplätze:STyp)
        var i: integer
```

```
        entry
        exit
            BestimmeRechnung = forAll i: integer ( sum (Sitzplätze[i].Preis ))
    end BestimmeRechnung;
```

Bemerkung: Die Metaoperation sum() ist eine aus dem Sprachumfang der Sprache ASTRAL entnommene Standardoperation.

Die folgende Metaoperation wird von der Objektmenge *Veranstaltungen* zur Verfügung gestellt (siehe Beispiel aus Abbildung 30). Durch sie ist es möglich, neue *Veranstaltungen* als Objektinstanzen hinzuzufügen.

```
create anlegen (Name, Beschreibung, Ort: string; Datum:real; Server: id);
var   instanz: id
entry
    forAll instanz (instanz is in idInstance(idObject(self)) &
        ( instanz.NameDerVeranstaltung ≠ Name | instanz.Datum ≠ Name |
        instanz.Austragungsort ≠ Ort ) )
exit
    NameDerVeranstaltung = Name & Veranstaltungen.Beschreibung = Beschreibung &
    Austragungsort= Ort & Veranstaltungstermin = Datum & BezugsServer = Server
end anlegen
```

In der Signatur werden Parameter spezifiziert, mit welchen die anzulegende Objektinstanz initialisiert wird. Durch den Parameter Klasse wird die Klasse der Objektinstanz angegeben. In der Voraussetzung (entry) wird spezifiziert, unter welchen Umständen eine neue Instanz der Objektmenge hinzugefügt werden kann. Im speziellen wird durch den forAll-Quantor abgeprüft, ob bereits eine Objektinstanz vorhanden ist, die in Namen, Ort und Datum mit der neu einzufügenden Instanz übereinstimmt. Ist dies nicht der Fall, wird eine neue Instanz erzeugt und die Attribute dieser Instanz wie in der Zusicherung angegeben spezifiziert. Im nächsten Beispiel wird durch spezializeTo für die Klasse Veranstaltung eine Mutationsmethode spezifiziert, um eine Veranstaltung in die Klasse Sonderveranstaltung zu transformieren (Sonderveranstaltung soll hier eine Unterklasse von Veranstaltung sein). Eine Sonderveranstaltung soll dieselben Attribute wie eine normale Veranstaltung besitzen und sich zusätzlich durch einen Stargast auszeichnen. Die aktiven Zustände sollen von der ursprünglichen Instanz übernommen werden.

```
spezializeTo Sonderveranstaltung (s: string )
entry
exit
    Name‘ = Name & Ort‘= Ort & Datum" = Datum & Server" = Server &
    Stargast = s &
```

```
        activeStates (self'') = activeStates(self)
    end.
```

Kapitel 7

ADORA-L Sprachdefinition

Eine Modellierungssprache ist im wesentlichen geprägt durch die zugrundeliegenden Konzepte und durch die Umsetzung dieser Konzepte in geeignete Sprachkonstrukte. Die beiden letzten Kapitel beschäftigt sich schwerpunktmäßig mit den Sprachkonzepten selber und mit einem groben Überblick über die Sprache ADORA-L. In folgendem Kapitel wird die im letzten Kapitel grob skizzierte Sprache im Detail definiert. Die hier vorgestellte Sprachdefinition beschreibt die einzelnen Sprachkonstrukte von ADORA-L im einzelnen und führt die entsprechenden Notationen auf. Zu Beginn werden die Beschreibungsmittel vorgestellt, mit deren Hilfe die Sprache selber definiert wird.

7.1 Aufbau und Form der ADORA-L-Sprachdefinition

Im folgenden werden der Aufbau der Sprachdefinition der Sprache ADORA-L erläutert sowie die für die Sprachdefinition verwendeten sprachlichen Mittel vorgestellt. Die Sprachdefinition umfaßt eine detaillierte Beschreibung aller Sprachkonstrukte. Die Adora-L-Sprachkonstrukte werden in fünf Gruppen eingeteilt, in Sprachkonstrukte der *Basisstruktur* (Kapitel 7.2), der *strukturellen Einblendung* des Objektmodells (Kapitel 7.3), der *verhaltensorientierten Einblendung* des Objektmodells (Kapitel 7.4), der *funktionalen Einblendung des Objektmodells* (Kapitel 7.5) und des *Typverzeichnisses* (Kapitel 7.6).

7.1.1 Beschreibungsstruktur für Sprachkonstrukte

Die einzelnen Sprachkonstrukte werden durch eine spezielle Beschreibungsstruktur beschrieben. Die Beschreibungsstruktur unterscheidet

- eine Klassifikation des Sprachkonstrukts – Klassifiziert, ob das Sprachkonstrukt *grafisch* oder *textuell* notiert ist und ob es *konkret* oder *abstrakt* ist. Ein *konkretes* Sprachkonstrukt wird direkt in der Modellierung eingesetzt und besitzt daher eine eigene Notation. Ein *abstraktes* Sprachkonstrukt wird verwendet, um andere Sprachkonstrukte in generalisierter

Form zusammenfassend zu beschreiben. Abstrakte Sprachkonstrukte werden zudem aus Plausibilitätsgründen verwendet, um relevante, aber nicht explizit zu modellierende Sachverhalte zu beschreiben (Beispiel: Das abstrakte Sprachkonstrukt *Objektinstanz*).

- die *Definition* – Definiert das Sprachkonstrukt in seiner Bedeutung. Teilweise wurden die hier aufgeführten Definitionen bereits in vorhergehenden Kapiteln eingeführt.
- *Bemerkungen* – An dieser Stelle wird das Sprachkonstrukt über die Definition hinausgehend näher beschrieben. Zudem werden relevante Zusammenhänge zu anderen Sprachkonstrukten erläutert.
- die *Notation* – An dieser Stelle wird festgelegt, wie das Sprachkonstrukt visualisiert wird.
- ein oder mehrere *Beispiele* – Hier werden Beispiele für die Notation des Sprachkonstruktes vorgestellt.

7.1.2 Metasprachen

Die Sprachdefinition von ADORA-L verwendet natürliche Sprache und EBNF als Metasprache.

- Die *Klassifikation*, die *Definition* und die *Bemerkungen* der Sprachkonstrukte werden ausnahmslos durch natürliche Sprache und Prosa-Text beschrieben.
- Die Notation wird ebenfalls durch natürliche Sprache beschrieben. Zusätzlich wird für textuelle Notationen die Syntaxsprache EBNF verwendet.

Die EBNF (Extended Backus Naur Form) ist eine Metasprache zur syntaktischen Beschreibung von textbasierten Sprachen. Für die Sprachdefinition wird die Notation nach [Wirth86] verwendet.

Anzumerken ist noch, daß ADORA-FSL, eine Teilsprache von ADORA-L, auf der formalen Sprache ASTRAL [Coen97] aufbaut. Ein Teil der Sprachkonstrukte wird daher nicht im Detail aufgeführt, sondern mit einem entsprechenden Verweis auf entsprechende Konstrukte von Astral definiert.

7.2 Objektmodell – Die Basisstruktur

Dieses Kapitel definiert die Sprachkonstrukte, welche zur Formulierung der ADORA-L-Basisstruktur notwendig sind. Im einzelnen sind dies die Sprachkonstrukte *Objektinstanz*, *Objekt*, *Objektmenge*, *externer Akteur/externe Akteurmenge* und *externes Objekt*. Da die Basisstruktur Grundlage für die aspektbezogenen Einblendungen ist, werden die hier vorgestellten Sprachkonstrukte in späteren Kapiteln aufgegriffen.

7.2.1 Objektinstanz

Abstraktes Sprachkonstrukt (ohne Notation) *siehe Seite 68ff*

Definition

Eine Objektinstanz (synonym: Instanz) ist eine im laufenden System konkret vorhandene und individuelle Einheit. Jede Objektinstanz ist einem abstrakten Objekt bzw. einer abstrakten Objektmenge zugeordnet. Sie enthält durch Attribute und Komponenten repräsentierte Information, deren Struktur im abstrakten Objekt (Objektmenge) definiert ist. Eine Objektinstanz kann die dort definierten Nachrichten empfangen, d.h. sie besitzt für jede definierte Nachricht eine entsprechende Leistungsbeschreibung und entsprechende Verhaltensweisen.

Beschreibung

Ein abstraktes Objekt ist die Definition einer Instanz, d.h. dessen abstrakte Beschreibung. Die Instanzen selber finden sich in ADORA-L in der Systemmodellierung nicht wieder. Das Verhalten einer Instanz wird beschrieben durch die möglichen Nachrichten, auf die sie reagieren kann. Weiterhin definiert das abstrakte Objekt alle möglichen Verhaltensweisen sowie die eingeschachtelten Instanzen als Komponenten der entsprechenden Instanz.

Notation

Instanzen haben in keine Notation, da sie allein zur Plausibilitätsklärung von Objekten dienen.

Beispiel

- Peter Müller, der durch das Objekt Person beschrieben wird.
- Die an ein Reisebuchungssystem gestellte Anfrage Nr. 13.493.8723 für die Pauschalreise Hawaii am 19.5.98, 15 Uhr, welche durch das Objekt Pauschalanfrage modelliert wird.

7.2.2 Objekt

Konkretes Sprachkonstrukt, grafisch notiert *siehe Seite 70f, 134f*

Definition

Ein abstraktes Objekt *(Synonym: Objekt)* ist Platzhalter für und Repräsentant von einer Instanz und definiert deren Attribute, deren Funktionalität und deren Verhalten auf abstrakter Ebene. Abstrahiert wird von der Individualität und der Konkretheit. Die genaue Identität des Objekts sowie die konkreten Attributwerte bleiben unbekannt, deren Eigenschaften, im speziellen Informationen über den speziellen Objektkontext, finden sich in der Objektdefinition wieder.

Neben Attribut-, Verhaltens- und Leistungsbeschreibungen beinhaltet ein Objekt einen Verweis auf dessen Objekttyp (die Klasse) und auf die gültigen *Stereotypen*.

Bemerkungen

Objekte sind das zentrale Modellierungsmittel in ADORA-L. Durch sie werden Struktur, Verhalten und Leistung eines Systemmodells konstruktiv modelliert. Sie stellen eine Abstraktion von konkret vorhandenen Entitäten oder Sachverhalten dar.

Die Modellierung von Objekten erfolgt im Objektmodell sowohl typbezogen als auch kontextbezogen. Die typbezogene Modellierung spezifiziert die Klasse dieses Objekts und ist damit für alle weiteren Objekte dieser Klasse (welche an anderer Stelle im Objektmodell aufgeführt sein können) gültig und bindend. Die kontextbezogene Modellierung beschreibt die Zusammenhänge zwischen dem Objekt und seinem im Objektmodell vorhandenen Kontext. Objekte werden beschrieben durch eine *Basisstruktur* sowie durch aspektbezogene Einblendungen in diese Basisstruktur (siehe Kapitel 5.2.2).

Notation

Objekte werden durch Rechtecke dargestellt, die den Namen des Objekts enthalten. Objekte können in *komprimierter* oder in *expandierter* Form dargestellt werden. In komprimierter Form wird allein der Objektname dargestellt, ergänzt durch das «…»-Symbol, falls es Objektkomponenten enthält. In expandierter Form sind neben dem Objektnamen (im Rechteck links oben aufgeführt) die Objektkomponenten als eingeschachtelte Rechtecke sichtbar. Je nach sichtbarer Einblendung werden zusätzlich die jeweiligen Aspekte eingeblendet. Ausnahme ist die funktionale Einblendung, die nicht überlappend, sondern separat annotiert wird.

Beispiel

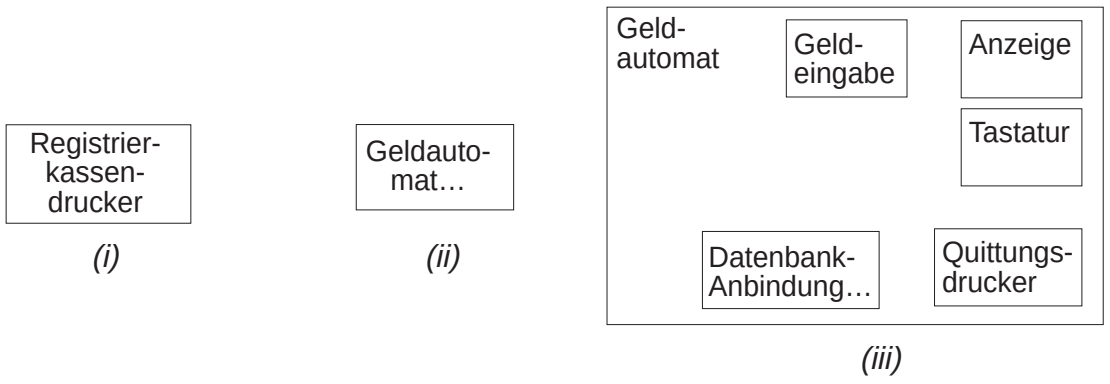


Abb. 46: Beispiele für Objekte in (i) und (ii). Das Beispiel (iii) ist die expandierte Darstellung von (ii)

Zu Abbildung 46, (i): Der Registrierkassendrucker stellt ein Objekt dar, welches in einem definierten Kontext verwendet werden kann. Seine Identität (beispielsweise eine Seriennummer etc.) ist unbekannt, nicht jedoch sein Kontext und seine Eigenschaften.

Zu (ii) und (iii): Hier wird das Objekt Geldautomat in komprimierter bzw. in expandierter Form dargestellt. Das Objekt besitzt insgesamt fünf Objekte als Komponenten, darunter das Objekt Quittungsdrucker.

Anmerkung: Die Objekte Registrierkassendrucker und Quittungsdrucker gehören derselben Klasse Belegdrucker an (nicht oben abgebildet), sind also in ihrem Typ identisch. Durch die Modellierung durch abstrakte Objekte ist es möglich, den unterschiedlichen Kontext dieser beiden Objekte zu modellieren (im Gegensatz zu klassenorientierten Ansätzen à la OOA, OOAD, UML etc.).

7.2.3 Objektmenge

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 70f, 109f*

Definition

Ein Objektmenge (synonym: abstrakte Objektmenge) ist eine Menge von Objektinstanzen derselben Klasse oder einer Unterklasse dieser Klasse. Die Instanzen der Objektmenge stehen alle im selben Kontext und werden durch eine gemeinsame Definition modelliert. Aus einer Objektmenge können Instanzen entfernt und in eine Objektmenge können neue Instanzen erzeugt und hinzugefügt werden. Eine Objektmenge besitzt also eine variable Größe.

Bemerkungen

Objektmengen setzen die Idee um, beliebig viele, im selben Kontext stehende gleichgeartete Objekte, (die in derselben Komposition aufgeführt sind) gemeinsam zu modellieren. Von besonderer Bedeutung sind Objektmengen, weil sie eine dynamische Erzeugung von neuen Objekten innerhalb des Systemmodells ermöglichen. Diese Eigenschaft ist häufig notwendig, um reale Systeme zu modellieren. Zudem lassen sich in Objektmengen nicht nur typgleiche Instanzen, sondern auch Spezialfälle davon einfügen (also Objekte polymorph zu verwenden).

Eine Objektmenge wird prinzipiell gleich modelliert normale abstrakte Objekte. Zusätzlich werden jedoch einige Eigenschaften zusätzlich erfaßt. So müssen beispielsweise Metaoperationen für die Erzeugung und die Löschung von Objekten definiert werden (siehe funktionale Einblendung). Zudem wird die Kardinalität der Objektmenge durch einen Minimalwert und einen Maximalwert eingeschränkt. Diese Kardinalität kann durch folgende Metaoperationen (siehe auch Kapitel 7.5.5) abgefragt werden:

- maxObjectNumber(id): integer – Liefert die maximale Größe einer Objektmenge zurück. Ist die Menge in ihrer Größe unbestimmt, wird der maximal mögliche Integer-Wert zurückgeliefert.
- minObjectNumber (id): integer – Liefert die minimale Größe einer Objektmenge zurück. Der Ergebniswert ist stets größer oder gleich Null.
- actObjectNumber (id): integer – Liefert die Größe der Objektmenge zum Zeitpunkt des Aufrufs der Operation zurück.

Wird die Operation auf ein normales Objekt angewendet, ist das Resultat stets 1 (vom Typ Integer).

Notation

Objektmengen werden durch einen Rechteckstapel dargestellt. An der rechten unteren Ecke werden die minimal und maximale Größe der Objektmenge geklammert beschrieben (Bsp: «(0, 1)», «(1, 10)», «(0, n)», «n» steht für unbeschränkte Objektmengengröße). Im übrigen gelten die syntaktischen Vereinbarungen der Objektnotation.

Beispiel

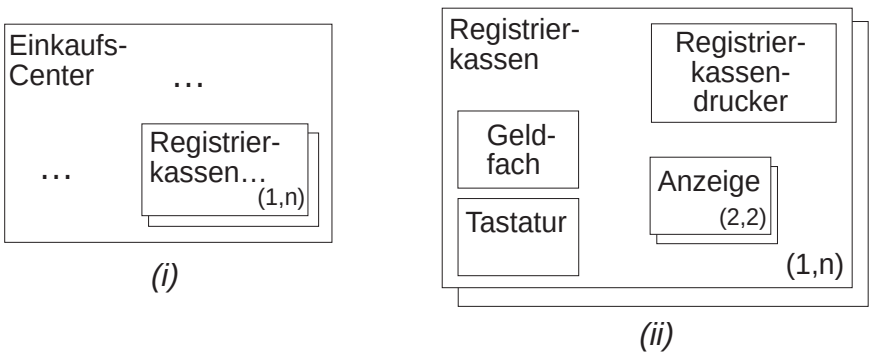


Abb. 47: Beispiele für Objektmengen in (i) und (ii)

Zu Abbildung 47, (i): Das Objekt Einkaufs-Center ist unter anderem definiert durch beliebig viele Registrierkassen, mindestens aber durch eine. Zu (ii): Die Objektbeschreibung einer Registrierkasse ist wiederum durch genau zwei Anzeigen gleichen Typs definiert. Durch das Teil/Ganzes-Paradigma kann eine konkrete Instanz des Objektes Anzeige nur in einer Instanz Registrierkasse enthalten sein. Insgesamt sind also zwischen *2* und *2·n* Anzeigen-Instanzen vorhanden.

7.2.4 Externer Akteur, externes Objekt

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 70f, 109f*

Definition

Ein *externer Akteur* ist ein eigenständiges externes System, eine Person, ein Sensor o. ä., welches mit dem Systemmodell bzw. mit den dortigen Objekten (bzw. Objektmengen) interagiert. Außer der statischen und der dynamischen Kommunikation werden keine weiteren Eigenschaften des externen Akteurs im Systemmodell erfaßt.

Eine *externe Akteurmenge* modelliert eine Menge von gleichgearteten externen Systemen. Analog zu einem einzelnen externen Akteur werden außer der statischen und dynamischen Kommunikation keine weiteren Eigenschaften erfaßt.

Ein *externes Objekt* ist ein System, welches mit dem Systemmodell bzw. mit den dortigen Objekten interagiert. Im Gegensatz zum *externen Akteur* ist es jedoch Bestandteil und damit Objektkomponente des Systemmodells. Es verhält sich daher wie ein gewöhnliches Objekt, dessen Interna nicht modelliert werden. Ein externes Objekt wird verwendet, um Drittkomponenten in das Systemmodell aufzunehmen und diese dort als Black-Box zu verwenden (zwar sind die Leistungen eines externen Objektes bekannt, nicht aber dessen Struktur und dessen Verhalten).

Eine *externe Objektmengen* ist analog dazu eine Menge gleichartiger externer Systeme, welche mit dem Systemmodell interagieren. Sie ist ebenfalls Bestandteil des Systemmodells. Sie verhält sich daher wie ein gewöhnliches Objekt, welches nur in vergröberter Form als Black-Box modelliert wird.

Bemerkungen

Externe Akteure/Akteurmengen und externe Objekte/Objektmengen werden verwendet, um den Kontext eines Systemmodells und die Kommunikation zwischen System und diesem Kontext zu beschreiben. Sie haben jedoch nicht den Status eines Objektes. Vielmehr sind sie als eine Art Quell- und Zieladresse von und für Nachrichten zu verstehen. Beziehungen modellieren entsprechend zwischen Akteuren (bzw. zwischen externen Akteuren) und Elementen des Systemmodells einen entsprechenden Nachrichtenfluß.

Sowohl externe Akteure als auch externe Objekte modellieren den Systemkontext. Ein externer Akteur wird dann verwendet, wenn ein eigenständiger, vom System klar getrennter Kontext modelliert werden soll. Externe Akteure stehen somit quasi als gleichberechtigte Partner «neben» dem modellierten System. Beispielsweise sind externe Akteure gut für die Darstellung menschlicher Systembenutzer geeignet. Externe Objekte hingegen werden verwendet, um

den Integrationsaspekt einer Drittkomponente in das zu modellierende System hervorzuheben. Ein externes Objekt hat somit prinzipiell alle Eigenschaften eines normalen Objekts (insbesondere sind sie in die Teil/Ganzes-Hierarchie eingeordnet), welches jedoch nur durch seine funktionale Schnittstelle spezifiziert ist, nicht durch seinen inneren Aufbau. Beispiel für externe Objekte finden sich in eingebetteten Systemen, in denen sie entweder eine bereits realisierte Lösung repräsentieren oder an anderer Stelle beschrieben werden.

Die Eigenschaften externer Akteure (Akteurmengen) werden nicht weiter spezifiziert. Der Leistungsumfang eines externen Objekts (Objektmenge) wird durch eine entsprechende Elementarbeschreibung notiert (siehe Kapitel 7.5.1).

Notation

- Externe Akteure werden dargestellt durch leicht verzerrte gleichmäßige Sechsecke, externe Akteurmengen entsprechend durch Sechseckstapel. Der Namen des externen Akteurs wird in das Sechseck gezeichnet.
- Externe Objekte/Objektmengen werden in derselben Notation wie Objekte bzw. Objektmengen notiert. Zusätzlich wird jedoch der Objektnamen durch «*:extern*» ergänzt. Die Elementarbeschreibung wird analog zur Elementarbeschreibung «normaler» Objekte (Objektmengen) notiert.

Beispiel

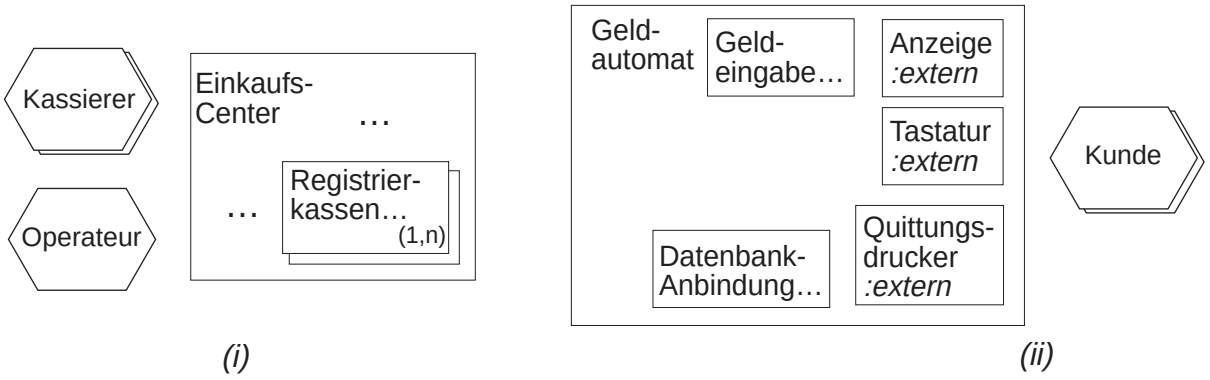


Abb. 48: Beispiel für die Modellierung eines Systemkontextes durch externe Akteure (i) und (ii) bzw. durch externe Systeme (ii)

Zu Abbildung 48, (i): Modelliert wird hier ein Operateur und beliebig viele Kassierer als externer Akteur bzw. als externe Akteurmenge.

Zu (ii): Das Systemmodell Geldautomat wird unter anderem beschrieben durch die drei Objektkomponenten Anzeige, Tastatur, und Quittungsdrucker, die ihrerseits externe Objekte sind (alle drei externen Objekte sind beispielsweise bereits durch Hardware realisiert und spezifiziert).

Anmerkung: Externe Akteure/Akteurmengen werden immer dann dargestellt, wenn sie mit Komponenten eines Systemmodells oder eines Teiles davon kommunizieren. Sie sind in diesem Fall also stets sichtbar. Externe Objekte (Objektmengen) sind jedoch dann verborgen, wenn deren Komposition komprimiert ist oder gar nicht sichtbar ist. In (i) beispielsweise sind externe Objekte des Objektes Registrierkasse in dieser Sicht nicht erkennbar.

7.3 Objektmodell – Die strukturorientierte Einblendung

In diesem Kapitel werden die Sprachkonstrukte der strukturellen Einblendung des Objektmodells definiert. Im einzelnen wird die *Beziehung*, die *strukturelle Beziehung*, und die *Benutzung* vorgestellt. Diese Sprachkonstrukte sind als Ergänzung und als Erweiterung der Konstrukte der Basisstruktur (siehe Kapitel 7.2) zu verstehen.

7.3.1 Beziehung

Abstraktes Sprachkonstrukt	*siehe Seite 84, 112*

Definition

Eine Beziehung beschreibt die gegenseitige Referenzierbarkeit der in Beziehung gesetzten Objekte (Objektmengen) und damit potentielle Informationsflüsse. Sie stellt eine Abstraktion einer Menge von Beziehungen auf Ausprägungsebene dar, welche zwischen den entsprechenden Objektinstanzen gültig sind. Ein Informationsfluß zwischen zwei Objekten (Objektmengen) repräsentiert ein oder mehrere Nachrichten, welche zwischen ihnen wechselseitig versendet und empfangen werden. Die Referenzierbarkeit macht eine Aussage darüber, ob ein Objekt (Objektmenge) die Existenz und die Identität eines anderen Objektes (Objektmenge) kennt und entsprechende Zugriffsrechte besitzt. Explizit ist es möglich, daß zwischen zwei Objekten (Objektmengen) mehrere Beziehungen existieren.

Bemerkung

Beziehungen werden verwendet, um die Kommunikation zwischen Objekten (bzw. Objektmengen) statisch zu modellieren. Eine Beziehung zwischen zwei Objekten/Objektmengen steht für einen oder für mehrere Nachrichtenflüsse zwischen den Objektinstanzen auf Ausprägungsebene:

- Beziehungen zwischen Objekten – Die Beziehung repräsentiert genau eine Beziehung zwischen den beiden Objektinstanzen, welche durch die beiden Objekte beschrieben werden.
- Beziehungen zwischen einem Objekt und einer Objektmenge – Die Beziehung repräsentiert eine Menge von Beziehungen zwischen einer Objektinstanz und den Objektinstanzen, die durch die Objektmenge beschrieben werden. Eine von der einzelnen Objektinstanz versendete Nachricht wird also über n Beziehungen an alle Objektinstanzen der Objektmenge geschickt (n beschreibt die Anzahl der Instanzen der Objektmenge).
- Beziehungen zwischen zwei Objektmengen – Die Beziehung repräsentiert ebenfalls eine Menge von Beziehungen. Die Beziehung repräsentiert jedoch nicht n, sondern $n \cdot m$ Beziehungen (n, m beschreiben die Anzahl der Instanzen der beiden Objektmengen).

Anmerkung: Bei optionalen Beziehungen (Beziehungen mit der Kardinalität (0, x)) kann, muß aber nicht jede Beziehung Beziehungen auf Instanzebene haben.

Beziehungen zwischen Objekten/Objektmengen sind Voraussetzung für die Versendung von Nachrichten. Die Modellierung der Versendung und des Empfangs von Nachrichten erfolgt explizit durch Nennung einer *Beziehung*. Beziehungen im hierarchischen Objektmodell (d.h. in der strukturellen Einblendung) werden hierarchisch modelliert. Tieferliegende Beziehungen werden durch höherliegende Beziehungen, die *Oberbeziehungen*, gebündelt und in vergröberter Form zusammengefaßt. Eine Oberbeziehung hat wiederum sämtliche Charakteristika einer Beziehung, faßt aber zusätzlich eine Reihe von tieferliegenden Beziehungen zusammen.

Notation & Beispiel

siehe Sprachkonstrukt «strukturelle Beziehung» (Seite 143) und «Benutzung» (Seite 145)

7.3.2 Strukturelle Beziehung

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 84, 112*

Definition

Eine *strukturelle Beziehung* ist eine Beziehung, welche einen strukturellen Zusammenhang zwischen zwei Objekten (Objektmengen) des Objektmodells beschreibt. Die strukturelle Beziehung drückt aus, daß zwei Objekte (Objektmengen) miteinander zusammenarbeiten und kooperieren, ohne auf die Art der Kommunikation näher einzugehen. Soll die Zusammenarbeit nur für eine der beiden Komponenten sichtbar sein, ist die Beziehung in diese Richtung *gerichtet*. Andernfalls ist sie für beide Komponenten sichtbar und *ungerichtet*.

Bemerkung

Ist eine strukturelle Beziehung ungerichtet, so ist für beide Komponenten (Objekte oder Objektmengen) die dadurch ausgedrückte Kooperation sichtbar (d.h. beide Komponenten haben Kenntnis von der gegenseitigen Identität). Hat also ein Objekt A eine strukturelle Beziehung zu Objekt B, so gilt dies auch in umgekehrter Richtung. Beide Richtungen werden jeweils durch einen Namen näher spezifiziert. Durch eine geeignet gewählte Namensgebung soll das Verhältnis von A zu B und von B zu A zum Ausdruck kommen. Ist die strukturelle Beziehung gerichtet, so ist nur für eine der beiden Komponenten die andere Komponente sichtbar. Entsprechend wird diese gerichtete Beziehung nur in eine Richtung durch einen Namen bezeichnet.

Strukturelle Beziehungen werden darüber hinaus durch Kardinalitäten näher spezifiziert. Die Kardinalitäten machen quantitative Aussagen über die Anzahl der Beziehungen, welche durch

die strukturelle Beziehung repräsentiert werden. Spezifiziert werden kann eine minimale und eine maximale Anzahl von Beziehungen von einer Objektinstanz des einen Objektes zu den Objektinstanzen der anderen Seite (und umgekehrt).

Notation

Strukturelle Beziehungen werden durch eine Linie zwischen den beteiligten Objekten/Objektmengen A und B dargestellt. Ist die strukturelle Beziehung eine Oberbeziehung, so wird sie fett, andernfalls durch eine normal dicke Linie notiert.

Die Bezeichnungen einer strukturelle Beziehung, welche die Verhältnisse zwischen A und B und umgekehrt näher beschreiben, werden durch einen textuellen Ausdruck in unmittelbarer Nähe der Linie dargestellt. Bei gerichteten strukturellen Beziehungen wird die Beziehung nur in eine Richtung bezeichnet (ist die Beziehung von A nach B gerichtet, so wird die Beziehung auch in Richtung von A nach B bezeichnet). Die Leserichtung der Bezeichnung wird durch einen vor die Bezeichnung vorangestellten Pfeil $\uparrow$, $\leftarrow$, $\downarrow$, $\rightarrow$ spezifiziert (falls werkzeugtechnisch auch schräge Linien erlaubt sind, sind auch entsprechend geneigte Pfeile möglich).

Die Kardinalitäten der Beziehung werden in Anschluß an die jeweilige Bezeichnung in Klammern hinzugefügt. In Richtung von Objekt/Objektmenge A nach B wird spezifiziert, zu wievielen Instanzen in B *eine* Instanz von A eine Beziehung besitzt. Für die Richtung von B nach A gilt entsprechendes. Die Kardinalitäten werden wie folgt notiert (in Richtung von A nach B):

- (0,1), (1,1); falls B ein Objekt ist – Erste Kardinalität drückt eine Optionalität aus (es kann von A nach B eine Beziehung existieren). Die zweite Kardinalität ist unbedingt: Zwischen A und B existiert stets eine Beziehung.
- (x,y), (x,n); falls B eine Objektmenge ist (y ist eine natürliche Zahl, x eine natürliche Zahl oder 0, stets gilt $x \leq y$; *n* ist ein feststehendes Sprachsymbol) – x und y geben die minimale bzw. maximale Anzahl an Beziehungen zwischen den Instanzen in B und einer Instanz in A an (zwischen einer Instanz in A und in B existiert maximal eine Beziehung). Das Sprachsymbol n gibt an, daß die Anzahl an Beziehungen nach oben nicht beschränkt ist.

Ist die *strukturelle Beziehung* eine Oberbeziehung, wird zwischen der Oberbeziehung und allen sichtbaren untergeordneten Beziehungen eine fein gestrichelte Linie gezeichnet.

Beispiel

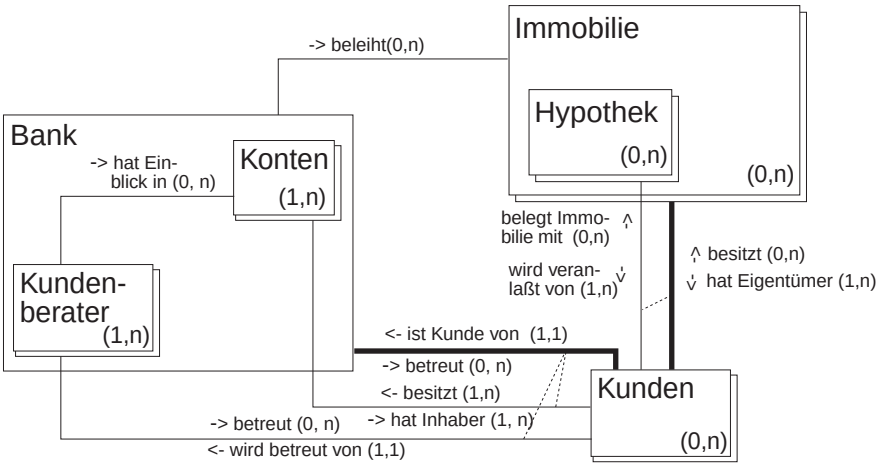


Abb. 49: Beispiel einer strukturellen Einblendung anhand eines Bank-Kunden-Immobilien-Verhältnisses

Das Modell in Abbildung 49 beschreibt die Zusammenhänge zwischen einer Bank, Kundenberatern, Konten sowie Hypotheken, Immobilien und Kunden. Innerhalb des Objektes Bank wird eine einfache strukturelle Beziehung verwendet. Die Beziehungen beleiht und hatEinblickIn sind gerichtet, da jeweils nur ein Objekt (eine Objektmenge) Kenntnis von diesem Zusammenhang hat. Alle anderen Beziehungen sind ungerichtet. Die Zusammenhänge zwischen Bank und Immobile sowie Bank und Kunden wird zweischichtig durch eine Oberbeziehung und durch untergeordnete strukturelle Beziehungen modelliert.

7.3.3 Benutzung

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 84f, 112f*

Definition

Eine Benutzung ist eine gerichtete Beziehung zwischen zwei Objekten/Objektmengen A und B, in der A eine von B bereitgestellte Leistung in Anspruch nimmt. Benutzungen sind stets gerichtete Beziehungen.

Bemerkungen

Die Benutzung macht eine gegenüber der strukturellen Beziehung präzisere Aussage über die Art der Kooperation, indem sie die beiden Rollen *Auftraggeber* und *Auftragnehmer* unterscheidet und dadurch eine Delegierung einer Aufgabe modelliert. Sie ist logisch gerichtet, d.h. sie drückt eine Zusammenhang von Objekt/Objektmenge A nach B aus, nicht jedoch in umgekehrter Richtung. Es ist jedoch explizit möglich, Nachrichten in beide Richtungen zu verschicken.

Notation

Eine Benutzung eines Objekts (bzw. einer Objektmenge) A und eines Objekts (bzw. einer Objektmenge) B wird durch eine bepfeilte Linie von A nach B dargestellt. Ist die Benutzung eine Oberbeziehung, so wird sie fett, andernfalls durch eine normal dicke Linie notiert.

Eine Benutzung wird durch eine Bezeichnung näher beschrieben, welche die Art der Benutzung von Auftraggeber zu Auftragnehmer hin beschreibt. Die Kardinalitäten der Benutzung werden analog zur strukturellen Beziehung in Anschluß an die Bezeichnung in Klammern hinzugefügt (siehe Seite 143). Die Benutzung wird jedoch nur durch eine Kardinalität näher spezifiziert, nämlich durch diejenige von Auftraggeber zu Auftragnehmer. Ist die Benutzung eine Oberbeziehung, wird zwischen der Oberbeziehung und allen sichtbaren untergeordneten Beziehungen eine fein gestrichelte Linie gezeichnet.

Beispiel

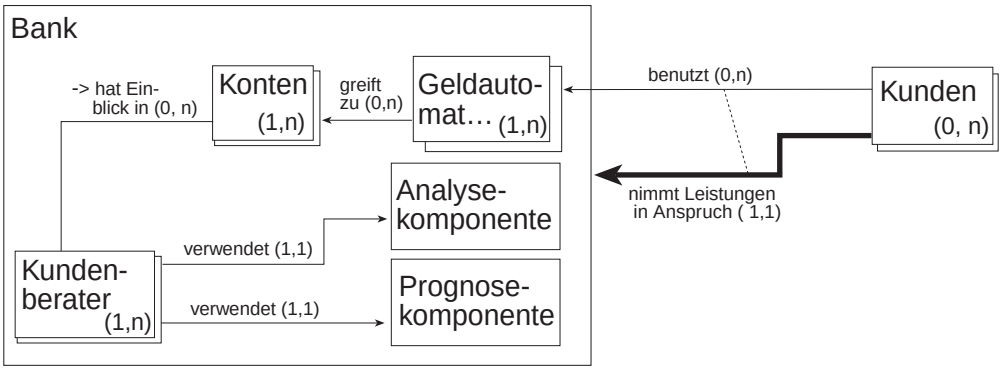


Abb. 50: Beispiel aus Abbildung 49 mit zusätzlichen Benutzungen

Das Modell in Abbildung 50 beschreibt die Zusammenhänge der Komponenten des Objektes Bank und der Objektmenge Kunden. Die Beziehung zwischen Kunde und Bank wird zweischichtig durch eine Oberbeziehungs-Benutzung und einer detaillierten Benutzungsbeziehung beschrieben.

7.4 Objektmodell – Die verhaltensorientierte Einblendung

In diesem Kapitel werden die Sprachkonstrukte der verhaltensorientierten Einblendung des Objektmodells definiert. Im einzelnen beschrieben werden die Sprachkonstrukte *Zustand, elementarer Zustand, komplexer Zustand, Komponentenzustand, Zustandsübergang, Übergangsbedingung* und *Übergangsaktion*.

7.4.1 Zustand

Abstraktes Sprachkonstrukt	*siehe Seite 92f, 116f*

Definition

Ein Zustand beschreibt einen genau abgegrenzten Zeitabschnitt, in dem ein Objekt (eine Objektmenge) ein bestimmtes Verhalten zeigt und auf eine bestimmte Menge von Nachrichten reagiert (in Anlehnung an die Definition von [Glinz98b]).

Bemerkungen

Zustände werden verwendet, um wichtige und bedeutende Zeitabschnitte explizit zu modellieren und dadurch hervorzuheben. Die tatsächlichen Zustände eines Objekts (einer Objektmenge) werden im einzelnen nicht modelliert. Diese ergeben sich implizit durch die Permutation aller möglichen Wertbelegungen des Objekts (der Objektmenge). Die Gesamtheit aller Zustände eines Objektes (einer Objektmenge) bildet somit eine Abstraktion aller möglichen Wertebelegungen, welche angenommen werden können. Veränderungen des Zustandes eines Objektes werden durch *Zustandsübergänge* modelliert (siehe Seite 151). Zustände können in andere Zustände (desselben Objekts) eingeschachtelt sein. Durch diese Einschachtelung wird der übergeordnete Zustand durch weitere Zustände präziser und detaillierter beschrieben.

Zwei besondere Zustandstypen sind *Start-* und *Endzustände*. Startzustände sind diejenigen Zustände, welche zu Beginn, d.h. zum Zeitpunkt der Initialisierung des Objektes gültig sind. Endzustände sind Zustände, die einen Abschluß der Verhaltensbeschreibung markieren. Eine Verhaltensbeschreibung, welche sich nicht in einem Endzustand befindet, kann nicht durch einen *bedingten Zustandsübergang* verlassen werden (siehe Seite 151).

In ADORA-L werden drei Arten von Zuständen unterschieden, die *elementaren Zustände* (siehe Kapitel 7.4.2), die *komplexen Zustände (*siehe Kapitel 7.4.3) und die *Komponentenzustände* (siehe Kapitel 7.4.4).

Notation & Beispiel

Siehe Kapitel „Komponentenzustand“ (S. 150), „Elementarer Zustand“ (S. 148) und „Komplexer Zustand“ (S. 149).

7.4.2 Elementarer Zustand

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 92f, 116f*

Definition

Elementare Zustände sind Zustände, welche direkt einen eigenständigen Objektzustand identifizieren. Sie enthalten weder eingeschachtelte Zustände, noch haben sie die Bedeutung eines eigenständigen Objekts. Sie werden verwendet, um einen Zeitabschnitt im Lebenslauf eines Objektes direkt und ohne eine zusätzliche eingeschachtelte Verhaltensbeschreibung näher zu spezifizieren.

Bemerkungen

Ein elementarer Zustand beschreibt einen einfachen Objektzustand, der keine weitere eingeschachtelte Verhaltensbeschreibung benötigt.

Notation

Ein elementarer Zustand wird durch eine waagrechte Ellipse notiert. In der Ellipse wird die Bezeichnung des Zustandes dargestellt.

Beispiel

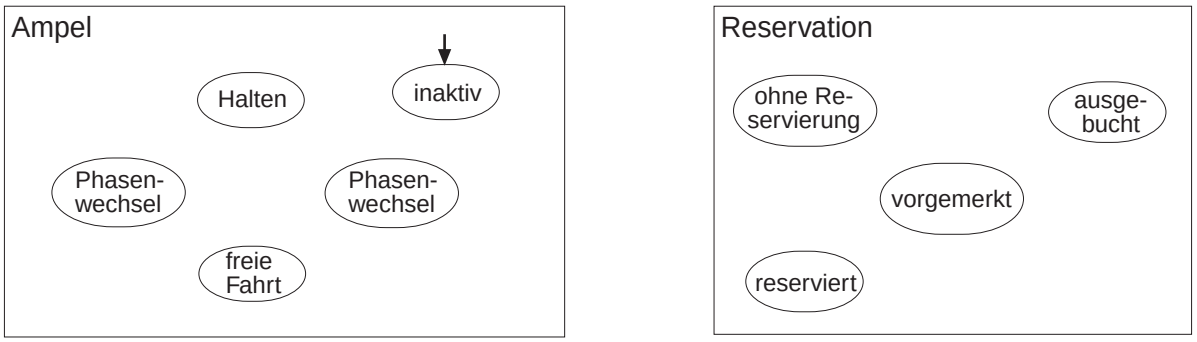


Abb. 51: Beispiele elementarer Zustände für die Objekte Ampel und Reservation

Zu Abbildung 51, (i): Das Objekt Ampel befindet sich in einem der Zustände Halten, freie Fahrt und Phasenwechsel (jeweils einmal für den Übergang stoppen-fahren und fahren/stoppen) oder ist inaktiv. Zu (ii): Eine Reservation (unbestimmter Art) ist in einem der vier abgebildeten Zustände.

7.4.3 Komplexer Zustand

Konkretes Sprachkonstrukt, grafisch notiert *siehe Seite 92f, 116f*

Definition

Ein komplexer Zustand ist ein Zustand, der ähnlich den elementaren Zuständen direkt einen konkreten Objektzustand beschreibt. Im Gegensatz zu elementaren Zuständen werden komplexe Zustände durch eingeschachtelte Zustände und durch entsprechende Verhaltensbeschreibungen näher spezifiziert.

Bemerkungen

Komplexe Zustände sind Zustände, die umfangreiche logisch zusammengehörige Zeitabschnitte mit Hilfe von eingeschachtelten Verhaltensbeschreibungen näher spezifizieren. Komplexe Zustände haben jedoch nicht den Charakter eines eigenständigen Objektes, d.h. sie können weder lokale Daten verwalten noch haben sie eine definierte und abrufbare Funktionalität. Sie sind ebenso wie die elementaren Zustände durch eine im Objekt eindeutigen Bezeichnung spezifiziert.

Notation

Analog zur Darstellung von Objekten kann ein komplexer Zustand in *expandierter* und in *komprimierter* Form notiert werden. Komprimiert wird er durch ein abgerundetes Rechteck notiert, in dem innen die Zustandsbeschreibung geschrieben wird, gefolgt von drei Punkten («…»). Die eingeschachtelten Zustände werden hier nicht dargestellt. In expandierter Form werden die eingeschachtelten Zustände innerhalb eines entsprechend vergrößerten Rechtecks (ebenfalls mit abgerundeten Ecken versehen) dargestellt. Die Bezeichnung des komplexen Zustands innerhalb des Rechtecks links oben geschrieben (ohne durch «…» ergänzt zu werden).

Beispiel

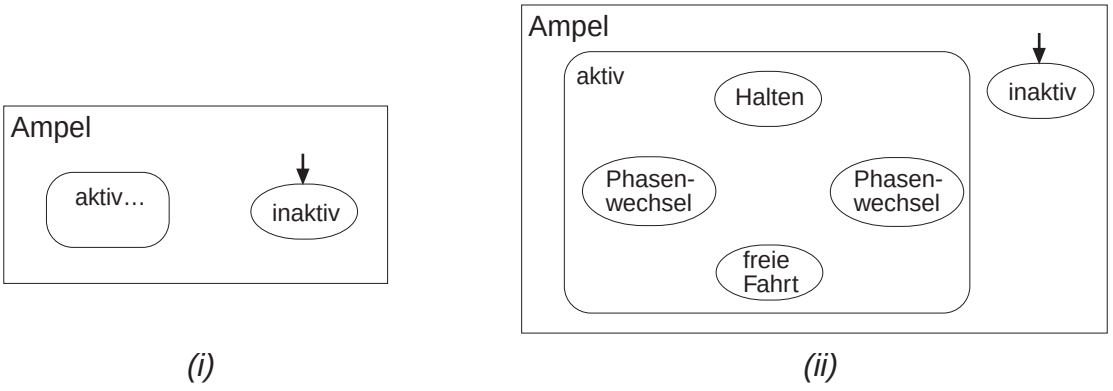


Abb. 52: Das Ampel-Beispiel aus Abbildung 51 mit komplexen Zuständen

Zu Abbildung 52: Der komplexe Zustand aktiv der Ampel wird in (i) komprimiert, in (ii) expandiert durch vier Subzustände modelliert.

7.4.4 Komponentenzustand

Konkretes Sprachkonstrukt, grafisch notiert *siehe Seite 92f, 116f*

Definition

Ein Komponentenzustand ist ein Objekt oder eine Objektmenge in der Rolle eines Zustandes. Der Zeitabschnitt des Komponentenzustandes entspricht der Gültigkeitsdauer des Objekts (der Objektmenge). Komponenten des Komponentenzustandes werden ihrerseits wieder als Komponentenzustände interpretiert.

Bemerkungen

Die Verhaltensbeschreibung ist in die gesamte Teil/Ganzes-Objekthierarchie integriert. Komponenten von Objekten (Objektmengen) müssen also entsprechend auch in der Verhaltensbeschreibung als Zustände aufgeführt werden. Die Komponentenzustände können zusätzlich durch weitere Zustandsarten (durch elementare oder komplexe Zustände) ergänzt werden, die Hierarchie selber ist jedoch durch die Objekthierarchie vorgegeben.

Ein nicht aktiver Komponentenzustand entspricht einem Objekt (einer Objektmenge), das zum momentanen Zeitpunkt nicht existiert und dessen Lebensdauer erst bei Erreichen des entsprechenden Komponentenzustandes beginnt. Das Objekt (die Objektmenge) selber wird zu diesem Zeitpunkt neu initialisiert. Wird der Komponentenzustand verlassen, so endet auch die Lebensdauer des Objekts (der Objektmenge) und damit auch sämtliche Wertebelegungen. Ein Objekt (eine Objektmenge) kann nur Nachrichten entgegennehmen, wenn es (sie) als Komponentenzustand aktiv ist, andernfalls verfällt die Nachricht.

Eine Objektmenge als Komponentenzustand wird entsprechend interpretiert als eine Menge nebenläufiger Zustände, mit einem Zustand pro Instanz der Objektmenge.

Notation & Beispiel

Komponentenzustände werden in der verhaltensorientierten Einblendung in der Notation von Objekten bzw. Objektzuständen notiert (siehe Kapitel 7.2.2 u. 7.2.3).

7.4.5 Zustandsübergang

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 92f, 116f*

Definition

Ein Zustandsübergang ist ein Zeitpunkt, an dem ein Objekt (eine Objektmenge) einen gegebenen Zustand verläßt und in einen neuen Zustand übergeht (in Anlehnung an die Definition in [Glinz98b]). Ein Zustandsübergang wird spezifiziert durch

- den *Quellzustand*, der aufgrund des Zustandsüberganges verlassen werden soll
- den *Zielzustand*, der aufgrund des Zustandes erreicht werden soll
- die Übergangsbedingung, eine logische Aussage über den Zeitpunkt des Überganges (siehe Kapitel 7.4.6)
- die Übergangsaktion, die Wirkung des Überganges auf die Umgebung (siehe Kapitel 7.4.7)

Unterschieden werden *bedingte* und die *absolute* Zustandsübergänge:

- *Bedingte Zustandsübergänge* sind nur dann möglich, wenn der Quellzustand entweder ein elementarer Zustand ist oder sich alle nebenläufigen Verhaltensbeschreibungen des Quellzustandes sich in einem Endzustand befinden.
- *Absolute Zustandsübergänge* führen stets zu einem Übergang, unabhängig, ob eingeschachtelte Zustände des Quellzustandes Endzustände sind oder nicht.

In beiden Fällen kann ein Zustandsübergang nur dann erfolgen, wenn die Übergangsbedingung erfüllt ist.

Bemerkungen

Zustandsübergänge werden verwendet, um den Zustand eines Objekts (einer Objektmenge) zu verändern. Durch die Spezifikation einer Übergangsbedingung und einer Übergangsaktion (siehe Kapitel 7.4.6 und 7.4.7) werden die Verhaltensweisen des Objektes diskret modelliert.

Im Gegensatz zu den Statecharts [Harel87] wird die Nebenläufigkeit in Adora-L nicht explizit modelliert (etwa wie bei [Harel87] durch Trennung mit Hilfe von gestrichelten Linien). Zustände, die innerhalb einer abgeschlossenen Verhaltensbeschreibung liegen (die also in einem Objekt, einem Komponentenzustand oder einem komplexen Zustand enthalten sind), sind zueinander nebenläufig, wenn kein Zustandsübergang zwischen ihnen vorhanden ist. Andernfalls sind sie gekoppelt und miteinander verbunden. Begründet liegt dies in der Annahme, daß die Nebenläufigkeit von Objekten (Objektmengen) die Regel ist und die Modelle durch eine zusätzliche Hervorhebung überladen würden.

Die Unterscheidung zwischen *bedingten* und *absoluten* Zustandsübergängen ist notwendig, um zwischen einem «regulärem» Verlassen und dem Verlassen im Sinne einer Unterbrechung zu unterscheiden. Die standardmäßig in ADORA-L verwendete Zustandsübergang ist bedingt. Soll ein absoluter Zustandsübergang modelliert werden, ist dieser entsprechend zu kennzeichnen (siehe Notation).

Notation

Ein Zustandsübergang wird durch eine fein gestrichelte Linie notiert, welche in Richtung vom Quell- zum Zielzustand bepfeilt ist. Die Strichelung wird verwendet, um einen Zustandsübergang von einer in der strukturellen Einblendung vorhandenen Benutzungsbeziehung notationell zu unterscheiden. Ein absoluter Zustandsübergang wird durch ein *Blitzsymbol* am Anfang des Übergangpfeiles notiert.

Die Übergangsbedingung wird neben den gestrichelten Pfeil textuell dargestellt. Die Übergangsaktion wird unterhalb des Textes der Übergangbedingung ebenfalls textuell notiert, getrennt von ihr durch eine waagrechte Linie.

Beispiel

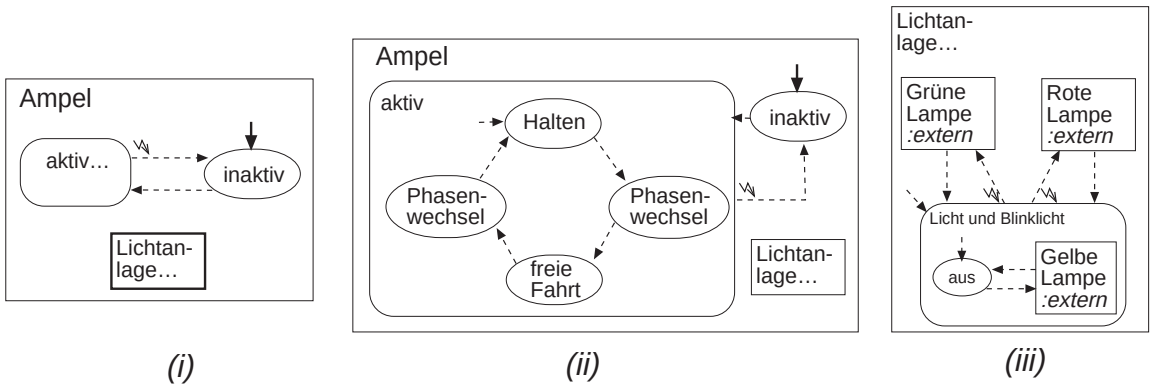


Abb. 53: Drei Beispiele von Zustandsübergängen für das Objekt Ampel und der Komponente Lichtanlage

In Abbildung 53 werden die Zustände und Zustandsübergänge einer Ampel modelliert.

Zu (i) und (ii): Die Ampel wird im wesentlichen durch zwei Zustandsübergänge spezifiziert, einen bedingten und einen absoluten Übergang. Der komplexe Zustand wird unabhängig von seinem internen Zustand verlassen.

Zu (iii): Zeigt die Zustände der Objektkomponente «Lichtanlage». Diese ist in diesem Fall zur restlichen Verhaltensbeschreibung nebenläufig, da kein Übergang von oder zu ihr in (ii) erfolgt.

7.4.6 Übergangsbedingung

Konkretes Sprachkonstrukt, textuell notiert *siehe Seite 92f, 116f*

Definition

Die *Übergangsbedingung* eines Zustandsübergangs gibt an, unter welchen Umständen ein Zustandsübergang erfolgt. Unter der Voraussetzung, daß der Ursprungszustand aktiv ist und die Übergangsbedingung zutrifft, erfolgt ein (zeitloser) Zustandsübergang zum Zielzustand. Eine Zustandsbedingung macht Aussagen über eingehende Nachrichten und deren Nachrichtenparameter und über Objektattribute.

Bemerkungen

Eine Übergangsbedingung besteht aus zwei optionalen Teilbeschreibungen, aus einem *Nachrichten-* und einem *Attributteil*. Der *Nachrichtenteil* gibt an, welche Nachricht den Zustandsübergang auslöst. der Der Attributteil ist ein logischer Ausdruck über die Objektattribute und die Nachrichtenparameter (falls im ersten Teil eine Nachricht spezifiziert wurde). Ein Übergang erfolgt nur dann, wenn die im Nachrichtenteil angegebene Nachricht an das Objekt (die Objektmenge) gesendet wurde, das Objekt sich im richtigen Zustand befindet und die logische Aussage im Attributteil erfüllt ist. Werden beide Teile nicht spezifiziert, so liegt ein ε-Übergang vor, der stets erfüllt ist. Sind zwei von einem Quellzustand abgehende Zustandsübergänge erfüllt, liegt ein Nichtdeterminismus vor. Solche Nichtdeterminismen müssen durch geeignete *Prioritätsmechanismen* aufgelöst werden (ein Mechanismus ist beispielsweise die Auswahl eines Zustandsüberganges nach dem Zufallsprinzip).

- Zum *Nachrichtenteil* der Übergangsbedingung – Hier wird eine Nachricht spezifiziert, die synchron oder asynchron an das Objekt (die Objektmenge) versendet wird. Versender dieser Nachricht ist entweder ein anderes Objekt (eine andere Objektmenge) des Objektmodells oder eine Übergangsaktion der Objektverhaltensbeschreibung selber. Ein Sonderfall ist der *asynchrone Multicast*, bei dem nicht nur eine, sondern mehrere Nachrichten eines Zustandsüberganges spezifiziert werden (siehe unten).
- Zum *Attributteil* der Übergangsbedingung – Spezifiziert neben der auslösenden Nachricht zusätzliche Bedingungen, die zur Auslösung des Zustandsüberganges zu erfüllen sind. Grundlage für die Formulierung der Bedingungen sind die Attribute des Objektes, die Nachrichtenparameter (falls im Nachrichtenteil eine Nachricht spezifiziert wurde) sowie Konstanten und zeitbezogene Aussagen. Zeitbezogene Aussagen nehmen Bezug auf relative oder absolute Zeitberechnungen (siehe auch Datentyp Time, Seite 173).

Der *asynchrone Multicast* ist eine spezielle Form der Nachrichtenkommunikation. Er wird verwendet, um eine asynchrone Nachricht an eine Objektmenge zu verschicken und anschließend die Reaktionen (in Form von Nachrichten) entgegenzunehmen. Die Antworten der Instanzen

der Objektmenge werden durch den *Multicast* gesammelt und können durch eine speziellen Nachrichtenteil (den *EmpfangMulticast*, siehe Notation) abgefragt werden. Der Nachrichtenteil ist folglich erst dann erfüllt, wenn alle Instanzen der Objektmenge geantwortet haben. Das spezielle Kommunikationskonzept wurde deshalb ADORA-L hinzugefügt, weil nur diejenigen Instanzen der Objektmenge bei der Antwort zu berücksichtigen sind, welche *zum Zeitpunkt des Erhalts der ursprünglichen Nachricht* in der Objektmenge enthalten sind (später hinzukommende Instanzen werden bzgl. Antwortnachricht jedoch nicht berücksichtigt).

Notation

Die Übergangsbedingung kann auf unterschiedlichen Formalitätsstufen spezifiziert werden (siehe Kapitel 5.4.1).

- Informal durch natürliche Sprache (in Anführungszeichen gesetzt)
- Formal durch ADORA-FSL
- Teilformal durch natürliche Sprache und ADORA-FSL – Schlüsselwörter der Sprache Adora-FSL anstatt von formal spezifizierten Ausdrücken durch Prosa-Text ergänzt.

Im folgenden wird die ADORA-FSL-Syntax für Übergangsbedingungen in EBNF vorgestellt:

Übergangsbedingung	=	(Nachrichtenteil "|" Attributteil | Nachrichtenteil | Attributteil) .
Nachrichtenteil	=	Nachrichtenempfang | EmpfangMulticast .
Nachrichtenempfang	=	"**receive**" Nachrichtenkopf .
EmpfangMulticast	=	"**receiveAll**" Nachrichtenkopf { "," Nachrichtenkopf } "**from**" Multicastname .
Nachrichtenkopf	=	Nachrichtenname "**(**" Parameterliste "**)**" .
Parameterliste	=	In-Liste | Out-Liste | In-Liste "**;**" Out-Liste | ε
In-Liste	=	"**in:**" *Liste von Parametern*
Out-Liste	=	"**out:**" *Liste von Parametern*

<Nachrichtenname> wird ausführlich in den Übersichtskapiteln beschrieben (siehe Kapitel 6.7.4). <Multicastname> erlaubt es, auf den Auslöser des asynchronen *Multicasts* Bezug zu nehmen (siehe Kapitel 7.4.7).

Attributteil	=	Ausdruck .	
Ausdruck	=	QuantifizierterAusdruck | BinärerAusdruck | UnärerAusdruck | AtomarerAusdruck .	
QuantifizierterAusdruck	=	("**forAll**" | "**exists**") Variable "**:**" Datentyp "**(**" Ausdruck "**)**" .	
BinärerAusdruck	=	Ausdruck ("**&**" | "**->**" | "**<->**" | "**|**") Ausdruck .	
UnärerAusdruck	=	"**~**" Ausdruck .	/ *Negation*
AtomarerAusdruck	=	Term RelOp Term .	
RelOp	=	"**=**" | "**<**" | "**>**" | "≤" | "≥" | "≠" .	

<Term> ist hierbei ein Ausdruck über Variablen. Variablen sind hier Objektattribute, Nachrichtenparameter und Ergebniswerte von Objektoperationen. Die Syntax für Terme wird von der Sprache ASTRAL übernommen und wird hier nicht explizit aufgeführt (siehe hierzu [Coen97]). <Variable> ist eine lokale Variable, welche ausschließlich für die Formulierung von Quantoren verwendet wird.

Beispiel

Siehe Beispiel in Kapitel 7.4.7

7.4.7 Übergangsaktion

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 92f, 116f*

Definition

Die Übergangsaktionen eines Zustandsüberganges gibt an, welche Tätigkeiten im Falle eines Zustandsüberganges ausgeführt werden. Eine Übergangsaktion definiert die Versendung einer Nachricht, das Löschen oder das Erzeugen von Objektinstanzen (zu einer Objektmenge) oder eine Manipulation eines Objektattributs. Die Ausführung der Übergangsaktionen muß logisch gesehen zeitlos erfolgen, da sonst Widersprüche zum Paradigma der Zustandsautomaten auftreten.

Bemerkungen

Die Übergangsaktionen spezifizieren die Wirkung eines Zustandsüberganges. Ein Zustandsübergang und damit auch die damit verknüpften Aktionen müssen zeitlos sein. Dies wirkt sich insbesondere auf die Verwendung von asynchronen Nachrichten aus. Diese können zwar innerhalb eines Zustandsüberganges ausgelöst werden, potentielle Antworten (also eingehende Nachrichten) sind innerhalb dieser Übergangsaktionen nicht verwendbar und müssen in späteren Zustandsübergängen verwertet werden. Die Übergangsaktionen eines Zustandsüberganges werden in zwei Teile unterteilt: In einen *Manipulationsteil* und in einen *Versendungsteil*.

- Der *Manipulationsteil* verändert die bestehenden Objektattribute und ist darüber hinaus in der Lage, synchrone Operationen auszulösen und deren Resultate direkt zu benutzen.
- Der *Versendungsteil* beschäftigt sich ausschließlich damit, *asynchrone* Nachrichten zu versenden. Die Versendung erfolgt logisch im Anschluß an den Manipulationsteil, d.h. eventuelle Berechnungen im Manipulationsteil können zur Ermittlung der Nachrichtenparameter des Versendungsteil mitverwendet werden. Nachrichten werden prinzipiell über eine spezifizierte Beziehung zum Zielobjekt geschickt. Ausnahme hierbei sind Zielobjekte, welche gleichzeitig Objektkomponenten sind. Diese werden durch direkte Nennung spezifiziert.

Von besonderer Bedeutung im Versendungsteil ist wiederum der *asynchrone Multicast* (siehe Kapitel 7.4.6). Er wird ausgelöst durch eine normale Versendung einer asynchronen Nachricht (an eine Objektmenge). Zusätzlich erhält der *Multicast* eine Bezeichnung, auf welche in einer nachfolgenden Übergangsbedingung Bezug genommen wird (siehe Seite 153).

Notation

Analog zur Übergangsbedingung können auch Übergangsaktionen auf den Formalitätsstufen informal (natürliche Sprache), teilformal (ADORA-FSL-Schlüsselwörter + Prosa) und formal (ADORA-FSL) spezifiziert werden. Die formale Spezifikation ist folgendermaßen notiert:

Übergangsaktion = (Manipulationsteil "**|**" Versendungsteil | Manipulationsteil | Versendungsteil) .

Manipulationsteil = Zuweisung { "**&**" Zuweisung } .

Zuweisung = EinfacheZuweisung | QuantifizierteZuweisung .

EinfacheZuweisung = NeuerAttributwert "**=**" Term | Term "**=**" NeuerAttributwert

NeuerAttributwert = Objektattribut "'" .

Versendungsteil = AsynchroneVersendung { "**;**" AsynchroneVersendung } .

AsynchroneVersendung = "**sent**" ("**multicast**"| ε) Nachrichtenname Parameterliste ("**over**" Beziehungsname) Beziehungsname | "**to**" Komponentenobjekt .

Parameterliste siehe Kapitel 7.4.6

Die Syntax von <Nachrichtenname> wurde in Kapitel 6.7.4 erläutert. <Beziehungsname> identifiziert eine existierende Beziehung zwischen dem Objekt (der Objektmenge) selber und dem Zielobjekt (Objektmenge), an welche die Nachricht verschickt werden soll.

Durch den Apostroph «'» wird der Attributwert *nach* dem Zustandsübergang spezifiziert. Attribute ohne Semikolon stellen entsprechend die Wertebelegungen *vor* dem Übergang dar.

Beispiel

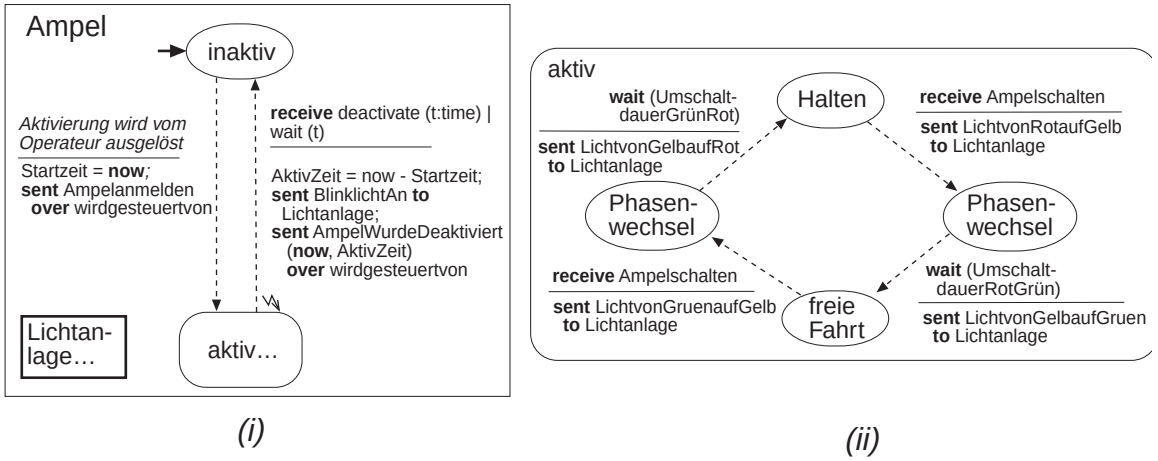


Abb. 54: Beispiel für Übergangsaktionen anhand des Ampelbeispiels (Abbildung 51)

Abbildung 54 zeigt die Verhaltensbeschreibung einer Ampel.

Zu (i): Bis auf die informal formulierte Übergangsbedingung vom Zustand inaktiv zu aktiv werden alle Bedingungen und Aktionen formal beschrieben. Startzeit und Aktivzeit sind Objektattribute des Ampelobjektes. Wird die Ampel durch die Nachricht deactivate deaktiviert, erfolgt nach Ablauf der in *t* definierten Zeit der Übergang zum Zustand inaktiv. Die aktive Betriebszeit wird beim Übergang dem Objekt Ampelsteuerung übermittelt (welche hier nicht aufgeführt ist und mit dem Objekt Ampel durch die Beziehung wirdgesteuertvon in Verbindung steht). Zusätzlich wird bei der Deaktivierung das gelbe Blinklicht aktiviert.

Zu (ii): Hier wird der komplexe Zustand aktiv näher beschrieben. Bedingungen und Aktionen sind ebenfalls formal spezifiziert. Der Phasenwechsel der Ampel wird durch die Nachricht Ampelschalten umgeschaltet (nicht ersichtlich ist der/die Versender dieser Nachricht. In diesem Fall ist es das Objekt Ampelsteuerung). Der Phasenwechsel erfolgt dann nach der entsprechenden Umschaltdauer. Bei jedem Übergang erfolgt eine entsprechende Umschaltung der Lichtanlage. Da das Objekt Lichtanlage Komponente des Ampelobjektes ist und daher keine explizit modellierte Beziehung zu ihr existiert, wird sie in der Übergangsaktion direkt spezifiziert.

7.5 Objektmodell – Die funktionale Einblendung

In diesem Kapitel werden die Sprachkonstrukte der funktionalen Einblendung des Objektmodells eingeführt. Die funktionale Einblendung ist in ADORA-L im Gegensatz zu den anderen aspektbezogenen Einblendungen ausschließlich textuell notiert. Sie wird daher nicht überlappend in die Basisstruktur, sondern vielmehr als ergänzende textuelle Beschreibung, als *Elementarbeschreibung* eingeblendet. Die Elementarbeschreibung und deren Elemente, die

Objektattribute, die *Objektoperationen* und die *Metaoperationen* werden im folgenden beschrieben.

7.5.1 Elementarbeschreibung

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 120f*

Definition

Eine Elementarbeschreibung spezifiziert den Funktionsumfang eines Objektes (einer Objektmenge). In ihr werden die jeweils gültig *Klasse*, die *Objektattribute*, *Stereotypen* sowie *Operationen* sowie *Erzeugungs*- und *Löschoperationen* spezifiziert.

Bemerkungen

Die Elementarbeschreibung ist eine Zusammenfassung der funktionalen Aspekte eines Objekts (einer Objektmenge). Eine Elementarbeschreibung spezifiziert

- die Klasse des Objektes
- welche Stereotypen auf das Objekt angewandt werden
- die Objektattribute, deren Datentypen sowie etwaige Initialisierungswerte
- sämtliche Objektoperationen (allgemeine und synchrone)
- sowie im Falle von Objektmengen die Erzeugungs- und Löschungsoperationen.

Die Elementarbeschreibung ist objektbezogen, jedoch enthält sie viele typspezifische Eigenschaften, also Eigenschaften, welche für alle Objekte (Objektmengen) einer Klasse zu gelten haben. Typspezifisch sind

- die Objektattribute sowie deren Datentypen, nicht aber die Initialisierungen derselben
- weitestgehend die Spezifikationen der Objektoperationen, nicht jedoch die dort versendeten und empfangenen Nachrichten)
- die Erzeugungs- und Löschungsoperationen

Stereotypen sind rein objektbezogen. Soll ein Stereotyp für alle Objekte (Objektmengen) einer Klasse gelten, ist diese speziell in der Klassendefinition zu definieren. Die einzelnen Aspekte sowie deren Notationen werden in den nachfolgenden Kapiteln im einzelnen näher erläutert.

Notation

Die Elementarbeschreibung kann wie andere Beschreibungen in ADORA-L in unterschiedlichen Formalitätsgraden spezifiziert werden (siehe Kapitel 6.5). Im folgenden wird ausschließlich die formale Syntax vorgestellt.

Elementarbeschreibung	=	"**object specification**" Objektname "**is**" ZugehörigeKlasse Stereotypenliste Objektattribute Operationsliste "**end specification** ".
ZugehörigeKlasse	=	"**class**" Klassenname "**;**" .
Stereotypenliste	=	ε | "**stereotypes**" Stereotypensignatur { "**,**" Stereotypensignatur } .
Stereotypensignatur	=	Stereotypenname ("**(**" STVariable "**=**" Ausdruck { "**,**" STVariable "**=**" Ausdruck } "**)**")
Objektattribute	=	ε | "**attributes**" Attributdeklaration { "**;**" Attributdeklaration } .
		siehe Kapitel „Objektattribute“ (S. 160)
Attributdeklaration	=	(ε | "**public**") Attributnamenliste "**:**" Datentyp ("**=**" InitialisierungAttributwert) .
Attributnamenliste	=	Attributname "{ "**,**" Attributname } .
Operationsliste	=	ε | { Operation | SyncOperation | Erzeugungsoperation | Erzeugungsoperation } .
Operation		*siehe Kapitel „Objektoperation“ (S. 161)*
SyncOperation		*siehe Kapitel „Die synchrone Operation“ (S. 163)*
Erzeugungsoperation		*siehe Kapitel „Die Metaoperation“ (S. 165)*
STVariable		*Stereotypen-Variable, siehe Kapitel „Stereotyp“ (S. 174)*

<Objektname>, <Klassenname>, <Stereotypenname> und <Attributname> sind Bezeichnernamen, die das jeweilige Modellelement namentlich beschreiben. <InitialisierungAttributwert> ist ein ASTRAL-Ausdruck, dessen Syntax an dieser Stelle nicht weiter spezifiziert wird (siehe hierzu [Coen95, 97]).

Beispiel

Im folgenden wird die Elementarbeschreibung des Objekts *Ampel* spezifiziert.

```
object specification Ampel is
    class Ampel ;
    stereotypes
        Versionsdokumentation ( Autor = 'Karl Napf', Erzeugungsdatum = 12.5.98,
                                Änderungsdatum=30.6.98, Version=1.5);
    attributes
        Startzeit, Aktivzeit : time ;
        UmschaltdauerRotGrün : time = 3s;
```

```
        UmschaltdauerGrünRot : time = 5s;
    operation …                                  siehe Kapitel 7.5.3
    syncoperation …                              siehe Kapitel 7.5.4
end specification
```

Anmerkung: Das Objekt «Ampel» hat hier denselben Namen wie die Klasse, welcher es angehört (dies kann, muß aber nicht immer der Fall sein).

7.5.2 Objektattribute

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 96f, 120f*

Definition

Das Objektattribut ist ein Datenelement, das in jedem Objekt (jeder Objektmenge) einer Klasse gleichermaßen enthalten ist und von jeder entsprechenden Objektinstanz mit einem individuellem Wert repräsentiert wird. Im Gegensatz zu Objekten haben Attribute außerhalb des Objektes, von dem sie Teil sind, keine Identität. Attribute sind vollständig unter Kontrolle des Objekts, von denen sie Teil sind (in Anlehnung an [Oestereich98, S. 233]).

Bemerkungen

Objektattribute werden durch einen Namen und eine Datenstruktur beschrieben. Die Datenstruktur wiederum wird durch einen Datentyp spezifiziert, welcher entweder lokal im Objekt oder global im Typverzeichnis (siehe Kapitel 7.6.4) festgehalten wird. Eingeführt und definiert wird ein Objektattribut in der Elementarbeschreibung eines Objektes bzw. einer Objektmenge. Objektattribute können sowohl einfache vordefinierte Datentypen (bsp.: Integer, Real, Char, String), wie auch komplexe und neu definierte Datentypen annehmen (bsp: structure of …, list of …).

Der Wert des Objektattributs kann bei Zustandsübergängen und innerhalb von Objektoperationen abgefragt und geändert werden. Bevor der Wert eines Objektattributs abgefragt werden kann, muß es zunächst initialisiert werden. Dies ist entweder direkt bei der Attributdefinition oder im Rahmen eines Zustandsüberganges bzw. einer Objektoperation möglich.

Notation

Objektattribute werden minimal durch einen Namen spezifiziert. Der Namen muß innerhalb eines Objektes (einer Objektmenge) eindeutig sein. Der Datentyp eines Objektattributs wird im Anschluß an den Attributnamen hinzugefügt (siehe Kapitel 7.5.1).

7.5.3 Objektoperation

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 96f, 120f*

Definition

Objektoperationen sind Dienstleistungen eines Objektes (einer Objektmenge), welche anderen Objekten (Objektmengen) zur Verfügung gestellt werden oder intern von anderen Objektoperationen verwendet werden. Objektoperationen verändern den Zustandsraum einer Objektinstanz, können weitere Nachrichten an Objekte (Objektmengen) versenden oder von anderen Objekten empfangen. Eine Objektoperation ist grundsätzlich zeitbehaftet (Ausnahme: synchrone Objektoperationen, siehe Kapitel 7.5.4). Angestoßen und aufgerufen wird sie durch eine entsprechende Nachricht, welche auch die für die Ausführung nötigen Parameter überbringt.

Bemerkungen

Objektoperationen werden durch ihre Signatur (durch einen Operationsnamen und durch Parameter) sowie durch eine *Voraussetzung* und eine *Zusicherung* beschrieben (auch unter der Bezeichnung *Vor-* und *Nachbedingung* bekannt). Eine Objektoperation trägt innerhalb einer Elementarbeschreibung eines Objekts (einer Objektmenge) eine eindeutige Signatur. Die Signatur ist identisch mit der Signatur der Nachricht, von welcher sie angestoßen wird.

- Die *Voraussetzung* legt fest, welche Bedingungen vor Ausführung einer Operation vorliegen müssen. Grundlage für die Formulierung einer Voraussetzung sind die Objektattribute, die Operationsparameter sowie weitere Objektoperationen.
- Die *Zusicherung* beschreibt den Zustand der Objektinstanz nach Ausführung der Objektoperation. Sie enthält Aussagen über Objektattribute (wobei die Attributwerte vor und nach Ausführung der Operation unterschieden werden), Nachrichtenparameter, andere Objektoperationen sowie über die Versendung und den Empfang von Nachrichten.

Da Objektoperationen zeitbehaftet sind, können sie nur eingeschränkt in zustandsbasierten Verhaltensbeschreibungen eingesetzt werden. Im speziellen können sie ausschließlich über Nachrichten aufgerufen und angestoßen werden, nicht aber eine Übergangsaktion spezifizieren (da ein Zustandsübergang zeitlos ist). Zeitlose Objektoperationen werden in ADORA-L als Objektoperationen bezeichnet. Sie werden im Detail in Kapitel 7.5.4 vorgestellt.

Notation

Eine Objektoperation wird syntaktisch formal folgendermaßen beschrieben:

Objektoperation = (ε | "**local**") Operationssignatur LokaleVariablen Voraussetzung Zusicherung "**end operation**".

Operationssignatur	=	"**operation**" Operationsname "**(**" Parameterliste "**)**" .
LokaleVariablen	=	ε | "**var**" Variablendeklaration { "**;**" Variablendeklaration } .
Variablendeklaration	=	Variablenname "**:**" Datentyp .
Voraussetzung	=	"**entry**" Ausdruck .
Zusicherung	=	"**exit**" Ausdruck .
Parameterliste		*siehe Kapitel „Spezifikation von Nachrichten“ (S. 127)*
Datentyp		*siehe Kapitel „Datentyp“ (S. 173)*
Ausdruck		*siehe Kapitel „Übergangsbedingung“ (S. 153)*
Parameterliste		*siehe Kapitel 7.4.6*

<Operationsname> bezeichnet den Namen der Objektoperation. Auf die genaue Syntax von Ausdrücken wird in dieser Arbeit nicht im Detail eingegangen. Sie ist identisch mit der Formulierung von Ausdrücken in der Sprache ASTRAL [Coen97]. <Variablenname> bezeichnet den Namen der verwendeten lokalen Variablen.

Darüber hinaus kann eine Objektoperation auch informal durch natürliche Sprache (in Anführungszeichen gesetzt) oder teilformal durch Adora-FSL-Schlüsselwörter und Prosa-Text beschrieben werden.

Beispiel

Die nachfolgende Objektoperation beschreibt die Initialisierung des Objektes «Geldautomat». Die Überprüfung der einzelnen Komponenten des Geldautomaten sind zeitbehaftet, die Rückmeldung kann also nur asynchron an den Versender zurückgeschickt werden (durch die Nachrichten *InitialisierungErfolgreich()* oder *InitialisierungFehlgeschlagen()*).

```
operation initialisieren ( ) ;          / Operation der Klasse Geldautomat (siehe Seite 139)
    entry
      AktiverZustand('inaktiv')
    exit
      Prüfung der Geldeingabe auf Ihre Funktionstüchtigkeit
      Prüfung der Datenbankverbindung und des Quittungsdruckers
      Falls alle Komponenten funktionstüchtig sind, schicke die InitialisierungErfolgreich()
an
      den Versender von initialisieren, andernfalls InitialisierungFehlgeschlagen().
end operation
```

7.5.4 Die synchrone Operation

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 96f, 120f*

Definition

Eine synchrone Operation ist eine Objektoperation, welche durch eine synchrone Nachricht ausgelöst wird. Die durch eine synchrone Operation spezifizierten Berechnungen sind aus Sicht des Nachrichtensenders (des Versenders der auslösenden synchronen Nachricht) zeitlos. Synchrone Operationen können daher unmittelbar Rückgabewerte an die auslösende Objektinstanz zur Verfügung stellen.

Bemerkungen

Synchrone Operationen sind zeitlose Objektoperationen. Sie sind weniger mächtig als zeitbehaftete Operationen, da sie nicht auf asynchrone und nebenläufige Berechnungen warten können. Synchrone Operationen können daher ausschließlich Objektattribute manipulieren, andere synchrone Operationen mitverwenden (da diese ja auch zeitlos sind) und allenfalls noch Nachrichten an andere Objekte (Objektmengen) abschicken.

Durch die zeitliche Einschränkung können synchrone Operationen jedoch sehr vielseitig eingesetzt werden. Sie können wie Operationen durch einen Nachricht angestoßen werden. Im Gegensatz zu Operationen ist es möglich, der anstoßenden Nachricht sofort einen Ergebnis- oder Rückgabewert zu übergeben. Dies führt zu einer einfachen und anschaulichen Modellierung der Kommunikation. Zudem können synchrone Operationen aufgrund der Eigenschaft der Zeitlosigkeit zur Spezifikation von Übergangsaktionen eingesetzt werden.

Notation

Eine synchrone Operation wird syntaktisch formal folgendermaßen beschrieben:

SyncOperation = (ε | "**local**") SyncOperationsskopf LokaleVariablen
Voraussetzung Zusicherung "**end**" Operationsname.

SyncOperationskopf = "**syncoperation**" Operationsname "**(**" " Parameterliste"**)**" "**:**"
Datentyp |
"**syncoperation**" Operationsname "**(**"Parameterliste
Parameterliste "**)**" .

LokaleVariablen	*siehe Kapitel „Objektoperation“ (S. 161)*
Voraussetzung	*siehe Kapitel „Objektoperation“ (S. 161)*
Zusicherung	*siehe Kapitel „Objektoperation“ (S. 161)*
Parameterliste	*siehe Kapitel 7.4.6*

Darüber hinaus kann eine *synchrone Operation* auch informal durch natürliche Sprache (in Anführungszeichen gesetzt) oder teilformal durch ADORA-FSL-Schlüsselwörter und durch Prosatext beschrieben werden.

Beispiel

Die erste synchrone Operation GibAmpelstatus fragt den Status des Objektes Ampel ab (siehe Seite 151). Die zweite synchrone Operation Aktivitätsdauer liefert die Zeit, in der die Ampel bisher aktiv ist.

```
syncoperation GibAmpelstatus ( ): StatusTyp ;
        entry
        exit
            ( istAktiverZustand ('aktiv') => GibAmpelstatus = aktiv ) &
            ( istAktiverZustand ('inaktiv') => GibAmpelstatus = inaktiv )
end GibAmpelstatus
```

StatusTyp = (aktiv, inaktiv) wird im entsprechenden Typverzeichnis als Datentyp definiert.

```
syncoperation Aktivitätsdauer ( ): TIME ;
        entry
            istAktiverZustand ('aktiv')
        exit
            Aktivitätsdauer = now - Startzeit ;
end Aktivitätsdauer
```

Die letzte Operation erhöhenDerPhasenwechseldauer erhöht die Zeitdauer, in der die Ampel umschaltet um den angegebenen Wert. Die veränderten Werte der Objektattribute werden hier durch die «'»-Symbolik notiert.

```
syncoperation erhöhenDerPhasenwechseldauer (Inkrement : TIME ) ;
        entry
        exit
            UmschaltdauerRotGrün' = UmschaltdauerRotGrün + Inkrement &
            UmschaltdauerGrünRot' = UmschaltdauerGrünRot + Inkrement
end erhöhenDerPhasenwechseldauer
```

7.5.5 Die Metaoperation

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 96f, 120f*

Definition

Metaoperationen sind spezielle, von der Modellierungssprache bereitgestellte Operationen, die Eigenschaften des Modellschemas liefern bzw. sie selber modifizieren.

Bemerkungen

Die Metaoperationen in ADORA-L lassen sich grundsätzlich in folgende Gruppen einteilen, in die

- Abfrage von Typeigenschaften und von Identitäten – Hier werden Operationen bereitgestellt, um die Klasse oder die Oberklassen einer Objektinstanz oder um den Datentyp eines Objektattributs zu bestimmen und zu verwalten. Darüber hinaus können die Identitäten von Objektinstanzen abgefragt und verwaltet werden.
- Abfrage der Modellstruktur – Mit diesen Operationen kann die eigentliche Struktur des Schemamodells abgefragt werden. Durch diese Operationen können Kardinalitäten von Beziehungen oder von Objektmengen abgefragt werden. Darüber hinaus können laufzeitspezifische Informationen abgefragt werden (beispielsweise die Identität des Versenders einer Nachricht oder den momentan gültigen Zustand).
- Erzeugung und Löschung von Objektinstanzen – Diese Operationen werden für die Realisierung von Objektmengen benötigt, im speziellen für Objektmengen variabler Größe. Zur Laufzeit kann durch eine Erzeugungsoperation eine Objektinstanz einer Objektmenge hinzugefügt und Instanzen aus einer Objektmenge gelöscht werden.
- Generalisierung und Spezialisierung von Objektinstanzen – Diese Operationen werden verwendet, um den Typ (die Klasse) einer Instanz dynamisch zu verändern.
- Ergänzung und Restriktion der Modellstruktur – Dieser Aspekt wird ausführlich in Kapitel 7.6.5 durch die «Stereotypen» diskutiert.

Nachfolgend werden die Metaoperationen im einzelnen aufgeführt.

Notation

Abfrage von Typeigenschaften und von Identitäten

Für die Abfrage der Objektidentität wird der spezielle Datentyp ID verwendet (siehe Kapitel 7.6.4). Jede Objektinstanz besitzt zur Ausführungszeit eine eindeutigen Identitätswert. Jedes Modellelement besitzt zudem auf Schemaebene eine eindeutige Kennung, welche durch

den Datentyp MID beschrieben wird. Unter anderem besitzt also jedes Objekt, jede Objektmenge sowie jede Klasse eine Identitätsnummer.

self: ID	Liefert die Identität der Objektinstanz, in welcher die Operation aufgerufen wird (Beispiel: Metaoperation anlegen in Kapitel 6.7.5)
idObject(ID): MID	Liefert die Identität des Objekts bzw. der Objektmenge einer Objektinstanz auf Schemaebene.
idType(ID): MID	Liefert die Identität der Klasse einer Objektinstanz auf Schemaebene
supertype (ID): MID	Liefert die Identität der Oberklasse einer Objektinstanz. Besitzt die Klasse der Instanz keine Oberklasse, ist der Rückgabewert noid
idInstance(MID): LIST OF id	Liefert zu einem gegebenen Objekt auf Schemaebene die Identität der Objektinstanz auf Instanzebene bzw. liefert eine Menge von Objektinstanzen, wenn es sich um einer Objektmenge handelt.
mid(STRING): MID	Liefert die Identität des angegebenen Elementes. Das Element muß auf der entsprechenden Schemaebene des Systemmodells aufgeführt sein und eine Bezeichnung haben.
elementName (MID): STRING	Liefert zu einer gegebenen Identität den Namen des Modellelementes auf Schemaebene
compositionOf (ID): ID	Liefert zu einer Objektinstanz die Komposition, in der sie enthalten ist.
componentsOf (ID): LIST OF ID	Liefert für eine Objektinstanz alle Objektinstanzen, welche in ihr enthalten sind.

Abfrage der Modellstruktur.

maxObjectNumber(ID): integer	Liefert die maximale Größe einer Objektmenge zurück. Ist die Menge in ihrer Größe unbestimmt, wird der maximal mögliche Integer-Wert zurückgeliefert.
minObjectNumber (ID): integer	Liefert die minimale Größe einer Objektmenge zurück. Der Ergebniswert ist stets größer oder gleich Null.
actObjectNumber (ID): INTEGER	Liefert die Größe der Objektmenge zum Zeitpunkt des Aufrufs der Operation zurück.
sourceObject (STRING): ID	Liefert zu einer Nachrichtensignatur (als String) die Identität der Objektinstanz, welche die Nachricht abgeschickt hat.
isActiveState (STRING): Boolean	TRUE, falls der angegebene Zustand ein aktiver Zustand ist, andernfalls wird FALSE zurückgegeben (funktioniert nur für die Zustände innerhalb des aufrufenden Objekts)
activeStates (): LIST of MID	Liefert eine Menge der Zustände, welche in der entsprechenden Instanz momentan aktiv sind (durch den Typ MID), (funktioniert nur für die Zustände innerhalb des aufrufenden Objekts)

Bemerkung: Die Abfrage der Kardinalitäten einer Objektmenge über maxObjectNumber, minObjectNumber, actObjectNumber wird ebenfalls im Kapitel „Objektmenge“ (S. 137) aufgeführt.

Erzeugung und Löschung von Objektinstanzen

Die Operationen zum Erzeugen und Löschen von Objektinstanzen werden in der jeweiligen Elementarbeschreibung der Objektmenge spezifiziert. Die Operationen werden analog zu den Objektoperationen durch Nachrichten gleichartiger Signatur ausgelöst.

Instanzerzeugung = Erzeugungssignatur LokaleVariablen Voraussetzung
Zusicherung "**end**" Erzeugungsname.

Erzeugungssignatur = "**create**" Erzeugungsname "**(**" Klassenname "**;**" Parameterliste "**)**" .

Instanzlöschung = Löschungssignatur LokaleVariablen Voraussetzung
Zusicherung "**end**" Löschungsname .

Erzeugungssignatur = "**dispose**" Löschungsname "**(**" Parameterliste "**)**" .

LokaleVariablen, Variablendeklaration, Voraussetzung, Zusicherung
siehe Kapitel „Objektoperation“ (S. 161)

Parameterliste *siehe Kapitel 7.4.6*

Durch <Voraussetzung> und <Zusicherung> wird die Initialisierung der neu anzulegenden Instanz bzw. vor der Löschung notwendige Aktionen spezifiziert.

Generalisierung und Spezialisierung von Objektinstanzen

Die Operationen zum Generalisieren und Spezialisieren von Objektinstanzen werden in der Elementarbeschreibung der Klasse spezifiziert. Dort wird spezifiziert, in welche andere Klasse wie spezialisiert und generalisiert wird.

Instanzspezialisierung = Spezialisierungssignatur LokaleVariablen Voraussetzung
Zusicherung "**end**".
Spezialisierungssignatur = "**specializeTo**" NameEinerUnterklasse "**(**" Parameterliste "**)**" .

InstanzGeneralisierung = Generalisierungssignatur LokaleVariablen Voraussetzung
Zusicherung "**end**".
Generalisierungssignatur = "**generalizeTo**" NameEinerOberklasse "**( )**" .

LokaleVariablen, Voraussetzung, Zusicherung
siehe Kapitel „Objektoperation“ (S. 161)

Aufgerufen werden diese Operationen entsprechend durch

spezializeTo (MID, Parameter1, Parameter2, ...) bzw.
generalize (MID) aufgerufen, wobie MID die Identität der Zielklasse spezifiziert.

Die Mutation in eine neue Klasse erfolgt also variabel, ohne in der Verhaltens- und Funktionsbeschreibung des Objekts (der Objektmenge). Die Zielklasse muß also zum Modellierungszeitpunkt nicht bekannt sein. Bedingung hierfür ist jedoch die entsprechend in der Klassenelementarbeschreibung vorhandene Metaoperation.

7.6 Das Typverzeichnis

In diesem Kapitel werden Sprachkonstrukte definiert, die für die Modellierung des Typverzeichnisses notwendig sind. Im Detail werden die Sprachkonstrukte *Klasse*, *Objektschablone*, *Klassenhierarchie*, *Datentyp* und *Stereotyp* vorgestellt.

7.6.1 Klasse

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 68f, 168f*

Definition

Eine Klasse (synonym: Objekttyp) ist ein Typ, der die gemeinsamen Eigenschaften einer Menge von gleichartigen Objekten intensional definiert. Alle Objekte und Objektmengen der Klasse sind also nach dieser Typdefinition aufgebaut.

Beschreibung

Die Klasse wird in ADORA-L rein intensional als ein Objekttyp verwendet. Die Bedeutung unterscheidet sich von der gängigen Vorstellung, in der sie dualistisch teilweise intensional als Typ als auch als Menge von Objekten verwendet wird (siehe *Kapitel 5.1*). Die ursprünglich extensionale Klassenbedeutung wird in ADORA-L durch die Einführung und Verwendung von abstrakten Objekten (und von Objektmengen) kompensiert und umgesetzt. Durch die klare Trennung von Intension und Extension können kontextbezogene Objekteigenschaften besser und präziser modelliert werden. Dies gilt insbesondere dann, wenn Objektkommunikation, Beziehungen zwischen Objekten und globales Verhalten von Objekten beschrieben werden soll.

Klassen werden in ADORA-L nur dann explizit aufgeführt, wenn die Gemeinsamkeit zweier oder mehrerer in unterschiedlichen Kontexten stehender Objekte oder wenn ein Spezialisierungszusammenhang zwischen Objekten modelliert werden soll.

Die Klassendefinition ist in ADORA-L eine Abbildung von typspezifischen Eigenschaften der zugehörigen Objektdefinitionen. Das Abbild der typspezifischen Objekteigenschaften wird als *Objektschablone* bezeichnet (siehe Kapitel 7.6.2). Klassen werden im Typverzeichnis in Form einer Klassenhierarchie (siehe Kapitel 7.6.3) modelliert. Dort werden neben der eigentlichen Klassendefinition Spezialisierungszusammenhänge modelliert (siehe Klassenspezialisierung)

Notation & Beispiel

Siehe Kapitel 7.6.2 und 7.6.3

7.6.2 Objektschablone

Konkretes Sprachkonstrukt, textuell & grafisch notiert	*siehe Seite 121f*

Definition

Eine Objektschablone beschreibt die typspezifischen Eigenschaften der Objekte und der Objektmengen einer Klasse. Die Objektdefinition eines Objektes (einer Objektmenge) muß konform sein mit den Definitionen der Objektschablone der zugehörigen Klasse.

Bemerkungen & Notation

Die *Objektschablone* ist maßgebend für alle Objekte dieser Klasse, d.h. die dort definierten Aussagen finden sich in allen zugehörigen Objekten wieder. Die *Objektschablone* einer Klasse enthält aus der

- Basisstruktur – alle Objektkomponenten der zugehörigen Objekte (die Bezeichnungen der Komponentenobjekte sind für alle Objekte einer Klasse identisch)
- strukturellen Einblendung – Sämtliche Beziehungen, die *zwischen* den Objektkomponenten der zugehörigen Objekte vorhanden sind. Strukturelle Beziehungen zu im Kontext stehende Objekte werden nicht aufgeführt.
- verhaltensorientierten Einblendung – Alle Zustände der zugehörigen Objekte, sowie alle Zustandsübergänge. Die in den Objekten aufgeführten Übergangsbedingungen und -aktionen werden in reduzierter Form übernommen. Reduziert werden Übergangsbedingungen, welche den Empfang von Nachrichten voraussetzen und Übergangsaktionen, welche Objektattribute initialisieren und welche Nachrichten versenden.
- funktionalen Einblendung – Alle Objektattribute sowie in reduzierter Form alle Objektoperationen bzw. Metaoperationen. Reduziert wird analog zur Reduktion der Übergangsaktionen.

Objektschablonen werden sowohl grafisch als auch textuell dargestellt. Sie werden weitestgehend in derselben Notation wie Objekte beschrieben, also ebenfalls durch eine Basisstruktur und durch aspektbezogene Einblendungen (siehe Kapitel 7.2, 7.3, 7.4 und 7.5). Kontextabhängige Ausdrücke in der funktionalen und der verhaltensorientierten Einblendung wird in eingeschränkter und abstrakter Form modelliert (siehe auch Notation). Kontextabhängige Ausdrücke in ADORA-L sind:

- Initialisierungen – Initialisierungswerte werden durch das Platzhaltersymbol «<>» ersetzt.
- Die Versendung und der Empfang von Nachrichten – Die Versendung und der Empfang einer asynchronen Nachricht (vgl. Kapitel 7.4.7) wird durch

"**sent**" <> Parameterliste *und* "**receive**" <Nachrichtenname> Parameterliste **over** <>

in verkürzter Form beschrieben. Synchrone Nachrichten werden entsprechend notiert durch

"<>" "(Parameterliste ")" *oder bei einem Rückgabeparameter*

"<>" "(Parameterliste ";" Parametertyp")"

Nicht dargestellt werden Nachrichtennamen und die Beziehungen, über welche die Nachrichten versendet werden. Diese verkürzte Beschreibung wird jedoch nur dann verwendet, wenn die Nachricht tatsächlich von einem Objekt versendet/empfangen, was außerhalb des Objektes liegt, welches also keine Objektkomponente ist (die Komponenten der Objekte einer Klasse sind typspezifisch, d.h. für alle Objekte dieser Klasse bindend).

Beispiel

Die Übergangsaktion des Zustandsüberganges des Objektes «Ampel» (siehe Kapitel 7.4.7, (i)) von inaktiv zu aktiv ändert sich in der Objektschablone der Klasse «Ampel» wie folgt:

```
AktivZeit = now - Startzeit
sent BlinklichtAn to Lichtanlage;
sent <> (now, AktivZeit);
```

Die Nachricht an das Objekt «Blinklichtanlage» bleibt ausführlich spezifiziert, da es eine Nachricht an eine Objektkomponente ist. Die Deaktivierungsnachricht mit den beiden Zeitinformationen jedoch wird ohne Nachrichtenname (der abhängig vom Objektkontext unterschiedlich sein kann) sowie ohne Nachrichtenpfad spezifiziert.

Die Elementarbeschreibung der *Klasse* «Ampel» (die zugehörige Klasse des *Objektes* «Ampel») sieht wie folgt aus:

```
class specification Ampel is
        stereotypes
            Versionsdokumentation ;
        attributes
            Startzeit, Aktivzeit : time ;
            UmschaltdauerRotGrün : time = 3s ;
            UmschaltdauerGrünRot : time = <> ;
         …
end specification
```

Aus der Elementarbeschreibung der Klasse wird ersichtlich, daß für sämtliche zugehörigen Objekte der Stereotyp *Versionsdokumentation* vorgeschrieben ist. Das Objektattribut *UmschaltdauerRotGrün* wird hier initialisiert, d.h. dieser Wert ist für alle Objekte (Objektmengen) der Klasse bindend. Das Objektattribut *UmschaltdauerGrünRot* hat an der Stelle der Initialisierung den Platzhalter «<>», d.h. dieser Wert muß von jedem Objekt selber neu definiert werden.

7.6.3 Klassenhierarchie

Konkretes Sprachkonstrukt, grafisch notiert	*siehe Seite 121f*

Definition

Die Klassenhierarchie teilt eine Menge von Klassen in eine Hierarchie von über- und untergeordneten Klassen ein. Eine übergeordnete Klasse wird hierbei als Oberklasse, eine untergeordnete Klasse als Unterklasse bezeichnet. Für alle Klassen der Klassenhierarchie muß gewährleistet sein, daß alle gültigen Aussagen der Oberklasse auch für die Unterklasse gültig sind.

Bemerkungen

Die Klassenhierarchie modelliert die Spezialisierungszusammenhänge zwischen Klassen. Zwischen zwei Klassen existiert ein Spezialisierungszusammenhang, wenn alle Eigenschaften der Oberklasse auch für die Unterklasse gelten (siehe Kapitel 5.6). In der Klassenhierarchie selber werden nur Spezialisierungszusammenhänge modelliert, nicht jedoch die Klassendefinitionen (diese finden sich in den jeweiligen Objektschablonen).

Eine Oberklasse kann beliebig viele Unterklassen besitzen. Umgekehrt kann eine Unterklasse beliebig vielen Oberklassen zugeordnet werden. Wird mehr als eine Oberklasse zugeordnet, spricht man auch von einer *Mehrfachspezialisierung*.

Notation

Die Klassenhierarchie wird grafisch folgendermaßen visualisiert:

- Klassen werden durch ein Rechteck notiert, in welchem in der rechten oberen Ecke ein schwarzes Dreieck eingespannt ist. Der Klassenname wird in dieses Rechteck eingefügt.
- Ein Spezialisierungszusammenhang wird durch einen Pfeil vom Klassensymbol der Unterklasse zum Klassensymbol der Oberklasse gezeichnet.
- Die Klassenhierarchie als Ganzes wird durch einen Graph gebildet, dessen Knoten die Klassensymbole und dessen Kanten die Pfeile der Spezialisierungszusamenhänge gebildet wird. Üblicherweise sollten Oberklassen oberhalb der Unterklassen dargestellt werden.

Beispiel

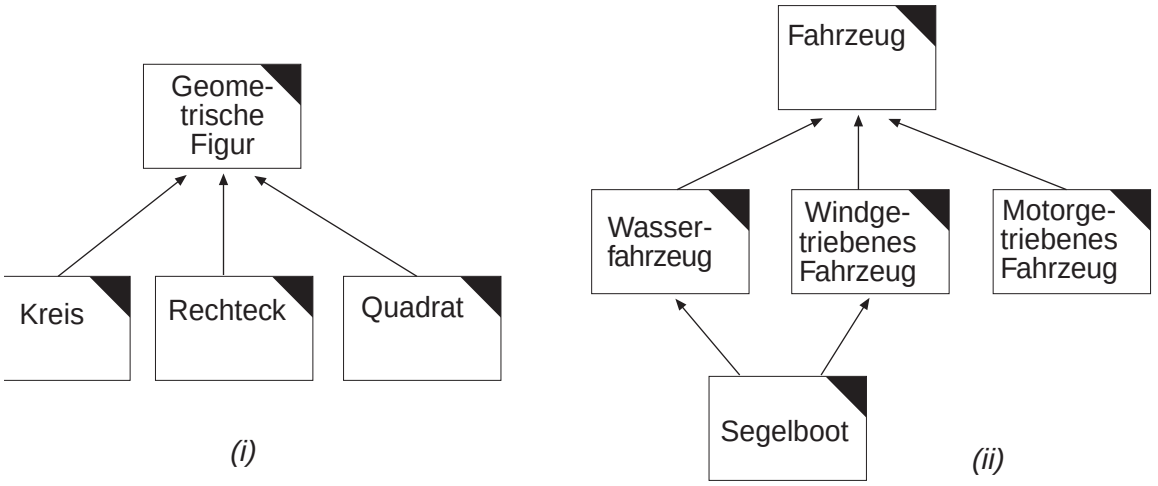


Abb. 55: Beispiele für Klassenhierarchien (in (i) streng baumartig, in (ii) als gerichteter Graph)

7.6.4 Datentyp

Konkretes Sprachkonstrukt, textuell notiert	*siehe* Seite 124*f*

Definition

Ein Datentyp definiert die Struktur und die Wertebereiche von Daten ohne eigenständige Identität. Datentypen werden in ADORA-L verwendet, um lokale Variablen in Operationen, Objektattribute und Nachrichtenparameter zu definieren.

Bemerkungen

Datentypen werden verwendet, um Datenstrukturen formal zu beschreiben. Sie sind Bestandteil der Teilsprache ADORA-FSL und werden sowohl in der verhaltensorientierten als auch in der funktionalen Einblendung verwendet. ADORA-FSL unterscheidet *primitive* und *konstruierte* Datentypen.

- Primitive Datentypen sind von Adora-FSL direkt vorgegeben. Folgende primitive Datentypen werden unterschieden: Integer, Real, Boolean, ID, MID, Time, Char, String sowie Aufzählungstypen.
- Konstruierte Datentypen sind Datentypen, die durch Verknüpfung von primitiven und anderen konstruierten Datentypen entstehen. Sie leiten sich aus den Konstruktionsregeln IS TYPEDEF, IS, IS LIST OF und IS STRUCTURE OF ab.

Die in ADORA-FSL verwendeten Datentypen sind größtenteil identisch mit denen der Sprache ASTRAL [Coen93, 95, 97]. Eine genau Beschreibung der jeweiligen Datentypen findet sich in [Coen97]. Von spezieller Bedeutung sind die Datentypen ID und MID[1]. Sie erlauben die Erfas-

sung und Verwaltung der expliziten Objekt- und Modellelementidentität. Ebenfalls von besonderer Bedeutung ist der Datentyp Time, mit dessen Hilfe relative und absolute Zeitaussagen festgehalten und formuliert werden können (siehe Kapitel 7.5.5).

Notation

Die Notation von Datentypen wird hier nur auszugsweise aufgeführt. Die ausführliche Syntax findet sich in [Coen97].

Deklaration	=	(Variablenname | Objektattributname) "**is**" Datentyp .
Datentyp	=	PrimitiverTyp | Aufzählungstyp | KonstruierterTyp .
PrimitiverTyp	=	"**Integer**" | "**Real**" | "**Boolean**" | "**ID**" | "**MID**" | "**Time**" | "**Char**" | "**String**" .
Aufzählungstyp	=	"**(**" Bezeichner { "**,**" Bezeichner} "**)**" .
KonstruierterTyp	=	Typeinschränkung | ListTyp | StrukturTyp .
Typeinschränkung	=	"**typedef**" Restriktion .
Restriktion		*Syntax und Bedeutung siehe [Coen97, S. 7f]*
ListTyp	=	"**is list of**" Datentyp .
StrukturTyp	=	"**is structure of (**" Variablenliste "**:**" Datentyp { "**;**" Variablenliste "**:**" Datentyp} "**)**".
Variablenliste	=	Variable { "**,**" Variable } .
Variable		*freie Variable, welche die Datenelemente einer Struktur identifizieren*
Bezeichner		*eine Zeichenkette, welche einen Sachverhalt oder eine Wert beschreibt.*

Beispiele

Siehe Seite 125

7.6.5 Stereotyp

Konkretes Sprachkonstrukt, textuell notiert	*siehe Seite 121f*

Definition

Stereotypen klassifizieren die Elemente eines Modells zusätzlich zu bereits vorhandenen Klassifikationsarten. Ein Stereotyp einer Modellierungssprache ist ein Beschreibungsmechanismus, der zu bestehenden Sprachkonstrukten präzisierende, ergänzende oder verändernde Aussagen über Modellelemente macht (nach [Joos98]).

[1] Der Datentyp MID wird nicht in der Sprache ASTRAL verwendet. Er wird in Kapitel 6.7 und in Kapitel 7.5.5 näher erläutert.

Bemerkungen

Stereotypen klassifizieren die möglichen Verwendungen eines Modellelementes. Die Klassifikation erfolgt durch Beschreibung gemeinsamer Eigenschaften. Stereotypen sind orthogonal zu anderen Typbeschreibungen wie etwa der Klassendefinitionen. Sie erhöhen die Flexibilität und die Aussagekraft einer Sprache und erlauben zudem eine Sprachanpassung auf vorgegebene Problembereiche. Stereotypen sollten jedoch mit Bedacht eingesetzt werden. Objektmodelle werden durch intensiven Einsatz von Stereotypen leicht überladen und werden dadurch unverständlich und wenig anschaulich. Es lassen sich generell vier Arten von Stereotypen unterscheiden (siehe [Joos98]):

- *Dekorative Stereotypen* – Zu einem Modellelement wird ein Stereotyp hinzugefügt, der die Darstellung des Modellelementes variiert. Das Modellelement erhält jedoch keine zusätzliche Information oder Bedeutung.
- *Deskriptive Stereotypen* – Zu einem Modellelement wird ein Stereotyp hinzugefügt, der das Modellelement mit standardisiertem Kommentar ergänzt. Das Modellelement wird also durch zusätzliche, nicht in seiner ursprünglichen Definition enthaltenen Eigenschaften näher spezifiziert.
- *Restriktive Stereotypen* – Ein Modellelement wird durch diese Stereotypenart formal eingeschränkt. Durch einen Stereotyp werden Restriktionen auf das Vorhandensein oder Nicht-Vorhandensein bestimmter Eigenschaften von Modellelementen spezifiziert.
- *Redefinierende Stereotypen* – Durch diese Stereotypenart werden Modifikationen auf Ebene des Metamodells vorgenommen, mit denen die Grundkonzepte der Ursprungssprache verletzt werden könnten.

In ADORA-L werden lediglich *deskriptive* und *restriktive* Stereotypen unterstützt. Zudem ist nicht möglich, die Darstellung von Modellelementen innerhalb der Sprache zu ändern. Fragen bezüglich der Visualisierung (d.h. auch die Umsetzung von dekorativen Stereotypen) sind durch entsprechende Werkzeuge zu realisieren. Redefinierende Stereotypen werden nicht unterstützt, da eine Modifikation der ADORA-L-Sprachkonzepte nicht notwendig und auch nicht sinnvoll erscheint.

Ein Modellierungselement kann mit beliebig vielen Stereotypen klassifiziert werden. Ebenso ist es möglich, daß ein Stereotyp unterschiedlichen Sprachkonstrukten zugewiesen wird.

Stereotypen werden durch folgende Eigenschaften spezifiziert:

- Durch eine eindeutige Bezeichnung
- durch die Sprachkonstrukte, welchen der Stereotyp zugewiesen werden kann. Ein konkreter Stereotyp kann darüber hinaus auch einer Gruppierung von Modellelementen zugewiesen werden. Durch diese Form der Zuweisung wird eine Modellstruktur vorgegeben.

- durch eine textuelle Kurzbeschreibung des Stereotypen in natürlicher Sprache
- durch Stereotypen-Variablen, welche einen Stereotypen zusätzlich parametrisieren
- durch Restriktionen, welche bei Anwendung eines Stereotypen vom Objektmodell zu erfüllen ist. Die Restriktionen werden aus Gründen der Vereinfachung ebenfalls durch eine informale textuelle Beschreibung formuliert (können aber also auch nicht formal verifiziert werden).

Notation

Im folgenden wird das Beschreibungsschemata für Stereotypen in EBNF vorgestellt:

Stereotyp	=	Signatur Beschreibung Variablen Bedingungen "**end stereotype**".
Signatur	=	"**stereotype**" Stereotypenname "**for**" Sprachkonstruktliste .
Sprachkonstruktliste	=	Sprachkonstrukt { "**,**" Sprachkonstrukt } .
Sprachkonstrukt	=	ElementaresSprachkonstrukt | SprachkonstruktGruppierung.
SprachkonstruktGruppierung	=	"**{**" ElementaresSprachkonstrukt { "**,**" ElementaresSprachkonstrukt } "**}**" .
Beschreibung	=	"**description**" TextuelleBeschreibung .
Variablen	=	ε | "**variables**" Variablenliste "**:**" Datentyp { "**;**" Variablenliste "**:**" Datentyp } .
Bedingungen	=	ε | "**conditions**" Bedingung { "**;**" Bedingung } .
Bedingung	=	TextuelleBeschreibung .

ElementaresSprachkonstrukt	*beliebiges Sprachkonstrukt aus Adora-L*
TextuelleBeschreibung	*Prosa-Text in natürlicher Sprache*
Variablenliste	*Syntax siehe Kapitel 7.5.3*
Datentyp	*siehe Kapitel 7.6.4*

Beispiel

Der folgende Stereotyp wird in der Elementarbeschreibung auf Seite 158 verwendet.

stereotype Versionsdokumentation
for Objekt, Objektmenge, Klasse
description
Weist dem zugeordneten Modellelement grundlegende Versionsattribute zu.
variables
Autor : STRING; Erzeugungsdatum, Änderungsdatum: TIME; Version=REAL
conditions
Änderungsdatum muß größer sein als das Erzeugungsdatum
end stereotype

Kapitel 8

Bewertung und Ausblick

In diesem abschließenden Kapitel wird die Arbeit und zusammengefaßt und die entwickelte Spezifikationssprache bewertet. Zum Schluß erfolgt eine Ausblick über weitere Aktivitäten, die mit ADORA-L und dem ADORA-Projekt in Verbindung stehen.

8.1 Schlußfolgerungen

Im Verlauf der letzten Jahre wurden eine Unmenge von objektorientierten Modellierungssprachen in Umlauf gebracht. Überraschend spärlich und verarmt fallen Versuche zur Validierung und zum Vergleich der einzelnen Sprachen aus. Zumeist wird nur das Vorhandensein und die verwendete Symbolik einzelner elementarer Sprachkonstrukte gegeneinander abgeglichen und diskutiert (siehe [Fichmann92], [Champeaux92], [Song94], [Stein94]). Weniger beurteilt werden die *Sprachkonzepte* selber, sondern vielmehr der möglichst umfangreiche Satz von *Sprachkonstrukten*. Würde man diese Art der Bewertung auf (prozedurale) Programmiersprachen anwenden, würde die Sprache *S1* mit den Schleifenkonstrukten *loop, repeat, while* und *goto* einer Sprache *S2* nur deshalb vorgezogen werden, weil diese nur *while*-Schleifen zur Verfügung stellt. Viel entscheidender für die Güte einer Programmiersprache sind aber die abstrakten Konzepte wie beispielsweise Eigenschaften des Typkonzepts (Namensäquivalenz, Strukturäquivalenz, strenge Typung, usw.), die Unterstützung von abstrakten Datentypen und Datenkapseln oder das Vorhandensein von lokalen Variablen und die Definition der entsprechenden Gültigkeitsbereiche.

Die Entwicklung der Unified Modeling Language [UML97] und deren Anspruch, ein «objektorientierter Standard» zu sein, offenbart zweierlei:

- Durch Zusammenlegen aller gebräuchlichen Diagrammarten und durch das Hinzufügen von frei definierbaren Sprachsymbolen (den *Stereotypen*) lassen sich nahezu alle klassischen objektorientierten Modellierungssprachen *emulieren*. Die konzeptuellen Unterschiede sind hingegen minimal.

- Sowohl die UML als auch die von ihr *emulierten* Sprachen weisen eklatante Schwächen bei der Spezifiation umfangreicher und komplexer Problemstellungen auf (siehe Kapitel 4.5). Die Ursachen hierfür liegen in der unübersichtlichen *Aspektmodellierung* und im Mangel an brauchbaren *Strukturierungsmechanismen*.

ADORA-L grenzt sich von der UML und von verwandten Sprachansätzen deutlich ab:

- Die Systemmodelle bilden sich in ADORA-L nicht durch ein Sammelsurium von unterschiedlichen Aspektmodellen, die jeweils durch viele unzusammenhängende gleichgeartete Diagramme aufgebaut sind. Basis ist hier ein integriertes Gesamtmodell, in dem Struktur, Verhalten und Funktionalität gemeinsam und zusammenhängend modelliert werden.
- ADORA-L-Modelle sind hierarchisch strukturiert auf Grundlage einer Teil/Ganzes-Abstraktion. Durch die hierarchische Modellierung können selbst umfangreiche Problemstellungen klar und verständlich beschrieben werden.
- Modelle werden auf der präziseren und verständlicheren Ebene der abstrakten Objekte beschrieben (und eben nicht auf Ebene von unpräzisen Klassenmodellen). Diese Form der Objektorientierung ermöglicht es, durch direkte Modellierung des Objektkontextes aussagekräftige hierarchische und integrierte Modelle zu entwickeln.
- Ein weiterer Schwerpunkt bei der ADORA-L-Sprachdefinition war die Beschreibung von Anforderungen in variablen Formalisierungsgraden. Diese Eigenschaft ist unumgänglich, um die Sprache auf reale Projekte anzuwenden. Durch die hierarchische Struktur von ADORA-L-Modellen und durch die unterschiedlich formalen Detailsprachen kann die Modellierung an Kosten und Risiko einer Entwicklung angepaßt werden.

In dieser Arbeit wurde ausführlich gezeigt, daß auf Grundlage dieser Sprachkonzepte eine praktikable und verständliche Form der Anforderungsmodellierung möglich ist und daß ADORA-L den hier diskutierten objektorientierten Ansätzen überlegen ist. Im Rahmen dieser Arbeit wurden die Vorzüge der Sprache ausschließlich theoretisch und anhand von Fallbeispielen gezeigt. Um jedoch die Adäquatheit und die Akzeptanz von ADORA-L praktisch zu zeigen, ist eine Erprobung in und durch industriellen Projekte im Sinne eines *experimentellen Software Engineerings* unumgänglich.

Voraussetzung für eine praktische Validierung ist jedoch die Entwicklung geeigneter CASE-Entwicklungswerkzeuge. Im Rahmen des ADORA-Projektes hat sich gezeigt, daß existierende Werkzeuge für eine sinnvolle hierarchische Objektmodellierung im hier vorgestellten Sinne nicht geeignet sind.

8.2 Ausblick

Die hier vorgestellte Sprache ADORA-L ist für sich gesehen vollständig und abgeschlossen. Im Verlauf der Entwicklung der Sprache ergaben sich jedoch auch eine Reihe offener Fragen, welche erst durch den praktischen Einsatz und durch tiefergehende Studien beantwortbar sind. Entsprechend muß damit gerechnet werden, daß die hier vorgestellte Version an der ein oder anderen Stelle modifiziert oder erweitert werden muß. Im folgenden werden eine Reihe offener Fragen angesprochen:

- Integration von Anwendungsfällen – Zu klären ist, ob und in welcher Form Anwendungsfälle selber in ADORA-L-Modelle aufgenommen werden sollen. Ebenfalls unklar ist, ob sich diese Anwendungsfälle überhaupt in die ADORA-L-Modellstruktur integrieren lassen oder ob diese besser separat modelliert werden sollten.
- Modellierung von Kommunikation zwischen variabel großen Objektmengen – ADORA-L modelliert im Gegensatz zu vielen anderen Methoden primär Objekte und Objektmengen und nicht Klassen. In dieser Arbeit wurden die Vorzüge eines solchen Vorgehens ausführlich diskutiert (siehe Kapitel 5). Speziell für die Modellierung des Lebenslaufs von Objektinstanzen, welche in Objektmengen eingefügt und gelöscht werden können, ergeben sich neue, präzisere Möglichkeiten. Diese neuen Möglichkeiten müssen jedoch zum Teil im praktischen Einsatz geprüft und unter Umständen präzisiert werden. Dies gilt insbesondere für die Klärung der Semantik von Objektbeziehungen und die Frage, ob und wie Beziehungen auf Instanzebene explizit modelliert werden sollen. Die Spezifikation expliziter Ausprägungsinstanzen würde zwar die Modellierungssprache zunächst komplizierter machen, andrerseits könnten so Nachrichtenflüsse besser modelliert werden, die ja über Beziehungs-“kanäle“ verschickt werden.
- Modellierung von *Stereotypen* – Stereotypen sind ein noch recht junges und unerforschtes Konzept innerhalb der Anforderungsmodellierung. In ADORA-L wurden sie zwar integriert, jedoch nur in abgeschwächter Form durch eine größtenteils informale Beschreibung notiert. Vorstellbar wäre jedoch auch eine präzisere formale Beschreibung und damit verbunden eine werkzeugbasierte Überprüfung der dort definierten Eigenschaften und Restriktionen. Hier muß geklärt werden, durch welche Eigenschaften ein Stereotyp genau definiert ist und durch welche sprachlichen Mittel diese Eigenschaften spezifiziert werden.
- Die formale Detailsprache ADORA-FSL – In der hier vorgestellten Fassung werden Funktionalitäten durch eine ASTRAL-ähnliche Notation spezifiziert. Diese Sprache ADORA-FSL ist zwar vollständig, jedoch nur auf das Nötigste beschränkt. Der praktische Einsatz muß zeigen, ob ADORA-FSL in dieser Form ausreicht oder ob sie durch zusätzliche komfortablere Sprachkonstrukte ergänzt werden muß (dies gilt insbesondere für den Bereich der Versendung und des Empfangs von Nachrichten). Zudem ist grundsätzlich zu prüfen, ob ein Programmiersprachen-ähnlicher Dialekt für die Spezifikation geeigneter wäre. Die Mächtigkeit

von Programmiersprachen ist zwar prinzipiell zu mächtig und zu komplex für die relativ einfachen Einsatzgebiete, jedoch sind Modellierer eher mit Programmiersprachen vertraut als mit der hier vorgestellten Sprache ADORA-FSL.

Die aufgeführten Fragestellungen zeigen, daß die Sprache ADORA-L in der bisherigen Form als Basissprache zu verstehen ist, die sinnvollerweise durch folgende Aspekte noch ergänzt und weiterentwickelt werden sollte. Grundlage für diese Weiterentwicklungen sind wie bereits erwähnt weitergehende konzeptionelle Klärungen auf Grundlage von praktischen Evaluierungen. Die folgende Abbildung zeigt eine Skizze möglicher Erweiterungen der Sprache ADORA-L:

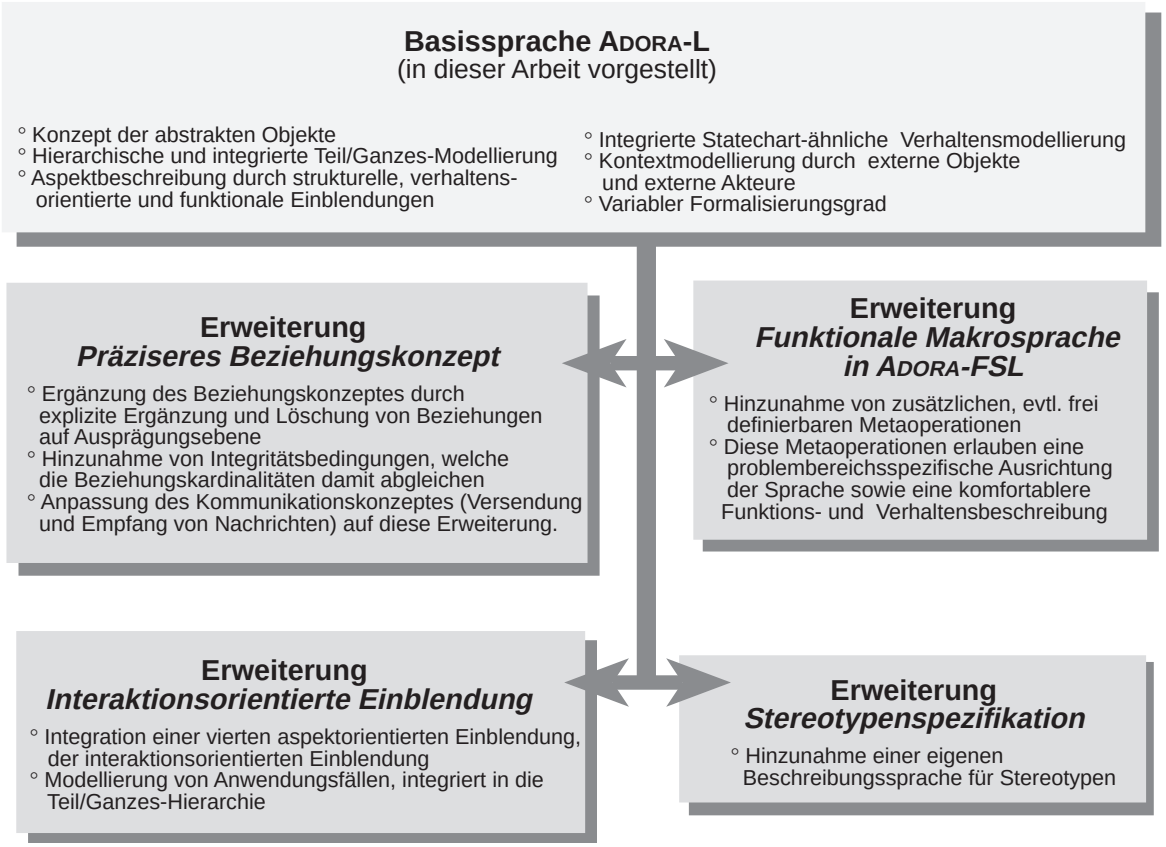


Abb. 56: Mögliche Ausbaustufen der Sprache ADORA-L für die Zukunft

Die Sprache ADORA-L ist Teil des ADORA-Projektes, welches die Entwicklung einer RE-Entwicklungsmethode zum Ziel hat. Weitere Schwerpunkte des ADORA-Projektes liegen in der Entwicklung einer Werkzeugumgebung für die Erstellung, Verwaltung und Prüfung von ADORA-Modellen und in der Einbettung von Sprache und Werkzeug in einen geeigneten RE-Prozeß. Im Werkzeugbereich sind im speziellen die Fragen zu klären,

- durch welche Konzepte ein hierarchisches Objektmodell visualisiert werden kann, und wie solche Modelle am besten erstellt und geändert werden können.

- wie aus ADORA-L-Modellen Simulationen und Prototypen generiert werden können und wie Modelle so verifiziert werden können. Problematisch ist hierbei die Tatsache, daß die Modelle nicht oder nur teilweise formal spezifiziert sind und die informalen Teile möglichst intelligent in die Ausführung integriert werden müssen.
- wie Anforderungen geeignet ermittelt und inwieweit Anwendungsfälle (engl.: Use Cases) hierfür eingesetzt werden können.
- welche Eigenschaften ein Spezifikationsprozeß aufweisen sollte, um Anforderungen mit Hilfe von ADORA-Modellen zu gewinnen, zu dokumentieren und zu prüfen.

Anhang A: Spezifikation «Ticketing System»

Auszug aus der Spezifikation des Ticketing Systems [Schmid98]

A.1 Einleitung

Im folgenden wird die Spezifikation des «Ticketing-Systems» vorgestellt. Diese Spezifikation ist ein Referenzmodell, welches innerhalb des Arbeitskreises “Simulationswerkzeuge für das RE” der GI-Fachgruppe 2.1.6 entwickelt wurde. Zielsetzung dort war es, verschiedene Methoden und Werkzeuge des Requirements Engineerings zur Modellierung von Anforderungen zu vergleichen. Das Referenzmodell wird in dieser Arbeit verwendet, um die Spezifikationssprache ADORA-L durch ein nichttriviales Beispiel vorzustellen und zu illustrieren (siehe Kapitel 6). Die Spezifikation wird an dieser Stelle nur auszugsweise aufgeführt (ausführlichere Informationen finden sich unter [Schmid98]. Sie unterscheidet folgende Teilspezifikationen:

- Das Ticketing-System
- Das Netzwerk
- Die Verkaufsstelle
- Der Veranstaltungsserver

Die Spezifikation des «Ticketing Systems» ist textuell (Prosa). Zusätzlich werden zentral geforderte Handlungsabläufe durch Anwendungsfälle (engl.: Use Cases) beschrieben.

A.2 Das Ticketing System

A.2.1 Globalanforderungen

(i) Das System soll den Verkauf von Tickets für beliebige Veranstaltungen an mehreren, geographisch verteilten Verkaufsstellen ermöglichen.

(ii) Eine Verkaufsstelle (VKS) gestattet den Kauf von Tickets einer wählbaren Veranstaltung. Die VKS soll Informationsanzeige, Benutzereingaben, Sitzplatzreservation, Buchung, Bezahlung und Ticketerstellung (Druck) handhaben.

(iii) Eine VKS akzeptiert sowohl Bargeld als auch Kredit- und Debitkarten als Zahlungsmittel.

(iv) Die VKS soll ein Selbstbedienungsautomat sein.

(v) An einer VKS werden die angebotenen Veranstaltungen angezeigt, sofern keine Verkaufstransaktion im Gange ist.

(vi) Der Benutzer einer VKS kann einen noch nicht abgeschlossenen Ticketkauf jederzeit abbrechen (Abbruchfunktion). Eine Verkaufstransaktion gilt als abgeschlossen, wenn der Benutzer bezahlt hat und die Bezahlung von der VKS akzeptiert wurde. Danach, d.h. während der Ticketerstellung und -ausgabe, ist ein Abbruch nicht mehr möglich.

A.2.2 Grundarchitektur

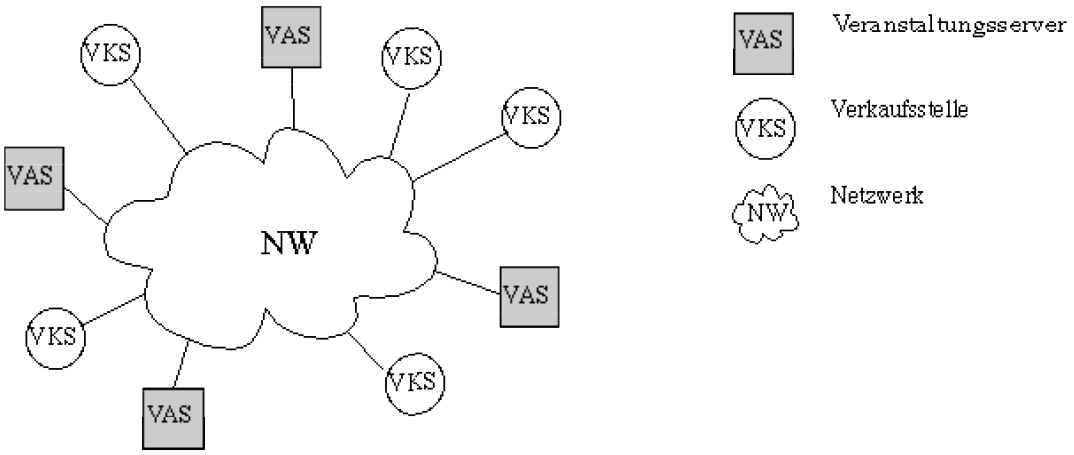
Definitionen

- Ein Knoten ist eine Verkaufsstelle oder ein Veranstaltungsserver, sozusagen deren gemeinsame abstrakte Superklasse.

Entwurfsentscheidungen

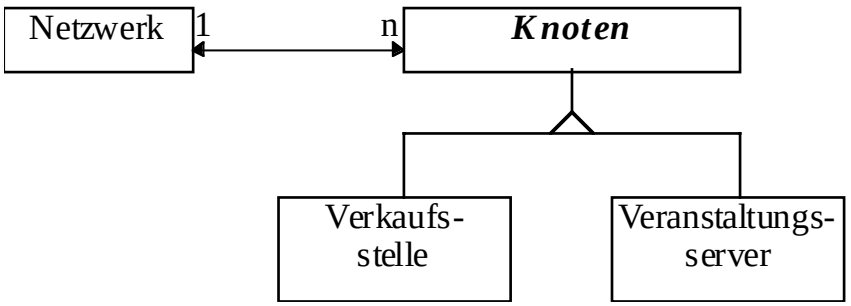
- Figur 1 zeigt die primäre Architektur des Ticketing Systems. Das System besteht aus den Komponenten:
  - Verkaufsstelle (VKS),
  - Veranstaltungsserver (VAS) und
  - Netzwerk (NW).

  Verkaufsstellen und Veranstaltungsserver können in beliebiger Anzahl vorhanden sein.



Figur 1: Schema des Ticketing Systems

- Figur 2 zeigt das Ticketing System als Klassenmodell.



Figur 2: Klassenmodell des Ticketing Systems

- Ein Veranstaltungsserver ist für die Verwaltung genau einer Veranstaltung zuständig (logisches Serverobjekt).
- Der VAS besitzt die notwendigen Informationen über die Veranstaltung: Bezeichnung, Datum, Ort, Beginn, Dauer, Beschreibung sowie den zugehörigen Sitzplätzen (Preis, Status (frei / reserviert / verkauft), Position (räumliche Anordnung der Sitzplätze)).
- Das Netzwerk dient als Kommunikationsmittel zwischen den geographisch verteilten Knoten, d.h. jeder Knoten kann nur mittels der Netzwerkdienste mit anderen Knoten kommunizieren.
- Knoten fallen nicht aus, sondern melden sich stets beim Netz ab.
- Das Netzwerk wird als gegeben angenommen (Blackbox-Sicht). Einzig dessen Dienstleistungen und Schnittstelle sind spezifiziert
- Da die Kommunikationsinfrastruktur explizit durch das Objekt “Netzwerk” modelliert wird, ergeben sich zwei Kommunikationsschichten:
  1. Die Netzmeldungen, die direkt zwischen Netzwerk und Knoten ausgetauscht werden
  2. Die Applikationsmeldungen, die indirekt zwischen Verkaufsstelle und Veranstaltungsserver mittels Netzwerkdienste ausgetauscht werden

A.3 Das Netzwerk

A.3.1 Globalanforderungen

Das Netzwerk soll...:

(i) ... An- und Abmeldungen von Knoten handhaben. Bei einer Anmeldung wird sowohl der Knotennamen als auch die Gruppenzugehörigkeit festgelegt.

(ii) ... Gruppen von Knoten verwalten, wobei ein Knoten zu mehreren Gruppen gehören kann. Diese Gruppen legen fest, an welche Knoten Broadcast-Meldungen zu verteilen sind.

(iii) ... Singlecast-Meldungen an einzelne Knoten versenden können. Rückmeldungen vom Empfängerknoten müssen von diesem bei Bedarf selbständig initiiert werden. Singlecasting ist somit auf der Ebene der Applikationsmeldungen ein asynchroner Dienst.

(iv) ... Broadcast-Meldungen an Gruppen vornehmen können. Rückmeldungen von den Empfängerknoten müssen von diesen bei Bedarf selbständig initiiert werden. Broadcasting ist somit auf der Ebene der Applikationsmeldungen ein asynchroner Dienst.

(v) ... ohne Paketverluste und “atomar” arbeiten, d.h. Meldungen erreichen den Zielknoten stets ganz und korrekt.

(vi) ... mehrere, an den gleichem Empfänger parallel abgeschickte Meldungen (bspw. Antworten auf eine Broadcast-Meldung) diesem sequentiell, jedoch in beliebiger Reihenfolge abliefern.

A.3.2 Bemerkungen

- Das Netzwerk wird als gegeben angenommen. Daher werden an dieser Stelle keinerlei Interna modelliert.
- Im Rahmen des Ticketing Systems werden die Gruppen “Verkaufsstelle” und “Veranstaltungsserver“ benötigt, wobei sich jeder Knoten nur bei einer dieser Gruppen eintragen läßt.

A.4 Die Verkaufsstelle

A.4.1 Globalanforderungen

(i) Eine VKS ist mit einem Touchscreen ausgestattet, welches sowohl als Eingabe- wie auch als Ausgabegerät zum Benutzer dient.

(ii) Eine VKS ist mit einem Drucker für die Ticketerstellung und einem Ausgabeschacht für die Ticketausgabe an den Benutzer ausgestattet. Gedruckte Tickets werden in den Ausgabeschacht abgelegt. Der Ausgabeschacht ist für den Benutzer frei zugänglich.

(iii) Anwendungsfall “Initialisierung“:

Akteure:	Systemtechniker Netzwerk
Auslöser:	Ein Systemtechniker startet die VKS. <Siehe “NodeStartup(...)”>
Vorbedingung:	Der Status der VKS ist {gestoppt}.
Ablauf:	
	Die VKS meldet sich beim Netzwerk mit ihrem Namen und der Gruppe “Verkaufsstelle” an. <Siehe “NodeConnect(...)”>
	Die VKS startet eine vollständige Angebotsabfrage. <Siehe “GetService(...)”>
	Die VKS zeigt auf dem Touchscreen den Begrüßungstext an.
	Die VKS initialisiert den eigenen Drucker.
	Die VKS initialisiert ihre interne Liste der verfügbaren Veranstaltungen als leere Liste.
	Die VKS setzt ihren Status auf {betriebsbereit, unbenutzt}.
Nachbedingung:	Der Status der VKS ist {betriebsbereit, unbenutzt}. Die VKS-interne Liste der verfügbaren Veranstaltungen ist leer. Der VKS-Touchscreen und -Drucker sind initialisiert. Die VKS ist beim Netzwerk angemeldet.
Ausnahmen:	-
Bemerkungen:	-

(iv) Anwendungsfall “Veranstaltungsübersicht”:

Akteure:	Benutzer Netzwerk Veranstaltungsserver
Auslöser:	Ein Benutzer startet die Veranstaltungsübersicht. <Siehe “StartDetailView(...)”>
Vorbedingung:	Der Status der VKS ist {betriebsbereit, unbenutzt}.
Ablauf:	
	Die VKS setzt ihren Status auf {betriebsbereit, Abfrage gestartet}.
	Die VKS zeigt auf dem Touchscreen eine Übersicht über die angebotenen Veranstaltungen anhand der internen Liste an. Der Benutzer kann die Anzeige “scrollen”, wenn nicht die gesamte Übersicht auf dem Touchscreen Platz findet.
Nachbedingung:	Die VKS zeigt eine Veranstaltungsübersicht auf dem Tochscreen an. Der Status der VKS ist {betriebsbereit, Abfrage gestartet}.
Ausnahme A:	
	Wenn (in Schritt 2) und ((der Benutzer die Abbruchfunktion betätigt) oder (seit 10 Minuten keine Eingabe seitens des Benutzers)) dann <Siehe “Cancel(...)”>
	Die VKS zeigt auf dem Touchscreen den Begrüßungstext an.
	Die VKS setzt ihren Status auf {betriebsbereit, unbenutzt}.
	Ende
Nachbed. A:	Die VKS zeigt auf dem Touchscreen den Begrüßungstext an. Der Status der VKS ist {betriebsbereit, unbenutzt}.
Bemerkungen:	Die VKS kann max. einen Benutzer gleichzeitig bedienen.

(v) Anwendungsfall “Veranstaltungsabfrage”:

Akteure:		Benutzer Netzwerk Veranstaltungsserver
Auslöser:		Der Benutzer wählt eine beliebige Veranstaltung, die in der Veranstaltungsübersicht enthalten ist, durch antippen der Beschreibung. <Siehe “StartDetailView(...)”>
Vorbedingung:		Der Status der VKS ist {betriebsbereit, Abfrage gestartet}.
Ablauf:		
	Die VKS verlangt die Detailinformationen der vom Benutzer gewählten Veranstaltung vom zuständigen VAS. <Siehe “GetDetailInformation(...)”>	
	Der VAS schickt die geforderten Informationen zur VKS. <Siehe “ReturnDetailInformation(...)”>	
	Die VKS setzt ihren Status auf {betriebsbereit, Veranstaltung gewählt}.	
	Die VKS zeigt die erhaltenen Informationen auf dem Touchscreen an. Der Benutzer kann die Anzeige “scrollen”, wenn nicht alle Informationen gleichzeitig auf dem Touchscreen Platz finden.	
Nachbedingung:		Der Benutzer hat eine spezifische Veranstaltung ausgewählt, in dem er deren Detailinformationen anforderte. Die VKS zeigt die Detailinformationen der vom Benutzer gewählten Veranstaltung auf dem Tochscreen an. Der Status der VKS ist {betriebsbereit, Veranstaltung gewählt}.
Ausnahme A:		
	Wenn (in Schritt 1 - 4) und ((der Benutzer die Abbruchfunktion betätigt) oder (seit 10 Minuten keine Eingabe seitens des Benutzers)) dann <Siehe “Cancel(...)”>	
	Die VKS zeigt auf dem Touchscreen den Begrüßungstext an.	
	Die VKS setzt ihren Status auf {betriebsbereit, unbenutzt}.	
	Ende	
Nachbed. A:		Die VKS zeigt auf dem Touchscreen den Begrüßungstext an. Der Status der VKS ist {betriebsbereit, unbenutzt}.
Bemerkungen:		Die VKS kann max. einen Benutzer gleichzeitig bedienen.

(vi) Anwendungsfall “Ticketkauf”:

Akteure:		Benutzer Netzwerk Veranstaltungsserver
Auslöser:		Ein Benutzer startet den Ticketkauf. <Siehe “StartPurchase(...)”>
Vorbedingung:		Der Status der VKS ist {betriebsbereit, Veranstaltung gewählt}.
Ablauf:		
	Wiederhole	
	Der Benutzer markiert seine gewünschten Plätze durch antippen der graphischen Darstellung der Sitzplatzanordnung. Der Benutzer kann keine bereits verkaufte Plätze markieren. Durch nochmaliges Antippen kann eine Markierung wieder aufgehoben werden.	
	Bis der Benutzer verlangt die Reservation der markierten Plätze.	
	Die VKS setzt ihren Status auf {betriebsbereit, Reservation gestartet}.	
	Die VKS nimmt die Reservation anhand der markierten Plätze bei dem zuständigen VAS vor. <Siehe “DoReservation(...)”>	
	Der VAS bestätigt die Reservation. <Siehe “ReservationConfirmed(...)”>	
	Die Bezahlung wird abgewickelt. <Siehe “PurchaseConfirmed(...)”>	
	Die VKS setzt ihren Status auf {betriebsbereit, Bezahlung erfolgt}.	
	Die VKS druckt pro bezahltem Platz ein Ticket.	
	Der Benutzer entnimmt die gedruckten Tickets dem Ausgabeschacht.	

	Die VKS zeigt auf dem Touchscreen eine neue Übersicht über die angebotenen Veranstaltungen anhand der internen Liste an.
	Die VKS setzt ihren Status auf {betriebsbereit, unbenutzt}.
Nachbedingung:	Der Status der VKS ist {betriebsbereit, unbenutzt}. Die VKS zeigt auf dem Touchscreen den Begrüßungstext an.
Ausnahme A:	
	Wenn (in Schritt 2 - 7) und ((der Benutzer die Abbruchfunktion betätigt) oder (seit 10 Minuten keine Eingabe seitens des Benutzers)) dann
	Wenn in Schritt 5 - 7 dann
	Die VKS meldet dem zuständigen VAS den Abbruch der Reservation. <Siehe “CancelReservation(...)”>
	Ende
	Die VKS zeigt auf dem Touchscreen den Begrüßungstext an.
	Die VKS setzt ihren Status auf {betriebsbereit, unbenutzt}.
	Ende
Nachbed. A:	Der Status der VKS ist {betriebsbereit, unbenutzt}. Die VKS zeigt auf dem Touchscreen den Begrüßungstext an.
Ausnahme B:	
	Wenn (in Schritt 6) und (der VAS weist die Reservation zurück) dann <Siehe “ReservationRejected(...)”>
	Die VKS hebt die Markierung der als verkauft gemeldeten Plätze auf.
	Die VKS zeigt auf dem Touchscreenden Text “Einige Ihrer markierten Plätze wurden soeben verkauft. Bitte wählen Sie andere Plätze” an.
	Weiter mit Schritt 1.
	Ende
Nachbed. B:	-
Bemerkungen:	Die VKS kann max. einen Benutzer gleichzeitig bedienen. Der Benutzer wählt seine gewünschte Veranstaltung mittels Anwendungsfall 4 und 5. Der Benutzer kann keine Plätze reservieren, die bereits der VAS als “verkauft” gemeldet hat.

(vii) Anwendungsfall “Betriebsbereitschaft”:

Akteure:	Veranstaltungsserver Netzwerk
Auslöser:	Ein VAS fragt den Status der VKS ab. <Siehe “GetAliveStatus(...)”>
Vorbedingung:	Der Status der VKS enthält {betriebsbereit}.
Ablauf:	
	Die VKS antwortet dem VAS mit “betriebsbereit”. <Siehe “ReturnAliveStatus(OK)”>
Nachbedingung:	-
Ausnahmen:	-
Bemerkungen:	Dieser Anwendungsfall wird von den VAS benötigt, damit VKS-Ausfälle (bspw. Störung) detektiert und deren noch nicht bezahlte Reservationen aufgehoben werden können.

(viii) Anwendungsfall “Veranstaltung neu verfügbar”:

Akteure:	Veranstaltungsserver Netzwerk
Auslöser:	Ein VAS meldet eine Veranstaltung an. <Siehe “ServiceAvailable(...)”>
Vorbedingung:	Der Status der VKS enthält {betriebsbereit}.
Ablauf:	

	Die VKS fügt die neu verfügbare Veranstaltung in die interne Liste der verfügbaren Veranstaltungen.	
	Wenn der Status der VKS ist {betriebsbereit, Abfrage gestartet} dann	
	Die VKS zeigt auf dem Touchscreen eine neue Übersicht über die angebotenen Veranstaltungen anhand der internen Liste an.	
	Ende	
Nachbedingung:		Die VKS-internen Liste der verfügbaren Veranstaltungen enthält die Angaben der neu verfügbaren Veranstaltung. Die VKS zeigt gegebenenfalls (siehe Schritt 2) auf dem Touchscreen eine neue Übersicht über die angebotenen Veranstaltungen anhand der internen Liste an.
Ausnahmen:		-
Bemerkungen:		-

(ix) Anwendungsfall “Veranstaltung nicht verfügbar”:

Akteure:		Netzwerk Veranstaltungsserver Benutzer
Auslöser:		Ein VAS meldet eine Veranstaltung ab. <Siehe “ServiceUnavailable(...)”>
Vorbedingung:		Der Status der VKS enthält {betriebsbereit}.
Ablauf:		
	Die VKS entfernt die nicht mehr verfügbare Veranstaltung von der internen Liste der verfügbaren Veranstaltungen.	
Nachbedingung:		Die Angaben über die abgemeldete Veranstaltung sind aus der internen Liste entfernt.
Ausnahme A:		
	Wenn (am Ende des Anwendungsfalls) und (der Status der VKS enthält ein Element aus {Abfrage gestartet, Veranstaltung gewählt, Reservation gestartet}) dann	
	Die VKS zeigt den Text “Diese Veranstaltung wird nicht mehr angeboten.” auf dem Touchscreen an.	
	Die VKS wartet max. 30 Sekunden auf eine Bestätigung vom Benutzer.	
	Die VKS zeigt auf dem Touchscreen eine neue Übersicht über die angebotenen Veranstaltungen anhand der internen Liste an.	
	Die VKS setzt ihren Status auf {betriebsbereit, Abfrage gestartet}.	
	Ende	
Nachbed. A:		Der Status der VKS ist {betriebsbereit, Abfrage gestartet}. Die VKS zeigt auf dem Touchscreen eine neue Übersicht über die angebotenen Veranstaltungen anhand der internen Liste an.
Bemerkungen:		-

(x) Anwendungsfall “Angebot aktualisieren”:

Akteure:		Netzwerk Veranstaltungsserver Benutzer
Auslöser:		Ein VAS meldet eine Änderung in seiner Veranstaltung. <Siehe “ServiceChanged(...)”>
Vorbedingung:		Der Status der VKS enthält {betriebsbereit, Veranstaltung gewählt}. Die gemeldete Änderung betrifft die vom Benutzer gewählte Veranstaltung.
Ablauf:		
	Die VKS verlangt die Detailinformation der geänderten Veranstaltung vom zuständigen VAS. <Siehe “GetDetailInformation(...)”>	
	Der VAS schickt die geforderten Informationen zur VKS. <Siehe “ReturnDetailInformation(...)”>	
	Die VKS zeigt die neu erhaltenen Veranstaltungsinformationen auf dem Touchscreen an.	
Nachbedingung:		Die VKS zeigt die neu erhaltenen Veranstaltungsinformationen auf dem Touchscreen an.
Ausnahmen:		-
Bemerkungen:		-

(xi) Anwendungsfall “Störung”:

Akteure:	<Systemintern>
Auslöser:	Die VKS detektiert eine Störung.
Vorbedingung:	Der Status der VKS enthält {betriebsbereit}.
Ablauf:	
	Wenn der Status der VKS enthält {Reservation gestartet} dann
	Die VKS meldet dem zuständigen VAS den Abbruch der Reservation. <Siehe “CancelReservation(...)”>
	Sonst wenn der Status der VKS enthält {Bezahlung erfolgt} dann
	Die VKS erstattet den bereits bezahlte Betrag dem Benutzer zurück. Je nach Zahlungsart gibt die VKS entweder das Bargeld zurück oder storniert die Buchung beim entsprechenden Finanzinstitut (Bank, Kreditfirma).
	Ende
	Die VKS den Text “Diese VKS ist defekt. Bitte diese VKS nicht mehr benützen.” auf dem Touchscreen an.
	Die VKS meldet sich beim Netzwerk mit seinem Namen ab. <Siehe “NodeDisconnect(...)”>
	Die VKS setzt ihren Status auf {Störung}.
Nachbedingung:	Die VKS zeigt die Störung mit einem entsprechenden Text auf dem Touchscreen an. Die VKS ist beim Netzwerk abgemeldet. Status der VKS ist {Störung}.
Ausnahmen:	-
Bemerkungen:	Beispiel einer Störung ist ein Defekt des Druckers.

(xii) Anwendungsfall “Deaktivieren”:

Akteure:	Systemtechniker Netzwerk
Auslöser:	Ein Systemtechniker stoppt die VKS. <Siehe “NodeShutdown(...)”>
Vorbedingung:	Der Status der VKS ist {betriebsbereit, unbenutzt} oder {Störung}.
Ablauf:	
	Wenn der Status der VKS ist {betriebsbereit, unbenutzt} dann
	Die VKS meldet sich beim Netzwerk mit seinem Namen ab. <Siehe “NodeDisconnect(...)”>
	Ende
	Die VKS setzt ihren Status auf {gestoppt}.
Nachbedingung:	Status der VKS ist {gestoppt}. Der Touchscreen ist deaktiviert. Die VKS ist beim Netzwerk abgemeldet.
Ausnahmen:	-
Bemerkungen:	-

(xiii) Randbedingung: Die VKS kann nur einen Benutzer gleichzeitig bedienen (siehe Anwendungsfall (ii) und (iii).

(xiv) Randbedingung: Vandalenschutz und ähnliche Sicherheitsmaßnahmen für eine VKS werden nicht modelliert.

(xv) Randbedingung: Die VKS stellt die Informationen auf dem Touchscreen lediglich zweistufig dar: Begrüßungstext - Veranstaltungsübersicht - Detailinformationen.

A.4.2 Bemerkungen

Die Randbedingung (xiv) dürfte für ein reelles System aufgrund der großen Anzahl von Veranstaltungen kaum sinnvoll sein. Mindestens eine zusätzliche Einteilung in Kategorien wie Kino, Theater, Zirkus, Open Air, etc. müßte hinzugefügt werden. Um jedoch das Referenzmodell überschaubar zu halten, sei diese Randbedingung gestattet.

Anhang B: Literaturverweise

[Alshawi92] Alshawi H., D. Carter, R. Crouch, S. Pulman, M. Rayner, A. Smith (1992): *CLARE, a Contextual Reasoning and Cooperative Response Framework for the Core Language Engine*. Final Report, SRI International.

[Alford82] M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, F. B. Schneider (1982): *Distributed Systems: Methods and Tools for Specification*. Lectures Notes in Computer Science, Springer-Verlag.

[Alford85] M. W. Alford (1985): *SREM at the Age of Eight, The Distributed Computing Design System*. IEEE Computer, Vol. 4, April 1985, pp. 36-46.

[Anderson93] Anderson, J. S. B. (1993): *Using Scenarios in Deficiency-Driven Requirements Engineering.* Proceedings IEEE Symposium on Requirements Engineering, San Diego, pp. 134-141.

[Basili75] Basili, V.R., A.J. Turner (1975): *Iterative Enhancement: A Practical Technique for Software Development*. IEEE Transactions on Software Engineering, Vol. SE-1 (6), pp. 390-396.

[Belina91] Belina, F., D. Hogrefe, A. Sarma (1991): *SDL with Applications from Prototypes to Specification*. Carl Hanser Verlag & Prentice Hall International.

[Berner98a] Berner, S., S. Joos., M. Glinz, M. Arnold (1998): *A Visualization Concept for Hierarchical Object Models*. Proceedings of the 13th IEEE International Automated Software Engineering Conference (ASE 1998), Honolulu, Hawaii. Washington, etc.: IEEE Computer Society, Oct. 1998, pp. 225-228.

[Berner98b] Berner, S., S. Joos., M. Glinz, M. Arnold (1998): *Visualizing Adora Models*. TR-98-09, Institut für Informatik, Universität Zürich.

[Bjorner78] Bjorner, D., C. Jones (1978): *The Vienna Development Method*. New York, Springer Verlag.

[Booch94] Booch, G. (1994): *Object-Oriented Analysis and Design with Applications,* 2nd ed. The Benjamin/Cummings Publishing Company, Inc.

[Booch98] Booch, G., I. Jacobson, J. Rumbaugh (1998): *The Unified Modeling Language for Object-Oriented Development*, Documentation Set Version 1.1, Rational Software Corporation.

[Boehm81] Boehm, B. (1981): *Software Engineering Economics*. Englewood Cliffs, N.J., Prentice-Hall.

[Blair91] Blair G., J. Gallagher, D. Hutchison, D. Shepherd (ed). (1991): *Object-Oriented Languages, Systems and Applications*. Pitman.

[Brockhaus94] Brockhaus F. A. (1994): *Der große Brockhaus*. Ausgabe 1994, Brockhaus Wiesbaden.

[Cameron86] Cameron, J. R. (1986): *An Overview of JSD*. IEEE Transactions on Software Engineering, Vol SE-12, 2, pp. 222-240.

[Celco83] Celco, J et al. (1983): *A Demonstration of Three Requirements Language Systems*. ACM SIGPLAN Notices Vol. 1 (18), pp. 9-14.

[Champeaux92] De Champeaux, D. P. Faure (1992): *A Comparative Study of Object-Oriented Analysis Methods*. Journal of Object-Oriented Programming, Vol. 5 (1), März/April 92.

[Chen76] Chen, P.P. (1976): *The Entity-Relationship Model – Toward a Unified View of Data*. ACM Transactions on Database Systems, Vol. 1 (1), pp. 9-36.

[Coad91] Coad, P., E. Yourdon (1991): *Object-Oriented Analysis*. Yourdon Press Computing Series.

[Coen93] Coen-Porisini, A., and R.A. Kemmerer (1993): *The Composability of ASTRAL Realtime Specifications*. Proceedings of the International Symposium on Software Testing and Analysis, Cambridge, Massachusetts (July 1993).

[Coen95] Coen-Porisini, A., R. Kemmerer and D. Mandrioli (1995): *A Formal Framework for ASTRAL Inter-level Proof Obligations*. Proceedings Fifth European Software Engineering Conference, Barcelona, Spain, September 1995.

[Coen97] Coen-Porisini, A., C. Ghezzi, R. A. Kemmerer (1997): *Specification of Real-time Systems Using ASTRAL*. IEEE Transactions on Software Engineering, Vol. 23 (9), pp. 704-736.

[Crow95] Crow J., J. Rushby, N. Shankar, M. Srivas (1995): *A Tutorial Indroduction to PVS*. WIFT ´95 Workshop on Industrial-Strength Formal Specifiaction Techniques, April 1995.

[Davis90] Davis, A. M. (1990): *Software Requirements, Analysis and Specification*. Prentice Hall, New Jersey.

[Davis93] Davis, A. M., S. Overmeyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, M. Theofanos (1993): *Identifying and Measuring Quality in a Software Requirements Specification.* Proc. Software Metrics Symposium, IEEE CS Press, Los Alamitos, California, pp. 141-153.

[DeMarco78] DeMarco, T. (1978): *Structured Analysis and System Specification*. Yourdon Press, New York.

[DOD85] Department of Defence, USA (1985): *Software Requirements Specification.* (SRS) DID, DI-IPSC-81433, DOD MIL-STD-490A, Specification Practices, (Juni 1985).

[Engesser88] Engesser H. (1988): *Informatik Duden*. Dudenverlag Mannheim.

[Eckert94] Eckert, G. (1994): *Types, Classes and Collections in Object-Oriented Analysis*. Proceedings ICRE´94, International Conference on Requirements Engineering, Colorado Springs, Colorado, USA.

[Embley92] Embley, D.W., , B.D. Kurtz, S.N. Woodfield(1992): *Object-Oriented Systems Analysis*. Prentice-Hall International, Inc.

[Fairley85] Fairley, R. (1985): *Software Engineering Concepts*. New York, etc.: McGrawHill.

[Firesmith96] Firesmith, D., , B. H. Henderson-Sellers, I. Graham, M. Page-Jones (1996): *Open Modeling Language (OML) – Reference Manual*. OPEN Consortium.

[Fichmann92] Fichmann, R. G., C. F. Kemerer (1992): *Object-Oriented and Conventional Analysis and Desing Methodologies*. IEEE Computer, Oktober 92, pp. 22-39.

[Floyd84] Floyd, C. (1984): *A Systematic Look at Prototyping*. In: Namur (Hrsg.), Approaches to Prototyping, Proceedings of the Working Conference on Prototyping, Springer-Verlag.

[Floyd89] Floyd, C. , W. Mehl, F. Reisen, G. Schmid, G. Wolf (1989): *Out of Scandinavia: Alternative Approaches to Software Design and System Development.* Human Computer Interaction 4, pp. 235-250.

[Forrester71] Forrester, J. (1971): *System Dynamics*. Betriebwirtschaftlicher Verlag, Dr. Th. Gabler, Wiesbaden.

[Frühauf91] Frühauf, K., J. Ludewig, H. Sandmayr (1991): *Software-Prüfung, eine Fibel*. vdf-Verlag der Fachvereine Zürich und Teubner Stuttgart.

[Gane79] Gane, C., T. Sarson (1979): *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, Englewood Cliffs, N. J.

[Glinz91] Glinz, M. (1991): *Probleme und Schwachstellen der strukturierten Analyse*. In: Timm (Hrsg.), Requirements Engineering ´91: Structured Analysis und verwandte Ansätze, April 91, Springer Verlag, pp. 14-39.

[Glinz93] Glinz, M. (1993): *Hierarchische Verhaltensbeschreibung in objektorientierten Systemmodellen – eine Grundlage für modellbasiertes Prototyping*. In Züllighofen, Heinz (ed.) / Altmann Werner (coed.) / Doberkat, Ernst-Heinrich (coed.), *Requirements Engineering 1993: Prototyping*, Bonn, Teubner Stuttgart, (pp. 175-192).

[Glinz95] Glinz, M. (1995): *An Integrated Formal Model of Scenarios Based on Statecharts*. In Schäfer, W. and Botella, P. (Hrsg.) (1995). Software Engineering - ESEC '95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. Berlin, etc.: Springer (Lecture Notes in Computer Science 989), pp. 254-271.

[Glinz98a] Glinz, M. (1998): *Software Engineering I*. Vorlesungsskript, WS 1998/99. Institut für Informatik, Universität Zürich.

[Glinz98b] Glinz, M. (1998): *Informatik, Teil II (Modellierung)*. Vorlesungsskript, SS 1998. Institut für Informatik, Universität Zürich.

[Greenbaum91] Greenbaum, J., M. Kyng (1991): *Design at Work: Cooperative Design of Computer Systems*. Hillsdale, NJ, Lawrence Erlbaum.

[Hall90] A. Hall (1990): *Seven Myth of Formal Methods*. IEEE Software, Vol. 7 (5), pp.11-19.

{Hallmann97] Hallmann, B. (1997): *Are Classes necessary?* Journal of Object-Oriented Programming, Vol. 10 (5), pp. 19-27.

[Harel87] Harel, D. (1987): *Statecharts: A Visual Formalism for Complex Systems*. Scie. Computer Program (1987), Vol. 8, pp. 231-274.

[Harel88] Harel, D. (1988): *On Visual Formalisms*. Communications of the ACM, Vol. 31 (5), pp. 514-530.

[Harel96a] Harel D. , M. Politi (1996): *Modeling Reactive Systems with Statecharts: The Statemate Approach*. Part No. D-1100-43, 6/96, I-Logix Inc. Three Riverside Drive, Andover, MA 01810.

[Harel96b] Harel D. , E. Gery (1996): *Executable Object Modeling with Statecharts*. 18th International Conference on Software Engineering. Conference Proceedings, Berlin, 1996, pp. 246-257.

[Harel97] Harel D. , E. Gery (1997): *Executable Object Modeling with Statecharts*. IEEE Computer, Vol. 30 (7), July 1997, pp. 31-42.

[Hatley87] Hatley, D. J., I. A. Pirbai (1987). *Strategies for Real Time System Specification*. Dorset House, New York.

[Heimdahl96] Heimdahl M. P. E., N. G. Leveson (1996): *Completeness and Consistency in Hierarchical State-Based Requirements*. IEEE Transactions on Software Engineering, Vol. 22 (6), pp. 363-377.

[Heninger80] Heninger, K. L. (1980): *Specifying Software Requirements For Compley Systems. New techniques and their applications*. IEEE Transactions on Software Engineering, Vol. 7 (5), pp. 510-18.

[Hopcroft90] Hopcroft, J. E., J. D. Ullman (1990): *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley Publishing Company.

[Hoare87] Hoare, C. A. R. (1987): *An Overview of some Formal Methods for Program Design*. In: C. A. R. Hoare, C. B. Jones: Essays in Computer Science, Prentice Hall, pp. 371-387.

[Hsia88] Hsia, P., A. Young (1988): *Another Approach to System Decomposition: Requirements Clustering*. In IEEE COMPSAC ´88, Washington D. C.: Com-

puter Society Press of the Institute of Electrical and Electronics Engineers, pp. 75-82.

[IEEE84] Institute of Electrical and Electronics Engineers (1984): *IEEE Guide to Software Requirements Specifications*. IEEE/ANSI Standard 830-1984, New York

[IEEE90] Institute of Electrical and Electronics Engineers (1990): *Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990, IEEE Computer Society Press.

[IEEE93] Institute of Electrical and Electronics Engineers (1993): *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Standard 830-1993, IEEE Computer Society Press.

[Jacobson94] Jacobson, I., Christerson, M., Jonsson, P, Övergaard, G. (1994): *Object-Oriented Software Engineering – A Use Case Driven Approach*.. Addison-Wesley Publishing Company.

[Jackson83] Jackson, M. (1983): *System Development*. C. A. R. Hoare series, Prentice-Hall 1983.

[Joos97] Joos, S., S. Berner, M. Arnold, M. Glinz (1997): *Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen*. Softwaretechnik-Trends, Vol. 17 (1), pp. 29-37.

[Joos98] Joos, S., S. Berner, M. Glinz(1998): *Stereotypen und ihre Verwendung in objektorientierten Modellen – eine Klassifikation*. In: K. Pohl, A. Schürr, G. Vossen (Hrsg.) Proceedings Modellierung ´98, Universität Münster, pp. 111-116.

[Kotonya98] Kotonya, G., I. Sommerville (1998): *Requirements Engineering: Processes and Techniques*. Chichester, etc.: John Wiley & Sons.

[Krogdahl93] Krogdahl, S., K. A. Ohlsen (1993): *Object Oriented Programming in the Beat Programming Language*, ACM Press, New York.

[Kyng91] Kyng, M. (1991): *Designing for Cooperation: Cooperating in Design*. Communication of the ACM, Vol. 34 (12), pp. 65-73.

[Lehman80] Lehman, M.M. (1980): *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE 68, 9, 1060-1076. (Nachgedruckt als Kapitel 19 in Lehman und Belady 1985).

[Lehman85] Lehman, M. M., L.A. Belady (Hrsg.) (1985): *Program Evolution: Processes of Software Change*. London, etc.: Academic Press.

[Leveson94] Leveson, N. G., M. P. E. Heimdahl, H. Hildreth, J. D. Reese (1994): *Requirements Specification for Process-Control Systems*. IEEE Transactions on Software Engineering, Vol. 20 (9), September 1994, pp. 684-707.

[Macaulay96] Macaulay, L. A. (1996): *Requirements Engineering*. Springer-Verlag Berlin

[Martin91] Martin, J. (1991): *Rapid Application Development*. Macmillan Publishing Company, New York.

[McDavid96] McDavid, D. W. (1996): *Buissiness Language Analysis for Object-Oriented Information Systems*. In IBM Systems Journal, Vol. 35 (2), pp. 128-150.

[McGrew89] McGrew, P. C., W. D. McDaniel (1989): *Online text management: hypertext and other techniques*. Publisher :McGraw-Hill New York, NY.

[McMenamin84] McMenamin, S. M., J. F. Palmer (1984): *Essential Systems Analysis*. Yourdon Press, New York.

[Mumford84] Mumford, E., G. Welter (1984): *Benutzerbeteiligung bei der Entwicklung von Computersystemen*. Erich Schmidt Verlag, Berlin.

[NASA76] National Aeronautics and Space Administration (1976): *NASA Software Specification and Evaluation Final Report*. NASA Report SMAP-DID-P200-SW.

[Naur76] Naur, P., B. Randell, J.N. Buxton (Hrsg.) (1976): *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*. (Neuausgabe der Proceedings der NATO-Konferenz von 1968 und der Nachfolgekonferenz von 1969), New York: Petrocelli.

[Östereich98] Östereich, B. (1998): *Objektorientierte Software-Entwicklung: Analyse und Design mit der Unified Modeling Language*. 4. aktualisierte Auflage, R. Oldenburg Verlag München.

[Ortner96] Ortner, E., B. Schienmann (1996): *Normsprachlicher Entwurf von Software-Systemen – Vorstellung einer Methode*. In: E. Ortner, B. Schienmann , H. Thoma (Hrsg.): Natürlichsprachlicher Entwurf von Informationssystemen – grundlagen, Methode, Werkzeuge, Anwendungen, GI-Workshop, Tutzing, 28.-30- Mai, pp. 109-129.

[Ortner98] Ortner, E. (1998): *Normalsprachliche Entwicklung von Informationssystemen*.. In: K. Pohl, A. Schürr, G. Vossen (Hrsg.) Proceedings Modellierung ´98, Universität Münster, pp. 19-23.

[Paech94] Paech B., B. Rumpe (1994): *A new Concept of Refinement used for Behaviour Modelling with Automata*. FME'94: Industrial Benefit of Formal Methods. Editors: Maurice Naftalin, Tim Denvir, Miquel Bertran. LNCS 873. Springer-Verlag, Berlin, pp. 42-53.

[Parnas72] Parnas, D.L. (1972): *On the Criteria To Be Used in Decomposing Systems into Modules*. Communications of the ACM Vol. 15 (12), pp. 1053-1058.

[Petri62] Petri, C. A. (1962): *Kommunikation mit Automaten*. Dissertation Universität Bonn.

[Petri82] Petri, C. A. (1982): *Petri-Netze – Eine Einführung*. Berlin-Heidelberg-New York-Tokyo- Springer.

[Pohl93] Pohl K. (1993): *The Three Dimensions of Requirements Engineering*. C. Rolland, F. Bodart, C. Cauvert (Hrsg.): Advanced Information Systems Engineering, CAiSE´93, Paris, Lecture Notes in Computer Science, Vol. 685, Springer, Berlin, pp. 275-292.

[Pullman94] Pullman, S. G. (1994): *Natural Language Processing and Requirements Specification*. Präsentation beim Prolog Forum, Institut für Informatik, Universität Zürich (Februar 1994).

[Ross77] Ross, D. T. (1977): *Structured Analysis. A Language for Communicating Ideas*. IEEE Trans. Software Engineering, Vol. 3 (1), pp. 34-49.

[Royce70] Royce, W. (1970): *Managing the Development of Large Software Systems*. Proceedings IEEE WESCON. 1-9. (Nachgedruckt in Proceedings 9th International Conference on Software Engineering, Monterey, 1987, pp. 328-338.

[Rumbaugh91] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991): *Object-Oriented Modeling and Design*. Englewood Cliffs, N. J.: Prentice Hall.

[Sang94] Sang, X, Osterweil, (1994): *Experience with an Approach to Comparing Software Design Methodologies*. IEEE Transactions on Software Engineering, Vol. 20 (5), pp. 364-384.

[Schmid98] Schmid, R., S. Berner, S. Joos, R. Reutemann, J. Ryser (1998): *Spezifikation des Referenzmodells*. Arbeitsdokument des AK´s Simulation für das RE, GI-Fachgruppe 2.1.6 , Version 1.3 (draft), *http://www.ifi.unizh.ch/groups/req/ak_simulation/reference_model/ReferenceModel.pdf*

[Schneider93] Schneider, Kurt (1993): *Modellierung und Simulation*. Fachpraktikum Modellierung & Simulation. Institut für Informatik, Universität Stuttgart.

[Schwitter98] Schwitter R. (1998): *Kontrolliertes Englisch für Anforderungsspezifikationen*.. Dissertation an der Universität Zürich, Institut für Informatik.

[Sedgewick92] Sedgewick R. (1992): *Algorithmen*. Addison-Wesley Publishing Company, Wokingham, England.

[Selic94] Selic, B., G. Gullekson, P. T. Ward (1994): *Real-Time Object-Oriented Modeling*. J. Wiley & Sons, Inc., New York.

[Sommerville96] Sommerville, I. (1996): *Software Engineering*. Fifth Edition, Addison-Wesley Publishing Company, Wokingham, England.

[Sowa92] Sowa, J. F. (1992): *Logical Structures in the Lexicon*. In: Knowledge Based Systems, 5, 3, pp. 173-182.

[Stachowiak73] Stachowiak, H. (1973): *Allgemeine Modelltheorie*. Springer-Verlag, Wien, New York.

[Stein94] Stein, W. (1994): *Objektorientierte Analysemethoden: Vergleich, Bewertung, Auswahl*. Mannheim [u.a.]: BI-Wissenschaftsverlag.

[Tavolato84] Tavolato, P., K. Vincena (1984): *A Prototyping Methodology and Its Tool*. In Approaches to Prototyping, R. Budde et al. (Hrsg.), Springer-Verlag, Berlin, pp. 434-445.

[Ungar94] Ungar, D., Smith R. B. (1994): *Self: The Power of Simplicity.* SMLI-TR-94-30, SUN Microsystems Laboratories Inc., Mountain View, CA.

[Ward85] Ward, P. T., S. J. Mellor (1985): *Structured Development for Real-Time Systems*. Vol. I, II, III. Prentice-Hall, Englewood Cliffs, N. J.

[Weber98] Weber W., P. Metz (1998): *Reuse of Models and Diagrams of the UML and Implementation Concepts Regarding Dynamic Modeling* . M. Schader, A. Korthaus, University of Mannheim, Germany (Hrsg.):, The Unified Modeling Language - Technical Aspects and Applications Physica-Verlag, Heidelberg, pp. 78-88.

[Wilson97] Wilson, W. M. (1998): *Writing Effective Requirements Specifications.* Proceedings of the STC´97, Software Technology Conference, Utah, April 1997, http://satc.gsfc.nasa.gov/SATC/PAPERS/WRITERT/writert.html

[Wirth86] Wirth, N. (1986): *Algorithmen und Datenstrukturen*. Teubner, Stuttgart, 3. Auflage.

[Wirfs93] Wirfs-Brock, R., B. Wilkerson, L. Wiener (1993): *Designing Object-Oriented Software*. Prentice Hall International.

[Wirfs94] Wirfs-Brock, R., B. Wilkerson, L. Wiener (1994): *Responsibility-Driven Design: Adding To Your Conceptual Toolkit.* ROAD, Vol. 1 (2), 27-34.

[Woodcock96] Woodcock, J., J. Davies (1996): *Using Z : Specification, Refinement and Proof*. Prentice-Hall, London.

[Yeh80] Yeh, R. , P. Zave (1980): *Specifying Software Requirements*. Proceedings of the IEEE Vol. 68 (9), pp. 1077-1085.

[Yourdon78] Yourdon, E. N., L. L. Constantine (1978): *Structured Design*. Prentice Hall, Englewood Cliffs, N. J.

[Yourdon89] Yourdon, E. N. (1989): *Modern Structured Analysis*. Prentice-Hall International Englewood Cliffs, NJ.

[Zehnder89] Zehnder, C. A. (1989): *Informationssysteme und Datenbanken*. 5. Auflage, Stuttgart, vdf und B. G. Teubner.

Anhang C: Index

F

G

H

I

J

K

M

N

O

P

R

S

T

U

V

W

Z

Lebenslauf

Name:	Stefan Maximilian Joos
Geburtstag:	30.01.1968 in Stuttgart
Staatsangehörigkeit:	deutsch

Schulausbildung:

1974-1987	Grundschule und Gymnasium in Stuttgart, Abschluß: Abitur

Berufsausbildung:

7/1987-10/1988	Grundwehrdienst in Ulm als Nachschub-Buchführer in einer EDV-Abteilung
10/1988-12/1993	Studium der Informatik mit Nebenfach technische Kybernetik/ Regelungstechnik an der Universität Stuttgart; Abschluß als «Diplom-Informatiker»; Tutorium am Institut für Informatik, Stuttgart
3/1994-3/1999	Doktorand an der Universität Zürich, Institut für Informatik (Forschungsgruppe Requirements Engineering) bei Prof. Dr. Martin Glinz
seit 4/1999	Mitarbeiter der Robert Bosch GmbH, Stuttgart-Feuerbach, Unternehmensbereich Automobiltechnik, Abteilung «Entwicklung EDC-Konzepte und Sensoren, Applikationsunterstützung»

Stuttgart, den 14.4.2000