

Lehrbücher der Informatik

Heide Balzert

# Lehrbuch der Objektmodellierung

Analyse und Entwurf



Spektrum  
AKADEMISCHER VERLAG

**Balzert, Heide**

## Lehrbuch der Objektmodellierung

**Analyse und Entwurf**

*1999, 600 S., inkl. CD-ROM, Geb.  
DM 98,- /öS 716,- /sFr 89,-  
ISBN 3-8274-0285-9  
Lehrbuch*



### Inhalt

Dieses praxisnahe, zweifarbig gestaltete Lehrbuch bietet auf 593 Seiten eine leicht verständliche Einführung in die Objektmodellierung. Ein erprobtes didaktisches Konzept mit Lehreinheiten, Lernzielen, ausführlichen Aufgaben und detaillierten Lösungen macht es zum idealen Lehr- und Arbeitsbuch für Studenten und alle Lernenden auf dem Gebiet der Objektorientierung. Zur Lernkontrolle stehen 70 Aufgaben mit einer Bearbeitungszeit von ca. 17 Stunden zur Verfügung. Das umfangreiche Glossar mit mehr als 170 Begriffen, über 100 Literaturhinweise und ein Index unterstützen den Einsatz als Nachschlagewerk.

- UML-Notation
- Konzepte der objektorientierten Analyse (OOA)
- Analysemuster
- Beispielanwendungen aus der Praxis
- Checklisten zur Erstellung und Beurteilung eines OOA-Modells
- Gestaltung von Benutzungsoberflächen
- Konzepte des objektorientierten Entwurfs (OOD)
- Entwurfsmuster
- Relationale Datenbanksysteme und objekt-relationale Abbildung
- Objektorientierte Datenbanksysteme (ODMG)
- Verteilte objektorientierte Anwendungen (CORBA)
- Drei-Schichten-Architektur

# Lehrbuch der Objektmodellierung

# Lehrbücher der Informatik

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

Helmut Balzert  
Lehrbuch der Software-Technik  
Software-Entwicklung

Helmut Balzert  
Lehrbuch der Software-Technik  
Software-Management  
Software-Qualitätssicherung  
Unternehmensmodellierung

Helmut Balzert  
Lehrbuch Grundlagen der Informatik  
Konzepte und Notationen in UML, Java und C++  
Algorithmik und Software-Technik

Zu diesen Bänden sind jeweils CD-ROMs mit den Inhalten der  
Bücher als PowerPoint-Präsentationen zum Einsatz in Vorlesungen,  
Schulungen und Seminaren erhältlich.  
Weitere Informationen finden Sie unter  
<http://www.spektrum-verlag.com>

Heide Balzert

# Lehrbuch der Objektmodellierung

Analyse und Entwurf

mit CD-ROM

Spektrum Akademischer Verlag Heidelberg • Berlin

*Autorin:*  
Prof. Dr. Heide Balzert  
Fachhochschule Dortmund  
Fachbereich Informatik  
e-mail: balzert@fh-dortmund.de  
<http://www.inf.fh-dortmund.de/balzert>

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

**Balzert, Heide:**

Lehrbuch der Objektmodellierung : Analyse und Entwurf / Heide Balzert. –  
Heidelberg ; Berlin : Spektrum, Akad. Verl., 1999  
(Lehrbücher der Informatik)  
ISBN 3-8274-0285-9

Titelbild: Wolfgang Nocke: »Durchblick« (1989)

Diesem Buch ist eine CD-ROM mit Informationen, Demonstrationen, Animationen, begrenzten Vollversionen und Vollversionen von Software-Produkten beigelegt. Der Verlag und die Autorin haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch und der beiliegenden CD-ROM zu publizieren.

Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder ähnliches zurückzuführen sind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich.

Der Verlag übernimmt keine Gewähr dafür, daß die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

© 1999 Spektrum Akademischer Verlag GmbH Heidelberg · Berlin

Alle Rechte, insbesondere die der Übersetzung in fremde Sprachen, sind vorbehalten. Kein Teil des Buches darf ohne schriftliche Genehmigung des Verlages photokopiert oder in irgendeiner anderen Form reproduziert oder in eine von Maschinen verwendbare Form übertragen oder übersetzt werden.

Lektorat: Dr. Georg W. Botz / Bianca Alton (Ass.)  
Herstellung: Katrin Froberg  
Gesamtgestaltung: Gorbach Büro für Gestaltung und Realisierung,  
Buchendorf  
Satz: Hagedorn Kommunikation, Viernheim  
Druck und Verarbeitung: Franz Spiegel Buch GmbH, Ulm

# Vorwort

In den letzten Jahren hat die Objektorientierung innerhalb der Softwaretechnik eine stürmische Entwicklung durchgemacht. Sie ist ihren Kinderschuhen längst entwachsen und stellt heute den Stand der Technik in der Softwareentwicklung dar. Die objektorientierten Sprachen C++ und Java sind weit verbreitet, objektorientierte Klassenbibliotheken und *Frameworks* für die verschiedensten Anwendungsbereiche drängen verstärkt auf den Markt und als grafische Standard-Notation etabliert sich die UML (*Unified Modeling Language*). Die Objektmodellierung, d.h. die objektorientierte Analyse und der objektorientierte Entwurf, ist ein Thema, mit dem sich jeder Softwareentwickler früher oder später auseinandersetzen muß.

Um Ihnen, liebe Leserin, lieber Leser, einen optimalen Einstieg in die Objektmodellierung zu ermöglichen, habe ich dieses Lehr- und Lernbuch geschrieben. Wenn Sie bereits Kenntnisse in einer objektorientierten Programmiersprache besitzen, dann tun Sie sich sicher etwas leichter mit dieser Lektüre. Aber auch, wenn sie bisher »nur« klassisch strukturiert entwickelt haben, bietet dieses Buch eine leicht verständliche Einführung in die Objektmodellierung.

Objektorientierte Entwicklung umfaßt mehr als Kenntnisse über UML-Diagramme und C++- oder Java-Programme. Objektorientierte Modellierung bedeutet, daß Sie

- die objektorientierten Konzepte in den Phasen Analyse und Entwurf anwenden können,
- die objektorientierten Konzepte mit der Standardnotation UML beschreiben können,
- wissen, wie Sie am besten beim Erstellen objektorientierter Modelle vorgehen und wie Sie gute von schlechten Modellen unterscheiden können,
- die objektorientierten Konzepte in C++ und Java umsetzen können,
- die Phasen Analyse, Entwurf und Implementierung präzise voneinander trennen können,
- Drei-Schichten-Architekturen systematisch entwickeln können,
- im Entwurf die GUI- und die Datenhaltungsschicht über standardisierte Schnittstellen mit Ihrem Fachkonzept verbinden können,
- nicht nur objektorientierte, sondern auch relationale Datenbanken systematisch anbinden können,
- *Frameworks* und Muster (*patterns*) selbstverständlich anwenden können.

## Vorwort

Dieses Lehrbuch ist geschrieben für:

- Studierende im Haupt- und Nebenfach der Informatik an Universitäten, Fachhochschulen und Berufsakademien.
- Studierende, die im Rahmen eines größeren Projekts – z.B. im Rahmen einer Diplomarbeit – objektorientierte Software entwickeln wollen.
- Softwareentwickler und Analytiker in der Praxis, die zukünftig objektorientiert entwickeln wollen.

Beim Arbeiten mit diesem Buch sollten Sie zunächst die Lehreinheiten der Reihe nach durcharbeiten. Bei der späteren Arbeit im Projekt soll es Ihnen als Nachschlagewerk dienen. Die Didaktik orientiert sich an den objektorientierten Konzepten und an der methodischen Vorgehensweise und *nicht* an den diversen Diagrammen der UML.

Behandelte Gebiete Das Lehrbuch gliedert sich in folgende Teile:

- Einführung (1 Lehreinheit),
- Konzepte und UML für die Analyse (3 Lehreinheiten),
- Analysemuster (1 Lehreinheit),
- Methodische Vorgehensweise in der Analyse (3 Lehreinheiten),
- Einführung in die Gestaltung von Benutzungsoberflächen (2 Lehreinheiten),
- Konzepte und UML für den Entwurf (2 Lehreinheiten),
- Entwurfsmuster (1 Lehreinheit),
- Einführung in Datenbanksysteme (2 Lehreinheiten),
- Einführung in Client-Server-Anwendungen (1 Lehreinheit),
- Entwurf einer Drei-Schichten-Architektur (2 Lehreinheiten).

Einführung Die erste Lehreinheit beschreibt das Rahmenwerk für die weiteren Lehreinheiten. Sie führt viele Begriffe ein, die in den späteren Lehreinheiten ausführlich erläutert werden.

Leser, die sich auf die objektorientierte Analyse beschränken wollen, benötigen nur die Lehreinheiten eins bis zehn und müssen sich nicht mit den Details des Entwurfs belasten.

Konzepte und UML in der Analyse Die Lehreinheiten zwei bis vier führen in die objektorientierten Konzepte und in die Notation mittels UML ein, die Sie in der Analysephase benötigen. Jedem Konzept ist ein Kapitel gewidmet, in dem Sie alle Informationen zum Konzept und zur Notation mittels UML finden. Ein Konzept wird zunächst definiert und dann durch ein Beispiel erläutert. Anschließend wird das jeweilige Konzept genauer erläutert. Am Ende des Kapitels werden verwandte Begriffe angegeben.

Die zweite Lehreinheit führt zunächst in die Konzepte Objekt, Klasse, Attribut und Operation ein. Aufbauend darauf werden in der dritten Lehreinheit die Konzepte Assoziation, Vererbung und Paket eingeführt. Anschließend können Sie bereits einfache Klassendiagramme erstellen und beliebige Klassendiagramme lesen. Um das dynamische Verhalten eines Softwaresystems zu beschreiben,



werden in der vierten Lehreinheit die Konzepte Geschäftsprozeß, Botschaft, Szenario und Zustandsautomat eingeführt. Damit Ihre Ausbildung in der Objektmodellierung eine sichere Investition darstellt, ist es wichtig, die objektorientierten Konzepte zu verstehen. Deren Darstellung in einer beliebigen Notation ist dann relativ einfach. Daher führe ich in diesen Lehreinheiten zuerst die Konzepte ein und gehe dann auf die spezifische Notation laut UML ein.

Ein ganz wichtiges Thema innerhalb der Objektmodellierung stellen heute die Muster dar. Das sind vor allem Entwurfsmuster, aber auch Analysemuster. In der Lehreinheit fünf werden zehn Analysemuster vorgestellt. Anschließend können Sie ihre Kenntnisse auf einige Beispielanwendungen anwenden. Dabei handelt es sich um OOA-Modelle, die ihren Ursprung in praktischen Projekten haben und die zeigen, wie reale Probleme objektorientiert modelliert werden.

Analysemuster,  
OOA-Modelle

Die Lehreinheiten sechs bis acht unterstützen Sie darin, ein gutes OOA-Modell zu erstellen und gute von schlechten Modellen unterscheiden zu können. Um ihnen das Nachschlagen zu erleichtern, ist dieser Lehrstoff in Form von Checklisten strukturiert. Für jedes objektorientierte Konzept gibt es eine Checkliste, in der alles Wissenswerte übersichtlich im Zugriff ist. In der Lehreinheit sechs wird zunächst der empfohlene Analyseprozeß beschrieben. Dann folgen die Checklisten für Geschäftsprozesse und Pakete, die sich unter dem Aspekt »Analyse im Großen« zusammenfassen lassen. Die Lehreinheit sieben enthält die Checklisten für die Konzepte Klasse, Assoziation, Attribut und Vererbung. Hier erfahren Sie, auf was Sie bei der Erstellung eines Klassendiagramms achten müssen. In der achten Lehreinheit finden Sie die Checklisten zu den Konzepten Szenario, Zustandsautomat und Operation, mit denen Sie die Diagramme des dynamischen Modells systematisch und konsistent mit dem Klassendiagramm erstellen können. Außerdem führt diese Lehreinheit in das Verfahren der Formalen Inspektion ein. Das ist ein manuelles Prüfverfahren, um einen Großteil der Fehler zu finden, die in der Analyse auch trotz moderner Entwicklungsmethoden gemacht werden.

Analysemethode

Ein großer Vorteil der objektorientierten Entwicklung ist, daß die objektorientierten Analysemodelle mittels objektorientierter Benutzungsoberflächen visualisiert werden können. Der hier beschriebene Entwicklungsprozeß geht davon aus, daß Sie aus Ihrem OOA-Modell einen Prototypen der Benutzungsoberfläche ableiten. Die Einheiten neun und zehn enthalten daher so viel Lehrstoff aus dem Gebiet der Software-Ergonomie, daß Sie einen solchen Prototypen erstellen können.

Objektorientierte  
Benutzungs-  
oberflächen

Die Lehreinheiten 11 und 12 führen in die Konzepte des objektorientierten Entwurfs ein und verwenden ebenfalls die UML als Notation. Um eine kompakte Darstellung zu ermöglichen, werden nur

Konzepte und  
UML im Entwurf

## Vorwort

noch diejenigen Elemente aufgeführt, die im Entwurf zusätzlich zur Analyse benötigt werden. Um einen problemlosen Übergang in die Implementierung zu demonstrieren, wird gezeigt, wie alle Konzepte auf die Programmiersprachen C++ und Java abgebildet werden können.

Entwurfsmuster Wie bereits erwähnt, nehmen Entwurfsmuster bei der objektorientierten Entwicklung heute einen besonderen Stellenwert ein. In der Lehreinheit 13 werden daher sieben Entwurfsmuster aus dem Standardwerk der *Gang of Four* vorgestellt. In späteren Lehreinheiten wird gezeigt, wie diese Muster beim Entwurf systematisch angewendet werden.

Anbindung an relationale Datenbanken Die meist verwendeten Datenbanken sind zur Zeit die relationalen Datenbanken. Die Lehreinheit 14 führt daher zunächst kurz in diese Thematik ein und zeigt anschließend, wie das Klassendiagramm auf Tabellen einer relationalen Datenbank abgebildet werden kann.

Objektorientierte Datenbanken Objektorientierte Datenbanken bilden eine optimale Ergänzung zur objektorientierten Entwicklung. Die Lehreinheit 15 führt in diese Datenbanken ein und zeigt deren typische Eigenschaften anhand des ODMG-Standards.

Verteilte objektorientierte Anwendungen Client-Server-Anwendungen sind heute ein Standardfall bei der Softwareentwicklung. Daher führt die Lehreinheit 16 in die Thematik der verteilten objektorientierten Anwendungen ein.

Entwurf mittels Drei-Schichten-Architektur Moderne Systeme sollten mittels einer Drei-Schichten-Architektur entworfen werden. Die Lehreinheiten 17 und 18 beschäftigen sich nicht nur mit den grundlegenden Architekturen, sondern hier wird ganz präzise vorgeführt, wie Sie eine solche Drei-Schichten-Architektur Schritt für Schritt entwerfen. Die Lehreinheit 17 zeigt, wie das OOA-Modell systematisch in den Entwurf transformiert wird, wie die GUI-Schicht entworfen und mit dem fachlichen Konzept verbunden wird. Die Lehreinheit 18 geht darauf ein, wie die Datenhaltung wahlweise mit einer objektorientierten Datenbank, einer flachen Dateiverwaltung und einer relationalen Datenbank entworfen wird. Die hier beschriebenen Entwurfsarchitekturen sind weitestgehend unabhängig von einer speziellen Entwicklungsumgebung gehalten. Zusätzlich sind konkrete Implementierungen in C++ auf der CD-ROM enthalten.

Ein wesentliches Ziel dieses Buchs ist, daß Sie nicht nur wissen, wie *man* objektorientierte Software entwickelt, sondern daß *Sie selbst* objektorientiert entwickeln können. Daher wurde dieses Buch um zwei praktische Exkurse erweitert.

Prototyp der Benutzungsoberfläche Der erste Exkurs zeigt, wie Schritt für Schritt ein Prototyp der Benutzungsoberfläche entwickelt wird. Wegen seiner großen Verbreitung habe ich mich für die Verwendung des *Microsoft Visual Studio* entschlossen.

Der zweite Exkurs zeigt, wie die Datenhaltung mit einer objektorientierten Datenbank realisiert werden kann. Ich habe mich für *Poet* entschieden, weil diese Datenbank relativ günstig zu erwerben und der Umgang einfach zu erlernen ist. Beide Exkurse sind auch für den Einsatz in Praktika geeignet, die eine Vorlesung auf dem Gebiet der objektorientierten Software-Technik begleiten.

Datenhaltung  
mit Poet

Die Exkurse beziehen sich auf das Fallbeispiel, das in Anhang 1 durchgängig beschrieben ist. Ein Beispielprojekt dieser Größenordnung ist insbesondere als Vorlage für eine Praktikumsarbeit gedacht, die den Lehrstoff des Buchs praktisch umsetzt. Auf der beiliegenden CD-ROM befindet sich dessen Implementierung in C++ mit der Anbindung an *Poet*.

Durchgängiges  
Fallbeispiel

Anhang 2 enthält detaillierte Lösungen zu allen Aufgaben. Sie enthalten zusätzliche Erläuterungen und die Antworten von Wissens-Aufgaben versuchen bewußt, etwas andere Perspektiven als in den Lehreinheiten zu beschreiben.

Lösungen

Das Gesamtglossar erleichtert ein schnelles Nachschlagen aller wichtigen Begriffe unabhängig von einer Zuordnung zu den Lehreinheiten. Insbesondere wird hier sichtbar, daß von einigen Begriffen mehrere Interpretationen existieren, die jeweils untereinander aufgeführt sind.

Gesamtglossar

Um Ihnen, liebe Leserin, lieber Leser, das Lernen optimal zu erleichtern, werden folgende methodisch-didaktischen Elemente benutzt:

Methodisch-  
didaktische  
Elemente

- Dieses Buch ist in 18 Lehreinheiten und zwei Exkurse (für jeweils eine Vorlesungsdoppelstunde) gegliedert.
- Jede Lehreinheit ist unterteilt in Lernziele, Voraussetzungen, Inhaltsverzeichnis, Text, Glossar, Zusammenhänge und Aufgaben.
- Zusätzlich sind die Themen nach fachlichen Gesichtspunkten in Kapitel gegliedert.
- Mehr als 170 Begriffe sind im Glossar definiert.
- Mehr als 100 Literaturangaben verweisen auf referenzierte und weiterführende Literatur.
- Zur Lernkontrolle stehen 70 Aufgaben zur Verfügung.
- Zu jeder Aufgabe gibt es eine Zeitangabe, die hilft, das Zeitbudget zu planen und die eigene Leistung zu kontrollieren. Zur Lösung aller Aufgaben werden rund 17 Stunden benötigt.
- Es wurde eine Typographie mit Marginalienspalte und Piktogrammen verwendet.
- Als Schrift wurde Lucida ausgewählt, die für dieses Lehrbuch besonders gut geeignet ist, da sie auch eine nichtproportionale Schrift für Programme zur Verfügung stellt.
- Das Buch ist durchgehend zweifarbig gestaltet.
- Zur Veranschaulichung enthält es mehr als 300 Abbildungen.
- Wichtige Inhalte sind zum Nachschlagen in Boxen angeordnet.

## Vorwort

Durch diese moderne Didaktik kann das Buch zur Vorlesungsbegleitung, zum Selbststudium und zum Nachschlagen verwendet werden.

Auf der beigefügten CD-ROM befinden sich Demoversionen und eingeschränkte Vollversionen von

- CASE-Werkzeugen zur Objektmodellierung,
- Java-Programmierumgebungen und dem
- Janus-Generator.

Desweiteren enthält sie

- die Implementierung des durchgängigen Fallbeispiels,
- Programme wichtiger Entwurfsarchitekturen und
- ein multimediales CBT (*Computer Based Training*) zur Objektmodellierung.

**blaue Schrift** Zur Vermittlung der Lerninhalte werden zahlreiche Beispiele verwendet. Um dem Leser diese unmittelbar kenntlich zu machen, sind sie in blauer Schrift gesetzt.

**Unter der Lupe** Für den Leser, der in die Tiefe eindringen möchte, werden ab und zu noch Informationen angeboten, die mit dem Piktogramm »Unter der Lupe« gekennzeichnet sind.

**Begriffe, Glossar**  
**halbfett, blau** Für dieses Lehrbuch habe ich sorgfältig überlegt, welche Begriffe eingeführt und definiert werden. Ziel ist es, einerseits die Anzahl der Begriffe möglichst gering zu halten und andererseits alle wichtigen Begriffe einzuführen. Blau markierte Begriffe sind am Ende einer Lehreinheit in einem Glossar alphabetisch angeordnet und definiert. Dabei wurde oft versucht, die Definition etwas anders abzufassen, als es im Text der Fall war, um dem Lernenden noch eine andere Sichtweise zu vermitteln. In Anhang 3 befindet sich zusätzlich ein alphabetisch sortiertes Gesamtverzeichnis. Halbfett gesetzte Begriffe sind zwar hervorgehoben, aber nicht im Glossar definiert.

**Zusammenhänge** Damit sich der Lernende eine Zusammenfassung der jeweiligen Lehreinheit ansehen kann, werden nach dem Glossar nochmals die Zusammenhänge in aller Kürze verdeutlicht.

**Aufgaben** Der Lernende kann nur durch das eigenständige Lösen von Aufgaben überprüfen, ob er die Lernziele erreicht hat. Vor jeder Aufgabe wird das Lernziel zusammen mit der Zeit, die zur Lösung dieser Aufgabe benötigt werden sollte, angegeben. Das ermöglicht es dem Lernenden, seine Zeit einzuteilen. Außerdem zeigt ihm ein massives Überschreiten dieser Zeit an, daß er die Lehrinhalte nicht voll verstanden hat. Alle Zeitangaben wurden durch Studenten evaluiert, wobei in der Regel der von den Studenten angegebene Zeitbedarf von mir etwas nach oben korrigiert wurde.

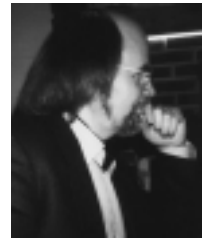
Um das selbständige Lernen zu unterstützen, sind alle Lösungen verfügbar. Bei denjenigen Aufgaben, in denen das Wissen und Verständnis abgefragt werden, wurde bei den Lösungen Wert darauf gelegt, eine andere Formulierung als in der Lehreinheit zu wählen. Bei vielen Aufgaben, in denen der gelernte Stoff angewendet wird,

sind in den Lösungen Begründungen für den Lösungsweg angegeben.

Ein Problem für ein Informatikbuch stellt die Verwendung englischer Begriffe dar. Da die Wissenschaftssprache der Informatik Englisch ist, gibt es für viele Begriffe – insbesondere in Spezialgebieten – keine oder noch keine geeigneten oder üblichen deutschen Fachbegriffe. Auf der anderen Seite gibt es jedoch für viele Bereiche der Informatik sowohl übliche als auch sinnvolle deutsche Bezeichnungen, z.B. Entwurf für *Design*. Da mit einem Lehrbuch auch die Begriffswelt beeinflusst wird, bemühe ich mich in diesem Buch, sinnvolle und übliche deutsche Begriffe zu verwenden. Da gerade auf dem Gebiet der Objektmodellierung der größte Teil der Literatur englischsprachig ist, wird in Klammern und kursiv der englische Begriff hinter dem deutschen Begriff aufgeführt. Gibt es noch keinen eingebürgerten deutschen Begriff, dann wird der englische Originalbegriff verwendet. Englische Bezeichnungen sind immer kursiv gesetzt, so daß sie sofort ins Auge fallen.

Objektmodellierung muß man sowohl theoretisch als auch praktisch begreifen. Das bedeutet, daß eigene Anwendungen durch Analysemodelle dargestellt werden. Um diesen Einstieg zu erleichtern, habe ich in der fünften Lehreinheit vier Beispielanwendungen aus verschiedenen Anwendungsbereichen ausführlich beschrieben. Der Exkurs 1 vermittelt den praktischen Hintergrund, um einen Prototypen der Benutzungsoberfläche erstellen zu können. Um die Entwurfskonzepte und die Drei-Schichten-Architektur zu verstehen, ist es meines Erachtens unumgänglich, die Entwurfsmodelle in einer objektorientierten Programmiersprache zu implementieren. Daher wurden die wichtigsten Architekturen exemplarisch in C++ realisiert. Beim Einsatz des Buch in einer Vorlesung sollte in einem begleitenden Praktikum ein kleines objektorientiertes System entwickelt werden. Als Vorlage kann das durchgängige Fallbeispiel in Anhang 1 dienen.

Ein Buch soll nicht nur vom Inhalt her gut sein, sondern Form und Inhalt sollten übereinstimmen. Daher wurde auch versucht, die Form anspruchsvoll zu gestalten. Ich freue mich darüber, daß der bekannte Buchgestalter und Typograph Rudolf Gorbach aus München die Aufgabe übernommen hat, diese Lehrbuchreihe zu gestalten. Da ich ein Buch als »Gesamtkunstwerk« betrachte, ist auf der Buchtitelseite ein Werk des Künstlers Wolfgang Nocke mit dem bezeichnenden Titel »Durchblick« abgedruckt. Entgegen den traditionellen Inhalten bedeutet bei Nocke »dunkel« nicht »schlecht«. Vielmehr steht die Nacht für die Entfaltung der Kreativität. Seine Motive entstammen einer abstrakten Welt ohne unreal zu sein. Obwohl dieses Bild anders als bei den anderen Büchern dieser Reihe nicht mit dem Computer erstellt wurde, habe ich mich bewußt für Wolfgang Nocke entschieden. Er versteht es auf faszinierende Art, abstrakte



**Rudolf Paulus Gorbach**

\*1939, nach der Schulzeit Buchdrucker und Musiker, dann Buchdruckmeister; Studium Drucktechnik und Typographie in Berlin; Hersteller und Herstellungsleiter in Buchverlagen, seit 1971 eigenes Büro in München; Lehraufträge an den Universitäten Ulm, Osnabrück und an der FH München; Software-Marketing-Preis 1991



**Wolfgang Nocke**

\*1960 in Linnich/Kreis Aachen, Studium an der freien Kunstschule »Pyramide« in Düsseldorf, Privatstudium bei Prof. Dr. Ernst Fuchs in Wien, Studium an der Wiener Kunstschule. Seit 1984 als freischaffender Maler in Recklinghausen tätig. Seine Werke werden weltweit in zahlreichen Einzelausstellungen gezeigt.

## Vorwort

Begriffe und Themen aus Gebieten der Technik in lebensfrohe, beja-  
hende Kunstwerke umzusetzen. Seine Bilder besitzen eine aus-  
drucksstarke Visualisierung, die Abstraktes begreifbar macht.

Danksagungen

Ein so aufwendiges Werk ist ohne die Mithilfe von vielen Perso-  
nen nicht realisierbar. Mein besonderer Dank gebührt meinem Mann  
Prof. Dr. Helmut Balzert, der viele wichtige Anregungen zur Gestal-  
tung und zum Inhalt gegeben hat. Auch Herrn Prof. Dr. Ulrich  
Eisenecker von der Fachhochschule Heidelberg und Herrn Prof. Dr.  
Klaus Zeppenfeld von der der Fachhochschule Dortmund möchte  
ich an dieser Stelle für viele wertvolle Hinweise ganz herzlich dan-  
ken. Herrn Dr. Marc Gille danke ich für seine Anregungen zum  
ODMG-Standard und für die Überprüfung der ODMG-Beispiele.

Dem Spektrum Akademischer Verlag in Heidelberg danke ich für  
die hervorragende Zusammenarbeit. Als Lektor trug Dr. Georg Botz  
wesentlich zur sprachlichen Klarheit und guten Form des Buches  
bei. Frau Anja Scharl und Herr Oliver Dewald haben mit viel Geduld  
die zahlreichen Grafiken gezeichnet.

Als Lehrbuch richtet sich dieses Werk natürlich insbesondere an  
Lernende. Daher habe ich viel Wert darauf gelegt, Anregungen mei-  
ner Studenten zu erhalten und einzuarbeiten. In dieser Hinsicht hat  
mich besonders Herr Dipl.-Inform. Andreas Schröter unterstützt,  
dem ich für die ausführlichen Diskussionen danke. Herr Schröter  
hat alle Aufgaben in diesem Buch bearbeitet und die dafür benöti-  
gten Zeiten evaluiert. Außerdem hat er die Programme der Drei-  
Schichten-Architektur und das Fallbeispiel in Anhang 1 für mich  
programmiert.

Internet

Nichts ist so beständig ist wie die Veränderung. Das gilt ganz be-  
sonders für ein sich so dynamisch entwickelndes Gebiet wie die  
Software-Technik. Deshalb hat sich der Spektrum Akademische Ver-  
lag entschlossen, unter <http://www.software-technik.de> eine Web-  
Seite einzurichten. Hier werden regelmäßig Neuerungen veröffent-  
licht, die sich auf den Inhalt dieses Buch beziehen. Dazu gehören  
aktuelle Informationen zur UML, neue Beispielanwendungen, weite-  
re Architekturen, z.B. Programme zur Anbindung an relationale Da-  
tenbanken, neue Aufgaben mit Lösungen und viel Wissenswertes  
und Aktuelles zum Thema Objektmodellierung. Es lohnt sich also,  
öfter mal »reinzuschauen«.

Mailing-Liste

Wenn Sie regelmäßig über Neuerungen zu diesem Buch infor-  
miert werden möchten, können Sie sich auf dieser Web-Seite in eine  
*Mailing-Liste* eintragen.

»Wer glaubt, gut zu sein, hat aufgehört besser zu werden.« Ein  
Buch enthält trotz aller Anstrengungen immer noch Fehler und  
Verbesserungsmöglichkeiten. Kritik und Anregungen sind daher je-  
derzeit willkommen. Über Erfahrungsberichte meiner Leser freue  
ich mich ganz besonders. Die aktuelle *Mailing-Adresse* und eine ak-

tuelle Liste mit Korrekturen und Informationen zu diesem Buch und der beigefügten CD-ROM finden Sie ebenfalls unter

<http://www.software-technik.de>.

Über zwei Jahre Arbeit stecken in diesem Lehrbuch. Ihnen, liebe Leserin, lieber Leser, erlaubt es, die Objektmodellierung in wesentlich kürzerer Zeit zu erlernen. Ich wünsche Ihnen viel Spaß beim Lesen. Möge Ihnen dieses Buch und die objektorientierte Softwareentwicklung in Zukunft die Freude bringen, mehr Software besserer Qualität in kürzerer Zeit zu entwickeln.

Ihre

Heide Balzer





# Inhalt

<b>LE 1</b>	<b>1</b>	<b>Objektorientierte Softwareentwicklung</b>	<b>1</b>
	1.1	Einführung und Überblick	2
	1.2	Objektorientierte Analyse	8
	1.3	Objektorientierter Entwurf	11
	<b>2</b>	<b>Konzepte und Notation der objektorientierten Analyse</b>	<b>17</b>
<b>LE 2</b>		<b>Basiskonzepte</b>	<b>17</b>
	2.1	Objekt	18
	2.2	Klasse	21
	2.3	Attribut	25
	2.4	Operation	30
<b>LE 3</b>		<b>Statische Konzepte</b>	<b>39</b>
	2.5	Assoziation	40
	2.6	Vererbung	51
	2.7	Paket	55
<b>LE 4</b>		<b>Dynamische Konzepte</b>	<b>61</b>
	2.8	Geschäftsprozeß	62
	2.9	Botschaft	69
	2.10	Szenario	70
	2.11	Zustandsautomat	78
<b>LE 5</b>	<b>3</b>	<b>Anlysemuster und Beispiel-Anwendungen</b>	<b>89</b>
	3.1	Katalog von Anlysemustern	90
	3.2	Beispiel Materialwirtschaft	98
	3.3	Beispiel Arztregister	103
	3.4	Beispiel Friseursalonverwaltung	108
	3.5	Beispiel Seminarorganisation	112
<b>LE 6</b>	<b>4</b>	<b>Checklisten zur Erstellung eines OOA-Modells</b>	<b>119</b>
	4.1	Analyseprozeß	120
	4.2	Checkliste Geschäftsprozeß	127
	4.3	Checkliste Paket	134
<b>LE 7</b>		<b>Statisches Modell</b>	<b>141</b>
	4.4	Checkliste Klasse	142
	4.5	Checkliste Assoziation	147
	4.6	Checkliste Attribut	157
	4.7	Checkliste Vererbung	162

## Inhalt

<b>LE 8</b>	<b>Dynamisches Modell</b>	169
4.8	Checkliste Szenario	170
4.9	Checkliste Zustandsautomat	177
4.10	Checkliste Operation	183
4.11	Formale Inspektion	185
<b>5</b>	<b>Gestaltung von Benutzungsoberflächen</b>	193
<b>LE 9</b>	<b>Teil 1</b>	193
5.1	Einführung in die Software-Ergonomie	194
5.2	Dialoggestaltung	195
5.3	Fenster	199
5.4	Menüs	202
<b>LE 10</b>	<b>Teil 2</b>	209
5.5	Vom Klassendiagramm zur Dialogstruktur	210
5.6	Interaktionselemente	215
5.7	Gestaltung von Fenstern	221
<b>6</b>	<b>Konzepte und Notation des objektorientierten Entwurfs</b>	227
<b>LE 11</b>	<b>Teil 1</b>	227
6.1	Objekt/Klasse	228
6.2	Attribut	235
6.3	Operation	238
6.4	Assoziation	244
<b>LE 12</b>	<b>Teil 2</b>	255
6.5	Polymorphismus	256
6.6	Vererbung	261
6.7	Paket	267
6.8	Szenario	269
6.9	Zustandsautomat	274
<b>LE 13</b>	<b>7 Entwurfsmuster</b>	281
7.1	Entwurfsmuster, <i>Frameworks</i> , Klassenbibliotheken	282
7.2	Fabrikmethode-Muster	286
7.3	<i>Singleton</i> -Muster	287
7.4	Kompositum-Muster	289
7.5	Proxy-Muster	291
7.6	Fassaden-Muster	293
7.7	Beobachter-Muster	295
7.8	Schablonenmethode-Muster	297

	<b>8</b>	<b>Datenbanken</b> 303
<b>LE 14</b>		<b>Relationale Datenbanken und objekt-relationale Abbildung</b> 303
	8.1	Was ist ein Datenbanksystem? 304
	8.2	Relationale Datenbanksysteme 306
	8.3	Abbildung des objektorientierten Modells auf Tabellen 314
<b>LE 15</b>		<b>Objektorientierte Datenbanken</b> 325
	8.4	Objektorientierte Datenbanksysteme 326
	8.5	ODL 332
	8.6	OQL 335
	8.7	Anbindung an C++ 339
	8.8	Objekt-relationale Datenbanksysteme 345
<b>LE 16</b>	<b>9</b>	<b>Verteilte objektorientierte Anwendungen</b> 351
	9.1	Kommunikation von verteilten Objekten 352
	9.2	ORB-Architektur 353
	9.3	OMA 356
	9.4	IDL 357
	9.5	Entwicklung eines verteilten Systems 361
	<b>10</b>	<b>Erstellen eines Entwurfsmodells mittels Drei-Schichten-Architektur</b> 369
<b>LE 17</b>	<b>Teil 1</b>	369
	10.1	Architekturentwurf 370
	10.2	Entwurf der Fachkonzeptschicht 377
	10.3	Entwurf der GUI-Schicht und Anbindung an die Fachkonzept-Klassen 382
<b>LE 18</b>	<b>Teil 2</b>	395
	10.4	Entwurf der Datenhaltung mit einem objektorientierten Datenbanksystem 396
	10.5	Entwurf der Datenhaltung mit flachen Dateien 401
	10.6	Entwurf der Datenhaltung mit einem relationalen Datenbanksystem 405
<b>Exkurs 1</b>	<b>Erstellen des Prototyps der Benutzungsoberfläche mit dem Ressourcen-Editor (Microsoft Visual Studio C++)</b>	417
	1	Interaktionselemente des Ressourcen-Editors 418
	2	Prototyp mit Dialogfenstern 420
	2.1	Arbeitsbereich anlegen 421
	2.2	Erstellen der Menüs 421
	2.3	Realisieren des Erfassungsfensters 422
	2.4	Realisieren des Listenfensters 424
	2.5	Verbinden der Menüs und der Fenster 425

## **Inhalt**

- 2.6 Programmieren der Schaltflächen in den Fenstern 426
- 2.7 Realisieren der *one*-Richtung einer Assoziation 427
- 2.8 Realisieren der *many*-Richtung einer Assoziation 429
- 2.9 Programmieren der Schaltflächen für die Assoziationen 430
- 2.10 Abbildung des vollständigen OOA-Modells auf die Benutzungsoberfläche 431
- 3 Prototyp mit MDI-Unterfenstern 434
- 3.1 Arbeitsbereich anlegen 434
- 3.2 Erstellen der Menüs 434
- 3.3 Realisieren des Erfassungsfensters 435
- 3.4 Realisieren des Listenfensters 438
- 3.5 Verbinden der Menüs und der Fenster 439
- 3.6 Programmieren der Schaltflächen in den Fenstern 440
- 3.7 Abbildung weiterer Klassen auf die Benutzungsoberfläche 441
- 3.8 Realisieren der Assoziation 442

## **Exkurs 2 Realisierung der Datenhaltung mit dem objektorientierten Datenbanksystem Poet für C++ 445**

- 1 Verwalten von Projekten 446
- 2 Erstellen einer einfachen Klasse 448
- 2.1 Schemadeklaration für eine Klasse 448
- 2.2 Speichern eines Objekts 449
- 2.3 Zugriff auf die Klassenextension 450
- 2.4 Selektion von Objekten 451
- 2.5 Öffnen und Schließen der Datenbank 451
- 3 Realisieren der Vererbung 452
- 4 Realisieren von Assoziationen 454
- 4.1 Realisierung mit *pointer*-Referenzen 454
- 4.2 Realisierung mit *ondemand*-Referenzen 457
- 4.3 Realisierung von *many*-Assoziationen 459

## **Anhang 1 Durchgängiges Fallbeispiel 463**

- 1 Pflichtenheft 464
- 2 OOA-Modell 468
- 3 Prototyp der Benutzungsoberfläche 471
- 4 Datenhaltung mittels einer relationalen Datenbank 475
- 5 Datenhaltung mittels einer objektorientierten Datenbank 477
- 6 OOD-Modell und Implementierung 481

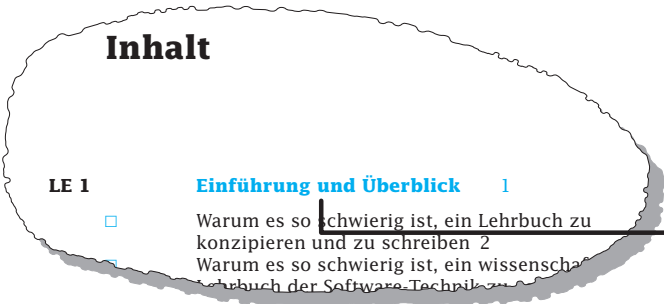
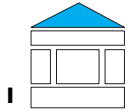
**Anhang 2    Lösungen** 483

**Anhang 3    Gesamtglossar** 533

**Referenzierte und ergänzende Literatur** 555

**Index** 567

# Navigation



Das Inhaltsverzeichnis führt direkt zu einem Kapitel bzw. Unterkapitel einer LE.

**Einführung und Überblick**

**LE 1**

- Aufzügen können, wenn sich Software von anderen technischen Profilen unterscheidet.
- Die Veränderungen, denen Software in den letzten zehn Jahren unterworfen war, beschreiben können.
- Darstellen können, daß Änderungen während der Entwicklung und hoher Portabilitätsanforderungen die Software-Erstellung zusätzlich erschweren.
- Die Ökonomie Software-Technik mit ihren Begriffen beschreiben können.
- Die Terminologie um die definierten Begriffe System und Software herum kennen und anwenden können.

■ Warum es so schwierig ist, ein Lehrbuch zu konzipieren und zu schreiben? 2

■ Warum es so schwierig ist, ein wissenschaftliches Lehrbuch der Software-Technik zu schreiben? 15

■ Was ist Software? 21

■ Warum ist Software so schwer zu entwickeln? 25

■ Warum ist marktreife Software so schwer zu entwickeln? 34

■ Was ist Software-Technik? 35

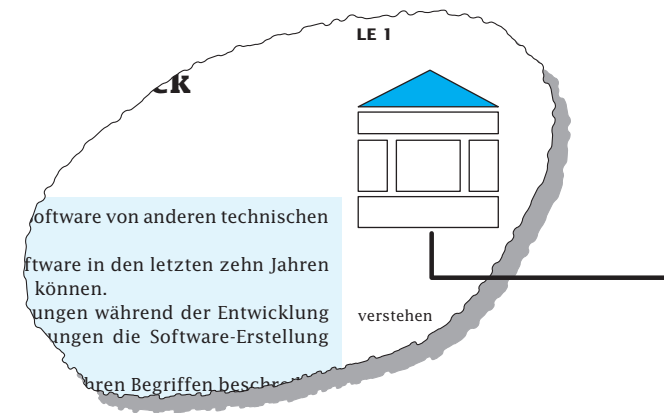
■ Wie ist dieses Buch gegliedert und aufgebaut? 41

■ Wie können Sie dieses Buch lesen? 42

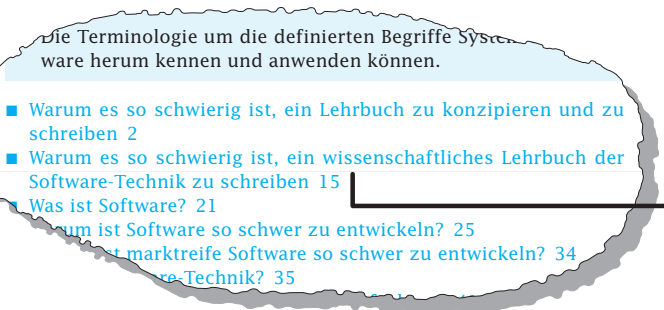
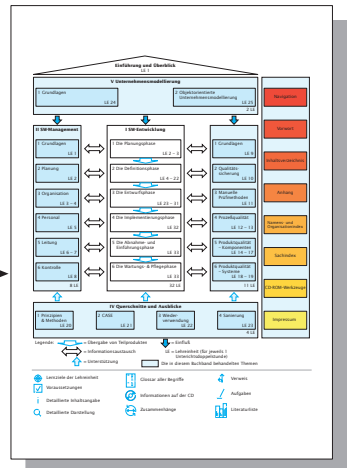
■ Was noch zu sagen bleibt? 45

Der obige Leserkann zunächst die folgenden zwei Abschnitte überspringen und mit dem Abschnitt "Was ist Software?" beginnen.

Für einige Leser



Dieses Piktogramm zu Beginn jeder LE führt zurück zum Haus der Software-Technik des Inhaltsverzeichnisses.



**Warum es so schwierig ist, ein SWT-Lehrbuch zu schreiben** LE 1

Abb. 20: SWT-Lehrbuch zum multidisziplinären CT

**Warum es so schwierig ist, ein wissenschaftliches Lehrbuch der Software-Technik zu schreiben**

Erschienen es schon schwierig genug, ein wissenschaftliches Lehrbuch zu schreiben, so kommen noch einige weitere Schwierigkeiten aus dem Fachgebiet der Software-Technik hinzu.

Die erste Frage, die sich stellt, betrifft die **Beziehung des Fachgebiets**, über das man ein Lehrbuch schreiben will.

Für das vorzunehmende Fachgebiet gab es drei Möglichkeiten:

- Software-Engineering
- Software-Technologie
- Software-Technik

Der Begriff **Software-Engineering** ist der international gebräuchlichste, englische Begriff. Ziel war es jedoch, ein Lehrbuch für deutsch-

Die Gliederung zu Beginn jeder LE führt zum entsprechenden Abschnitt innerhalb der LE.

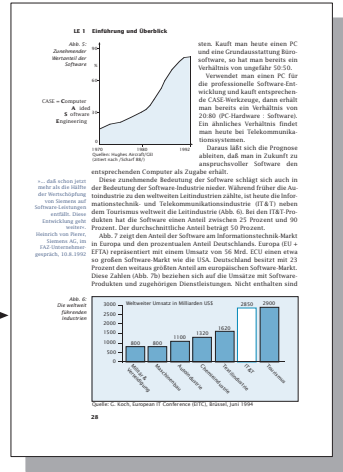
# Navigation



zunehmendem Maße entwickelt sich Software zu einem wichtigen Wirtschaftsgut und spielt eine entscheidende Rolle in der Gesellschaft. Software ist Bestandteil der meisten hochwertigen technischen Produkte und Dienstleistungen geworden. In einigen Bereichen, wie Banken und Versicherungen, werden nahezu alle Dienstleistungen durch Software-Einsatz realisiert« /BMFT 94, S. 3/.

Der relative Wertanteil der Software an den Gesamtkosten eines Computersystems bzw. eines Anwendungssystems ist in den letzten Jahren ständig gestiegen (Abb. 5).

Die Softwarekosten 90 Prozent d



Abbildungsverweise, die auf eine Abbildung auf einer anderen Seite zeigen, führen zu dieser Abbildung.

Für das vorgesehene Fachgebiet gab es drei Möglichkeiten:

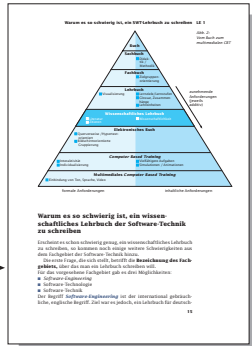
- Software-Engineering
- Software-Technologie
- Software-Technik

Der Begriff **Software-Engineering** ist der international gebräuchliche, englische Begriff. Ziel war es jedoch, ein Lehrbuch für deutsch-



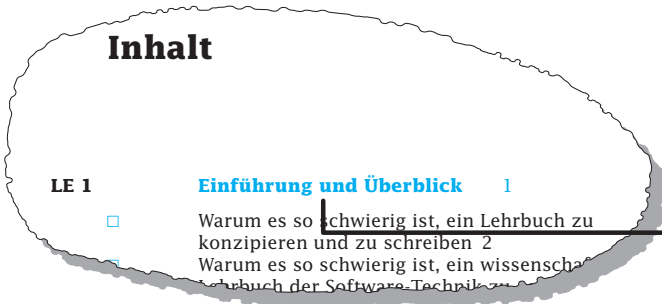
### Sachindex

Einfluss des JANUS-Systems 654	stepwise refinement 937, 958
beim Einsatz eines Maskengenerators 652	Steuerbarkeit 521, 539
beim Einsatz eines UIMS/UI-Builders 652	Steuersystem 656 f, 664 f
modulare 848	stored procedures 691
strukturierte 810	Strefstest 963
Software-Engineering 15, 35, 46 f	structure 185
Software-Entwicklung 39, 46 f, 53, 154	structure charts 824
komponentenbasierte 870,	structured analysis 99, 118, 398, 415
	structured programming 238
	Struktogramm 98, 217, 238
	Struktogramm 238



Im Sachindex führt der Begriff – falls vorhanden – zur Erklärung im Glossar, die Seitenangabe zur entsprechenden Seite im Buch.

# Navigation



Das Inhaltsverzeichnis führt direkt zu einem Kapitel bzw. Unterkapitel einer LE.

**Einführung und Überblick**

**LE 1**

- Aufzählen können, worin sich Software von anderen technischen Profiklassen unterscheidet.
- Die Veränderungen, denen Software in den letzten zehn Jahren unterworfen war, beschreiben können.
- Darstellen können, daß Änderungen während der Entwicklung und hoher Portabilitätsanforderungen die Software-Erstellung zusätzlich erschweren.
- Die Ökonomie Software-Technik mit ihren Begriffen beschreiben können.
- Die Terminologie um die definierten Begriffe System und Software herum kennen und anwenden können.

■ Warum es so schwierig ist, ein Lehrbuch zu konzipieren und zu schreiben? 2

■ Warum es so schwierig ist, ein wissenschaftliches Lehrbuch der Software-Technik zu schreiben? 15

■ Was ist Software? 21

■ Warum ist Software so schwer zu entwickeln? 25

■ Warum ist marktreife Software so schwer zu entwickeln? 34

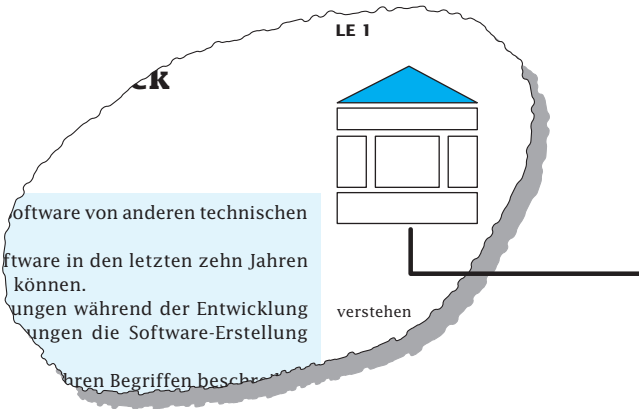
■ Was ist Software-Technik? 35

■ Wie ist dieses Buch gegliedert und aufgebaut? 41

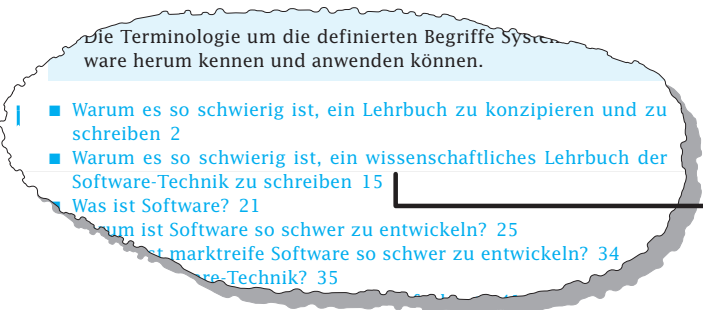
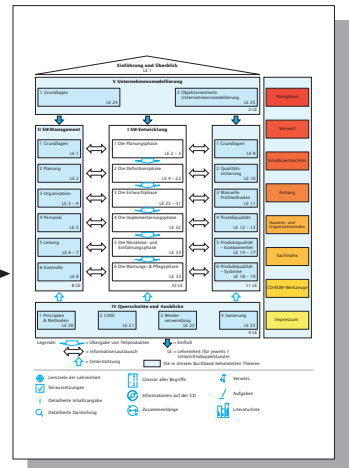
■ Wie können Sie dieses Buch lesen? 42

■ Was noch zu sagen bleibt 45

Der obige Leserkann zunächst die folgenden zwei Abschnitte überspringen und mit dem Abschnitt "Was ist Software" beginnen.



Dieses Piktogramm zu Beginn jeder LE führt zurück zum Haus der Software-Technik des Inhaltsverzeichnisses.



Die Gliederung zu Beginn jeder LE führt zum entsprechenden Abschnitt innerhalb der LE.

**Warum es so schwierig ist, ein SWT-Lehrbuch zu schreiben** LE 1

Abb. 20: SWT-Lehrbuch zum multimedialen CTT

**Warum es so schwierig ist, ein wissenschaftliches Lehrbuch der Software-Technik zu schreiben**

Erschienen es schon schwierig genug, ein wissenschaftliches Lehrbuch zu schreiben, so kommen noch einige weitere Schwierigkeiten aus dem Fachgebiet der Software-Technik hinzu.

Die erste Frage, die sich stellt, betrifft die **Beziehung des Fachgebiets**, über das man ein Lehrbuch schreiben will.

Für das vorzunehmende Fachgebiet gab es drei Möglichkeiten:

- Software-Engineering
- Software-Technologie
- Software-Technik

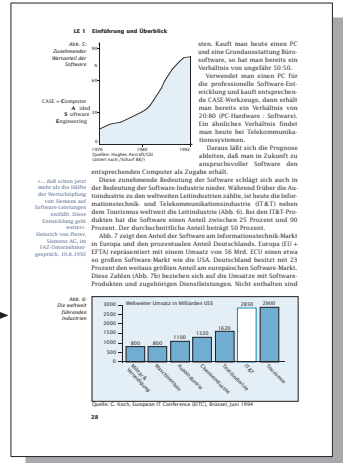
Der Begriff **Software-Engineering** ist der international gebräuchlichste, englische Begriff. Ziel war es jedoch, ein Lehrbuch für deutsch-



# Navigation



zunehmendem Maße entwickelt sich Software zu einem wichtigen Wirtschaftsgut und spielt eine entscheidende Rolle in der Gesellschaft. Software ist Bestandteil der meisten hochwertigen technischen Produkte und Dienstleistungen geworden. In einigen Bereichen, wie Banken und Versicherungen, werden nahezu alle Dienstleistungen durch Software-Einsatz realisiert« /BMFT 94, S. 3/. Der relative Wertanteil der Software an den Gesamtkosten eines Computersystems bzw. eines Anwendungssystems ist in den letzten Jahren ständig gestiegen (Abb. 5).



Abbildungsverweise, die auf eine Abbildung auf einer anderen Seite zeigen, führen zu dieser Abbildung.

Für das vorgesehene Fachgebiet gab es drei Möglichkeiten:

- Software-Engineering
- Software-Technologie
- Software-Technik

Der Begriff **Software-Engineering** ist der international gebräuchlichere, englische Begriff. Ziel war es jedoch, ein Lehrbuch für deutsch-

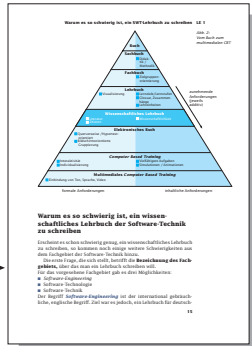
LE 1 Einführung und Überblick

Alexander Mitglied einer Institution oder Organisation, die der Entwicklung der Software-Industrie...  
 Philipps Graduiert, der den meisten...  
 15

Glossarbegriffe (fett und blau) führen zur Erklärung im Glossar.

**Sachindex**

Einfluss des JANUS-Systems	654	stepwise refinement	937, 958
Steuerbarkeit	521, 539	Steuerbarkeit	656 f, 664 f
beim Einsatz eines Maskengenerators	652	stored procedures	691
beim Einsatz eines UIMS/UI-Builders	652	Strefstest	963
modulare	848	structure	185
strukturierte	810	structure charts	824
Software-Engineering	15, 35, 46 f	structured analysis	99, 118, 398, 415
Software-Entwicklung	39, 46 f, 53, 154	structured programming	Struktogramm
komponentenbasierte	870,	Struktogramm	238
			238



LE 1 Einführung und Überblick

Alexander Mitglied einer Institution oder Organisation, die der Entwicklung der Software-Industrie...  
 Philipps Graduiert, der den meisten...  
 15

Im Sachindex führt der Begriff – falls vorhanden – zur Erklärung im Glossar, die Seitenangabe zur entsprechenden Seite im Buch.

# 1 Objektorientierte Softwareentwicklung



- Wissen, was unter objektorientierter Softwareentwicklung zu verstehen ist.
- Historische Entwicklung der Objektorientierung kennen.
- Die Begriffe objektorientierte Analyse (OOA) und objektorientierter Entwurf (OOD) kennen.
- Wissen, welche objektorientierten Konzepte es gibt.
- Erklären können, warum Objektorientierung sinnvoll ist.
- Erklären können, was eine Methode ist.
- Erklären können, wie sich die Phasen Analyse und Entwurf voneinander unterscheiden.
- Erklären können, welche Ergebnisse in der Analyse zu erstellen sind.
- Erklären können, welche Ergebnisse im Entwurf zu erstellen sind.

wissen

verstehen

- i 1.1 Einführung und Überblick 2
- 1.2 Objektorientierte Analyse 8
- 1.3 Objektorientierter Entwurf 11

## 1.1 Einführung und Überblick

Seit Beginn der 90er Jahre nehmen die Bedeutung und die Anzahl der objektorientierten Analyse- und Entwurfsmethoden ständig zu. Thema dieses Buches ist die objektorientierte Entwicklung in den Phasen Analyse und Entwurf. Bei einer **objektorientierten Softwareentwicklung** werden die Ergebnisse der Phasen Analyse, Entwurf und Implementierung objektorientiert erstellt. Für letztere werden objektorientierte Programmiersprachen verwendet. Zusätzlich kann die Benutzungsoberfläche der Software objektorientiert gestaltet sein, und es können objektorientierte Datenbanken verwendet werden. Auch die Verteilung auf einem Netz kann objektorientiert erfolgen. Es entsteht eine Softwareentwicklung »aus einem Guß«.

objektorientiert  
vs. strukturiert

Bei der strukturierten bzw. klassischen Entwicklung werden in der Systemanalyse Datenfluß- und *Entity-Relationship*-Diagramme eingesetzt. Die Datenflußdiagramme der strukturierten Analyse /DeMarco 79/ dienen zur Modellierung der Prozesse. Mittels der *Entity-Relationship*-Diagramme bzw. der semantischen Datenmodelle werden Informationsstrukturen dargestellt. Beim klassischen Entwurf werden die Ergebnisse der Analyse entweder in Funktionsbäume des strukturierten Entwurfs /Page-Jones 88/ oder in Module im Sinne von Datenabstraktionen umgesetzt. Gegenüber der objektorientierten Entwicklung besitzt die klassische (strukturierte) Entwicklung folgende Nachteile:

- Die damit erstellten Produkte (Dokumentation, Programme) sind weniger flexibel bezüglich zukünftiger Änderungen und Erweiterungen.
- Umfangreiche Modelle der strukturierten Analyse sind schwierig zu lesen und zu ändern.
- Insbesondere bei großen Projekten ist es ausgesprochen schwierig, die Konsistenz zwischen Datenfluß- und *Entity-Relationship*-Diagrammen sicherzustellen.
- Wird der Entwurf nur mittels Funktionen realisiert, so entstehen Programme, die aufwendig in der Wartung sind.
- Beim Übergang von der strukturierten Analyse zur Datenabstraktion entsteht ein eklatanter Strukturbruch. Die Durchgängigkeit zwischen den Phasen wird dadurch erheblich erschwert.

Die bessere Durchgängigkeit wird bei den objektorientierten Techniken dadurch erreicht, daß in allen Phasen – Analyse, Entwurf und Implementierung – dieselben Konzepte verwendet werden. In Analyse und Entwurf wird sogar dieselbe Notation eingesetzt. Beim Übergang von der objektorientierten Analyse zum objektorientierten Entwurf tritt somit kein Strukturbruch auf. Damit wird eine wichtige Anforderung an die Softwareentwicklung besser erfüllt als bei den strukturierten Techniken: die Nachvollziehbarkeit.

Eine Änderung im Entwurf kann leicht in der Analyse nachgetragen werden und umgekehrt.

Allein das Klassenkonzept der Objektorientierung bringt eine ganze Reihe von Vorteilen mit sich. Durch die Bildung von Kapseln kann eine Klasse leichter verstanden und geändert bzw. erweitert werden, ohne daß die anderen Klassen stark davon betroffen sind. Damit unterstützen Verkapselung und Geheimnisprinzip die einfache Wartbarkeit der entstandenen Systeme. Eine sorgfältig aufgebaute Vererbungshierarchie garantiert einerseits eine leichte Erweiterbarkeit und darüber hinaus ein hohes Maß an Wiederverwendbarkeit im gleichen Projekt und ggf. auch über mehrere Projekte hinweg.

### Historische Entwicklung

Die objektorientierte Softwareentwicklung fand ihren Anfang in der ersten objektorientierten Programmiersprache Smalltalk-80. Sie wurde in den Jahren 1970 bis 1980 am Palo Alto Research Center (PARC) der Firma Xerox entwickelt. Das Klassenkonzept wurde von der Programmiersprache Simula-67 übernommen und weiterentwickelt. Mit Beginn der 90er Jahre hat sich C++ als dominierende Sprache der objektorientierten Programmierung (OOP) durchgesetzt. Seit 1996 nimmt Java eine signifikante Stellung neben C++ ein, während Smalltalk im gleichen Maß zurückgedrängt wurde.

Ende der 80er und zu Beginn der 90er Jahre wurden die ersten Bücher über Methoden der objektorientierten Analyse (OOA, *Object Oriented Analysis*) und des objektorientierten Entwurfs (OOD, *Object Oriented Design*) publiziert, denen inzwischen viele folgten. Im Gegensatz zu den textuellen Notationen der Programmiersprachen werden hier grafische Notationen verwendet. Von den zahlreichen veröffentlichten Methoden werden jedoch nur wenige von einer größeren Anzahl von Softwareentwicklern weltweit angewendet. Die Bücher von /Booch 91, 94/, /Coad,Yourdon 91, 91a/, /Jacobson 92/, /Rumbaugh et al. 91/ und /Shlaer, Mellor 88, 92/ gelten heute als Standardwerke. Für diese Methoden stehen den Entwicklern eine zunehmende Anzahl von Werkzeugen zur Verfügung.

Im Oktober 1994 haben sich Grady Booch und Jim Rumbaugh bei der Rational Software Corporation zusammengeschlossen, um ihre erfolgreichen Methoden zu einem einheitlichen Industriestandard weiterzuentwickeln. Es entstand zunächst der Vorgänger der *Unified Modeling Language* (UML), der unter dem Namen *Unified Method* 0.8 /Booch, Rumbaugh 95/ publiziert wurde. Seit Herbst 1995 wirkt auch Ivar Jacobson an der Entwicklung der UML mit. Im Oktober 1996 wurde die Version 0.91 der UML veröffentlicht /UML 96/. Seit September 1997 gilt die Version 1.1 der UML /UML 97/, in die zusätzlich die Ideen verschiedener UML-Partner eingeflossen sind. UML 1.1 wurde von der *Object Management Group* (OMG) am

## LE 1 1 Objektorientierte Softwareentwicklung

17. November 1997 als Standard verabschiedet. Zur Zeit sprechen alle Anzeichen dafür, daß die **UML** die Notation der Zukunft für die Objektorientierung wird.

OMG, OMA, CORBA

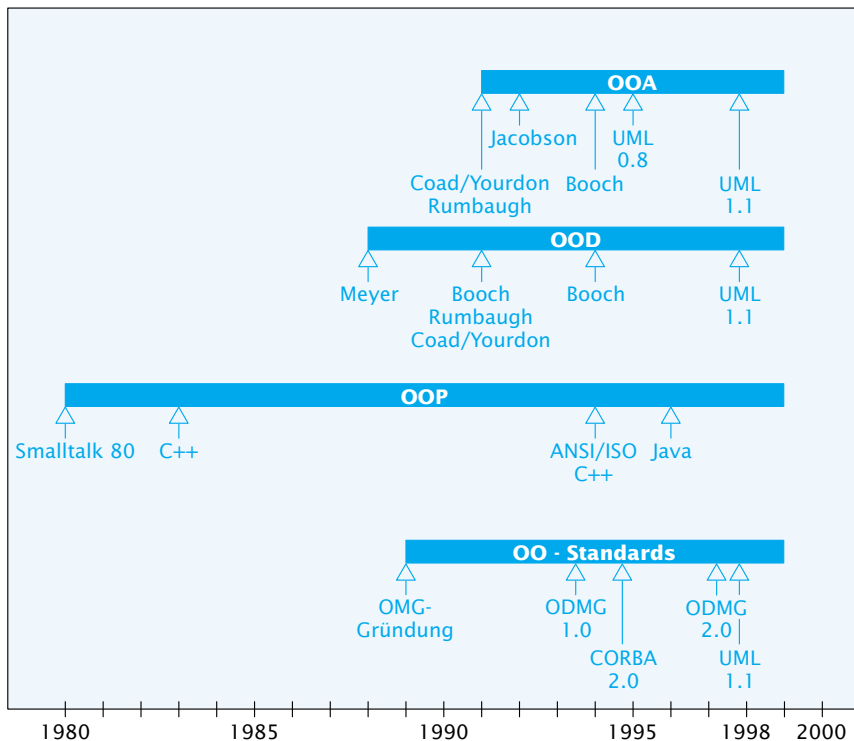
1989 wurde die **OMG** (*Object Management Group*) von acht Firmen gegründet. Im November 1997 hatte sie bereits über 750 Mitglieder. Das Ziel der OMG ist ein allgemeiner Architektur-Rahmen für objektorientierte Anwendungen, der auf einer weltweit verbreiteten Schnittstellenspezifikation basiert («*The Architecture for a Connected World*»). Das Ergebnis ist die *Object Management Architecture* (OMA). Die zentrale Komponente dieses Standards bildet der *Object Request Broker* (ORB), der unter dem Namen CORBA bekannt ist.

ODMG

1991 gründeten Hersteller und Anwender von objektorientierten Datenbanken die *Object Database Management Group* (ODMG). 1993 wurde von dieser Gruppe ein Standard für objektorientierte Datenbanken vorgeschlagen: der ODMG-93-Standard. Im Juli 1997 wurde ODMG 2.0 freigegeben, der außer den Schnittstellen für C++ und Smalltalk auch eine Java-Schnittstelle definiert.

Abb. 1.1-1 zeigt die historische Entwicklung der Objektorientierung.

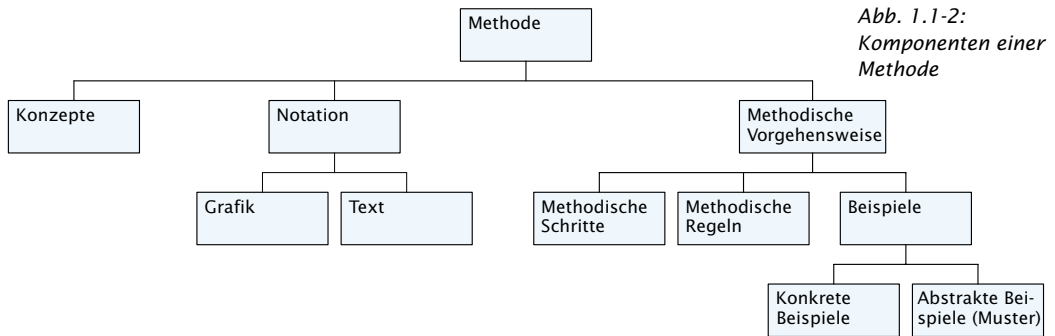
Abb. 1.1-1:  
Historische  
Entwicklung der  
Objektorientierung



## Was ist eine objektorientierte Methode?

Ich möchte zunächst den Methodenbegriff genauer erläutern und dann den Umfang der hier vorgestellten Methode beschreiben.

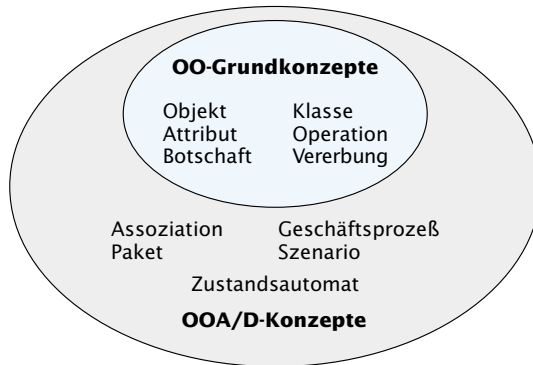
Der Begriff »Methode« beschreibt die systematische Vorgehensweise zur Erreichung eines bestimmten Ziels (*griech. *methodos**). In der Softwaretechnik wird der Begriff **Methode** jedoch auch als Oberbegriff von Konzepten, Notation und methodischer Vorgehensweise verstanden (Abb. 1.1-2). Beispielsweise spricht man von der OMT-Methode oder der Booch-Methode. Bei der UML handelt es sich dagegen um eine reine Notation, welche die objektorientierten Konzepte unterstützt.



Ein großer Vorteil der objektorientierten Softwareentwicklung ist, daß die meisten **Konzepte** durchgängig über die Phasen Definition, Entwurf und Implementierung verwendet werden können. Diejenigen Konzepte, die in allen Phasen vorhanden sind, werden als objektorientierte Grundkonzepte bezeichnet. Sie besitzen ihren Ursprung in den objektorientierten Programmiersprachen. Daher sind nur solche Programmiersprachen objektorientiert, die alle diese Grundkonzepte unterstützen. Sprachen, die nur einen Teil unterstützen, werden als objektbasiert bezeichnet. Da die objektorientierten Grundkonzepte nicht ausreichen, um ein Fachkonzept zu modellieren, wurden sie um Konzepte aus der semantischen Datenmodellierung erweitert. Erst die Synthese zwischen der Welt der objektorientierten Programmiersprachen und der semantischen Datenmodellierung machte die objektorientierte Systemanalyse praktikabel. Um das dynamische Verhalten geeignet modellieren zu können, haben viele Autoren die objektorientierte Analyse um Geschäftsprozesse, Szenarios und Zustandsautomaten erweitert. Man braucht also verschiedene Sichten auf das System, um alle notwendigen Informationen übersichtlich modellieren zu können. Die in Abb. 1.1-3 dargestellten Konzepte werden von den meisten objektorientierten Notationen und natürlich auch von der UML unterstützt.

## LE 1 1 Objektorientierte Softwareentwicklung

Abb. 1.1-3:  
Objektorientierte  
Konzepte



Die **Notation** einer Methode besteht aus Grafiken (z.B. Klassendiagramm) und Texten (z.B. Spezifikation), die durch entsprechende Dokumentationsstandards ergänzt werden. Bei den Notationen herrschte bisher große Divergenz /Stein 94/. Zur Zeit spricht jedoch alles dafür, daß sich die UML als Standard-Notation durchsetzt und von allen Werkzeugen unterstützt wird.

Die **methodische Vorgehensweise** zur Erstellung von objektorientierten Modellen besteht aus einer Anzahl von methodischen Schritten. Die meisten objektorientierten Methoden beinhalten zwischen 4 und 13 Schritten. Beim praktischen Einsatz zeigt sich, daß ein solch grobes Methodenraster nicht ausreicht. Andererseits ist ein sehr detailliertes Vorgehensmodell oft nur für bestimmte Anwendungen geeignet und kann nicht problemlos auf andere Bereiche übertragen werden. Der erfahrene Entwickler wendet – meist mehr oder weniger intuitiv – Hunderte von Regeln an, die er situationsspezifisch einsetzt. Im Gegensatz zu den methodischen Schritten gibt es für die Anwendung dieser Regeln keine festgelegte Reihenfolge. Außerdem greift der erfahrene Entwickler in vielen Fällen auf bereits gelöste ähnliche Problemstellungen zurück. Ist dieser Fundus nicht vorhanden, dann hat es sich bewährt, eine Sammlung von spezifischen Beispielen oder Mustern (*patterns*) zu verwenden.

Die hier beschriebene Methode legt großen Wert auf eine präzise Trennung von Analyse und Entwurf, bewahrt jedoch andererseits die leichte Durchgängigkeit zwischen beiden Phasen. Die Analyse erfordert andere Fähigkeiten und Kenntnisse als der Entwurf. Für die Erstellung des OOA-Modells ist fachliches Expertenwissen notwendig, denn nur der Fachexperte weiß, *was* das System aus Benutzersicht leisten soll. Um dieses fachliche Wissen zu erlangen, kann durchaus eine Einarbeitungszeit von einem Jahr oder länger notwendig sein. Es wäre also nicht profitabel, wenn sich OOA-Experten dieses Fachwissen aneignen würden. Stattdessen ist es sinnvoller, die Fachexperten in objektorientierter Analyse zu schulen und ihnen zusätzlich einen OOA-Experten für die methodische Projekt-

betreuung zur Seite zu stellen. Die Erstellung eines OOA-Modells ist immer eine neue kreative Leistung. Vorhandene Modelle oder Muster leisten nur eine geringe Hilfestellung. Um dem Fachexperten die Einarbeitung in die objektorientierte Analyse so einfach wie möglich zu machen, werden im Kapitel 2 nur die absolut notwendigen Konzepte und Notationselemente für die Analyse aufgeführt, während alle komplexeren Notationselemente und Konzepte erst im entsprechenden Entwurfskapitel (Kapitel 6) eingeführt werden.

Entwurf und Implementierung sind bei der objektorientierten Entwicklung so stark verzahnt, daß eine personelle Trennung hier nicht sinnvoll ist. Der Entwurf wird sinnvollerweise von einem Softwarekonstrukteur oder Programmierer erstellt. Im Gegensatz zur Analyse nimmt die Standardisierung hier durch ein großes Angebot von Mustern, *Frameworks* und Klassenbibliotheken ständig zu. Viele Probleme müssen und sollen nicht mehr individuell gelöst werden, sondern es geht darum, vorhandene Problemlösungen den Softwarekonstruktoren zugänglich zu machen. Im Buch werden diese verschiedenen Tätigkeitsfelder durch sehr unterschiedliche Methoden in Analyse und Entwurf sichtbar. Außer dieser relativ groben Trennung in Fachexperten und Softwarekonstruktoren ist heute aufgrund der hohen Komplexität der Softwareentwicklung eine weitere Spezialisierung notwendig. Beispielsweise gibt es Experten für die Gestaltung von Benutzungsoberflächen und Spezialisten für die Anbindung des Fachkonzepts an eine bestimmte Datenbanktechnik. Insgesamt unterstützt die in diesem Buch beschriebene Methode

- alle notwendigen objektorientierten Konzepte für Analyse und Entwurf,
- die Notation der UML,
- eine präzise Trennung der Modellelemente der UML in Analyse und Entwurf,
- die Verwendung von Analysemustern,
- eine methodische Vorgehensweise für die objektorientierte Analyse,
- die Abbildung des Analysemodells auf eine objektorientierte Benutzungsoberfläche,
- die Verwendung von Entwurfsmustern,
- die Realisierung der Drei-Schichten-Architektur im Entwurf,
- eine standardisierte Anbindung der Benutzungsoberfläche an Fachkonzept und Datenhaltung,
- die objekt-relationale Abbildung zur Anbindung an relationale Datenbanken,
- die Anbindung an objektorientierte Datenbanken,
- die standardisierte Verteilung objektorientierter Systeme im Netz,
- die Transformation des Entwurfs in C++ und in Java und
- die analytische Qualitätssicherung in Analyse und Entwurf.



## 1.2 Objektorientierte Analyse

Ziel der Analyse Das Ziel der **Analyse** ist es, die Wünsche und Anforderungen eines Auftraggebers an ein neues Softwaresystem zu ermitteln und zu beschreiben. Es muß ein Modell des Fachkonzepts erstellt werden, das konsistent, vollständig, eindeutig und realisierbar ist. Es ist wichtig, daß bei der Modellbildung in der (System-)Analyse alle Aspekte der Implementierung bewußt ausgeklammert werden. Es wird in der Analyse von einer perfekten Technik ausgegangen. Perfekte Technik /McMenamin, Palmer 88/ bedeutet, daß der Prozessor jede Funktion ohne Verzögerung ausführen kann, keine Fehler macht oder gar ausfällt. Der Speicher kann unendlich viele Informationen aufnehmen und der Prozessor ohne Zeitverzögerung darauf zugreifen. Wir abstrahieren also von allen technischen Randbedingungen, wie z.B. Zugriffszeiten und Speichergröße. Auch die Verteilungen der Software auf mehrere Computersysteme betrachten wir vorerst nicht. Es ist auch nicht von Bedeutung, in welcher Form die Daten gespeichert werden. Zusammenfassend können wir sagen: Es ist die Aufgabe des Systemanalytikers, die »wahren« Anforderungen seines Auftraggebers zu modellieren und dies in einer Weise, die durch keine Implementierungstechnik eingeschränkt ist.

Die Systemanalyse gehört zu den anspruchsvollsten Tätigkeiten der Softwareentwicklung, da die Anforderungen des Auftraggebers in der Regel unklar, widersprüchlich sowie fallorientiert sind und sich auf unterschiedlichen Abstraktionsebenen befinden. Das liegt daran, daß der Auftraggeber kein vollständiges Modell des zukünftigen Systems »im Kopf« hat. Es ist die schwierige Aufgabe des Systemanalytikers, daraus ein konsistentes, vollständiges und eindeutiges Modell zu erstellen, das anschließend realisiert werden kann. Die »wahren« Anforderungen sind nicht plötzlich da, sondern die Systemanalyse bildet einen kontinuierlichen Prozeß, um Informationen zu sammeln, zu filtern und zu dokumentieren.

Es ist nicht die Aufgabe des Auftraggebers, den Systemanalytiker zu verstehen, sondern der Systemanalytiker wird dafür bezahlt, sich dem Auftraggeber verständlich zu machen!

Bei der **objektorientierten Analyse (OOA)** gehen wir von Objekten aus, die sich in der realen Welt befinden. Das sind nicht nur »anfaßbare« Objekte oder Personen, sondern – häufig – Begriffe oder Ereignisse aus dem jeweiligen Anwendungsbereich. Aus einem realen Objekt wird durch Modellbildung und geeignete Abstraktion ein Objekt unseres objektorientierten Modells. Objekte, die sich durch die gleichen Eigenschaften beschreiben lassen, gehören der gleichen Klasse an. Bei der objektorientierten Analyse wird *nicht* beschrieben, *wie* Objekte auf der Benutzungsoberfläche dargestellt werden oder *wie* sie gespeichert und selektiert werden. Ziel der objektorientierten Analyse ist es, das zu realisierende Problem zu

verstehen und in einem OOA-Modell zu beschreiben. Dieses Modell soll die essentielle Struktur und Semantik des Problems, aber noch keine technische Lösung beschreiben. Es darf keinerlei Optimierungen für das verwendete Computersystem oder die benutzte Basissoftware enthalten.

Es wird häufig die Frage gestellt, ob ein »normaler« Auftraggeber ein OOA-Modell wirklich verstehen kann. Dazu wird sicher nicht jeder Auftraggeber in der Lage sein. Andererseits ist es fraglich, ob derselbe Auftraggeber mit einer Hierarchie von Datenflußdiagrammen, *Entity-Relationship*-Diagrammen oder Zustandsautomaten keine Schwierigkeiten hätte. Um in der Systemanalyse dasjenige System zu modellieren, das der Auftraggeber *will*, sollten Sie einen Prototyp der Benutzungsoberfläche erstellen. Dieser Prototyp kann nach festen Transformationsregeln aus dem OOA-Modell abgeleitet werden. Er ist sozusagen das »OOA-Modell«, das dem Auftraggeber präsentiert wird. Anhand dieses Prototyps wird er seine Änderungswünsche artikulieren, die dann vom Systemanalytiker in das OOA-Modell übertragen werden. Da beide Modelle die gleiche objektorientierte Struktur realisieren, ist eine Konsistenzprüfung sehr einfach durchzuführen.

Prototyp der Benutzungsoberfläche

### Welche Produkte sind in der Analysephase zu erstellen?

Der erste Schritt der Systemanalyse sollte darin bestehen, zunächst ein Pflichtenheft zu erstellen, das dann als Ausgangsbasis für eine systematische Modellbildung dient. Das **Pflichtenheft** ist eine textuelle Beschreibung dessen, *was* das zu realisierende System leisten soll. Es soll bei diesem Vorgehen zwei Zielsetzungen erfüllen. Zum einen ist es das »Einstiegsdokument« in das Projekt für alle, die das System später pflegen und warten sollen. Zweitens soll es den Systemanalytiker dazu in die Lage versetzen, das OOA-Modell zu erstellen. Das Pflichtenheft besitzt also ein niedrigeres Detaillierungsniveau als das OOA-Modell. Es ist *nicht* das Ziel, anhand des Pflichtenheftes das System zu implementieren. Weitere Informationen zum Pflichtenheft finden Sie in Anhang 1.

Pflichtenheft



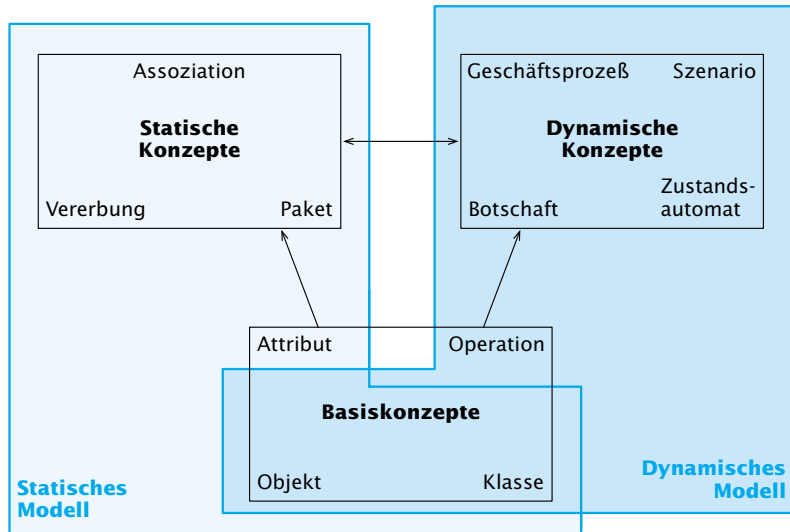
Das **OOA-Modell** (Analysemodell) bildet die fachliche Lösung des zu realisierenden Systems. Wir sprechen daher vom Fachkonzept. Es besteht aus einem statischen und einem dynamischen Modell. Welches dieser beiden Modelle in der Systemanalyse das größere Gewicht besitzt, hängt wesentlich von der jeweiligen Anwendung ab. Das statische Modell ist bei typischen Datenbank-Anwendungen besonders wichtig. Das dynamische Modell ist insbesondere bei stark interaktiven Systemen von Bedeutung. Abb.1.2-1 zeigt, wie die objektorientierten Konzepte auf diese Modelle abgebildet werden.

Anhang 1  
OOA-Modell

Das **statische Modell** beschreibt insbesondere die Klassen des Systems, die Assoziationen zwischen den Klassen und die Verer-

## LE 1 1 Objektorientierte Softwareentwicklung

Abb.1.2-1:  
Statisches und  
dynamisches  
Modell



ungsstrukturen. Außerdem enthält es die Daten des Systems (Attribute). Die Pakete dienen dazu, Teilsysteme zu bilden, um bei großen Systemen einen besseren Überblick zu ermöglichen.

Das **dynamische Modell** zeigt Funktionsabläufe. Geschäftsprozesse beschreiben die durchzuführenden Aufgaben auf einem sehr hohen Abstraktionsniveau. Szenarios zeigen, wie Objekte miteinander kommunizieren, um eine bestimmte Aufgabe zu erledigen. Zustandsautomaten beschreiben in der Analyse die Lebenszyklen von Objekten, d.h. die Reaktionen eines Objekts auf verschiedene Ereignisse (Botschaften).

Das Modell des Fachkonzepts muß alle Informationen enthalten, um daraus einen Prototyp der Benutzungsoberfläche abzuleiten. Oft ermöglicht erst das Vorhandensein dieses Prototyps, mit dem zukünftigen Benutzer bzw. mit dem Auftraggeber abzuklären, ob das System wirklich wie gewünscht spezifiziert ist. In diesen Prototypen gehen natürlich außer der Information aus dem Fachkonzept auch Informationen über die optimale ergonomische Gestaltung ein.

Benutzungs-  
oberfläche

Der **Prototyp** der Benutzungsoberfläche ist ein ablauffähiges Programm, das alle Attribute des OOA-Modells auf die Oberfläche abbildet. Es realisiert weder Anwendungsfunktionen, noch besitzt es die Fähigkeit, Daten zu speichern. Der Prototyp besteht aus Fenstern, Dialogen, Menüs usw. Für die effektive Erstellung gibt es heute zahlreiche Werkzeuge. Der Zweck des Prototyps ist es, das erstellte OOA-Modell mit dem zukünftigen Benutzer oder einem Repräsentanten zu evaluieren. Das Ziel sollte es sein, möglichst die vollständige Benutzungsoberfläche durch diesen Prototyp auszudrücken. Wo das nicht möglich ist, muß eine ergänzende Dokumen-

tation erstellt werden. Dazu gehört beispielsweise das Konzept der Zugriffsrechte. Da sich Benutzungsoberflächen aufgrund des technischen Fortschritts schneller ändern als die Funktionalität des Fachkonzepts, ist die Trennung von Fachkonzept und Benutzungsoberfläche ein Grundprinzip der Entwicklung. Im Fachkonzept wird festgelegt, *welche Informationen* auf dem Bildschirm sichtbar sind. Bei der Benutzungsoberfläche wird festgelegt, in *welchem Format* sie dargestellt werden.

Das OOA-Modell wird im Team von Systemanalytikern, Fachexperten und zukünftigen Benutzern oder den Benutzerrepräsentanten erstellt. Bei der Entwicklung eines Buchhaltungssystems sind das beispielsweise außer dem Systemanalytiker der Buchhaltungsexperte, der alle einzuhaltenden Vorschriften kennt, und die Angestellte des Steuerberaters, die das System später benutzen soll. Die Größe dieser Gruppe sollte zwischen zwei und fünf Personen betragen. Dabei ist zu beachten, daß die Entwicklung natürlich nicht strikt in der Reihenfolge »OOA-Modell – Prototyp« erfolgt, sondern es finden ständig Iterationen folgender Art statt: Eine erste Version des OOA-Modells wird erstellt, daraus der Prototyp abgeleitet, mit dem Benutzer oder dessen Repräsentanten evaluiert, die daraus gewonnenen Erkenntnisse in das OOA-Modell umgesetzt, ein neuer Prototyp abgeleitet usw. Diese Iterationen finden solange statt, bis ein befriedigender Prototyp erstellt wurde, der mit dem OOA-Modell konsistent ist. Wichtig ist, daß der Prototyp *immer* aus dem OOA-Modell abgeleitet wird. Die umgekehrte Vorgehensweise führt bei nicht-trivialen Anwendungen zu schwer verständlichen Dialogabläufen. Zur Erstellung des Prototyps werden außer dem Systemanalytiker und der Fachabteilung (Benutzerrepräsentant) auch Experten auf dem Gebiet der Softwareergonomie benötigt.

Erstellung des  
OOA-Modells

## 1.3 Objektorientierter Entwurf

In der Analyse sind wir bei der Modellierung des Systems von einer idealen Umgebung ausgegangen. Aufgabe des **Entwurfs** ist es nun, die spezifizierte Anwendung auf einer Plattform unter den geforderten technischen Randbedingungen zu realisieren. Dabei befinden wir uns im Entwurf allerdings noch auf einem höheren Abstraktionsniveau als in der Implementierung. In der Entwurfsphase wird das OOD-Modell unter den Gesichtspunkten der Effizienz und Standardisierung konzipiert. Der **objektorientierte Entwurf (OOD)** wird dadurch erheblich vereinfacht, daß von der Analyse zum Entwurf kein Paradigmenwechsel stattfindet. Entwurfs- und Implementierungsphase sind sehr stark miteinander verzahnt. Das bedeutet, daß jede entworfene Klasse direkt implementiert werden kann.

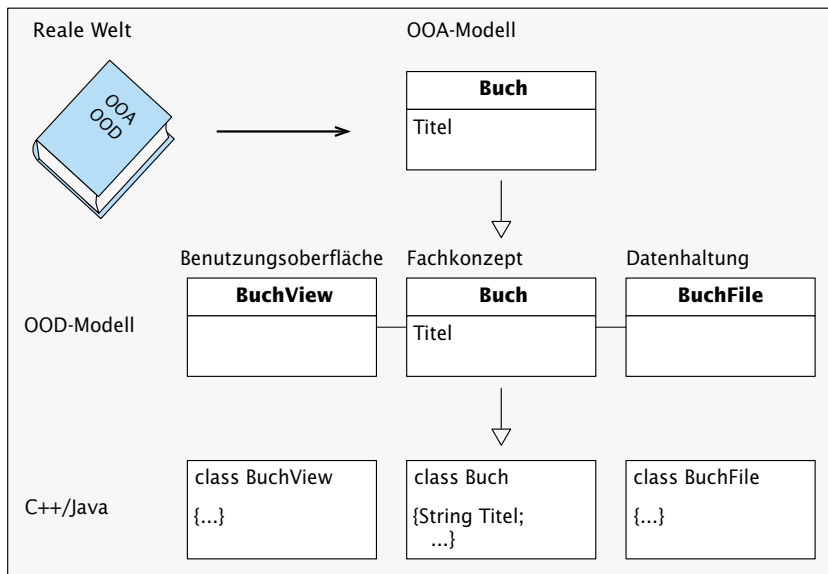
## LE 1 1 Objektorientierte Softwareentwicklung

**Entwurfsziel** Viele der heute »veralteten« Systeme sind bezüglich ihrer anwendungsspezifischen Funktionalität noch ganz »modern«, während ihre Benutzungsoberfläche und ihre Datenhaltung veraltet sind. Um die Benutzungsschnittstelle zu aktualisieren, muß oft das ganze System neu geschrieben werden. Ähnlich sieht es aus, wenn z.B. aus Gründen der Leistungsfähigkeit oder des Datenaustauschs eine andere Datenbank verwendet werden soll. Wir verfolgen daher das Ziel, Fachkonzept, Benutzungsoberfläche und Datenhaltung weitgehend zu entkoppeln. Wie die Benutzungsoberfläche aussieht, hängt ganz entscheidend von dem verwendeten GUI (*graphical user interface*) ab. Die Datenhaltung wird entscheidend durch die verwendete (relationale oder objektorientierte) Datenbank bestimmt. Alternativ kann die Datenhaltung mittels flacher Dateien realisiert werden.

**Drei-Schichten-Architektur** Aus dem Entwurfsziel läßt sich direkt die Verwendung einer Drei-Schichten-Architektur ableiten, d.h. wir trennen die Schichten Benutzungsoberfläche, Fachkonzept und Datenhaltung (Abb. 1.3-1). Das OOA-Modell bildet die erste Version der Fachkonzeptschicht. Das vorhandene OOA-Modell wird unter Gesichtspunkten der Effizienz und der Wiederverwendung überarbeitet. Dazu gehören die Berücksichtigung vorhandener Klassenbibliotheken und Schnittstellen zu anderer Software.

Aus dem Prototyp der Benutzungsoberfläche wird die Schicht der Benutzungsoberfläche erstellt. Im Gegensatz zu einem *quick and dirty*-Prototyp handelt es sich also hier um einen Prototyp, der systematisch weiterentwickelt wird. Die Komponente der Datenhaltung muß für den Anschluß an objektorientierte oder relationale

Abb. 1.3-1:  
Drei-Schichten-  
Architektur



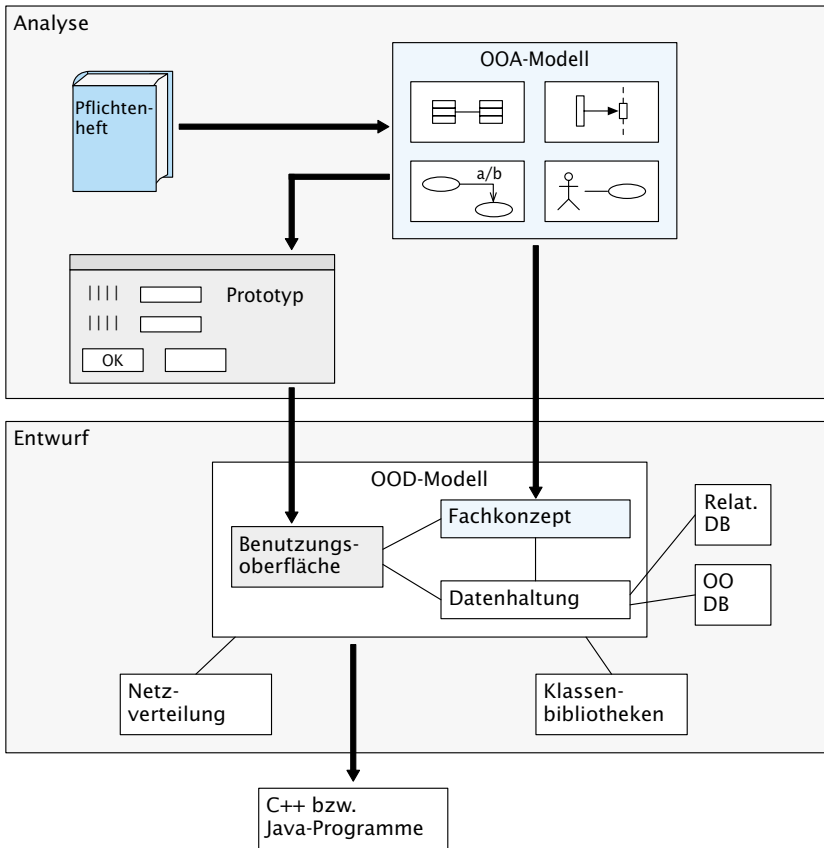


Abb. 1.3-2:  
Zur Abgrenzung  
von Analyse und  
Entwurf

Datenbanken sorgen bzw. eine eigene Datenhaltung realisieren. Bei Client-Server-Anwendungen muß die Software auf mehrere Computersysteme verteilt werden. Diese Zusammenhänge sind in Abb. 1.3-2 beschrieben.

**Welche Produkte sind in der Entwurfsphase zu erstellen?**

Das **OOD-Modell** wird analog zum OOA-Modell dokumentiert. Das OOA-Modell beschreibt die essentielle Struktur und Semantik des Problems, aber noch keine technische Lösung. Im Gegensatz dazu soll das OOD-Modell ein Abbild des späteren Programms sein. Jede Klasse, jedes Attribut und jede Operation des OOD-Modells kommt auch in den Programmen vor. Es werden exakt die gleichen Namen wie im Programm verwendet. Im Gegensatz zum objektorientierten Programmcode zeigt das OOD-Modell das System auf einer höheren Abstraktionsebene und macht vor allem das Zusammenwirken einzelner Elemente deutlich.

Auch beim OOD-Modell werden ein statisches und ein dynamisches Modell erstellt. Das – gegenüber der Analyse wesentlich um-

fangreichere – **statische Modell** soll alle Klassen des Programms enthalten, welche die Architektur des Systems beschreiben. Klassen, die nur Typen beschreiben, z.B. String, werden nicht eingetragen. Die Pakete dienen nicht nur zur Modellierung von Teilsystemen, sondern auch und vor allem zur Darstellung der verschiedenen Schichten. Das **dynamische Modell** ist im Entwurf für alle Anwendungsbereiche von besonders großer Bedeutung. Es ermöglicht eine übersichtliche Beschreibung der komplexen Kommunikation zwischen den Objekten, die anhand des Programmcodes nur schwer nachzuvollziehen ist.

**Analyse (analysis)** Aufgabe der Analyse ist die Ermittlung und Beschreibung der Anforderungen eines Auftraggebers an ein Softwaresystem. Das Ergebnis soll die Anforderungen vollständig, widerspruchsfrei, eindeutig, präzise und verständlich beschreiben.

**Dynamisches Modell** Das dynamische Modell ist der Teil des OOA-Modells, welches das Verhalten des zu entwickelnden Systems beschreibt. Es realisiert außer den Basiskonzepten (Objekt, Klasse, Operation) die dynamischen Konzepte (Geschäftsprozeß, Szenario, Botschaft, Zustandsautomat).

**Entwurf (design)** Aufgabe des Entwurfs ist – aufbauend auf dem Ergebnis der Analyse – die Erstellung der Softwarearchitektur und die Spezifikation der Komponenten, d.h. die Festlegung von deren Schnittstellen, Funktions- und Leistungsumfang. Das Ergebnis soll die zu realisierenden Programme auf einem höheren Abstraktionsniveau widerspiegeln.

**Konzept (concept)** Der Begriff des Konzepts wird in der Informatik im Sinne von Leitidee verwendet, z.B. Konzepte der Programmierung, Konzepte der Objektorientierung. Ein Konzept beschreibt einen definierten Sachverhalt (z.B. eine Klasse) unter einem oder mehreren Gesichtspunkten.

**Methode (method)** Der Begriff »Methode« beschreibt die planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. In der Softwaretechnik wird der Begriff »Methode« als Oberbegriff von →Konzepten, →Notation und →methodischer Vorgehensweise verwendet.

### Methodische Vorgehensweise

**(method)** Eine methodische Vorgehensweise ist eine planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. Sie wird häufig als → Methode bezeichnet.

**Notation (notation)** Darstellung von →Konzepten durch eine festgelegte Menge von grafischen und/oder textuellen Symbolen, zu denen eine Syntax und Semantik definiert ist.

**Objektorientierte Analyse (object oriented analysis)** Ermittlung und Beschreibung der Anforderungen an ein Softwaresystem mittels objektorientierter Konzepte und Notationen. Das Ergebnis ist ein OOA-Modell.

**Objektorientierter Entwurf (object oriented design)** Aufbauend auf dem OOA-Modell erfolgt die Erstellung der Softwarearchitektur und die Spezifikation der Klassen aus Sicht der Realisierung. Das Ergebnis ist das OOD-Modell, das ein Spiegelbild der objektorientierten Programme auf einem höheren Abstraktionsniveau bildet.

**Objektorientierte Softwareentwicklung (object oriented software development)** Bei einer objektorientierten Softwareentwicklung werden die Ergebnisse der Phasen Analyse, Entwurf und Implementierung objektorientiert erstellt. Für letztere werden objektorientierte Programmiersprachen verwendet. Auch die Verteilung auf einem Netz kann objektorientiert erfolgen.

**OOA** →Objektorientierte Analyse

**OOA-Modell** Fachliche Lösung des zu realisierenden Systems, die in einer objektorientierten →Notation modelliert wird. Das OOA-Modell besteht aus



dem →statischen und dem →dynamischen Modell und ist das wichtigste Ergebnis der →Analyse.

**OOD** →Objektorientierter Entwurf

**OOD-Modell** Technische Lösung des zu realisierenden Systems, die in einer objektorientierten →Notation modelliert wird. Das OOD-Modell ist ein Abbild des späteren (objektorientierten) Programms.

**Prototyp** Der Prototyp dient dazu, bestimmte Aspekte vor der Realisierung des Softwaresystems zu überprüfen. Der Prototyp der Benutzungsoberfläche zeigt die vollständige Oberfläche des zukünftigen Systems, ohne daß bereits Funktionalität realisiert ist.

**Statisches Modell** Das statische Mo-

dell realisiert außer den Basiskonzepten (Objekt, Klasse, Attribut) die statischen Konzepte (Assoziation, Vererbung, Paket). Es beschreibt die Klassen des Systems, die Assoziationen zwischen den Klassen und die Vererbungsstrukturen. Außerdem enthält es die Daten des Systems (Attribute). Die Pakete dienen dazu, Teilsysteme zu bilden, um bei großen Systemen einen besseren Überblick zu ermöglichen.

**Systemanalyse** →Analyse

**UML** *Unified Modeling Language*, die von Booch, Rumbaugh und Jacobson bei der *Rational Software Corporation* entwickelt und 1997 von der *OMG (Object Management Group)* als Standard akzeptiert wurde.



Eine (objektorientierte) **Methode** setzt sich aus **Konzepten**, einer **Notation** und einer **methodischen Vorgehensweise** zusammen. Die **UML** bildet zur Zeit den Standard für eine objektorientierte Notation. In der **Analyse** muß ein Fachkonzept des zu realisierenden Systems erstellt werden. Das **OOA-Modell** beschreibt die essentielle Struktur und Semantik des Problems, aber noch keine technische Lösung. Aus dem OOA-Modell wird ein **Prototyp** der Benutzungsoberfläche abgeleitet. Aufgabe des Entwurfs ist es, das Fachkonzept auf einer Plattform unter den geforderten technischen Randbedingungen zu realisieren. Das **OOD-Modell** soll ein Abbild des späteren objektorientierten Programms sein.

**1 Lernziel:** Vorteile der objektorientierten Entwicklung nennen können.

Aufgabe  
5 Minuten

- a** Wodurch wird bei der objektorientierten Softwareentwicklung die gute Durchgängigkeit von der Analyse bis zur Implementierung erreicht?
- b** Welche Vorteile ergeben sich aus dem Klassenkonzept?
- c** Welche Vorteile ergeben sich durch das Konzept der Vererbung?

**2 Lernziel:** Bedeutung der Phasen Analyse und Entwurf in der Sprache des Auftraggebers ausdrücken können.

Aufgabe  
10 Minuten

Stellen Sie sich einen potentiellen Auftraggeber vor, der ein sogenannter DV-Laie ist, z.B. den Inhaber einer Firma für Sportartikel. Sie sollen den Auftrag erhalten, ein maßgeschneidertes kleines Warenwirtschaftssystem für Ihren Auftraggeber zu entwickeln. Ihr Auftraggeber möchte wissen, warum Sie nicht sofort mit der Programmierung anfangen. Erklären Sie ihm, was Analyse und Entwurf sind.



## LE 1 1 Objektorientierte Softwareentwicklung

Aufgabe 3 *Lernziel: Überprüfen, ob die Aufgaben von Analyse und Entwurf sowie deren Abgrenzung gegeneinander verstanden wurde.*  
10–15 Minuten

- a Warum ist es sinnvoll, in der Analyse einen Prototyp der Benutzungsoberfläche zu erstellen?
- b Welche Aufgaben soll das Pflichtenheft erfüllen?
- c Wie lassen sich die Phasen Analyse und Entwurf voneinander abgrenzen?
- d Warum ist es wichtig, in der Analyse von allen Implementierungsdetails zu abstrahieren?
- e Warum handelt es sich bei der Systemanalyse um eine besonders anspruchsvolle Tätigkeit?
- f Warum ist es sinnvoll, die fachliche Funktionalität einer Anwendung, deren Benutzungsschnittstelle und die Datenhaltung strikt zu trennen?

Aufgabe 4 *Lernziel: Erkennen, welche Informationen in der Analyse und welche im Entwurf dokumentiert werden müssen.*  
5–10 Minuten

Der Systemanalytiker Mayer führt bei einer Videothek eine Systemanalyse durch, wobei er folgende Informationen aufnimmt:

- a Für jeden Videofilm sind Titel, Laufzeit und Jahr zu speichern.
- b Die erfaßten Videofilme sind nach Titeln aufsteigend sortiert in der Datenbank *xy* zu speichern.
- c Jede Ausleihe von Videofilmen wird im System gespeichert.
- d Defekte Videofilme werden aus der Videothek entfernt und in der Datei mit einem »L« gekennzeichnet.
- e Das System soll jederzeit einen Überblick über die Ausleihhäufigkeit der einzelnen Filme erlauben.
- f Für die Realisierung der Benutzungsoberfläche wird die Klassenbibliothek *abc* verwendet.
- g Da es sich um eine große Videothek handelt, ist eine Client-Server-Anwendung notwendig, wobei alle zentralen Daten auf dem Server liegen.

Welche der genannten Informationen sind *nicht* Gegenstand der (System-)Analyse?

## 2 Konzepte und Notation der objektorientierten Analyse (Basiskonzepte)



- Erklären können, was ein Objekt ist.
- Externe von internen Objekten unterscheiden können.
- Erklären können, was eine Klasse ist.
- Erklären können, was Objektverwaltung bedeutet.
- Erklären können, was ein Attribut ist.
- Klassenattribut und Objektattribut unterscheiden können.
- Erklären können, was eine Operation ist.
- Objektoperation, Konstruktoroperation und Klassenoperation unterscheiden können.
- UML für Objekt, Klasse, Attribut und Operation anwenden können.
- Objekte und Verbindungen identifizieren und im Objektdiagramm modellieren können.
- Klassen, Attribute und Operationen in einem Text identifizieren und im Klassendiagramm modellieren können.
- Attribute spezifizieren können.

verstehen

anwenden



Das Kapitel 1 sollte bekannt sein, um den Inhalt dieser und der folgenden Lehreinheiten in einen Rahmen einzuordnen.



- 2.1 Objekt 18
- 2.2 Klasse 21
- 2.3 Attribut 25
- 2.4 Operation 30

## 2.1 Objekt

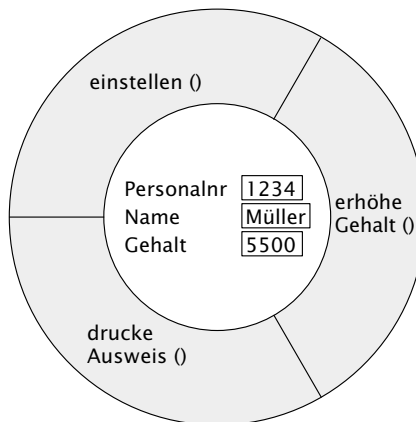
**Definition** Im allgemeinen Sprachgebrauch ist ein Objekt ein Gegenstand des Interesses, insbesondere einer Beobachtung, Untersuchung oder Messung. Objekte können Dinge (z.B. Fahrrad, Büro), Personen (z.B. Kunde, Mitarbeiter) oder Begriffe (z.B. Programmiersprache, Krankheit) sein. In der objektorientierten Softwareentwicklung besitzt ein **Objekt** (*object*) einen bestimmten Zustand und reagiert mit einem definierten Verhalten auf seine Umgebung. Außerdem besitzt jedes Objekt eine Identität, die es von allen anderen Objekten unterscheidet. Ein Objekt kann ein oder mehrere andere Objekte kennen. Wir sprechen von Verbindungen (*links*) zwischen Objekten.

Der **Zustand** (*state*) eines Objekts umfaßt die Attribute bzw. deren aktuelle Werte und die jeweiligen Verbindungen zu anderen Objekten. Attribute sind inhärente, unveränderliche Merkmale des Objekts, während die Attributwerte Änderungen unterliegen können.

Das **Verhalten** (*behavior*) eines Objekts wird durch eine Menge von Operationen beschrieben. Eine Änderung oder eine Abfrage des Zustandes ist nur mittels der Operationen möglich.

**Beispiel** Ein Mitarbeiter besitzt eine Personalnummer, einen Namen und erhält ein bestimmtes Gehalt. Neue Mitarbeiter werden eingestellt, das Gehalt vorhandener Mitarbeiter kann erhöht werden und es kann ein Mitarbeiterausweis gedruckt werden. Wie die Abb. 2.1-1 zeigt, werden die Attribute durch die Operationen vor der Außenwelt verborgen.

Abb. 2.1-1:  
Mitarbeiter-Objekt



**Notation** Das Objekt wird in der UML als Rechteck dargestellt (Abb. 2.1-2), das in zwei Felder aufgeteilt werden kann. Im oberen Feld wird das Objekt wie folgt bezeichnet:

:Klasse bei einem anonymem Objekt wird nur der Klassenname angegeben.



Abb. 2.1-2:  
Notation von  
Objekten

Objekt: Klasse wenn das Objekt über einen Namen angesprochen werden soll.

Objekt wenn der Objektname ausreicht, um das Objekt zu identifizieren und der Name der Klasse aus dem Kontext ersichtlich ist.

Die Bezeichnung eines Objekts wird immer unterstrichen. Anonyme Objekte werden verwendet, wenn es sich um irgendein Objekt der Klasse handelt. Objektname dienen dazu, ein bestimmtes Objekt der Klasse für den Systemanalytiker zu benennen.

Im unteren Feld werden – optional – die im jeweiligen Kontext relevanten Attribute des Objekts eingetragen. Die UML ermöglicht folgende Alternativen:

Attribut : Typ = Wert

Attribut = Wert empfehlenswert, da der Typ bereits bei der Klasse definiert ist und diese Angabe daher redundant ist.

Attribut sinnvoll, wenn der Wert des Attributs nicht von Interesse ist.

Die Operationen, die ein Objekt ausführen kann, werden in der UML nicht angegeben.

Objekte und ihre Verbindungen untereinander werden im **Objektdiagramm** (*object diagram*) spezifiziert (Abb. 2.1-3). Es beschreibt Objekte, Attributwerte und Verbindungen zwischen Objekten zu einem bestimmten Zeitpunkt. Objektdiagramme sind sozusagen Momentaufnahmen bzw. Schnappschüsse des Systems. Meistens werden anonyme Objekte verwendet. Konkrete Objekte sind nur in Ausnahmefällen interessant.

**Hinweis**  
In diesem Buch wird für viele UML-Diagramme Farbe verwendet. Diese Farbe ist nicht fester Bestandteil der UML-Notation. Ihre Verwendung ist jedoch nach /UML 97/ zulässig.

Objektdiagramm

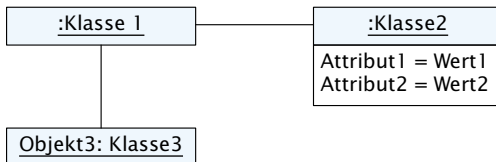


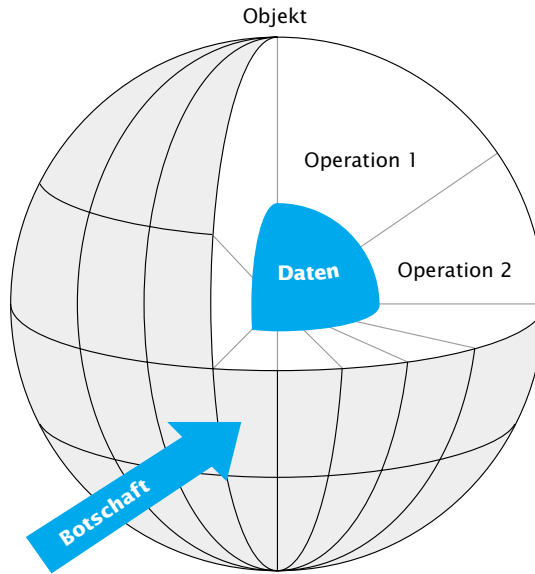
Abb. 2.1-3:  
Notation des  
Objektdiagramms

Zustand und Verhalten eines Objekts bilden eine Einheit. Wir sagen auch: ein Objekt kapselt Zustand (Daten) und Verhalten (Operationen). Die Daten eines Objekts können nur mittels der Operationen gelesen und geändert werden. Das bedeutet, daß die Repräsentation dieser Daten nach außen verborgen sein soll. Wir sagen: ein Objekt realisiert das **Geheimnisprinzip** (Abb. 2.1-4).

Datenkapsel und  
Geheimnisprinzip

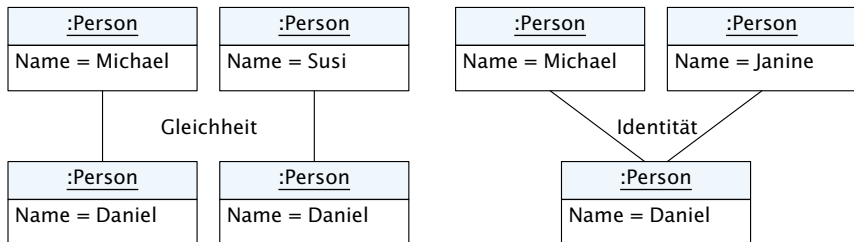
## LE 2 2 Konzepte und Notation für OOA

Abb. 2.1-4:  
Objekt realisiert  
das Geheimnis-  
prinzip



Objektidentität Die **Objektidentität** (*object identity*) ist die Eigenschaft, die ein Objekt von allen anderen Objekten unterscheidet. Sie bedeutet, daß alle Objekte aufgrund ihrer Existenz unterscheidbar sind, auch wenn sie zufällig identische Attributwerte besitzen. Die Identität eines Objekts kann sich nicht ändern. Keine zwei Objekte können dieselbe Identität besitzen. Besitzen zwei Objekte – mit unterschiedlichen Identitäten – dieselben Attributwerte, so sprechen wir von der Gleichheit der Objekte. Wir unterscheiden also zwischen identischen und gleichen Objekten. In der Abb. 2.1-5 haben die Personen Michael und Susi beide ein Kind mit dem Namen Daniel (Gleichheit), während Michael und Janine Eltern desselben Kindes sind (Identität).

Abb. 2.1-5:  
Gleichheit und  
Identität von  
Objekten



Objektname Der Objektname identifiziert ein Objekt im Objektdiagramm. Im Gegensatz zur Objektidentität muß er nur im betrachteten Kontext, d.h. innerhalb eines Diagramms, eindeutig sein. Besitzen Objekte in verschiedenen Diagrammen denselben Namen, so kann es sich um unterschiedliche Objekte handeln.

Alle gleichartigen Objekte, d.h. Objekte mit denselben Operationen und gleichen Attributen – aber im allgemeinen unterschiedlichen Attributwerten! – gehören zu der gleichen Klasse. Jedes Objekt ist Exemplar einer Klasse. Auf das Konzept der Klasse gehen wir in Kapitel 2.2 noch ausführlich ein.

Klasse

Kapitel 2.2



Es ist wichtig zwischen externen und internen Objekten zu unterscheiden. Externe Objekte existieren in der realen Welt, während interne Objekte für ein Softwaresystem relevant sind. Betrachten wir beispielsweise den realen Kunden Müller, der Bankgeschäfte durchführt. Herr Müller ist in seiner Freizeit ein begeisterter Golfspieler, eine Eigenschaft, die für die Modellierung des internen Objekts Müller in unserem Softwaresystem völlig uninteressant ist. Wird aus dem externen Objekt das interne Objekt abgeleitet, so müssen wir die für das jeweilige Modell (hier: Bankgeschäfte) relevanten Eigenschaften abstrahieren. Soll dagegen ein Golfturnier modelliert werden, so sind sicher andere Eigenschaften interessant. Beim Übergang von der realen Welt ins OOA-Modell tritt folgender Effekt auf: in der realen Welt sind Objekte aktiv (z.B. Herr Müller schickt Überweisungsaufträge an die Bank). Im OOA-Modell sind die entsprechenden (internen) Objekte passiv (z.B. werden über den Kunden Müller Daten und Vorgänge gespeichert).

externe und interne Objekte

Die Begriffe *instance*, *class instance* und Exemplar werden synonym für den Begriff Objekt gebraucht. Der Begriff »Instanz«, der in der deutschen Literatur häufig verwendet wird, ist ein Anglizismus, der auf einer fehlerhaften Übersetzung von *instance* beruht.

verwandte Begriffe

## 2.2 Klasse

Eine **Klasse** definiert für eine Kollektion von Objekten deren Struktur (Attribute), Verhalten (Operationen) und Beziehungen. Sie besitzt einen Mechanismus, um neue Objekte zu erzeugen (*object factory*). Jedes erzeugte Objekt gehört zu genau einer Klasse. Unter den Beziehungen (*relationships*) sind Assoziationen und Vererbungsstrukturen zu verstehen (siehe Kapitel 2.5 und 2.6). Das Verhalten (*behavior*) einer Klasse wird durch die Botschaften (Nachrichten) beschrieben, auf die diese Klasse bzw. deren Objekte reagieren können. Jede Botschaft aktiviert eine Operation gleichen Namens.

Definition

Kapitel 2.5 und 2.6



Die beiden Mitarbeiter-Objekte in der Abb. 2.2-1 besitzen die gleichen Attribute und Operationen. Sie gehören daher beide zur Klasse Mitarbeiter.

Beispiel

Für die Darstellung von Klassen gibt es verschiedene Möglichkeiten (Abb. 2.2-2). Die entsprechenden Kurzformen werden verwendet, wenn die fehlenden Details unwichtig sind oder in einem ande-

Notation

## LE 2 2 Konzepte und Notation für OOA

Abb. 2.2-1:  
Klasse Mitarbeiter

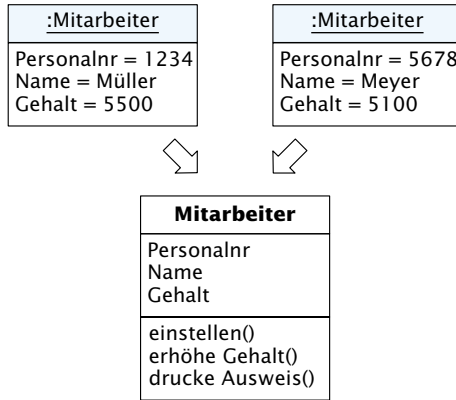
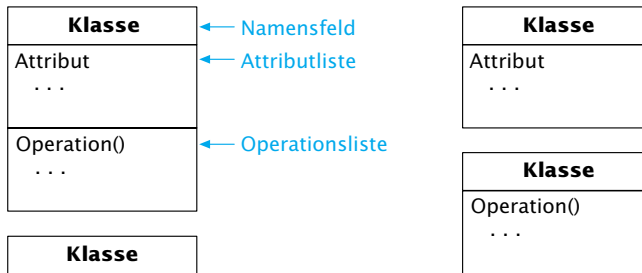


Abb. 2.2-2:  
Notation von  
Klassen



Kapitel 2.3 und 2.4

ren Klassendiagramm definiert sind. Attribute und Operationen können näher spezifiziert werden. Wir gehen in Kapitel 2.3 und 2.4 ausführlich darauf ein. Der Klassenname wird immer fettgedruckt, zentriert dargestellt und beginnt mit einem Großbuchstaben.

Klassendiagramm

Die Klassensymbole werden zusammen mit weiteren Symbolen, z.B. Assoziation und Vererbung in das **Klassendiagramm** eingetragen. Das Klassendiagramm beschreibt das statische Modell des Systems. Bei großen Systemen ist es im allgemeinen sinnvoll oder notwendig, mehrere Klassendiagramme zu erstellen.

Klassenname

Der **Klassenname** ist stets ein Substantiv im Singular, das durch ein Adjektiv ergänzt werden kann. Er beschreibt also ein einzelnes Objekt der Klasse. Beispiele: Mitarbeiter, PKW, Kunde. Der Klassenname muß innerhalb eines Pakets (siehe Kapitel 2.7), besser jedoch innerhalb des gesamten Systems, eindeutig sein. Bei Bedarf wird er in der UML wie folgt erweitert: Paket: : Klasse.

Kapitel 2.7

Das Namensfeld einer Klasse kann in der UML um

- einen Stereotypen und
- eine Liste von Merkmalen erweitert werden.

Stereotyp

Ein **Stereotyp** (*stereotype*) klassifiziert Elemente (z.B. Klassen, Operationen) des Modells. Die UML enthält einige vordefinierte Stereotypen, und es können weitere Stereotypen definiert werden. Ste-

reotypen werden in französischen Anführungszeichen (*guillemets*) mit Spitzen nach außen angegeben, z.B. «Stammdaten». In der Abb. 2.2-3 sagt der Stereotyp aus, daß die Objekte der Klasse Mitarbeiter als Stammdaten geführt werden.

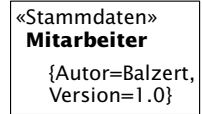


Abb. 2.2-3:  
Stereotyp und Merkmale einer Klasse

Ein **Merkmal** (*property*) beschreibt Eigenschaften eines bestimmten Elements des Modells. Mehrere Merkmale können in einer Liste zusammengefaßt werden. Sie werden in der folgenden Form beschrieben: {Schlüsselwort = Wert, ...}



Objektbasierte Programmiersprachen wie Ada verwenden den Begriff des abstrakten Datentyps. Hier handelt es sich um ein Konzept der Entwurfsphase. Der **abstrakte Datentyp** (ADT) läßt sich als »benutzerdefinierter Datentyp« umschreiben. Er wird ausschließlich über seine Operationen definiert, die auf Exemplare dieses Typs angewendet werden. Die interne Repräsentation der Daten und die Wahl der Algorithmen zur Realisierung der Operationen sind verkapselt, d.h. nach außen nicht sichtbar. Der abstrakte Datentyp realisiert folglich das Prinzip der Trennung von Schnittstelle und Implementierung. Von einem abstrakten Datentyp können beliebig viele Exemplare erzeugt werden.

abstrakter  
Datentyp vs. Klasse

Die Klasse stellt eine Form des abstrakten Datentyps dar. Sie beschreibt jedoch außer den Operationen des abstrakten Datentyps auch die zugrundeliegende Datenstruktur (Attribute der Klasse). Warum werden die verborgenen Attribute der Klasse dargestellt? Dieses Sichtbarmachen der Attribute ist aus Gründen der Vererbung notwendig. Außerdem werden die Attribute aus methodischer Sicht zur Identifikation der Klassen benötigt. Auf die Attribute der Klasse darf nur mittels der Operationen zugegriffen werden. Die Attribute sind zwar sichtbar für den Systemanalytiker, jedoch – mit Ausnahme in Vererbungsstrukturen – nicht sichtbar für andere Objekte und Klassen.

Wie aus der Definition der Klasse zu entnehmen ist, besitzt jede Klasse einen Mechanismus, um Objekte zu erzeugen. Es gibt jedoch auch Klassen, von denen keine Objekte erzeugt werden können. Sie werden abstrakte Klassen genannt. Eine abstrakte Klasse wird entweder durch einen kursiven Klassennamen oder das Merkmal {abstract} gekennzeichnet. Das Konzept der abstrakten Klasse ist besonders für die Vererbung von Bedeutung und wird dort ausführlich erläutert (Kapitel 3.7).

abstrakte Klasse

Kapitel 3.7  
Klasse oder Typ?



Die Begriffe »Klasse« und »Typ« werden oft synonym verwendet. Genau genommen gibt es jedoch einen Unterschied, den wir im folgenden betrachten wollen. Der **Typ** (*type*) legt fest, auf welche Botschaften das Objekt reagieren kann, d.h. er definiert die Schnittstelle eines Objekts. Die **Klasse** (*class*) definiert, wie Objekte implementiert werden, d.h. sie definiert den internen Zustand der Objekte und die Implementierung der Operationen. Eine Klasse ist eine mögliche Implementierung eines Typs. Das bedeutet, daß die Klasse



## LE 2 2 Konzepte und Notation für OOA

die Implementierung von Objekten definiert, während der Typ festlegt, wie diese Objekte verwendet werden können. Kurz ausgedrückt können wir sagen: Die Klasse implementiert den Typ.

In Übereinstimmung mit den meisten Methoden, Werkzeugen und Programmiersprachen zur objektorientierten Softwareentwicklung verwenden wir in diesem Buch für Analyse- und Entwurfsmodelle durchgängig den Begriff der »Klasse«. Auch die Programmiersprachen C++ und Java verwenden den Klassenbegriff, um sowohl den Typ als auch dessen Implementierung zu beschreiben.

Kurzbeschreibung  
der Klasse

Jede Klasse soll einen ganz bestimmten Zweck innerhalb des Softwaresystems erfüllen. Wird für jede Klasse eine **Kurzbeschreibung** von wenigen Zeilen Umfang verlangt, dann wird dieser Zweck deutlich herausgestellt. Wir erweitern daher die UML wie in folgendem Beispiel.

Beispiel **Klasse Student**  
**Studierender, der an einer Hochschule immatrikuliert ist.**

Objekt kennt  
seine Klasse

Jedes Objekt »weiß«, zu welcher Klasse es gehört. Da alle Objekte zwar unterschiedliche Attributwerte, jedoch gleiche Operationen besitzen, ist es sinnvoll, die Operationen und deren Spezifikationen der Klasse zuzuordnen. Da jedes Objekt seine Klasse kennt, kann es dort alle benötigten Operationen vorfinden.

Umgekehrt »weiß« eine Klasse nicht, welche Objekte sie »besitzt« bzw. welche Objekte von ihr erzeugt wurden. Da dieses Wissen jedoch ausgesprochen nützlich wäre, gehen wir *in der Systemanalyse* davon aus, daß eine Klasse ihre Objekte kennt, d.h. die Klasse »führt Buch« über das Erzeugen und Löschen ihrer Objekte. Wir nennen diese Eigenschaft **Objektverwaltung** (Abb. 2.2-4). Damit erhält die Klasse die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte einer Klasse durchzuführen (*class extension, object warehouse*). Beachten Sie, daß diese Vereinfachung nur in der Analyse gilt und im Entwurf und in der Implementierung je nach verwendeter Umgebung vom Programmierer realisiert werden muß oder von der verwendeten Software generiert wird. Die verschiede-

Objektverwaltung

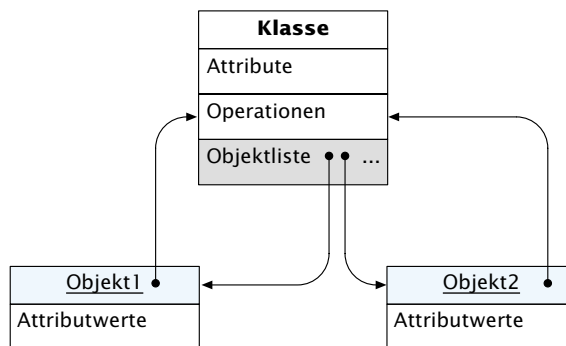


Abb. 2.2-4:  
Objektverwaltung

↩️ nen Realisierungsmöglichkeiten werden im Kapitel 10 ausführlich erläutert. Kapitel 10

Verwechseln Sie nicht die Klasse und die Menge aller Objekte dieser Klasse (*extension*). Die Klasse ist eine Abstraktion, die Gemeinsamkeiten von Objekten und Regeln zu ihrer Erzeugung beschreibt. Eine Menge von Objekten ist dagegen einfach eine Ansammlung von Objekten. Die Objektverwaltung wird beispielsweise mittels einer solchen Objektmenge realisiert. Klasse vs. Menge aller Objekte

Der Begriff der »Klasse« hat sich inzwischen allgemein durchgesetzt. Wenn es um die Spezifikation von Klassen geht, wird teilweise der Begriff »Typ« verwendet. verwandte Begriffe

## 2.3 Attribut

Die **Attribute** beschreiben die Daten, die von den Objekten einer Klasse angenommen werden können. Jedes Attribut ist von einem bestimmten Typ. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch unterschiedliche Attributwerte. Definition

Die Abb. 2.3-1 zeigt die Klasse Student und eines ihrer Objekte. Während die Klasse festlegt, welche Attribute ihre Objekte besitzen, enthalten die Objekte die Attributwerte. Wie das Beispiel zeigt, darf das Feld für den Attributwert leer sein. Wir sprechen von einem optionalen Attribut. Das bedeutet, daß dieses Attribut nicht bei der Erzeugung des Objekts, sondern zu irgendeinem späteren Zeitpunkt – evtl. auch nie – einen definierten Wert erhält. Beispiel

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="text-align: center; padding: 2px;">Student</th> </tr> <tr> <td style="padding: 2px;">                     Matrikelnr                      Name                      Geburtsdatum                      Immatrikulation                      Vordiplom                      Noten                 </td> </tr> </table>	Student	Matrikelnr Name Geburtsdatum Immatrikulation Vordiplom Noten	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="text-align: center; padding: 2px;">:Student</th> </tr> <tr> <td style="padding: 2px;">                     Matrikelnr = 7002345                      Name = (Hans, Meyer)                      Geburtsdatum = 4.7.1974                      Immatrikulation = 1.9.1994                        Noten = ((2.3, Analysis),                                (1.3, Informatik))                 </td> </tr> </table>	:Student	Matrikelnr = 7002345 Name = (Hans, Meyer) Geburtsdatum = 4.7.1974 Immatrikulation = 1.9.1994  Noten = ((2.3, Analysis), (1.3, Informatik))
Student					
Matrikelnr Name Geburtsdatum Immatrikulation Vordiplom Noten					
:Student					
Matrikelnr = 7002345 Name = (Hans, Meyer) Geburtsdatum = 4.7.1974 Immatrikulation = 1.9.1994  Noten = ((2.3, Analysis), (1.3, Informatik))					

Das Attribut Vordiplom besitzt  
– noch – keinen Wert.

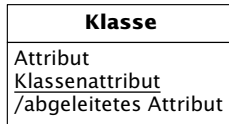
Abb. 2.3-1:  
Klasse Student und  
Student-Objekt

Attribute werden durch ihren Namen und ihren Typ beschrieben (Abb. 2.3-2). Optional können angegeben werden: Notation

- Anfangswert (*initial-value*)  
Er legt fest, welchen Wert ein neu erzeugtes Objekt für dieses Attribut annimmt.
- Liste von Merkmalen  
Hier können die Merkmale bzw. die Eigenschaften des Attributs angegeben werden. Wir gehen später in diesem Kapitel bei der Spezifikation der Attribute genauer darauf ein.

## LE 2 2 Konzepte und Notation für OOA

Abb. 2.3-2:  
Notation für  
Attribute




Attribut: Typ = Anfangswert  
{Merkmal1, Merkmal2, ...}

Der besseren Lesbarkeit halber tragen wir im Analysemodell für ein Attribut nur dessen Namen in das Klassendiagramm ein und beschreiben seine weiteren Informationen separat (siehe Abb. 2.3-2). Auf Klassenattribute und abgeleitete Attribute gehen wir später in diesem Kapitel ein.

Attributname Der **Attributname** muß im Kontext der Klasse eindeutig sein. Er beschreibt die gespeicherten Daten. Im allgemeinen wird ein Substantiv dafür verwendet. In der UML beginnen Attributnamen generell mit einem Kleinbuchstaben. Bei deutschen Bezeichnungen beginnen wir wegen der besseren Lesbarkeit jedoch Attributnamen mit einem Großbuchstaben, wenn es sich um ein Substantiv handelt. Wird die englische Sprache zur Modellierung verwendet, so sollte die UML-Regel angewendet werden. Da ein Attributname nur innerhalb der Klasse eindeutig ist, verwenden wir außerhalb des Klassenkontextes die Bezeichnung *Klasse. Attribut*.

Geheimnisprinzip Die Attribute dürfen nur über die Operationen der zugehörigen Klasse geändert und gelesen werden. Wir sagen daher: Die Attribute sind zwar sichtbar für den Systemanalytiker, aber nicht sichtbar für andere Klassen bzw. deren Objekte. Ein Attribut ist daher äquivalent zu zwei Zugriffsoperationen, je eine zum Lesen und Schreiben des Attributwertes /OMA 97/.

### Unterschiede zum *Entity-Relationship-Modell*

Für Leser, die mit *Entity-Relationship-Modellen* vertraut sind, sind hier zwei wichtige Unterschiede hervorzuheben. 

- Künstliche Schlüsselattribute sind im Klassendiagramm nicht notwendig.

Beim *Entity-Relationship-Modell* ist ein eindeutiges Identifizieren der Objekte nur mittels eines Schlüsselattributs möglich. Der Schlüssel kann sich auch aus mehreren Attributen zusammensetzen. In diesem Fall ist es wichtig, daß er aus einer minimalen Kombination von Attributen gebildet wird. Minimal bedeutet, daß die eindeutige Identifizierbarkeit verloren geht, wenn ein Attribut entfernt wird. Es ist möglich, daß ein fachlich notwendiges Attribut gleichzeitig als Schlüssel fungiert (z.B. Matrikelnr in Abb. 2.3-1). Andernfalls muß beim *Entity-Relationship-Modell* ein künstliches Schlüsselattribut hinzugefügt werden.

- Die Normalisierung der Attribute ist im Klassendiagramm nicht notwendig.

Die Attribute einer Klasse müssen nicht die erste Normalform der relationalen Datenbanken erfüllen. Die Entscheidung, ob die Daten

normalisiert werden müssen und welche Normalform ggf. zu wählen ist, soll erst in der Entwurfsphase getroffen werden. Bei der objektorientierten Modellierung definiert der Systemanalytiker die Attribute – frei von irgendwelchen technischen Randbedingungen – ausschließlich unter problemadäquaten Gesichtspunkten. Beispielsweise besteht das Attribut *Noten* in der Abb. 2.3-1 aus einer Liste von Einzelnoten.

Zwischen den Attributwerten eines Objekts können Beziehungen existieren, die während der Ausführung des Systems unverändert erhalten bleiben müssen. Wir sprechen hier von **Restriktionen** (*constraints*). Eine Restriktion wird auch als Invariante bezeichnet. Es ist eine Zusicherung, die immer wahr sein muß.

Restriktionen  
(*constraints*)

Für die Attribute der Klasse *Student* in Abb. 2.3-1 gilt:  
{*Vordiplom* > *Immatrikulation* > *Geburtsdatum*}

Beispiel 1

Für die Klasse *Artikel* mit den Attributen *Einkaufspreis* und *Verkaufspreis* gilt, daß der Verkaufspreis mindestens 150 Prozent des Einkaufspreises betragen soll. Dann muß durch die Implementierung sichergestellt werden, daß beim Ändern des einen Preises auch der andere geändert wird.

Beispiel 2

{*Verkaufspreis* >= 1.5 \* *Einkaufspreis*}

Außer den oben beschriebenen (Objekt-)Attributen sind manchmal Klassenattribute notwendig. Ein **Klassenattribut** liegt vor, wenn nur ein Attributwert für alle Objekte einer Klasse existiert. Klassenattribute existieren auch dann, wenn es zu einer Klasse – noch – keine Objekte gibt. Um die Klassenattribute von den (Objekt-)Attributen zu unterscheiden, werden sie in der UML unterstrichen (z.B. *Klassenattribut*).

Klassenattribut

Der Wert eines **abgeleiteten Attributs** (*derived attribute*) kann jederzeit aus anderen Attributwerten berechnet werden. Abgeleitete Attribute werden mit dem Präfix »/« gekennzeichnet (Abb. 2.3-3). Ein abgeleitetes Attribut darf nicht geändert werden.

abgeleitete  
Attribute

In der UML ist nicht festgelegt, wie der **Typ** eines Attributs definiert wird. Um ein standardisiertes OOA-Modell zu erstellen, verwenden wir in der Systemanalyse folgende Typen:

- Standardtypen,
- Aufzählungstypen,
- (elementare) Klassen,
- **list of** Typ, wobei ein beliebiger Typ erlaubt ist.

Mit anderen Worten:

Typ = [Standardtyp | Aufzählungstyp | Klasse | **list of** Typ]

In der Analyse dient die Typdefinition dem Zweck, das Attribut aus fachlicher Sicht möglichst präzise zu beschreiben. In Entwurf und Implementierung wird dann in Abhängigkeit von der gewählten Programmiersprache der Typ neu definiert.

Attributtyp

<b>Person</b>
Geburtsdatum /Alter

Abb. 2.3-3:  
Abgeleitetes Attribut

## LE 2 2 Konzepte und Notation für OOA

Standardtypen Als **Standardtypen** stehen dem Systemanalytiker zur Verfügung:

- String = String (Länge)
- Int: ganze Zahl von 32 Bit
- UInt: positive ganze Zahl 32 Bit
- Float: Gleitkommazahl 32 Bit
- Double: Gleitkommazahl 64 Bit
- Fixed (Vorkommastellen, Nachkommastellen): Festkommazahl
- Boolean
- Date
- Time

Aufzählungstyp Für einen **Aufzählungstyp** sind anzugeben:

- Bereich  
Hier werden alle Werte aufgezählt, die das Attribut annehmen kann.
- Selektionsart  
Es soll möglich sein, ein oder mehrere Werte zu selektieren, wobei individuell festgelegt wird, wie viele das sein können. Daher sind Angaben zur minimalen und zur maximalen Anzahl der zu selektierenden Elemente notwendig. Als Voreinstellung wird angenommen, daß genau ein Wert selektiert wird.
- Die Werteliste kann erweiterbar sein. In diesem Fall kann der spätere Benutzer neue Werte eingeben, die in die Liste permanent aufgenommen werden. Wir gehen davon aus, daß diese Erweiterbarkeit standardmäßig gilt.

Für die Definition von Aufzählungstypen verwenden wir folgende Notation:

{values: W1, W2, W3,	Wertebereich.
select: 1..n,	Selektionsart mit minimaler und maximaler Anzahl.
noAdd}	Festlegung, daß der Wertebereich <i>nicht</i> erweiterbar ist.

Wenn bei der Selektionsart und der Erweiterbarkeit die Voreinstellungen gelten, dann reicht die Angabe der Werte, d.h. {values: W1, W2, W3}.

Alternativ kann ein Aufzählungstyp auch mit Hilfe des Klassenkonstrukts beschrieben werden. Der Typ wird als Klassenname eingetragen und mit dem Stereotypen «enumeration» gekennzeichnet /UML 97a/. Die Werte des Aufzählungstyps werden als Attribute eingetragen.

Der Typ eines Attributs kann selbst wieder durch eine Klasse beschrieben werden. Wir bezeichnen diese Klassen als **elementare Klassen** (*support classes*) und tragen sie im Gegensatz zu den (Architektur-) Klassen nicht in das Klassendiagramm des Gesamtmodells ein. Sie werden in der Regel – in Abhängigkeit von der jeweiligen Anwendung – einmal definiert und bei jedem Projekt wiederverwendet. Um Konflikte mit Attributnamen zu vermeiden, verwende ich bei den Namen selbstdefinierter Typen das Postfix »T«.

Damit ergeben sich für die Klasse Student (Abb. 2.3-1) folgende Attributtypen:

- Matrikelnr: String (7)
- Name: NameT
- Geburtsdatum: Date
- Immatrikulation: Date
- Vordiplom: Date
- Noten: List of NoteT

Bei den Typen NameT und NoteT handelt es sich um elementare Klassen (Abb. 2.3-4).

Beim Aufzählungstyp NotenwertT soll genau ein Wert ausgewählt werden, wobei die Liste für den Benutzer nicht erweiterbar sein darf. Daher ergibt sich für NotenwertT folgende Definition: {values: 1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.7, 4.0, 5.0, noAdd}

NameT	NoteT
Vorname: String Nachname: String	Fach: String Wert: NotenwertT

Abb. 2.3-4:  
Elementare Klassen

Besitzt ein Objekt Attribute, die selbst wieder Objekte sind, so wird es als **komplexes Objekt** (*composite object, complex object, structured object, molecular object*) bezeichnet /Bertino, Matino 93/. Dieses (Unter-) Objekt kann ebenfalls komplex sein. Rekursive Objekte entstehen, wenn die Klasse eines Objekts dieselbe ist wie die eines direkten oder indirekten Unterobjekts.

### Attributspezifikation

Um aus einem OOA-Modell den Prototyp der Benutzungsoberfläche abzuleiten muß jedes Attribut durch folgende Angaben spezifiziert werden:

- 1 Name
- 2 Typ
- 3 Anfangswert
- 4 Muß-Attribut (*mandatory*)
- 5 Schlüssel (*key*)
- 6 Attributwert nicht änderbar (*frozen*)
- 7 Einheit
- 8 Beschreibung

Auf die Angaben 1 bis 3 sind wir bereits eingegangen. Ein Muß-Attribut 4 liegt vor, wenn der Attributwert beim Erzeugen des Objekts eingetragen werden muß. Ein Attribut ist Schlüssel 5, wenn es jedes Objekt innerhalb einer Klasse eindeutig identifiziert. Diese Information wird unter anderem für Überprüfungen bei der Dialogeingabe verwendet. Sind mehrere Attribute einer Klasse mit *key* gekennzeichnet, dann bilden sie einen zusammengesetzten Schlüssel. Attribute, deren Wert nicht geändert werden kann 6 – nachdem er ein-

## LE 2 2 Konzepte und Notation für OOA

mal definiert wurde – werden mit {frozen} gekennzeichnet. Die Angabe der Einheit **7** wird für die Benutzungsoberfläche benötigt, z.B. Fahrzeit in [min], gefahrene Strecke in [km]. Falls notwendig, muß die Bedeutung des Attributs noch durch eine Beschreibung **8** erläutert werden. Bei abgeleiteten Attributen enthält diese Beschreibung die Ableitungsregel.

Notation Attribut-  
spezifikation  
(UML-Erweiterung)

Für die Spezifikation der Attribute verwenden wir die Liste der Merkmale in der folgende Form, die bei Bedarf leicht modifiziert werden kann:

Name: Typ = Anfangswert  
{mandatory, key, frozen, Einheit: . . . . , Beschreibung: . . . }

Fehlen die blauen Angaben, dann gelten die Voreinstellungen, d.h. das Attribut ist ein Kann-Attribut, *kein* Schlüsselattribut und beliebig änderbar. Die Angabe der Einheit entfällt, wenn sie für die Benutzungsoberfläche nicht benötigt wird. Die Beschreibung kann immer dann weggelassen werden, wenn der Name selbsterklärend ist.

**Beispiel** Wir spezifizieren als Beispiel die Attribute der Klasse Student aus der Abb. 2.3-1.

- Matrikelnr: String(7) {mandatory, key, frozen}
- Name: NameT {mandatory}
- Geburtsdatum: Date {mandatory}
- Immatrikulation: Date {mandatory, Beschreibung: Datum des Studienbeginns}
- Vordiplom: Date {Beschreibung: Datum der abschließenden Vordiplomprüfung}
- Noten: List of NoteT
- Restriktionen: {Geburtsdatum < Immatrikulation < Vordiplom}

verwandte Begriffe

In objektorientierten Programmiersprachen wird anstelle von Attributen auch von Member-Variablen oder von *instance variables* gesprochen.

## 2.4 Operation

**Definition** Eine **Operation** ist eine ausführbare Tätigkeit. Alle Objekte einer Klasse verwenden dieselben Operationen. Jede Operation kann auf alle Attribute eines Objekts dieser Klasse direkt zugreifen. Die Menge aller Operationen wird als das Verhalten der Klasse oder als die Schnittstelle der Klasse bezeichnet.

**Beispiel** Auf jedes Objekt der Klasse Student sind die angegebenen Operationen anwendbar (Abb. 2.4-1).

Notation

Operationen werden analog zu den Attributen in das Klassensymbol eingetragen (Abb. 2.4-2). Auch für jede Operation kann eine Liste von Merkmalen angegeben werden. Ein Merkmal ist beispiels-

## 2.4 Operation LE 2

Student
Matrikelnummer Name Geburtsdatum Immatrikulation Vordiplom Noten <u>Anzahl</u>
immatrikulieren() exmatrikulieren() drucke Studienbescheinigung() notiere Noten() berechne Durchschnitt() <u>drucke Vordiplomliste()</u> anmelde Praktikum() drucke Prakt.bescheinigung()

Firma
Name Ort Anzahl Mitarbeiter Branche

Abb. 2.4-1: Klassen Student und Firma

Klasse
Operation () <u>Klassenoperation ()</u> <i>abstrakte Operation ()</i>

Operation () {Merkmal1, Merkmal2, ...}

Abb. 2.4-2: Notation für Operationen



weise {abstract}, das eine abstrakte Operation kennzeichnet. Wir gehen im Kapitel 2.6 auf abstrakte Operationen ein. Auf Klassenoperationen gehen wir später in diesem Kapitel ein.

Kapitel 2.6

Wir unterscheiden drei Arten von Operationen:

- Objektoperationen, kurz Operationen genannt,
- Konstruktoroperationen und
- Klassenoperationen.

Diese Kategorisierung ermöglicht beim Erstellen des Analysemodells eine systematische Zuordnung der Operationen zu den Klassen.

**Objektoperationen** oder kurz Operationen werden stets auf ein einzelnes (bereits existierendes) Objekt angewendet. Typische Beispiele dafür sind die Operationen `drucke Studienbescheinigung()`, `notiere Note()` und `berechne Durchschnitt()` der Abb. 2.4-1. Auch `exmatrikulieren()`, die ein Objekt der Klasse Student löscht, ist eine Objektoperation. Da diese Operationen jeweils auf *einen* Studenten angewendet werden, gehören sie zur Klasse Student.

Objektoperation

Eine **Konstruktoroperation** erzeugt ein neues Objekt und führt entsprechende Initialisierungen und Datenerfassungen durch. Bei der Operation `immatrikulieren()` (Abb. 2.4-1) handelt es sich um eine derartige Operation. Auch diese Operation wird bei der Klasse Student eingetragen.

Konstruktoroperation

Eine **Klassenoperation** ist eine Operation, die der jeweiligen Klasse zugeordnet ist und nicht auf ein einzelnes Objekt der Klasse angewendet werden kann. Sie wird durch Unterstreichen gekennzeichnet, z.B. `drucke Vordiplomliste()`.

Klassenoperation



## LE 2 2 Konzepte und Notation für OOA

In der Systemanalyse verwenden wir Klassenoperationen in folgenden Fällen:

- 1 Die Operation manipuliert Klassenattribute ohne Beteiligung eines einzelnen Objekts. Ein Beispiel ist `erhöhe Stundenlohn()` der Abb. 2.4-3. Diese Aufgabe ist unabhängig von einem ausgewählten Objekt. Daher sprechen wir hier von einer Klassenoperation. Da sich diese Operation auf ein Klassenattribut von Student bezieht, wird sie bei der Klasse Student als Klassenoperation eingetragen. Bezieht sich die Operation allerdings auf ein einzelnes Objekt, und werden im Rahmen der Operation zusätzlich Klassenattribute manipuliert, so handelt es sich nicht um eine Klassenoperation. Beispielsweise inkrementiert die Konstruktoroperation `immatrikulieren()` die Anzahl der Studenten.

Abb. 2.4-3:  
Klassenoperation

Aushilfe
Name
Adresse
Stundenzahl
<u>Stundenlohn</u>
<u>erhöhe Stundenlohn ()</u>
...

- 2 Die Operation bezieht sich auf alle oder mehrere Objekte der Klasse. Hier nutzen wir die Eigenschaft einer Klasse aus, ihre Objekte zu kennen (Objektverwaltung). Beispielsweise wählt die Operation `drucke Vordiplomliste()` unter allen Studenten diejenigen aus, die ein Vordiplom besitzen. Wir sprechen von einer Selektion. Da sich diese Klassenoperation auf *alle* Studenten bezieht, wird sie der Klasse Student zugeordnet.

Operationsarten Operationen lassen sich nach ihren Aufgaben klassifizieren (siehe /Khoshafian, Abnous 90/, /Coad, Yourdon 91/, /Booch 94/):

- 1 Operationen mit lesendem Zugriff (*accessor operation*) auf Attribute derselben Klasse.

Beispiel: `drucke Studienbescheinigung()`.

- 2 Operationen mit schreibendem Zugriff (*update operation*) auf Attribute derselben Klasse.

Beispiel: `notiere Note()`

- 3 Operationen zur Durchführung von Berechnungen.

Beispiel: `berechne Durchschnitt()`.

- 4 Operationen zum Erzeugen (*constructor operation*) und Löschen (*destructor operation*) von Objekten.

Beispiel: `immatrikulieren()`, `exmatrikulieren()`.

- 5 Operationen, die Objekte einer Klasse nach bestimmten Kriterien selektieren (*query operation, select operation*). Das ist beispiels-

weise eine Operation, die alle Studenten ermittelt, die in diesem Jahr das Vordiplom bestanden haben. Diese Art von Operationen werden im Analysemodell als Klassenoperationen eingetragen.

Beispiel: `drucke Vordiplomliste()`.

- 6 Operationen zum Herstellen von Verbindungen zwischen Objekten (*connect operation*). Wenn der Student s1 ein Praktikum bei einer Firma absolviert, dann wird von s1 zum Firmenobjekt eine Verbindung aufgebaut (Abb. 2.4-4). Analog gibt es Operationen zum Abbauen der Verbindungen.

Beispiel: `anmeldePraktikum()`.

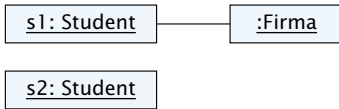


Abb. 2.4-4: Aufbauen und Lesen von Verbindungen zwischen Student und Firma

- 7 Operationen, die Operationen anderer Klassen aktivieren. Damit für den Studenten s1 ein Praktikumsnachweis gedruckt werden kann, muß das Objekt s1 über die Objektverbindung Operationen des Firmenobjekts verwenden, um dessen Attributwerte zu lesen (Abb. 2.4-4).

Beispiel: `druckePraktikumsnachweis()`.

Besitzt eine Klasse viele Operationen, dann können sie mit Hilfe von Stereotypen gruppiert werden (Abb. 2.4-5). Dabei kann die obige Klassifikation sinnvoll eingesetzt werden.

Stereotyp

Aushilfe
«constructor» einstellen()
«update» ändereStundenzahl() erhöheStundenlohn()
«query» druckeAdreßliste() druckeStundenliste()

Abb. 2.4-5: Gruppieren von Operationen mittels Stereotypen

Eine Operation heißt extern, wenn sie direkt von der Benutzungsoberfläche aktiviert wird. Eine externe Operation kann weitere – interne – Operationen aufrufen. Eine interne Operation wird immer von einer anderen Operation innerhalb des Systems aktiviert. Das Ziel der Systemanalyse ist es, alle externen Operationen zu ermitteln. Interne Operationen werden nur dann in das Klassendiagramm eingetragen, wenn es für das Verständnis notwendig ist.

externe und interne Operationen

Der Operationsname soll ausdrücken, was die Operation leistet. Er muß daher im allgemeinen ein Verb enthalten, z.B. `verschiebe()`, `erhöheGehalt()`. Der Name einer Operation muß im Kontext der Klasse eindeutig ein. Außerhalb der Klasse wird die Operation mit `Klasse.Operation()` bezeichnet.

Operationsname

## LE 2 2 Konzepte und Notation für OOA

Beschreibung von Operationen

Jede Operation wird – sofern ihre Funktionsweise nicht bereits aus dem Namen hervorgeht – aus Benutzersicht beschrieben. Bewährt hat sich hier eine umgangssprachliche Formulierung. Die Erfahrung hat gezeigt, daß sich viele Analytiker durch eine formale Spezifikation überfordert fühlen. Im allgemeinen reicht eine umgangssprachliche Beschreibung aus. Zur Beschreibung komplexer Operationen können Diagramme des dynamischen Modells verwendet werden.

Notation Beschreibung (UML-Erweiterung)

Das Ziel ist eine leicht erstellbare und leicht lesbare Beschreibung, wobei »leicht« in diesem Zusammenhang bedeutet, daß der Aufwand für die Beschreibung deutlich geringer sein muß als für die spätere Programmierung.

Funktion: tueEtwas()

Eingabe: Eingabedaten

Ausgabe: Ausgabedaten

Wirkung: Beschreibung der Wirkung aus Benutzersicht wobei der Fokus auf dem Normalverhalten liegt. Sonderfälle sind anschließend zu beschreiben.

Verwaltungsoperationen

**Verwaltungsoperationen** sind grundlegende Operationen, die fast jede Klasse benötigt.

Bei den folgenden Operationen handelt es sich um interne, elementare Basisoperationen, die wir aus Gründen der Lesbarkeit *nicht* in das Klassendiagramm eintragen. Diese Operationen werden vor allem für eine detaillierte Spezifikation von Interaktionsdiagrammen (siehe Kapitel 2.10) benötigt.

- `new()`: Erzeugen eines neuen Objekts.
- `delete()`: Löschen eines Objekts.
- `setAttribute()`: Schreiben eines Attributwertes, z.B. `setGehalt()`.
- `getAttribute()`: Lesen eines Attributwertes, z.B. `getGehalt()`.
- `link()`: Aufbauen einer Verbindung zwischen Objekten.
- `unlink()`: Entfernen einer Verbindung zwischen Objekten.
- `getlink()`: Lesen einer Verbindung zwischen Objekten.

Außer dieser Basisfunktionalität verwenden wir die folgenden externen Verwaltungsoperationen. Diese Operationen werden zum Modellieren in den Interaktions- und Zustandsdiagrammen (siehe Kapitel 2.10 und 2.11) benötigt. In die Klassendiagramme werden sie bei »echten« Projekten aus Gründen der Übersichtlichkeit meistens nicht eintragen. Bei vielen Beispielen in diesem Buch verwende ich sie der besseren Verständlichkeit halber auch im Klassendiagramm.

- `erfassen()`: Erfassen eines neuen Objekts, wobei im Unterschied zur Basisoperation `new()` weitere Aufgaben, z.B. das Senden von Botschaften an andere Objekte, damit verbunden sein können.
- `ändern()`: Ändern eines vorhandenen Objekts.
- `löschen()`: Löschen eines Objekts.
- `erstelleListe()`: Alle Objekte der Klasse anzeigen.

Kapitel 2.10 und 2.11





**Abgeleitetes Attribut (*derived attribute*)** Abgeleitete Attribute lassen sich aus anderen Attributen berechnen. Sie dürfen nicht direkt geändert werden.

**Abstrakter Datentyp (*abstract data type*)** Der abstrakte Datentyp (ADT) ist ursprünglich ein Konzept des Entwurfs. Ein abstrakter Datentyp wird ausschließlich über seine (Zugriffs-) Operationen definiert, die auf Exemplare dieses  $\rightarrow$ Typs angewendet werden. Die Repräsentation der Daten und die Wahl der Algorithmen zur Realisierung der  $\rightarrow$ Operationen sind nach außen nicht sichtbar, d.h. der ADT realisiert das Geheimnisprinzip. Von einem abstrakten Datentyp können beliebig viele Exemplare erzeugt werden. Die  $\rightarrow$ Klasse stellt eine Form des abstrakten Datentyps dar.

**Attribut (*attribute*)** Attribute beschreiben Daten, die von den  $\rightarrow$ Objekten der  $\rightarrow$ Klasse angenommen werden können. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch i.allg. unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten  $\rightarrow$ Typ und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muß jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig. Abgeleitete Attribute lassen sich aus anderen Attributen berechnen.

**Attributspezifikation (*attribute specification*)** Ein  $\rightarrow$ Attribut wird durch folgende Angaben spezifiziert:

Name: Typ = Anfangswert  
{mandatory, key, frozen, Einheit: ..., Beschreibung: ...}

wobei gilt: mandatory = Muß-Attribut, key = Schlüsselattribut, frozen = Attributwert nicht änderbar.

**Elementare Klasse (*support class*)** Wird der Typ eines  $\rightarrow$ Attributs wieder durch eine  $\rightarrow$ Klasse realisiert, dann spricht man von einer elementaren Klasse. Sie wird nicht in das  $\rightarrow$ Klassendiagramm eingetragen.

**Geheimnisprinzip (*information hiding*)** Die Einhaltung des Geheimnisprinzips bedeutet, daß die Attribute und die Realisierung der Operationen außerhalb der Klasse nicht sichtbar sind.

**Klasse (*class*)** Eine Klasse definiert für eine Kollektion von  $\rightarrow$ Objekten deren Struktur (Attribute),  $\rightarrow$ Verhalten (Operationen) und Beziehungen (Assoziationen, Vererbungsstrukturen). Klassen besitzen – mit Ausnahme von abstrakten Klassen – einen Mechanismus, um neue Objekte zu erzeugen. Der Klassenname muß mindestens im Paket, besser im gesamten System eindeutig sein.

**Klassenattribut (*class scope attribute*)** Ein Klassenattribut liegt vor, wenn nur ein Attributwert für alle  $\rightarrow$ Objekte der  $\rightarrow$ Klasse existiert. Klassenattribute sind von der Existenz der Objekte unabhängig.

**Klassenoperation (*class scope operation*)** Eine Klassenoperation ist eine Operation, die für eine  $\rightarrow$ Klasse statt für ein  $\rightarrow$ Objekt der Klasse ausgeführt wird.

**Komplexes Objekt (*composite object, complex object*)** Besitzt ein  $\rightarrow$ Objekt  $\rightarrow$ Attribute, die selbst wieder Objekte sind, so wird es als komplexes Objekt bezeichnet. Ein (Unter-) Objekt kann ebenfalls komplex sein.

**Objekt (*object*)** Ein Objekt besitzt einen  $\rightarrow$ Zustand (Attributwerte und Verbindungen zu anderen Objekten), reagiert mit einem definierten  $\rightarrow$ Verhalten (Operationen) auf seine Umgebung und besitzt eine  $\rightarrow$ Objektidentität, die es von allen anderen Objekten unterscheidet. Jedes Objekt ist Exemplar einer  $\rightarrow$ Klasse.

**Objektdiagramm (*object diagram*)** Das Objektdiagramm stellt  $\rightarrow$ Objekte und ihre Verbindungen untereinander dar. Objektdiagramme werden im allgemeinen verwendet, um einen Ausschnitt des Systems zu einem bestimmten Zeitpunkt zu modellieren. Objekte können einen – im jeweiligen Objektdiagramm – eindeutigen Namen besitzen oder es können anonyme Objekte sein. In verschiedenen Objektdiagrammen kann der gleiche Name unterschiedliche Objekte kennzeichnen.

**Objektidentität (*object identity*)** Jedes  $\rightarrow$ Objekt besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Selbst wenn zwei Objekte zufällig dieselben Attributwerte besitzen,

## LE 2 Glossar/Zusammenhänge

haben sie eine unterschiedliche Identität. Im Speicher wird die Identität durch unterschiedliche Adressen realisiert.

**Objektverwaltung (*class extension, object warehouse*)** In der Systemanalyse besitzen Klassen implizit die Eigenschaft der Objektverwaltung. Das bedeutet, daß die Klasse weiß, welche →Objekte von ihr erzeugt wurden. Damit erhält die Klasse die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte einer →Klasse durchzuführen.

**Operation (*operation*)** Eine Operation ist eine Funktion, die auf die internen Daten (Attributwerte) eines →Objekts Zugriff hat. Auf alle Objekte einer →Klasse sind dieselben Operationen anwendbar. Für Operationen gibt es in der Analyse im allgemeinen eine fachliche Beschreibung. Abstrakte Operationen besitzen nur einen Operationskopf. Externe Operationen werden vom späteren Bediener des Systems aktiviert. Interne Operationen werden dagegen immer von anderen Operationen aufgerufen.

**Typ (*type*)** Jedes →Attribut ist von einem bestimmten Typ. Er kann ein Standardtyp (z.B. Int), ein Aufzählungstyp, eine →elementare Klasse oder eine Liste (*list of <Typ>*) sein.

Der Typ wird auch im Sinne von Klassen-Spezifikation verwendet. Er legt fest, auf welche Operationsaufrufe die →Objekte einer →Klasse reagieren können, d.h. der Typ definiert die Schnittstelle der Objekte. Ein Typ wird implementiert durch ein oder mehrere →Klassen.

**Verhalten (*behavior*)** Unter dem Verhalten eines →Objekts sind die beobachtbaren Effekte aller →Operationen zu verstehen, die auf das Objekt angewendet werden können. Das Verhalten einer →Klasse wird bestimmt durch die Operationsaufrufe, auf die diese Klasse bzw. deren Objekte reagieren.

**Zustand (*state*)** Der Zustand eines →Objekts wird bestimmt durch seine Attributwerte und seine Verbindungen (*links*) zu anderen Objekten, die zu einem bestimmten Zeitpunkt existieren.

Die objektorientierte Softwareentwicklung basiert auf folgenden Konzepten: Ein **Objekt** besitzt einen Zustand, reagiert auf ein definiertes Verhalten und hat eine Identität. Objekte und ihre Verbindungen werden im **Objektdiagramm** dargestellt. Eine **Klasse** beschreibt eine Kollektion von Objekten mit gleicher Struktur, gleichem Verhalten und gleichen Beziehungen. Sie wird im **Klassendiagramm** dargestellt. Die **Attribute** beschreiben die Daten, die von den Objekten einer Klasse angenommen werden können. Jedes Attribut ist von einem bestimmten Typ. Wir unterscheiden (Objekt-) **Attribute** und **Klassenattribute**. Die Operationen beschreiben das Verhalten bzw. die Schnittstelle der Klasse. Wir unterscheiden (Objekt-) Operationen, Konstruktoroperationen und **Klassenoperationen**.



**1** *Lernziel: Wichtige Begriffe erläutern können.*

Aufgabe  
5–10 Minuten

- a** Erläutern Sie den Begriff »Objektidentität«.
- b** Was ist der Unterschied zwischen einer Objektidentität und einem Objektnamen?
- c** Was ist der Unterschied zwischen einer Klasse und einer Menge von Objekten dieser Klasse?
- d** Was ist ein Klassenattribut?
- e** Was ist ein abgeleitetes Attribut?
- f** Wofür verwenden Sie eine Klassenoperation?
- g** Was sind Verwaltungsoperationen?

**2** *Lernziel: Objektdiagramm erstellen können.*

Aufgabe  
5–10 Minuten

Identifizieren Sie anhand der folgenden Beschreibung Objekte und deren Verbindungen und stellen sie als Objektdiagramm dar. In einer Bibliothek sind die Regale voller Bücher. Da stehen beispielsweise

- Ken Follet, Die Säulen der Erde, 1990,
- Noah Gordon, Der Medicus, 1987 und
- Nicholas Evans, Der Pferdeflüsterer, 1995

Für jeden Leser werden Name, Adresse und Geburtsdatum gespeichert. Außerdem erhält jeder Leser eine Nummer. Hans Müller, geb. am 1.3.1975 und wohnhaft in Bochum leiht sich »Die Säulen der Erde« aus. Spätestens am 12.5.1998 muß er es zurückgeben. Dieses Rückgabedatum wird ins Buch eingetragen. Else Wallersee aus Dortmund, geb. am 26.3.1975 leiht sich »Der Medicus« und »Der Pferdeflüsterer« aus. Beide Bücher muß sie am 14.5.1998 zurückgeben.

**3** *Lernziel: Klassendiagramm, Klassenbeschreibungen und Attributspezifikationen erstellen können.*

Aufgabe  
5–10 Minuten

Identifizieren Sie anhand folgender Beschreibung Klassen und Attribute und stellen sie als Diagramm dar. Für jede Klasse ist eine Klassenbeschreibung zu erstellen. Alle Attribute sind vollständig zu spezifizieren.

An einer Hochschule sind studentische Hilfskräfte und Angestellte zu verwalten. Für alle Personen sind der Name, bestehend aus Vor- und Nachname, und die Adresse, bestehend aus PLZ, Ort und Straße, zu speichern. Für studentische Hilfskräfte sind außer der Matrikelnummer auch Beginn und Ende aller Arbeitsverträge sowie die jeweilige wöchentliche Stundenzahl einzutragen. Alle studentischen Hilfskräfte erhalten den gleichen Stundenlohn. Für jeden Angestellten wird das Eintrittsdatum gespeichert.

## LE 2 Aufgaben

Aufgabe 4 *Lernziel: Die Konzepte Klassenattribut und Objektattribut unterscheiden können.*  
5 Minuten

Modellieren Sie die Klasse Videofilm mit allen Attributen. In einem Videoverleih werden für Videofilme folgende Informationen festgehalten: Titel des Films, Laufzeit und Erscheinungsjahr. Jeder Videofilm besitzt eine individuelle Ausleihgebühr. Wird ein Film beschädigt zurückgegeben, so ist eine fixe Entschädigungsgebühr zu entrichten (für alle Filme gleich). Außerdem soll die Anzahl aller Videofilme der Videothek festgehalten werden.

Aufgabe 5 *Lernziel: Klassendiagramm erstellen und Attribute spezifizieren können.*  
5–10 Minuten

Identifizieren Sie anhand folgender Beschreibung Klassen mit Attributen/Operationen und stellen sie grafisch dar. Spezifizieren Sie jedes Attribut.

Eine Artikelverwaltung ist zu modellieren. Jeder Artikel besitzt eine eindeutige Nummer, eine Bezeichnung, einen Einkaufs- und einen Verkaufspreis. Neue Artikel müssen erfaßt und bei vorhandenen Artikeln die Preise geändert werden. Artikelzu- und abgänge müssen gebucht werden können. Ist der Mindestbestand von Artikeln unterschritten, so muß für alle betreffenden Artikel ein Bestellvorschlag gedruckt werden, der jeden Artikel bis zum Maximalbestand auffüllt. Außerdem soll eine Liste aller Artikel erstellt werden.

## 2 Konzepte und Notation der objektorientierten Analyse (Statische Konzepte)



- Erklären können, was eine Assoziation ist.
- Erklären können, was eine assoziative Klasse und eine qualifizierte Assoziation ist.
- Erklären können, was Aggregation und Komposition bedeuten.
- Erklären können, was Vererbung ist.
- Erklären können, was ein Paket ist.
- UML-Notation für Assoziation, Vererbung und Paket anwenden können.
- Assoziationen in einem Text identifizieren und darstellen können.
- Vererbungsstrukturen in einem Text identifizieren und darstellen können.
- Klassen zu Paketen gruppieren können.

verstehen

anwenden



Die Kapitel 2.1 bis 2.4 müssen bekannt sein.



- 2.5 [Assoziation](#) 40
- 2.6 [Vererbung](#) 51
- 2.7 [Paket](#) 55



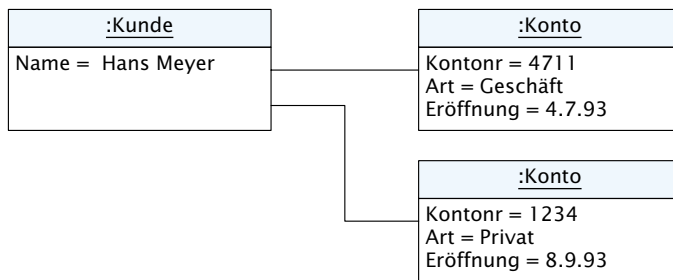
## 2.5 Assoziation

**Definition** Eine **Assoziation** modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Eine Assoziation modelliert stets Beziehungen zwischen Objekten, nicht zwischen Klassen. Es ist jedoch üblich von einer Assoziation zwischen Klassen zu sprechen, obwohl streng genommen die Objekte dieser Klassen gemeint sind. Eine **reflexive** Assoziation besteht zwischen Objekten derselben Klasse.

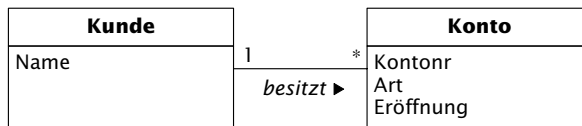
**Beispiel** Wir betrachten eine Bank. Hans Meyer eröffnet am 4.7.1993 ein Geschäftskonto mit der Kontonummer 4711. Er wird dadurch zum Kunden der Bank. Zwei Monate später eröffnet er bei der gleichen Bank noch ein privates Konto, das die Kontonummer 1234 erhält. Jedes Konto lautet nur auf den Namen Hans Meyer. Bei unserem Beispiel in Abb. 2.5-1 existiert eine Verbindung zwischen Hans Meyer und den Konten 4711 und 1234.

Abb. 2.5-1:  
Assoziation  
zwischen Kunde  
und Konto

Objektdiagramm



Klassendiagramm



Für die Objekte der Klassen **Kunde** und **Konto** gilt in dem betrachteten Modell:

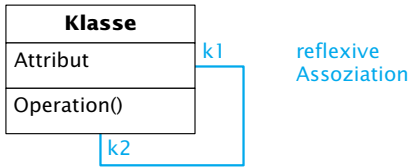
- Jeder Kunde kann mehrere Konten besitzen.
- Jedes Konto gehört zu genau einem Kunden.

Die Menge aller Verbindungen bezeichnen wir als Assoziation zwischen den Objekten der Klassen **Kunde** und **Konto**. Assoziationen sind in der Systemanalyse inhärent **bidirektional**, d.h. der Kunde kennt seine Konten und jedes Konto kennt seinen Kunden.

**Notation** Die UML kennt binäre und höherwertige Assoziationen. Wir betrachten zunächst nur die **binäre** Assoziation, d.h. die Assoziation zwischen zwei Objekten. Sie wird durch eine Linie zwischen einer



Abb. 2.5-2:  
Notation für  
Assoziationen



oder zwei Klassen beschrieben (Abb. 2.5-2). An jedem Ende der Linie muß die Wertigkeit bzw. Kardinalität (*multiplicity*) angegeben sein.

Wie das obige Beispiel zeigt, kann sich ein Objekt (der Kunde) auf mehrere andere Objekte (die Konten) beziehen, während umgekehrt jedes Konto zu genau einem Kunden gehört. Dieser Sachverhalt wird durch die **Kardinalitäten** der Assoziation beschrieben. Während die Assoziationslinie zunächst nur aussagt, daß sich Objekte der beteiligten Klassen kennen, spezifiziert die Kardinalität *wie viele* Objekte ein bestimmtes Objekt kennen kann. Abb. 2.5-3 zeigt mögliche Kardinalitäten der UML.

Kardinalitäten

1	<input type="text"/>	genau 1
0..1	<input type="text"/>	0 bis 1
*	<input type="text"/>	0 bis viele
3..*	<input type="text"/>	3 bis viele
0..2	<input type="text"/>	0 bis 2
2	<input type="text"/>	genau 2
2, 4, 6	<input type="text"/>	2, 4 oder 6
1..5, 8, 10..*	<input type="text"/>	nicht 6, 7 oder 9

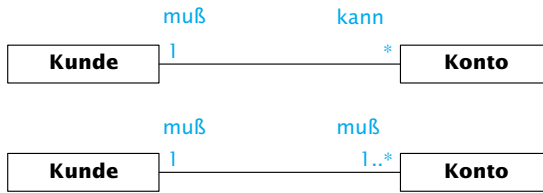
Abb. 2.5-3:  
Notation für  
Kardinalität

Wir unterscheiden Kann- und Muß-Assoziationen. Eine Kann-Assoziation hat als Untergrenze die Kardinalität 0, eine Muß-Assoziation die Kardinalität 1 oder größer. Die Kardinalitäten der Abb. 2.5-4 sind wie folgt zu interpretieren: Die Kann-Assoziation von Kunde zu Konto (\*) bedeutet, daß es (Bank-) Kunden geben kann, die kein Konto besitzen. Die Muß-Assoziation von Konto zu Kunde (1) bedeutet, daß ein Konto nicht auf mehrere Namen laufen kann. Ein neues Konto darf nur für einen existierenden Kunden eingerichtet

Muß- und Kann-  
Assoziation

### LE 3 2 Konzepte und Notation für OOA

Abb. 2.5-4:  
Kann- und Muß-  
Assoziationen



werden. Wird dagegen auch die Assoziation von Kunde zu Konto als Muß-Assoziation (1..\*) modelliert, so darf es keine Kunden geben, die kein Konto besitzen. Wird das letzte Konto eines Kunden gelöscht, so muß auch der entsprechende Kunde gelöscht werden. Wird umgekehrt ein Kunde im System gelöscht, so werden auch alle seine Konten gelöscht, sofern sie nicht einem anderen Kunden zugeordnet werden.

Assoziationsname

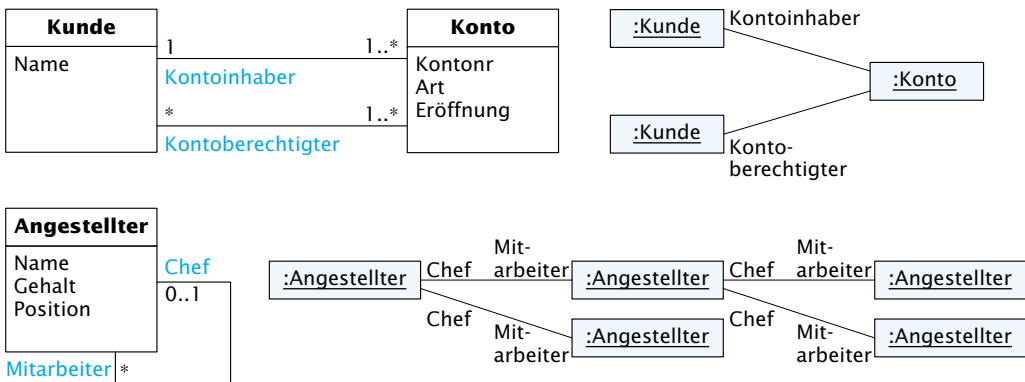
Assoziationen können benannt werden. Der Name beschreibt im allgemeinen nur eine Richtung der Assoziation, wobei ein schwarzes Dreieck die Leserichtung angibt (Abb. 2.5-1). Der Name kann fehlen, wenn die Bedeutung der Assoziation offensichtlich ist.

Rolle

Während der Assoziationsname die Semantik der Assoziation beschreibt, enthält der Rollename oder kurz die **Rolle** Informationen über die Bedeutung einer Klasse – bzw. ihrer Objekte – in der Assoziation. Eine binäre Assoziation besitzt maximal zwei Rollen. Der Rollename wird jeweils an ein Ende der Assoziation geschrieben, und zwar bei der Klasse, deren Bedeutung in der Assoziation sie näher beschreibt. Die geschickte Wahl der Rollennamen kann zur Verständlichkeit des Modells mehr beitragen als der Name der Assoziation.

**Beispiel** In der Abb. 2.5-5 beschreiben die Rollen, daß der Kunde in Bezug auf das Konto sowohl als Kontoinhaber als auch als Kontoberechtigter auftreten kann. Bei der reflexiven Assoziation kann ein Angestellter Chef anderer Angestellter sein. Umgekehrt ist ein Angestellter Mitarbeiter eines anderen Angestellten.

Abb. 2.5-5:  
Rollennamen



Rollennamen oder Assoziationsnamen *müssen* angegeben werden, wenn zwischen zwei Klassen mehr als eine Assoziation besteht. Auch bei reflexiven Assoziationen müssen die Rollen stets angegeben werden, um die Verständlichkeit zu gewährleisten. In allen anderen Fällen sind Rollennamen optional.

**Vergleich mit den relationships des Entity-Relationship-Modells**

Assoziationen sind vergleichbar mit den *relationships* des *Entity-Relationship-Modells*. Sie stellen jedoch nicht nur eine statische Struktur zwischen den Klassen dar, sondern bilden vor allem die Voraussetzung für die Kommunikation zwischen Objekten. Im Gegensatz zur relationalen Datenbank oder zum *information model* /Shlaer, Mellor 88/ ist das Wissen, welche Objekte miteinander in Verbindung stehen, ausschließlich in der Assoziation vorhanden. Fremdschlüssel oder Referenzattribute sind beim objektorientierten Modell nicht anzugeben.

Assoziation vs. relationship

Die Ähnlichkeit zwischen einem *Entity-Relationship-Diagramm* (ERD) und einem Klassendiagramm ist sehr groß. Dennoch ist das ERD nicht einfach eine Projektion des Klassendiagramms.

Klassendiagramm vs. Entity-Relationship-Diagramm

Das Klassendiagramm unterscheidet sich beispielsweise vom ERD in folgenden Punkten:

- Es ist keine Normalisierung der Attribute notwendig.
- Künstliche Schlüsselattribute sind nicht notwendig.
- Fremdschlüssel sind nicht notwendig.

Wenn die Kardinalität größer als eins ist, kann die Menge der Objektverbindungen (*links*) geordnet oder ungeordnet sein. Eine vorliegende Ordnung wird durch das Schlüsselwort {ordered} gekennzeichnet, das an ein Ende der Assoziation angetragen wird. Diese Angabe sagt jedoch nichts darüber aus, wie die Ordnung definiert ist (z.B. zeitlich, alphabetisch) oder wie die Ordnung erreicht wird. In der Abb. 2.5-6 drückt {ordered} die Startreihenfolge der Teilnehmer in einem Wettbewerb aus.

geordnete Assoziation

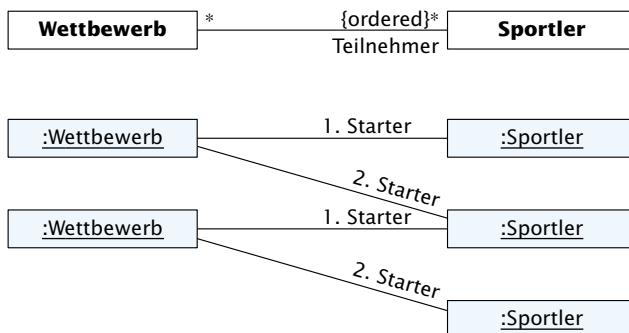


Abb. 2.5-6: Geordnete Assoziationen

### LE 3 2 Konzepte und Notation für OOA

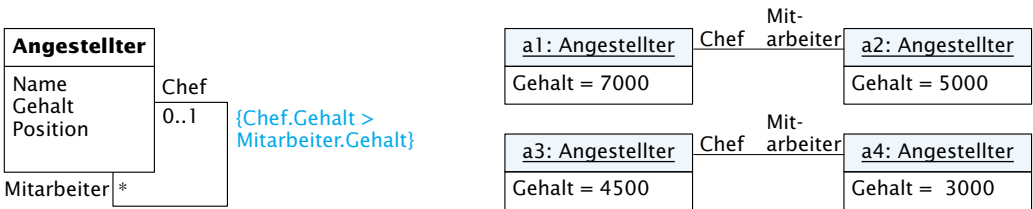
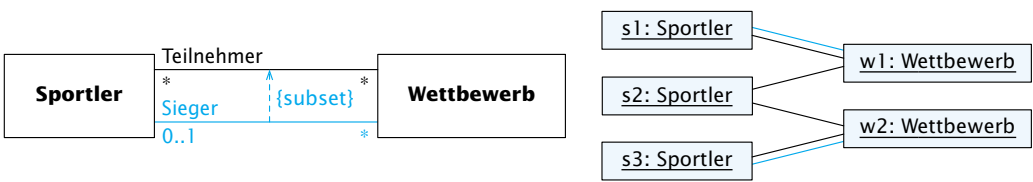
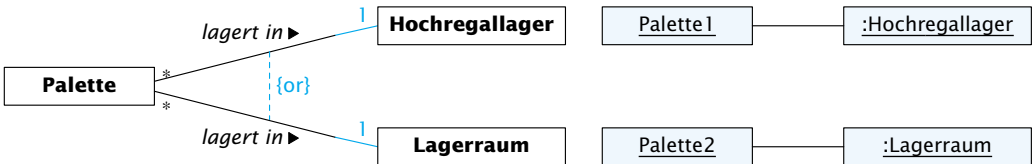
Restriktionen (constraints) Assoziationen können um **Restriktionen** (*constraints*) ergänzt werden. Restriktionen können frei formuliert werden. Für häufig wiederkehrende Fälle ist es sinnvoll, sich Standards zu schaffen.

**Beispiel** Für die Assoziationen zwischen Kunde und Konto der Abb. 2.5-1 fordert die folgende Restriktion, daß ein Kunde für ein bestimmtes Konto nicht gleichzeitig Kontoinhaber und Kontoberechtigter ist.  
{Kunde. Kontoinhaber <> Kunde. Kontoberechtigter}

**or-Restriktion** Die *or*-Restriktion in Abb. 2.5-7 sagt aus, daß eine Palette zu einem Zeitpunkt entweder mit einem Objekt von Hochregallager oder von Lagerraum in Verbindung steht. Allgemein ausgedrückt: Zu jedem beliebigem Zeitpunkt kann nur eine der Assoziationen, die von »Palette« ausgehen, gelten. Beachten Sie, daß bei dieser Modellbildung (Kardinalität = 1) für jede Palette eine Verbindung zu einem Lager aufgebaut werden muß. Soll diese sofortige Zuordnung nicht erfolgen, dann ist die Kardinalität 0..1 zu wählen.

Eine *or*-Restriktion kann sich auch auf mehr als zwei Assoziationen beziehen.

**subset-Restriktion** Die *subset*-Restriktion in Abb. 2.5-7 bedeutet, daß die Sieger eine Teilmenge der Teilnehmer bilden. Das Objektdiagramm zeigt, daß die Sportler s1 und s2 am gleichen Wettbewerb w1 teilgenommen haben und s1 Sieger wurde. Eine blaue Verbindung kann nur dann zwischen



schen zwei Objekten aufgebaut werden, wenn es auch eine schwarze Objektverbindung gibt.

Eine Restriktion kann sich auch nur auf eine einzige Assoziation beziehen. In der Abb. 2.5-7 bezieht sie sich auf eine Verbindung zwischen zwei Objekten derselben Klasse. Wie das Objektdiagramm zeigt, ist das Gehalt des Mitarbeiters a2 geringer als dasjenige des eigenen Chefs, kann aber durchaus höher sein als das Gehalt des Angestellten a3, der ebenfalls Chef ist.

Eine Assoziation kann zusätzlich die Eigenschaften einer Klasse besitzen, d.h. sie hat Attribute und Operationen sowie Assoziationen zu anderen Klassen. Zur Darstellung wird ein Klassensymbol verwendet, das über eine gestrichelte Linie mit der Assoziation verbunden wird (Abb. 2.5-8). Wir sprechen von einer **assoziativen Klasse** (*association class*).

assoziative Klasse

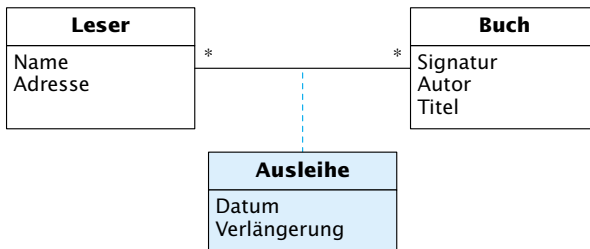


Abb. 2.5-8:  
Assoziative Klasse

Die **Qualifikationsangabe** (*qualifier*) ist ein spezielles Attribut der Assoziation, dessen Wert ein oder mehrere Objekte auf der anderen Seite der Assoziation selektiert. Mit anderen Worten: Die Qualifikationsangabe zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Der *qualifier* kann auch aus mehreren Attributen bestehen. Bei Verwendung einer Qualifikationsangabe besitzt die Kardinalität auf der gegenüberliegenden Seite der Assoziation folgende Interpretation. »0..1« bedeutet, daß ein einziges Objekt selektiert wird, aber es gibt nicht zu jedem möglichen *qualifier*-Wert ein Objekt. »1« bedeutet, daß jeder mögliche Wert genau ein Objekt selektiert, d.h. die Menge der *qualifier*-Werte muß endlich sein. Die Kardinalität »\*« sagt aus, daß die Qualifikationsangabe eine Menge von Objekten selektiert.

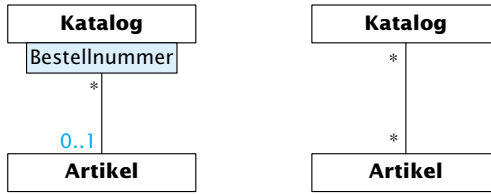
qualifizierte  
Assoziation

Ein Katalog enthält viele Artikel. Innerhalb eines Katalogs bezeichnet jede Bestellnummer genau einen Artikel. Mit anderen Worten: Ein Katalog-Objekt zusammen mit der Bestellnummer selektiert genau einen Artikel. Auf der linken Seite der Abb. 2.5-9 wird diese Problemstellung mittels Qualifikationsangabe modelliert. Im Vergleich zur »normalen« Modellierung auf der rechten Seite ändert sich durch die Qualifikationsangabe die *many*-Kardinalität auf der Seite des Artikels in 0..1. Das bedeutet, daß es gültige Bestellnummern gibt, zu denen kein Artikel-Objekt existiert. Wäre die

Beispiel

### LE 3 2 Konzepte und Notation für OOA

Abb. 2.5-9:  
Assoziation mit  
und ohne  
Qualifikations-  
angabe

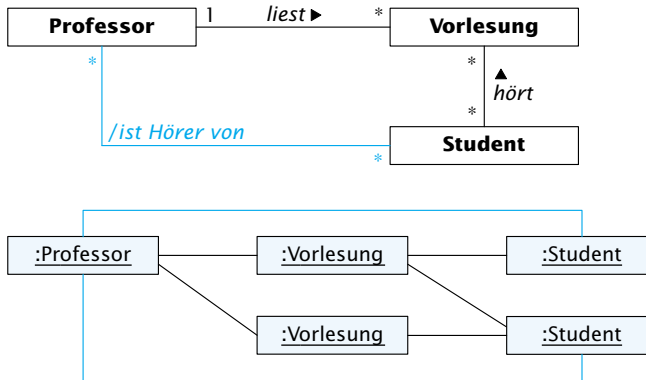


Kardinalität gleich 1, dann müsste zu jeder möglichen Bestellnummer auch ein Artikel-Objekt existieren. Wie dieses Beispiel zeigt, erhöhen Qualifikationsangaben den Informationsgehalt des Klassendiagramms: Dem linken Teil der Abb. 2.5-9 entnimmt der Leser, daß ein Katalog-Objekt zusammen mit der Bestellnummer einen Artikel selektiert. Im rechten Teil erfährt er nur, daß ein Katalog viele Artikel enthält.

abgeleitete  
Assoziation

Eine Assoziation heißt abgeleitet (*derived association*), wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sie fügt keine neue Information zum Modell hinzu und ist daher redundant. Eine abgeleitete Assoziation wird durch das Präfix »/« vor dem Assoziationsnamen oder einem Rollennamen gekennzeichnet. Wie das Objektdiagramm der Abb. 2.5-10 zeigt, gibt es einen »direkten Weg« von Professor zu Student und einen »Umweg« über die Vorlesung.

Abb. 2.5-10:  
Abgeleitete  
Assoziation



Die UML kennt außer der einfachen Assoziation (*ordinary association*) noch zwei weitere Arten:

- Aggregation und
- Komposition.

Aggregation

Eine **Aggregation** (*aggregation*) liegt vor, wenn zwischen den Objekten der beteiligten Klassen (kurz: den beteiligten Klassen) eine Rangordnung gilt, die sich durch »ist Teil von« bzw. »besteht aus« beschreiben läßt. Wir sprechen auch vom Ganzen und seinen Teilen. Die Objekte der Aggregation bilden einen gerichteten azykli-

schen Graphen. Das bedeutet: Wenn B Teil von A ist, dann darf A nicht Teil von B sein. *Shared aggregation (weak ownership)* bedeutet, daß ein Teilobjekt mehreren Aggregatobjekten zugeordnet werden kann. Das entsprechende Objektdiagramm bildet eine Netzstruktur.

Eine **Komposition** (*composition, composite aggregation*) ist eine starke Form der Aggregation. Auch hier muß eine *whole-part*-Beziehung vorliegen und die Objekte formen einen gerichteten azyklischen Graphen. Darüber hinaus gelten:

Komposition

- Jedes Objekt der Teilklasse kann – zu einem Zeitpunkt – nur Komponente eines einzigen Objekts der Aggregatklasse sein, d.h. die bei der Aggregatklasse angetragene Kardinalität darf nicht größer als eins sein (*unshared aggregation, strong ownership*). Ein Teil darf jedoch auch einem anderen Ganzen zugeordnet werden.
- Die dynamische Semantik des Ganzen gilt auch für seine *Teile (propagation semantics)*. Wird beispielsweise das Ganze kopiert, so werden auch seine Teile kopiert.
- Wird das Ganze gelöscht, dann werden automatisch seine Teile gelöscht (*they live and die with it*). Ein Teil darf jedoch zuvor explizit entfernt werden.

In beiden Fällen kennzeichnet eine Raute das Ganze. Bei einer Aggregation ist es eine weiße bzw. transparente, bei der Komposition eine schwarze bzw. gefüllte Raute. Alle anderen Angaben (Kardinalitäten, Namen, Rollen, Restriktionen, etc.) werden analog zur Assoziation angegeben.

In der Abb. 2.5-11 kann ein Hypertext-Buch aus mehreren Kapiteln bestehen. Jedes Kapitel kann in mehreren Hypertext-Büchern referenziert werden. Es liegt daher eine *shared aggregation* vor. Die rechte Seite der Abbildung modelliert ein Verzeichnis, das mehrere Dateien enthält, wobei jede Datei nur in einem Verzeichnis enthal-

Beispiel

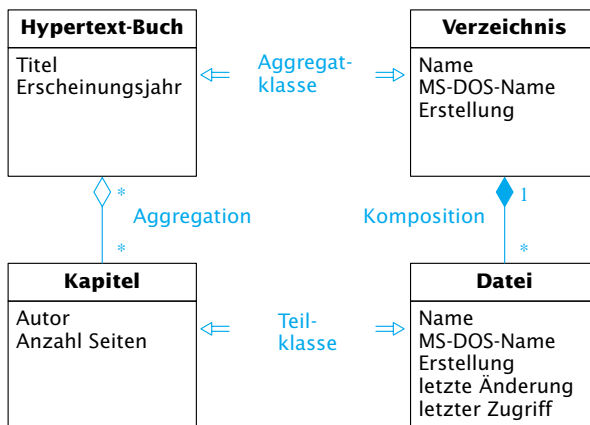


Abb. 2.5-11: Aggregation vs. Komposition



ten sein kann. Wird das Verzeichnis kopiert, dann werden auch alle darin enthaltenen Dateien kopiert. In diesem Fall liegt eine Komposition vor.

**Was ist eine Aggregation bzw. Komposition?**



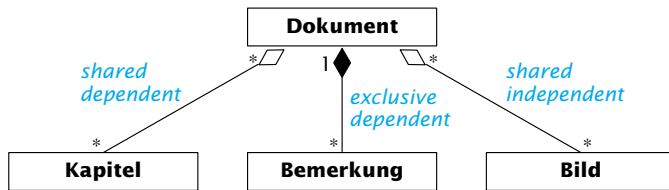
Obwohl die Definitionen von Aggregation und Komposition zunächst plausibel klingen, ist die Abgrenzung zwischen der »einfachen« Assoziation, der Aggregation und der Komposition in der Praxis oft problematisch. Auch die Methoden-Experten sind sich in diesem Punkt nicht immer einig. Ich möchte hier einen kleinen Einblick in die Sichtweise anderer Methoden geben. Die meisten Notationen unterscheiden nur zwischen »einfacher« Assoziation und Aggregation. Einige Autoren, z.B. /Fowler 97/ verwenden nur die »einfache« Assoziation, um diese Abgrenzungsproblematik zu vermeiden.

mögliche Definitionen

/Bertino, Matino 93/ diskutieren verschiedene Definitionsmöglichkeiten für die Aggregation. Die Abb. 2.5-12 zeigt, wie diese verschiedenen Definitionen auf die UML abgebildet werden.

- 1** Ein Teil-Objekt darf nur zu einem Aggregat-Objekt gehören (*exclusive*).  
Diese Einschränkung ist zu eng, da sie die *shared aggregation* ausschließt.
- 2** Das Teil-Objekt darf nicht vor dem Aggregat-Objekt erzeugt werden.  
Diese Einschränkung führt zu Problemen, wenn ein Teil-Objekt bereits existiert und einem Ganzen zugeordnet werden soll.
- 3** Wenn das Aggregat-Objekt gelöscht wird, dann müssen alle Teil-Objekte ebenfalls gelöscht werden.  
Diese Einschränkung ermöglicht es *nicht*, Teil-Objekte für ein neues Aggregat-Objekt zu verwenden. Es ist daher sinnvoll, zwischen abhängigen (*dependent*) und nicht-abhängigen (*independent*) Teil-Objekten zu unterscheiden. Wird das Aggregat-Objekt gelöscht, dann werden nur die abhängigen Teil-Objekte gelöscht.

Abb. 2.5-12:  
Verschiedene Möglichkeiten zur Definition einer Aggregation/ Komposition



Coad, Yourdon

/Coad, Yourdon 91/ unterscheiden die folgenden Aggregationsstrukturen (*whole part*):

- a** Das Ganze und seine Teile, z.B. der PKW (Ganzes) und sein Motor (Teil),
- b** der Behälter und sein Inhalt, z.B. das Flugzeug (Behälter) und sein Pilot (Inhalt),

**c** die Kollektion und ihre Mitglieder, z.B. Firma (Kollektion) und Angestellte (Mitglieder).

/Odell 94/ unterscheidet sechs verschiedene Arten der Aggregation (composition): Odell

**a** Konfiguration von Teilen in einem Ganzen (*component-integral object composition*). Sie definiert, aus welchen Teilen ein Objekt besteht und ist die häufigste Art der Aggregation. Teil-Objekte dürfen entfernt werden.

Beispiele: Szenen sind Teile eines Films. Räder sind Teile eines Autos.

**b** Invariante Konfiguration von Teilen in einem Ganzen (*material-object composition*). Diese Aggregation definiert, »aus was ein Objekt gemacht ist«. Hier dürfen Teil-Objekte nicht entfernt werden.

Beispiele: Ein Baum besteht teilweise aus Holz. Ein Auto besteht teilweise aus Blech.

**c** Bei der Gleichartigkeit von Teilen und Ganzem (*portion-object composition*) sind die Teile im Prinzip dasselbe wie das Ganze.

Beispiele: Ein Meter ist ein Teil eines Kilometers. Eine Brotscheibe ist Teil eines Brotlaibs.

**d** Invariante und gleichartige Konfiguration von Teilen in einem Ganzen (*place-area composition*). Die Teile können nicht von dem Ganzen getrennt werden.

Beispiele: München ist ein Teil von Bayern. Ein Gipfel ist Teil eines Berges.

**e** Kollektion von Teilen in einem Ganzen (*member-bunch composition*).

Beispiele: Ein Student ist Teil einer Universität. Ein Schiff ist Teil einer Flotte.

**f** Invariante Kollektion von Teilen in einem Ganzen (*member-partnership composition*). Wird ein Mitglied entfernt, so wird auch das Ganze zerstört.

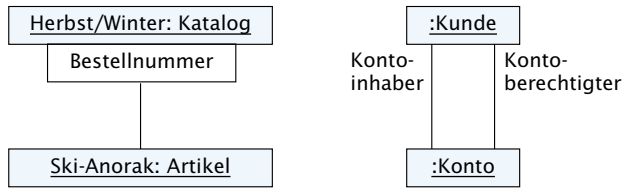
Beispiel: Stan Laurel ist Teil von »Laurel und Hardy«.

Viele Notationselemente der Assoziation können auch bei Objektdiagrammen verwendet werden, um dessen Aussagegehalt zu steigern. Am Ende einer Objektverbindung können Rollennamen, Qualifikationsangabe oder Symbole für die Aggregation bzw. Komposition bei Bedarf eingetragen werden (Abb. 2.5-13). Wird der Assoziationsname an die Objektverbindung angetragen, dann muß er unterstrichen werden.

Objektdiagramm-  
Notation

### LE 3 2 Konzepte und Notation für OOA

Abb. 2.5-13:  
Objektdiagramme  
mit Qualifikations-  
angabe und Rollen

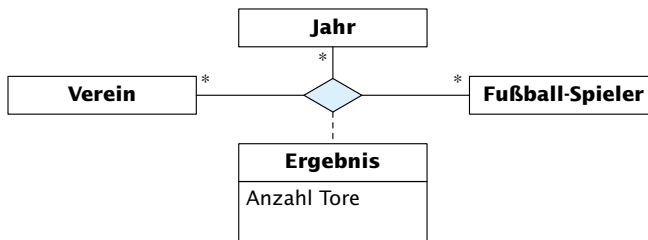


höherwertige  
Assoziationen

Bisher haben wir nur binäre Assoziationen, d.h. Assoziationen zwischen zwei Objekten, betrachtet. Prinzipiell sind auch Assoziationen zwischen drei und mehr Objekten möglich. Wir sprechen von n-ären Assoziationen. Die Abb. 2.5-14 modelliert, daß ein Fußball-Spieler innerhalb eines Jahres in verschiedenen Vereinen aktiv sein kann. Hier ist die ternäre Assoziation zusätzlich mit einer assoziativen Klasse verbunden. Beispielsweise kann für den Fußballer »Müller« festgehalten werden, welches Ergebnis er im Jahr 1998 für den Verein »FC« erzielt hat.

Ternäre und höhere Assoziationen können keine Aggregation oder Komposition bilden.

Abb. 2.5-14:  
ternäre  
Assoziation



### CRC – Class/Responsibility/Collaboration

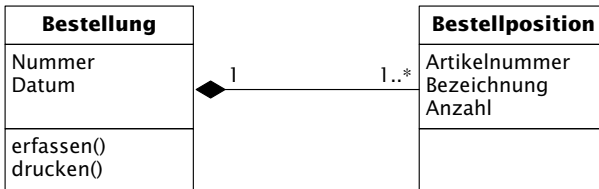
CRC-Karten wurden erstmalig von Beck und Cunningham als Hilfsmittel für die Ausbildung in der objektorientierten Programmierung eingeführt. Sie sind ein wesentlicher Bestandteil der Methode von /Wirfs-Brock 90/. Inzwischen handelt es sich bei den CRC-Karten um eine weit verbreitete Technik, die in zahlreiche objektorientierte Methoden integriert wurde.

Eine **CRC-Karte** ist eine Karteikarte. Oben auf der Karte wird der Name der Klasse (*class*) eingetragen. Die restliche Karte wird in zwei Hälften geteilt. Auf der einen Hälfte werden die Verantwortlichkeiten (*responsibilities*) der Klasse notiert. Darunter sind sowohl das Wissen der Klasse als auch die zur Verfügung gestellten Operationen zu verstehen. Ein Objekt der beschriebenen Klasse kann seine Aufgabe selbst erfüllen oder es kann hierzu die Hilfe anderer Objekte in Anspruch nehmen. Die dafür notwendigen Klassen (*collaborations*) werden auf der anderen Kartenseite eingetragen.

CRC-Karten sind nicht als Alternative, sondern als Ergänzung zum OOA-Modell zu verstehen. Wie die Abb. 2.5-15 zeigt, werden die

Abb. 2.5-15:  
CRC-Karte vs. UML-  
Klassendiagramm

<b>Class</b> Bestellung	
<b>Responsibilities</b>	<b>Collaborations</b>
<ul style="list-style-type: none"> <li>■ verwaltet eine Bestellung</li> <li>■ delegiert Aufgaben an Bestellpositionen</li> </ul>	<ul style="list-style-type: none"> <li>■ Bestellposition</li> </ul>



Informationen auf einer CRC-Karte auf einer höheren Abstraktionsebene dargestellt als im Klassendiagramm. Die ermittelten Klassen bilden immer einen Stapel von Karteikarten und können je nach Verwendungszweck entsprechend angeordnet werden. Zur Modellierung der dynamischen Aspekte werden die Karten so angeordnet, daß sie den Nachrichtenfluß aufzeigen. Bei der Darstellung des statischen Modells werden die Karten entsprechend der Vererbungsstrukturen und Aggregat-Hierarchien angeordnet /Booch 94/.

## 2.6 Vererbung

Die **Vererbung** (*generalization*) beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer spezialisierten Klasse. Die spezialisierte Klasse ist vollständig konsistent mit der Basisklasse, enthält aber zusätzliche Informationen (Attribute, Operationen, Assoziationen). Ein Objekt der spezialisierten Klasse kann überall dort verwendet werden, wo ein Objekt der Basisklasse erlaubt ist. Wir sprechen von einer **Klassenhierarchie** oder einer **Vererbungsstruktur**. Die allgemeine Klasse wird auch als **Oberklasse** (*super class*), die spezialisierte als **Unterklasse** (*sub class*) bezeichnet.

Definition

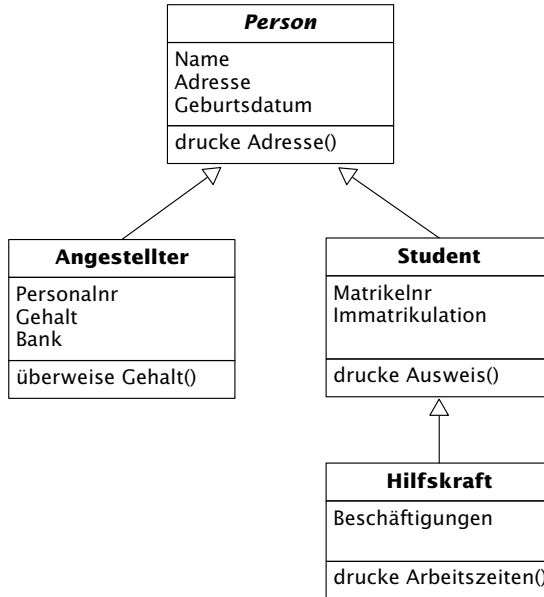
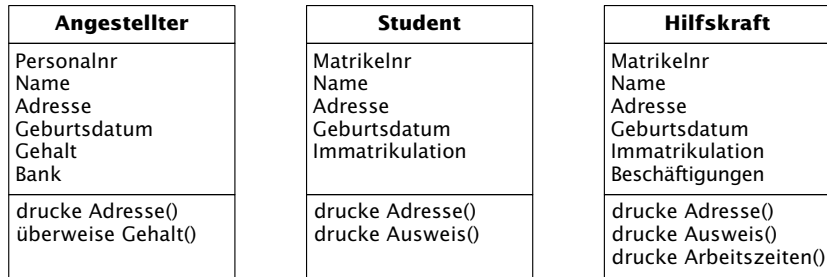
Das Konzept der Vererbung ist nicht nur gedacht, um gemeinsame Eigenschaften und Verhaltensweisen zusammenzufassen, sondern sie muß immer auch eine Generalisierung bzw. Spezialisierung darstellen, d.h. jedes Objekt der Unterklasse »ist ein« Objekt der Oberklasse.

Betrachten wir die Klassen Angestellter, Student und (studentische) Hilfskraft (Abb. 2.6-1). Eine gleichwertige Information erhalten wir durch die Angabe der dargestellten Klassenhierarchie. Wir sagen: die Klassen Angestellter und Student *spezialisieren* die Klasse Person, die Klasse Hilfskraft *spezialisiert* die Klasse Student.

Beispiel

### LE 3 2 Konzepte und Notation für OOA

Abb. 2.6-1:  
Beispiel einer  
Vererbungs-  
struktur



Person ist als **abstrakte Klasse** modelliert, weil es – in diesem Modell – keine Objekte der Klasse Person geben kann. Abstrakte Klassen werden durch einen kursiv geschriebenen Namen gekennzeichnet. Sie können alternativ oder zusätzlich im Namensfeld der Klasse als {abstract} spezifiziert werden. Diese zweite Form ist vor allem bei handschriftlichen Modellen sinnvoll. Von einer abstrakten Klasse können keine Objekte erzeugt werden. Sie wird nur modelliert, um ihre Informationen an spezialisierte Klassen zu vererben.

Notation

Die Vererbung wird durch ein weißes bzw. transparentes Dreieck bei der Basisklasse gekennzeichnet. Die beiden Darstellung der Abb. 2.6-2 sind gleichwertig und können alternativ verwendet werden.



Abb. 2.6-2:  
Notation für Vererbung

**Was wird vererbt?**

- 1 Besitzen alle Objekte von SuperClass ein Attribut A, dann besitzen es auch alle Objekte von SubClass. Auch die Spezifikation des Attributs A hat in der Unterklasse Gültigkeit. Der Wert von AttributA wird hingegen nicht vererbt (Abb. 2.6-3).
- 2 Alle Operationen, die auf Objekte von SuperClass angewendet werden können, sind auch auf Objekte von SubClass anwendbar. Analoges gilt für Klassenoperationen.
- 3 Besitzt SuperClass ein Klassenattribut mit dem Wert W, so besitzt auch SubClass dieses Klassenattribut mit dem Wert W.
- 4 Existiert eine Assoziation zwischen SuperClass und einer Klasse AnyClass, dann wird diese Assoziation an SubClass vererbt.
- 5 Auf Objekte von SubClass können OperationA() und OperationB() angewendet werden.

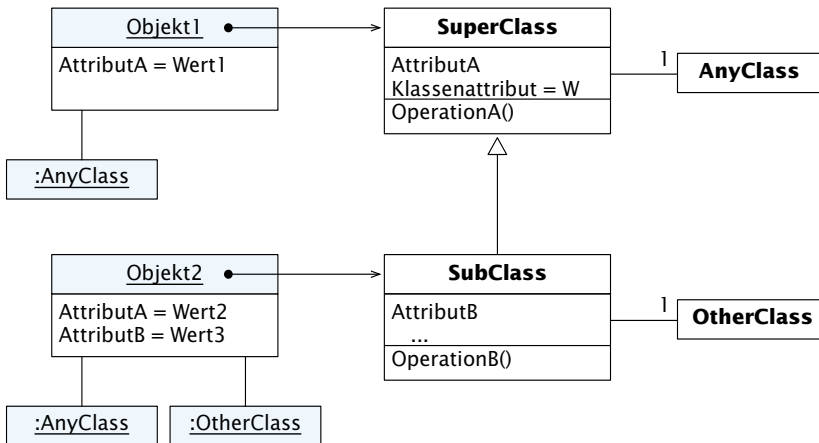


Abb. 2.6-3:  
Mechanismus der Vererbung

Unterklassen können das Verhalten ihrer Oberklassen verfeinern und redefinieren bzw. überschreiben (*redefine, override*). Das wird erreicht, indem die Unterklasse eine Operation gleichen Namens wie in der Oberklasse enthält. In der Abb. 2.6-4 wird auf Objekte der Klasse Sparkonto die Operation Sparkonto.buchen() und auf Objekte von Konto die Operation Konto.buchen() angewendet. Bei der Beschreibung von Sparkonto.buchen() wird im allgemeinen Konto.buchen() verwendet.

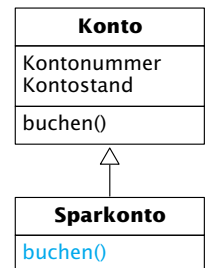


Abb. 2.6-4:  
Überschreiben einer Operation

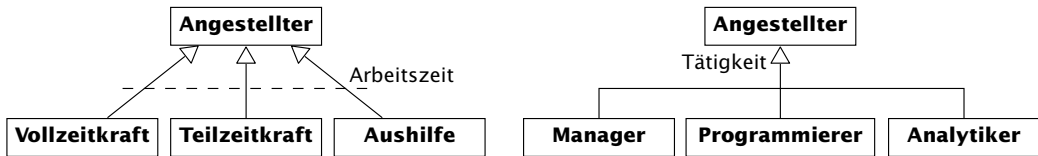
### LE 3 2 Konzepte und Notation für OOA

Einfachvererbung Die Vererbung wird in diesem Kapitel so definiert, daß jede Klasse höchstens eine direkte Oberklasse besitzt. Es entsteht eine Baumstruktur. Diese Form der Vererbung wird auch als **Einfachvererbung** bezeichnet. Die Mehrfachvererbung, bei der eine Klasse mehrere direkte Oberklassen besitzen kann, führe ich erst im Entwurf ein (Kapitel 6.6), weil sie bei der Erstellung von OOA-Modellen im allgemeinen nicht benötigt wird.

Kapitel 6.6  
Diskriminator Eine Vererbung kann zusätzlich durch einen Diskriminator (*discriminator*) bzw. ein Unterscheidungsmerkmal beschrieben werden. Er gibt an, nach welchem Kriterium eine Vererbungsstruktur erstellt wird. Die Unterklassen einer Oberklasse können verschiedene Diskriminatoren besitzen, die an die jeweiligen Vererbungs Pfeile angetragen werden. Besitzen alle Vererbungs Pfeile dasselbe Unterscheidungsmerkmal, dann bilden die Unterklassen eine homogene Spezialisierung.

Beispiel Von der Klasse **Angestellter** können nach dem Kriterium **Arbeitszeit** die Unterklassen **Vollzeitkraft**, **Teilzeitkraft** und **Aushilfe** gebildet werden. Werden **Angestellte** nach der Art der **Tätigkeit** spezialisiert, dann ergeben sich die Unterklassen **Manager**, **Programmierer** und **Analytiker** (Abb. 2.6-5).

Abb. 2.6-5:  
discriminator



### Vorteile und Nachteile der Vererbung

Das Konzept der Vererbung besitzt wesentliche Vorteile. Aufbauend auf existierenden Klassen können mit wenig Aufwand neue Klassen erstellt werden. Die Änderbarkeit wird unterstützt. Beispielsweise wirkt sich die Änderung von Attributen in der Oberklasse automatisch auf alle Unterklassen der Vererbungshierarchie aus. Nachteilig ist, daß diese automatische Änderung immer in Kraft tritt, auch dann, wenn sie vielleicht nicht erwünscht ist. Ein weiterer Nachteil ist die Verletzung des Geheimnisprinzips.

Das Konzept der Vererbung steht im Widerspruch zum Geheimnisprinzip. Das Geheimnisprinzip bedeutet, daß keine Klasse die Attribute einer anderen Klasse sieht. Barbara Liskov hat den Konflikt zwischen der Verkapselung und der Vererbung sehr elegant beschrieben /Khoshafian, Abnous 90/: »Ein Problem fast aller Vererbungsmechanismen ist, daß sie das Prinzip der Verkapselung auf das Äußerste strapazieren ... Wenn die Datenkapsel verletzt ist, verlieren wir die Vorteile der Lokalität. ... Um die Unterklasse zu verstehen, müssen wir sowohl die Ober- als auch die Unterklasse

betrachten. Falls die Oberklasse neu implementiert werden muß, dann müssen wir eventuell auch ihre Unterklassen neu implementieren.«

## 2.7 Paket

Ein **Paket** (*package*) faßt Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Sie können sich das vollständige Softwaresystem als ein großes Paket vorstellen, das alles andere enthält. Das Konzept des Pakets wird benötigt, um die Elemente des Modells in sinnvoller Weise zu gruppieren und die Systemstruktur auf einer hohen Abstraktionsebene zu beschreiben. Definition

Ein Warenwirtschaftssystem enthält die in Abb. 2.7-1 dargestellten Pakete. Beispiel



Abb. 2.7-1  
Pakete eines  
Warenwirtschafts-  
systems

Ein Paket wird als Rechteck mit einem Reiter dargestellt (Abb. 2.7-2). Wird der Inhalt des Pakets nicht gezeigt, dann wird der Paketname in das Rechteck geschrieben. Andernfalls wird der Paketname in den Reiter eingetragen. Der Paketname muß im gesamten System eindeutig sein. Notation

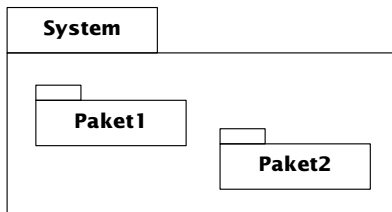


Abb. 2.7-2:  
Notation von  
Paketen

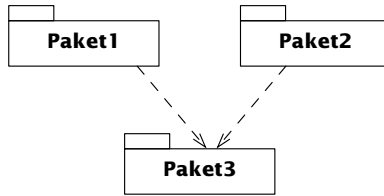
Eine gestrichelte Linie mit Pfeil (gerichtete Kante) modelliert eine Abhängigkeit zwischen zwei Paketen. Sie bedeutet, daß bei einer Änderung des Pakets an der Pfeilspitze *eventuell* auch das Paket am anderen Ende der gestrichelten Linie geändert werden muß »*the model element at the tail of the arrow depends on the model element at the arrowhead*« /UML 97/. Ob wirklich eine Änderung erforderlich ist, muß jeweils im Einzelfall geprüft werden. Abhängigkeiten

In Abb. 2.7-3 sind beispielsweise Paket1 und Paket2 von Paket3 abhängig. Dann *kann* eine Änderung in Paket3 auch eine Änderung in Paket1 und Paket2 zur Folge haben. Beispiel



### LE 3 2 Konzepte und Notation für OOA

Abb. 2.7-3:  
Abhängigkeit von  
Paketen



Paketdiagramm

Pakete werden in der UML im Klassendiagramm eingetragen. Enthält ein Klassendiagramm nur Pakete und deren Abhängigkeiten, so sprechen wir von einem Paketdiagramm.

Paket und Klasse

Jede Klasse (allgemeiner: jedes Modellelement) gehört zu höchstens einem Paket. Es kann jedoch in mehreren anderen Paketen darauf verwiesen werden. Ein Paket definiert einen Namensraum (*namespace*) für alle in ihm enthaltenen Modellelemente. Wird eine Klasse eines bestimmten Pakets in einem anderen Paket verwendet, dann wird als Klassenname `Paket::Klasse` verwendet. Bei geschichteten Paketen werden alle Paketnamen – jeweils durch `»::«` getrennt – vor den Klassennamen gesetzt, z.B.:

`Paket1::Paket11::Paket111::Klasse`.

verwandte  
Begriffe

Viele Methoden verwenden anstelle von `Paket` den Begriff `Subsystem` (*subsystem*). Auch *subject* und *category* sind gebräuchlich.

#### Abstrakte Klasse (*abstract class*)

Von einer abstrakten Klasse können keine Objekte erzeugt werden. Die abstrakte Klasse spielt eine wichtige Rolle in Vererbungsstrukturen, wo sie die Gemeinsamkeiten einer Gruppe von  $\rightarrow$ Unterklassen definiert. Damit eine abstrakte Klasse verwendet werden kann, muß von ihr zunächst eine Unterklasse abgeleitet werden.

**Aggregation (*aggregation*)** Eine Aggregation ist ein Sonderfall der  $\rightarrow$ Assoziation. Sie liegt dann vor, wenn zwischen den Objekten der beteiligten Klassen eine Beziehung besteht, die sich als *»ist Teil von«* oder *»besteht aus«* beschreiben läßt.

**Assoziation (*association*)** Eine Assoziation modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwischen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch  $\rightarrow$ Kardinalitäten und einen optionalen Assoziationsnamen oder Rollennamen. Sie kann um Re-

striktionen ergänzt werden. Besitzt eine Assoziation selbst wieder Attribute und ggf. Operationen und Assoziationen zu anderen Klassen, dann wird sie zur  $\rightarrow$ assoziativen Klasse. Die Qualifikationsangabe (*qualifier*) zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Eine abgeleitete Assoziation liegt vor, wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sonderfälle der Assoziation sind die  $\rightarrow$ Aggregation und die  $\rightarrow$ Komposition. In der Analyse ist jede Assoziation inhärent bidirektional.

#### Assoziative Klasse (*association class*)

Eine assoziative Klasse besitzt sowohl die Eigenschaften der  $\rightarrow$ Assoziation als auch die der Klasse.

#### CRC-Karte (*Class/Responsibility/Collaboration*)

Eine CRC-Karte ist eine Karteikarte. Oben auf der Karte wird der Name der Klasse (*class*) eingetragen. Die restliche Karte wird in zwei Hälften geteilt. Auf der einen Hälfte werden die Verantwortlichkeiten (*re-*



*sponsibilities*) der Klasse notiert. Darunter sind sowohl das Wissen der Klasse als auch die zur Verfügung gestellten Operationen zu verstehen. Auf der rechten Seite wird eingetragen, mit welchen anderen Klassen die beschriebene Klasse zusammenarbeiten muß (*collaborations*).

**Einfachvererbung** Bei der Einfachvererbung besitzt jede Unterklasse genau eine direkte Oberklasse. Es entsteht eine Baumstruktur.

**Kardinalität (*multiplicity*)** Die Kardinalität bezeichnet die Wertigkeit einer →Assoziation, d.h. sie spezifiziert die Anzahl der an der Assoziation beteiligten Objekte.

**Klassendiagramm (*class diagram*)** Das Klassendiagramm stellt die Klassen, die →Vererbung und die →Assoziationen zwischen Klassen dar. Zusätzlich können →Pakete modelliert werden.

**Komposition (*composition*)** Die Komposition ist eine besondere Form der →Aggregation. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch einem anderen Ganzen zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile.

**Oberklasse (*super class*)** In einer Vererbungsstruktur heißt jede Klasse, von der eine Klasse Eigenschaften und Verhalten erbt, Oberklasse dieser Klasse. Mit anderen Worten: Eine Oberklasse ist eine Klasse, die mindestens eine Unterklasse besitzt.

**Paket (*package*)** Ein Paket faßt Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene auszudrücken. Pakete können im Paketdiagramm dargestellt werden.

**Qualifikationsangabe (*qualifier*)** Die Qualifikationsangabe ist ein spezielles Attribut der →Assoziation, dessen Wert ein oder mehrere Objekte auf der anderen Seite der Assoziation selektiert. Mit anderen Worten: Die Qualifikationsangabe zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Der *qualifier* kann auch aus mehreren Attributen bestehen.

**Rolle (*role name*)** Die Rolle beschreibt, welche Bedeutung ein Objekt in einer →Assoziation wahrnimmt. Eine binäre Assoziation besitzt maximal zwei Rollen.

**Unterklasse (*sub class*)** Jede Klasse, die in einer Vererbungshierarchie Eigenschaften und Verhalten von anderen Klassen erbt, ist eine Unterklasse dieser Klassen. Mit anderen Worten: Eine Unterklasse besitzt immer Oberklassen.

**Vererbung (*generalization*)** Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse (Basisklasse) und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und →Assoziationen der Basisklasse. Operationen der Basisklasse dürfen redefiniert werden. Es entsteht eine Klassenhierarchie.



Aufbauend auf den Basiskonzepten, ermöglichen die folgenden Konzepte die Erstellung eines statischen Modells. Die **Assoziation** modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Wir verwenden insbesondere binäre Assoziationen. Sonderfälle der Assoziation sind die **Aggregation** und die **Komposition**. Die **Vererbung** beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer spezialisierten Klasse. Das **Paket** gruppiert Modellelemente und ermöglicht eine Darstellung des Softwaresystems auf einem höheren Abstraktionsniveau. Das statische Modell wird in der UML im **Klassendiagramm** spezifiziert.

### LE 3 Aufgaben

- Aufgabe 1 *Lernziel: Klassendiagramm und Objektdiagramm erstellen können.*  
20–25 Minuten

Identifizieren Sie anhand der folgenden Beschreibung Klassen, Attribute, Operationen und Assoziationen und zeichnen Sie sie in ein Klassendiagramm ein. Erstellen Sie zusätzlich für selbstgewählte Beispieldaten ein Objektdiagramm.

Eine Tagung (z.B. Softwaretechnik-Tagung in Hamburg) ist zu organisieren. Für jeden Teilnehmer der Tagung werden der Name, die Adresse und der Status (Student, Mitglied, Nichtmitglied) gespeichert. Jeder Teilnehmer kann sich für ein oder mehrere halbtägige Tutorien, die zusätzlich zum normalen Tagungsprogramm angeboten werden, anmelden. Für jedes Tutorium werden dessen Nummer, die Bezeichnung sowie das Datum gespeichert. Alle Tutorien kosten gleich viel. Damit ein Tutorium stattfindet, müssen mindestens 10 Anmeldungen vorliegen. Jedes Tutorium wird von genau einem Referenten angeboten. Für jeden Referenten werden dessen Name und Firma gespeichert. Ein Referent kann sich auch für ein oder mehrere Tutorien – anderer Referenten – anmelden und kann bei diesen kostenlos zuhören. Diese Anmeldungen zählen bei der Ermittlung der Mindestanmeldungen nicht mit. Ein Teilnehmer kann nicht gleichzeitig Referent sein. Ein Referent kann mehrere Tutorien anbieten. An einem Tutorium können mehrere Referenten kostenlos teilnehmen. Ein Teilnehmer kann sich in der Tagungsanmeldung auch für einige Rahmenprogramme (z.B. Besuch eines Musicals) eintragen lassen. Für jedes Rahmenprogramm werden dessen Bezeichnung, das Datum, die Zeit, der Ort und die Kosten gespeichert.

- Aufgabe 2 *Lernziele: Klassendiagramm und Objektdiagramm erstellen können.*  
15–20 Minuten

Identifizieren Sie anhand der folgenden Beschreibung Klassen, Attribute, Operationen, Assoziationen und Vererbungsstrukturen und zeichnen Sie sie in ein Klassendiagramm ein. Prüfen Sie, welche Art der Assoziation vorliegt. Erstellen Sie zusätzlich für selbstgewählte Beispieldaten ein Objektdiagramm.

Wir betrachten eine Bank und ihre Kunden. Eine Person wird Kunde, wenn sie ein Konto eröffnet. Ein Kunde kann beliebig viele weitere Konten eröffnen. Für jeden neuen Kunden werden dessen (nicht notwendigerweise eindeutiger) Name, Adresse und das Datum der ersten Kontoeröffnung erfaßt. Bei der Kontoeröffnung muß der Kunde gleich eine erste Einzahlung vornehmen. Wir unterscheiden Girokonten und Sparkonten. Girokonten dürfen bis zu einem bestimmten Betrag überzogen werden. Für jedes Konto wird ein individueller Habenzins, für Girokonten auch ein individueller Sollzins festgelegt; außerdem besitzt jedes Konto eine eindeutige Kontonummer. Für jedes Sparkonto wird die Art des

Sparens – z.B. Festgeld – gespeichert. Ein Kunde kann Beträge einzahlen und abheben. Desweiteren werden Zinsen gutgeschrieben und bei Girokonten Überziehungszinsen abgebucht. Um die Zinsen zu berechnen, muß für jede Kontobewegung das Datum und der Betrag notiert werden. Die Gutschrift/Abbuchung der Zinsen erfolgt bei den Sparkonten jährlich und bei den Girokonten quartalsweise. Ein Kunde kann jedes seiner Konten wieder auflösen. Bei der Auflösung des letzten Kontos hört er auf, Kunde zu sein.

**3 Lernziel: Pakete bilden können.**

Aufgabe  
5 Minuten

Von den folgenden Klassen gehört jede zu einem Paket. Gruppieren Sie die aufgeführten Klassen in Pakete. Wählen Sie für jedes Paket einen aussagefähigen Namen.

Artikel  
Auftragsposten  
Bestellartikel  
Bestellposten  
Bestellung an Lieferanten  
Kunde  
Kundenauftrag  
Lager  
Lagerartikel  
Lagerplatz  
Lagerverwalter  
Lieferant  
Lieferkondition

## 2 Konzepte und Notation der objektorientierten Analyse (Dynamische Konzepte)



- Erklären können, was ein Geschäftsprozeß ist.
- Erklären können, was eine Botschaft ist.
- Erklären können, was ein Szenario ist.
- Erklären können, was ein Zustandsautomat ist und welche Rolle er im dynamischen Modell spielt.
- Erklären können, was ein Aktivitätsdiagramm ist.
- Erklären können, wie das Klassendiagramm und Diagramme des dynamischen Modells zusammenwirken.
- Geschäftsprozesse modellieren können.
- Geschäftsprozesse spezifizieren können.
- Sequenz- und Kollaborationsdiagramme erstellen können.
- Zustandsdiagramme erstellen können.

verstehen

anwenden



Sie müssen die Kapitel 2.1 bis 2.7 durchgearbeitet haben.



- 2.8 [Geschäftsprozeß](#) 62
- 2.9 [Botschaft](#) 69
- 2.10 [Szenario](#) 70
- 2.11 [Zustandsautomat](#) 78

## 2.8 Geschäftsprozeß

Definition Jacobson hat den Begriff des *use case* in Zusammenhang mit einer objektorientierten Methode erstmalig 1987 auf einer Konferenz vorgestellt. Durch sein Buch *Software Engineering: Use Case Driven Approach* /Jacobson92/ wurde der Geschäftsprozeß zum allgemeinen Gedankengut in der Objektmodellierung. Diese Ideen wurden von Jacobson weiterentwickelt und in seinem Buch *The Object Advantage: Business Process Reengineering with Object Technology* veröffentlicht. Obwohl das Konzept des *use case* prinzipiell völlig unabhängig von der objektorientierten Modellierung ist, besitzt es heute einen festen Platz in den meisten objektorientierten Methoden und auch in der UML.

zur Terminologie /Jacobson 94/ unterscheidet zwischen dem *use case* in einem Informationssystem und dem *use case* in einem Unternehmen.

Der **use case in einem Informationssystem** wird definiert »als eine Sequenz von zusammengehörenden Transaktionen, die von einem Akteur im Dialog mit einem System ausgeführt werden, um für den Akteur ein Ergebnis von meßbarem Wert zu erstellen«. Meßbarer Wert bedeutet, daß die durchgeführte Aufgabe einen sichtbaren, quantifizierbaren und/oder qualifizierbaren Einfluß auf die Systemumgebung hat. Eine Transaktion ist eine Menge von Verarbeitungsschritten, von denen entweder alle oder keiner ausgeführt werden.

Der *use case* in einem Informationssystem spezifiziert die Interaktionen zwischen einem Akteur und dem System, d.h. er beschreibt eine spezielle Benutzung des Systems. Ein solcher *use case* kann mehr oder weniger umfangreich sein und auf eine oder mehrere Benutzerfunktionen abgebildet werden. Alle *use cases* zusammen dokumentieren alle Möglichkeiten der Benutzung des Systems (*use case model*).

Der **use case in einem Unternehmen** (*business system*) wird von /Jacobson 94/ »als eine Sequenz von Transaktionen in einem System (=Unternehmen)« definiert. Die ausgeführte Aufgabe soll für den Akteur des Unternehmens von meßbarem Wert sein. In diesem Kontext setzt Jacobson den Begriff *use case* mit dem *business process* gleich (»a use case is our construct for business process«).

Entsprechend obiger Definitionen kann ein *use case* auf zwei unterschiedlichen Abstraktionsebenen erstellt werden. Auf Unternehmensebene handelt es sich um einen Unternehmensprozeß (*business process*), der aus einer Anzahl von unternehmensinternen Aktivitäten besteht, die durchgeführt werden, um die Wünsche eines Kunden zu befriedigen. Handelt es sich dagegen um ein Softwaresystem, dann definiert ein *use case* eine spezielle Benutzung der Software.

Bei der in diesem Buch verwendeten Vorgehensweise ist die Identifikation der *uses cases* der erste Schritt in der Analyse. Hier soll

zunächst ermittelt werden, welche Aufgaben mit dem neuen Softwaresystem zu bewältigen sind, um die gewünschten Ergebnisse zu erzielen. Zu diesem frühen Zeitpunkt ist noch nicht abzusehen, ob jeder *use case* ausschließlich durch Software realisiert wird oder auch organisatorische Schritte enthält, in denen der Bediener Entscheidungen treffen oder bestimmte Aktivitäten durchführen muß. Ich übersetze daher den Begriff *use case* im folgenden mit Geschäftsprozeß (siehe auch /Hruschka 98/), um auszudrücken, daß nicht die Funktionalität der Software, sondern die ergebnisorientierten Arbeitsabläufe bei Benutzung der zu realisierenden Software zu spezifizieren sind.

Ein **Geschäftsprozeß** (*use case*) besteht aus mehreren zusammenhängenden Aufgaben, die von einem Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen. Definition

Ein **Akteur** (*actor*) ist eine Rolle, die ein Benutzer des Systems spielt. Jeder Akteur hat einen gewissen Einfluß auf das System. Ein Akteur ist häufig eine Person. Es kann sich ebenso um eine Organisationseinheit oder ein externes System handeln, das mit dem zu modellierenden System kommuniziert. Akteure befinden sich stets außerhalb des Systems. Akteur

Stellt das betrachtete System ein Handelshaus dar (Abb. 2.8-1), dann sind Kunde und Lieferant Akteure. Die Buchhaltung ist dagegen kein Akteur des Handelshauses, denn sie befindet sich innerhalb dieses Systems. Bei dem Softwaresystem, das Auftrags- und Bestellwesen unterstützt, ist dagegen die Buchhaltung ein Akteur, denn diese Abteilung ist außerhalb dieses Systems, muß jedoch mit dem Softwaresystem kommunizieren. In einer kleinen Firma können die Aufgaben des Kunden- und Lieferantensachbearbeiters durchaus von der gleichen Person ausgeführt werden. Trotzdem werden zwei Akteure – die Rollen, die diese Person spielt – identifiziert. Beispiel

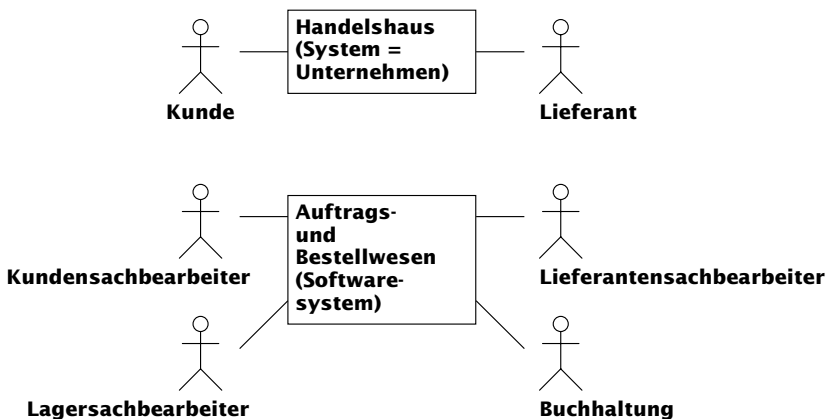


Abb. 2.8-1:  
Wer ist Akteur?

## LE 4 2 Konzepte und Notation für OOA

ziert. Bei der Modellierung von Softwaresystemen sind die Akteure also diejenigen, die das System später bedienen bzw. Ergebnisse dieses Systems erhalten.

**Spezifikation** Ein Geschäftsprozeß wird semiformal oder informal (umgangssprachlich) beschrieben. Ein Grundprinzip der Modellierung ist die Trennung von Funktionalität und Benutzungsoberfläche. Die Oberfläche ändert sich aufgrund neuer Techniken sehr schnell und oft muß die Software auf verschiedenen Plattformen laufen. Beschreiben Sie daher die Funktionalität eines Geschäftsprozesses ohne Bezüge zur Benutzungsoberfläche.

**Beispiel** Die Bearbeitung eines Auftrags in einem Versandhaus hängt von vielen Faktoren ab. Beispielsweise, ob es sich um einen Neukunden handelt oder ob alle gewünschten Artikel lieferbar sind. Alle diese Aufgaben werden unter dem Geschäftsprozeß Auftrag ausführen zusammengefaßt, den wir zunächst umgangssprachlich beschreiben.

Geschäftsprozeß: Auftrag ausführen

Eine Kundenbestellung kommt in der Versandabteilung an. Neukunden werden im System registriert und der Versand an diese Kunden erfolgt ausschließlich per Nachnahme oder per Bankeinzug. Für alle lieferbaren Artikel wird die Rechnung erstellt und als Auftrag an das Lager weitergegeben. Sind einige der gewünschten Artikel nicht lieferbar, so wird der Kunde informiert. Alle erstellten Rechnungen werden an die Buchhaltung weitergegeben.

An diesen Geschäftsprozeß sind folgende Akteure beteiligt: Kundensachbearbeiter, Lagersachbearbeiter und Buchhaltung.

**Notation** Während bei einfachen Geschäftsprozessen eine umgangssprachliche Beschreibung ausreicht, kann bei umfangreicheren Spezifikationen folgende **Geschäftsprozeßschablone** (*use case template*) sinnvoll eingesetzt werden /Cockburn 97/. Diese Schablone sollte als Checkliste betrachtet werden, d.h. sie ist nicht für jeden Geschäftsprozeß vollständig auszufüllen.

**Schablone für Geschäftsprozesse** *Geschäftsprozeß*: Name, bestehend aus zwei oder drei Wörtern (was wird getan?).

*Ziel*: globale Zielsetzung bei erfolgreicher Ausführung des Geschäftsprozesses.

*Kategorie*: primär, sekundär oder optional.

*Vorbedingung*: Erwarteter Zustand, bevor der Geschäftsprozeß beginnt.

*Nachbedingung Erfolg*: Erwarteter Zustand nach erfolgreicher Ausführung des Geschäftsprozesses, d.h. Ergebnis des Geschäftsprozesses.

*Nachbedingung Fehlschlag*: Erwarteter Zustand, wenn das Ziel nicht erreicht werden kann.



*Akteure:* Rollen von Personen oder andere Systeme, die den Geschäftsprozeß auslösen oder daran beteiligt sind.

*Auslösendes Ereignis:* Wenn dieses Ereignis eintritt, dann wird der Geschäftsprozeß initiiert.

*Beschreibung:*

**1** Erste Aktion

**2** Zweite Aktion

*Erweiterungen:*

**1a** Erweiterung des Funktionsumfangs der ersten Aktion

*Alternativen:*

**1a** Alternative Ausführung der ersten Aktion

**1b** Weitere Alternative zur ersten Aktion

Der betrachtete Geschäftsprozeß kann nur ausgeführt werden, wenn die genannte **Vorbedingung** erfüllt ist. Die **Nachbedingung** eines Geschäftsprozesses A kann für einen Geschäftsprozeß B eine Vorbedingung bilden. Diese Angaben bestimmten also, in welcher Reihenfolge Geschäftsprozesse ausgeführt werden können.

Unter »Beschreibung« erfolgt eine umgangssprachliche Spezifikation des Geschäftsprozesses. Die einzelnen Aufgaben werden der besseren Übersicht halber nummeriert. Wichtig ist, daß hier zunächst der Standardfall, d.h. der Fall, der am häufigsten auszuführen ist, beschrieben wird. Alle seltener eingesetzten Fälle werden unter »Erweiterungen« ausgeführt, wenn sie zusätzlich zu einer Aktion der Standardverarbeitung ausgeführt werden und unter »Alternativen«, wenn sie eine Aktion der Normalverarbeitung ersetzen.

Die Kategorie eines Geschäftsprozesses ist

- primär, wenn er notwendiges Verhalten beschreibt, das häufig benötigt wird,
- sekundär, wenn er notwendiges Verhalten beschreibt, das selten benötigt wird,
- optional, wenn er ein Verhalten beschreibt, das für den Einsatz des Systems zwar nützlich, aber nicht unbedingt notwendig ist.

Wir spezifizieren nun obiges Beispiel eines Geschäftsprozesses mittels Schablone, die bereits bei diesem einfachen Geschäftsprozeß sehr vorteilhaft eingesetzt werden kann. Beispielsweise muß sich der Analytiker Gedanken darüber machen, ob existierende Kunden oder Neukunden den Standardfall bilden, wobei hier der erste Fall gewählt wurde. Beispiel

*Geschäftsprozeß:* Auftrag ausführen

*Ziel:* Ware an Kunden geliefert

*Vorbedingung:* –

*Nachbedingung Erfolg:* Ware ausgeliefert (auch Teillieferungen), Rechnungskopie bei Buchhaltung

*Nachbedingung Fehlschlag:* Mitteilung an Kunden, daß nichts lieferbar ist

*Akteure:* Kundensachbearbeiter, Lagersachbearbeiter, Buchhaltung

*Auslösendes Ereignis:* Bestellung des Kunden liegt vor

## LE 4 2 Konzepte und Notation für OOA

*Beschreibung:*

- 1 Kundendaten abrufen
- 2 Lieferbarkeit prüfen
- 3 Rechnung erstellen
- 4 Auftrag vom Lager ausführen lassen
- 5 Rechnungskopie an Buchhaltung geben

*Erweiterung:*

- 1a Kundendaten aktualisieren

*Alternativen:*

- 1a Neukunden erfassen
- 3a Rechnung mit Nachnahme erstellen
- 3b Rechnung mit Bankeinzug erstellen

Geschäftsprozeßdiagramm Das Zusammenspiel mehrerer Geschäftsprozesse untereinander und mit den Akteuren wird im **Geschäftsprozeßdiagramm** (*use case diagram*) beschrieben (Abb. 2.8-2). Es gibt auf hohem Abstraktionsniveau einen guten Überblick über das System und seine Schnittstellen zur Umgebung. Die Akteure werden – auch wenn es sich um ein externes System handelt – als Strichmännchen eingetragen, die Geschäftsprozesse als Ovale. Eine Linie zwischen Akteur und Geschäftsprozeß bedeutet, daß eine Kommunikation stattfindet.

Abb. 2.8-2:  
Notation für  
Geschäftsprozeß-  
diagramm

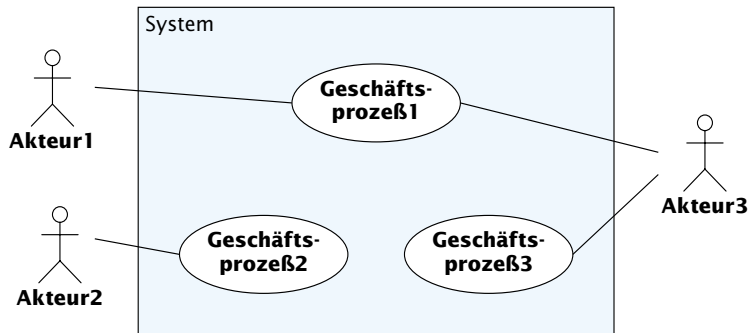
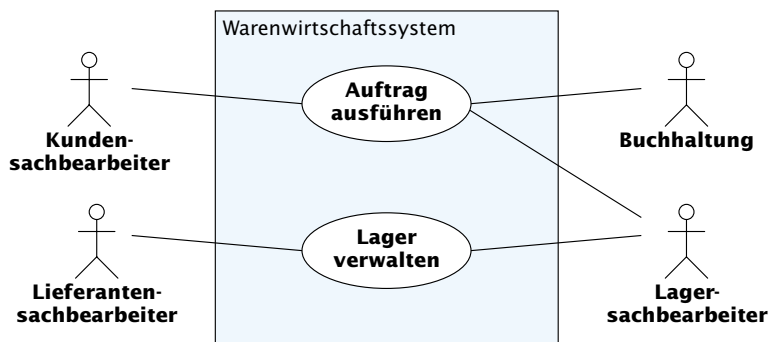


Abb. 2.8-3 modelliert einen Ausschnitt aus einem vereinfachten Informationssystem als Geschäftsprozeßdiagramm.

Abb. 2.8-3:  
Geschäftsprozeß-  
diagramm für ein  
Informationssystem



Zwischen Geschäftsprozessen können zwei Arten von Beziehungen existieren.

Eine *extends*-Beziehung liegt vor, wenn ein Geschäftsprozeß B ähnlich einem Geschäftsprozeß A ist, aber »etwas mehr als A tut«. Für die Darstellung der *extends*-Beziehung wird der Vererbungs Pfeil verwendet und mit dem Stereotypen «extends» beschriftet. Die *extends*-Beziehung ermöglicht es, einen komplexen Geschäftsprozeß zunächst in vereinfachter Form zu spezifizieren und komplexe Sonderfälle in die Erweiterungen zu verlagern.

*extends*-Beziehung

Wir können den Geschäftsprozeß Auftrag ausführen durch den Geschäftsprozeß Auftrag mit Nachlieferung ausführen erweitern (Abb. 2.8-4). Während der Kunde im ersten Fall nur informiert wird, welche Artikel nicht lieferbar sind, erhält er im zweiten Fall die Information, daß diese Artikel zu einem späteren Termin geliefert werden. Kann dieser Termin nicht gehalten werden, so wird der Kunde wieder entsprechend informiert.

Beispiel

Eine *uses*-Beziehung liegt vor, wenn zwei Geschäftsprozesse B1 und B2 ein gemeinsames Verhalten besitzen, das in dem Geschäftsprozeß A spezifiziert ist. Der Geschäftsprozeß A wird analog zu einem Unterprogramm aufgerufen bzw. benutzt. Die *uses*-Beziehung erspart die mehrmalige – redundante – Beschreibung des gleichen Verhaltens. Für die Darstellung wird ebenfalls der Vererbungs Pfeil verwendet, der mit dem Stereotypen «uses» beschriftet wird.

*uses*-Beziehung

In der Abb. 2.8-4 verwenden die beiden Geschäftsprozesse Wareneingang aus Einkauf bearbeiten und Wareneingang aus Produktion bearbeiten beide den Geschäftsprozeß Ware einlagern.

Beispiel

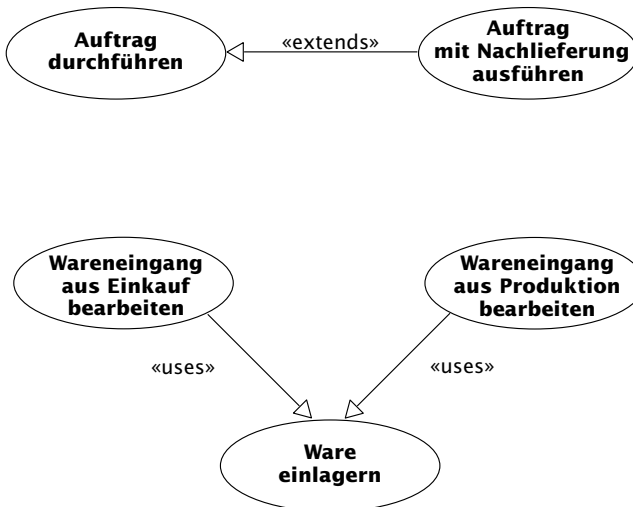


Abb. 2.8-4: *extends*- und *uses*-Beziehung

## LE 4 2 Konzepte und Notation für OOA

konkret und  
abstrakt

Die beiden erstgenannten sind Geschäftsprozesse, die sich aus den Anforderungen des Ein- und Verkaufs ableiten lassen und beschreiben jeweils komplette Abläufe. Sie werden als **konkrete Geschäftsprozesse** bezeichnet. Der Geschäftsprozeß Ware einlagern ist dagegen ein künstliches Gebilde, das nur zum Zweck der Verwendung durch konkrete Geschäftsprozesse existiert. Wir sprechen daher analog zur abstrakten Klasse von einem **abstrakten Geschäftsprozeß**. Abstrakte und konkrete Geschäftsprozesse werden in der UML-Notation nicht unterschieden.

Aktivitätsdiagramm

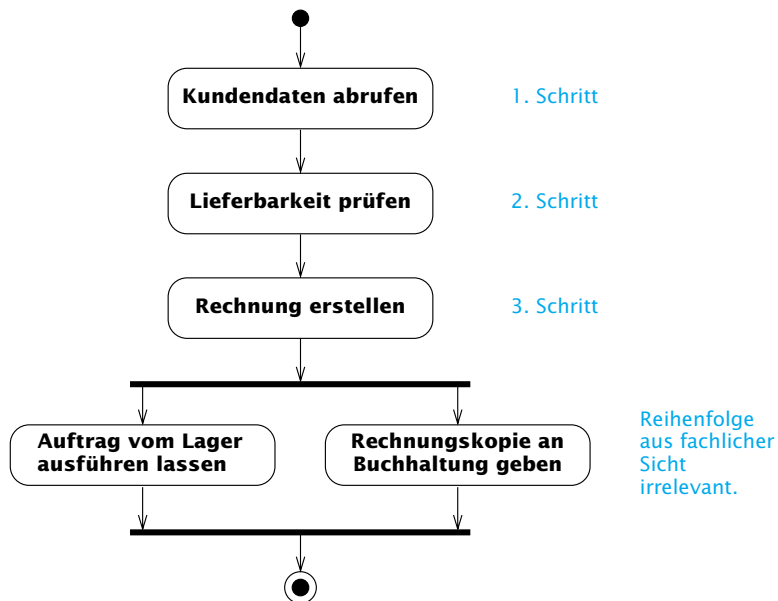
Die oben verwendete Schablone ist ein einfaches, aber effektives Hilfsmittel zur Beschreibung von Geschäftsprozessen. Ein Nachteil ist jedoch, daß sie die Sequenz der durchzuführenden Schritte festlegt. Es kann nicht ausgedrückt werden, daß für bestimmte Schritte die Reihenfolge aus fachlicher Sicht keine Rolle spielt. Diese Möglichkeit bietet in der UML das **Aktivitätsdiagramm** (*activity diagram*). Es ist ein Sonderfall des Zustandsdiagramms, dessen Notation im Kapitel 2.11 ausführlich behandelt wird.

Kapitel 2.11



**Beispiel** Die Abb. 2.8-5 modelliert die (Standardfall-) Beschreibung des Geschäftsprozesses Auftrag ausführen als Aktivitätsdiagramm. Dieses Diagramm sagt aus, daß die ersten drei Schritte – aus fachlicher Sicht – in der angegebenen Reihenfolge auszuführen sind. Die beiden weiteren Schritte können in beliebiger Reihenfolge oder auch – durch zwei Prozessoren – parallel bearbeitet werden. Eine weitere Verarbeitung ist aber erst dann möglich, wenn beide Teilaufgaben erledigt wurden.

Abb. 2.8-5:  
Aktivitätsdiagramm  
für die Standard-  
Beschreibung von  
Auftrag bearbeiten



In der deutschen Literatur wird der Begriff *use case* teilweise mit »Anwendungsfall« übergesetzt. Häufig wird auch der englische Originalbegriff verwendet. Andere Begriffe sind: Geschäftsfall, Geschäftsvorfall, *workflow* und *business process*.

verwandte  
Begriffe

## 2.9 Botschaft

Eine **Botschaft** (*message*) ist die Aufforderung eines Senders (*client*) an einen Empfänger (*server, supplier*) eine Dienstleistung zu erbringen. Der Empfänger interpretiert diese Botschaft und führt eine Operation aus. Der Sender der Botschaft weiß nicht, *wie* die entsprechende Operation ausgeführt wird. Die Menge der Botschaften, auf die Objekte einer Klasse reagieren können, wird als Protokoll (*protocol*) der Klasse bezeichnet.

Definition

Eine Botschaft löst eine Operation gleichen Namens aus. Das Verhalten eines objektorientierten Systems wird somit durch die Botschaften beschrieben, mit denen Objekte untereinander kommunizieren. Botschaften können in der UML in verschiedenen Diagrammen dargestellt werden.

Für jede Filiale einer Versicherung soll der Durchschnittsumsatz berechnet werden. Um für jeden Versicherten die Beitragssumme zu ermitteln, muß der Jahresbeitrag eines jeden Vertrags dieses Versicherten bekannt sein (Abb. 2.9-1). Wenn die Filiale die Botschaft `berechneDurchschnittsumsatz()` erhält, dann sendet sie jedem ihrer Versicherten die Botschaft `berechneBeitrag()`, die wiederum die Botschaft `getJahresbeitrag()` an alle ihre Vertrags-Objekte schickt. Beachten Sie, daß wegen der modellierten Assoziationen jede Filiale ihre Versicherten und jedes Versicherten-Objekt seine Vertrags-Objekte kennt. Für die Darstellung dieser Botschaften verwende ich ein Kollaborationsdiagramm, das im Kapitel 2.10 ausführlich beschrieben wird.

Beispiel



Kapitel 2.10

Zur Erinnerung: Verwaltungsoperationen, z.B. `getJahresbeitrag()`, werden nicht in das Klassendiagramm eingetragen.

Wenn ein Objekt in einer Vererbungsstruktur eine Botschaft erhält, dann läßt sich dies folgendermaßen erklären: Das Objekt »schaut« in seiner eigenen Liste der Operationen nach, ob es eine entsprechende Operation besitzt. Wenn ja, dann wird diese Operation ausgeführt. Andernfalls wird die Suche bei der direkten Oberklasse fortgesetzt.

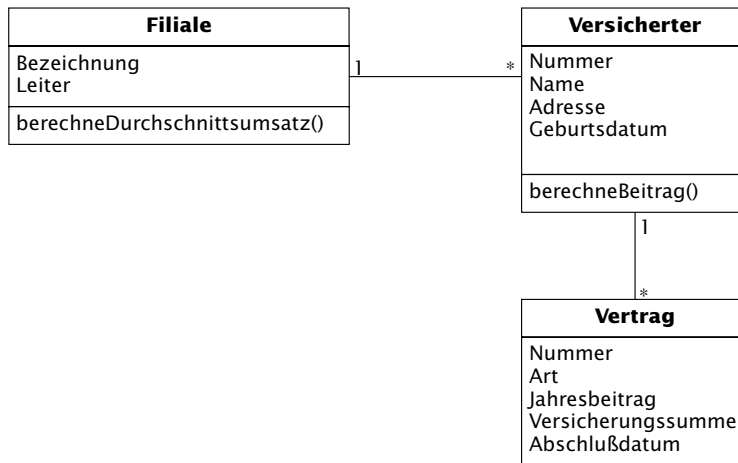
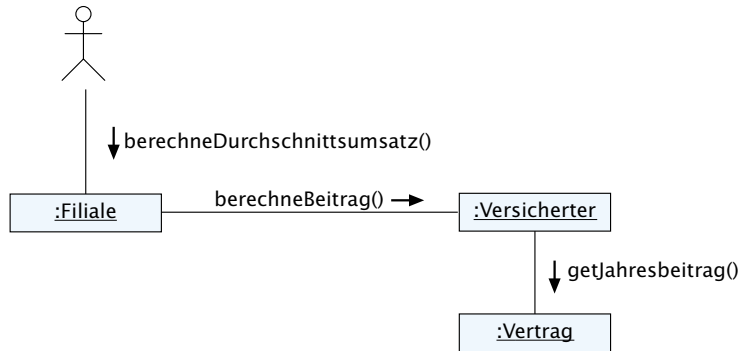
Botschaften und  
Vererbung

Anstelle des Begriffs *Botschaft* wird in der deutschen objektorientierten Literatur auch *Nachricht* verwendet. Der Begriff *Nachricht* ist jedoch bereits in den Bereichen der Datenübertragung und der Betriebssysteme mit einer anderen Bedeutung belegt /Rechen-

verwandte  
Begriffe

## LE 4 2 Konzepte und Notation für OOA

Abb. 2.9-1:  
Senden von  
Botschaften



berg, Pomberger 97/, während *Botschaft* nur mit Objektorientierung assoziiert ist. Teilweise wird auch von *Operationsaufruf* und *Methodenaufruf* gesprochen. In der englischen Literatur ist der Begriff *message* üblich.

## 2.10 Szenario

Definition Ein **Szenario** ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen ist. Diese Schritte sollen das Hauptziel des Akteurs realisieren und ein entsprechendes Ergebnis liefern. Sie beginnen mit dem auslösenden Ereignis und werden fortgesetzt, bis das Ziel erreicht ist oder aufgegeben wird /Cockburn 97/.

Ein Geschäftsprozeß wird durch eine Kollektion von Szenarios dokumentiert. Jedes Szenario wird durch eine oder mehrere Bedin-

gungen definiert, die zu einem speziellen Ablauf des jeweiligen Geschäftsprozesses führen. Es lassen sich zwei Kategorien von Szenarios unterscheiden: Szenarios, die eine erfolgreiche Bearbeitung des Geschäftsprozesses beschreiben, und Szenarios, die zu einem Fehlschlag führen.

In der deutschen Literatur wird häufig Szenarien als Plural von Szenario verwendet. Der Plural von Szenario ist laut Duden jedoch Szenarios, während Szenarien der Plural von Szenarium ist. Daher verwende ich den korrekten – wenn auch nicht so gebräuchlichen – Begriff.

Hinweis

Aus dem Geschäftsprozeß Auftrag bearbeiten aus Kapitel 2.8 lassen sich folgende Szenarios ableiten:

Beispiele

- 1 Auftrag für einen Neukunden bearbeiten, wenn mindestens ein Artikel lieferbar ist.
- 2 Auftrag bearbeiten, wenn der Kunde bereits existiert und mindestens ein Artikel lieferbar ist.
- 3 Auftrag bearbeiten, wenn der Kunde bereits existiert, sich seine Daten geändert haben und mindestens ein Artikel lieferbar ist.

Szenarios werden durch **Interaktionsdiagramme** (*interaction diagrams*) modelliert. Die UML bietet zwei Arten von Diagrammen an: das **Sequenzdiagramm** (*sequence diagram*) und das **Kollaborationsdiagramm** (*collaboration diagram*).

Notation

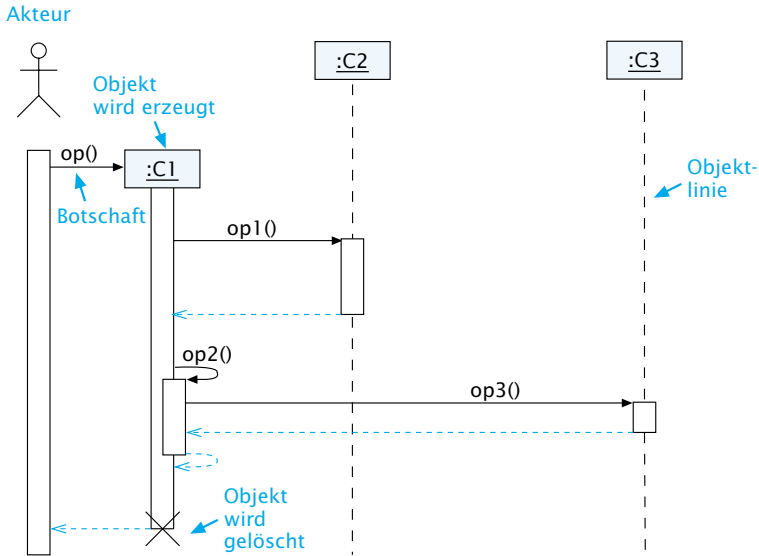
Ein Sequenzdiagramm besitzt zwei Dimensionen: die Vertikale repräsentiert die Zeit, auf der Horizontalen werden die Objekte eingetragen. Jedes Objekt wird durch eine gestrichelte Linie im Diagramm – die **Objektlinie** – dargestellt (Abb. 2.10-1). Diese Linie repräsentiert die Existenz eines Objekts während einer bestimmten Zeit. Die Linie beginnt nach dem Erzeugen des Objekts und endet mit dem Löschen des Objekts. Existiert ein Objekt während der gesamten Ausführungszeit des Szenarios, so ist die Linie von oben nach unten durchgezogen. Am oberen Ende der Linie wird ein **Objektsymbol** angetragen. Wird ein Objekt im Laufe der Ausführung erst erzeugt, dann zeigt eine Botschaft auf dieses Objektsymbol. Das Löschen des Objekts wird durch ein großes »X« markiert. Die Reihenfolge der Objekte ist beliebig. Sie soll so gewählt werden, daß ein übersichtliches Diagramm entsteht. Bei diesen Objekten handelt es sich im allgemeinen nicht um spezielle Objekte, sondern sie bilden Stellvertreter für beliebige Objekte der angebenen Klasse. Daher werden sie häufig als anonyme Objekte (d.h. :Klasse) bezeichnet.

Notation Sequenzdiagramm

In das Sequenzdiagramm werden die Botschaften eingetragen, die zum Aktivieren der Operationen dienen. Jede Botschaft wird als gerichtete Kante (mit gefüllter Pfeilspitze) vom Sender zum Empfänger gezeichnet. Der Pfeil wird mit dem Namen der aktivierten Operation beschriftet. Die Botschaft aktiviert eine Operation glei-

## LE 4 2 Konzepte und Notation für OOA

Abb. 2.10-1:  
Notation Sequenz-  
diagramm  
(Basiselemente)



chen Namens. Diese wird durch ein schmales Rechteck auf der Objektlinie angezeigt. Nach dem Beenden der Operation zeigt eine gestrichelte – blaue – Linie mit offener Pfeilspitze, daß der Kontrollfluß zur aufrufenden Operation zurückgeht. Bei der Modellierung sequentieller Systeme kann auf diese Angabe verzichtet werden, da nach der Ausführung einer Operation der Kontrollfluß *immer* an den Aufrufer zurückkehrt. Bei sehr komplexen Sequenzdiagrammen – wie sie teilweise im Entwurf vorkommen – erleichtert ein Rückgabepfeil allerdings die Nachvollziehbarkeit des Kontrollflusses. Bei paralleler Datenverarbeitung und asynchronen Botschaften sollten die Rückgabepfeile dagegen immer eingetragen werden.

In der Abb. 2.10-1 sendet der Akteur die Botschaft `op()`, die ein Objekt der Klasse `C1` erzeugt. Dieses Objekt sendet die Botschaft `op1()` an ein bereits existierendes Objekt der Klasse `C2`. Anschließend ruft die Operation `op()` die Operation `op2()` auf, die ebenfalls zur Klasse `C1` gehört. In diesem Fall werden die aktivierten Operationen »geschachtelt« an die Objektlinie angetragen. Die Operation `op2()` aktiviert nun ihrerseits die Operation `op3()`. Am Ende der Operation `op()` wird das zuvor erzeugte Objekt von `C1` wieder gelöscht.

In der Analyse werden Sequenzdiagramme verwendet, um Szenarios so präzise zu beschreiben, daß deren fachliche Korrektheit diskutiert werden kann und um eine geeignete Vorgabe für Entwurf und Implementierung zu erstellen. Es gibt daher keinen absoluten Maßstab für den jeweiligen Detaillierungsgrad, sondern er muß im Einzelfall auf die Zielgruppe abgestimmt werden. Im Gegensatz dazu werden Sequenzdiagramme im Entwurf für eine sehr detail-



lierte Spezifikation der Operationsaufrufe verwendet und enthalten dann alle beteiligten Operationen (siehe Kapitel 10).

Abb. 2.10-2 modelliert das Szenario 1 (Auftrag für einen Neukunden bearbeiten, wenn mindestens ein Artikel lieferbar ist) als Sequenzdiagramm. Auf die Darstellung der Auftragsposition wird zunächst verzichtet. Abb. 2.10-3 zeigt zum Vergleich die detailliertere Modellierung einschließlich der Klasse Auftragsposition.

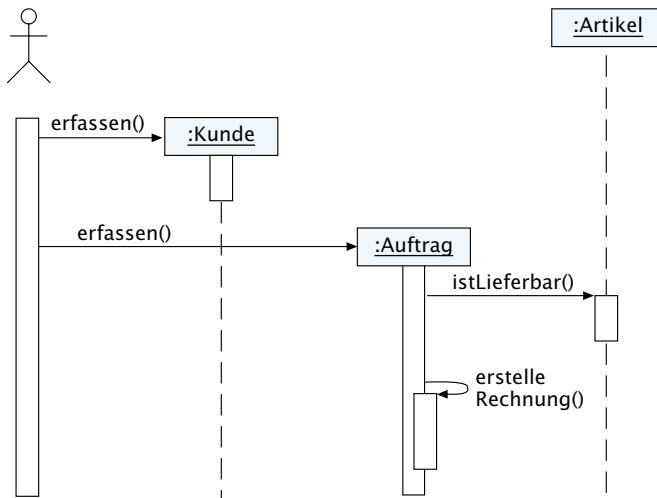
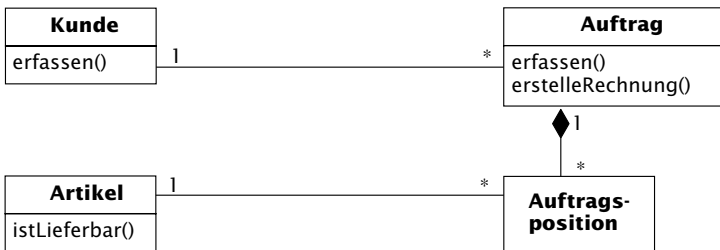


Abb. 2.10-2:  
Sequenzdiagramm  
Auftrag für einen  
Neukunden  
bearbeiten

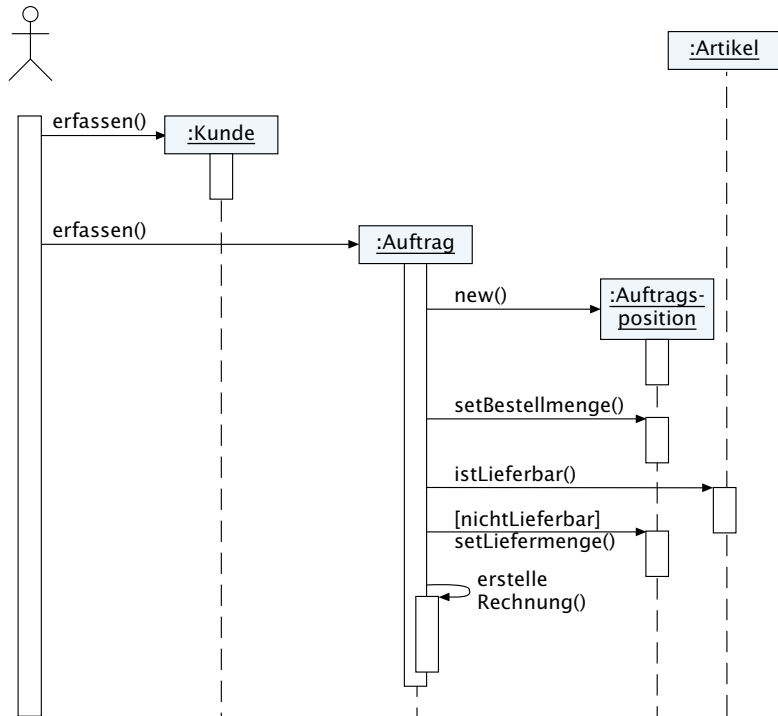


Die UML erlaubt die Angabe von Bedingungen und Wiederholungen im Sequenzdiagramm. Die **Bedingung** (*condition*) wird in eckigen Klammern angegeben, d.h. [Bedingung] Operation(). Die genannte Operation wird nur dann aufgerufen, wenn die Bedingung erfüllt ist. **Wiederholungen** (*iterations*) können durch \* Operation() oder \* [Bedingung] Operation() spezifiziert werden. Wenn keine Iterationen in ein Diagramm eingetragen werden, so bedeutet dies laut UML, daß die Anzahl der Wiederholungen un spezifiziert ist. Die Bedingung wird in der Analyse meistens umgangssprachlich formuliert.

Bedingungen  
Wiederholungen

## LE 4 2 Konzepte und Notation für OOA

Abb. 2.10-3:  
Detailliertes  
Sequenzdiagramm  
Auftrag für einen  
Neukunden  
bearbeiten



**Beispiel** In der Abb. 2.10-2 wird die Operation `setLiefermenge()` nur ausgeführt, wenn die Bedingung `nichtLieferbar` erfüllt ist. Auf die Spezifikation der Iteration habe ich hier verzichtet.

**Beispiel** Abb. 2.10-4 zeigt die beiden Szenarios 2 und 3 für die Bearbeitung von Aufträgen existierender Kunden. Wir können hier beide Szenarios in einem Sequenzdiagramm modellieren, indem das Konzept der Bedingung (*condition*) benutzt wird.

**Konsistenz** Ein Sequenzdiagramm muß mit dem Klassendiagramm konsistent sein, d.h. alle Botschaften, die an ein Objekt einer Klasse gesendet werden, müssen im Klassendiagramm in der Operationsliste dieser Klasse enthalten sein. Verwaltungsoperationen werden im Sequenzdiagramm eingetragen, um die Kommunikation der Objekte

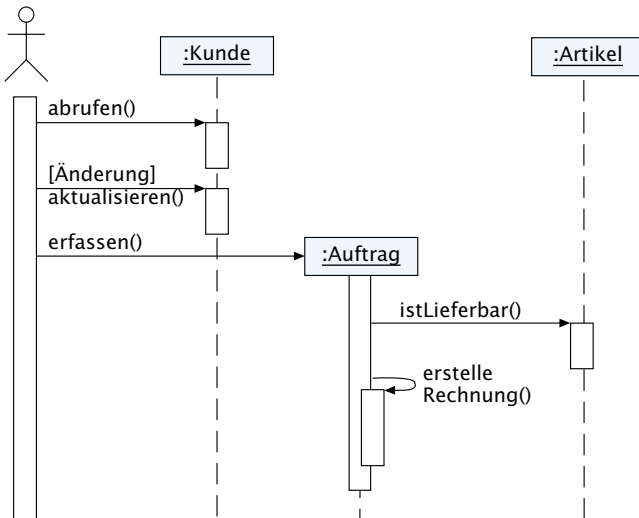


Abb. 2.10-4:  
Sequenzdiagramm  
Auftrag für  
existierende  
Kunden bearbeiten

vollständig zu beschreiben, während sie im Klassendiagramm nicht explizit modelliert werden.

In der Systemanalyse besitzen Klassen – der Einfachheit halber – die Eigenschaft der Objektverwaltung (siehe Kapitel 2.2), d.h. sie kennen alle ihre Objekte. Alle Operationen, die auf der Menge aller Objekte ausgeführt werden, sind daher als Klassenoperationen zu modellieren. Um eine Klassenoperation zu aktivieren, wird die Botschaft direkt an die Klasse geschickt. Es ist deshalb notwendig, die UML wie folgt zu erweitern: Anstelle eines Objekts wird das Klassensymbol in das Sequenzdiagramm eingetragen (Abb. 2.10-7).

Klassen im  
Sequenzdiagramm

Falls ein Werkzeug es nicht ermöglicht, Klassen in ein Sequenzdiagramm einzutragen, dann verwenden Sie in der Analyse eine Hilfsnotation, z.B. al l : Klasse, wobei »al l« für die Menge aller Objekte dieser Klasse steht.

Tip

Sequenzdiagramme wurden lange Zeit im Bereich der Telekommunikation benutzt /Jacobson 92/. Jacobson hat Sequenzdiagramme – von ihm als Interaktionsdiagramme bezeichnet – bereits 1987 für die objektorientierte Entwicklung eingeführt /Jacobson 95/. Die in der UML verwendete Form basiert auf dem *object message sequence chart* von Buschmann, Meunier, Rohnert, Sommerlad und Stal, der wiederum aus der *message sequence chart notation* hergeleitet wurde.

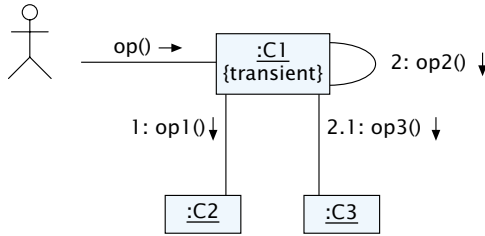
Historie

Ein **Kollaborationsdiagramm** (*collaboration diagram*) bildet eine Alternative zum Sequenzdiagramm. Das in der Abb. 2.10-5 dargestellte Kollaborationsdiagramm modelliert den gleichen Ablauf wie das Sequenzdiagramm der Abb. 2.10-1. Es beschreibt die Objekte und – zusätzlich zum Sequenzdiagramm – die Verbindungen zwischen diesen Objekten. An jede Verbindung (*link*) kann eine Bot-

Notation  
Kollaborations-  
diagramm

## LE 4 2 Konzepte und Notation für OOA

Abb. 2.10-5:  
Notation  
Kollaborations-  
diagramm



schaft in Form eines Pfeiles angetragen werden. In der Abb. 2.10-5 sendet der Akteur die Botschaft `op()`, die ein Objekt der Klasse `C1` erzeugt. Dieses Objekt aktiviert dann zuerst die Operation `op1()` und dann `op2()`. Diese Reihenfolge wird durch die Numerierung ausgedrückt. Die Operation `op2()` – mit der Nummer 2 – ruft nun ihrerseits die Operation `op3()` – mit der Nummer 2.1 – auf.

Objekte, die während der Ausführung neu erzeugt werden, sind mit `{new}`, Objekte, die während der Ausführung gelöscht werden, mit `{destroyed}` gekennzeichnet. Objekte, die während der Ausführung sowohl erzeugt als auch wieder gelöscht werden, sind `{transient}`. Analog dazu können Objektverbindungen, die im Laufe der Ausführung erstellt werden mit `{new}`, gelöschte *links* mit `{destroyed}` und Verbindungen, die innerhalb des Szenario sowohl auf- als auch abgebaut werden, mit `{transient}` beschriftet werden.

Hinweis

Im Englischen bedeutet *collaboration* Zusammenarbeit. Das *collaboration diagram* dokumentiert die Zusammenarbeit von Objekten. Im Deutschen bedeutet Kollaboration in erster Linie die aktive Unterstützung einer feindlichen Besatzungsmacht gegen die eigenen Landsleute. Das wollen wir den Objekten nun wirklich nicht unterstellen. Da in deutschen Publikationen jedoch konsequent der Begriff Kollaborationsdiagramm verwendet wird, schließe ich mich an.

Ein Kollaborationsdiagramm sieht dem Objektdiagramm – in dem Objekte und ihre Verbindungen (*links*) beschrieben werden – relativ ähnlich. Im Gegensatz zum Objektdiagramm modelliert es jedoch nicht einen Schnappschuß der Systemstruktur, sondern zeigt, wie Objekte für die Ausführung einer bestimmten Operation zusammenarbeiten.

**Beispiel** Im Kollaborationsdiagramm der Abb. 2.10-6 stellt jedes aufgeführte Objekt einen Platzhalter für ein beliebiges Objekt der Klasse dar. Dagegen modelliert das Objektdiagramm exemplarisch die Dortmunder und Bochumer Filiale einer Versicherung, zu denen die angegebenen Versicherten und deren Verträge existieren.

permanente und  
temporäre  
Verbindungen

Im Kollaborationsdiagramm unterscheiden wir permanente Verbindungen, die den Assoziationen entsprechen und temporäre Verbindungen, die nur für die Dauer der Kommunikation bestehen. Eine temporäre Verbindung liegt vor, wenn das angesprochene

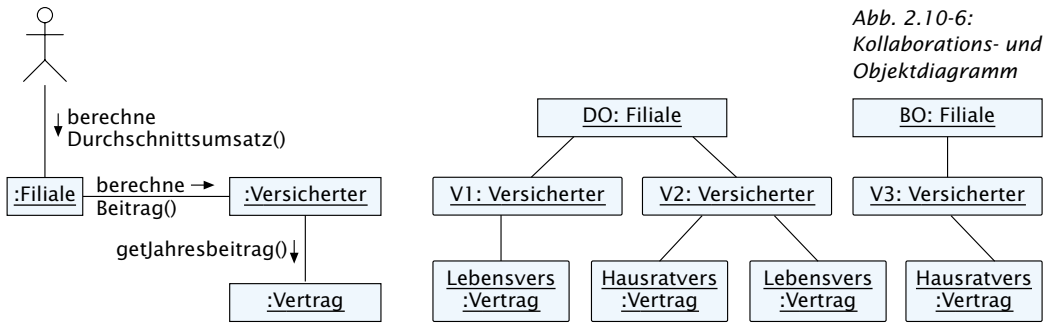


Abb. 2.10-6:  
Kollaborations- und  
Objektdiagramm

Empfängerobjekt auch ohne Vorliegen einer Assoziation vom Sender eindeutig identifiziert werden kann. Die Auswahl eines Objekts durch den Benutzer könnte z.B. dadurch geschehen, daß alle Objekte in einer Auswahlliste (*list box*) aufgeführt werden und der Benutzer das entsprechende Objekt auswählt. In der Systemanalyse ist der verwendete Mechanismus jedoch nicht von Bedeutung. In einem Kollaborationsdiagramm können Objekte nur dann kommunizieren, wenn zwischen ihnen eine Verbindung eingetragen ist. Wir kennzeichnen diese temporären Beziehungen mit dem Stereotypen «temp», um sie von den Assoziationen zu unterscheiden. Jedes Objekt kann jederzeit Botschaften an sich selbst schicken (*self link*).

Analog zum Sequenzdiagramm müssen auch hier Klassenoperationen aktiviert werden können. Anstelle eines Objekts wird dann das Klassensymbol in das Diagramm eingetragen (siehe /Larman 98/).

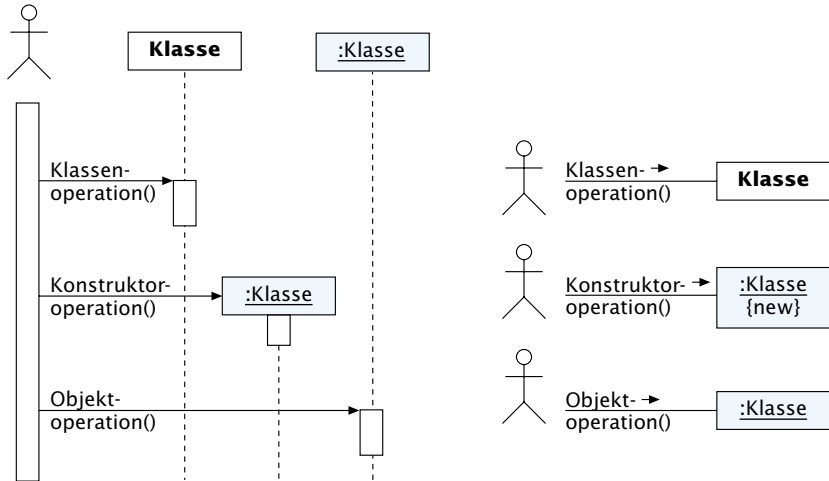
Klassen im  
Kollaborations-  
diagramm

**Sequenzdiagramme** heben den zeitlichen Aspekt des dynamischen Verhaltens hervor. Die Reihenfolge und die Verschachtelung der Operationen ist sehr leicht zu erkennen. Sequenzdiagramme werden für die Modellierung komplexer Szenarios empfohlen, weil hier die Reihenfolge der Botschaften transparent ist. In ein Sequenzdiagramm können mehrere externe Operationen, die von einem Akteur nacheinander aktiviert werden, eingetragen werden. **Kollaborationsdiagramme** betonen die Verbindungen (*links*) zwischen Objekten. Die Reihenfolge und die Verschachtelung der Operationen werden durch eine hierarchische Numerierung angegeben. Nachteilig ist, daß dadurch die Reihenfolge weniger deutlich sichtbar ist. Der Vorteil für den Analytiker ist jedoch, daß er sich beim Erstellen des Diagramms nicht gleich auf die Ausführungsreihenfolge festlegen muß, sondern zunächst die Objekte und ihre Kommunikation beschreiben und in einem weiteren Schritt die Reihenfolge hinzufügen kann. Kollaborationsdiagramme eignen sich sehr gut, um die Wirkung komplexer Operationen zu beschreiben. Im Gegensatz zu Sequenzdiagrammen ist für jede externe Operation ein separates Kollaborationsdiagramm zu erstellen.

Vergleich

## LE 4 2 Konzepte und Notation für OOA

Abb. 2.10-7:  
Sequenzdiagramm  
vs. Kollaborations-  
diagramm (UML-  
Erweiterung)



Prinzipiell können beide Diagrammarten für die Beschreibung von Szenarios verwendet werden. Abb. 2.10-7 zeigt beide Interaktionsdiagramme im Vergleich und die Darstellung der verschiedenen Operationsarten.

verwandte Begriffe

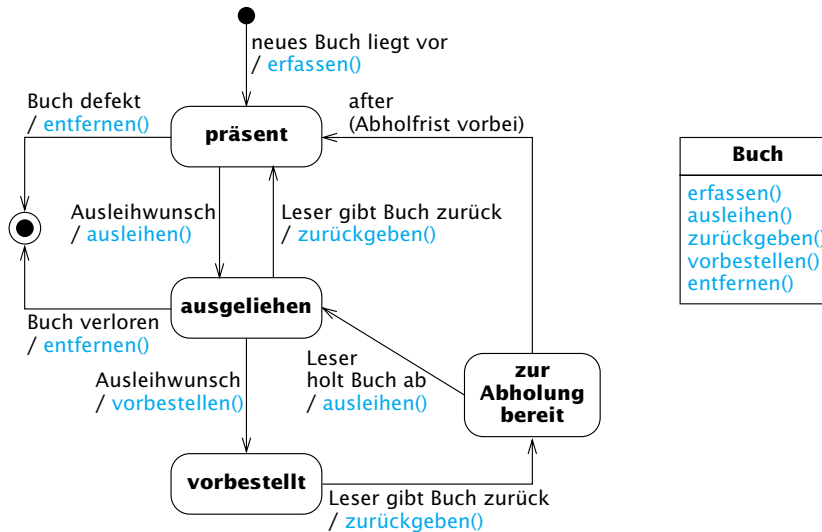
Viele objektorientierte Methoden verwenden den Begriff Interaktionsdiagramm anstelle von Sequenzdiagramm.

## 2.11 Zustandsautomat

**Definition** Ein **Zustandsautomat** (*finite state machine*) besteht aus Zuständen und Zustandsübergängen (Transitionen). Ein Zustand ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet, d.h. das Objekt verweilt eine bestimmte Zeit in diesem Zustand. In diesen Zustand gelangt das Objekt durch ein entsprechendes Ereignis. Ein Ereignis tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer. Ein Objekt kann – nacheinander – mehrere Zustände durchlaufen. Zu einem Zeitpunkt befindet es sich in genau einem Zustand. Tritt in einem beliebigen Zustand ein Ereignis ein, so hängt der nächste Zustand sowohl vom aktuellen Zustand als auch vom jeweiligen Ereignis ab. Der Zustand eines Objekts beinhaltet also implizit Informationen, die sich aus den bisherigen Eingaben ergeben haben.

In der Objektorientierung wird das **Zustandsdiagramm** (*state-chart diagram*) zur graphischen Darstellung des Zustandsautomaten verwendet.

**Beispiel** Wenn in einer Bibliothek ein Buch beschafft wird, dann werden seine Daten erfasst und ein neues Objekt der Klasse Buch erzeugt (Abb. 2.11-1). Der Einfachheit halber gebe es von jedem Buch nur ein einziges Exemplar. Jedes Buch kann ausgeliehen werden. Wird ein aus-

Abb. 2.11-1:  
Zustandsautomat  
der Klasse Buch

geliehenes Buch von einem anderen Leser gewünscht, dann muß es vorbestellt werden. Nicht vorbestellte Bücher stehen nach der Rückgabe sofort für eine erneute Ausleihe bereit. Vorbestellte Bücher werden nach der Ausleihe für den entsprechenden Leser zur Abholung bereitgelegt und der Leser wird informiert. Wird das Buch nicht fristgemäß abgeholt, dann steht es für eine neue Ausleihe bereit. Defekte Bücher oder Bücher, die nicht zurückgegeben wurden, werden aus dem Bestand entfernt.

Wenn ein neues Buch im System gespeichert wird, dann befindet es sich zunächst im Zustand präsent. Das Löschen von Büchern im System wird durch den Übergang in den Endzustand (Bullauge) angezeigt. Er sagt aus, daß das Objekt aufhört, zu existieren.

Der Name des **Zustands** ist optional. Zustände ohne Namen heißen *anonyme Zustände* und sind alle voneinander verschieden. Ein benannter Zustand kann dagegen – der besseren Lesbarkeit halber – mehrmals in das Diagramm eingetragen werden. Diese Zustände sind alle identisch. Der Zustandsname soll kein Verb sein, auch wenn mit dem Zustand eine Verarbeitung verbunden sein kann. Wir wählen beispielsweise den Namen »ausgeliehen« statt »ausleihen«. Innerhalb eines Zustandsautomaten muß jeder Zustandsname eindeutig sein.

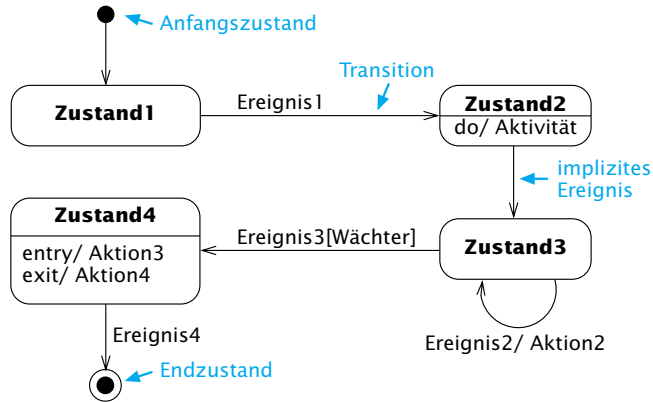
Mit einem Zustand können **Aktionen** oder Aktivitäten verbunden sein (Abb. 2.11-2). Eine **entry-Aktion** ist atomar. Sie wird beim Eintritt in den Zustand – unabhängig davon durch welche Transition der Eintritt in diesen Zustand erfolgt – immer ausgeführt und terminiert selbständig. Eine **exit-Aktion** ist ebenfalls atomar. Sie wird immer ausgeführt, wenn der entsprechende Zustand – durch eine beliebige Transition – verlassen wird. Eine **Aktivität** beginnt, wenn

Zustandsname

Aktionen/  
Aktivitäten

## LE 4 2 Konzepte und Notation für OOA

Abb. 2.11-2:  
Notation des  
Zustands-  
diagramms



das Objekt den Zustand einnimmt und endet, wenn es ihn verläßt. Sie kann alternativ durch ein Paar von Aktionen, eine zum Starten und eine zum Beenden der Aktivität, beschrieben oder durch ein weiteres Zustandsdiagramm verfeinert werden. Zusätzlich können weitere interne Aktionen angegeben werden, die durch bestimmte Ereignisse aktiviert werden, z.B. `hel p/ zei geHi l fe`. Welche Arten von Ereignissen auftreten können, wird unten im Rahmen der Transition näher erläutert.

Jeder Zustandsautomat besitzt einen Anfangszustand und kann einen Endzustand besitzen.

Anfangszustand

Der Anfangszustand (*initial state*) wird durch einen kleinen schwarzen Kreis dargestellt (Abb. 2.11-2). Es handelt sich um einen Pseudozustand, der mit einem »echten« Zustand durch eine Transition verbunden ist. Diese Transition kann mit einem Ereignis – zum Erzeugen des Objekts – beschriftet sein. Der Anfangszustand ist ein grafisches Hilfsmittel; ein Objekt kann sich nicht in diesem Zustand befinden, sondern ein neu erzeugtes Objekt befindet sich zunächst in dessen »echten« Folgezustand.

**Beispiel** In der Abb. 2.11-1 befindet sich ein neu erzeugtes Buch-Objekt im Zustand präsent.

Endzustand

Im Endzustand (*final state*) hört ein Objekt auf zu existieren. Aus diesem Zustand führen keine Transitionen heraus. Der Endzustand, der ebenfalls ein Pseudozustand (Abb. 2.11-2) ist, wird durch ein »Bullauge« dargestellt und kann optional beschriftet sein.

**Beispiel** In der Abb. 2.11-1 wird ein Buch-Objekt durch die Operation `entfernen()` gelöscht, was durch den Zustandsübergang in den Endzustand ausgedrückt wird.

Transition

Eine **Transition** bzw. ein Zustandsübergang verbindet zwei Zustände. Die Transition wird durch einen Pfeil dargestellt. Eine Transition kann nicht unterbrochen werden und wird stets durch



ein Ereignis ausgelöst. Wir sagen: die Transition »feuert«. Tritt ein Ereignis ein und das Objekt befindet sich nicht in einem Zustand, in dem es darauf reagieren kann, dann wird das Ereignis ignoriert. Meistens ist mit einer Transition ein Zustandswechsel verbunden. Es ist aber auch möglich, daß Ausgangs- und Folgezustand identisch sind. Beachten Sie, daß in einem solchen Fall die *entry*- und *exit*-Aktionen bei jedem neuen Eintritt – in denselben Zustand – ausgeführt werden.

Ein **Ereignis** kann sein:

- eine Bedingung, die wahr wird,
- ein Signal,
- eine Botschaft (Aufruf einer Operation),
- eine verstrichene Zeit (*elapsed time event*) oder
- das Eintreten eines bestimmten Zeitpunkts.

Ereignisse

Wir sprechen in den beiden letzten Fällen von zeitlichen Ereignissen.

Eine Bedingung ist beispielsweise: `when(Temperatur > 100 Grad)`. Die zugehörige Transition feuert, wann immer diese Bedingung wahr wird. Signale und Botschaften sind durch die Notation nicht unterscheidbar. Sie werden durch einen Namen beschrieben und können Parameter besitzen, z.B. `rechte Maustaste gedrückt (Mausposition)`. Die Transition feuert, wenn das Signal oder die Botschaft gesendet wird. Eine verstrichene Zeitspanne ist beispielsweise: `after (10 Sekunden)`. Die Transition feuert, wenn die angegebene Zeitspanne nach einem definierten Zeitpunkt (z.B. Eintritt in den Ausgangszustand der Transition) verstrichen ist. Ein Zeitpunkt wird beispielsweise durch `when (Datum = 1. 1. 2000)` beschrieben.

Notation

Es ist möglich, ein Ereignis mit einem Wächter (*guard condition*) zu kombinieren. Der Wächter ist eine Bedingung, die sich jedoch von der oben beschriebenen Bedingung unterscheidet. Wenn das zugehörige Ereignis eintritt, dann wird der Wächter ausgewertet. Ist die dort spezifizierte Bedingung erfüllt, dann feuert die Transition (*guarded transition*). Ein Beispiel finden Sie in der Abb. 2.11-4.

Wächter

Jeder Zustand darf eine – ausgehende – Transition ohne explizite Angabe eines Ereignisses besitzen. Diese Transition wird ausgeführt, wenn die mit dem Zustand verbundene Verarbeitung beendet ist. Wir sprechen in diesem Fall von einem impliziten Ereignis.

implizites  
Ereignis

Mit der Transaktion kann eine Aktion verbunden sein. Sie wird ausgeführt, wenn die Transition feuert. Es handelt sich um eine atomare Operation, d.h. sie kann nicht unterbrochen werden.

Aktion

Die meisten objektorientierten Methoden verwenden den Zustandsautomaten, um für eine bestimmte Klasse den Lebenszyklus (*life cycle*) ihrer Objekte zu beschreiben. Alle Objekte einer Klasse besitzen denselben Zustandsautomaten. Jede Klasse besitzt einen Lebenszyklus. In den meisten Fällen handelt es sich jedoch um einen trivialen Lebenszyklus, der nicht spezifiziert werden muß. Zu-

Lebenszyklus

## LE 4 2 Konzepte und Notation für OOA

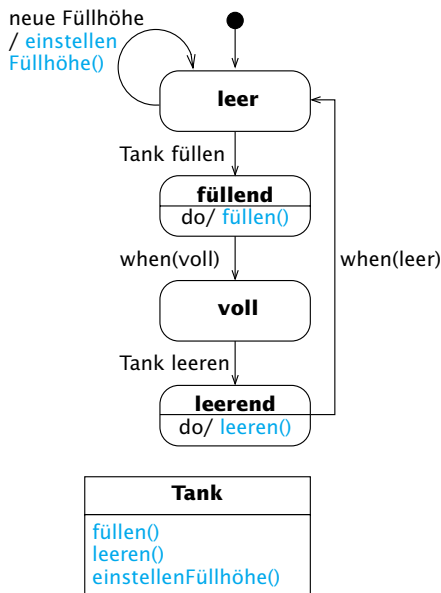
standsautomaten können sowohl einen kontinuierlichen Lebenszyklus (*circular lifecycle*) beschreiben als auch einen Endzustand (*born-and-die lifecycle*) besitzen.

Konsistenz In der Abb. 2.11-1 modelliert der Zustandsautomat den Lebenszyklus der Klasse Buch. In diesem Fall müssen Zustands- und Klassendiagramm konsistent sein. Wir verwenden folgende Konsistenzregeln, die über die Notation der UML hinausgehen:

- Als Aktionen und Aktivitäten sind nur Operationen der jeweiligen Klasse zulässig.
- Operationsnamen werden in der Form `Operation()` eingetragen.
- Wenn eine Operation in mehreren Zuständen aktiviert werden kann, so kann sie in Abhängigkeit vom jeweiligen Zustand eine unterschiedliche Wirkung besitzen.
- Erhält ein Objekt in einem Zustand einen Operationsaufruf, wobei diese Operation weder als Aktivität noch als Aktion zur Verfügung steht, dann besitzt die Botschaft keine Wirkung, d.h. »das Objekt tut nichts«.

Beispiel Lebenszyklus Abb. 2.11-3 zeigt den zyklischen Zustandsautomaten eines Tanks. Im Zustand `leer` kann die Operation `einstellenFüllhöhe()` aktiviert werden. Weiterhin kann bei leerem Tank die Operation `füllen()` gestartet werden. Der Zustand `füllend` wird verlassen, wenn die Bedingung `when(voll)` wahr wird. Die Operation `füllen()` wird als Aktivität eingetragen, weil sie auch durch die Aktionen *starte füllen* und *terminiere füllen* beschrieben werden könnte. Analog wird bei der Operation `leeren()` verfahren.

Abb. 2.11-3:  
Zustandsautomat  
eines Tanks



Auch die Wirkungsweise komplexer Operationen kann mit einem Zustandsautomaten beschrieben werden. In der Analyse ist dies beispielsweise dann sinnvoll, wenn die Operation in Abhängigkeit vom Benutzerverhalten unterschiedliche Schritte ausführt.

komplexe Operationen

Abb. 2.11-4 beschreibt, wie die Parkgebühr an einem Kartenautomat in einem Parkhaus bezahlt wird. Tritt im Zustand *wartet auf Geld* das Ereignis *Geld eingeworfen* ein, so hängt die Reaktion von dem angegebenen Wächter, d.h. der jeweiligen Bedingung [*reicht nicht*] oder [*reicht aus*] ab. Im Zustand *wartet auf Quittung* kann während maximal fünf Sekunden nach Eintritt in diesen Zustand eine Quittung angefordert werden. Nach 5 Sekunden erfolgt eine Transition in den Zustand *bereit*. Der Zustand mit der Aktion *prüfe Karte* wurde als *anonymer Zustand* modelliert, weil ein Zustandsname (z.B. *prüfend*) in diesem Fall keine zusätzliche Information enthielte.

Beispiel

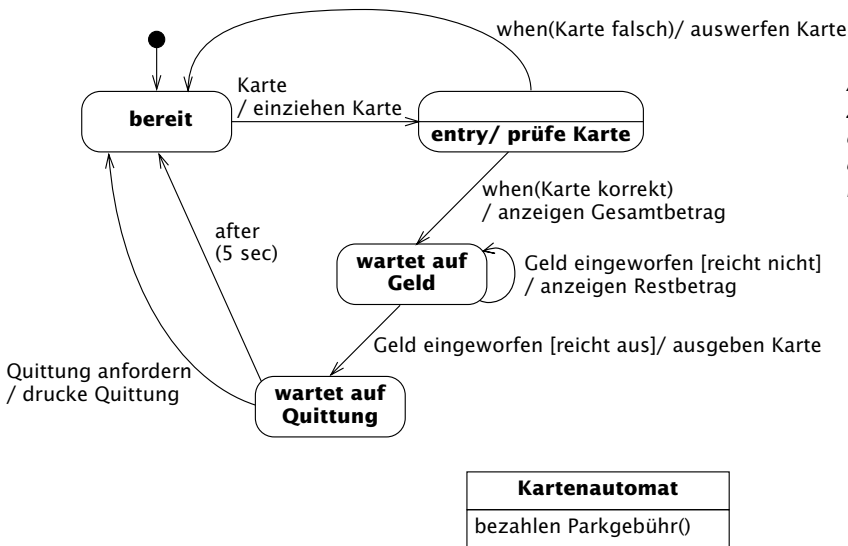


Abb. 2.11-4: Zustandsautomat eines Kartenautomaten im Parkhaus



Eine Klasse vererbt ihren Zustandsautomaten an ihre Unterklassen. Unterklassen können darüber hinaus eigene Zustandsautomaten besitzen. Um Konflikte zu vermeiden, sind einige Restriktionen zu beachten.

Zustandsautomat und Vererbung

### Verfeinerung von Zuständen

Ein Zustand kann durch Unterzustände (*substates*) verfeinert werden. Alle Unterzustände sind disjunkt, d.h. sie schließen sich gegenseitig aus. Ein Zustand, der verfeinert wird, heißt auch zusammengesetzter Zustand.

Auch jede Verfeinerung besitzt genau einen Anfangszustand. Eine Transition in einen verfeinerten Zustand entspricht der

## LE 4 2 Konzepte und Notation für OOA

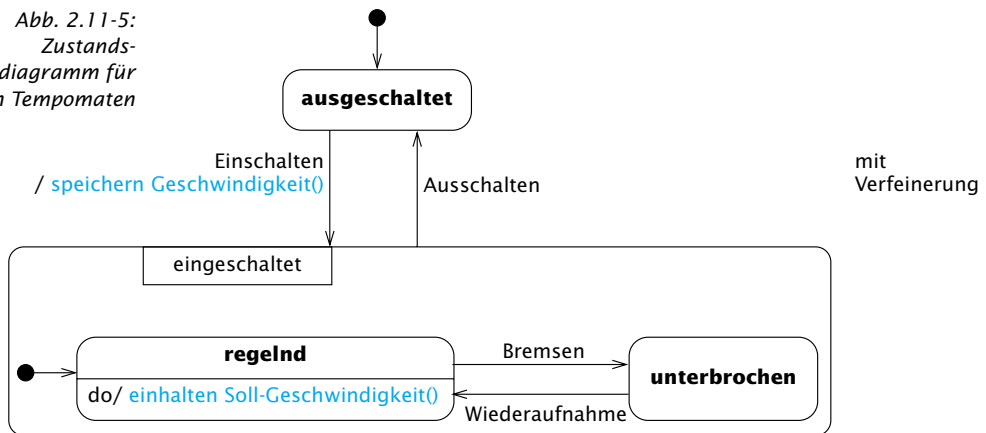
Transition in den Anfangszustand der Verfeinerung. Das Verlassen eines verfeinerten Zustands wird im entsprechenden Zustandsdiagramm durch den Endzustand angezeigt.

Wird ein verfeinerter Zustand durch eine Transition verlassen, dann wird jeder Unterzustand – egal auf welcher Verfeinerungsstufe – verlassen und die entsprechenden *exit*-Aktion ausgeführt. Wird ein Zustand mit einer rekursiven Transition verfeinert, dann wird beim erneuten Zustandseintritt der Anfangszustand eingenommen und die *entry*-Aktion ausgeführt.

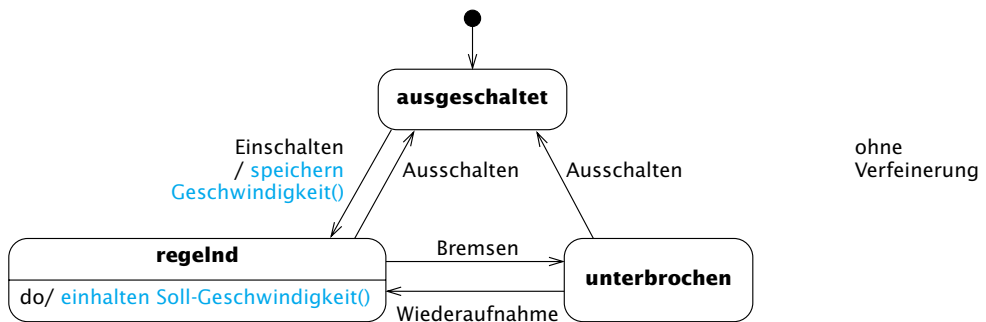
**Beispiel** Abb. 2.11-5 wendet die Verfeinerung auf einem Tempomaten an. Beide Zustandsdiagramme modellieren die gleiche Problemstellung.

Aktivitätsdiagramm Als Sonderfall des Zustandsdiagramms bietet die UML das **Aktivitätsdiagramm** (*activity diagram*) an. Bei diesem Diagramm

Abb. 2.11-5:  
Zustandsdiagramm für  
einen Tempomaten



mit  
Verfeinerung



ohne  
Verfeinerung

Tempomat
speichern Geschwindigkeit() einhalten Soll-Geschwindigkeit()

sind alle – oder zumindest die meisten – Zustände mit einer Verarbeitung verknüpft (*action states*). Ein Zustand wird verlassen, wenn die mit ihm verbundene Verarbeitung beendet ist. Weder in einem Zustand noch bei einer Transition sollten explizite Ereignisse vorkommen. Transitionen können jedoch Wächter (*guard conditions*) beinhalten, mit denen die Verzweigung des Kontrollflusses beschrieben wird.

Abb. 2.11-6 zeigt die Notation eines Aktivitätsdiagramms. Jeder Zustand modelliert einen Schritt innerhalb der Gesamtverarbeitung. Wenn die Verarbeitung1 beendet ist, wird entsprechend [Bedingung1a] oder [Bedingung1b] verzweigt. Eine Entscheidung kann auch mittels einer Raute dargestellt werden. Außerdem kann spezifiziert werden, ob die Verarbeitungsschritte grundsätzlich parallel durchgeführt werden können oder ob eine sequentielle Bearbeitung notwendig ist. Nach dem »Splitting« können die Verarbeitungsschritte 2 und 3 parallel zur Verarbeitung4 ausgeführt werden. Erst wenn alle Verarbeitungsschritte ausgeführt sind, kann nach der »Synchronisation« mit der Verarbeitung5 fortgefahren werden. Überall dort, wo eine parallele Ausführung beschrieben ist, kann prinzipiell auch eine sequentielle Bearbeitung stattfinden, wobei die Reihenfolge jedoch explizit offen bleibt.

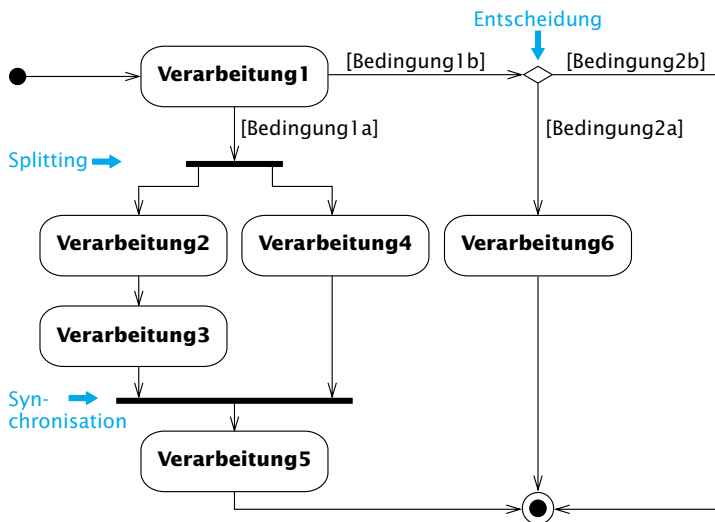


Abb. 2.11-6:  
Notation des  
Aktivitäts-  
diagramms

Im Gegensatz zum »herkömmlichen« Zustandsdiagramm beschreibt das Aktivitätsdiagramm nicht die Reaktion auf Ereignisse, sondern spezifiziert die interne Verarbeitung. Es kann daher sowohl zur Beschreibung von Geschäftsprozessen als auch zur Spezifikation komplexer Operationen eingesetzt werden.

Da der Zustandsautomat häufig für die Modellierung des Lebenszyklus verwendet wird, sprechen einige Autoren vom Objekt-

verwandte Begriffe

Lebenszyklus (*object life cycle*). Für die Darstellung des Zustandsautomaten ist bei den objektorientierten Methoden das Diagramm üblich – obwohl er auch als Tabelle oder Matrix modelliert werden kann. Daher wird auch vom Zustandsdiagramm (*object state diagram*) gesprochen.

**Akteur (*actor*)** Ein Akteur ist eine Rolle, die ein Benutzer des Systems spielt. Akteure befinden sich außerhalb des Systems. Akteure können Personen oder externe Systeme sein.

**Aktion (*action*)** Eine Aktion ist eine atomare Operation, die durch ein Ereignis ausgelöst wird und sich selbst beendet. Sie kann mit einer →Transition verbunden sein. *Entry*-Aktionen werden bei Eintritt und *exit*-Aktionen bei Verlassen des →Zustandes ausgeführt.

**Aktivität (*activity*)** Eine Aktivität ist eine Operation, die mit einem →Zustand eines →Zustandsautomaten verbunden ist. Sie beginnt bei Eintritt und endet bei Verlassen des Zustandes. Sie kann alternativ durch ein Paar von Aktionen, eine zum Starten und eine zum Beenden der Aktivität, beschrieben oder durch ein weiteres →Zustandsdiagramm verfeinert werden.

**Aktivitätsdiagramm (*activity diagram*)** Ein Aktivitätsdiagramm ist der Sonderfall eines →Zustandsdiagramms, bei dem – fast – alle Zustände mit einer Verarbeitung verbunden sind. Ein →Zustand wird verlassen, wenn die Verarbeitung beendet ist. Außerdem ist es möglich, eine Verzweigung des Kontrollflusses zu spezifizieren und zu beschreiben, ob die Verarbeitungsschritte in festgelegter oder beliebiger Reihenfolge ausgeführt werden können.

**Botschaft (*message*)** Eine Botschaft ist die Aufforderung eines Senders (*client*) an einen Empfänger (*server*, *supplier*) eine Dienstleistung zu erbringen. Der Empfänger interpretiert diese Botschaft und führt eine Operation aus.

**Ereignis (*event*)** Ein Ereignis tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer. Es kann sein: eine wahr werdende Bedingung, ein Signal, eine Botschaft (Aufruf einer Operation), eine verstrichene Zeitspanne oder das Eintreten eines bestimmten Zeitpunkts.

In den beiden letzten Fällen spricht man von zeitlichen Ereignissen.

**Geschäftsprozeß (*use case*)** Ein Geschäftsprozeß (*use case*) besteht aus mehreren zusammenhängenden Aufgaben, die von einem →Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.

**Geschäftsprozeßdiagramm (*use case diagram*)** Ein Geschäftsprozeßdiagramm beschreibt die Beziehungen zwischen →Akteuren und →Geschäftsprozessen in einem System. Auch Beziehungen zwischen Geschäftsprozessen (*extends* und *uses*) können eingetragen werden. Es gibt auf einem auf hohem Abstraktionsniveau einen guten Überblick über das System und seine Schnittstellen zur Umgebung.

**Geschäftsprozeßschablone (*use case template*)** Die Geschäftsprozeßschablone ermöglicht eine semiformale Spezifikation von →Geschäftsprozessen. Sie enthält folgende Informationen: Name, Ziel, Kategorie, Vorbedingung, Nachbedingung Erfolg, Nachbedingung Fehlschlag, Akteure, auslösendes Ereignis, Beschreibung des Standardfalls sowie Erweiterungen und Alternativen zum Standardfall.

**Interaktionsdiagramm (*interaction diagram*)** In der UML ist Interaktionsdiagramm der Oberbegriff von →Sequenz- und →Kollaborationsdiagramm. Bei anderen Methoden wird der Begriff Interaktionsdiagramm für das Sequenzdiagramm verwendet.

**Kollaborationsdiagramm (*collaboration diagram*)** Ein Kollaborationsdiagramm beschreibt die Objekte und die Verbindungen zwischen diesen Objekten. An jede Verbindung (*link*) kann eine →Botschaft in Form eines Pfeiles angetragen werden. Die Reihenfolge und Verschachtelung der Operationen wird durch eine hierarchische Numerierung angegeben.



**Nachbedingung (postcondition)** Die Nachbedingung beschreibt die Änderung, die durch eine Verarbeitung bewirkt wird, unter der Voraussetzung, daß vor ihrer Ausführung die Vorbedingung erfüllt war.

**Nachricht (message)** →Botschaft

**Sequenzdiagramm (sequence diagram)** Ein Sequenzdiagramm besitzt zwei Dimensionen. Die Vertikale repräsentiert die Zeit und auf der Horizontalen werden die Objekte angetragen. In das Diagramm werden die →Botschaften eintragen, die zum Aktivieren der Operationen dienen.

**Szenario (scenario)** Ein Szenario ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind. Diese Schritte sollen das Hauptziel des →Akteurs realisieren und ein entsprechendes Ergebnis liefern. Ein →Geschäftsprozeß wird durch eine Kollektion von Szenarios dokumentiert.

**Transition (transition)** Eine Transition (Zustandsübergang) verbindet ei-

nen Ausgangs- und einen Folgezustand. Sie kann nicht unterbrochen werden und wird stets durch ein →Ereignis ausgelöst. Ausgangs- und Folgezustand können identisch sein.

**Vorbedingung (precondition)** Die Vorbedingung beschreibt, welche Bedingungen vor dem Ausführen einer Verarbeitung erfüllt sein müssen, damit die Verarbeitung definiert ausgeführt werden kann.

**Zustand (state)** Ein Zustand eines →Zustandsautomaten ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet. Ein Zustand besteht solange, bis ein →Ereignis eintritt, das eine →Transition auslöst.

**Zustandsautomat (finite state machine)** Ein Zustandsautomat besteht aus →Zuständen und →Transitionen. Er hat einen Anfangszustand und kann einen Endzustand besitzen.

**Zustandsdiagramm (statechart diagram)** Das Zustandsdiagramm ist eine grafische Repräsentation des →Zustandsautomaten.



Bei der Erstellung des dynamischen Modells sind folgende Konzepte anzuwenden:

Die **Geschäftsprozesse** beschreiben die Arbeitsabläufe der **Akteure** mit dem System auf einer hohen Abstraktionsebene. Die Dokumentation erfolgt in **Geschäftsprozeßdiagrammen**. Zur Spezifikation einzelner Geschäftsprozesse kann eine Schablone sinnvoll eingesetzt werden. Eine **Botschaft** ist die Aufforderung eines Senders an einen Empfänger, eine bestimmte Dienstleistung zu erbringen. Ein Geschäftsprozeß wird durch mehrere **Szenarios** verfeinert, die in Form von **Sequenzdiagrammen** oder **Kollaborationsdiagrammen** dokumentiert werden. Das **Zustandsdiagramm** verwenden wir, um den Lebenszyklus einer Klasse zu beschreiben oder die Wirkung einer komplexen Operation zu spezifizieren. Das **Aktivitätsdiagramm** ist ein Sonderfall des Zustandsdiagramm. Es kann zur Beschreibung von Geschäftsprozessen und von Operationen verwendet werden.



**1 Lernziel:** Geschäftsprozesse identifizieren und ein Geschäftsprozeßdiagramm erstellen.

Für eine Stadtbibliothek soll ein Softwaresystem entwickelt werden. Analysieren Sie die typischen Geschäftsprozesse zur Ausleihe und Verwaltung von Büchern und erstellen Sie ein Geschäftsprozeßdiagramm.

Aufgabe  
5–10 Minuten

## LE 4 Aufgaben

- Aufgabe 2 *Lernziel: Geschäftsprozesse spezifizieren.*  
5–10 Minuten Für die nachfolgende Problembeschreibung ist der Geschäftsprozeß mittels Schablone zu spezifizieren.  
Für eine Seminarverwaltung ist eine Anmeldung zu bearbeiten. Ist es ein neuer Kunde, dann sind die Daten zu erfassen. Existiert der Kunde bereits, dann ist zu prüfen, ob die Daten aktualisiert werden müssen. Weiterhin ist zu prüfen, ob der Kunde bereits angemeldet ist, ob das gewünschte Seminar angeboten wird und ob noch ein Platz im Seminar frei ist. Wenn die Anmeldung durchgeführt werden kann, erhält der Kunde eine Anmeldebestätigung. Wenn kein Platz mehr frei ist oder das angegebene Seminar nicht angeboten wird, dann muß beim Kunden nachgefragt werden, ob ein alternatives Seminar in Frage kommt.
- Aufgabe 3 *Lernziel: Szenario mittels Interaktionsdiagramm beschreiben.*  
10–15 Minuten Beschreiben Sie folgende Problemstellung als Sequenz- oder als Kollaborationsdiagramm. Skizzieren Sie das Klassendiagramm und achten Sie auf Konsistenz zwischen Klassen- und Interaktionsdiagramm.  
Ein neuer Kunde eröffnet bei einer Bank ein Sparkonto. Zuerst werden die Daten dieses Kunden erfaßt. Bei der Kontoeröffnung muß der Kunde gleich eine Einzahlung vornehmen, d.h. es findet die erste Kontobewegung für dieses Konto statt.
- Aufgabe 4 *Lernziel: Zustandsdiagramm erstellen.*  
10–15 Minuten Anhand der folgenden Problembeschreibung ist ein Zustandsdiagramm zu erstellen, um das Abheben von Geld an einem – vereinfachten – Geldautomaten zu beschreiben.  
Zu Beginn ist der Automat im Zustand »bereit«. Wird eine Karte eingegeben, so wird die Karte eingezogen. Falsche Karten werden sofort ausgeworfen und der Geldautomat ist wieder bereit. Ist die Karte korrekt, dann wartet der Automat auf die Eingabe der Geheimzahl. Wird eine falsche Geheimzahl eingegeben, dann bricht der Automat die Verarbeitung ab (d.h. Auswerfen der Karte) und ist wieder bereit. Bei korrekter Geheimzahl wartet der Automat auf die Eingabe des Betrags. Ist der gewünschte Betrag zu hoch, kann er erneut eingegeben werden. Weist das Konto die notwendige Deckung auf, dann wird die Karte aktualisiert ausgegeben und anschließend der Kundenauftrag bearbeitet. Anschließend ist der Automat bereit für den nächsten Kunden. Solange der Automat den Auftrag noch nicht bearbeitet, kann jederzeit die Rückgabetaste gedrückt werden. Der Automat wirft die Karte aus und ist wieder bereit.



## 3 Analysemuster und Beispielanwendungen



- Wichtige Muster der Systemanalyse kennen.
- Erklären können, was ein Muster ist.
- Klassendiagramme und Spezifikationen von Geschäftsprozessen lesen und verstehen können.
- Analysemuster in einer Textbeschreibung erkennen und darstellen können.
- Analysemuster in einem Klassendiagramm erkennen können.

wissen  
verstehen

anwenden



Voraussetzungen für diese Lehreinheit sind die Kapitel 2.1 bis 2.6 und das Kapitel 2.8.

- i 3.1 Katalog von Analysemustern 90
- 3.2 Beispiel Materialwirtschaft 98
- 3.3 Beispiel Arztregister 103
- 3.4 Beispiel Friseursalonverwaltung 108
- 3.5 Beispiel Seminarorganisation 112

### 3.1 Katalog von Analysemustern

Es hat sich gezeigt, daß bei der Modellierung häufig ähnliche Probleme vorkommen. Muster beschreiben – wiederkehrende – Problemstellungen und ihre Lösungen. Im Sinne einer effizienten Softwareentwicklung ist es sinnvoll, bereits existierende Problemlösungen wiederzuverwenden. Erfahrene Softwareentwickler wenden Muster meist mehr oder weniger intuitiv an. Einige Autoren systematisieren und katalogisieren diese Muster, um sie allen Softwareentwicklern für eine systematische und effektive Softwareproduktion zur Verfügung zu stellen. Dabei handelt es sich jedoch überwiegend um Entwurfsmuster (z.B. /Gamma et al. 95/, /Buschmann et al. 96/). Im Bereich der Analysemuster haben sich insbesondere Coad /Coad 92, 95/ und Fowler /Fowler 97a/ hervorgetan.

**Definition** Ganz allgemein gesehen ist ein **Muster** (*pattern*) eine Idee, die sich in einem praktischen Kontext als nützlich erwiesen hat und es wahrscheinlich auch in anderen sein wird /Fowler 97a/. Ein **Anlysemuster** ist eine Gruppe von Klassen mit feststehenden Verantwortlichkeiten und Interaktionen /Coad 95/. Es kann eine Gruppe von Klassen sein, die durch Beziehungen verknüpft ist, oder eine Gruppe von kommunizierenden Objekten.

Muster gestatten Softwareentwicklern eine effektive Kommunikation. Eine der wichtigen Verwendungsmöglichkeiten der Muster ist die standardisierte Lösung bestimmter Probleme. Sie lassen sich aber auch sehr gut zur Modellierung einsetzen, z.B. zum Unterscheiden der verschiedenen Arten der Assoziation und zum Identifizieren von Klassen.

Wie gut vorhandene Muster in neuen Projekten anwendbar sind, hängt sehr stark vom Anwendungsbereich ab. Prinzipiell lassen sich allgemeine Muster und anwendungsspezifische Muster unterscheiden. Letztere bieten beispielsweise Problemlösungen für Planungssysteme oder Warenwirtschaftssysteme /Fowler 97a/. Das Ziel dieses Kapitels ist es, Ihnen möglichst allgemeine Muster zur Verfügung zu stellen.

**Beschreibung von Mustern** Jedes Muster wird über einen eindeutigen Namen identifiziert. Es wird durch ein oder mehrere Beispiele erläutert, die skizzieren, für welche Problemstellung das Muster eine Lösung anbietet. Anschließend werden die typischen Eigenschaften dieses Musters aufgeführt.

#### **Muster 1: Liste**

**Motivation** Die Informationen einer Bestellung und ihrer Bestellpositionen lassen sich wie in Abb. 3.1-1 darstellen. Eine Bestellung besteht sozusagen aus einem Bestellungskopf und den einzelnen Positionen. Analog läßt sich ein Lager mit all seinen einzelnen Lagerplätzen als Liste visualisieren. Ein Lagerplatz kann nicht ohne Lager existieren.

Bestellung				
Nr.	4711			1.1.1999
Hiermit bestelle ich folgende Artikel				
Nr.	Bezeichnung	Einzelpreis	Anzahl	Preis
47	Kugelschreiber	12,00	2	24,00
11	Folienstift	2,50	5	12,50
Summe				36,50

Lager				
Bezeichnung	Süd			
Standort	Dortmund			
Verwalter	HansMüller			
Ebene	Gang	Platz	Belegt mit	
1	1	1	XYZ	
1	1	2	ABC	
1	2	1		

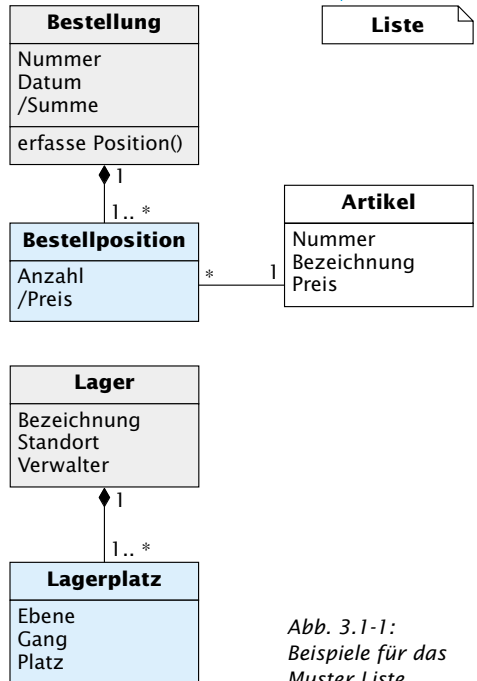


Abb. 3.1-1: Beispiele für das Muster Liste

Es macht vom Problem her auch kein Sinn, ihn einem anderen Lager zuzuordnen. Die Attributwerte des Lagers gelten auch für jeden Lagerplatz. Ist beispielsweise das Lager gekühlt, dann ist auch jeder Lagerplatz gekühlt.

Diese Problemstellung kommt in vielen Anwendungsbereichen immer wieder vor und wird als Komposition modelliert.

Die UML erlaubt es, auf jedem Diagramm wichtige Informationen als Notiz anzugeben.

- Es liegt eine Komposition vor.
- Ein Ganzes besteht aus gleichartigen Teilen, d.h. es gibt nur eine Teil-Klasse.
- Teil-Objekte bleiben einem Aggregat-Objekt fest zugeordnet. Sie können jedoch gelöscht werden, bevor das Ganze gelöscht wird.
- Die Attributwerte des Aggregat-Objekts gelten auch für die zugehörigen Teil-Objekte.
- Das Aggregat-Objekt enthält im allgemeinen mindestens ein Teil-Objekt, d.h. die Kardinalität ist meist 1..\*.

Notiz

Eigenschaften

**Muster 2: Exemplartyp**

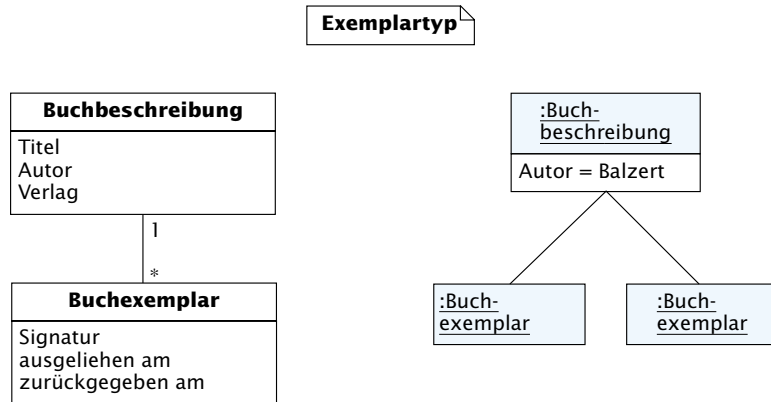
Von einem Buch sind mehrere Exemplare zu verwalten. Würde diese Problemstellung durch eine einzige Klasse Buch modelliert, dann

Motivation

## LE 5 3 Analysemuster und Beispielanwendungen

würden mehrere Objekte bei Titel, Autor und Verlag identische Attributwerte besitzen. Eine bessere Modellierung ergibt sich, wenn die gemeinsamen Attributwerte mehrerer Buchexemplare in einem neuen Objekt `Buchbeschreibung` zusammengefaßt werden (Abb. 3.1-2).

Abb. 3.1-2:  
Beispiel für das  
Muster  
Exemplartyp

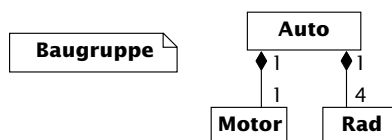


- Eigenschaften
- Es liegt eine einfache Assoziation vor, denn es besteht keine *whole-part*-Beziehung.
  - Erstellte Objektverbindungen werden nicht verändert. Sie werden nur gelöscht, wenn das betreffende Exemplar entfernt wird.
  - Der Name der neuen Klasse enthält oft Begriffe wie Typ, Gruppe, Beschreibung, Spezifikation.
  - Eine Beschreibung kann – zeitweise – unabhängig von konkreten Exemplaren existieren. Daher ist die Kardinalität im allgemeinen *many*.
  - Würde auf die neue Klasse verzichtet, so würde als Nachteil lediglich die redundante »Speicherung« von Attributwerten auftreten.

### Muster 3: Baugruppe

Motivation In der Abb. 3.1-3 soll ausgedrückt werden, daß jedes Auto exakt einen Motor und vier Räder haben soll. Da es sich hier um physische Objekte handelt, liegt ein physisches Enthaltensein vor, das mittels Komposition modelliert wird. Wenn ein Auto verkauft wird, dann gehören Motor und Räder dazu. Die Zuordnung der Teile zu ihrem Ganzen bleibt normalerweise über einen längeren Zeitraum bestehen. Der Motor kann jedoch durch ein neuen Motor ersetzt werden und der alte Motor in ein anderes Objekt eingebaut werden.

Abb. 3.1-3:  
Beispiel für das  
Muster Baugruppe



- Es handelt sich um physische Objekte.
- Es liegt eine Komposition vor.
- Objektverbindungen bestehen meist über eine längere Zeit. Ein Teil-Objekt kann jedoch von seinem Aggregat-Objekt getrennt werden und einem anderen Ganzen zugeordnet werden.
- Ein Ganzes kann aus unterschiedlichen Teilen bestehen.

Eigenschaften

**Muster 4: Stückliste**

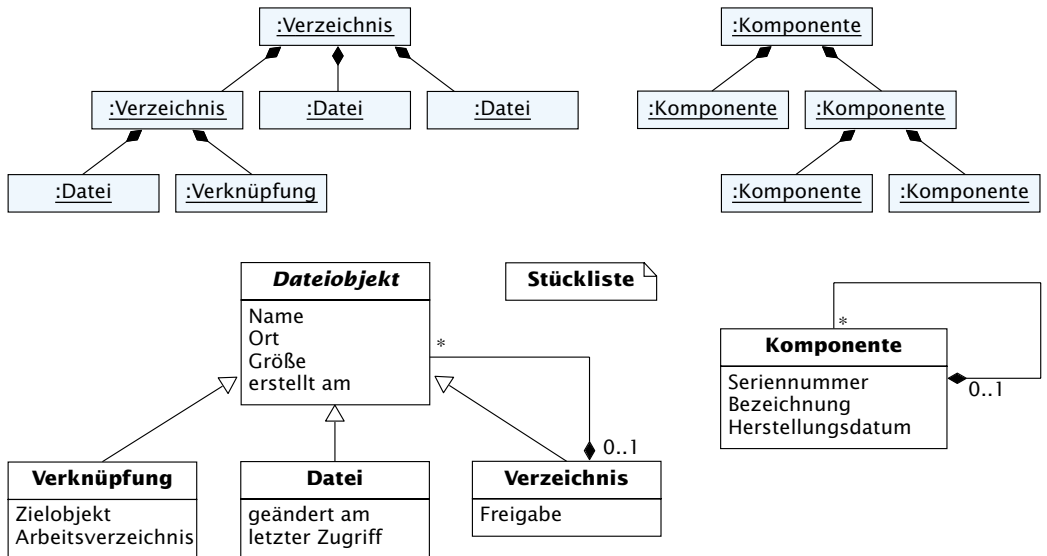
Es soll modelliert werden, daß ein Verzeichnis Verknüpfungen, Dateien und weitere Verzeichnisse enthalten kann (Abb.3.1-4). Dabei soll sowohl das Verzeichnis und alle darin enthaltenen Objekte als Einheit als auch jedes dieser Objekte einzeln behandelt werden können. Wird beispielsweise das Verzeichnis kopiert, dann sollen alle darin enthaltenen Dateiobjekte kopiert werden. Wird das Verzeichnis gelöscht, dann werden auch alle seine Teile gelöscht. Ein Dateiobjekt kann jedoch vorher einem anderen Verzeichnis zugeordnet werden.

Motivation

Ein Sonderfall liegt vor, wenn sich diese Enthaltensein-Beziehung auf gleichartige Objekte bezieht. Beispielsweise setzt sich jede Komponente aus mehreren Komponenten zusammen. Umgekehrt ist jede Komponente in einer oder keiner anderen Komponente enthalten.

Diese Problemstellung wird durch eine Komposition modelliert, wobei die verschiedenen Teil-Objekte durch eine Vererbung dargestellt werden. Beachten Sie die 0..1-Kardinalität bei der Klasse Verzeichnis. Eine 1-Kardinalität würde bedeuten, daß jedes Dateiobjekt – also auch jedes Verzeichnis – in einem anderen Verzeichnis enthalten sein müßte.

Abb. 3.1-4: Beispiele für das Muster Stückliste



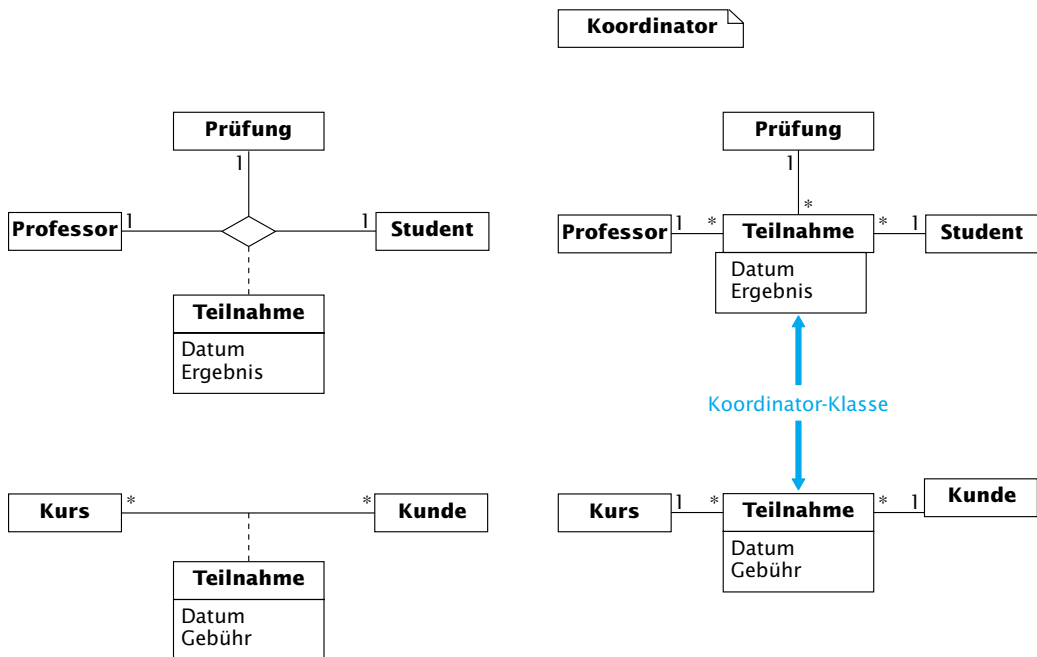
## LE 5 3 Analysemuster und Beispielanwendungen

- Eigenschaften
- Es liegt eine Komposition vor.
  - Das Aggregat-Objekt und seine Teil-Objekte müssen sowohl als Einheit als auch einzelnen behandelt werden können.
  - Teil-Objekte können anderen Aggregat-Objekten zugeordnet werden.
  - Die Kardinalität bei der Aggregat-Klasse ist 0..1.
  - Ein Objekt der Art A kann sich aus mehreren Objekten der Arten A, B und C zusammensetzen.
  - Der Sonderfall der Stückliste ist, daß ein Stück nicht aus Objekten unterschiedlicher Art, sondern nur aus einer einzigen Art besteht.

### Muster 5: Koordinator

Motivation In der Abb. 3.1-5 verbindet eine ternäre Assoziation Objekte der Klassen Professor, Prüfung und Student und »merkt« sich Informationen über eine abgelegte Prüfung in der assoziativen Klasse Teilnahme. Diese ternäre Assoziation kann wie abgebildet in binäre Assoziationen und eine Koordinator-Klasse aufgelöst werden. Für eine Koordinator-Klasse ist typisch, daß sie oft selbst nur wenige Attribute und Operationen besitzt, sondern sich vor allem merkt »wer wen kennt«. Als Sonderfall dieser Problemstellung kann eine binäre Assoziation mit einer assoziativen Klasse betrachtet werden.

Abb. 3.1-5:  
Beispiele für das  
Muster Koordinator



- Es liegen einfache Assoziationen vor.
- Die Koordinator-Klasse ersetzt eine n-äre ( $n \geq 2$ ) Assoziation mit assoziativer Klasse.
- Die Koordinator-Klasse besitzt kaum Attribute/Operationen, sondern mehrere Assoziationen zu anderen Klassen, im allgemeinen zu genau einem Objekt jeder Klasse.

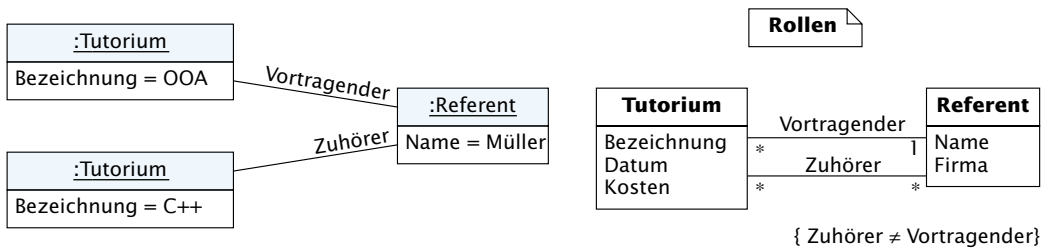
Beschreibung

**Muster 6: Rollen**

Zu einem Tutorium sind Vortragende und Zuhörer zu verwalten (siehe Aufgabe 1 der Lehreinheit 3). Dabei kann ein Referent sowohl Vortragender als auch Zuhörer von Tutorien sein. Mit anderen Worten: der Referent spielt – zur selben Zeit – in Bezug auf die Klasse Tutorium um mehrere Rollen. Diese Problemstellung kommt relativ häufig vor und wird wie in Abb. 3.1-6 modelliert. Würden anstelle der Klasse Referent die Klassen Vortragender und Zuhörer verwendet, dann hätten beide Klassen dieselben Attribute (und Operationen). Außerdem könnte nicht modelliert werden, daß ein bestimmtes Referent-Objekt sowohl Vortragender als auch Zuhörer – bei anderen Tutorien – ist.

Motivation

Abb. 3.1-6: Beispiel für das Muster Rollen



- Zwischen zwei Klassen existieren zwei oder mehrere einfache Assoziationen.
- Ein Objekt kann – zu einem Zeitpunkt – in Bezug auf die Objekte der anderen Klasse verschiedene Rollen spielen.
- Objekte, die verschiedene Rollen spielen können, besitzen unabhängig von der jeweiligen Rolle die gleichen Eigenschaften und ggf. gleiche Operationen.

Eigenschaften

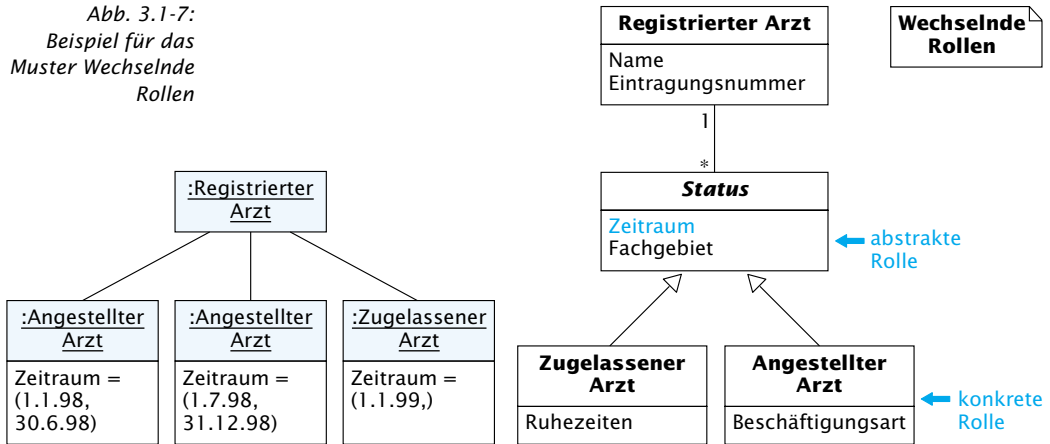
**Muster 7: Wechselnde Rollen**

In der Abb. 3.1-7 wird modelliert, daß ein kassenärztlich registrierter Arzt im ersten und zweiten Halbjahr 1998 in unterschiedlichen Praxen jeweils eine Tätigkeit als angestellter Arzt ausübt, bevor er am 1.1.1999 seine Zulassung erhält. Für angestellte Ärzte sind teilweise andere Informationen zu speichern als für die Zugelassenen. Im Gegensatz zum Rollen-Muster spielt der registrierte Arzt während eines Zeitraum verschiedene Rollen. Da es hier darum geht, Informationen über einen Zeitraum festzuhalten, werden neue ärztli-

Motivation

## LE 5 3 Analysemuster und Beispielanwendungen

Abb. 3.1-7:  
Beispiel für das  
Muster Wechselnde  
Rollen



che Tätigkeiten und deren Objektverbindungen zu Registrierter Arzt immer nur hinzugefügt.

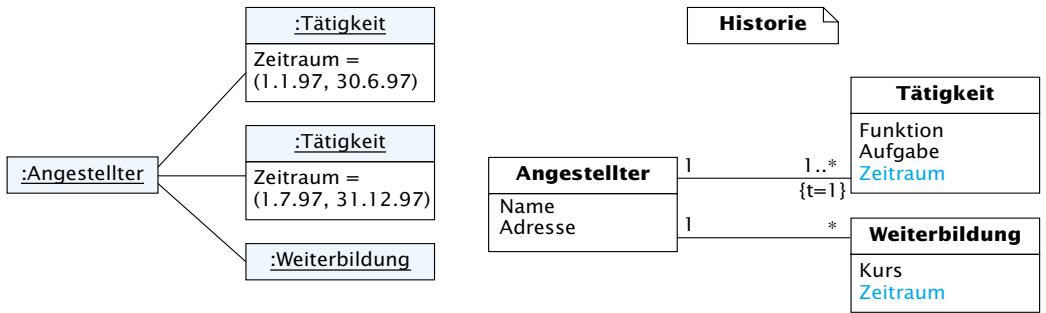
- Eigenschaften
- Ein Objekt der realen Welt kann zu verschiedenen Zeiten verschiedene Rollen spielen. In jeder Rolle kann es unterschiedliche Eigenschaften (Attribute, Assoziationen) und Operationen besitzen.
  - Die unterschiedlichen Rollen werden mittels Vererbung modelliert.
  - Objektverbindungen zwischen dem Objekt und seinen Rollen werden nur erweitert, d.h. weder gelöscht noch zu anderen Objekten aufgebaut.

### Muster 8: Historie

Motivation Für einen Angestellten sollen alle Tätigkeiten, die er während der Zugehörigkeit zu einer Firma ausübt, festgehalten werden. Dabei darf zu jedem Zeitpunkt nur eine aktuelle Tätigkeit gültig sein. Außerdem sind alle Weiterbildungskurse, die er im Laufe seiner Firmenzugehörigkeit besucht, aufzuzeichnen. Diese Problemstellung wird wie in Abb. 3.1-8 mittels Assoziationen modelliert. Für jede Tätigkeit und jede Weiterbildung wird der Zeitraum eingetragen. Die Restriktion  $\{t=1\}$  spezifiziert, daß ein Angestellter zu einem Zeitpunkt genau eine Tätigkeit ausübt. Wenn alle Tätigkeiten und Weiterbildungen gespeichert sein sollen, dann bedeutet dies, daß die aufgebauten Verbindungen zu Tätigkeit und Weiterbildung bestehen bleiben bis der Angestellte die Firma verläßt und seine Daten gelöscht werden.

- Eigenschaften
- Es liegt eine einfache Assoziation vor.
  - Für ein Objekt sind mehrere Vorgänge bzw. Fakten zu dokumentieren.
  - Für jeden Vorgang bzw. jedes Faktum ist der Zeitraum festzuhalten.





- Aufgebaute Objektverbindungen zu den Vorgängen bzw. Fakten werden nur erweitert.
- Die zeitliche Restriktion {t=k} (k = gültige Kardinalität) sagt aus, was zu einem Zeitpunkt gelten muß.

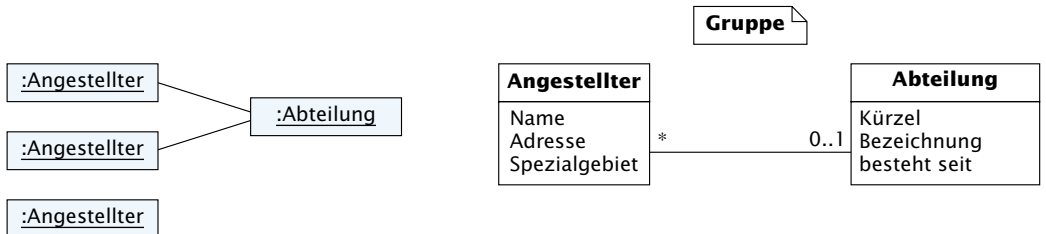
Abb. 3.1-8 :  
Beispiel für das  
Muster Historie

**Muster 9: Gruppe**

In der Abb. 3.1-9 bildet sich eine Gruppe, wenn mehrere Angestellte zu einer Abteilung gehören. Da die Abteilung auch – kurzfristig – ohne zugehörige Angestellte existieren soll, wird die *many*-Kardinalität gewählt. Sollte modelliert werden, daß beim Eintragen der Abteilung mindestens ein Angestellter zugeordnet wird, dann wäre die Kardinalität 1..\* zu wählen. Wenn ein Angestellter aus einer Abteilung ausscheidet, dann wird die entsprechende Objektverbindung getrennt.

Motivation

Abb. 3.1-9:  
Beispiel für das  
Muster Gruppe



- Es liegt eine einfache Assoziation vor.
- Mehrere Einzel-Objekte gehören – zu einem Zeitpunkt – zum selben Gruppen-Objekt.
- Es ist jeweils zu prüfen, ob die Gruppe – zeitweise – ohne Einzel-Objekte existieren kann oder ob sie immer eine Mindestanzahl von Einzel-Objekten enthalten muß.
- Objektverbindungen können auf- und abgebaut werden.

Eigenschaften

**Muster 10: Gruppenhistorie**

Soll die Zugehörigkeit zu einer Gruppe nicht nur zu einem Zeitpunkt, sondern über einen Zeitraum festgehalten werden, dann ist eine Problemstellung wie in Abb. 3.1-10 zu modellieren. Für jeden

Motivation

## LE 5 3 Analysemuster und Beispielanwendungen

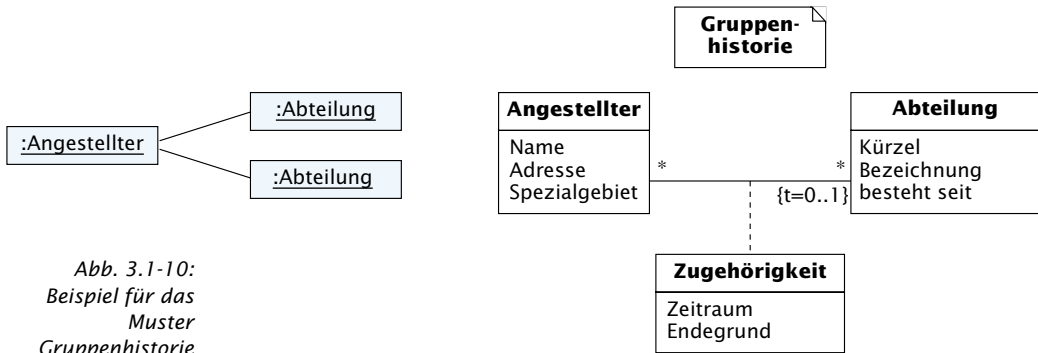


Abb. 3.1-10:  
Beispiel für das  
Muster  
Gruppenhistorie

Angestellten wird festgehalten, über welchen Zeitraum er zu einer Abteilung gehört hat. Die Restriktion  $\{t=0..1\}$  sagt aus, daß er zu einem Zeitpunkt in maximal einer Abteilung tätig sein kann. Wenn ein Angestellter eine Abteilung verläßt, dann wird dies durch die Attributwerte im entsprechenden Objekt von Zugehörigkeit beschrieben.

- Eigenschaften
- Ein Einzel-Objekt gehört – im Laufe der Zeit – zu unterschiedlichen Gruppen-Objekten.
  - Die Historie wird mittels einer assoziativen Klasse modelliert. Dadurch ist die Zuordnung zwischen Einzel-Objekten und Gruppen deutlich sichtbar.
  - Die zeitliche Restriktion  $\{t=k\}$  ( $k$  = gültige Kardinalität) sagt aus, was zu einem Zeitpunkt gelten muß.
  - Da Informationen über einen Zeitraum festzuhalten sind, bleiben erstellte Objektverbindungen bestehen und es werden nur Verbindungen hinzugefügt.

## 3.2 Beispiel Materialwirtschaft

### Problembeschreibung

/1/ Es ist die Materialwirtschaft für ein chargenorientiertes Informationssystem zu modellieren.

/2/ Es sind verschiedene Materialien (z.B. Joghurt) zu verwalten, von denen mehrere zur gleichen Gruppe (z.B. Milchprodukte) gehören können.

/3/ Die Joghurtmenge, die zusammen hergestellt wird, gehört zu einer Charge und erhält eine gemeinsame Chargennummer.

/4/ Das System soll sowohl komplette Paletten als auch einzelne Gebinde verwalten. Jede Palette enthält im allgemeinen mehrere Gebinde. Ein solches Gebinde ist beispielsweise eine 6-er Packung Joghurts und eine Palette ein Karton mit 24 Gebinden.

/5/ Eine Palette, deren Gebinde alle zur gleichen Charge – und damit automatisch zum gleichen Material – gehören, heißt chargenhomogene Palette. Enthält sie Gebinde aus unterschiedlichen Char-

gen des gleichen Materials oder Gebinde unterschiedlichen Materials, dann liegt eine Mischpalette vor.

**/6/** Es sind zwei Lagertypen zu verwalten. Das Stellplatzlager ist ein strukturiertes Lager, das aus einzelnen Stellplätzen besteht. Stellplätze können unterschiedlich groß sein. Diese Größe wird durch die Anzahl der Segmente ausgedrückt. Dementsprechend können auf einem Stellplatz eine oder mehrere Paletten bzw. Gebinde eingelagert werden. Das offene Lager ist ein Lagerraum ohne weitere Struktur.

**/7/** Zu jedem Lager ist der Standort aufzuzeichnen. An einem Standort können sich mehrere Lager befinden.

**/8/** Es können folgende Transaktionen durchgeführt werden: Eine Palette kann eingelagert werden. Ganze Paletten oder einzelne Gebinde – als Einzelbestände bezeichnet – können umgelagert oder auch ausgelagert werden. Ausgelagerte Paletten und Gebinde werden im System gelöscht.

**/9/** Jede Bewegung (Transaktion) muß nachvollziehbar sein. Das heißt, daß für jede Einlagerung, jede Umlagerung und jede Auslagerung ein Protokolleintrag erstellt werden muß.

Das Beispiel Materialwirtschaft basiert auf dem »echten« Analysemodell der Materialwirtschaft des chargenorientierten Informationssystems Charisma, das von der Firma G.U.S. in Köln entwickelt und vertrieben wird. Ich danke der G.U.S. für die Genehmigung zur Publikation dieses Beispiels.

Praxisbezug

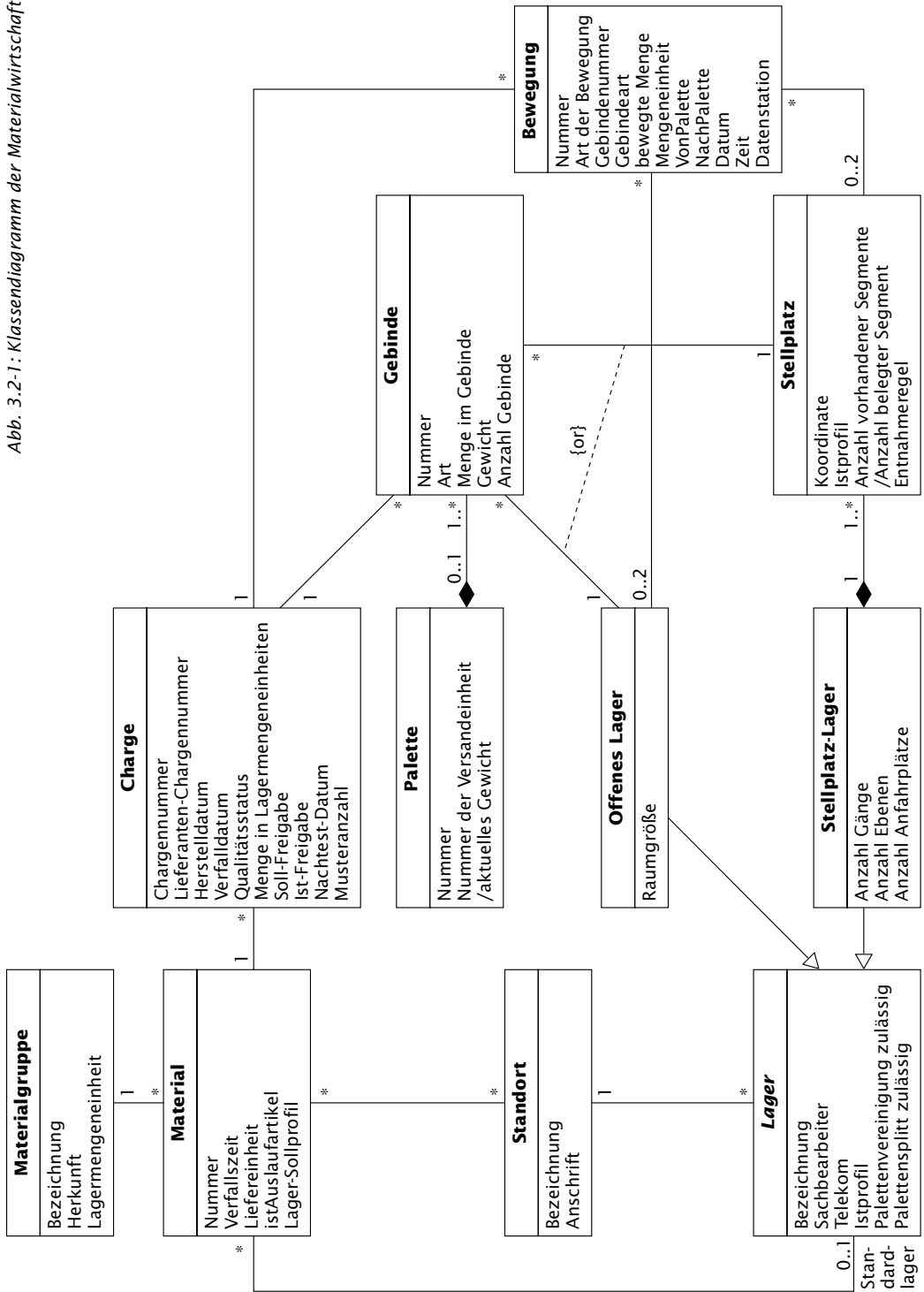
### **Klassendiagramm**

Die Abb. 3.2 -1 zeigt das Klassendiagramm. Beachten Sie, daß nicht jedes physikalische Gebinde auf ein Gebinde-Objekt abgebildet wird, sondern alle gleichartigen physikalischen Gebinde und deren Anzahl werden als ein logisches Gebinde-Objekt gespeichert. Bei einer chargenhomogenen Palette stehen alle Gebinde dieser Palette mit dem gleichen Chargen-Objekt in Verbindung, bei einer Mischpalette mit unterschiedlichen Objekten. Daher muß die Information, ob eine chargenhomogene Palette vorliegt, nicht als Attribut gespeichert werden. Sie könnte jedoch als abgeleitetes Attribut angegeben werden. Das Klassendiagramm sagt für jedes logische Gebinde aus, auf welchem Stellplatz bzw. in welchem Lager es steht. Die Verwaltung der Bestände erfolgt somit auf Gebinde-Ebene. Die Palette dient lediglich dazu, Gebinde für den Bediener zu gruppieren und als Gesamtheit zu bearbeiten, z.B. Umlagern einer kompletten Palette anstelle aller Gebinde dieser Palette. Wird die Palette aus dem System gelöscht, dann werden auch alle ihre Gebinde entfernt. Zwischen Palette und Gebinde besteht daher eine Komposition.

Die Klasse Bewegung besitzt nur Assoziationen zu Charge, Offenem Lager und Stellplatz, obwohl sie die Bewegungen von Palette und Gebinden protokolliert. Das ist notwendig, weil Paletten und

# LE 5 3 Analysemuster und Beispielanwendungen

Abb. 3.2-1: Klassendiagramm der Materialwirtschaft



Gebinde nach ihrer Auslagerung im System gelöscht werden, während die Bewegungsdaten bestehen bleiben sollen.

Im Klassendiagramm lassen sich folgende – in Kap. 3.1 beschriebene – Muster identifizieren:

Liste: Stellplatzlager – Stellplatz

Exemplartyp: Materialgruppe – Material

### **Geschäftsprozesse**

Folgende Geschäftsprozesse sind in dieser vereinfachten Version auszuführen:

- Buchen ungeplanter Zugänge,
- Umlagern von Einzelbeständen und
- Buchen ungeplanter Entnahmen.

Wir betrachten den ersten Geschäftsprozeß genauer.

*Geschäftsprozeß:* buchen ungeplanter Zugänge

*Ziel:* Paletten, für die keine Bestellung im System existiert, sollen eingelagert werden

*Vorbedingung:* Material existiert, Lager mit ausreichender Kapazität existieren

*Nachbedingung Erfolg:* Bewegungsprotokoll über Einlagerung erstellt

*Nachbedingung Fehlschlag:* falsch gelieferte Ware abgewiesen

*Akteure:* Warenannahme

*Auslösendes Ereignis:* Paletten werden angeliefert

*Beschreibung:*

- 1 Material abrufen
- 2 Charge abrufen
- 3 Angaben für chargenhomogene Palette eingeben
- 4 automatische Suche eines Lagerplatzes
- 5 sofortige Einlagerung der Ware
- 6 Bewegungsprotokoll aktualisieren

*Erweiterungen:*

- 4a manuelle Suche eines Lagerplatzes
- 5a drucken Palettenbegleitschein
- 5b drucken Gebinde-Ident-Etiketten

*Alternativen:*

- 2a neue Charge anlegen
- 3a Angaben für eine Mischpalette eingeben
- 4a Mischpaletten mit unterschiedlichen Materialien müssen unter Umständen auf verschiedene Lagerplätze aufgeteilt werden
- 5a Ausstellung eines Fahrbefehls, um die Einlagerung später durchzuführen

Dieser Geschäftsprozeß ist auf einem sehr hohen Abstraktionsniveau beschrieben. Einige Verarbeitungsschritte sind selbst wieder – abstrakte – Geschäftsprozesse. Für diese abstrakten Geschäftsprozesse können die Angaben zu Akteur (identisch mit dem übergeordneten Geschäftsprozeß) und auslösendem Ereignis (Aktivierung durch den übergeordneten Geschäftsprozeß) entfallen.

## LE 5 3 Analysemuster und Beispielanwendungen

*Geschäftsprozess:* automatische Suche eines Lagerplatzes

*Ziel:* anhand der Vorgaben Standardlager und Sollprofil wird ein Lagerplatz ermittelt

*Vorbedingung:* Standardlager erfüllt Sollprofil

*Nachbedingung Erfolg:* Lagerplatz gefunden

*Nachbedingung Fehlschlag:* kein Lagerplatz am Standort des Standardlagers gefunden

*Beschreibung:*

1 prüfe, ob im Standardlager noch Platz ist

2 prüfe, ob am gleichen Standort noch andere Lager existieren und dort Platz ist

3 prüfe zuerst, ob ein Stellplatzlager das Sollprofil erfüllt und prüfe dann die offenen Lager

*Erweiterungen:* -

*Alternativen:* -

*Geschäftsprozess:* manuelle Suche eines Lagerplatzes

*Ziel:* alle Lager ermitteln, die das Sollprofil erfüllen

*Vorbedingung:*

*Nachbedingung Erfolg:* Lagerplatz entsprechend Sollprofil gefunden

*Nachbedingung Fehlschlag:* kein Lagerplatz mit passendem Sollprofil gefunden

*Beschreibung:*

1 Auswahl der Orte

2 Ermitteln aller Lager mit passendem Sollprofil und freiem Platz

3 Auswahl eines Lagers

4 Bei einem Stellplatzlager wird ein Stellplatz vorgeschlagen

*Erweiterungen:* -

*Alternativen:* -

Die Abhängigkeiten zwischen den betrachteten Geschäftsprozessen sind in der Abb. 3.2-2 spezifiziert. Die automatische Suche eines Lagerplatzes geht immer vom Standardlager des einzulagernden Materials aus und wird daher nur bei Einlagerungen von Paletten (buchen ungeplanter Zugänge) durchgeführt.

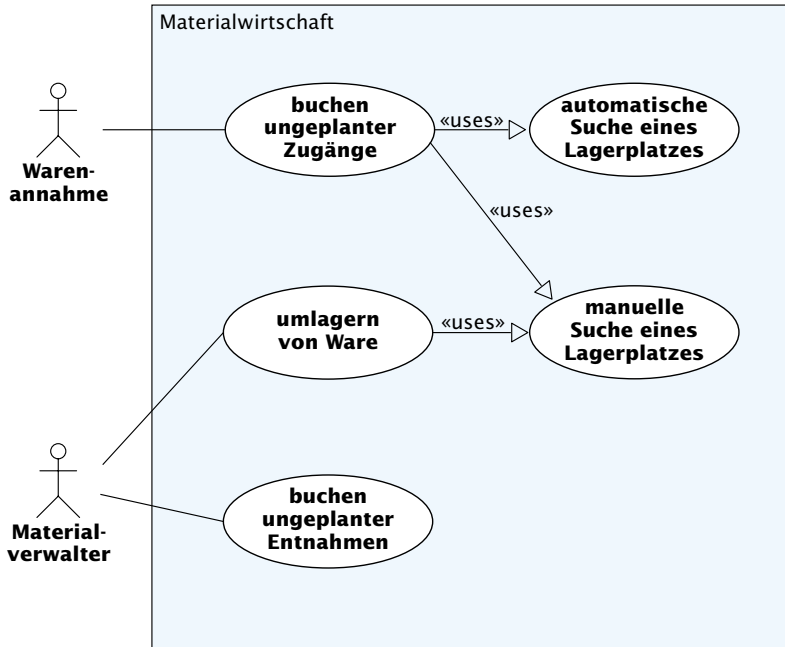


Abb. 3.2-2:  
Geschäftsprozeß-  
diagramm für  
Materialwirtschaft

### 3.3 Beispiel Arztregister

#### Problembeschreibung

**/1/** Im Arztregister werden Informationen über Ärzte gespeichert. Jeder Arzt muß sich in derjenigen Arztregister-Stelle (AR-Stelle) eintragen lassen, in deren Bereich er seine Wohnung hat. Wechselt er innerhalb der AR-Stelle die Wohnung, dann sind auch diese Angaben festzuhalten. Er kann auch die AR-Stelle wechseln.

**/2/** Ein Arzt kann – zu einem Zeitpunkt – einen von drei Status besitzen: zugelassener Arzt, angestellter Arzt und Job-Sharing-Arzt.

**/3/** Als zugelassener Arzt kann er selbständig praktizieren. Er kann zusätzlich an höchstens zwei Krankenhäusern als Belegarzt tätig sein. Zugelassene Kinderärzte und Internisten können wählen, ob sie als Hausarzt oder als Facharzt tätig sein wollen. Diese Wahl kann bei Bedarf geändert werden.

**/4/** Den Angestellten-Status erhält ein Arzt, wenn er bei einem zugelassenen Arzt angestellt ist. Ein zugelassener Arzt darf zu einem Zeitpunkt entweder zwei Angestellte halbtätig oder einen Angestellten gantztägig beschäftigen.

**/5/** Erhält ein Arzt eine Job-Sharing-Zulassung, dann ist er zwar selbständig tätig, die Zulassung ist jedoch an die Zulassung seines Senior-Partners gebunden, d.h. sie gilt für die gleichen Fachgebiete, sie ändert sich mit ihr und erlischt mit ihr. Er praktiziert immer in

## LE 5 3 Analysemuster und Beispielanwendungen

der gleichen Praxis wie der Senior-Partner, der ein zugelassener Arzt sein muß.

**/6/** Ein zugelassener Arzt kann in einer Einzelpraxis oder in einer Gemeinschaftspraxis praktizieren. Bei einer Gemeinschaftspraxis müssen alle aktiv beteiligten Ärzte die gleiche Abrechnungsnummer besitzen. Einer der Ärzte kann als verantwortlicher Arzt eingetragen sein.

Praxisbezug Das Beispiel Arztregister basiert auf dem »echten« Analysemodell des Bundesarztregisters (BAR) der Kassenärztlichen Bundesvereinigung (KBV) in Köln. Ich danke der KBV für die Genehmigung zur Publikation dieses Beispiels.

### Klassendiagramm

Die Abb. 3.3-1 zeigt das Klassendiagramm. Die Trennung der Klassen Arzt und Person, die hier nicht unbedingt notwendig ist, ermöglicht es, das Arztregister auch für die Verwaltung anderer Personengruppen, z.B. Psychotherapeuten, zu erweitern. Von der Aufgabenstellung her soll eine möglichst vollständige Historie möglich sein, d.h. alle Anfragen sollen sich nicht nur auf den Status quo, sondern auf beliebige Zeiten in der Vergangenheit beziehen können.

Abb. 3.3-2 zeigt eine Variante dieses Modells ohne Historie. In diesem zweiten Fall wird für einen Arzt nur die letzte AR-Stelle gespeichert. Da es zu jedem Arzt maximal eine Registrierung geben kann, werden diese beiden Klassen zu der neuen Klasse *Registrierter Arzt* zusammengefaßt. Auf die Klasse *Wohnung* wird verzichtet, und ihre Attribute werden in die Klasse *Person* integriert. Für jeden registrierten Arzt wird nur der letzte Status (Anstellung, Zulassung oder *Job-Sharing-Zulassung*) festgehalten, was durch die 0..1-Kardinalität festgehalten wird. Für die Zulassung müssen nun nicht mehr alle fach- und hausärztlichen Tätigkeiten festgehalten werden, sondern das Attribut *Haus/Facharzt* gibt an, in welcher Funktion der zugelassene Arzt zur Zeit tätig ist. Daß ein zugelassener Arzt zu einem Zeitpunkt maximal zwei Angestellte beschäftigen und zwei *Job-Sharing-Partner* haben kann, wird durch die 0..2-Kardinalität ausgedrückt. Ein Schnappschuß impliziert außerdem, daß zu einer Zulassung genau eine Praxis (entweder Einzelpraxis oder Gemeinschaftspraxis) eingetragen ist.

Folgende Muster lassen sich im Klassendiagramm der Abb. 3.3-1 identifizieren:

Historie: Arzt – Registrierung

Historie: Registrierung – Wohnung

Wechselnde Rollen: Registrierung – *Status* – Anstellung, Zulassung, *Job-Sharing-Zulassung*

Gruppenhistorie: Zulassung – Zugehörigkeit – Gemeinschaftspraxis



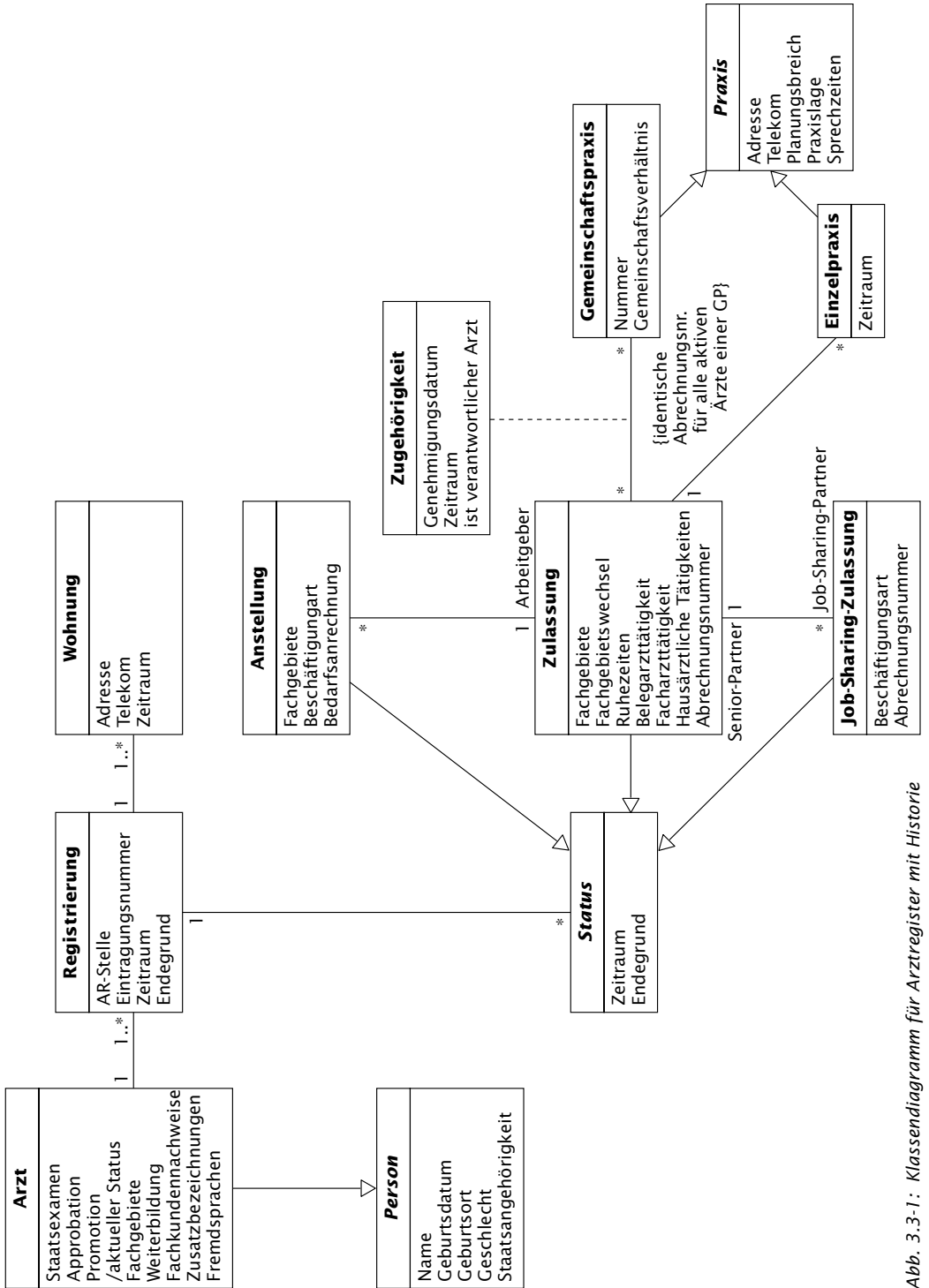
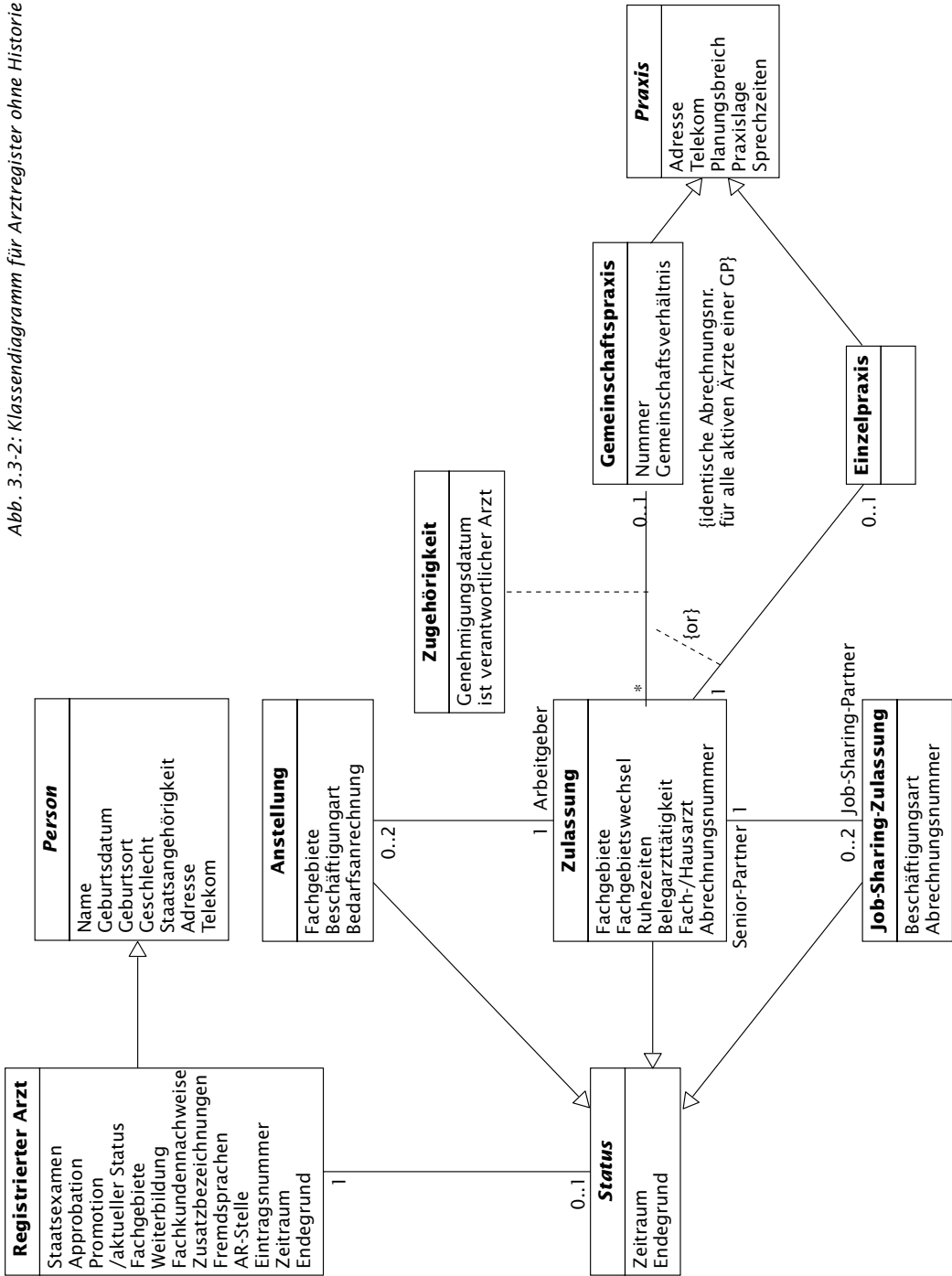


Abb. 3.3-1: Klassendiagramm für Arztregister mit Historie

Abb. 3.3-2: Klassendiagramm für Arztregister ohne Historie



**Geschäftsprozesse**

Aus der Problemstellung lassen sich zahlreiche Geschäftsprozesse ableiten. Einige davon sind:

- Zulassung eintragen,
- Job-Sharing eintragen,
- Angestellten eintragen und
- Gemeinschaftspraxis gründen.

Abb. 3.3-3 zeigt das Geschäftsprozeßdiagramm.

*Geschäftsprozeß:* Zulassung eintragen

*Ziel:* Arzt kann als zugelassener Arzt praktizieren

*Vorbereitung:*

*Nachbedingung Erfolg:* Zulassung erteilt

*Nachbedingung Fehlschlag:* Abweisen des Antrags

*Akteure:* Ärztesachbearbeiter

*Auslösendes Ereignis:* Antrag auf Zulassung

*Beschreibung:*

- 1 Registrierung des Arztes prüfen
- 2 Überprüfen der Zulassungsvoraussetzung und Eintragen der Zulassung
- 3 Eintragen der Einzelpraxis

*Erweiterungen:*

- 1a Arzt neu registrieren
- 1b Arzt in neuer AR-Stelle registrieren
- 1c Arztdaten aktualisieren

*Alternativen:*

- 3a Arzt einer vorhandenen Gemeinschaftspraxis zuordnen

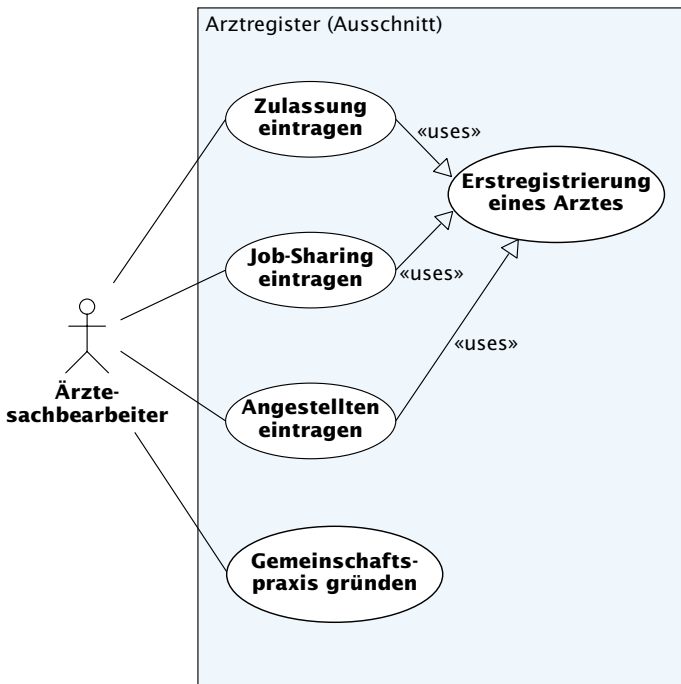


Abb. 3.3-3:  
Geschäftsprozeß-  
diagramm für  
Arztregister

## LE 5 3 Analysemuster und Beispielanwendungen

*Geschäftsprozeß:* Job-Sharing eintragen

*Ziel:* Arzt kann als Job-Sharing-Partner praktizieren

*Vorbereitung:*

*Nachbedingung Erfolg:* Job-Sharing-Zulassung erteilt

*Nachbedingung Fehlschlag:* Abweisen des Antrags

*Akteure:* Ärztesachbearbeiter

*Auslösendes Ereignis:* Antrag auf Job-Sharing-Zulassung

*Beschreibung:*

- 1 Registrierung des Arztes prüfen
- 2 Überprüfen des Senior-Partners
- 3 Überprüfen der Voraussetzungen für Job-Sharing und Eintragen der Zulassung

*Erweiterungen:*

1a Arzt neu registrieren

1b Arzt in neuer AR-Stelle registrieren

1c Arztdaten aktualisieren

*Alternativen:* -

### 3.4 Beispiel Friseursalonverwaltung

#### Problembeschreibung

**/1/** Der Friseursalon verkauft ein bestimmtes Artikelsortiment an seine Kunden. Außerdem werden diese Artikel auch im Salon verbraucht. Jeder Artikel wird von genau einer Firma geliefert.

**/2/** Für jeden Kunden wird angezeigt, welche Dienstleistungen (z.B. Färben) er zuletzt erhalten hat. Der Friseur kann z.B. erkennen, daß das letzte Färben am 23.6. durchgeführt wurde und die Kundin Pfeiffer bei ihrem letzten Besuch (30.8.) eine Tönung wählte. Außerdem wird festgehalten, welcher Mitarbeiter diese Dienstleistung erbracht hat.

**/3/** Für jeden Kunden existiert eine zweite Maske, in der Informationen über die durchgeführten chemischen Behandlungen, z.B. verwendete Haarfärbemittel, gespeichert werden. Im Fachjargon wird von »Chemie« gesprochen.

**/4/** Ein Kunde meldet sich für einen Salonbesuch fest an. Dabei werden außer dem Datum auch die Zeit und der Mitarbeiter festgehalten, der den Kunden hauptverantwortlich betreuen soll. Dieser Mitarbeiter wird im Fachjargon als »Stylist« bezeichnet. Der Kunde kann jedoch auch von anderen Mitarbeitern – den Assistenten – Dienstleistungen erhalten.

**/5/** Alle Mitarbeiter des Salons werden verwaltet. Der Friseursalon beschäftigt auch Mitarbeiter, die keinen Dienst am Kunden verrichten, z.B. für Verwaltungsaufgaben. Für alle Mitarbeiter sind die Anwesenheitsdaten zu speichern.

**/6/** Der Salon kann bis zu 40 verschiedene Dienstleistungen (z.B. Schnitt) anbieten. Zu jeder Dienstleistung kann es mehrere Ausprägungen (z.B. Schnitt kurz, Schnitt lang, Chef-Schnitt) geben, die sich in ihrem Zeitbedarf und im Preis unterscheiden.

/7/ Jeder Salonbesuch endet mit dem Kassieren der erbrachten Dienstleistungen.

/8/ Haarpflege-Artikel können im Rahmen eines Salonbesuchs oder unabhängig davon erworben werden. Im zweiten Fall handelt es sich um Laufkundschaft. Beim Verkauf ist es möglich, daß Preisnachlässe gegeben werden.

/9/ Ein Kunde kann beliebig viele Abonnements erwerben, mit denen er erbrachte Dienstleistungen bezahlt. Jedes Abonnement bezieht sich auf genau einen Kunden und eine Dienstleistung.

Das Beispiel Friseursalonverwaltung habe ich aus meinem Buch »Methoden der objektorientierten Systemanalyse« /Balzert 96a/ entnommen und geringfügig modifiziert. Es hat seinen Ursprung in dem »echten« Analysemodell des Coiffeur Information Systems (CIS) der Firma Schleupen Computersysteme AG in Moers, der ich für die Genehmigung zur Publikation dieses Beispiels danke.

Praxisbezug

#### **Klassendiagramm**

Die Abb. 3.4-1 zeigt das Klassendiagramm, das im Gegensatz zu den bisherigen Beispielen eine Reihe von Operationen enthält. Diejenigen Salonmitarbeiter, die Dienst am Kunden verrichten, besitzen mehr Attribute, Assoziationen und Operationen als die anderen Salonmitarbeiter. Sie werden daher als Unterklasse von Mitarbeiter (ohne Dienst am Kunden) modelliert.

Wenn sich ein Kunde anmeldet, wird ein neues Objekt der Klasse Salonbesuch erzeugt und mit je einem Objekt von Kunde und Kundenmitarbeiter verbunden. Beim Abrechnen des Salonbesuchs werden die erbrachten Dienstleistungen zugeordnet und dann ein Objekt des Kassiervorgangs erzeugt. Der Kassiervorgang muß sich jedoch nicht unbedingt auf einen Salonbesuch beziehen. Daher werden die Klassen Salonbesuch und Kassiervorgang nicht zusammengefaßt.

#### **Geschäftsprozesse**

Der zentrale Geschäftsprozeß befaßt sich mit der Durchführung eines Salonbesuchs.

*Geschäftsprozeß:* Kunden im Salon bedienen

*Ziel:* Kunde erhält im Salon Dienstleistungen und erwirbt Artikel

*Vorbereitung:* Kunde ist angemeldet

*Nachbereitung Erfolg:* Kunde hat bezahlt

*Nachbereitung Fehlschlag:*

*Akteure:* Stylist, Rezeptionist

*Auslösendes Ereignis:* Kunde trifft im Salon ein

*Beschreibung:*

- 1 Laufzettel mit Kundendaten ausfüllen
- 2 Dienstleistungen erbringen und auf Laufzettel eintragen
- 3 Laufzettel erfassen
- 4 Verkäufe an Kunden eintragen
- 5 Rechnung erstellen und Bezahlung verbuchen

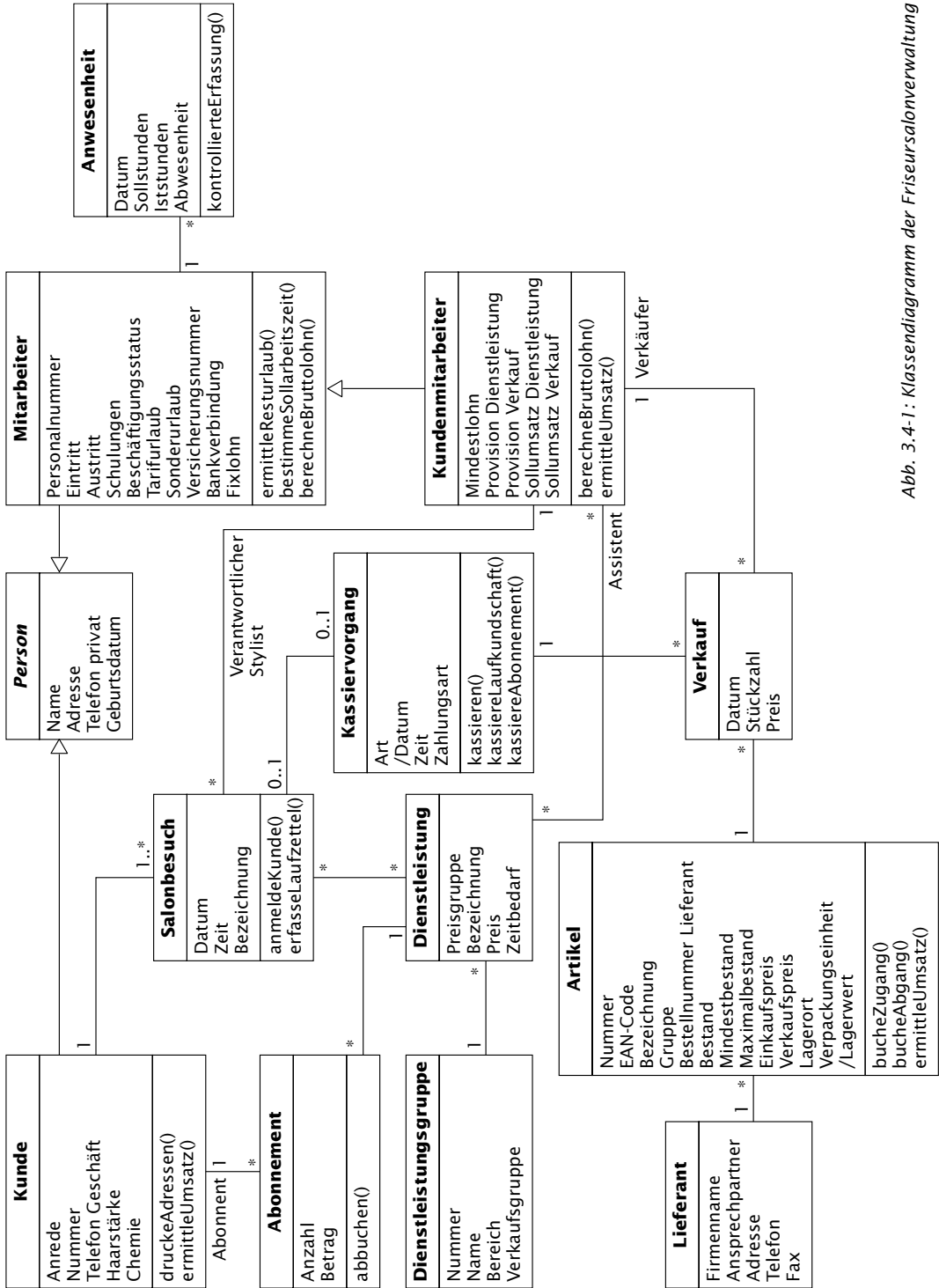


Abb. 3.4-1: Klassendiagramm der Friseursalonverwaltung

### 3.4 Beispiel Friseursalonverwaltung LE 5

Erweiterungen:

- 1a Kundendaten sind zu aktualisieren
- 1b Anmelddaten sind zu aktualisieren
- 2a Entnahme von Artikeln für die Durchführung von Dienstleistungen
- 3a Chemische Behandlung für Kunden eintragen
- 4a Abo verkaufen
- 5a Teilweise Bezahlung durch vorhandenes Abo

Alternativen: -

Weitere Geschäftsprozesse sind:

- Anmelden eines Kunden für einen Salonbesuch,
- Kassieren von Verkäufen bei Laufkundschaft,
- Nachbestellung von Artikeln,
- Kontrollieren der Arbeitszeiten für Mitarbeiter und
- Berechnung der Löhne für Mitarbeiter.

Abb. 3.4-2 zeigt das Geschäftsprozeßdiagramm. Sowohl beim Bedienen des Kunden im Salon als auch beim Kassieren der Laufkundschaft müssen Verkäufe kassiert werden. Diese gemeinsame Funktionalität wird durch die *uses*-Beziehung beschrieben.

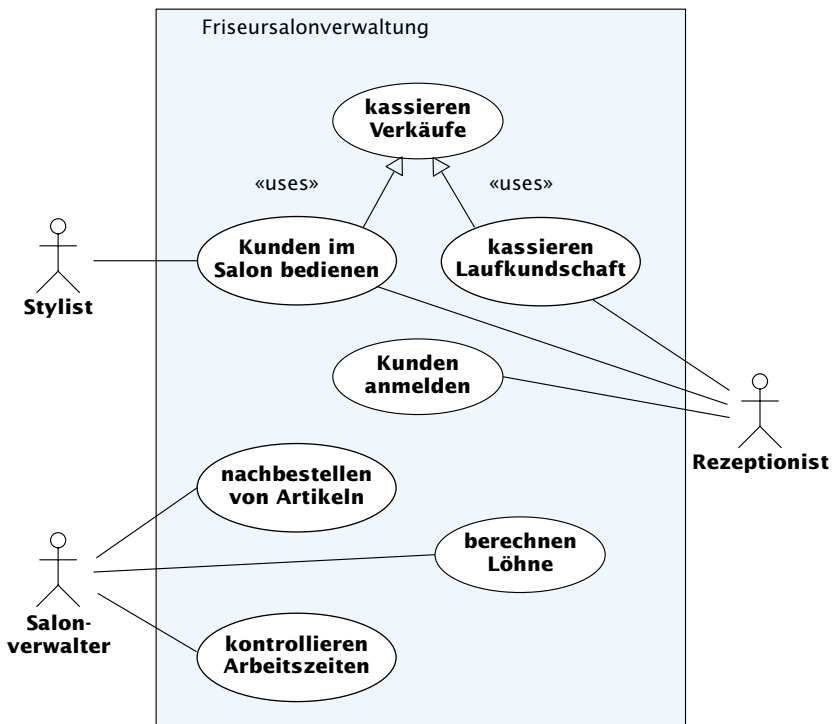


Abb. 3.4-2:  
Geschäftsprozeß-  
diagramm für  
Friseursalon-  
verwaltung

## 3.5 Beispiel Seminarorganisation

### Problembeschreibung

/1/ Es werden mehrere Seminare unterschiedlichen Typs angeboten, die im Katalog veröffentlicht werden. Beispielsweise werden in diesem Jahr drei Seminare des Typs *Java-Einführung* angeboten.

/2/ Für diese Seminare können sich Kunden (Teilnehmer) anmelden. Das können sowohl Privatpersonen als auch Mitarbeiter einer Firma sein.

/3/ Um Kunden mit schlechter Zahlungsmoral leichter zu erkennen merkt sich das System die abgerechneten und noch nicht bezahlten Seminare.

/4/ Gebuchte Seminare können durch die Kunden abgesagt werden oder vom Seminaranbieter storniert werden. Im zweiten Fall erhält jeder gebuchte Kunde eine Absage.

/5/ Ein Seminar kann auch von einer Firma als internes Firmenseminar gebucht werden. Auch dieses interne Seminar kann durch die Firma abgesagt oder vom Seminaranbieter storniert werden.

/6/ Wenn ein Seminar durchgeführt ist, erhalten alle Teilnehmer mit der Rechnung eine Urkunde.

/7/ Firmen erhalten nach Durchführung des internen Seminars nur eine Rechnung. Die Teilnehmer sind in diesem Fall dem Seminaranbieter nicht bekannt.

/8/ Im Fall von Änderungen müssen Kunden und Firmen benachrichtigt werden.

/9/ Bei jedem Seminar können mehrere Dozenten referieren. Einige Dozenten sind gleichzeitig als Seminarleiter tätig. Um die Planung zu erleichtern, wird für jeden Dozenten vermerkt, welche Seminartypen er fachlich abhalten kann.

Bemerkung Das Beispiel Seminarorganisation wird als Fallstudie in dem »Lehrbuch der Software-Technik« von Helmut Balzert /Balzert 96/ durchgängig verwendet. Ich habe es aufgeführt, um den Vergleich mit der hier verwendeten Notation und Methode zu erleichtern.

### Klassendiagramm

Die Abb. 3.5-1 zeigt das Klassendiagramm. Jede Kundenbuchung bezieht sich auf genau einen Kunden, der hier die Rolle des Teilnehmers spielt. Diese Verbindung bleibt bestehen, wenn sie einmal aufgebaut ist. Die Forderung /3/ wird auf die Assoziation *Debitor* zwischen Kunde und Kundenbuchung abgebildet. Wenn für eine Kundenbuchung die Rechnung erstellt ist, wird die Debitor-Verbindung zum Kunden aufgebaut. Nach dem Bezahlen der Rechnung wird sie wieder entfernt. Das Vorhandensein dieser Verbindung sagt also aus, daß eine Rechnung gestellt, aber noch nicht bezahlt ist.



### 3.5 Beispiel Seminarorganisation LE 5

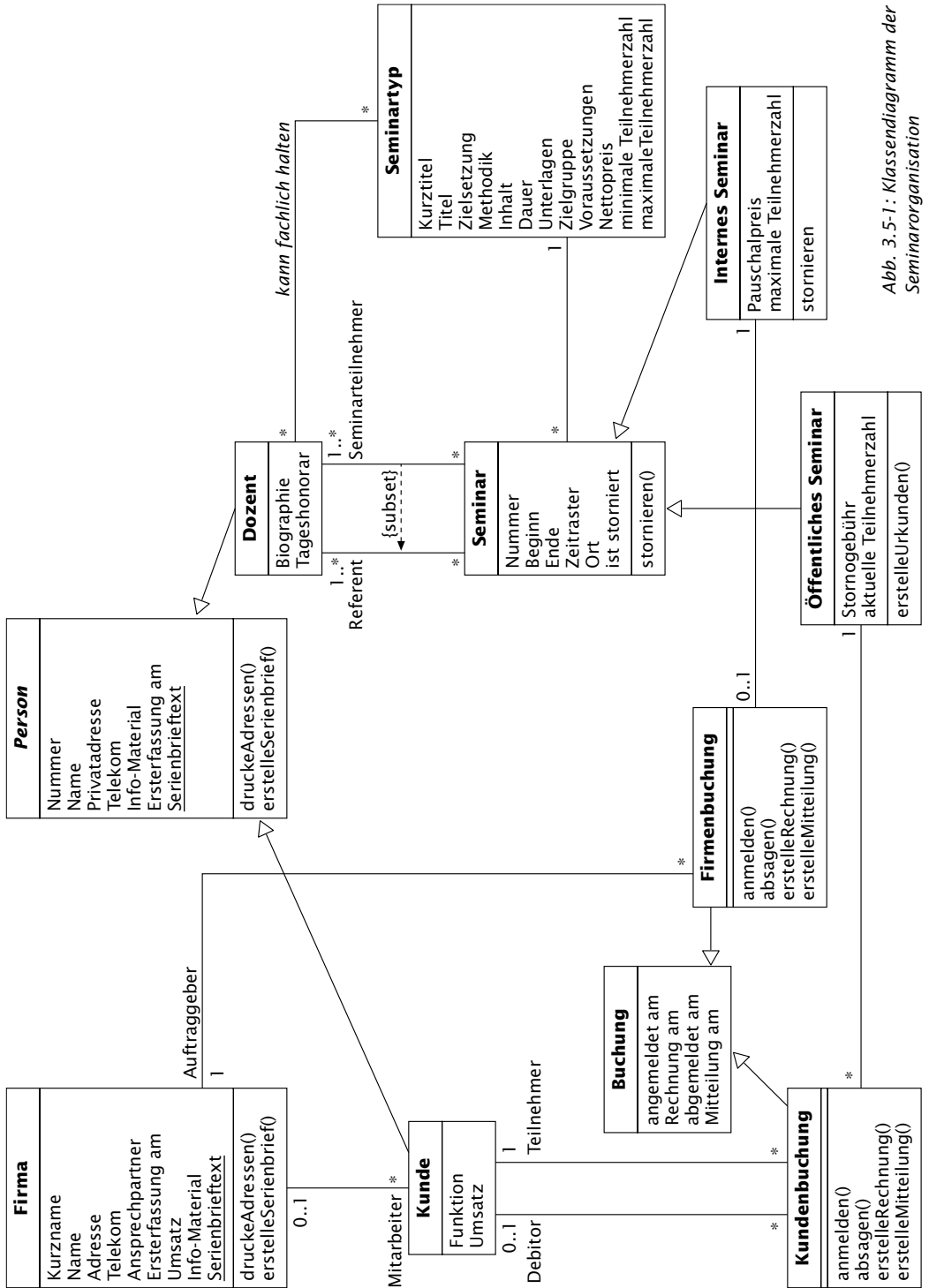
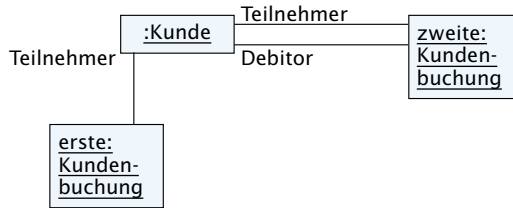


Abb. 3.5-1: Klassendiagramm der Seminarorganisation

## LE 5 3 Analysemuster und Beispielanwendungen

Abb. 3.5-2:  
Zur Modellierung  
der Debitoren



### Geschäftsprozesse

Folgende Geschäftsprozesse (Abb. 3.5-3) sind durchzuführen:

- bearbeite Anmeldung,
- bearbeite Firmenanmeldung,
- bearbeite Absage,
- bearbeite Firmenabsage,
- Seminar abrechnen,
- Firmenseminar abrechnen,
- storniere Seminar,
- storniere Firmenseminar und
- informiere Kunden und Firmen über Seminare.

*Geschäftsprozeß:* bearbeite Anmeldung

*Ziel:* Teilnehmer für Seminar anmelden

*Vorbereitung:*

*Nachbereitung Erfolg:*

*Nachbereitung Fehlschlag:*

*Akteure:* Sachbearbeiter für Kunden

*Auslösendes Ereignis:* Anmeldung trifft ein

*Beschreibung:*

- 1 prüfen, ob Kunde schon vorhanden ist
- 2 prüfen, ob Kunde schon angemeldet ist
- 3 prüfen, ob gewünschtes Seminar existiert
- 4 prüfen, ob Seminar frei ist
- 5 erstelle Anmeldebestätigung

*Erweiterungen:*

- 1a evtl. Kundendaten ändern
- 3a Nachfragen, welches Seminar gewünscht wird
- 4a Ersatzseminare anbieten

*Alternativen:*

- 1a neuen Kunden erfassen
- 5a erstelle Absage

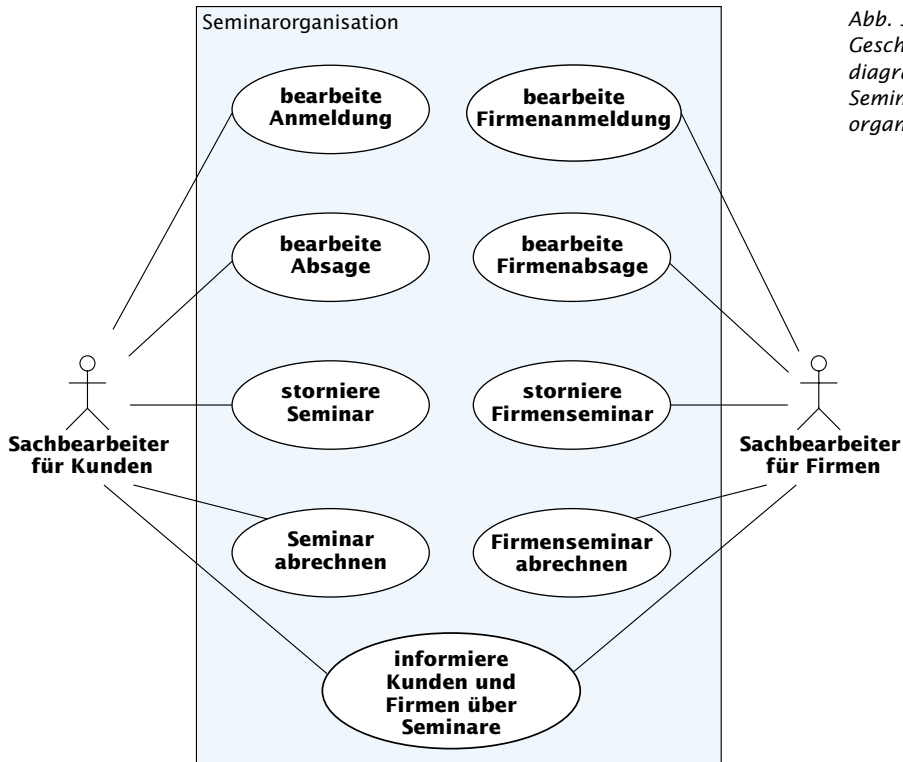


Abb. 3.5-3:  
Geschäftsprozess-  
diagramm für  
Seminar-  
organisation



#### Analysemuster (analysis pattern)

Ein Analysemuster ist eine Gruppe von Klassen mit feststehenden Verantwortlichkeiten und Interaktionen, die eine bestimmte – wiederkehrende – Problemlösung beschreiben.

**Muster (pattern)** Ein Muster ist – ganz allgemein – eine Idee, die sich in einem praktischen Kontext als nützlich erwie-

sen hat und es wahrscheinlich auch in anderen sein wird. Muster beschreiben Strukturen von Klassen bzw. Objekten, die sich in Softwaresystemen wiederholt finden und dienen zur Lösung bekannter Probleme. Entsprechend ihrer Anwendung in der jeweiligen Phase unterscheidet man →Analyse- und Entwurfsmuster.



**Muster** ermöglichen die Standardisierung bestimmter Probleme; sie sind katalogisierte Projekterfahrungen. Hier wurden folgende **Analysemuster** beschrieben: Liste, Exemplartyp, Baugruppe, Stückliste, Koordinator, Rollen, wechselnde Rollen, Historie, Gruppe, Gruppenhistorie. Bei den Beispielen werden das Klassendiagramm für die aufgeführte Problembeschreibung, ein einfaches Geschäftsprozessdiagramm und Spezifikationsschablonen ausgewählter Geschäftsprozesse dokumentiert. Die Materialwirtschaft ist ein Ausschnitt aus einem chargenorientierten Informationssystem. Im Arztregister werden Informationen über Ärzte gespeichert. Für dieses Beispiel wurden zwei Versionen des Klassendiagramms erstellt:

## LE 5 Zusammenhänge/Aufgaben

mit und ohne Historie. Die Friseursalonverwaltung unterstützt einen Friseurbetrieb und die Seminarorganisation einen Seminarveranstalter.

Aufgabe  
30 Minuten

### 1 *Lernziel: Modellieren und Erkennen von Mustern.*

Modellieren Sie folgende Problemstellungen als Klassendiagramme. Denken Sie sich für die Klassen geeignete Attribute aus. Prüfen Sie, welches der beschriebenen Muster vorliegt.

- a** Mehrere Personen schließen sich zu einer Fahrgemeinschaft zusammen.
- b** Ein Projektplan (z.B. ein Balkendiagramm) besteht aus mehreren Planungsschritten.
- c** Ein Mitarbeiter tritt als Programmierer in ein Unternehmen ein. Nach ein paar Jahren wird er als Manager tätig und steigt später zum Geschäftsführer auf. Für Programmierer, Manager und Geschäftsführer sind unterschiedliche Eigenschaften festzuhalten.
- d** In einem Sportverein sind Sportler zu verschiedenen Zeiten in unterschiedlichen Mannschaften aktiv.
- e** In einem Grafiksystem bilden Kreise und Rechtecke eine Gruppe. Diese Gruppe kann wiederum Teil einer anderen Gruppe sein.
- f** Zu einem Inventarstück in einem Museum sollen – sofern die Daten vorhanden sind – der derzeitige Eigentümer, der Vorbesitzer, der Finder und/oder der Überbringer festgehalten werden, die jeweils die gleichen Eigenschaften besitzen. Eine Person kann beispielsweise sowohl Eigentümer als auch Finder sein.
- g** Bei mehreren Videokassetten in einer Videothek handelt es sich um den gleichen Film.
- h** Für Personen sollen die Wohnsitze der letzten 10 Jahre ermittelt werden können. Zu einem Zeitpunkt muß jede Person mindestens einen und kann höchstens zwei Wohnsitze besitzen.

Aufgabe  
15–20 Minuten

### 2 *Lernziel: Systematisches Identifizieren von Analysemustern in einem Klassendiagramm.*

Geben Sie an, welche Muster in den Klassendiagrammen

- a** der Seminarverwaltung (Abb. 3.5-1) und
- b** der Friseursalonverwaltung (Abb. 3.4-1) vorhanden sind.

- 3** *Lernziele: Systematisches Identifizieren von Analysemustern beim Erstellen eines Klassendiagramms.* Aufgabe  
20 Minuten

Erstellen Sie anhand der folgenden Problembeschreibung ein Klassendiagramm und verwenden Sie dabei systematisch Muster. Kennzeichnen Sie alle identifizierten Muster im Diagramm.

Eine Praxis mit mehreren Ärzten soll intern verwaltet werden. Für jeden Patienten sind Name, Adresse und Geburtsdatum zu speichern. Jeder Arzt vertritt bestimmte Fachgebiete. Der Patient kann mehrere Ärzte dieser Praxis konsultieren. Für jede Behandlung werden das Datum, die Diagnose und die erteilten Verordnungen festgehalten. Jede Verordnung umfaßt die Packungsgröße, das Medikament und ggf. eine Vorschrift für die Anwendung. Mehrere Behandlungen werden gemeinsam abgerechnet. Die Abrechnung enthält das Rechnungsdatum und den Behandlungszeitraum, sowie die einzelnen Abrechnungspositionen. Jede Position besteht aus einer laufenden Nummer, der Leistung, dem Abrechnungssatz und den Kosten.

## 4 Checklisten zum Erstellen eines OOA-Modells



- Erklären können, wie der Analyseprozeß ablaufen soll.
- Geschäftsprozesse systematisch identifizieren und dokumentieren können.
- Pakete systematisch identifizieren können.

verstehen  
anwenden



- Die objektorientierten Konzepte und die UML-Notation, wie sie in Kapitel 2 beschrieben werden, müssen bekannt sein.
- Die Kenntnisse der Kapitel 1 und 3.1 sind nützlich.
- Aus dem Kapitel 3 sollten Sie mindestens ein Fallbeispiel durchgearbeitet haben.



- 4.1 Analyseprozeß 120
- 4.2 Checkliste Geschäftsprozeß 127
- 4.3 Checkliste Paket 134

## 4.1 Analyseprozeß

Definition Wie in vielen anderen Bereichen gilt auch für die Softwareentwicklung, daß nur durch eine Prozeßverbesserung eine Produktverbesserung erreicht werden kann. Gegenstand dieser und der nächsten beiden Lehreinheiten ist dieser **Analyseprozeß**, d.h. die methodische Vorgehensweise zur Erstellung eines objektorientierten Analysemodells. Die UML enthält keine Angaben zur methodischen Vorgehensweise /UML 97/.

### Methodische Vorgehensweise

Die Auswahl der richtigen methodischen Vorgehensweise ist eine Gratwanderung zwischen Formalismus und Formlosigkeit. Sehr formelle Vorgehensweisen fordern soviel Formalismus und Umstände, daß sie jede Kreativität ersticken. Formlose Vorgehensweisen sind chaotisch und nicht tragbar, da der Erfolg der Projekte nicht vorhersehbar ist.

Methoden-Schulen /Booch 94/ sieht fünf – sich etwas überlappende – Schulen bei objektorientierten Methoden:

*anarchists*: Sie ignorieren alle methodischen Vorgehensweisen und verlassen sich nur auf die eigene Kreativität.

*behaviorists*: Sie konzentrieren sich auf Rollen und Verantwortlichkeiten.

*storyboarder*: Sie sehen die Welt als Menge von Geschäftsprozessen.

*information modeler*: Sie betrachten zunächst nur die Daten; das Verhalten ist sekundär.

*architects*: Sie haben ihren Fokus auf *frameworks* und *patterns* gerichtet.

In den Anfängen der Objektorientierung wurde zunächst das statische Modell stark betont. Oft entstand ein semantisches Datenmodell in objektorientierter Notation, in dem die Dynamik des System außer acht gelassen wurde. Andere Vorgehensweisen wie *use case driven approach* oder *scenario driven approach* stützen sich überwiegend auf das dynamische Modell. Eine rein funktionale Zerlegung besitzt jedoch alle Nachteile der strukturierten Analysetechniken, da sich die funktionale Struktur nicht direkt auf eine objektorientierte Architektur abbilden läßt. Für eine erfolgreiche Modellierung ist das Zusammenwirken von statischem und dynamischem Modell unabdingbar. Zur Validierung des statischen Modells wird das dynamische Modell benötigt und umgekehrt. Der Analytiker muß daher permanent zwischen beiden Modellen wechseln, bis ein akzeptables Analysemodell erstellt ist.

Analyseprozeß Der hier beschriebene Analyseprozeß besteht aus

- dem Makroprozeß, der die methodischen Schritte festlegt und
- methodischen Regeln, die in Form von Checklisten zur Verfügung stehen.

Die methodischen Schritte beschreiben auf einem hohen Abstraktionsniveau, in welcher Reihenfolge die einzelnen Aufgaben zur Erstellung eines OOA-Modells auszuführen sind. Wir sprechen daher von einem **Makroprozeß**. Bei der praktischen Anwendung zeigt sich, daß eine solche grobe Prozeßbeschreibung nicht ausreicht. Andererseits ist ein sehr detailliertes Vorgehensmodell oft nur für bestimmte Anwendungen geeignet und kann nicht problemlos auf andere Bereiche übertragen werden. Der erfahrene Systemanalytiker wendet – meist mehr oder minder intuitiv – Hunderte von Regeln an, die er situationsspezifisch einsetzt. Für die Anwendung der Regeln gibt es keine festgelegte Reihenfolge. Diese Regeln stehen in Form von **Checklisten** zur Verfügung. Außerdem greift ein erfahrener Analytiker in vielen Fällen auf bereits gelöste ähnliche Problemstellungen zurück (Muster). Durch die Verwendung von **Mustern** reduzieren Sie einerseits den eigenen Aufwand und erreichen andererseits eine höhere Standardisierung, was es anderen Lesern erleichtert, sich in Ihr Modell einzuarbeiten. Analysemuster wurden in Kapitel 3.1 beschrieben.



Kapitel 3.1

### Makroprozeß

Der erste Schritt des Makroprozesses ist die Identifikation der relevanten Geschäftsprozesse. Formulieren Sie diese Geschäftsprozesse informal oder semiformal und erstellen Sie Geschäftsprozeßdiagramme. Leiten Sie daraus die Klassen ab. Konzentrieren Sie sich zunächst auf das statische Modell. In vielen Firmen existieren Formulare, Dateibeschreibungen und weitere Dokumente, aus denen die Daten des Systems sehr gut entnommen werden können. Andere Daten und ihre Abhängigkeiten können durch Interviews gewonnen werden. Auf diese Weise kommen Sie sehr schnell zu einem ersten Kern des Modells, der eine gute Basis für weitere Arbeiten ist. Im nächsten Schritt konzentrieren Sie sich auf das dynamische Modell. Dabei treten Rückkopplungen zum statischen Modell auf. Achten Sie auf die Konsistenz beider Modelle.

Makroprozeß

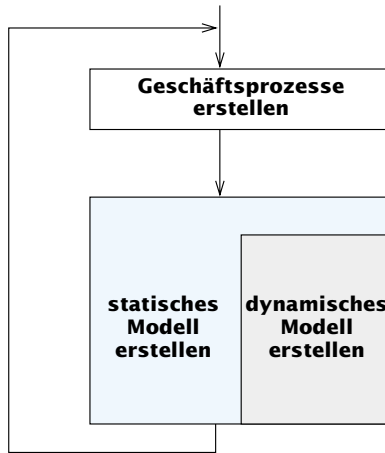
Der beschriebene Makroprozeß berücksichtigt die Gleichgewichtigkeit (*balancing*) von statischem und dynamischem Modell. Wir sprechen daher von einem **balancierten Makroprozeß**. Die Konzentration auf das statische Modell vor dem dynamischen sorgt für eine größere Stabilität des Modells und schafft durch die Bildung von Klassen eine wesentliche Abstraktionsebene. Steht zu Beginn der Modellierung nur das dynamische Modell im Vordergrund, dann besteht meines Erachtens das Problem, daß der Analytiker in der großen Menge von – oft geänderten – Funktionen den Überblick verliert. Wichtig ist, daß nach dem Erstellen des ersten statischen Modellkerns das dynamische und das statische Modell parallel weiter entwickelt werden, um deren Wechselwirkungen adäquat berücksichtigen zu können (Abb. 4.1-1).



**Makroprozeß**

- Ermitteln Sie die relevanten Geschäftsprozesse.
- Leiten Sie daraus die Klassen ab.
- Erstellen Sie das statische Modell.
- Erstellen sie parallel dazu das dynamische Modell.
- Berücksichtigen Sie die Wechselwirkung beider Modelle.

Abb. 4.1-1:  
Zur Modellbildung



evolutionär

Die hier beschriebene Methode realisiert einen evolutionären Entwicklungsprozeß. Das bedeutet, daß zunächst eine objektorientierte Analyse für den Produktkern erstellt wird, der anschließend zu entwerfen und implementieren ist. Dieser Kern wird in weiteren Zyklen erweitert, bis ein auslieferbares System entsteht. Dabei soll die Arbeit früherer Zyklen nicht neu gemacht, sondern korrigiert und verbessert werden. Ein evolutionärer Prozeß ist immer iterativ, weil er eine ständige Verfeinerung der Systemarchitektur erfordert. Alle Erfahrungen und Ergebnisse eines Iterationsschrittes fließen in den nächsten Schritt ein.

integrierte Qualitätssicherung

Eine systematische Softwareentwicklung ist heute im allgemeinen mit dem Einsatz von Werkzeugen verbunden. Werkzeuge können viele formale Prüfungen durchführen, erlauben es, Änderungen und Erweiterungen einfach und sicher durchzuführen und verwalten alle erstellten Dokumente. Der Einsatz von Werkzeugen garantiert also von vornherein eine bestimmte Qualität. Viele Qualitätskriterien sind jedoch semantischer Natur und lassen sich nicht automatisch prüfen. Wir verwenden dazu das Verfahren der formalen Inspektion (vgl. Kapitel 4.11). Diese integrierte Qualitätssicherung ist ein weiterer Faktor für den Erfolg eines Projekts.

Kapitel 4.11

Der beschriebene Makroprozeß umfaßt folgende Aufgabenbereiche:

- Analyse im Großen (Tab. 4.1-1),
- 6 Schritte zum statischen Modell (Tab. 4.1-2) und
- 3 Schritte zum dynamischen Modell (Tab. 4.1-3).



**1 Geschäftsprozesse aufstellen**

Erstellen Sie die essentiellen Geschäftsprozesse.

- Beschreibung Geschäftsprozesse
- Geschäftsprozeßdiagramm

**2 Pakete bilden**

Bilden Sie Teilsysteme, d.h. fassen Sie Modellelemente zu Paketen zusammen. Bei großen Systemen, die im allgemeinen durch mehrere Teams bearbeitet werden, muß die Bildung von Paketen am Anfang stehen.

- Paketdiagramm

**Tab. 4.1-1:  
Analyse  
im Großen**

**1 Klassen identifizieren**

Identifizieren Sie für jede Klasse nur so viele Attribute und Operationen, wie für das Problemverständnis und das einwandfreie Identifizieren der Klasse notwendig sind.

- Klassendiagramm
- Kurzbeschreibung Klassen

**2 Assoziationen identifizieren**

Tragen Sie zunächst nur die reinen Verbindungen ein, d.h. machen Sie noch keine genaueren Angaben, z.B. Kardinalität, Art der Assoziation.

- Klassendiagramm

**3 Attribute identifizieren**

Identifizieren Sie alle Attribute des Fachkonzepts

- Klassendiagramm

**4 Vererbungsstrukturen identifizieren**

Erstellen Sie aufgrund der identifizierten Attribute Vererbungsstrukturen.

- Klassendiagramm

**5 Assoziationen vervollständigen**

Treffen Sie die endgültige Festlegung, ob eine »normale« Assoziation, Aggregation oder Komposition vorliegt und geben Sie Kardinalitäten, Rollen, Namen und Restriktionen an.

- Klassendiagramm
- Objektdiagramm

**6 Attribute spezifizieren**

Erstellen Sie für alle identifizierten Attribute eine vollständige Spezifikation.

- Attributspezifikation

Die angegebenen Schritte können nicht immer sequentiell durchlaufen werden. Beispielsweise lassen sich oft gleichzeitig mit dem Identifizieren der Klassen auch Assoziationen finden.

**Tab. 4.1-2:  
6 Schritte zum  
statischen Modell**

Bei der Analyse im Großen handelt es sich um Aufgaben, die nicht spezifisch für eine objektorientierte Entwicklung sind, während die statische und dynamische Modellierung einen objektorientierten Charakter besitzen. Diese einzelnen Vorgehensweisen werden in den angegebenen Tabellen näher beschrieben. Für jeden Schritt wird angegeben, welche Diagramme bzw. Spezifikationen zu erstellen sind (mit »→« gekennzeichnet).

**Tab. 4.1-3:**  
**3 Schritte zum**  
**dynamischen**  
**Modell**

**1 Szenarios erstellen**

Präzisieren Sie jeden Geschäftsprozeß durch eine Menge von Szenarios.

→Sequenzdiagramm, Kollaborationsdiagramm

**2 Zustandsautomat erstellen**

Prüfen Sie für jede Klasse, ob ein nicht-trivialer Lebenszyklus erstellt werden kann.

→Zustandsdiagramm

**3 Operationen beschreiben**

Überlegen Sie, ob eine Beschreibung notwendig ist. Wenn ja, dann ist je nach Komplexitätsgrad die entsprechende Form zu wählen.

→Klassendiagramm

→fachliche Beschreibung der Operationen, Zustandsautomaten, Aktivitätsdiagramme

Natürlich gibt es immer wieder Anwendungen und Situationen, in denen ein anderer Weg sinnvoll ist. Mögliche Alternativen zum oben beschriebenen balancierten Makroprozeß sind der szenario-basierte und der daten-basierte Makroprozeß /IBM 97/.

szenario-basiert

Der **szenario-basierte Makroprozeß** ist empfehlenswert, wenn umfangreiche funktionale Anforderungen vorliegen und alte Datenbestände nicht existieren. Er besteht aus den Schritten:

- 1 Geschäftsprozesse formulieren,
- 2 Szenarios aus den Geschäftsprozessen ableiten,
- 3 Interaktionsdiagramme aus den Szenarios ableiten,
- 4 Klassendiagramme erstellen und
- 5 Zustandsdiagramme erstellen.

daten-basiert

Der **daten-basierte Makroprozeß** ist empfehlenswert, wenn ein umfangreiches Datenmodell oder alte Datenbestände existieren und der Umfang der funktionalen Anforderungen zunächst unbekannt ist. Er ist auch dann zu wählen, wenn es sich um ein Auskunftssystem handelt, das später mehr oder weniger flexibel gestaltete Anfragen handhaben muß. Er umfaßt die Schritte:

- 1 Klassendiagramme erstellen,
- 2 Geschäftsprozesse formulieren,
- 3 Szenarios aus den Geschäftsprozessen ableiten,
- 4 Interaktionsdiagramme aus den Szenarios und dem Klassendiagramm ableiten und
- 5 Zustandsdiagramme erstellen.

**Checklisten**

Aufbau der  
Checklisten

Für jedes Konzept bietet meine Methode eine **Checkliste** mit methodischen Regeln an, die wie in Abb. 4.1-2 dargestellt aufgebaut ist. Für das Erstellen der entsprechenden Diagramme und Spezifikationen sollten Sie alle Fragen der Checkliste durchgehen. Sie können die gleichen Checklisten auch für die Inspektion benutzen. Hier sind dann nur die blau markierten Teile relevant.

Abb. 4.1-2:  
Aufbau der  
Checklisten

<b>Konstruktive Schritte</b>	Wie findet man ein Modellelement?
<b>Analytische Schritte</b>	Ist es ein «gutes» Modellelement? Konsistenzprüfung Fehlerquellen
<b>Für klassische Entwickler</b>	Welche methodischen Regeln helfen beim Übergang von der Datenmodellierung zur objektorientierten Analyse?

Das Erstellen eines OOA-Modells ist ein hochgradig kreativer Prozeß, der niemals nach einem starren Schema abläuft. Die Anwendung der Checklisten unterstützt den Analytiker, ohne ihn andererseits einzuengen. Verschiedene Entwickler können also durchaus unterschiedlich vorgehen. Sie sollten aber zu einem qualitativ gleichwertigen Modell kommen. Unabhängig von einer speziellen Vorgehensweise sollten Sie folgende Grundsätze beachten (vergleiche /Fowler 97a/):

- 1** Es gibt keine richtigen oder falschen Modelle. Es gibt nur Modelle, die mehr oder weniger gut ihren Zweck erfüllen.
- 2** Ein gutes Modell ist immer verständlich, d.h. es sieht einfach aus.
- 3** Die Erstellung verständlicher Modelle erfordert viel Aufwand.
- 4** Das Wissen von kompetenten Fachexperten ist absolut notwendig für ein gutes Modell.
- 5** Modellieren Sie kein System, das zu flexibel ist und zu viele Sonderfälle enthält. Diese Modelle sind aufgrund ihrer Komplexität immer schwer verständlich und damit schlechte Modelle.
- 6** Prüfen Sie für jeden Sonderfall, ob er es wert ist, die Komplexität des Modells und des zu realisierenden Systems zu erhöhen.

### Arbeitstechnik

Für einen zügigen Projektfortschritt und die Unterstützung der Kommunikation im Team ist es wichtig, schnell zu einer ersten Version des Modells zu gelangen, die dann kontinuierlich verbessert wird. Die ersten Modelle, die Sie entwickeln, werden wahrscheinlich weder besonders gut, noch in jedem Fall korrekt sein. Perfekte Ideen sind nicht plötzlich da, sie entwickeln sich.

*»If you wait for a complete and perfect concept to germinate in your mind, you are likely to wait forever.« DeMarco*

Modellieren Sie Ihre ersten Gedanken und überarbeiten Sie Ihr Modell so lange, bis ein wirklich gutes Modell entsteht. Diese Vorgehensweise wird durch folgende **Arbeitstechnik** unterstützt. Voraussetzung ist ein geeignetes objektorientiertes Werkzeug. Sobald eine Klasse identifiziert ist, wird ihr Name zusammen mit den wichtigsten Attributen und/oder Operationen eingetippt. »Wichtig« be-

Arbeitstechnik

## LE 6 4 Checklisten zum Erstellen eines OOA-Modells

deutet in diesem Zusammenhang, daß diese Attribute/Operationen zum Identifizieren der Klasse dienen. Ist das Vorliegen einer Klasse offensichtlich, kann auf die Angabe der Attribute und Operationen verzichtet werden. Wenn Ihnen nicht gleich der optimale Name für eine Klasse einfällt, so wählen Sie einen vorläufigen Arbeitstitel. Klassen, zwischen denen vermutlich eine Beziehung besteht, werden gleich in räumlicher Nähe angeordnet. Die Assoziation wird als einfache Linie eingetragen. Ein häufiger Fehler ist, zu diesem Zeitpunkt (zu) viel Zeit mit Diskussionen über Kardinalitäten zu verschwenden. Ganz wichtig für den Projektfortschritt in dieser frühen Phase ist, daß sich Ihre Diskussionen auf die fachlich korrekte Darstellung konzentrieren und nicht auf deren optimale Modellierung. Perfektion ist erst bei späteren Iterationen ein Ziel. Wählen Sie für jeden Iterationsschritt ein Ziel, das Sie nicht aus dem Auge verlieren dürfen. Das Ziel des ersten Schrittes ist, schnell ein Klassendiagramm zu erstellen. Drucken Sie dieses Diagramm (mit Versionsnummer und Datum) aus, kleben Sie die Seiten zusammen und heften Sie es an eine Wand, wo es alle Mitglieder des Teams einsehen können. Es stellt das Zentrum Ihres objektorientierten Projekts dar und bildet die Diskussionsgrundlage für weitere Änderungen und Erweiterungen. Führen Sie alle Änderungen handschriftlich in diesem Ausdruck durch. Bei entsprechend vielen handschriftlichen Änderungen müssen sie gebündelt mit dem Werkzeug dokumentiert werden. Anschließend ist sofort ein aktuelles Klassendiagramm auszudrucken, das dann das alte Diagramm ersetzt. Dieser Prozeß wird iteriert, bis eine stabile Version des Modells vorhanden ist, die einer formalen Inspektion unterworfen wird. Voraussetzung für die Erstellung eines OOA-Modells ist ein vorliegendes Pflichtenheft (Fachkonzept, Anforderungsspezifikation). Das von mir verwendete Gliederungsschema ist in Anhang 1 aufgeführt.

Anhang 1

Erfahrungen in der Praxis haben gezeigt, daß häufig die gleichen Fehler gemacht werden.

häufige Fehler

### 1 Das 100%-Syndrom

Das Problemverständnis muß nicht 100%ig sein, um mit dem Entwurf zu beginnen. In einer objektorientierten Architektur ist es leicht, fehlende Attribute und Operationen zu einem späteren Zeitpunkt zu ergänzen.

### 2 Zu frühe Qualitätsoptimierung

Oft diskutieren die Analytiker zu einem frühen Zeitpunkt Qualitätsverbesserungen. Änderungen des fachlichen Konzepts haben später dann neue Modellierungen zur Folge, durch die durchgeführte Qualitätsverbesserungen wieder gelöscht werden. Konzentrieren Sie sich daher im ersten Schritt immer auf das fachliche Konzept, unabhängig von der Qualität des Modells. Verbessern Sie dann im zweiten Schritt das fachlich korrekte Modell unter Gesichtspunkten eines optimalen OOA-Modells.

**3** Bürokratische Auslegung der Methode

Ich habe Teams viel Zeit mit Diskussionen verschwenden sehen, wie »Methoden-Guru XY« eine Aussage auf Seite z wohl gemeint haben könnte (*follow the spirit, not the letter of a method*).

**4** Entwurfskriterien in der Analyse berücksichtigen

Systemanalytikern, die zuvor jahrelang entworfen und programmiert haben, fällt oft die präzise Trennung von Analyse und Entwurf schwer.

## 4.2 Checkliste Geschäftsprozeß

Das Formulieren von Geschäftsprozessen bietet eine ausgezeichnete Möglichkeit, die Anforderungen an ein Softwaresystem besser zu verstehen. Konzentrieren Sie sich zunächst auf die primären Geschäftsprozesse, um ein Verständnis für den Kern des Systems zu erarbeiten. Die Anzahl der Geschäftsprozesse hängt stark vom jeweiligen Anwendungstyp ab. Bei einem umfangreichen System müssen Sie zuvor Teilsysteme (Pakete) bilden.

Arbeiten Sie zu einem Zeitpunkt immer nur an einem Geschäftsprozeß. Interviewen Sie die Benutzerrepräsentanten und die Experten des jeweiligen Fachgebiets. Geschäftsprozesse sollen so dokumentiert werden, daß sie sowohl für die Interviewten als auch für andere Analytiker verständlich sind. Das Geschäftsprozeßdiagramm dient dem besseren Überblick.

Formulieren Sie die Geschäftsprozesse zunächst auf einer hohen Abstraktionsebene und lassen Sie Sonderfälle außer acht. Die Verwendung der Schablone zwingt Sie dazu, einen Standardfall festzulegen und getrennt über mögliche Erweiterungen und Alternativen nachzudenken. Benennen Sie jeden Geschäftsprozeß möglichst aussagekräftig und präzise. Der Name soll ein Verb enthalten, das durch ein Substantiv ergänzt wird (Was wird gemacht? Womit wird etwas gemacht?). Wenn ausgedrückt werden soll, daß bestimmte Verarbeitungsschritte aus fachlicher Sicht parallel ausgeführt werden können, dann ist das Aktivitätsdiagramm anzuwenden.

Schablone

Geschäftsprozeß:

Ziel :

Kategorie:

Vorbedingung:

Nachbedingung Erfolg:

Nachbedingung Fehlschlag:

Akteure:

Auslösendes Ereignis:

Beschreibung:

1

2

## LE 6 4 Checklisten zum Erstellen eines OOA-Modells

Erweiterungen:

1a

Alternativen:

1a

Diagramm Um einen guten Überblick über die Geschäftsprozesse zu erhalten, sollten Sie parallel zu den Beschreibungen Geschäftsprozeßdiagramme erstellen. Für ein kleines System muß nur *ein* Diagramm erstellt werden. Für mittlere bis große Systeme sind mehrere Diagramme zu modellieren.

### Konstruktive Schritte zum Identifizieren von Geschäftsprozessen

**1** Wer ist der Akteur? Die Definition eines Akteurs legt eindeutig fest, daß sich Akteure immer außerhalb des betrachteten Systems befinden und mit den Geschäftsprozessen des Systems kommunizieren. Handelt es sich bei dem zu modellierenden System um ein Softwaresystem, dann ist der Akteur derjenige, der später die entsprechenden Aufgaben mit dem Softwaresystem durchführt. Obwohl diese Definition zunächst plausibel klingt, ist es in der Praxis nicht immer einfach, den Akteur zu ermitteln. Betrachten wir beispielsweise den Geschäftsprozeß kaufen einer Fahrkarte. Wird die Karte am Schalter erworben, dann ist der betreffende Sachbearbeiter der Akteur. Es könnte sich jedoch auch um einen Fahrkartenautomaten handeln, an dem der Fahrgast (Akteur) sich seine Fahrkarte selbständig zieht. Bei diesem Beispiel läßt sich der Akteur daher nur dann ermitteln, wenn Informationen über den Einsatz des Gesamtsystems bekannt sind.

Die Identifikation von Akteuren sollte unter folgenden Gesichtspunkten erfolgen:

- a** Welche Personen führen diese Aufgaben zur Zeit durch und besitzen daher wichtige Kenntnisse über die durchzuführenden Arbeitsabläufe? Welche Rollen spielen diese Personen?
- b** Welche Personen werden zukünftig diese Aufgaben durchführen und auf welche Vorkenntnisse muß die Benutzungsoberfläche abgestimmt werden? Welche Rollen spielen diese Personen?
- c** Wo befindet sich die Schnittstelle des betrachteten Systems bzw. was gehört nicht mehr zu dem System?

Es gibt mehrere Möglichkeiten, um Geschäftsprozesse zu identifizieren. Welcher Weg der jeweils beste ist, variiert je nach dem Typ der Anwendung (vergleiche /Hruschka 98/).

**2** zuerst Standardfälle Ein typischer Fehler beim Erstellen von Geschäftsprozessen ist, sich in einer Flut von Sonderfällen und Details zu verlieren, die den Blick für das Wesentliche versperren. Konzentrieren Sie sich zunächst bei allen Geschäftsprozessen im ersten Schritt auf die Standardfälle und lassen Sie Sonderfälle konsequent außer acht.

/Jacobson 92/ empfiehlt, von den Akteuren des Systems auszugehen. Handelt es sich um Personen, dann analysieren Sie deren typische Arbeitsabläufe. Stellen Sie in Interviews folgende Fragen:

- Welches Ereignis löst den Arbeitsablauf aus?
- Welche Eingabedaten werden benötigt?
- Welche Schritte sind auszuführen?
- Ist eine Reihenfolge der Schritte festgelegt?
- Welche Zwischenergebnisse werden erstellt?
- Welche Endergebnisse werden erstellt?
- Welche Vorbedingungen müssen erfüllt sein?
- Welche Nachbedingungen (Vorbedingungen anderer Geschäftsprozesse) werden sichergestellt?
- Wie wichtig ist diese Arbeit?
- Warum wird diese Arbeit durchgeführt?
- Kann die Durchführung verbessert werden?

In der Seminarorganisation ist ein Akteur der Kundensachbearbeiter, dessen Aufgabe es ist, Anmeldungen von Kunden anzunehmen und zu bearbeiten. Daraus läßt sich der Geschäftsprozeß bearbeiten ableiten.

Wenn es sich bei den Akteuren nicht um Personen handelt, sondern beispielsweise um organisatorische Einheiten oder technische Schnittstellen, dann sollten Sie eine Ereignisliste erstellen. Überlegen Sie, welche Ereignisse der Umgebung für das System relevant sind. Für jedes Ereignis muß ein Geschäftsprozeß (GP) existieren, der darauf reagiert bzw. entdeckt, daß ein entsprechendes Ereignis vorliegt. Es lassen sich externe Ereignisse und zeitliche Ereignisse unterscheiden. Externe Ereignisse treten außerhalb des betrachteten Systems auf. Zeitliche Ereignisse werden im allgemeinen im System produziert.

Für die Seminarorganisation kann eine Ereignisliste beispielsweise folgende Punkte enthalten:

- Seminaranmeldung trifft ein (→GP bearbeiten Anmeldung).
- Dozent sagt wegen Krankheit ab (→GP suche Ersatz oder storniere Seminar).
- Teilnehmer sagt ab (→GP bearbeiten Absage).
- Seminar durchgeführt (zeitliches Ereignis) (→GP erstellen Rechnungen).

Lassen sich weder mittels Akteuren noch mittels Ereignissen Geschäftsprozesse identifizieren, dann beschreiben Sie die Aufgaben des Systems. Formulieren Sie zunächst den Zweck bzw. die Ziele des Systems. Leiten Sie aus diesen Zielen die notwendigen Aufgaben ab. Überlegen Sie, welches die zehn wichtigsten Aufgaben sind. Beschreiben Sie jede Aufgabe umgangssprachlich mit 25 – oder weniger – Wörtern. Beantworten Sie dazu die Frage: Was ist das Ziel



## LE 6 4 Checklisten zum Erstellen eines OOA-Modells

dieser Aufgabe bzw. der Nutzen dieser Aufgabe für das Gesamtsystem?

**3 Sonderfälle** Wenn die Standardfälle der Geschäftsprozesse erstellt sind, modellieren Sie im zweiten Schritt die Erweiterungen und Sonderfälle, die nur unter bestimmten Bedingungen auftreten.

Erweiterungen sind beispielsweise

- optionale Teile eines Geschäftsprozesses,
- komplexe oder alternative Möglichkeiten und
- Aufgaben, die nur selten durchgeführt werden.

Der Vorteil dieser Vorgehensweise besteht darin, daß die Basisfunktionalität leicht zu verstehen ist und erst im zweiten Schritt die Komplexität in das System integriert wird. Sonderfälle können bei Verwenden der Schablone unter *Erweiterungen* und *Alternativen* aufgeführt werden. Umfangreiche Sonderfälle sollten als eigenständige Geschäftsprozesse spezifiziert und mit *extends* an die Standardverarbeitung angebunden werden.

**Beispiel** Bei einer Versicherungsgesellschaft ist ein Schadensfall zu bearbeiten (vgl. /Cockburn 97/).

*Geschäftsprozeß:* bearbeite Schadensfall

*Ziel:* Bezahlung des Schadens durch die Versicherung

*Kategorie:* primär

*Vorbereitung:* -

*Nachbedingung Erfolg:* Schaden ganz oder teilweise bezahlt

*Nachbedingung Fehlschlag:* Forderung abgewiesen

*Akteure:* Schadenssachbearbeiter

*Auslösendes Ereignis:* Schadensersatzforderung des Antragstellers, d. h. der versicherten Person

*Beschreibung:*

- 1 Der Sachbearbeiter prüft die Forderung auf Vollständigkeit.
- 2 Der Sachbearbeiter prüft, ob eine gültige Police vorliegt.
- 3 Der Sachbearbeiter prüft alle Details der Police.
- 4 Der Sachbearbeiter errechnet den Betrag und überweist ihn an den Antragsteller.

*Erweiterungen:*

- 1a Die vorliegenden Daten vom Antragsteller sind nicht vollständig. Dann muß der Sachbearbeiter diese Informationen nachfordern.
- 2a Der Antragsteller besitzt keine gültige Police. Der Sachbearbeiter teilt ihm mit, daß keine Ansprüche bestehen und schließt den Fall ab.
- 4a Der Schaden wird durch die Police nicht abgedeckt. Der Sachbearbeiter teilt dies dem Antragsteller mit und schließt den Fall ab.
- 4b Der Schaden wird durch die Police nur unvollständig abgedeckt. Der Sachbearbeiter verhandelt mit dem Antragsteller, bis zu welchem Grad der Schaden bezahlt wird.

*Alternativen:* -

**4 Aufsplitten** Im allgemeinen besteht ein Geschäftsprozeß aus mehreren Teilaufgaben. Unter Umständen ist ein einzelner Schritt eines Ge-

schäftsprozesses, der auf einer hohen Abstraktionsebene formuliert wurde, so komplex, daß er selbst ein Geschäftsprozeß ist. Diesen Zusammenhang notieren Sie im Diagramm mittels der *uses*-Beziehung.

Wenn bei der Beschreibung eines Geschäftsprozesses zu viele Sonderfälle auftreten, sollten Sie prüfen, ob Sie das Verhalten nicht besser durch mehrere Geschäftsprozesse beschreiben können, wobei gemeinsames Verhalten mittels der *uses*-Beziehung modelliert wird. Sind die Erweiterungen eines Geschäftsprozesses sehr umfangreich, so sollten Sie einen eigenen Geschäftsprozeß spezifizieren, der mittels *extends* mit dem Standard-Geschäftsprozeß verbunden wird.

Analysieren Sie mehrere Geschäftsprozesse auf gemeinsames Verhalten. Besitzen zwei Geschäftsprozesse einen gemeinsamen Teil, dann ist dieser herauszulösen und mit *uses* zu verknüpfen. Die *uses*-Beziehung dient in erster Linie der redundanzfreien Beschreibung von Geschäftsprozessen. Sie können *uses*-Beziehungen auch als eine Art Funktionsaufruf betrachten.

5 Gemeinsamkeiten

Mit der *uses*-Beziehung kann die funktionale Zerlegung eines Systems beschrieben werden. Achten Sie darauf, daß Sie Geschäftsprozesse nicht zu stark verfeinern und eine Art Funktionsbaum erstellen. Das ist nicht der Sinn dieses Konzepts.

Was ist der Unterschied zwischen den Geschäftsprozessen und der klassischen funktionalen Zerlegung? Geschäftsprozesse beschreiben die mit dem System auszuführenden Arbeitsabläufe, d.h. welche Aufgaben mit dem System durchgeführt werden können. Diese Aufgaben werden meistens vom Softwaresystem ausgeführt, können aber auch organisatorischer Natur sein. Die klassische funktionale Zerlegung gibt an, welche Funktionen das System – unabhängig von den jeweiligen Arbeitsabläufen – zur Verfügung stellt. Außerdem unterscheiden sich die beiden Konzepte ganz wesentlich im Abstraktionsniveau, auf dem die Zerlegung endet. Vermeiden Sie eine zu detaillierte Beschreibung von Geschäftsprozessen. Sie dienen als *high level documentation* des Systemverhaltens und werden durch Szenarios und Operationen verfeinert. Geschäftsprozesse sind ein reines Analysekonzept, denn sie beschreiben das System aus Sicht der zukünftigen Benutzer und werden in einem späteren Schritt auf die Operationen des Systems abgebildet. Die klassische funktionale Zerlegung kann dagegen sowohl in der Analyse als auch im Entwurf eingesetzt werden.

Geschäftsprozeß vs. Funktion

Ein Geschäftsprozeß beschreibt immer einen kompletten Ablauf von Anfang bis Ende. Er besteht daher im allgemeinen aus mehreren Schritten oder Transaktionen. Jeder Schritt kann einen weiteren Geschäftsprozeß oder eine Operation (z.B. drucke Rechnung) darstellen. Im Extremfall kann ein Geschäftsprozeß auf eine einzige Operation abgebildet werden.

## LE 6 4 Checklisten zum Erstellen eines OOA-Modells

Anzahl Geschäftsprozesse

Die folgenden Angaben zeigen, daß die Anzahl der Geschäftsprozesse nicht linear mit dem Entwicklungsaufwand zunimmt, sondern von verschiedenen Faktoren bestimmt wird. Dazu gehören das gewählte Abstraktionsniveau und der Anwendungsbereich. Nach /Jacobson 95/ besteht ein kleineres System (zwei bis fünf Mitarbeiterjahre) aus 3 bis 20 Geschäftsprozessen (*use cases*). Ein mittleres System (10 bis 100 Mitarbeiterjahre) kann 10 bis 60 Geschäftsprozesse enthalten. Größere Systeme, z.B. Anwendungen für Banken, Versicherungen, Verteidigung und Telekommunikation können Hunderte von Geschäftsprozessen enthalten. /Booch 96/ erwartet bei einem Projekt mittlerer Komplexität etwa ein Dutzend Geschäftsprozesse. /Cockburn 97/ gibt folgende Größen an: Ein Projekt von 50 Mitarbeiterjahren mit 50 Geschäftsprozessen und ein Projekt mit 30 Mitarbeiterjahren (18 Monate Entwicklungsdauer) und mit 200 Geschäftsprozessen.

### Analytische Schritte zum Validieren der Geschäftsprozesse

6 »gute«  
Beschreibung

Formulieren Sie die Beschreibung der Geschäftsprozesse so, daß Ihr Auftraggeber sie lesen und verstehen kann. Konzentrieren Sie sich auf die Kommunikation der Akteure mit dem System und beschreiben Sie weder die interne Struktur noch die Algorithmen. Achten Sie darauf, daß der Standardfall immer komplett spezifiziert ist. Eine Beschreibung sollte maximal eine Seite umfassen.

7 Konsistenz

Wenn – zu einem späteren Zeitpunkt des Makroprozesses – das Klassendiagramm vorliegt, dann sollten Sie prüfen, ob die Geschäftsprozesse konsistent mit dem Klassendiagramm sind. Erstellen Sie dazu für jeden Geschäftsprozeß ein Objektdiagramm. Kennzeichnen Sie die Objekte und Verbindungen, die vor Durchführung des Geschäftsprozesses vorhanden waren.

- Für jede Klasse soll mindestens ein Objekt erzeugt werden.
- Für jede Assoziation soll mindestens eine Verbindung aufgebaut werden.
- Wenn für Assoziationen Restriktionen modelliert sind, sollen diese Restriktionen vollständig durch die Geschäftsprozesse abgedeckt werden.

Beispiel Wir verwenden aus Kapitel 3.3 den Geschäftsprozeß Zulassung eintragen und das Klassendiagramm des Arztregisters mit Historie. Die Registrierung des Arztes führt zu den Objekten :Arzt, :Registrierung, :Wohnung, :Zulassung, :Einzelpraxis

Abb. 4.2-1:  
Objektdiagramm  
zum Geschäfts-  
prozeß Zulassung  
eintragen



ung und :Wohnung. Beim Eintragen der Zulassung und der Einzelpraxis werden diese Objekte erzeugt. Alternativ zur Einzelpraxis kann eine Verbindung zu einem bereits vorhandenen Gemeinschaftspraxis-Objekt erstellt werden.

In der Tabelle 4.2-1 sind typische Fehlerquellen aufgeführt, die beim Identifizieren von Geschäftsprozessen auftreten. Insbesondere ist darauf zu achten, daß ein Geschäftsprozeß keine Dialogsteuerung beschreibt, auch wenn der Begriff *use case (the way in which a user uses a system)* dies vielleicht suggeriert. Dadurch würde die Trennung zwischen Fachkonzept und Benutzungsoberfläche verloren gehen.

### Ergebnisse

#### ■ Geschäftsprozeßdiagramm

Alle Geschäftsprozesse und Akteure werden eingetragen.

#### ■ Beschreibung der Geschäftsprozesse

Alle Geschäftsprozesse sind umgangssprachlich oder mittels Schablone zu beschreiben.

### Konstruktive Schritte

#### 1 Akteure ermitteln

- Welche Personen führen diese Aufgaben durch?
- Welche Schnittstellen besitzt das System?

#### 2 Geschäftsprozesse für die Standardverarbeitung ermitteln

- Primäre und ggf. sekundäre Geschäftsprozesse betrachten.
- Welche Standardverarbeitung besitzen sie?

#### 2a mittels Akteuren

- Sind die Akteure Personen?
- Welche Arbeitsabläufe lösen sie aus?
- An welchen Arbeitsabläufen wirken sie mit?

#### 2b mittels Ereignissen (Akteure sind externe Systeme)

- Erstellen Sie eine Ereignisliste.
- Identifizieren Sie für jedes Ereignis einen Geschäftsprozeß.
- Unterscheiden Sie externe und zeitliche Ereignisse.

#### 2c mittels Aufgabenbeschreibungen

- Was sind die Gesamtziele des Systems?
- Welches sind die zehn wichtigsten Aufgaben?
- Was ist das Ziel jeder Aufgabe?

#### 3 Geschäftsprozesse für die Sonderfälle formulieren

- Erweiterungen und Alternativen mittels Schablone erstellen.
- Aufbauend auf Standardfunktionalität mit *extends* die Sonderfälle formulieren, d.h. erweiterte Geschäftsprozesse beschreiben.

#### 4 Aufsplitten komplexer Geschäftsfälle

- Komplexe Schritte als Geschäftsprozesse spezifizieren (*uses*).
- Komplexe Geschäftsprozesse (viele Sonderfälle) in mehrere Geschäftsprozesse zerlegen und Gemeinsamkeiten mit *uses* modellieren.
- Umfangreiche Erweiterungen als Geschäftsprozesse spezifizieren (*extends*).

**Tab. 4.2-1a:**  
**Checkliste**  
**Geschäfts-**  
**prozesse**

**Tab. 4.2-1b:**  
**Checkliste**  
**Geschäfts-**  
**prozesse**

**5 Gemeinsamkeiten von Geschäftsprozessen ermitteln**

- Auf redundanzfreie Beschreibung achten (*uses*).

**Analytische Schritte**

**6 »gute« Beschreibung**

- verständlich für den Auftraggeber.
- extern wahrnehmbares Verhalten.
- fachliche Beschreibung des Arbeitsablaufs.
- beschreibt Standardfall vollständig und Sonderfälle separat.
- maximal eine Seite.

**7 Konsistenz mit Klassendiagramm**

- Objektdiagramm erstellen.

**8 Fehlerquellen**

- Zu kleine und damit zu viele Geschäftsprozesse.
- Zu frühe Betrachtung von Sonderfällen.
- Zu detaillierte Beschreibung der Geschäftsprozesse.
- Verwechseln von *uses*- und *extends*-Beziehungen.
- Geschäftsprozesse beschreiben Dialogabläufe.

### 4.3 Checkliste Paket

**Konstruktive Schritte zum Identifizieren von Paketen**

**1 top down** Bei großen Systemen muß – noch vor der Formulierung von Geschäftsprozessen – das Gesamtsystem in Pakete unterteilt werden. Ein Paket entspricht bei der klassischen Verarbeitung einem in sich abgeschlossenen Teilsystem. Umfangreiche Pakete sind in weitere Pakete zu zerlegen.

**2 bottom up** Während bei kleineren Systemen ganz auf Pakete verzichtet werden kann, sollten Sie bei Systemen mittlerer Größenordnung nach der Erstellung der Geschäftsprozesse oder spätestens nach der statischen Modellierung Pakete bzw. Teilsysteme bilden.

**Analytische Schritte zum Validieren der Pakete**

**3 abgeschlossene Einheit** Ein in sich abgeschlossenes Paket besitzt folgende Eigenschaften:

- Es führt den Leser durch das System.
- Es enthält einen Themenbereich, der für sich allein betrachtet und verstanden werden kann, oder es ist mit minimalen Bezügen zu anderen Paketen verständlich.
- Es besteht nicht einfach aus einer Menge von Modellelementen, sondern Pakete sollen eine Betrachtung des Systems auf höherer Abstraktionsebene ermöglichen. Es muß daher Geschäftsprozesse und Klassen zusammenfassen, die dem gleichen Themenbereich angehören.

In sich abgeschlossene Pakete unterstützen die Arbeitsteilung, denn jedes Paket bildet eine Arbeitseinheit für ein Team. Um die Abgeschlossenheit zu erreichen, sollen Vererbungsstrukturen möglichst innerhalb eines Pakets liegen.

Der Paketname soll auf einem höheren Abstraktionsniveau beschreiben, was der Inhalt dieses Pakets ist. Beschreiben Sie dazu den Inhalt dieses Pakets mit 25 Wörtern oder weniger. Versuchen Sie dann diese Kurzbeschreibung auf einen einzigen Begriff zu reduzieren.

4 Paketname

Damit mittels Paketen eine Betrachtung des Systems auf höherer Abstraktionsebene möglich ist, sollen Pakete nicht zu klein sein. Sonst entsteht ein Modell, das schwer lesbar ist und eine nicht vorhandene Komplexität vortäuscht. Ein Grund für diesen Fehler liegt vermutlich darin, daß in Büchern oft Beispiele von sehr kleinen Paketen angegeben sind. Das ist oft notwendig, um Beispiele und Aufgaben in angemessener Zeit zu bewältigen, darf aber nicht als Kriterium auf eine »echte« Anwendung übertragen werden.

5 zu kleine Pakete

## Ergebnisse

### ■ Paketdiagramme

Erstellen Sie ein oder mehrere Paketdiagramme. Ordnen Sie jedem Paket Modellelemente zu und spezifizieren Sie Abhängigkeiten zwischen den Paketen.

### Konstruktive Schritte

#### 1 Welche Pakete ergeben sich durch *top-down*-Vorgehen?

Bei großen Anwendungen:

- Unterteilen Sie das Gesamtsystem in Teilsysteme (Pakete).
- Zerlegen Sie umfangreiche Pakete in weitere Pakete.

#### 2 Welche Pakete ergeben sich durch *bottom-up*-Vorgehen?

Bei kleinen und mittleren Anwendungen:

- Fassen Sie Klassen unter einem Oberbegriff zusammen.

### Analytische Schritte

#### 3 Bildet das Paket eine abgeschlossene Einheit?

- Es enthält einen Themenbereich, der für sich allein betrachtet und verstanden werden kann.
- Es erlaubt eine Betrachtung des Systems auf einer höheren Abstraktionsebene.
- Vererbungsstrukturen liegen möglichst innerhalb eines Pakets.

#### 4 Ist der Paketname geeignet?

- Beschreiben Sie den Inhalt eines Pakets mit 25 Wörtern oder weniger.
- Leiten Sie daraus den Namen ab.

#### 5 Fehlerquellen

- Zu kleine Pakete.

**Tab. 4.3-1**  
**Checkliste**  
**Pakete**

## LE 6 Glossar/Zusammenhänge/Aufgaben

**Analyseprozeß** Der Analyseprozeß beschreibt die methodische Vorgehensweise zur Erstellung eines objektorientierten Analysemodells. Er besteht aus einem →Makroprozeß, der die grundlegenden Vorgehensschritte vorgibt und der situations- und anwendungsspezifischen Anwendung von methodischen Regeln.

**Balancierter Makroprozeß** Der balancierte →Makroprozeß unterstützt die Gleichgewichtigkeit von statischem und dynamischem Modell. Er beginnt mit dem Erstellen von →Geschäftsprozessen und der Identifikation von Klassen. Dann werden statisches und dynamisches Modell parallel erstellt und deren Wechselwirkungen berücksichtigt.

**Daten-basierter Makroprozeß** Beim daten-basierten →Makroprozeß wird zunächst das Klassendiagramm erstellt und aufbauend darauf werden die →Geschäftsprozesse und die anderen Diagramme des dynamischen Modells entwickelt.

**Geschäftsprozeß (use case)** Ein Geschäftsprozeß (*use case*) besteht aus

mehreren zusammenhängenden Aufgaben, die von einem Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.

**Makroprozeß** Der Makroprozeß beschreibt auf einem hohen Abstraktionsniveau die einzelnen Schritte, die zur systematischen Erstellung eines OOA-Modells durchzuführen sind. Der Makroprozeß kann die Gleichgewichtigkeit von statischem und dynamischem Modell (balancierter Makroprozeß) unterstützen oder →daten-basiert bzw. →szenario-basiert sein.

**Paket (package)** Ein Paket faßt Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene auszudrücken. Pakete können im Paketdiagramm dargestellt werden.

**Szenario-basierter Makroprozeß**

Der szenario-basierte →Makroprozeß beginnt mit dem Erstellen von →Geschäftsprozessen und Interaktionsdiagrammen und leitet daraus das Klassendiagramm ab.



Der **Analyseprozeß** besteht aus einem **Makroprozeß**, der die Gleichgewichtigkeit von statischem und dynamischem Modell berücksichtigt, und aus einheitlich aufgebauten Checklisten für jedes objektorientierte Konzept. Die Checkliste **Geschäftsprozeß** zeigt, wie Geschäftsprozesse ermittelt werden, wie *uses/extends* eingesetzt werden und was eine gute Beschreibung ausmacht. Die Checkliste **Paket** beschreibt, wie Pakete gebildet werden und deren Güte geprüft wird.



Aufgabe  
15–20 Minuten

**1 Lernziel: Wissen über den Analyseprozeß prüfen.**

- a** Erläutern Sie, was evolutionäre Vorgehensweise bedeutet.
- b** Welche methodischen Schritte können unabhängig davon durchgeführt werden, ob objektorientiert oder klassisch entwickelt wird?
- c** Sie erhalten die Aufgabe, ein 20 Jahre altes Informationssystem neu zu entwickeln, d.h. ein Re-Engineering-Projekt durchzuführen. Für Ihre Arbeit erhalten Sie das ablauffähige System, die Benutzerhandbücher und die Dateibesreibungen. Außerdem stehen Ihnen die Benutzer des alten Systems für Interviews zur Verfügung. Wie gehen Sie vor?
- d** Welche Vorteile besitzen die hier vorgestellten Checklisten für die Entwicklung und die Qualitätssicherung?



**2 Lernziel:** *Einen Geschäftsprozeß dokumentieren können.*Aufgabe  
15 Minuten

Für das Fallbeispiel der Friseursalonverwaltung aus Kapitel 3.4 ist der Geschäftsprozeß für das Kassieren von Verkäufen bei Laufkundschaft zu beschreiben. Verwenden Sie die Schablone für Geschäftsprozesse.

**3 Lernziel:** *Mehrere Geschäftsprozesse mittels Ereignisliste identifizieren können.*Aufgabe  
15 Minuten

Stellen Sie die Ereignisliste für nachfolgende Problembeschreibung auf, identifizieren Sie Geschäftsprozesse und erstellen Sie für jeden Geschäftsprozeß eine kurze informale Beschreibung. Übernehmen Sie alle Informationen des Textes in die Beschreibungen.

Eine Bibliothek ist zu verwalten. Jeder registrierte Leser kann sich Bücher ausleihen. Ist ein gewünschtes Buch nicht vorhanden, so kann es von den Lesern vorbestellt werden. Ein Buch kann zu einem Zeitpunkt von mehreren Lesern vorbestellt sein, d.h. es wird eine Warteliste gebildet. Wird ein vorbestelltes Buch zurückgegeben, dann ist der erste Leser auf der Warteliste zu benachrichtigen. Reservierte Bücher, die nach eine Woche nicht abgeholt wurden, werden wieder zur Ausleihe bereitgestellt oder der nächste Leser der Warteliste wird informiert. Bei der Aufnahme in die Bibliothek erhält jedes Buchexemplar eine eindeutige Inventarnummer. Für jeden Leser werden der Name und die Adresse gespeichert. Bei der Ausleihe werden das Ausleihdatum und das Rückgabedatum gespeichert. Bei allen Büchern, deren Ausleihfrist um eine Woche überschritten ist, werden deren Leser automatisch gemahnt.

**4 Lernziel:** *Mehrere Geschäftsprozesse identifizieren und dokumentieren können.*Aufgabe  
40 Minuten

Stellen Sie für folgende Problembeschreibung die Geschäftsprozesse auf. Beschreiben Sie jeden Geschäftsprozeß mittels Schablone und erstellen Sie ein Geschäftsprozeßdiagramm.

Die Veranstaltungen und Prüfungen einer Hochschule sollen verwaltet werden.

Jede Veranstaltung gehört zu einem bestimmten Typ. Einige Veranstaltungstypen sind Pflicht, d.h. Voraussetzung für die Anmeldung zur Diplomarbeit. Eine Veranstaltung wird an einem bestimmten Wochentag, zu einer festgelegten Zeit in einem festgelegten Raum eingetragen. Sie wird von genau einem Dozenten durchgeführt. Nach einem vorliegenden Terminplan muß von jedem Fachbereich ein Veranstaltungsplan für das nächste Semester erstellt werden.

Für alle durchzuführenden Prüfungen wird vom Prüfungsausschuß in jedem Semester ein aktueller Prüfungsplan erstellt. Jede



## LE 6 4 Checklisten zum Erstellen eines OOA-Modells

Prüfung findet zu einer festgelegten Zeit in einem dafür reservierten Raum statt. Sie bezieht sich immer auf einen Prüfer.

Eine Prüfung bezieht sich auf genau einen Veranstaltungstyp. Für die Prüfung, die sich auf einen bestimmten Veranstaltungstyp bezieht, kann als Teilnahmevoraussetzungen das Bestehen von Prüfungen anderer Veranstaltungstypen gefordert werden.

Studenten müssen sich beim Prüfungssekretariat mit einem Formular für die Prüfungen anmelden. In dieses Formular müssen Matrikelnummer und der Name des Studenten sowie die Nummern aller gewünschten Prüfungen eingetragen werden. Wenn das Prüfungssekretariat festgestellt hat, daß alle Teilnahmevoraussetzungen erfüllt sind, wird der Student in die entsprechenden Zulassungslisten eingetragen.

14 Tage vor einer jeden Prüfung erstellt das System automatisch eine Liste aller zugelassenen Studenten für den Prüfer. Außer den obigen Angaben enthält diese Liste Informationen über die Anzahl der Prüfungsversuche. Nach Durchführung der Prüfung trägt der Prüfer die Ergebnisse in diese Liste ein und gibt sie dem Prüfungssekretariat zurück, das die Angaben überprüft und eine Ergebnisliste mit Matrikelnummern und Noten veröffentlicht.

Jeder Student muß ein Praktikum nachweisen, das bei einer Firma durchgeführt wird.

Ein Student schließt sein Studium mit einer Diplomarbeit ab. Bei der Beantragung wird geprüft, ob er für alle Pflichtveranstaltungen die Prüfungen bestanden hat. Nach Abschluß der Arbeit wird vom Betreuer die Note eingetragen.

Hinweis: Diese Aufgabe wird in der nächsten Lehreinheit weiter bearbeitet (Erstellen des Klassendiagramms). Die obige Problembeschreibung enthält daher eine Reihe von Informationen, die für die Lösung in dieser Lehreinheit noch nicht benötigt werden.

Aufgabe  
5–10 Minuten

### 5 Lernziel: Identifizieren von Paketen.

Folgende Geschäftsprozesse wurden bei einem System für den Versandhandel erstellt. Welche Pakete können Sie bilden?

- Auswerten von Informationen der Lieferanten, um neue Kataloge zu erstellen
- Auswertung von Sonderwünschen für das Marketing
- Bearbeiten von Kundenaufträgen laut Katalog
- Bearbeiten von Kundenaufträgen laut Katalog mit Nachlieferungen
- Bearbeiten von Sonderwünschen der Kunden
- Ermittlung von Informationen für das Marketing (Penner-Renner-Liste)
- Erstellen von Bestellungen an Lieferanten, um gängige Artikel am Lager zu haben

## **Aufgaben LE 6**

- Erstellen von Bestellungen an Lieferanten, um Kundenaufträge zu erfüllen
- Informieren der Kunden über neue Produkte
- Versenden von Probesendungen an gute Kunden
- Weitergabe aller Aufträge an die Buchhaltung
- Weitergabe aller Bestellungen an die Buchhaltung

## 4 Checklisten zum Erstellen eines OOA-Modells (Statisches Modell)



- Klassen systematisch identifizieren können.
- Assoziationen systematisch identifizieren und spezifizieren können.
- Attribute systematisch identifizieren können.
- Vererbungsstrukturen systematisch identifizieren können.
- Systematisch ein Klassendiagramm (ohne Operationen) erstellen können.
- Beurteilen können, ob ein »gutes« Klassendiagramm erstellt wurde.

verstehen

anwenden



- Die objektorientierten Konzepte und die UML-Notation, wie sie in Kapitel 2 beschrieben werden, müssen bekannt sein.
- Die Kenntnisse der Kapitel 1 und 3.1 sind nützlich.
- Aus dem Kapitel 3 sollten Sie mindestens ein Fallbeispiel durchgearbeitet haben.
- Der Inhalt der Kapitel 4.1 bis 4.3 muß bekannt sein.

- i 4.4 [Checkliste Klasse](#) 142
- 4.5 [Checkliste Assoziation](#) 147
- 4.6 [Checkliste Attribut](#) 157
- 4.7 [Checkliste Vererbung](#) 162

## 4.4 Checkliste Klasse

Nach dem Formulieren der Geschäftsprozesse ist für den aktuellen Entwicklungszyklus das Klassendiagramm zu erstellen. Dieses Klassendiagramm enthält als **statisches Modell** außer den Klassen deren Attribute sowie Assoziationen und Vererbungsstrukturen. Die Operationen werden erst nach der Erstellung des dynamischen Modells hinzugefügt. Anstelle eines globalen Klassendiagramms kann es sinnvoll sein, für jeden Geschäftsprozeß oder eine Gruppe von zusammengehörenden Geschäftsprozessen ein separates Klassendiagramm zu erstellen. Geschäftsprozesse können auf diese Weise benutzt werden, um ein großes Klassendiagramm in mehrere handliche Teile zu strukturieren. Jedes Klassendiagramm enthält dabei eine bestimmte Sicht des Gesamtsystems.

Durch die Bildung der Klassen wird die entscheidende Abstraktionsebene für die gesamte Modellbildung erstellt. Auf den Punkt gebracht kann man auch sagen, daß die Vorteile der Objektorientierung dann am größten sind, wenn die »richtigen« Klassen identifiziert werden.

Die Ausgangsbasis für das Identifizieren der Klassen bilden die Beschreibungen der Geschäftsprozesse. In vielen Fällen existieren für einen Arbeitsablauf Formulare, Listen und andere Dokumente. Handelt es sich um das *Re-Engineering* eines alten Softwaresystems, dann gibt es auch Benutzerhandbücher, Bildschirmmasken und Dateibesreibungen. Anhand des laufenden Systems sollten Sie die Funktionen des betreffenden Geschäftsprozesses ausführen und die Klassen mit Hilfe der Bildschirmmasken ermitteln.

### Konstruktive Schritte zum Identifizieren von Klassen

- 1** Dokumentanalyse  
Besonders einfach lassen sich **Klassen** mittels der Dokumentanalyse identifizieren. Die Dokumentanalyse ist eine Weiterentwicklung der Formularanalyse, die aus dem Bereich der semantischen Datenmodellierung /Vetter 87/ stammt. Die Dokumente enthalten **Attribute**, die mittels *bottom-up*-Vorgehen zu Klassen zusammengefaßt werden. Wählen Sie den Klassennamen nach der Gesamtheit der Attribute. Da die Dokumentanalyse auch zum Identifizieren von Assoziationen dient, werden meist gleichzeitig mit den Klassen Assoziationen ermittelt.

**Beispiel** Aus dem Formular zur Seminaranmeldung lassen sich mittels Dokumentanalyse die Klassen Teilnehmer, Seminar und Rechnungsempfänger ableiten (Abb. 4.4-1).

- 2** Beschreibung Geschäftsprozesse  
Aus der Beschreibung des Geschäftsprozesses können Sie mittels der *top-down*-Vorgehensweise Klassen ableiten. Gehen Sie den Text durch und durchsuchen Sie ihn nach Klassen. Oft sind die Substantive potentielle Klassen. Ebenso kann sich eine Klasse hinter Verben

Anmeldung zu TEACHWARE-Seminaren

Als Teilnehmer zu nachfolgenden TEACHWARE-Seminaren wird angemeldet:

_____	_____	_____
Titel	Vorname	Nachname
_____	_____	_____
Seminar-Nr.	Seminarbez.	vom - bis
_____	_____	_____
_____	_____	_____

Anmeldebestätigung und Rechnung erbeten an:

_____	_____	_____
Titel	Vorname	Name
_____	_____	_____
Firma	Str./Postfach	
_____	_____	
LKZ	PLZ	Ort
_____	_____	_____
Telefon		

Annotations (blue boxes with arrows):

- Teilnehmer (points to the first row of the first table)
- Seminar (points to the second and third rows of the first table)
- Rechnungsempfänger (points to the first row of the second table)

Abb. 4.4-1:  
Beispiel zur  
Dokumentanalyse

verbergen. Eine Klasse läßt sich relativ leicht durch ihre Attribute identifizieren. Der Erfolg dieser Methode wird entscheidend durch die Sicherheit des Systemanalytikers im Erkennen der potentiellen Klassen bestimmt.

#### Beispiel: Seminarorganisation

Beispiel

##### *Geschäftsprozeß:* Anmelden eines neuen Teilnehmers

Interessenten melden sich schriftlich an. Der betreffende Interessent und die gebuchten Seminare werden in die Kundenkartei aufgenommen. Alle Seminartypen und Seminare sind in der Seminar-kartei gespeichert. Die Seminargebühr ist für alle Seminare eines Typs gleich. Einige Kunden können zu einer ermäßigten Gebühr teilnehmen, die jeweils individuell festgelegt wird. Jede Anmeldung wird von den Mitarbeitern schriftlich bestätigt.

##### *Geschäftsprozeß:* Absagen eines gebuchten Seminars

Bereits gebuchte Seminarveranstaltungen können durch die Kunden zu folgenden Bedingungen abgesagt werden: Bei einer Absage bis zu drei Wochen vor Seminarbeginn entstehen keine Kosten, bei einer späteren Absage werden 50 Prozent der Seminargebühr in Rechnung gestellt. Jede Absage wird schriftlich bestätigt. Wenn der Kun-

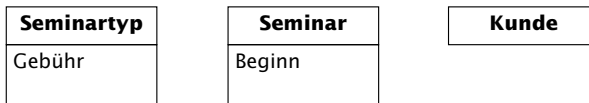
## LE 7 4 Checklisten zum Erstellen eines OOA-Modells

de nicht rechtzeitig absagt, erhält er nach der Veranstaltung eine Rechnung, obwohl er nicht teilgenommen hat.

Abb. 4.4-2 zeigt diejenigen Klassen, die sich aufgrund der *top-down*-Methode aus den Geschäftsprozessen identifizieren lassen. Keine Klassen sind beispielsweise:

- Mitarbeiter, denn über diese Personen werden keine Daten gespeichert,
- Rechnung, denn es handelt sich um eine Ausgabe, deren Daten bereits in anderen Klassen enthalten sind,
- Gebühr, denn es handelt sich um ein Attribut von Seminartyp.
- Interessenten und Kunden sind identisch. Daher bilden wir nur die Klasse Kunde.

Abb. 4.4-2:  
Beispiel zur  
*top-down*-Methode



Verwechseln Sie diese Vorgehensweise nicht mit der Substantiv-Methode (*abbott's noun approach*). Bei diesem Verfahren werden zunächst *alle* Substantive als potentielle Klassen betrachtet und in einer Liste aufgeführt. Der Vorteil besteht zweifelsohne darin, daß ein Systemanalytiker sehr leicht und schnell viele Klassen findet. Nachteilig wirkt sich aus, daß zunächst sehr viele überflüssige Klassen (u.a. viele Attribute) identifiziert werden, die dann in einem weiteren Schritt wieder aussortiert werden müssen.

### 3 Kategorien

In vielen Projekten kommen Klassen bestimmter Kategorien vor (vergleiche /Larman 98/, /Coad, Yourdon 91/, /Booch 94/).

Beispielsweise können Personen verschiedene Rollen im System spielen, wobei in jeder Rolle andere Attribute und ein anderes Verhalten von Interesse sind. Dann ist für jede Rolle eine Klasse zu modellieren (Kategorie *Personen und deren Rollen*).

Über durchgeführte Aktionen müssen Daten gespeichert werden. Beispiele: Eine Banküberweisung wird getätigt; eine Bestellung wird erteilt. Dann wird für jede Aktion eine Klasse erstellt (Kategorie *Informationen über Aktionen*).

Fragen Sie, welche Orte im System vorkommen. Dabei kommt es weniger auf die echte räumliche Trennung (z.B. durch Wände) als auf die Trennung der Funktionsbereiche an (Kategorie *Orte*). Beispiel: Eine Arztpraxis enthält die Orte Wartezimmer, Anmeldungsbereich und Behandlungsraum.

Weitere Kategorien sind:

- Konkrete Objekte bzw. Dinge,
- Organisationen,
- Behälter,
- Dinge in einem Behälter,

- Ereignisse,
- Kataloge und
- Verträge.

### Analytische Schritte zum Validieren der Klassen

Die Bedeutung des Klassennamens kann gar nicht überschätzt werden. Die Verständlichkeit Ihres Modells wird wesentlich durch wirklich gute Namen bestimmt. Widmen Sie dieser Aufgabe daher genügend Zeit.

4 Klassenname

Theoretisch ist es möglich, ein System durch eine einzige Klasse zu modellieren, die alle Attribute und alle Operationen enthält. Es ist klar, daß eine solche Vorgehensweise zu keinem befriedigenden Ergebnis führt. Das andere Extrem ist, daß sehr viele Klassen gebildet werden. Oft sind deren Attribute dann vom elementaren Typ. Bei näherer Betrachtung kann man sehr leicht feststellen, daß es sich bei vielen »Klassen« in Wirklichkeit um Attribute handelt. Eine solche Vorgehensweise führt mit Sicherheit zu einem unübersichtlichen Modell, obwohl das Kriterium des aussagefähigen Namens sogar erfüllt wäre. Es ist also von besonderer Bedeutung, das richtige Abstraktionsniveau zu finden. Das Ziel muß es daher sein, in sich abgeschlossene Klassen adäquater Komplexität zu modellieren.

5 Abstraktionsniveau

Folgende Angaben sollen eine ungefähre Größenordnung über die Anzahl der Klassen in einem objektorientierten System – also nicht nur auf die Analyse bezogen – vermitteln. Nach /Booch 96/ ist bei Projekten mittlerer Komplexität (d.h. Entwicklungsdauer von einem Jahr) mit 50 bis 100 Klassen zu rechnen. Ein großes System (d.h. 10 bis 100 Mitarbeiterjahre) besteht nach /Jacobson 95/ aus mehreren 100 oder sogar mehreren 1000 Klassen. Es gibt Systeme, die 10.000 bis 100.000 Klassen enthalten.

In der Systemanalyse werden keine Klassen modelliert, um Mengen von Objekten zu verwalten. Dadurch würde die Anzahl der Klassen in vielen Fällen wesentlich erhöht, ohne die Aussagefähigkeit des Klassendiagramms zu verbessern. Statt dessen werden Operationen, die alle Objekte der Klasse betreffen, als Klassenoperationen spezifiziert.

6 implizierte Objektverwaltung

Die mangelnde Bildung von komplexen Attributen führt zu vielen kleinen Klassen, die mittels Assoziationen verbunden sind. Dieser Fehler führt nicht nur zu vielen Klassen, sondern auch zu vielen überflüssigen Assoziationen, welche die Verständlichkeit des Gesamtmodells erschweren. Eine häufige – jedoch falsche – Vorstellung ist, daß sich alle konkreten Objekte des Problembereichs im Analysemodell als Klasse wiederfinden. Beispielsweise muß eine Rechnung für ein gebuchtes Seminar, die im allgemeinen in Papierform vorliegt, nicht als Klasse modelliert werden. Diese Klasse würde keine neue Information enthalten, weil alle Daten bereits in den Klassen Buchung, Seminar und Kunde gespeichert sind (vergleiche

7 Fehlerquellen

## LE 7 4 Checklisten zum Erstellen eines OOA-Modells

Abb. 4.5-1). Das Erstellen der Rechnung ist eine Operation, die der Klasse Buchung zugeordnet wird.

8 für Datenmodellierer

Systemanalytiker, die bisher Datenmodelle erstellt haben, können im ersten Ansatz jede Entitätsmenge mit einer Klasse und jede Beziehungsmenge mit einer assoziativen Klasse gleichsetzen. Weil im Datenmodell nur Attribute vom einfachen Typ vorkommen, ist es im allgemeinen notwendig, aus einigen dieser »Klassen« komplexe Attribute zu machen. Eine 1:1-Abbildung des Datenmodells führt nicht zu einem objektorientierten Modell guter Qualität.

**Tab. 4.4-1a:**  
**Checkliste**  
**Klassen**

### Ergebnisse

#### ■ Klassendiagramm

Tragen Sie jede Klasse – entweder nur mit Namen oder mit wenigen wichtigen Attributen/Operationen – in das Klassendiagramm ein.

#### ■ Kurzbeschreibung der Klassen

Erstellen Sie für jede Klasse, deren Name nicht selbsterklärend ist, eine Kurzbeschreibung von 25 oder weniger Wörtern.

### Konstruktive Schritte

#### 1 Welche Klassen lassen sich mittels Dokumentanalyse identifizieren?

- Formulare, Listen.
- *Re-Engineering*-System: Benutzerhandbücher, Bildschirmmasken, Dateibeschreibungen, Funktionalität des laufenden Systems.

#### 2 Welche Klassen lassen sich aus der Beschreibung der Geschäftsprozesse identifizieren?

- Beschreibung nach Klassen durchsuchen.
- Potentielle Klasse auf Attribute überprüfen.

#### 3 Sind Klassen der folgenden Kategorien zu modellieren?

- Konkrete Objekte bzw. Dinge
- Personen und deren Rollen
- Informationen über Aktionen
- Orte
- Organisationen
- Behälter
- Dinge in einem Behälter
- Ereignisse
- Kataloge
- Verträge

### Analytische Schritte

#### 4 Liegt ein aussagefähiger Klassenname vor?

Der Klassenname soll

- der Fachterminologie entsprechen,
- ein Substantiv im Singular sein,
- so konkret wie möglich gewählt werden,
- dasselbe ausdrücken wie die Gesamtheit der Attribute,
- nicht die Rolle dieser Klasse in einer Beziehung zu einer anderen Klasse beschreiben,
- eindeutig im Paket bzw. im System sein und
- nicht dasselbe ausdrücken wie der Name einer anderen Klasse.



**5 Ist das gewählte Abstraktionsniveau richtig?****6 Wann liegt keine Klasse vor?**

- Bilden Sie keine Klassen, um Mengen von Objekten zu verwalten.

**7 Fehlerquellen**

- Zu kleine Klassen.
- Aus jedem Report eine Klasse modellieren.
- Klasse modelliert Benutzungsoberfläche.
- Klasse modelliert Entwurfs- oder Implementierungsdetails.

**Für klassische Entwickler****8 Identifizieren von Klassen für Datenmodellierer**

Eine potentielle Klasse ist jede Entitäts- und jede Beziehungsmenge mit Attributen.

*Tab. 4.4-1b:  
Checkliste  
Klassen*

## 4.5 Checkliste Assoziation

Identifizieren Sie diejenigen **Assoziationen** zwischen den Klassen, die für die betrachteten Geschäftsprozesse benötigt werden. Es geht also nicht darum alle – fachlich möglichen – Assoziationen zu ermitteln, sondern diejenigen, die notwendig sind, damit die Objekte die beabsichtigten Aufgaben des Geschäftsprozesses ausführen können.

Konzentrieren Sie sich im ersten Schritt nur auf das Identifizieren der Assoziationen (Tab. 4.5-1). Ermitteln Sie erst später die zugehörigen Kardinalitäten und machen Sie sich Gedanken darüber, ob eine »einfache« Assoziation, eine **Aggregation** oder eine **Komposition** vorliegt. Das bedeutet, daß Sie nicht mit beträchtlichem Zeitaufwand danach suchen sollten. Wenn es offensichtlich ist, welche Kardinalitäten und Assoziationsarten vorliegen, dann können Sie diese Informationen natürlich gleich eintragen. In diesem Fall verwenden Sie die entsprechenden Checklisten zur Prüfung und ggf. zur Verbesserung.

### Konstruktive Schritte zum Identifizieren von Assoziationen

Aus den gleichen Dokumenten, die zum Identifizieren der Klassen eingesetzt wurden, lassen sich oft Assoziationen ermitteln. In der traditionellen Datenverarbeitung, in der es keine Objekte mit impliziter Identität gibt, werden die »Objekte« durch Nummern eindeutig identifiziert. Diese Nummern finden sich als Primär- und Fremdschlüssel in den Dokumenten. Beachten Sie, daß diese Vorgehensweise auch zu abgeleiteten Assoziationen führt, die entsprechend zu kennzeichnen sind.

Auch aus den Geschäftsprozessen lassen sich Assoziationen identifizieren.

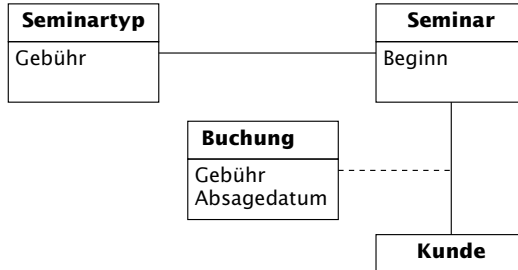
**1** Dokumentanalyse

**2** Beschreibung Geschäftsprozesse

## LE 7 4 Checklisten zum Erstellen eines OOA-Modells

**Beispiel** Abb. 4.5-1 zeigt die Assoziationen, die sich für die Seminarorganisation aus Kapitel 4.4 ergeben. Jedes Seminar gehört zu einem bestimmten Typ. Zwischen Kunde und Seminar besteht eine Assoziation, die selbst Attribute besitzt und daher als assoziative Klasse modelliert wird.

Abb. 4.5-1:  
Seminar-  
organisation



**3 Kategorien** Analog zu den Klassen lassen sich viele Assoziationen einer bestimmten Kategorie zuordnen. Diese Kategorien können beim Modellieren dazu beitragen, konkretere Fragen zu stellen und mehr Informationen über das Analysemodell zu gewinnen. /Larman 98/ führt für zwei Klassen A und B folgende Kategorien an:

- A ist physische Komponente von B,
- A ist logische Komponente von B,
- A ist eine Beschreibung für B,
- A ist eine Zeile einer Liste B,
- A ist ein Mitglied von B,
- A ist eine organisatorische Einheit von B,
- A benutzt B,
- A kommuniziert mit B und
- A besitzt B.

**4 Restriktionen** Um Restriktionen zu ermitteln, die zwei oder mehrere Assoziationen betreffen, ist die Erstellung von Objektdiagrammen nützlich. Eine *ordered*-Assoziation liegt vor, wenn die Reihenfolge der Objektverbindungen relevant ist. Auch hier ist ein Objektdiagramm sinnvoll. Natürlich können Sie Restriktionen auch frei formulieren.

**Beispiel** Wird in Abb. 4.5-2 keine Restriktion angegeben, so kann jeder Student bei allen Professoren – auch außerhalb seines Fachbereichs – Vorlesungen belegen. Dann kann beispielsweise der Student s1 beim Professor p1 eine Vorlesung hören, aber nicht beim Professor p3.

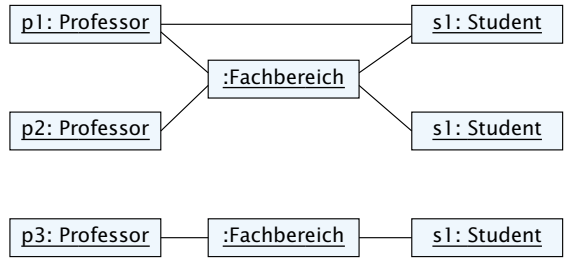
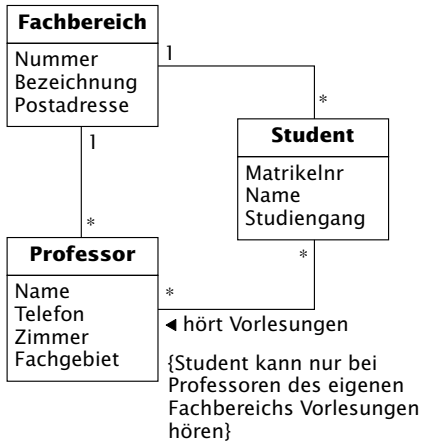


Abb. 4.5-2:  
Restriktion einer  
Assoziation

**Analytische Schritte zum Validieren der Assoziationen**

Abb. 4.5-3 zeigt, wie sehr die Verständlichkeit eines Modells durch geeignete Rollennamen erhöht werden kann. Um Rollennamen zu finden, wählen Sie eine Klasse, z.B. Firma, und prüfen Sie für jede Assoziation, die von ihr ausgeht: Welche Rolle spielt die Klasse? Der Rollename entspricht mehr dem objektorientierten Ansatz und der Assoziationsname mehr der Datenmodellierung. Außerdem führt der Rollename oft zu besseren Bezeichnungen (vergleiche /Jacobson 92/). Aus diesen Gründen ziehe ich die Rolle – wenn möglich – dem Assoziationsnamen vor. Es ist nicht notwendig, alle Assoziationen zu benennen.

5 Benennung

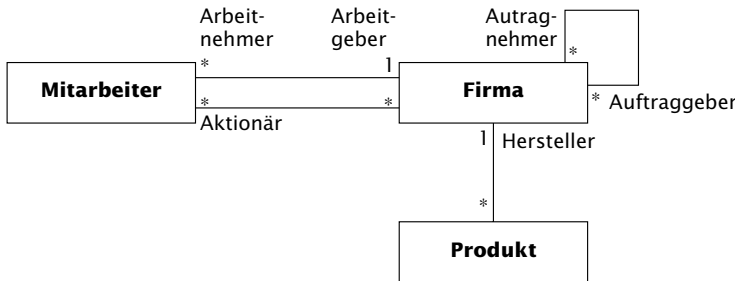


Abb. 4.5-3: Rollen

Besteht zwischen zwei Klassen eine 1:1-Assoziation, dann ist zu prüfen, ob eine Zusammenfassung sinnvoll ist oder nicht. Zwei Klassen sind zu modellieren, wenn

6 1:1-Assoziation

- die Verbindung in einer oder beiden Richtungen optional ist und sich die Verbindung zwischen beiden Objekten ändern kann,
- es sich um zwei umfangreiche Klassen handelt,
- die beiden Klassen eine unterschiedliche Semantik besitzen.

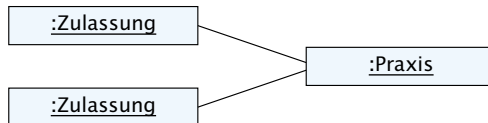
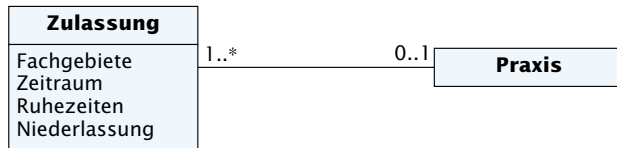
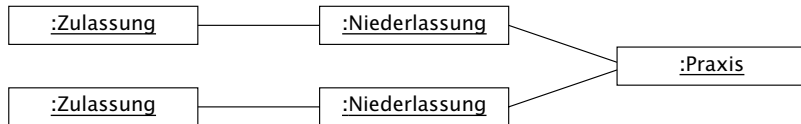
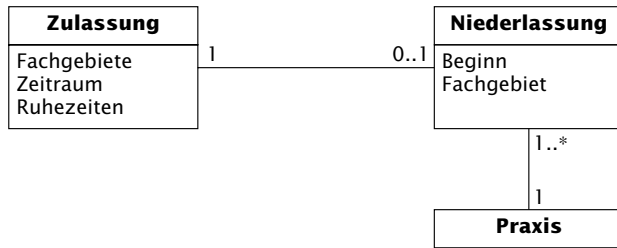
Um diese Kriterien zu überprüfen, ist es sinnvoll Objektdiagramme zu erstellen.

## LE 7 4 Checklisten zum Erstellen eines OOA-Modells

**Beispiele** Die 1:1-Assoziation im weißen Modell der Abb. 4.5-4 kann aus folgenden Gründen zusammengefaßt werden. Aus fachlicher Sicht gilt, daß auf jede Zulassung eines Arztes in spätestens sechs Monaten eine Niederlassung erfolgen muß, die mit dem Einrichten einer Praxis verbunden ist. Die einmal aufgebaute Verbindung zwischen Zulassung und Niederlassung kann nicht geändert werden. Da die Klasse Niederlassung nur zwei elementare Attribute enthält, integrieren wir sie in die Klasse Zulassung, wodurch sich das blaue Modell ergibt.

Eine andere Situation liegt bei der 1:1-Assoziation der Abb. 4.5-5 vor. Die Verbindung zwischen den beteiligten Objekten ändert sich häufig. Daher bleiben in diesem Fall zwei Klassen bestehen.

Abb. 4.5-4:  
Zusammenfassung  
zweier Klassen  
mit 1:1-Assoziation



**7** mehrere  
Assoziationen

Zwischen zwei Klassen können mehrere Assoziationen existieren. Sie müssen eine unterschiedliche Semantik besitzen, was sich immer in unterschiedlichen Rollen- oder Assoziationsnamen zeigt. Oft unterscheiden sie sich auch in ihren Kardinalitäten.

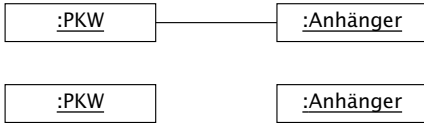
**8** abgeleitete  
Assoziationen

Die abgeleitete Assoziation wird beim fachlichen Sollkonzept benötigt, um unbeabsichtigte Redundanz zu markieren. Sie kann auch verwendet werden, wenn dadurch das Modell übersichtlicher und besser lesbar wird. Bei der Ist-Analyse ist sie notwendig, um ein

## 4.5 Checkliste Assoziation LE 7



Abb. 4.5-5:  
1:1-Assoziation



vorhandenes System – mit allen enthaltenen Redundanzen – zu dokumentieren.

Eine abgeleitete Assoziation liegt vor, wenn die beteiligten Objekte auch über Objektverbindungen anderer Assoziationen erreichbar sind. Sie kann – anhand von Objektdiagrammen – geprüft werden, in dem alle Objektverbindungen der abgeleiteten Assoziation – gedanklich – entfernt werden und überprüft wird, ob die Objekte über die anderen Verbindungen noch erreichbar sind.

In Abb. 4.5-6 ist die Assoziation zwischen Bestellung und Artikel redundant, da sich dieser Zusammenhang auch aus den beiden anderen Assoziationen ergibt. Vergleichen Sie dazu die Abb. 4.5-2. Obwohl beide Modelle auf den ersten Blick sehr ähnlich erscheinen, ist im zweiten Fall keine Assoziation abgeleitet.

Beispiel

Abb. 4.5-6:  
Erkennen abgeleiteter Assoziationen



Eine assoziative Klasse besitzt die Eigenschaften einer Klasse und einer Assoziation. Beim Aufbau einer Objektverbindung wird ein Objekt der assoziativen Klasse erzeugt und mit den entsprechenden Attributwerten gefüllt. Eine assoziative Klasse wird benötigt, wenn eine Assoziation Attribute und/oder Operationen enthält. Durch die Modellierung mit einer assoziativen Klasse bleibt die ursprüngliche Assoziation zwischen den beteiligten Klassen bestehen und damit im Modell deutlich sichtbar. Prinzipiell kann jede assoziative Klasse nach dem Schema von Abb. 4.5-7 in zwei Assoziationen und eine eigenständige Klasse aufgelöst werden.

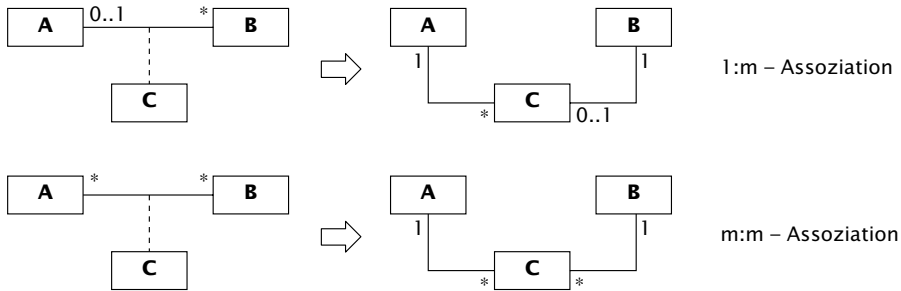
9 assoziative Klasse

Gelegentlich wird eine Vererbung gewählt, wo eine Assoziation angebracht wäre. Diesen Fehler können Sie leicht erkennen und beheben, wenn Sie Ihr Klassendiagramm durch Objektdiagramme ergänzen.

10 Fehlerquelle

## LE 7 4 Checklisten zum Erstellen eines OOA-Modells

Abb. 4.5-7:  
Auflösen  
einer asso-  
ziativen  
Klasse



**Tab. 4-5-1a:**  
**Checkliste**  
**Assoziationen**

### Ergebnis

#### ■ Klassendiagramm

Assoziationen im ersten Schritt nur als Linie eintragen. Noch keine Kardinalitäten, Aggregation, Komposition, Rollen, Namen, Restriktionen.

### Konstruktive Schritte

#### 1 Welche Assoziationen lassen sich mittels Dokumentanalyse ableiten?

- Aus Primär- und Fremdschlüsseln ermitteln.

#### 2 Welche Assoziationen lassen sich aus Beschreibungen der Geschäftsprozesse ermitteln?

#### 3 Liegt eine Assoziation der folgenden Kategorien vor?

- A ist physische Komponente von B.
- A ist logische Komponente von B.
- A ist eine Beschreibung für B.
- A ist eine Zeile einer Liste B.
- A ist ein Mitglied von B.
- A ist eine organisatorische Einheit von B.
- A benutzt B.
- A kommuniziert mit B.
- A besitzt B.

#### 4 Welche Restriktionen muß die Assoziation erfüllen?

- Eine Assoziation: {ordered}.
- mehrere Assoziationen: {or}, {subset}.

Erstellen Sie Objektdiagramme.

### Analytische Schritte

#### 5 Ist eine Benennung notwendig oder sinnvoll?

- Notwendig, wenn zwischen zwei Klassen mehrere Assoziationen bestehen.
- Rollennamen (Substantive) sind gegenüber Assoziationsnamen (Verben) zu bevorzugen.
- Rollennamen sind bei reflexiven Assoziationen immer notwendig.

#### 6 Liegt eine 1:1-Assoziation vor?

Zwei Klassen sind zu modellieren, wenn

- die Verbindung in einer oder beiden Richtungen optional ist und sich die Verbindung zwischen beiden Objekten ändern kann,
- es sich um zwei umfangreiche Klassen handelt,
- die beiden Klassen eine unterschiedliche Semantik besitzen.

**7 Existieren zwischen zwei Klassen mehrere Assoziationen?**

Prüfen Sie, ob die Assoziationen

- eine unterschiedliche Bedeutung besitzen oder/und
- unterschiedliche Kardinalitäten haben.

**8 Sind abgeleitete Assoziationen korrekt verwendet?**

- Abgeleitete Assoziationen fügen keine neue Information zum Modell hinzu.
- Sie lassen sich leicht mittels Objektdiagrammen erkennen.

**9 Soll eine assoziative Klasse oder eine eigenständige Klasse modelliert werden?**

- Assoziative Klasse betont die Assoziation zwischen den beteiligten Klassen.

**10 Fehlerquelle**

- Verwechseln von Assoziation mit Vererbung.

**Tab. 4-5-1b:  
Checkliste  
Assoziationen**

**Konstruktive und analytische Schritte zum Identifizieren und Validieren von Kardinalitäten**

Die Frage, ob ein Schnappschuß oder die Historie zu modellieren ist, läßt sich am einfachsten beantworten, indem Sie Anfragen formulieren. Die Fragestellung »Wer hat sich zur Zeit den roten Mercedes ausgeliehen?« weist auf einen Schnappschuß hin und die Frage »Welche Personen haben den roten Mercedes im letzten Jahr ausgeliehen?« zeigt, daß die Historie modelliert werden muß. Während bei einem Schnappschuß eine alte Verbindung gelöst wird bevor eine neue aufgebaut wird, wird bei der Historie eine neue Verbindung zwischen den jeweiligen Objekten hinzugefügt. Auch hier können Sie Objektdiagramme nutzbringend einsetzen.

1 Schnappschuß oder Historie?

Bei der Überlegung, ob eine Muß- oder Kann-Assoziation vorliegt, sollten Sie folgende Fragen stellen:

2 Muß- oder Kann-Beziehung

- Wie und wann werden Objekte der Klassen erzeugt?
- Können beteiligte Objekte gelöscht werden und welche Konsequenzen hat dies?

Abb. 4.5-8 zeigt die Assoziation zwischen Kunde und Bestellung. Das blaue Modell enthält eine wechselseitige Muß-Assoziation. Das bedeutet: Wer keine Bestellung erteilt hat, ist auch kein Kunde. Wenn jemand Kunde wird, so ist dafür zu sorgen, daß auch sofort eine Bestellung erfaßt und anschließend die Assoziation zwischen Bestellung und Kunde aufgebaut wird. Daher müssen diese Funktionen softwaretechnisch miteinander verbunden sein. Wird eine neue Bestellung für einen bereits existierenden Kunden angelegt, so ist mit dem Erfassen der Bestellung sofort die Assoziation zu dem Kun-

Beispiele

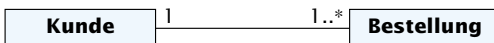


Abb. 4.5-8: Muß- vs. Kann-Assoziation

den aufzubauen. Werden die Bestellungen gelöscht, dann muß mit dem Löschen der letzten Bestellung auch der Kunde gelöscht werden. Dem grauen Modell liegt eine andere Überlegung zugrunde. Eine Person wird Kunde, auch wenn sie noch keine Bestellung erteilt hat (z.B. nur einen Katalog angefordert hat). Bestellungen können dem Kunden später beliebig zugeordnet und entfernt werden. Wird allerdings der Kunde gelöscht, dann müssen auch alle seine Bestellungen gelöscht werden.

- 3 feste Grenzen      Bei technischen Systemen liegt oft eine vom Problem her fest vorgegebene Anzahl von Objekten vor. Bei folgenden Beispielen ist die Obergrenze durch die Bauweise von Waggonen und Abteilen vorgegeben (Abb. 4.5-9). Ist vom Problem her keine zwingende Obergrenze vorgegeben, so sollte die variable Obergrenze *many* gewählt werden. Bei Informationssystemen liegt meistens eine unbestimmte Anzahl von Objekten vor. Hier sollten Sie eine variable Anzahl auch dann wählen, wenn der Auftraggeber eine willkürliche Obergrenze angibt.

**Tab. 4.5-2:**  
**Checkliste**  
**Kardinalitäten**

**Konstruktive/analytische Schritte**

**1 Ist ein Schnappschuß oder ist die Historie zu modellieren?**

Aus den Anfragen an das System ergibt sich, ob

- ein Schnappschuß (1- bzw. 0..1-Kardinalität) oder
- die Historie (*many*-Kardinalität) zu modellieren ist.

**2 Liegt eine Muß- oder Kann-Assoziation vor?**

- Bei einer einseitigen Muß-Assoziation (Untergrenze  $\geq 1$  auf einer Seite) gilt:  
Sobald das Objekt A erzeugt ist, muß auch die Beziehung zu dem Objekt B aufgebaut, und B vorhanden sein bzw. erzeugt werden.
- Bei einer wechselseitigen Muß-Beziehung (Untergrenze  $\geq 1$  auf beiden Seiten) gilt:  
Sobald das Objekt A erzeugt ist, muß auch die Beziehung zu dem Objekt B aufgebaut und ggf. das Objekt B erzeugt werden. Wenn das letzte Objekt A einer Beziehung gelöscht wird, dann muß auch Objekt B gelöscht werden.
- Bei einer Kann-Beziehung (Untergrenze = 0) kann die Beziehung zu einem beliebigen Zeitpunkt nach dem Erzeugen des Objekts aufgebaut werden.

**3 Enthält die Kardinalität feste Werte?**

- Ist eine Obergrenze vom Problembereich her zwingend vorgegeben (z.B. maximal 6 Spieler)? Im Zweifelsfall mit variablen Obergrenzen arbeiten.
- Ist die Untergrenze vom Problembereich her zwingend vorgegeben (z.B. mindestens 2 Spieler)? Im Zweifelsfall mit »0« arbeiten.
- Gelten besondere Restriktionen für die Kardinalitäten (z.B. eine gerade Anzahl von Spielern)?

**4 Fehlerquelle**

- Oft werden Muß-Assoziationen verwendet, wo sie nicht benötigt werden.



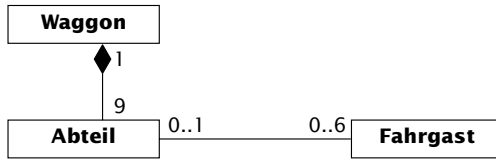


Abb. 4.5-9:  
Assoziation  
mit festen  
Obergrenzen

Unter Umständen können die **Kardinalitäten** nicht sinnvoll in der Notation ausgedrückt werden. Dann müssen Sie zusätzlich eine Restriktion verwenden.

### Konstruktive und analytische Schritte zum Identifizieren und Validieren der Assoziationsart

Die Abgrenzung zwischen »einfacher« Assoziation, Aggregation und Komposition ist für viele Analytiker problematisch. Man kann in einem Projekt endlose Stunden mit Diskussionen verbringen, die letztendlich zur Qualität des Modells nichts beitragen. Manche Autoren (z.B. /Fowler 97/) empfehlen, auf Aggregation und Komposition völlig zu verzichten. Andere sind der Meinung, daß Aggregation und Komposition unbedingt benötigt werden. Während die Komposition in der UML relativ präzise definiert ist, wurde die Aggregation bewußt allgemeiner definiert, um einen gewissen Spielraum zu ermöglichen /UML 97/.

Wählen Sie eine Komposition, wenn folgende Bedingungen eindeutig erfüllt sind:

- Die Beziehung kann durch »besteht aus« oder »ist enthalten in« beschrieben werden (*whole part*).
- Die Kardinalität der Aggregatklasse darf nicht größer als eins sein (*unshared aggregation, strong ownership*).
- Wird das Ganze gelöscht, dann werden automatisch seine Teile gelöscht (*they live and die with it*). Ein Teil darf jedoch zuvor explizit entfernt werden.

Die dynamische Semantik des Ganzen gilt auch für seine Teile. Diese letzte Bedingung ist besonders wichtig für das Beurteilen einer Komposition, denn die erste kann auch bei einer Aggregation und die zweite und dritte können auch bei einfachen Assoziationen vorliegen. Insbesondere das *live and die with it*-Kriterium reicht nicht aus, da es bei jeder Muß-Assoziation vorkommt.



Auch die in Kapitel 3.1 genannten Muster bilden eine gute Orientierungshilfe. Dazu gehören

- Liste (Bestellung – Bestellposition),
- Baugruppe (Auto – Motor) und
- Stückliste mit physikalischem Enthaltensein (Verzeichnis – Verzeichnis).

1 Komposition

Kapitel 3.1

**LE 7 4 Checklisten zum Erstellen eines OOA-Modells**

**2 Aggregation** Die Aggregation kommt relativ selten vor. Da ihre Anwendung in der Praxis wegen der unpräzisen Definition problematisch ist und die Modellierung genauso gut mit einer »einfachen« Assoziation erfolgen kann, verzichte ich auf dieses Konstrukt. Die folgenden Beispiele sollen zeigen, in welchen Fällen die Assoziation angewendet werden kann.

**Beispiele** Die Abb. 4.5-10 modelliert die Stücklistenproblematik, bei der physisches und logisches Enthaltensein zu unterschiedlichen Modellen führen.

Abb. 4.5-10:  
Physisches und  
logisches  
Enthaltensein

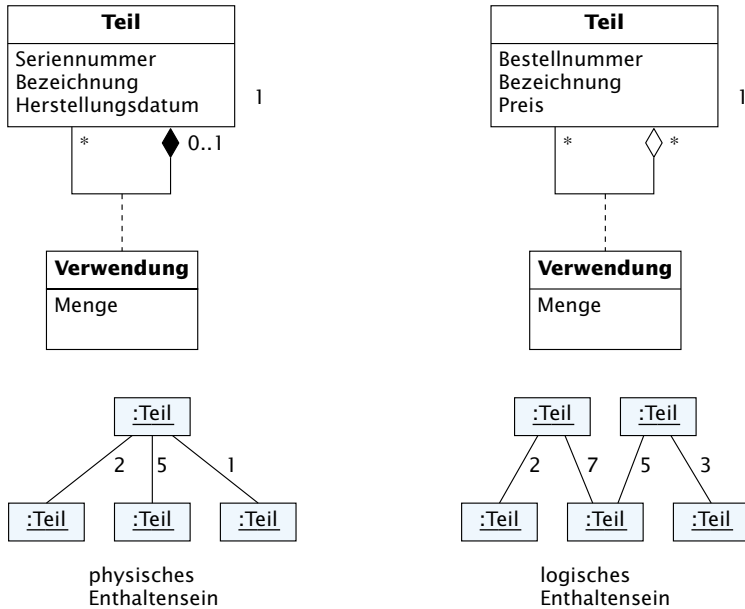
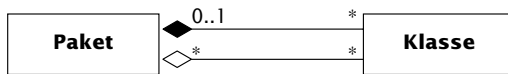


Abb. 4.5-11 zeigt ein Beispiel, wo Komposition und Aggregation nebeneinander verwendet werden /UML 97a/. Jede Klasse kann in höchstens einem Paket enthalten sein. Sie kann jedoch in mehreren – anderen – Paketen referenziert werden.

Abb.4.5-11:  
Komposition und  
Aggregation



**3 im Zweifel  
Assoziation**

Sie sollten es sich zur Angewohnheit machen, daß bei dem geringsten Zweifel an einer Komposition oder Aggregation immer die Assoziation zu wählen ist. Ich beschränke mich bei meinen OOA-Modellen auf die Komposition und die »einfache« Assoziation.

**4 Komposition  
oder Attribut**

Prinzipiell ist es möglich, jedes Attribut als Klasse zu modellieren und mittels einer Komposition mit der ursprünglichen Klasse

**Konstruktive/Analytische Schritte****1 Für eine Komposition gilt:**

- Es liegt eine *whole-part*-Beziehung vor.
- Die Kardinalität bei der Aggregatklasse beträgt 0..1 oder 1.
- Die Lebensdauer der Teile ist an die des Ganzen gebunden.
- Die Funktionen des Ganzen werden automatisch auf seine Teile angewendet.

**2 Für eine Aggregation gilt:**

- Sie ist selten.
- Es liegt eine *whole-part*-Beziehung mit *shared aggregation* vor.

**3 Im Zweifelsfall immer eine einfache Assoziation verwenden.****4 Fehlerquelle**

- Modellieren von Attributen mittels Komposition.

**Tab. 4.5-3:**  
**Checkliste**  
**»Einfache«**  
**Assoziation,**  
**Aggregation,**  
**Komposition**

zu verbinden. Da Attribute und Klassen jedoch eine unterschiedliche Bedeutung für die Gesamtarchitektur besitzen, würde diese Vorgehensweise zu keinem guten Modell führen. Ich gehe in der »Checkliste Attribute« genauer auf diese Problematik ein.

## 4.6 Checkliste Attribut

Die Ausgangsbasis für das Identifizieren der Attribute sind die Klassen, die im allgemeinen schon einige Attribute enthalten. Nun sollen diese Attributlisten vervollständigt werden.

Oft können die Attribute einer Klasse ganz unterschiedlich ausgedrückt werden. Beispielsweise kann die Position eines Punkts durch seine Polar- oder seine kartesischen Koordinaten beschrieben werden. In der Analysephase kann eine dieser Formen gewählt werden. Während der Implementierung können bei Bedarf Operationen geschrieben werden, um beispielsweise kartesische Koordinaten in Polarkoordinaten umzurechnen.

### Konstruktive Schritte zum Identifizieren von Attributen

Analog zu den Klassen und Assoziationen lassen sich aus vorhandenen Dokumenten Attribute identifizieren. Da diese Dokumente meistens nur Daten einfachen Typs enthalten, müssen entsprechende Datenstrukturen gebildet werden. Es besteht jedoch die Gefahr, daß zu viele Attribute unreflektiert übernommen werden. Prüfen Sie daher für jedes Attribut, ob es »im Laufe seines Lebens« einen Wert annehmen kann und ob diese Werte an der Benutzungsoberfläche sichtbar sind.

Analysieren Sie, welche Daten zur Ausführung der Aufgaben aller Geschäftsprozesse benötigt werden. Zusätzlich sollten Sie Attribute beschreiben, die für eine gewünschte Listenfunktionalität benötigt werden.

**1** Dokument-  
analyse

**2** Beschreibung  
Geschäftsprozesse

### Analytische Schritte zum Validieren der Attribute

- 3** Attributname In der Systemanalyse tragen sinnvoll gewählte Attributnamen wesentlich zum Verständnis des Modells bei. Bei komplexen (strukturierten) Attributen ist darauf zu achten, daß der Name der Gesamtheit der Komponenten entspricht. Im Sinne einer hohen Aussagekraft sollten möglichst keine Abkürzungen verwendet werden. Eine Ausnahme von dieser Regel bilden fachspezifische und allgemein übliche Abkürzungen (z.B. PLZ).
- 4** Klasse oder komplexes Attribut Einige Methoden empfehlen, daß Attribute nur von einem einfachen Typ sein sollen und daß sie andernfalls als eigenständige Klasse darzustellen sind. Ich halte es für eine wesentliche Eigenschaft des objektorientierten Modells, daß sich der Analytiker bei der Modellierung nicht an den vorgegebenen Typen orientieren muß, sondern daß er unter problemadäquaten Gesichtspunkten Attribute beliebigen Typs definieren kann. Für komplexe Attribute verwenden wir zur Konstruktion der Typen das Klassenkonzept. Daher wurde in Kapitel 2.1 zwischen Architekturklassen und elementaren Klassen unterschieden. ↕
- Kapitel 2.1

Prinzipiell kann jedes Attribut auch als Komposition modelliert werden. Es gilt die *whole-part*-Beziehung und die Existenz eines Attributwerts ist immer an sein Objekt gebunden.

Worin unterscheiden sich Objekte und Attributwerte? Attribute besitzen – gleichgültig, ob einfaches oder komplexes Attribut – in der Analyse keine eigene Objektidentität. Der Zugriff auf Attributwerte erfolgt immer über das entsprechende Objekt. Das – für die Systemanalyse – wichtigste Kriterium ist, daß Attribute immer Eigenschaften der jeweiligen Objekte beschreiben und bei der Modellbildung im Vergleich zu den Klassen von untergeordneter Bedeutung sind. Dagegen besitzt jedes Objekt stets eine Identität. Teilklassen haben im Modell die gleiche Bedeutung wie Aggregat-Klassen. Es können daher auch Zugriffe von Teil-Objekten auf das zugehörige Aggregat-Objekt erfolgen.

**Beispiel** Wir betrachten die Klasse *Artikel* mit dem Attribut *Preis*. Ist der Typ *Fixed*, dann ist offensichtlich, daß es sich bei *Preis* um ein Attribut handelt. Als Typ könnte aber auch eine Datenstruktur gewählt werden (Abb. 4-6-1). Im zweiten Fall wird die Währungseinheit zusätzlich gespeichert, im dritten werden in Abhängigkeit vom Käufertyp (Händler, Großkunde, Einzelkunde) verschiedene Preise aufgeführt. Bei einem *Preis* handelt es sich unabhängig vom Typ ganz offensichtlich um ein Attribut, das im Vergleich zur Klasse von untergeordneter Bedeutung für das Modell ist. Der *Preis* benötigt keine eigene Objektidentität, denn ohne das entsprechende Artikelobjekt ergibt er keinen Sinn.

Dagegen wird das Lager als eigenständige Klasse modelliert. Ein konkretes Lager soll unabhängig davon im System existieren, ob es gerade Artikel enthält. Es soll Anfragen der Art »Welche Artikel sind im Lager Viktoriastraße?« ermöglichen.

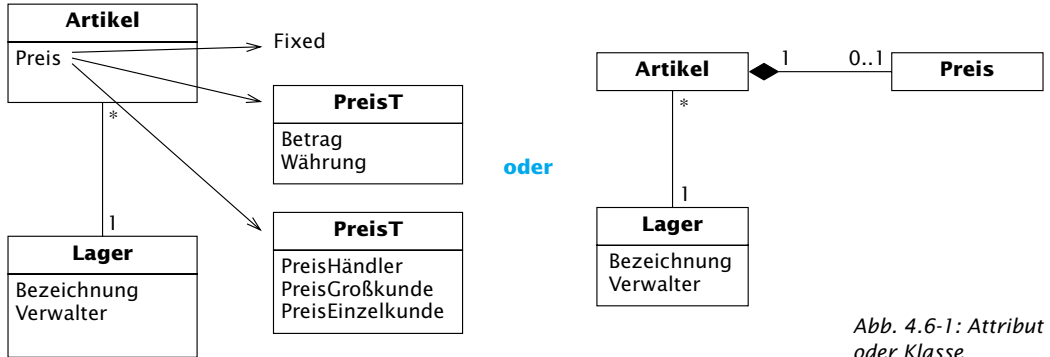


Abb. 4.6-1: Attribut oder Klasse

Die spezifizierten Attribute sollen sowohl dem Auftraggeber als auch dem Entwerfer und Programmierer deutlich machen, welche Informationen vom System verwaltet werden. Dabei ist es wichtig, einzelne Attribute zu geeigneten Strukturen zusammenzufassen. Damit ist eine kompaktere und übersichtlichere Darstellung des Klassendiagramms möglich. Eine geeignete Datenstruktur läßt sich daran erkennen, daß Sie einen aussagefähigen Namen dafür finden. Ein nichtssagender Name weist meist auf ein willkürliches Datenbündel hin. Um ein gut lesbares Modell zu erhalten, ist eine Ausgewogenheit zwischen der Klassenanzahl und der Klassengröße besonders wichtig.

5 Abstraktionsniveau

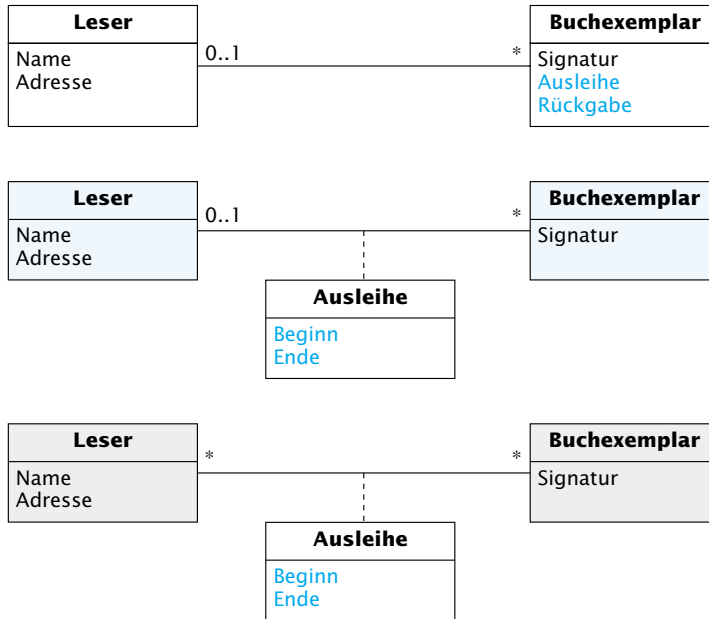
Um die Zugehörigkeit eines Attributs zu einer Klasse oder zu einer Assoziation zu prüfen, verwendet /D'Souza 94/ folgenden Test: Muß das Attribut auch dann zu jedem Objekt der Klasse gehören, wenn die betreffende Klasse isoliert von allen anderen Klassen betrachtet wird? Wenn ja, dann gehört das Attribut zu dieser Klasse. Wenn nein, dann ist zu prüfen, ob es sich einer Assoziation zuordnen läßt. Ist keine Zuordnung möglich, dann spricht viel für eine vergessene Klasse oder Assoziation.

6 Attribut einer Klasse oder einer Assoziation ?

In der Abb. 4.6-2 stellt das weiße Modell eine 1:m-Assoziation dar, bei der die Attribute über die Ausleihe dem Buchexemplar zugeordnet wurden. Beim blauen Modell sind diese Attribute bei der assoziativen Klasse der 1:m-Assoziation eingetragen. Beide Modellierungen zeigen einen Schnappschuß. Welches Modell ist vorzuziehen? Ich wähle hier die weiße Modellbildung, weil die Ausleihe eine inhärente Eigenschaft eines Buchexemplars in einer Leihbibliothek

## LE 7 4 Checklisten zum Erstellen eines OOA-Modells

Abb. 4.6-2:  
Zuordnung der  
Attribute



ist und die Menge der Attribute in Buchexemplar eher gering ist. Bei der grauen Modellbildung handelt es sich um eine m:m-Assoziation. Hier muß eine (assoziative) Klasse verwendet werden, weil die Historie modelliert wird und sich jedes Ausleihe-Objekt auf genau einen Leser und ein Buchexemplar bezieht.

### 7 Klassenattribute

Klassenattribute beschreiben Eigenschaften, die für alle Objekte der Klasse gelten. Sie werden verwendet, wenn alle Objekte einer Klasse für ein bestimmtes Attribut immer denselben Wert besitzen sollen, z.B. wenn alle Hilfskräfte denselben Stundenlohn erhalten sollen. Es kann sich auch um eine Information über die Gesamtheit der Objekte handeln, die keinem einzelnen Objekt sinnvoll zugeordnet werden kann, z.B. die Anzahl der Objekte in einer Klasse.

### 8 Schlüsselattribute

Ein Schlüsselattribut identifiziert jedes Objekt innerhalb einer Klasse eindeutig. Schlüsselattribute werden nur dann eingetragen, wenn sie – unabhängig von ihrer identifizierenden Eigenschaft – Bestandteil des Fachkonzepts sind. Sie dürfen nicht verwendet werden, um auszudrücken, auf welche anderen Objekte sich ein bestimmtes Objekt bezieht. Objekte kennen einander ausschließlich über die Objektverbindungen.

### 9 abgeleitete Attribute

In der Klasse Seminarveranstaltung werden für die Implementierung nur zwei der drei Attribute gebraucht, es sind jedoch alle drei Attribute an der Benutzungsoberfläche sichtbar. Wir verwenden daher die in Abb. 4.6-3 gezeigte Modellierung. Der Programmierer kann später entscheiden, ob er lieber eine Datenredundanz in Kauf

**Ergebnisse****■ Klassendiagramm**

Tragen Sie für jedes Attribut den Namen in das Klassendiagramm ein. Kennzeichnen Sie Klassenattribute und abgeleitete Attribute.

**■ Attributspezifikation**

Spezifizieren Sie jedes Attribut wie in Kapitel 2.3 angegeben.

Für komplexe Attribute sind ggf. entsprechende Typen zu definieren.

**Konstruktive Schritte****1 Welche Attribute lassen sich mittels Dokumentanalyse identifizieren?**

- Einfache Attribute sind ggf. zu Datenstrukturen zusammenzufassen.
- Prüfen, ob alle Attribute wirklich notwendig sind.

**2 Welche Attribute lassen sich anhand der Beschreibung der Geschäftsprozesse identifizieren?**

- Benötigte Daten zur Ausführung der Aufgaben eines Geschäftsprozesses.
- Benötigte Daten für Listenfunktionalität.

**Analytische Schritte****3 Ist der Attributname geeignet?**

Der Attributname soll

- kurz, eindeutig und verständlich im Kontext der Klasse sein,
- ein Substantiv oder Adjektiv-Substantiv sein (kein Verb!),
- den Namen der Klasse nicht wiederholen (Ausnahme: feststehende Begriffe),
- bei komplexen (strukturierten) Attributen der Gesamtheit der Komponenten entsprechen,
- nur fachspezifische oder allgemein übliche Abkürzungen enthalten.

**4 Klasse oder komplexes Attribut?**

- Klasse: Objektidentität, gleichgewichtige Bedeutung im System, Existenz unabhängig von der Existenz anderer Objekte, Zugriff in beiden Richtungen grundsätzlich möglich.
- Attribut: keine Objektidentität, Existenz abhängig von Existenz anderer Objekte, Zugriff immer über das Objekt, untergeordnete Bedeutung.

**5 Wurde das richtige Abstraktionsniveau gewählt?**

- Wurden komplexe Attribute gebildet?
- Bilden komplexe Attribute geeignete Datenstrukturen?
- Ist die Anzahl der Attribute pro Klasse angemessen?

**6 Gehört das Attribut zu einer Klasse oder einer Assoziation?****7 Liegen Klassenattribute vor?**

Ein Klassenattribut liegt vor, wenn gilt:

- Alle Objekte der Klasse besitzen für dieses Attribut denselben Attributwert.
- Es sollen Informationen über die Gesamtheit der Objekte modelliert werden.

**8 Sind Schlüsselattribute fachlich notwendig?**

Schlüsselattribute werden nur dann eingetragen, wenn sie – unabhängig von ihrer identifizierenden Eigenschaft – Bestandteil des Fachkonzepts sind.

**Tab. 4.6-1 a:**  
**Checkliste**  
**Attribute**

**Tab. 4.6-1b:**  
**Checkliste**  
**Attribute**

- 9 Werden abgeleitete Attribute korrekt verwendet?**
- Information ist für den Benutzer sichtbar.
  - Lesbarkeit wird verbessert.
- 10 Wann wird ein Attribut nicht eingetragen?**
- Es handelt sich um ein Attribut, das den internen Zustand eines Lebenszyklus beschreibt und außerhalb des Objekts nicht sichtbar ist.
  - Es beschreibt Entwurfs- oder Implementierungsdetails.
  - Es handelt sich um ein abgeleitetes Attribut, das nur aus *Performance*-Gründen eingefügt wurde.
- 11 Fehlerquellen**
- Verwenden atomarer Attribute anstelle von komplexen Datenstrukturen.
  - Formulieren von Assoziationen als Attribute (Fremdschlüssel!).

nimmt – und für Konsistenz sorgen muß – oder das dritte Attribut aus den beiden anderen berechnet.

Abb. 4.6-3:  
Abgeleitetes  
Attribut

<b>Seminarveranstaltung</b>
Beginn
Ende
/Dauer

## 4.7 Checkliste Vererbung

Während jedes OOA-Modell Assoziationen enthält, können nur wenige oder keine Vererbungsstrukturen darin vorkommen. In der Analysephase sollen Vererbungsstrukturen Zusammenhänge und Unterschiede von Klassen im fachlichen Konzept deutlich machen. Sie werden nicht verwendet, um ein paar Attribute an Unterklassen zu vererben, sondern nur dann, wenn sie das Modell verbessern.

### Konstruktive Schritte zum Identifizieren von Vererbungsstrukturen

- 1 Generalisierung** Vererbungsstrukturen können prinzipiell mittels Generalisierung oder mittels Spezialisierung identifiziert werden. Bei der Generalisierung prüfen Sie für zwei oder mehrere Klassen, ob sie genügend Gemeinsamkeiten besitzen, damit sich eine neue Oberklasse bilden läßt.
- 2 Spezialisierung** Bei der Spezialisierung gehen wir von den allgemeineren Klassen aus und suchen nach spezialisierten Klassen. Betrachten Sie eine Klasse und prüfen Sie für jedes ihrer Objekte, ob dieses Objekt alle Attribute mit Werten besetzt, und ob jede Operation angewendet werden kann.

**Beispiel** In der Abb. 4.7-1 besitzen die Attribute *Teilnehmerzahl* und *Nettopreis* nur dann einen Wert, wenn es sich um eine öffentliche Veranstaltung handelt, während *Auftraggeber* und *Pauschalpreis* nur im



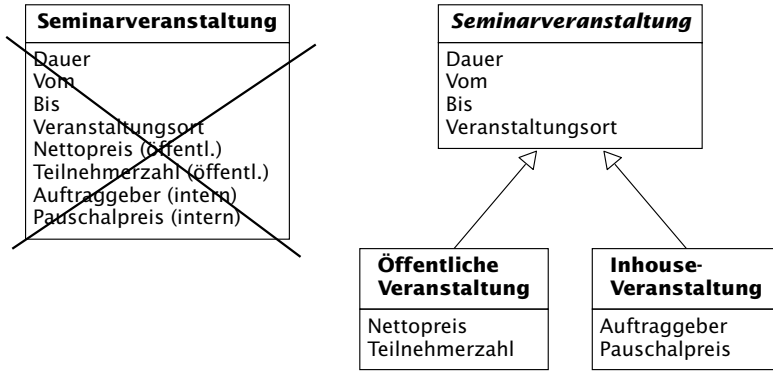


Abb. 4.7-1:  
Vererbung durch  
Spezialisierung

Fall einer *Inhouse-Veranstaltung* Werte annehmen. Die Klasse *Seminarveranstaltung* wird daher um die Unterklassen *Öffentliche Veranstaltung* und *Inhouse-Veranstaltung* ergänzt.

**Analytische Schritte zum Validieren der Vererbungsstrukturen**

Nur »gute« Vererbungsstrukturen können ein Klassendiagramm verbessern. Das bedeutet:

3 »gute« Vererbung

- a** Jede Unterklasse soll die geerbten Attribute und Assoziationen der Oberklasse auch benötigen, d.h. jedes Objekt der Unterklasse belegt die geerbten Attribute mit Werten und kann entsprechende Verbindungen besitzen. Diese Art der Modellierung führt zu tiefen Vererbungsstrukturen. Hier ist kritisch abzuwägen, ob nicht Unterklassen Attribute und/oder Assoziationen besitzen können, die nicht von allen Objekten benötigt werden und dadurch eine flachere Hierarchie erreicht wird.
- b** Eine ISA-Beziehung liegt vor, d.h. ein Objekt der Unterklasse »ist ein« Objekt der Oberklasse. Diese Beziehung ist transitiv. Die ISA-Beziehung macht deutlich, daß es für eine gute Vererbungsstruktur nicht ausreicht, wenn die Unterklasse zu den geerbten Attributen und Operationen eigene Attribute und Operationen hinzufügt.
- c** Modellierung des Problembereichs  
Das OOA-Modell dient entweder zur direkten Kommunikation mit dem Anwender oder es wird mit seiner Hilfe eine Benutzungsoberfläche erstellt. Daher soll die hier entwickelte Struktur den »natürlichen« Strukturen des Problembereichs entsprechen.
- d** Flache Hierarchien  
Die Vererbungshierarchie sollte nicht zu tief sein, denn um eine Unterklasse zu verstehen, müssen alle ihre Oberklassen betrachtet werden. Bis zu einer Tiefe von drei Ebenen gibt es normalerweise keine Verständnisprobleme, bei der Tiefe von fünf oder sechs Ebenen können bereits Schwierigkeiten auftreten /Rumbaugh 91/.

**Tab. 4.7-1:**  
**Checkliste**  
**Vererbung**

### Ergebnis

#### ■ Klassendiagramm

Tragen Sie alle Vererbungsstrukturen in das Klassendiagramm ein. Bilden Sie abstrakte Klassen.

### Konstruktive Schritte

#### 1 Ergibt sich durch Generalisierung eine Einfachvererbung ?

- Gibt es gleichartige Klassen, aus denen sich eine neue Oberklasse bilden läßt?
- Ist eine vorhandene Klasse als Oberklasse geeignet?

#### 2 Ergibt sich durch Spezialisierung eine Einfachvererbung?

- Kann jedes Objekt der Klasse für jedes Attribut einen Wert annehmen?
- Kann jede Operation auf jedes Objekt der Klasse angewendet werden?

### Analytische Schritte

#### 3 Liegt eine »gute« Vererbungsstruktur vor?

- Verbessert die Vererbungsstruktur das Verständnis des Modells?
- Benötigt jede Unterklasse alle geerbten Attribute, Operationen und Assoziationen?
- Ist die ISA-Beziehung erfüllt?
- Entspricht die Vererbungsstruktur den »natürlichen« Strukturen des Problembereichs?
- Besitzt sie maximal drei bis fünf Hierarchiestufen?

#### 4 Wann liegt keine Vererbung vor?

- Die Unterklassen bezeichnen nur verschiedene Arten, unterscheiden sich aber weder in ihren Eigenschaften noch in ihrem Verhalten.

**Aggregation (aggregation)** Eine Aggregation ist ein Sonderfall der →Assoziation. Sie liegt dann vor, wenn zwischen den Objekten der beteiligten Klassen eine Beziehung vorliegt, die sich als »ist Teil von« oder »besteht aus« beschreiben läßt.

**Assoziation (association)** Eine Assoziation modelliert Verbindungen zwischen Objekten einer oder mehrerer →Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwischen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch →Kardinalitäten und einen optionalen Assoziationsnamen oder Rollennamen. Sie kann um Restriktionen ergänzt werden. Besitzt eine Assoziation selbst wieder Attribute und ggf. Operationen und Assoziationen zu anderen Klassen, dann wird sie zur assoziativen Klasse. Die Qualifikationsangabe (*qualifier*) zerlegt die Menge der Objekte am anderen Ende

der Assoziation in Teilmengen. Eine abgeleitete Assoziation liegt vor, wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sonderfälle der Assoziation sind die →Aggregation und die →Komposition. In der Analyse ist jede Assoziation inhärent bidirektional.

**Attribut (attribute)** Attribute beschreiben Daten, die von den Objekten der →Klasse angenommen werden können. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch im allgemeinen unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten Typ und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muß jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig. Abgeleitete Attribute lassen sich aus anderen Attributen berechnen oder ableiten.



**Kardinalität (*multiplicity*)** Die Kardinalität bezeichnet die Wertigkeit einer →Assoziation, d.h. sie spezifiziert die Anzahl der an der Assoziation beteiligten Objekte.

**Klasse (*class*)** Eine Klasse definiert für eine Kollektion von Objekten deren Struktur (Attribute), das Verhalten (Operationen) und Beziehungen (Assoziationen, Vererbungsstrukturen). Klassen besitzen – mit Ausnahme von abstrakten Klassen – einen Mechanismus, um neue Objekte zu erzeugen. Der Klassenname muß mindestens im Paket, besser im gesamten System eindeutig sein.

**Komposition (*composition*)** Die Komposition ist eine besondere Form der →Aggregation. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch einem anderen Ganzen zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile.

**Statisches Modell** Das statische Modell realisiert außer den Basiskonzepten (Objekt, Klasse, Attribut) die statischen Konzepte (Assoziation, Vererbung, Paket). Es beschreibt die →Klassen des Systems, die →Assoziationen zwischen den Klassen und die Vererbungsstrukturen. Außerdem enthält es die Daten des Systems (Attribute). Die Pakete dienen dazu, Teilsysteme zu bilden, um bei großen Systemen einen besseren Überblick zu ermöglichen.

**Vererbung (*generalization*)** Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren →Klasse (Basisklasse) und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der →Attribute, Operationen und →Assoziationen der Basisklasse. Operationen der Basisklasse dürfen redefiniert werden. Es entsteht eine Klassenhierarchie oder Vererbungsstruktur. Außer der Einfachvererbung, bei der Klassen eine Baumstruktur bilden, gibt es die Mehrfachvererbung (Netzstruktur).



Die Checkliste **Klasse** zeigt, wie Klassen mittels Dokumentanalyse, Geschäftsprozessen und Kategorien identifiziert werden. Analog lassen sich **Assoziationen** identifizieren. Weitere Checklisten helfen, die richtigen **Kardinalitäten** zu ermitteln und zu entscheiden, ob eine **Komposition**, eine **Aggregation** oder eine einfache Assoziation vorliegt. Die Checkliste **Attribute** unterstützt beim Modellieren der Attribute. Die Checkliste **Vererbung** zeigt, wie Vererbungsstrukturen entwickelt werden.



**1 Lernziel: Anwenden der Dokumentanalyse und Erstellen eines Klassendiagramms.**

Die Abb. LE7-A1 auf der folgenden Seite zeigt einen Ausschnitt aus einer Praktikumsliste. Für jeden Praktikumstermin wird eine solche Liste erstellt, in der jeweils eingetragen wird, welche Studenten erfolgreich teilgenommen haben.

Identifizieren Sie alle Klassen und Assoziationen. Tragen Sie bei jeder Klasse diejenigen Attribute ein, die zur Identifikation geführt haben. Spezifizieren Sie die Assoziationen möglichst vollständig.

Aufgabe  
10–15 Minuten

## LE 7 Aufgaben

Abb. LE7-A1:  
Ausschnitt  
aus einer  
Praktikumsliste

Praktikumsliste				
Vorlesung:	41332	Gruppe:	B	
Bezeichnung:	Softwaretechnik	Termin:	5.11.1997, 14.00 Uhr	
Dozent:	Prof. Dr. Balzert	Raum:	4.6.09	
Projekt	Sprache	Student	Matrikelnr	Teilnahme
Stechuhr	C++	Müller	7001234	ok
		Mayer	7004567	ok
Roboter	Java	Schmidt	7009876	ok
		Winter	7002468	fehlt

Aufgabe  
15 Minuten

### 2 Lernziel: Assoziationen vollständig spezifizieren können.

Modellieren Sie folgende Problemstellungen durch Klassen und Assoziationen.

- Auf einer Palette sind mehrere Fässer. Paletten werden im Normalfall komplett geliefert und im Normalfall komplett weiterverkauft. Es können jedoch auch einzelne Fässer verkauft werden. Leere Paletten und einzelne Fässer können nicht vorkommen.
- Jedes Projekt hat genau einen Projektleiter, der für mehrere Projekte verantwortlich sein kann. Die meisten Programmierer arbeiten für mehrere Projekte. Einige Programmierer sind gleichzeitig Projektleiter eines anderen Projekts.
- Jede Hauptabteilung besteht aus mehreren Abteilungen. Die meisten Abteilungen sind in einer Hauptabteilung eingegliedert, die anderen (z.B. Stabsabteilungen) berichten direkt an die Geschäftsleitung. In einer Abteilung sind mehrere Mitarbeiter tätig. Jeder Mitarbeiter ist genau einer Abteilung zugeordnet.
- Ein Lexikon besteht aus mehreren Bänden. Erscheint eine neue Auflage des Lexikons, so gilt dies für alle Bände. Jedes Buch kann Teil einer Buchreihe sein.

Aufgabe  
5–10 Minuten

### 3 Lernziel: Systematisches Identifizieren von Vererbungsstrukturen und Assoziationen.

Erstellen Sie ein Klassendiagramm für folgende Problemstellung. Für Paletten ist eine Lagerverwaltung zu organisieren. Eine Palette kann in einem offenen Lager (z.B. eine große Lagerhalle) stehen. Für jedes offene Lager sind dessen Bezeichnung, der Standort, das Lagerprofil (z.B. Kühlung vorhanden) zu speichern. Eine Palette kann alternativ auf einem Stellplatz in einem Stellplatzlager gelagert werden. Für jeden Stellplatz, der mehrere Paletten aufnehmen kann, sind festzuhalten: Koordinaten und Angabe, ob

er frei oder belegt ist. Für das Stellplatzlager sind prinzipiell die gleichen Informationen zu speichern wie für das offene Lager, jedoch bezieht sich das Lagerprofil immer auf einzelne Stellplätze. Paletten sollen auch ohne Zuordnung zu einem Lager erfaßt werden.

**4** *Lernziel: Systematisches Erstellen eines Klassendiagramms mittels Geschäftsprozessen.* Aufgabe  
30 Minuten

Aus der Aufgabe 4 der Lehreinheit 6 ist das Klassendiagramm abzuleiten. Zusätzlich erhält der Systemanalytiker folgende Informationen:

- Für jeden Veranstaltungstyp sind ein Kürzel, eine Bezeichnung und die Angabe, ob es eine Pflichtveranstaltung ist, zu speichern.
- Für den Dozenten, der mehrere Veranstaltungen durchführen kann, werden dessen Name, Fachgebiet und Telefon gespeichert.
- Für die Verwaltung der Räume müssen die Anzahl der Plätze und die Ausstattung angegeben werden. In einem Raum finden im allgemeinen mehrere Veranstaltungen statt.
- Für jede Prüfung werden das Datum, die Zeit und der Raum, in dem sie stattfindet, festgehalten. In einem Raum finden im allgemeinen mehrere Prüfungen statt.
- Jeder Student bearbeitet am Ende seines Studiums genau eine Diplomarbeit, über die das Thema und das Datum der Abgabe gespeichert werden sollen. An einer Diplomarbeit können mehrere Studenten arbeiten, die alle am gleichen Tag abgeben müssen. Jede Diplomarbeit wird von genau einem Dozenten betreut, der im allgemeinen mehrere Diplomarbeiten vergibt.
- Über die Firmen, die im allgemeinen mehrere Studenten als Praktikanten beschäftigen, werden der Firmenname, die Anzahl der Mitarbeiter und die Branche gespeichert.

## 4 Checklisten zum Erstellen eines OOA-Modells (Dynamisches Modell)



- Szenarios systematisch erstellen können.
- Zustandsautomaten (Lebenszyklen) systematisch erstellen können.
- Operationen systematisch beschreiben können.
- Konsistenz von Klassendiagramm und dynamischem Modell prüfen können.
- Verfahren der formalen Inspektion anwenden können.

anwenden



- Die objektorientierten Konzepte und die UML-Notation, wie sie in Kapitel 2 beschrieben werden, müssen bekannt sein.
- Die Kenntnisse der Kapitel 1 und 3.1 sind nützlich.
- Aus dem Kapitel 3 sollten Sie mindestens ein Fallbeispiel durchgearbeitet haben.
- Eine weitere Voraussetzung bilden die Kapitel 4.1 bis 4.7.

- i 4.8 [Checkliste Szenario](#) 170
- 4.9 [Checkliste Zustandsautomat](#) 177
- 4.10 [Checkliste Operation](#) 183
- 4.11 [Formale Inspektion](#) 185

## 4.8 Checkliste Szenario

**Definition** Szenarios sind Verfeinerungen der Geschäftsprozesse, die in Form von Interaktionsdiagrammen dokumentiert werden. Die Bedeutung von Szenarios kann gar nicht überschätzt werden. Szenarios unterstützen Sie darin, die **Operationen** der Klassen zu identifizieren und den Fluß der Botschaften durch das System zu definieren. Außerdem dienen sie dazu, die Vollständigkeit und Korrektheit des statischen Modells zu validieren. Darüber hinaus lassen sich die beschriebenen Szenarios gleichzeitig als Test-Szenarios verwenden.

### Konstruktive Schritte zum Modellieren von Szenarios

1 Geschäftsprozeß  
→ Szenarios

**Szenarios** werden direkt aus den Geschäftsprozessen abgeleitet. Wählen Sie einen Geschäftsprozeß aus und überlegen Sie, welche Variationen auftreten können. Jede Variation führt zu einem unterschiedlichen Ergebnis des Geschäftsprozesses und bildet ein Szenario. Auch wenn eine Anwendung eine überschaubare Anzahl von Geschäftsprozessen enthält, kann es eine Vielzahl möglicher Szenarios geben. Es ist oft notwendig, sich auf die wichtigsten Szenarios zu beschränken. Im Gegensatz zum vollständigen statischen Modell beschreiben die Szenarios nur das wesentliche dynamische Verhalten.

Die Szenarios, die sich aus einem Geschäftsprozeß ableiten lassen, sind nicht alle von gleicher Bedeutung für die Modellbildung.

Es lassen sich primäre und sekundäre Szenarios unterscheiden. Primäre Szenarios stellen die fundamentalen bzw. die am häufigsten verwendeten Funktionen des Systems dar. Sekundäre Szenarios präsentieren Variationen primärer Szenarios. Sie beschreiben Ausnahmesituationen und enthalten die weniger oft verwendeten Funktionen.

Darüber hinaus lassen sich die Szenarios in positive und negative Fälle unterteilen. Ein positiver Fall bedeutet eine erfolgreiche Bearbeitung, z.B. kann im Fall der Seminarorganisation die Anmeldung erfolgreich durchgeführt werden. Negative Fälle liegen bei allen erfolglosen Ausgängen vor, z.B. kann eine Seminaranmeldung – aus welchen Gründen auch immer – nicht durchgeführt werden.

Szenarios können – unabhängig von der später verwendeten Notation – zunächst in folgender Form dokumentiert werden:

Dokumentation  
Szenario

- Name des Szenarios.
- Bedingungen, die zu dieser Variation des Geschäftsprozesses führen (Unter welchen Voraussetzungen wird dieses Szenario ausgeführt?).
- Ergebnis des Szenarios.

Beschreiben Sie zunächst nicht, *wie* das Szenario abläuft. Verwenden Sie hier noch keine Objekte oder Attribute.

Aus dem Geschäftsprozeß bearbeite Schadensfall I aus Kapitel 4.2 Beispiel  
leiten wir folgende Szenarios ab:

*Szenario 1:* bearbeite Schadensfall (Schaden bezahlt)

Bedingungen:

- notwendige Daten vorhanden
- Police gültig
- Police deckt Schaden ab

Ergebnis:

- Schadensersatzanspruch wird voll beglichen

*Szenario 2:* bearbeite Schadensfall (Police ungültig)

Bedingungen:

- notwendige Daten vorhanden
- Police ungültig

Ergebnis:

- Antragsteller erhält Schreiben

*Szenario 3:* bearbeite Schadensfall (unvollständige Deckung)

Bedingungen:

- notwendige Daten vorhanden
- Police gültig
- Police deckt Schadensersatzforderung nur unvollständig ab
- Erfolgreiches Verhandeln mit Antragsteller

Ergebnis:

- Schaden entsprechend Vergleich bezahlt

Achten Sie darauf, *alle* Bedingungen, die zu einem Szenario führen, explizit anzugeben. Oft lassen sich neue Szenarios durch Variationen der schon dokumentierten Szenarios finden. In diesem Fall sollten Sie das ursprüngliche Szenario um die neue Bedingung ergänzen.

Jedes relevante Szenario wird durch ein Interaktionsdiagramm dokumentiert. Es beschreibt, *wie* die Objekte der beteiligten Klassen durch Senden von Botschaften das Szenario ausführen.

2 Kommunikation  
der Objekte

Die UML bietet Sequenz- und Kollaborationsdiagramme an. Ich ziehe Sequenzdiagramme vor, um ein vollständiges Szenario übersichtlich zu beschreiben und verwende das Kollaborationsdiagramm, um das Zusammenwirken ausgewählter Objekte deutlich zu machen.

- Überlegen Sie, welche Klassen bzw. Objekte an dem Szenario beteiligt sind. Fügen Sie diese Objekte in das Diagramm ein. Die meisten Klassen, die Sie hier verwenden, sind bereits im Klassendiagramm enthalten. Eventuell finden Sie auch neue Klassen, die dann im statischen Modell nachzutragen sind.
- Wenn mehrere Objekte einer Klasse verschiedene Aufgaben im Szenario haben, dann wird jedes Objekt aufgeführt und entsprechend seiner Bedeutung benannt, z.B. können beim Kopieren von einem Verzeichnis in ein anderes die Namen *Quelle: Verzeichnis* und *Ziel: Verzeichnis* verwendet werden.



## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

- Zerlegen Sie die Aufgaben des Geschäftsprozesses in Teile, bis jeder Teil einer Operation entspricht. Fragen Sie immer wieder »Was ist nun zu tun?« und »Wer führt es aus?«
- Die UML erlaubt es, im **Sequenzdiagramm** Bedingungen anzugeben. Damit können mehrere Variationen durch ein einziges Sequenzdiagramm beschrieben werden.

Beispiel Abb. 4.8-1 zeigt das Sequenzdiagramm für folgendes Szenario der Seminarorganisation, dessen Geschäftsprozeß in Kapitel 3.5 beschrieben ist.

*Szenario:* bearbeite Anmeldung (positiver Fall)

*Bedingungen:*

- Seminar existiert
- Neuer Kunde

*Ergebnis:*

- Anmeldebestätigung erstellt

Abb. 4.8-1:  
Sequenzdiagramm  
für eine erfolgreiche  
Anmeldung  
eines neuen  
Kunden zum  
Seminar

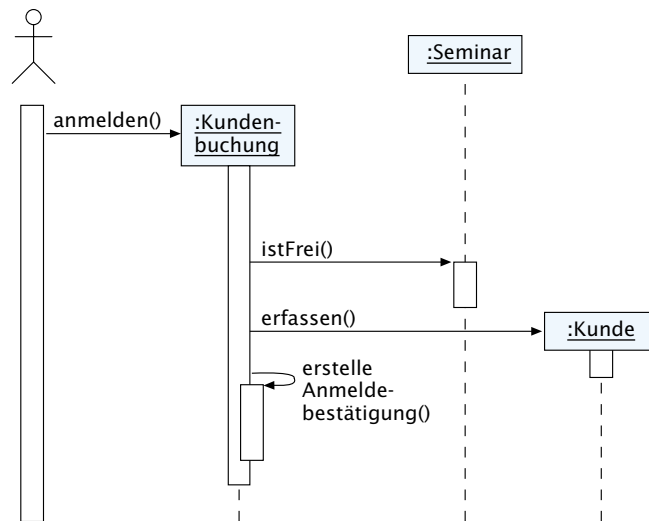


Abb. 4.8-2 zeigt das Sequenzdiagramm für folgendes Szenario der Friseursalonverwaltung, dessen Geschäftsprozeß in Kapitel 3.4 beschrieben ist. Bei diesem Szenario werden mehrere externe Operationen in festgelegter Reihenfolge ausgeführt. Zuerst wird für einen angemeldeten Kunden ein Laufzettel ausgedruckt, der im Rahmen der Behandlung ausgefüllt wird. Diese Daten werden anschließend ins System übernommen. Beim Kassieren der erbrachten Dienstleistungen können auch Artikel verkauft werden.

*Szenario:* bediene Kunden im Salon (einschließlich Verkäufe)

*Bedingungen:*

- Kunde ist angemeldet

*Ergebnis:*

- Kunde hat bezahlt

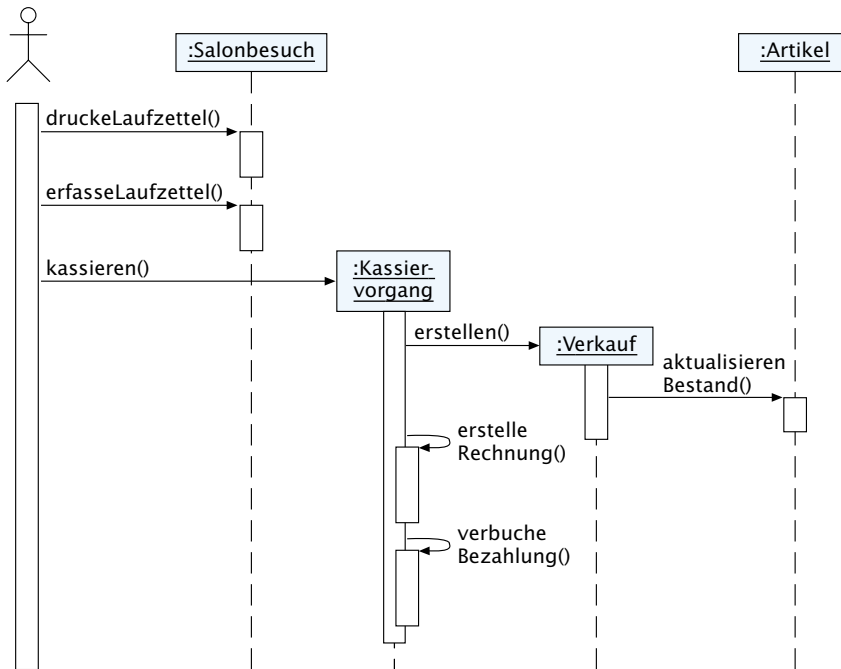


Abb. 4.8-2:  
Sequenzdiagramm für die  
Bedienung eines  
angemeldeten  
Kunden im  
Friseursalon

Bei der Erstellung der Interaktionsdiagramme in der Analyse kommt es weniger darauf an, den exakten Fluß der Botschaften durch das System zu ermitteln, wie er später durch das System ausgeführt wird, sondern das fachliche Verhalten soll möglichst präzise beschrieben werden. Dennoch verfolgen wir das Ziel, Operationen im Interaktionsdiagramm konsistent mit dem Klassendiagramm zu halten. Bei der Zuordnung der Operationen zu den Klassen sind folgende Regeln nützlich:

- Werden nur Attribute einer Klasse benötigt, dann ist die Operation hier zuzuordnen.
- Konstruktoroperationen ordnen Sie der jeweiligen Klasse selbst und bei einer Komposition der Aggregatklasse zu.
- Eine analoge Zuordnung ergibt sich für das Löschen von Objekten.

Ein Szenario kann aus einer oder mehreren Transaktionen bestehen (Abb. 4.8-3). Jede Benutzerfunktion, die weitere auslösen kann, bildet eine Transaktion. Außerdem haben Sie nach /Jacobson 92/ einen Gestaltungsspielraum, der sich zwischen zwei Extremen bewegt (Abb. 4.8-4).

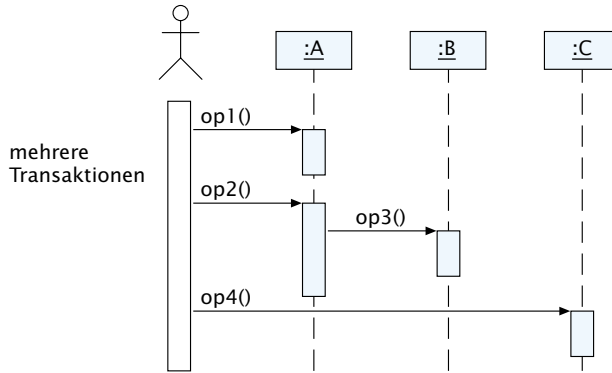
Bei der zentralen Organisation gibt es ein Objekt, das die Steuerung aller anderen Objekte übernimmt (*fork diagram*). Dieses Objekt verkapselt das Wissen, welche Operationen in welcher Reihenfolge aufgerufen werden. Wenn sich diese Abläufe häufig ändern,

3 Operationen →  
Klasse

4 Struktur des  
Szenarios

## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

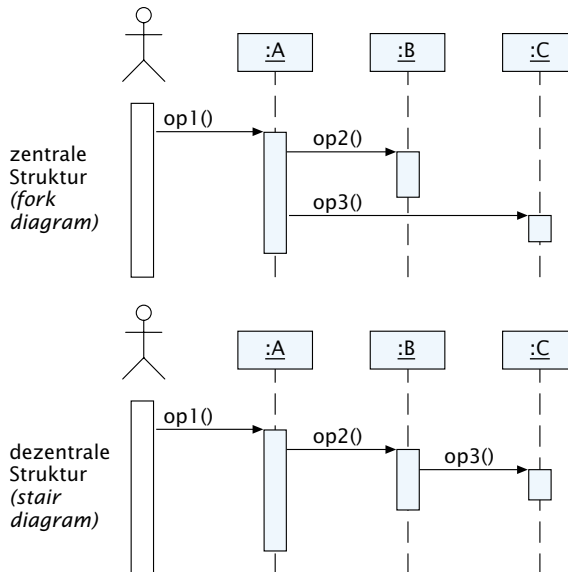
Abb. 4.8-3:  
Szenario bestehend  
aus mehreren  
Transaktionen



dann wirken sich diese Änderungen nur auf das Steuerungsobjekt aus. In einer dezentralen Struktur kommuniziert jedes beteiligte Objekt lediglich mit einem Teil der anderen Objekte (*stair diagram*). Bei dieser Organisationsform ist die Steuerungslogik im System verteilt. Sie vermeidet, daß einige Objekte besonders aktiv und andere übermäßig passiv sind.

Überlegen Sie für jedes Ihrer Interaktionsdiagramme, ob eine dieser Strukturen zutrifft oder ob sie einen Mittelweg wählen, d.h. treffen Sie diese Entscheidung ganz bewußt. Eventuell sollten Sie mehrere Alternativen modellieren und im Team diskutieren. Gestalten Sie Interaktionsdiagramme so einfach wie möglich. Komplexe Diagramme sind gewöhnlich erst im Entwurf anzutreffen.

Abb. 4.8-4:  
fork diagram und  
stair diagram



**Analytische Schritte zum Validieren der Szenarios**

In den meisten Fällen existiert zwischen Klassen, deren Objekte miteinander kommunizieren, auch eine Assoziation, d.h. eine permanente Verbindung zwischen den Objekten. In diesem Fall kennt jedes Senderobjekt seine zugehörigen Empfängerobjekte. Es ist jedoch auch möglich, daß Objekte ohne eine solche permanente Verbindung kommunizieren können. Beispielsweise kann eine Operation die Identität eines Objekts ermitteln und als Ergebnis zurückliefern. Müssen Objekte miteinander kommunizieren, die nicht durch eine Assoziation miteinander verbunden sind, so ist zu prüfen, woher das Senderobjekt die Identität des Empfängerobjektes erhält. Sie können auf eine Assoziation verzichten, wenn sich die Objekte nicht permanent kennen müssen, weil eine entsprechende Kommunikation nur selten stattfindet und das gewünschte Objekt auch anders identifiziert werden kann.

5 Empfänger erreichbar?

Interaktionsdiagramme stellen eine Verbindung zum Klassendiagramm her. Sie lassen sich daher einerseits zur Modellbildung und andererseits zur Validierung des Klassendiagramms verwenden. Können die beteiligten Klassen die genannte Aufgabe wirklich gemeinsam lösen? Enthält das Interaktionsdiagramm nur Klassen, anonyme oder konkrete Objekte von Klassen des Klassendiagramms?

6 Konsistenz

Abb. 4.8-5 zeigt zwei Szenarios und den Ausschnitt aus dem Klassendiagramm aus dem Beispiel der Bibliotheksverwaltung.

Beispiel

Szenarios sollen keine Beschreibungen der Benutzungsoberfläche, z.B. *OkButtonGedrückt*, enthalten. Diese Informationen gehören in den Entwurf. Beschränken Sie sich auf die wichtigsten Szenarios. Beschreiben Sie nicht jeden Sonderfall. Vermeiden Sie die Überspezifikation von Szenarios, d.h. verzichten Sie auf unnötige Details.

7 Fehlerquellen

Szenario: Ausleihen von Büchern (registrierter Leser)

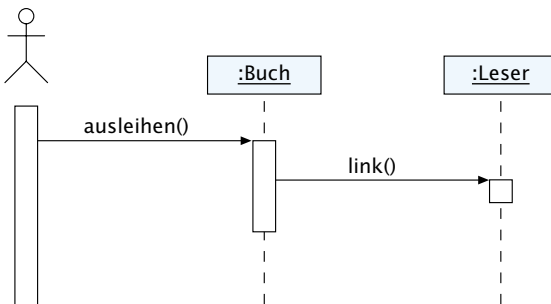
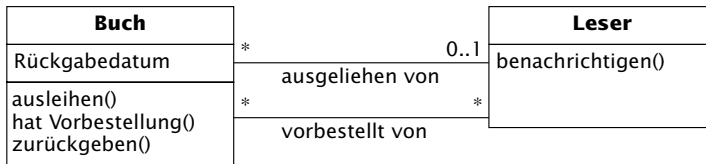
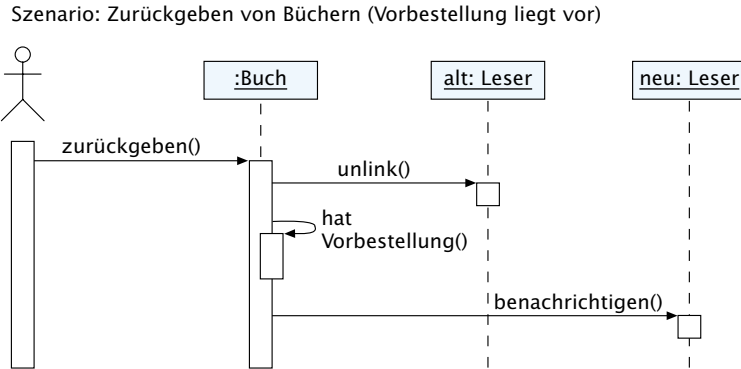


Abb. 4.8-5a: Szenarios und Klassendiagramm der Bibliotheksverwaltung

## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

Abb. 4.8-5b:  
Szenarios und  
Klassendiagramm  
der Bibliotheks-  
verwaltung



Tab. 4.8-1a:  
Checkliste  
Szenarios

### Ergebnisse

#### ■ Sequenzdiagramm, Kollaborationsdiagramm

Für jedes relevante Szenario ist ein Sequenzdiagramm zu erstellen. Alternativ können Kollaborationsdiagramme verwendet werden.

### Konstruktive Schritte

#### 1 Entwickeln Sie aus jedem Geschäftsprozeß mehrere Szenarios.

- Variationen von Geschäftsprozessen ermitteln.
- Standardausführung und Alternativen.
- positive und negative Fälle unterscheiden.
- Prüfen, welche Szenarios wichtig sind.
- Interaktionsdiagramme benennen und beschreiben.

#### 2 Wie läuft das Szenario ab?

- Beteiligte Klassen.
- Aufgaben in Operationen zerlegen.
- Reihenfolge der Operationen prüfen.

#### 3 Zu welcher Klasse gehören Operationen?

#### 4 Wie ist das Szenario zu strukturieren?

- Eine oder mehrere Transaktionen.
- Zentrale Struktur (*fork diagram*).
- Dezentrale Struktur (*stair diagram*).

### Analytische Schritte

#### 5 Sind die Empfänger-Objekte erreichbar?

- Assoziation existiert (permanente Verbindung).
- Identität kann dynamisch ermittelt werden (temporäre Verbindung).

**6 Ist das Sequenzdiagramm konsistent mit dem Klassendiagramm?**

- Alle Klassen sind auch im Klassendiagramm enthalten.
- Mit Ausnahme von impliziten Operationen werden nur Operationen aus dem Klassendiagramm eingetragen.

**7 Fehlerquellen**

- Benutzungsoberfläche beschreiben.
- Zu viele Details beschreiben.

**Tab. 4.8-1b:**  
**Checkliste**  
**Szenarios**

## 4.9 Checkliste Zustandsautomat

In der objektorientierten Analyse kann der **Zustandsautomat** besonders effektiv eingesetzt werden, um den Lebenszyklus einer Klasse zu spezifizieren. Diese Modellbildung unterstützt Sie darin, Operationen zu identifizieren und die Abhängigkeiten der Operationen in einer Klasse zu verstehen. Normalerweise ist nur für wenige Klassen der Lebenszyklus zu modellieren. Die erste Frage, die sich stellt, ist daher, für welche Klassen ein Zustandsautomat zu erstellen ist. /IBM 97/ gibt an, daß bei typischen Anwendungen (Informationssysteme) nur ein bis zwei Prozent der Klassen einen nicht-trivialen Lebenszyklus besitzen, während deren Anzahl bei Echtzeitanwendungen wesentlich zunimmt.

### Konstruktive Schritte zum Modellieren von Lebenszyklen

Ein Lebenszyklus ist sinnvoll, wenn eine der folgenden Situationen zutrifft:

- Ein Objekt kann auf eine bestimmte Botschaft – in Abhängigkeit vom aktuellen Zustand – unterschiedlich reagieren.
- Einige Operationen sind nur in bestimmten Situationen (Zuständen) auf ein Objekt anwendbar und werden sonst ignoriert.

Verzichten Sie auf den Lebenszyklus, wenn seine Beschreibung nichts zum Verständnis der Problematik beiträgt.

Bei der Seminarorganisation (Abb. 4.9-1) würde sich für den Seminartyp nur ein trivialer Lebenszyklus (Abb. 4.9-2) ergeben. Wir können daher auf diesen Zustandsautomaten verzichten. Bei einer Seminarveranstaltung können wir jedoch nur dann Anmeldungen eintragen, wenn diese Veranstaltung weder ausgebucht noch storniert ist. Auch wenn sich ein Kunde irrtümlich für eine bereits durchgeführte Veranstaltung anmeldet, soll die Anmeldung abgelehnt werden. Hier bildet der Lebenszyklus (Abb. 4.9-3) ein nützliches Hilfsmittel, um einerseits die Problematik besser zu verstehen und andererseits die Operationen vollständig zu ermitteln. Wir gehen im folgenden noch weiter auf diesen Zustandsautomaten ein.

1 nicht-trivialer  
Lebenszyklus

Beispiel

## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

Abb. 4.9-1: Seminarorganisation  
Klassendiagramm der Seminarorganisation

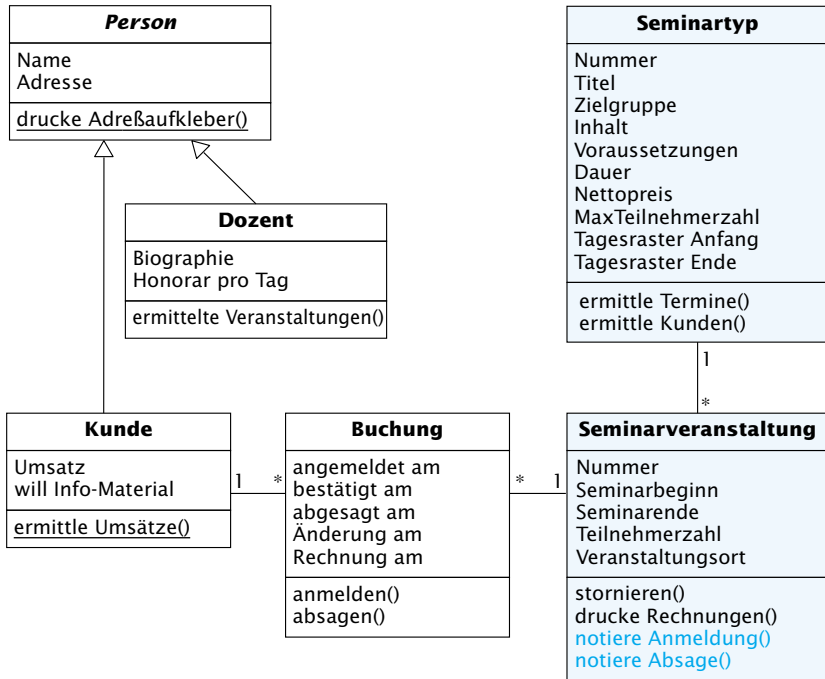
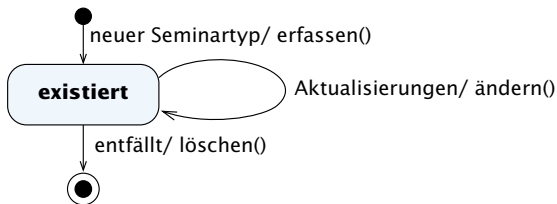


Abb. 4.9-2:  
Trivialer Zustands-  
automat der Klasse  
Seminarartyp



### 2 Arbeitstechnik

Bevor Sie mit der grafischen Modellierung des Lebenszyklus beginnen, sollten Sie die wichtigsten Informationen im Sinne eines *brain storming* notieren. Bilden Sie dazu drei Spalten. In der ersten Spalte tragen sie alle Zustände ein, in der zweiten alle Ereignisse und in der dritten Spalte alle Operationen dieser Klasse. Formulieren Sie die Ereignisse nicht als Botschaften, sondern beschreiben Sie umgangssprachlich »was von außen auf das Objekt einwirkt«.

### 3 Zustände

Beginnen Sie mit dem Anfangszustand, d.h. dem Zustand, in dem sich das Objekt befindet, nachdem es erzeugt wurde. Entwickeln Sie ausgehend vom Anfangszustand weitere Zustände. Betrachten Sie dazu jeweils einen Zustand und überlegen Sie durch welche Ereignisse er verlassen wird. In welche Folgezustände erfolgt ein Übergang? Prüfen Sie, wodurch ein Zustand definiert

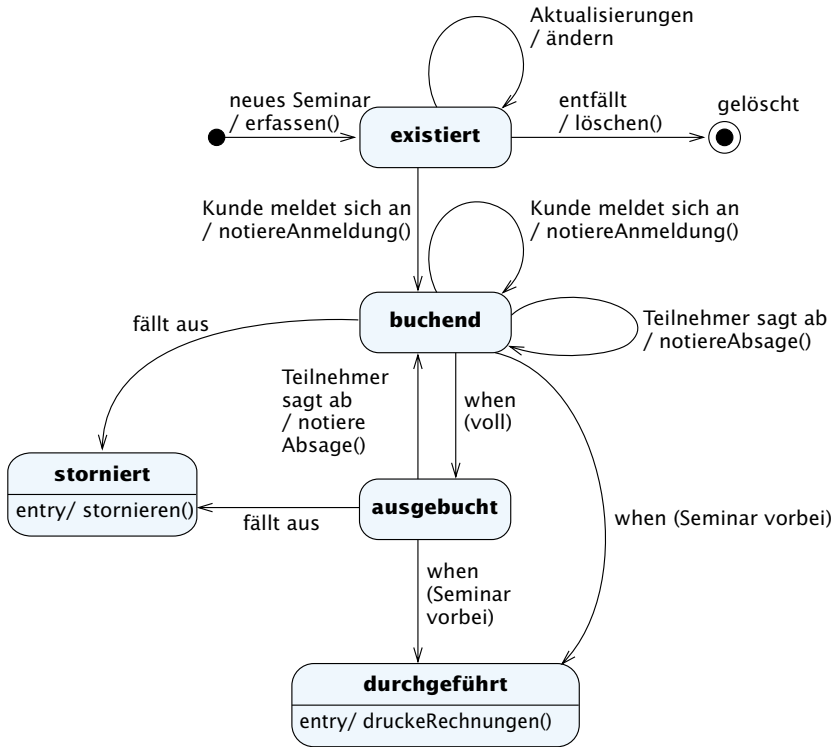


Abb. 4.9-3: Nicht-trivialer Zustandsautomat der Klasse Seminarveranstaltung

wird, z.B. das Vorliegen bestimmter Attributwerte oder Verbindungen zu anderen Objekten.

In der Abb. 4.9-3 löst das Ereignis *Kunde meldet sich an* im Zustand *existiert* einen Übergang in den Zustand *buchend* aus. Dieser Zustand wird durch folgende Attributwerte beschrieben:

Beispiel

$0 < Teilnehmerzahl < MaxTeilnehmerzahl$ . Im Zustand *buchend* kann das Objekt auf folgende Ereignisse reagieren und in die entsprechenden Folgezustände übergehen:

Seminar fällt aus → *storniert*,

Teilnehmer meldet sich an und Seminar noch nicht voll → *buchend*,

Teilnehmer sagt ab → *buchend*,

Seminar ist voll → *ausgebucht*,

Seminarbeginn heute → *durchgeführt*,

Einige Zustände werden auch durch Attributwerte ausgedrückt:

*existiert*:  $Teilnehmerzahl = 0$ ,

*ausgebucht*:  $Teilnehmerzahl = MaxTeilnehmerzahl$ .

Prüfen Sie, ob Endzustände existieren, d.h. ob es Zustände gibt, aus denen keine Transition herausführt. Dazu ist es hilfreich, zyklische und lineare Automaten zu unterscheiden. Endzustände existieren nur bei einem linearen Lebenszyklus (*born and die*) /Shlaer 92/.

4 Endzustände



## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

Bei diesem Automaten werden Objekte erzeugt und gelöscht. Eine Variante dieses Lebenszyklus liegt vor, wenn das Objekt nicht gelöscht, sondern »schlafen gelegt« wird. Bei einem zyklischen Zustandsautomaten (*circular lifecycle*) werden die Zustände iterativ durchlaufen. Es gibt keinen Endzustand. Dieser Lebenszyklus ist typisch für Geräte.

**Beispiel** Eine Seminarveranstaltung wird erfaßt, anschließend werden Anmeldungen und Absagen eingetragen. Wird die Seminarveranstaltung storniert oder durchgeführt, so ist das dynamische Verhalten des Objekts nicht länger von Interesse. Das Objekt wird also »schlafen gelegt«. Gelöscht werden darf das Objekt nur, wenn noch keine Anmeldungen durchgeführt wurden. Es handelt sich also um einen linearen Automaten.

**5 Operationen** Wenn Sie den Lebenszyklus mit den Zuständen und Transitionen dargestellt haben, sollten Sie die Operationen eintragen. Überlegen Sie, welche der bereits identifizierten Operationen der Klasse ein zustandsabhängiges Verhalten besitzen. Operationen, die in jedem Zustand ausgeführt werden können, sind nicht einzutragen. Normalerweise finden Sie in diesem Schritt noch weitere – interne – Operationen.

**Beispiel** Die Operation `notiereAnmeldung()` wird nur aktiviert, wenn die Veranstaltung existiert und eine Buchung möglich ist, d.h. die Veranstaltung weder ausgebucht, noch storniert oder bereits durchgeführt ist. Geändert oder gelöscht werden kann eine Seminarveranstaltung nur dann, wenn sie existiert, aber noch keine Buchung stattgefunden hat. Aus der Abb. 4.9-3 ist zu entnehmen, daß die Klasse Seminarveranstaltung um die internen Operationen `notiereAnmeldung()` und `notiereAbsage()` zu ergänzen ist. Diese beiden Operationen werden von den externen Operationen `Buchung.anmelden()` und `Buchung.absagen()` aktiviert. In der Abb. 4.9-1 sind diese beiden internen Operationen blau eingetragen.

**6 Aktion oder Aktivität** Um die Operationen in das **Zustandsdiagramm** eintragen zu können, müssen sie entscheiden, ob es sich um Aktivitäten oder um Aktionen handelt.

Ist die Operation atomar und nicht-unterbrechbar, dann wird sie als **Aktion** modelliert. Wenn eine Aktion an alle eingehenden Transitionen eines Zustandes einzutragen wäre, dann modellieren wir sie als *entry*-Aktion in diesem Zustand. Analog ist bei der *exit*-Aktion zu verfahren. Bei Informationssystemen liegen meistens Aktionen vor.

Eine **Aktivität** ist immer fest mit einem Zustand verbunden. Sie beginnt bei Eintritt in den Zustand und endet bei Verlassen des Zustandes. Sie kann alternativ durch eine Start-Aktion und eine Beende-Aktion ausgedrückt werden.

Beim Beispiel der Seminarverwaltung sind alle Operationen als Aktionen einzutragen. Beim Eintritt in den Zustand storniert wird immer die Operation stornieren() ausgeführt, unabhängig davon, über welche Transition der Übergang erfolgt. Analoges gilt beim Zustand durchgeführt. Daher werden die zugehörigen Operationen als entry-Aktionen modelliert. Alle anderen Aktionen werden an die Zustandsübergänge angetragen. Beispiel

Außer den externen Ereignissen, die vom Benutzer oder anderen Objekten ausgehen, sind auch zeitliche Ereignisse zu modellieren. Einige Ereignisse muß das Objekt selbst generieren, um Nicht-Endzustände zu verlassen. 7 Ereignisse

Überlegen Sie, welche Fehlersituationen auftreten können und wie das Objekt darauf reagieren soll. Unter einem Fehler verstehen wir hier jede Abweichung vom normalen bzw. gewünschten Verhalten. Ein besserer Begriff dafür ist Ausnahmebehandlung (*exception handling*). Beachten Sie, daß zuerst immer das Normalverhalten und erst im zweiten Schritt das Fehlerverhalten betrachtet wird.

Wird der letzte Platz belegt, dann muß die Operation notiere Anmeldung() das Ereignis when(vol1) erzeugen, um den Zustand buchend zu verlassen. Beispiel

### Analytische Schritte zum Validieren der Lebenszyklen

Der Zustandsname soll die Zeitspanne beschreiben, in der sich das Objekt in einem Zustand befindet. Wenn mit dem Zustand eine Verarbeitung (z.B. prüfe Karte) verbunden ist und der Zustandsname keine zusätzliche Information enthalten würde (z.B. prüfend Karte), dann kann er entfallen. 8 Zustandsname

Modelliert der Zustandsautomat den Lebenszyklus einer Klasse, dann müssen die Operationen konsistent mit dem Klassendiagramm sein. Das bedeutet, daß alle Aktivitäten und Aktionen auch Operationen des Klassendiagramms sind, wobei Verwaltungsoperationen in letzteres nicht eingetragen werden. Prüfen Sie, ob es für jede Operation des Klassendiagramms mindestens einen Zustand gibt, in dem das Objekt auf die entsprechende Botschaft reagieren kann. 9 Konsistenz

Alle Zustände müssen erreichbar sein. Besitzt ein Zustand keine ausgehende Transition, dann muß es sich um einen Endzustand handeln. Alle Transitionen, die von einem Zustand ausgehen, müssen mit unterschiedlichen Ereignissen beschriftet sein. Nur bei einer Transition darf das Ereignis fehlen, d.h. ein implizites Ereignis vorliegen. 10 Transitionen

Lassen Sie ihre Zustandsdiagramme nicht zu Programmablaufplänen bzw. Flußdiagrammen entarten. »Schleifen« dürfen in Zustandsdiagrammen nicht vorkommen. Bedingungen sind in Zustandsautomaten immer mit einem Ereignis verknüpft. 11 keine Flußdiagramme

**Tab. 4.9-1a:**  
**Checkliste**  
**Zustands-**  
**automaten**  
**(Lebenszyklen)**

### Ergebnis

#### ■ Zustandsdiagramm

Für jeden nicht-trivialen Lebenszyklus ist ein Zustandsdiagramm zu erstellen.

### Konstruktive Schritte

#### 1 Existiert ein nicht-trivialer Lebenszyklus?

- Das gleiche Ereignis kann – in Abhängigkeit vom aktuellen Zustand – unterschiedliches Verhalten bewirken.
- Operationen sind nur in bestimmten Situationen (Zuständen) auf ein Objekt anwendbar und werden sonst ignoriert.

#### 2 Arbeitstechnik

- 1. Spalte: Zustände.
- 2. Spalte: Ereignisse.
- 3. Spalte: Operationen.

#### 3 Welche Zustände enthält der Automat?

- Ausgangsbasis ist der Anfangszustand.
- Durch welche Ereignisse wird ein Zustand verlassen?
- Welche Folgezustände treten auf?
- Wodurch wird der Zustand definiert (Attributwerte, Objektverbindungen)?

#### 4 Existieren Endzustände?

Nur bei linearen Lebenszyklen kann folgendes zutreffen:

- Das Objekt hört auf zu existieren.
- Das Objekt existiert weiterhin, aber sein dynamisches Verhalten ist nicht länger von Interesse (schlafendes Objekt).

#### 5 Welche Operationen besitzt das Objekt?

- Jede zustandsabhängige Operation aus dem Klassendiagramm eintragen.
- Operationen, die in jedem Zustand ausgeführt werden können, nicht eintragen.
- Prüfen, ob weitere Operationen notwendig sind.

#### 6 Sind Operationen als Aktivitäten oder als Aktionen zu modellieren?

- Aktion = atomar, nicht-unterbrechbar (Transition, *entry*, *exit*).
- Aktivität = fest mit einem Zustand verbunden (Start-Aktion + Beende-Aktion).

#### 7 Welche Ereignisse sind zu modellieren?

- Externe Ereignisse:  
vom Benutzer,  
von anderen Objekten.
- zeitliche Ereignisse:  
Zeitdauer,  
Zeitpunkt.
- intern generierte Ereignisse des betrachteten Objekts.

### Analytische Schritte

#### 8 Geeigneter Zustandsname

- Beschreibt eine bestimmte Zeitspanne.
- Kein Verb.
- Kann entfallen, wenn er keine zusätzliche Information enthält.

**9 Ist der Objekt-Lebenszyklus konsistent mit der Liste der Operationen?**

- Gibt es für jede Operation mindestens einen Zustand, in dem das Objekt auf die entsprechende Botschaft reagieren kann?
- Sind alle Aktivitäten und Aktionen auch Operationen des Klassendiagramms?

**10 Sind alle Transitionen korrekt eingetragen?**

- Ist jeder Zustand erreichbar?
- Kann jeder Zustand – mit Ausnahme der Endzustände – verlassen werden?
- Sind die Transitionen eindeutig?

**11 Fehlerquellen**

- Modellierung der Benutzungsoberfläche im Lebenszyklus.
- Gedankengut aus den Programmablaufplänen übernommen.

**Tab. 4.9-1b:**  
**Checkliste**  
**Zustands-**  
**automaten**  
**(Lebenszyklen)**

## 4.10 Checkliste Operation

Operationen kommen nicht nur im dynamischen Modell vor, sondern werden auch ins Klassendiagramm eingetragen. Sie stellen daher eine Verbindung zwischen dem statischen und dem dynamischen Modell her.

### Konstruktive Schritte zum Identifizieren von Operationen

Außer den Operationen zur Ausführung des Szenarios besitzen die meisten Systeme auch Listenoperationen (*query operations*). Wichtige Listenoperationen werden in der Analyse eingetragen, damit die Vollständigkeit und Korrektheit des statischen Modells überprüft werden kann. Listenoperationen sind häufig Klassenoperationen. Geben Sie – bei Modellierungsalternativen – der Objektoperation stets den Vorzug gegenüber einer Klassenoperation. Listenoperationen müssen – in der Analyse – nicht vollständig angegeben werden.

Jedes System enthält darüber hinaus Verwaltungsoperationen. Außer den Basisoperationen (`new()`, `delete()`, `getAttribute()`, `setAttribute()`, `link()`, `unlink()`, `getlink()`), die niemals im Klassendiagramm angegeben werden, gibt es externe Verwaltungsoperationen (`erfassen()`, `ändern()`, `löschen()` und `erstelleListe()`). Diese Operationen werden bei komplexen Klassendiagrammen wegen der besseren Übersichtlichkeit nicht eingetragen.

In der Abb. 4.9-1 der Seminarorganisation ist `ermittleUmsätze()` eine Listenoperation, die *alle* Kunden betrifft. Daher ist sie eine Klassenoperation der Klasse `Kunde`. Die Operation `ermittleTermine()` betrifft alle Seminarveranstaltungen *eines* gegebenen Typs. Sie wird als Objektoperation der Klasse `Seminar` zugeordnet. Eine weitere Listenoperation ist `ermittleKunden()`. Sie prüft, welche Kunden bisher *einen* bestimmten Seminartyp gebucht haben. Sie wird daher

1 Listen- und Verwaltungsoperationen

Beispiel

## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

ebenfalls als Objektoperation der Klasse *Seminar* zugeordnet und ermittelt über die Assoziationen die notwendigen Daten.

- 2 Vererbung von Operationen** Machen Sie sich in der Analyse keine Gedanken über die Vererbung von Operationen. Tragen Sie alle Operationen so hoch wie möglich in Vererbungshierarchien ein. Die Semantik der Operation entscheidet über ihre Einordnung in die Vererbungsstruktur. Die Operation soll so hoch eingetragen werden, daß deren Beschreibung bei allen tieferen Klassen »paßt«. In der Entwurfs- und Implementierungsphase wird dann neu festgelegt, wann eine Operation überschrieben wird.
- 3 Beschreibung** Eine Beschreibung, was die Operation aus fachlicher Sicht leisten soll, ist nur dann zu erstellen, wenn ihre Funktionsweise anhand des Namens und der Interaktionsdiagramme nicht klar wird. Diese Beschreibung wird im allgemeinen informal sein. Prinzipiell sind jedoch auch alle anderen Beschreibungsformen, z.B. mittels Vor- und Nachbedingungen, möglich.

### Analytische Schritte zum Validieren der Operationen

- 4 Name** Der Name einer Operation soll
- mit einem Verb beginnen,
  - dasselbe aussagen, was die Operation »tut« und
  - im Kontext der Klasse verständlich und eindeutig sein.
- 5 Qualitätskriterien** /Page-Jones 88/ fordert für den strukturierten Entwurf die Einhaltung einer Reihe von Kriterien, die auch bei der objektorientierten Analyse Gültigkeit besitzen. Dazu gehört insbesondere die funktionale Bindung, d.h. jede Operation realisiert eine in sich abgeschlossene Funktion.
- 6 *balancing*** Bei der Erstellung des statischen Modells passiert es leicht, daß unreflektiert Attribute übernommen werden. Um die Ausgewogenheit (*balancing*) von statischem und dynamischen Modell zu sichern, sollen Klassen nur solche Attribute enthalten, die von den Operationen der Klasse oder einer ihrer Unterklassen benötigt werden.
- 7 Fehlerquelle** Prüfen Sie, ob jede Operation die reine Funktionalität beschreibt oder Aspekte der Benutzungsoberfläche enthält.

**Ergebnisse**■ **Klassendiagramm**

In das Klassendiagramm werden die Operationen eingetragen.

■ **Beschreibung der Operationen****Konstruktive Schritte****1 Operationen ins Klassendiagramm eintragen**

- Operationen aus Szenarios und Zustandsautomaten übernehmen.
- Listenoperationen hinzufügen.
- Keine Verwaltungsoperationen eintragen.

**2 Vererbung von Operationen berücksichtigen**

- Operationen so hoch wie möglich in der Hierarchie eintragen.

**3 Beschreibungen erstellen**

- Kann entfallen.
- Im allgemeinen informal.
- Bei Bedarf auch semiformale Spezifikation.

**Analytische Schritte****4 Besitzt die Operation einen geeigneten Namen?**

- Beginnt mit einem Verb.
- Beschreibt, was die Operation »tut«.

**5 Erfüllt jede Operation die geforderten Qualitätskriterien?**

- Angemessener Umfang.
- Funktionale Bindung.

**6 Ist das *balancing* erfüllt?**

- Alle Attribute werden von den Operationen benötigt.

**7 Fehlerquelle**

- Keine Benutzungsoberfläche.

**Tab. 4.10-1:**  
**Checkliste**  
**Operationen**

## 4.11 Formale Inspektion

Auch beim Einsatz von Werkzeugen, was heute für eine objekt-orientierte Entwicklung selbstverständlich sein sollte, müssen bei der Qualitätsprüfung manuelle Verfahren eingesetzt werden. Die hier beschriebene Inspektion ist optimal auf den Analyse- und Entwurfsprozeß der hier beschriebenen Methode abgestimmt. Die Inspektionsmethode wurde von M.E. Fagan bei IBM entwickelt. Er übertrug statistische Qualitätsmethoden, die in der industriellen Hardwareentwicklung benutzt wurden, auf seine Softwareprojekte, die er von 1972 bis 1974 durchführte. 1976 berichtete er über seine Erfahrungen in /Fagan 76/ (siehe auch /Fagan 86/).

Die Inspektion ist definiert als »*a formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards*« /ANSI/IEEE Std. 729-1983/.

Definition

## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

Es gibt in der Literatur zahlreiche zum Teil abweichende Auffassungen über Inspektionen. Die folgenden Ausführungen orientieren sich an der Inspektionsmethode von /Gilb, Graham 93/, die in /Balzert 98/ an die dort beschriebene objektorientierte Methode angepaßt wurde. Eine Inspektion wird bei der hier beschriebenen Methode dann vorgenommen, wenn eine Version 0.x des OOA-Modells fertiggestellt ist und es durch die Inspektion für den Entwurf freigegeben werden soll. Am Anfang einer Inspektion wird davon ausgegangen, daß das Prüfobjekt mit verschiedenen »Defekten« behaftet ist. Auch in der Entwurfs- und Implementierungsphase kann die Inspektion zur Qualitätsprüfung eingesetzt werden.

Inspektion  
beantragen

Eine Inspektion wird durch den Autor des Modells beantragt. Es wird ein Moderator ausgewählt, der für die Organisation und Durchführung der Inspektion verantwortlich ist. Der Moderator darf nicht der Linienvorgesetzte des Mitarbeiters sein, dessen Produkt geprüft wird. Er muß für diese Tätigkeit ausgebildet sein.

Eingangskriterien  
prüfen

Der Moderator muß zuerst prüfen, ob die Eingangskriterien für eine Inspektion erfüllt sind. Stellt er beispielsweise mit einem kurzen Blick auf das Prüfobjekt eine große Anzahl kleinerer Fehler oder gravierende Defekte fest, dann gibt er das Prüfobjekt gleich an den Autor zurück.

Planung

Erfüllt das Prüfobjekt die Eingangskriterien, dann plant der Moderator den Inspektionsprozeß:

- Festlegen und Einladen des Inspektorenteams.
- Festlegen und Zuordnen von Rollen an jeden Inspektor. Jede Rolle ist verknüpft mit der Prüfung spezieller Aspekte.
- Festlegen aller notwendigen Referenzunterlagen (Ursprungsprodukt, Checkliste etc.).
- Aufteilen des Prüfobjekts in kleinere Einheiten, wenn es für eine maximal zweistündige Inspektionssitzung zu umfangreich ist.
- Festlegen von Terminen.

Inspektorenteam

Das Inspektorenteam besteht außer dem Moderator und dem Autor aus ein bis vier Inspektoren. Bei einem kleinen Team führt der Moderator Protokoll. Bei einem großen Team sollte es einen Protokollführer geben.

Beispiel Rollen

Bei der Prüfung von Ergebnissen der Analysephase können folgende Rollen vergeben werden:

- Statisches Modell,
- Dynamisches Modell und
- Prototyp der Benutzungsoberfläche.

Ausschnitts-  
prüfung

Eine Alternative zur vollständigen Prüfung eines Dokuments besteht darin, nur einen Teil zu inspizieren. Eine solche Überprüfung kann dazu benutzt werden, die Fehlerdichte pro Seite der ungeprüften Teile zu schätzen. Die Ausschnittsprüfung basiert auf der Tatsache, daß Fehler, die in einem Teil des Dokuments zu finden sind,

sich in den anderen Teilen wiederholen. Die Ausschnittsprüfung kann dazu benutzt werden, das gesamte Dokument freizugeben oder dem Autor genügend Hinweise zu geben, die ihm erlauben, selbst die Defekte in den ungeprüften Teilen zu finden. Ausschnittsprüfungen sind vor allem im Entwurf sinnvoll.

Jeder Inspektor erhält folgende Unterlagen:

Unterlagen

- Prüfobjekt,
- Ursprungsprodukt, auf dessen Basis das Prüfobjekt erstellt wurde, z.B. Pflichtenheft,
- Checklisten,
- Inspektionsregeln, die angeben, wie die Inspektion abläuft,
- Inspektionsplan (Zeit- und Mitarbeiter-Einsatzplan).

Jedes Mitglied des Inspektorenteams bereitet sich individuell auf die Sitzung vor. Dabei sind zu beachten:

individuelle Vorbereitung

- Die Vorbereitungen müssen bis zur Sitzung abgeschlossen sein.
- Das Prüfobjekt ist auf diejenigen Defekte hin zu untersuchen, die sich aus der zugewiesenen Rolle ergeben.
- Gefundene Defekte sind formlos zu notieren.

Bei der Prüfung sollen leichte und schwere Defekte unterschieden werden. Ein schwerer Defekt verursacht mit großer Wahrscheinlichkeit große Folgekosten, wenn er nicht sofort beseitigt wird. Das Identifizieren und Beheben schwerer Defekte ist ein Hauptgrund für Inspektionen in der frühen Entwicklungsphase. Es ist wichtig, sich auf gravierende Defekte zu konzentrieren. Sonst besteht die Gefahr, daß zuviel Zeit für ökonomisch unwichtige Defekte verbraucht wird. In der Praxis fällt es oft schwer, sich auf die wichtigen Defekte zu konzentrieren. Daher sollten Sie unbedingt folgende Maßnahmen beachten:

leichte & schwere Defekte

- Jeder Inspektor soll genau festgelegte Aspekte prüfen.
- Alle Inspektoren sind darauf hinzuweisen, daß primär schwere Defekte zu suchen sind.
- In den Checklisten werden alle Defekte durch »M« (*major*, schwer) und »m« (*minor*, leicht) gekennzeichnet.
- In der Inspektionssitzung sind zuerst alle schweren Defekte anzugeben. Leichte Defekte sollten bei zeitlichen Engpässen schriftlich an den Autor weitergegeben werden.

Auf die individuelle Vorbereitung folgt eine gemeinsame Inspektionssitzung (*logging meeting*). Mit ihr werden drei Ziele verfolgt:

Inspektionssitzung

- Protokollieren aller Defekte, die während der Vorbereitung identifiziert werden.
- Identifizieren und Protokollieren zusätzlicher Defekte.
- Protokollieren von Verbesserungsvorschlägen und Fragen an den Autor.

Eine Inspektion ist vergleichbar mit einer *Brainstorming*-Sitzung. Anstelle der Ideen geht es jedoch um Defekte. Das Ziel ist es, Defekte zu identifizieren, nicht sie zu diskutieren.



## LE 8 4 Checklisten zum Erstellen eines OOA-Modells

Die Inspektionssitzung soll pünktlich beginnen und nicht länger als zwei Stunden dauern, da die Teilnehmer sonst ermüden. Die Sitzung beginnt damit, daß von jedem Inspektor folgende Daten anonym protokolliert werden: benötigte Zeit für die Vorbereitung, Anzahl schwerer Defekte, Anzahl geprüfter Seiten. Es hat sich bewährt, das Protokoll während der Sitzung direkt mit einem Computer zu erfassen. Da der Autor die Defekte anschließend beheben muß, ist es wichtig, daß er alle protokollierten Defekte versteht.

Protokoll Das wichtigste Ergebnis der Inspektionssitzung ist das Inspektionsprotokoll. Es soll folgende Informationen enthalten:

- Inspektionsdatum,
- Name des Moderators,
- Prüfobjekt,
- Referenzunterlagen,
- Defekte mit Angaben zu
- Kurzbeschreibung,
- Ort,
- Bezug zu Regeln oder Checklisten,
- leichter oder schwerer Defekt,
- in der Sitzung oder bei der Vorbereitung identifiziert.

Das Protokoll enthält jedoch keine Informationen darüber, welches Mitglied des Inspektorenteams den Defekt gemeldet hat.

Prozeß-Brainstorming Nach der Inspektionssitzung kann optional noch eine Prozeß-*Brainstorming*-Sitzung durchgeführt werden, um Defektursachen zu analysieren und den Erstellungsprozeß so bald wie möglich zu verbessern. Eine wichtige Basis für Prozeßverbesserungen stellen auch Datensammlungen über durchgeführte Inspektionen dar, z.B. Anzahl schwerer Defekte, benötigte Zeiten.

Überarbeitung Anhand des Protokolls überarbeitet der Autor das Prüfobjekt und vermerkt die durchgeführten Aktionen im Protokoll. Er kann die Defekt-Klassifikation (leicht, schwer) selbständig ändern.

Nachprüfung Hat der Autor seine Überarbeitung abgeschlossen, dann prüft der Moderator die Sorgfalt und Vollständigkeit der Überarbeitung, aber nicht die Korrektheit. Die erfolgreiche Nachprüfung (*follow up*) ist die Voraussetzung für die formale Freigabe des Prüfobjekts.

**Dynamisches Modell** Das dynamische Modell ist der Teil des OOA-Modells, welches das Verhalten des zu entwickelnden Systems beschreibt. Es realisiert außer den Basiskonzepten (Objekt, Klasse, Operation) die dynamischen Konzepte (Geschäftsprozeß, Szenario, Botschaft, Zustandsautomat).

**Formale Inspektion** Die formale Inspektion ist ein formales Verfahren zur manuellen Prüfung der Dokumentation.

**Operation (*operation*)** Eine Operation ist eine Funktion, die auf die internen Daten (Attributwerte) eines Objekts Zugriff hat. Sie kann Botschaften an andere Objekte senden. Auf alle Objekte einer Klasse sind dieselben Operationen anwendbar. Abstrakte Operationen besitzen nur einen Operationskopf. Externe Operationen werden vom späteren Bediener des Systems aktiviert. Interne Operation werden dagegen im-



mer von anderen Operationen aufgerufen.

**Sequenzdiagramm (sequence diagram)** Ein Sequenzdiagramm besitzt zwei Dimensionen. Die Vertikale repräsentiert die Zeit und auf der Horizontalen werden die Objekte angetragen. In das Diagramm werden die Botschaften eingetragen, die zum Aktivieren der Operationen dienen.

**Szenario (scenario)** Ein Szenario ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind. Diese Schritte sollen

das Hauptziel des Akteurs realisieren und ein entsprechendes Ergebnis liefern. Ein Geschäftsprozeß wird durch eine Kollektion von Szenarios dokumentiert.

**Zustandsautomat (finite state machine)** Ein Zustandsautomat besteht aus Zuständen und Transitionen. Er hat einen Anfangszustand und kann einen Endzustand besitzen.

**Zustandsdiagramm (statechart diagram)** Das Zustandsdiagramm ist eine grafische Repräsentation des Zustandsautomaten.



Die Checkliste **Szenario** zeigt, wie aus einem Geschäftsprozeß Szenarios abgeleitet und mittels Sequenzdiagrammen dokumentiert werden. Die Checkliste **Zustandsautomat** beschreibt die systematische Modellierung eines Objekt-Lebenszyklus. Die **Operationen** im Klassendiagramm stellen die Verbindung zwischen dem statischen und dem dynamischen Modell her. Die **formale Inspektion** ist ein Review-Verfahren, das auf den vorgestellten Checklisten aufbaut.



**1 Lernziel:** Ein Szenario als Sequenzdiagramm beschreiben können. Für folgende Beschreibung sind die Bedingungen und das Ergebnis zu identifizieren. Erstellen Sie dann ein Sequenzdiagramm. Eine Palette ist aus einem Lager in ein anderes umzulagern. Jedes Lager besteht aus mehreren Lagerplätzen. Prüfen Sie zuerst, ob die Palette eine Klimatisierung benötigt und prüfen Sie dann, ob im gewünschten Lager ein geeigneter Platz frei ist. Anschließend wird die Palette aus der Quelle entfernt und am Ziel eingefügt.

Aufgabe  
10 Minuten

**2 Lernziel:** Aus einem Geschäftsprozeß ein Klassendiagramm und Szenarios erstellen können.

Aufgabe  
20 Minuten

Leiten Sie aus folgendem Geschäftsprozeß ein Klassendiagramm ab und spezifizieren Sie für folgende Szenarios Sequenzdiagramme.

- Existierendem Kunden wird der Kredit gewährt.
- Existierendem Kunden wird der Kredit nur mit einem Bürgen gewährt.

*Geschäftsprozeß:* bearbeite Kreditantrag

*Ziel:* Dem Kunden den gewünschten Kredit gewähren

*Kategorie:* -

*Vorbereitung:* -

*Nachbedingung Erfolg:* Kunde erhält den Kredit

*Nachbedingung Fehlschlag:* Kreditantrag wird abgelehnt

*Akteure:* Kreditbearbeiter

*Auslösendes Ereignis:* Kreditantrag liegt vor

## LE 8 Aufgaben

*Beschreibung:*

- 1 prüfen, ob Kunde der Bank bekannt ist
- 2 prüfen, ob entsprechende Sicherheiten vorhanden sind
- 3 prüfen, ob Kunde feste Beschäftigung hat
- 4 Kreditwürdigkeit prüfen
- 5 Kredit gewähren

*Erweiterungen:*

1a neuer Kunde

*Alternativen:*

2a prüfen, ob ein Bürge vorhanden ist

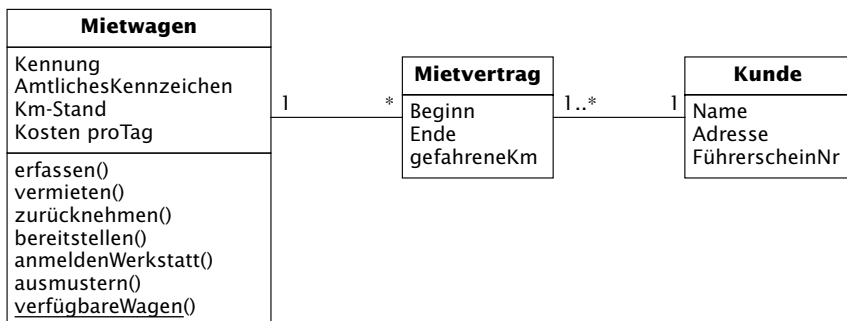
5a niedrigeren Kredit gewähren

Aufgabe 3 **Lernziel:** Für eine Klasse den Zustandsautomaten (Lebenszyklus) erstellen können.  
15 Minuten

Spezifizieren Sie für die Klasse Mietwagen (Abb. LE8-A3) den Lebenszyklus. Die Wirkung der Operationen ist wie folgt definiert:

- vermieten()  
Ein neues Objekt von Mietvertrag erzeugen und entweder vorhandenem Kunden zuordnen oder neuen Kunden erfassen.
- zurücknehmen()  
Gibt der Kunde den Mietwagen vertragsgemäß zurück, so wird das Auto gewaschen und überprüft.
- bereitstellen()  
Ergibt sich bei der Prüfung die einwandfreie Funktion, so wird der Wagen wieder in den Fuhrpark überstellt.
- anmeldenWerkstatt()  
Ein defekter Mietwagen wird zur Reparatur angemeldet. Sobald ein Mietwagen repariert ist, wird er wieder bereitgestellt.
- ausmustern()  
Ergibt sich bei der Prüfung, daß der Kilometerstand des Mietwagens größer als 80000 km ist, dann wird er aus dem Fuhrpark des Mietwagen-Verleihs entfernt und zum Verkauf bereitgestellt. Ist der Wagen defekt, dann wird er vorher noch repariert, um einen höheren Verkaufspreis zu erzielen.

Abb. LE8-A3:  
Klassendiagramm des Mietwagenverleihs



**4 Lernziel: OOA-Modelle prüfen können.**

Muß-Aufgabe  
10 Minuten

Prüfen Sie, wie die Modelle **a** und **b** der Abb. LE8-A4 verbessert werden können. Mit welcher Checkliste und welchem Prüfpunkt stellen Sie die Fehler fest?

Prüfen Sie, ob die beiden Modelle in **c** der Abb. LE8-A4 gleichwertig sind. Erstellen Sie dazu Objektdiagramme.

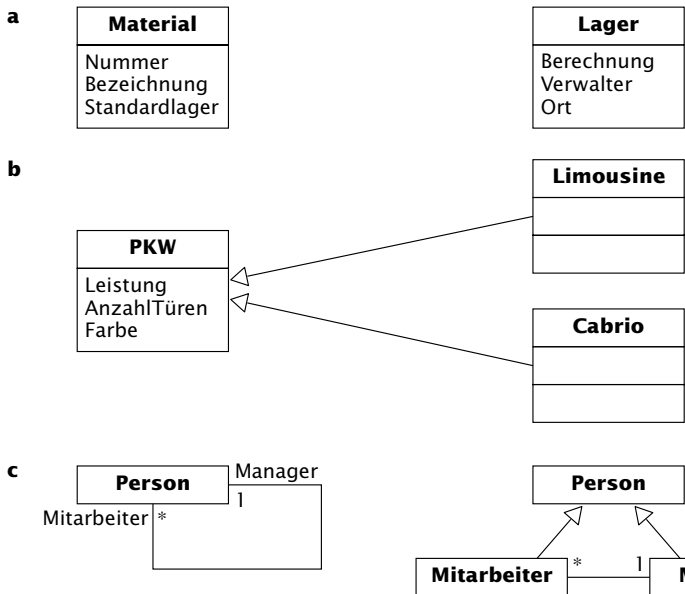


Abb. LE8-A4: Zu prüfende Klassendiagramme

**5 Lernziel: Wissen und Verständnis über die Inspektion prüfen.**

Aufgabe  
10-15 Minuten

- a** Wie viele Mitglieder soll ein Inspektionsteam besitzen und wie werden die Aufgaben verteilt?
- b** Ein umfangreiches OOA-Modell muß möglichst schnell inspiziert werden. Was können sie tun, um dieses Ziel zu erreichen?
- c** Auf Ihren Vorschlag, die Inspektion einzuführen, erhalten Sie als ablehnendes Argument, daß dies zu viel Zeit kosten würde. Was können Sie dem entgegensetzen?
- d** Auf Ihren Vorschlag, die Inspektion einzuführen, werden Sie gefragt »Wieso brauchen wir eine Inspektion? Wir setzen doch schon ein Werkzeug ein«. Was haben Sie dem entgegensetzen?

## 5 Gestaltung von Benutzungsoberflächen (Teil 1)



- Wissen, was ein GUI-System ist.
- Wissen, was ein Gestaltungsregelwerk (*style guide*) ist.
- Wissen, was Primär- und Sekundärdialoge sind.
- Wissen, was modale und nicht-modale Dialoge sind.
- Wissen, was SDI und MDI bedeutet.
- Wissen, wie sich funktions- und objektorientierte Bedienung unterscheiden.
- Wissen, wie direkte Manipulation funktioniert.
- Wissen, welche Anwendungstypen es gibt.
- Erklären können, welche Fenstertypen es gibt.
- Erklären können, welche Menüs es gibt.

wissen

verstehen



- Diese Lehreinheit kann prinzipiell unabhängig von den bisherigen Lehreinheiten gelesen werden.
- Empfehlenswert zur Vorbereitung sind die Kapitel 1 und 2.

- i 5.1 Einführung in die Software-Ergonomie 194
- 5.2 Dialoggestaltung 195
- 5.3 Fenster 199
- 5.4 Menüs 202

## 5.1 Einführung in die Software-Ergonomie

- Definition** Wenn das OOA-Modell vorliegt, erstellen wir einen **Prototyp** der Benutzungsoberfläche, d.h. wir gestalten die Oberfläche des zukünftigen Systems. Für diese Aufgabe werden Grundkenntnisse in der Software-Ergonomie benötigt.
- Software-Ergonomie** Die **Software-Ergonomie** befaßt sich mit der menschengerechten Gestaltung von Softwaresystemen. Sie verfolgt das Ziel, die Software an die Eigenschaften und Bedürfnisse der Benutzer anzupassen /Balzert 96/.
- Das Ziel dieses Kapitels ist es, Ihnen die grundlegenden Kenntnisse für die Gestaltung eines Oberflächenprototypen und den Entwurf der Benutzungsoberfläche – unter Windows – zu vermitteln. Es soll weder eine Software-Ergonomie-Veranstaltung noch ein entsprechendes Fachbuch ersetzen. Einen guten Überblick zu diesem Thema bietet /Balzert 96/, das als Basis für dieses Kapitel diene.
- Prototyp der Benutzungsoberfläche** Ein Prototyp der Benutzungsoberfläche realisiert Fenster, Menüs und die globale Dialogführung. Beispielsweise kann der Bediener vom Menü die entsprechenden Fenster öffnen, kann zwischen Fenstern wechseln und Fenster schließen. Der Prototyp enthält jedoch keine Daten und keine Dialogführung innerhalb von Datenfeldern (z.B. *scrolling* in einer Liste). Außerdem sind natürlich keine Funktionen realisiert.
- GUI-System** Ein **GUI** (*graphical user interface*) ist eine grafische Benutzungsoberfläche. Sie besteht aus einer Dialogkomponente (Bedienungsabläufe) und einer E/A-Komponente (Gestaltung der Informationsdarstellung). Das **GUI-System** ist das Software-System, das diese grafische Benutzungsoberfläche verwaltet und die Kommunikation mit den Anwendungen abwickelt. Ein GUI-System wird vereinfachend auch Fenstersystem genannt. Beispiele für GUI-Systeme sind Windows (Microsoft), Presentation Manager (IBM), Motif (Open Software Foundation), OpenLook (SUN) und Nextstep (Next). Für die Erstellung des Prototyps wird idealerweise das gleiche GUI-System verwendet, das im Entwurf für die Realisierung der Benutzungsoberfläche benutzt wird. Auf diese Weise entsteht kein »Wegwerf«-Prototyp, sondern der Prototyp kann evolutionär weiterentwickelt werden.
- style guide** Ein **Gestaltungsregelwerk** (*style guide*) schreibt vor, wie die Benutzungsoberfläche von Anwendungen gestaltet wird. Beispielsweise bestimmt es entscheidend das Aussehen von Fenstern, Menüs und Interaktionselementen. Es soll sicherstellen, daß das **look and feel** über verschiedene Anwendungen hinweg gleich bleibt, damit Benutzungsoberflächen weitgehend einheitlich gestaltet werden. Mit dem prägnanten Begriff *look and feel* bezeichnet man das visuelle Erscheinungsbild und die Bedienungseigenschaften grafischer Benutzungsoberflächen. *Style guides* können sowohl Regelwerke

des GUI-Herstellers oder auch unternehmenseigene Gestaltungsregelwerke sein. Ich beschränke mich hier bei der Beschreibung der Benutzungsoberfläche auf den *style guide* von Windows /MS 95/.

## 5.2 Dialoggestaltung

Ein **Dialog** ist eine Interaktion zwischen einem Benutzer und einem Dialogsystem, um ein bestimmtes Ziel zu erreichen. Ein Benutzer ist ein Mensch, der mit dem Dialogsystem arbeitet /ISO 9241-10 : 1996/.

Aus Sicht der Benutzer lassen sich Primär- und Sekundärdialoge unterscheiden. Die Arbeitsschritte, die zur direkten Aufgabenerfüllung dienen, bilden den **Primärdialog**. Er ist erst dann beendet, wenn die zu bearbeitende Aufgabe fertiggestellt ist. Benötigt der Benutzer situationsabhängig zusätzliche Informationen, dann werden diese Hilfsdienste durch **Sekundärdialoge** erledigt. Ist der Sekundärdialog beendet, dann wird der Primärdialog fortgesetzt.

Wenn Sie in *Winword* ein Textdokument bearbeiten, so führen Sie einen Primärdialog aus. Wollen Sie das Dokument drucken, so starten Sie den Sekundärdialog *Drucken* und wählen dort den gewünschten Drucker aus und führen die gewünschten Einstellungen durch. Erst wenn Sie diesen Dialog beendet haben, können Sie mit der Bearbeitung des Dokuments fortfahren.

Aus technischer Sicht lassen sich folgende **Dialogmodi** unterscheiden. Ein **modaler Dialog** (*modal dialog*) muß beendet sein, bevor eine andere Aufgabe der Anwendung durchgeführt werden kann, d.h. bevor ein anderes Fenster aktiviert werden kann. Ein **nicht-modaler Dialog** (*modeless dialog*) ermöglicht es dem Benutzer, den aktuellen Dialog zu unterbrechen, d.h. andere Aktionen durchzuführen, während das ursprüngliche Fenster geöffnet bleibt. Bei dieser Dialogform wird also kein bestimmter Arbeitsmodus (*mode*) vorgeschrieben. Das Ziel der Dialoggestaltung sollte es sein, möglichst viele nicht-modale Dialoge zu verwenden, da dadurch die Handlungsflexibilität optimiert wird. In bestimmten Situationen muß die Flexibilität jedoch eingeschränkt werden. Tritt beispielsweise ein Fehler auf, dann kann erst nach dessen Behebung weitergearbeitet werden.

Wenn Sie in *Winword* während der Bearbeitung eines Dokuments den Dialog *Bearbeiten/Ersetzen* starten, so können Sie, ohne diesen Dialog zu beenden, mit der Texterstellung fortfahren (nicht-modaler Dialog). Dagegen handelt es sich beim *Drucken* um einen modalen Dialog. Erst wenn dieser Dialog beendet ist, kann eine andere Bearbeitung durchgeführt werden.

## LE 9 5 Benutzungsoberflächen

SDI und MDI Eine **SDI-Anwendung** (*single document interface*) ermöglicht es dem Benutzer zu einem Zeitpunkt genau ein Dokument zu öffnen und zu bearbeiten. Bei einer **MDI-Anwendung** (*multiple document interface*) können zu einem Zeitpunkt beliebig viele Dokumente geöffnet sein. Der Benutzer wählt bei mehreren gleichzeitig geöffneten Dokumenten das jeweils aktive durch Anklicken mit der Maus oder über das Menü aus.

Beispiel *WordPad* ist eine SDI-Anwendung. Bevor Sie ein neues Dokument öffnen können, müssen Sie das aktuelle Dokument zuerst schließen. *Winword* ist eine MDI-Anwendung. Sie können beliebig viele Dokumente parallel öffnen und ein neues Dokument bearbeiten, ohne das vorherige zu schließen. Zwischen den Dokumenten können Sie beliebig wechseln.

### **Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten (Teil 10: Grundsätze der Dialoggestaltung: ISO 9241-10 : 1996)**

Europäische Norm  
ISO 9241-10 Für die Dialoggestaltung gibt es eine Reihe von Richtlinien. Die folgenden sieben Grundsätze sind für die Gestaltung und Bewertung eines Dialogs als wichtig anerkannt worden /ISO 9241-10 : 1996/.

#### ■ **Aufgabenangemessenheit**

»Ein Dialog ist aufgabenangemessen, wenn er den Benutzer unterstützt, seine Arbeitsaufgabe effektiv und effizient zu erledigen.«

Die Positionsmarke wird automatisch auf das erste Eingabefeld positioniert, das für die Arbeitsaufgabe relevant ist.

#### ■ **Selbstbeschreibungsfähigkeit**

»Ein Dialog ist selbstbeschreibungsfähig, wenn jeder einzelne Dialogschritt durch Rückmeldung des Dialogsystems unmittelbar verständlich ist oder dem Benutzer auf Anfrage erklärt wird.«

Kann das Löschen von Daten nicht rückgängig gemacht werden, verlangt das Dialogsystem eine Bestätigung.

#### ■ **Steuerbarkeit**

»Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.«

Das Dialogsystem bewegt die Positionsmarke auf das nächste Eingabefeld; der Benutzer kann aber statt dessen ein anderes Feld auswählen.

#### ■ **Erwartungskonformität**

»Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht, z.B. seinen Kenntnis-



sen aus dem Arbeitsgebiet, seiner Ausbildung und seiner Erfahrung sowie den allgemein anerkannten Konventionen.«

Meldungen des Dialogsystems werden stets an derselben Stelle ausgegeben. Der Dialog wird stets durch das Drücken derselben Taste beendet.

#### ■ Fehlertoleranz

»Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.«

Wird vom Dialogsystem ein Fehler festgestellt, der sich eindeutig auf ein bestimmtes Eingabefeld bezieht, dann wird dieses Feld markiert und die Positionsmarke automatisch auf den Anfang des Feldes gesetzt.

#### ■ Individualisierbarkeit

»Ein Dialog ist individualisierbar, wenn das Dialogsystem Anpassungen an die Erfordernisse der Arbeitsaufgabe sowie an die individuellen Fähigkeiten und Vorlieben des Benutzers zulässt.«

Das Dialogsystem erlaubt es dem Benutzer, bei Informationsausgaben die Geschwindigkeit des *scrolling* zu steuern.

#### ■ Lernförderlichkeit

»Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen des Dialogsystems unterstützt und anleitet.«

*Learning by doing* wird dadurch unterstützt, daß der Benutzer ermutigt wird, zu experimentieren, ohne daß die Gefahr besteht, potentiell katastrophale Ergebnisse herbeizuführen.

### Prinzipielle Alternativen der Dialoggestaltung

Unabhängig vom verwendeten GUI-System sollten Sie sich zuerst die prinzipiellen Alternativen der Dialoggestaltung verdeutlichen und dann überlegen, welche Alternative für die jeweilige Anwendung aus Sicht des Benutzers am besten geeignet ist. Prinzipiell läßt sich jede Anwendung in Objekttypen und Funktionen gliedern. Aufgrund dieser Zweiteilung lassen sich folgende Bedienungsarten unterscheiden.

**1** objektorientierte Bedienung mit direkter Manipulation

**2** objektorientierte Bedienung mit Menüs und Fenstern

**3** funktionsorientierte Bedienung mit Menüs und Fenstern

Viele Anwendungen realisieren jedoch keine der drei Bedienungsarten in »Reinform«, sondern eine Kombination zweier oder dreier Arten.

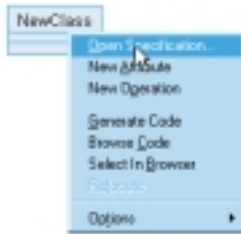
Bedienungsarten

## LE 9 5 Benutzungsoberflächen

objektorientiert Bei einer **objektorientierten Bedienung** wählt der Benutzer zuerst das Objekt, das er bearbeiten will und anschließend die Funktion, die auf dem Objekt ausgeführt werden soll, wobei die Eigenschaften des Objekts die zulässigen Operationen bestimmen.

Beispiel Abb. 5.2-1 zeigt die objektorientierte Bedienung des CASE-Werkzeugs *Rational Rose* mit der Maus. Das Objekt *NewClass* wird selektiert. Anschließend können mit der rechten Maustaste alle zulässigen Funktionen bzw. Operationen angezeigt werden. Der Benutzer wählt beispielsweise *Open Specification*.

Abb. 5.2-1:  
Beispiel für  
objektorientierte  
Bedienung



funktionsorientiert Bei der **funktionsorientierten Bedienung** wählt der Benutzer zunächst eine Funktion und bestimmt anschließend für welches Objekt diese Funktion ausgeführt werden soll.

Beispiel Der Benutzer wählt in der Anwendung *Rational Rose* zunächst im *File*-Menü die Funktion *Open...* und dann im entsprechenden Fenster das gewünschte Modell aus (Abb. 5.2-2).

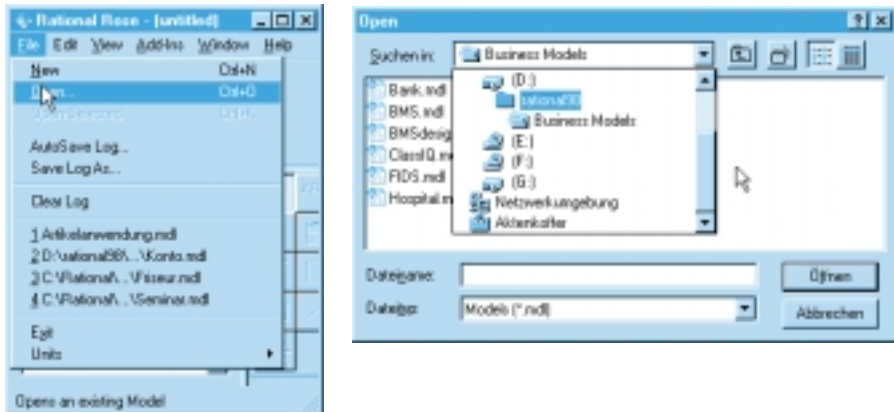

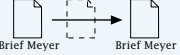
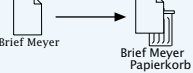


Abb. 5.2-2: Empirische Untersuchungen haben gezeigt, daß beide Bedienungsarten gleichwertig sind. Eine objektorientierte Entwicklung muß nicht zwangsläufig zu einer reinen objektorientierten Bedienung führen. Umgekehrt impliziert eine objektorientierte Bedienung nicht automatisch eine objektorientierte Entwicklung.

Die objektorientierte Bedienung kann sowohl mittels Menüs und Fenstern als auch mit der direkten Manipulation erfolgen.

direkte Manipulation Bei der **direkten Manipulation** werden vom Benutzer – in Analogie zur Schreibtischarbeit ohne Computer – Objekte (z.B. Dokumente) selektiert und bearbeitet. Die Bedienungstechnik »Selektie-

Funktion	Bedienung	Reaktion	Beispiel
Selektion eines Objekts	Einfacher Mausklick auf das Objekt	z.B. invertierte Darstellung	
Bewegen eines Objekts	Objekt selektieren, mit gedrückter Maustaste zum Zielort bewegen, Maustaste loslassen	Objekt folgt der Mausbewegung	
Löschen eines Objekts	Objekt selektieren und auf Papierkorb bewegen	Objekt verschwindet, Papierkorb-Piktogramm ändert sich	
Aktivieren eines Objekts	Doppelclick auf Objekt	Anwendung wird gestartet	

ren, Ziehen und Loslassen« (*pick, drag & drop*) ist ein Beispiel für direkte Manipulation. Ein Ziel der direkten Manipulation besteht darin, Funktionen über mehrere Anwendungen konsistent zu verwenden. Man spricht von generischen Funktionen. Eine **generische Funktion** besitzt in verschiedenen Anwendungen die gleiche Bezeichnung, die gleiche Semantik und die gleiche Bedienung (Abb. 5.2-3).

Abb. 5.2-3: Generische Funktionen der direkten Manipulation

Windows erlaubt sowohl die direkte Manipulation als auch die Bedienung mittels Menüs. Eine Anwendung kann sowohl über das Start-Menü als auch mit einem Doppelclick auf das Objekt gestartet werden.

Beispiel

Hinweis: Beim Bewegen eines Objekts (*pick, drag & drop*) verstößt Windows gegen die Erwartungskonformität, da je nach aktuellem Kontext unterschiedliche Wirkungen auftreten. Auf unterschiedlichen Laufwerken wird kopiert, auf dem gleichen Laufwerk verschoben und bei *exe*-Dateien wird eine Verknüpfung erstellt.

## 5.3 Fenster

Zentrales Element der Dialoggestaltung ist das Fenster (*window*). Abb. 5.3-1 zeigt den Aufbau und die Begriffe eines Fenster bei Windows.

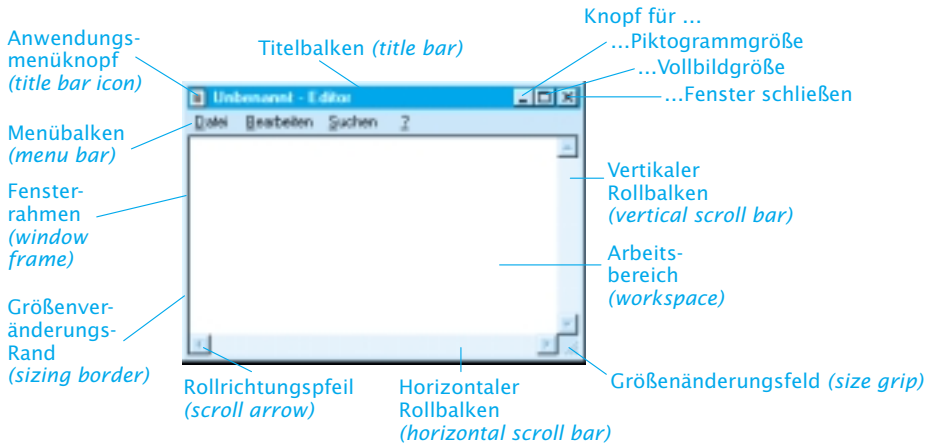
Der *style guide* von Windows /MS 95/ unterscheidet folgende **Fenstertypen**:

- Primärfenster (*primary window*), in denen die Hauptaktivitäten des Benutzers (Primärdialoge) stattfinden, und
- Sekundärfenster (*secondary window*), die der Eingabe von Optionen und der Durchführung sekundärer Aktivitäten dienen (Sekundärdialoge).

Das wichtigste Primärfenster ist das **Anwendungsfenster** (*application window*). Es erscheint nach dem Aufruf der Anwendung. Aus diesem Fenster heraus lassen sich alle weiteren Fenster der An-

Anwendungsfenster

## LE 9 5 Benutzungsoberflächen



**Abb. 5.3-1:** Typischer Fenster-aufbau bei Windows

wendung öffnen. Ein Anwendungsfenster enthält mindestens den Titelbalken, den Menübalken und den Arbeitsbereich. Wird das Anwendungsfenster geschlossen, dann werden alle zur Zeit geöffneten Fenster dieser Anwendung ebenfalls automatisch geschlossen. Bei einer SDI-Anwendung erfolgt die Interaktion mit dem Benutzer schwerpunktmäßig im Arbeitsbereich des Fensters. Bei einer MDI-Anwendung ist der Arbeitsbereich leer.

Bei einer MDI-Anwendung können vom Anwendungsfenster aus **Unterfenster** (*child windows*) geöffnet werden. Es ist die Aufgabe eines Unterfensters, den Primärdialog des Benutzers zu unterstützen. Das äußere Erscheinungsbild eines Unterfensters kann mit dem Anwendungsfenster identisch sein. Unterfenster sind verschiebbar, in der Größe änderbar und stapelbar. Sie können – bei typischen Windows-Anwendungen – nicht aus dem Anwendungsfenster herausgeschoben werden, d.h. der hinausragende Teil wird abgeschnitten.

Ein Unterfenster befindet sich im Arbeitsbereich des Anwendungsfensters und ist gleichzeitig durch diesen begrenzt. Auch wenn ein Unterfenster als Piktogramm dargestellt wird, liegt es im Arbeitsbereich des Anwendungsfensters. Der Benutzer wählt bei gleichzeitig geöffneten Unterfenstern das jeweils aktive durch Anklicken mit der Maus oder über ein Menü aus. Unterfenster können überlappend (*cascaded*) oder nebeneinander (*tiled*) dargestellt werden. Ein aktives Fenster liegt immer oben auf dem Fensterstapel. Wird für ein Fenster die maximale Größe gewählt, dann wird der Arbeitsbereich des Anwendungsfensters vollständig genutzt. Unterfenster einer MDI-Anwendung sind sinnvollerweise nicht-modal. Falls zwischen den Fenstern Abhängigkeiten bestehen, dann muß gegebenenfalls davon abgewichen werden.

**Beispiel 1** Nach dem Start von *Winword* erhalten Sie das Anwendungsfenster, aus dem automatisch ein Unterfenster *Dokument1* geöffnet wird.

Sie können beliebig viele Dokumente in Unterfenstern öffnen. Ihr Primärdialog ist die Erstellung eines Dokuments. Zwischen den Unterfenstern können Sie beliebig wechseln.

Beim CASE-Werkzeug *Rational Rose* können Sie Unterfenster für die Geschäftsprozeß-Sicht (*Use Case View*), die logische Sicht (*Logical View*) und die Komponenten-Sicht (*Component View*) öffnen und für jede Sicht mehrere Objekte – d.h. Diagramme – anlegen. Beispiel 2

Windows kennt mehrere Arten von Sekundärfenstern. Dazu gehören das Dialogfenster (*dialog box*) und das Mitteilungsfenster (*message box*).

**Dialogfenster** werden für Sekundärdialoge benötigt. Sie sind daher häufig als modale Dialoge realisiert, können aber auch nicht-modal sein. Ein Sekundärdialog beschränkt sich auf die Dateneingabe über Interaktionselemente im Arbeitsbereich. Dialogfenster

Dialogfenster sind nicht in der Größe veränderbar. Sie können wahlweise verschiebbar sein oder nicht, wobei ein Verschieben nur bei modalen Dialogfenstern notwendig ist. Dialogfenster können über den Rahmen des Anwendungsfensters hinausgeschoben werden. Ein Dialogfenster sollte möglichst wenig Fläche des darunterliegenden Fensters verdecken. Es sollte daher nur die wichtigsten Elemente enthalten und dem Benutzer die Möglichkeit geben, bestimmte Informationen zu expandieren. Windows bietet dazu den *unfolding dialog*. Die Expansion erfolgt durch Betätigung einer entsprechenden Schaltfläche.

In *Winword* öffnen die Menüoptionen *Datei/Speichern unter* und *Datei/Drucken* typische Dialogfenster, die hier für Sekundärdialoge verwendet werden. Beispiel

Beachten Sie, daß Primärdialoge grundsätzlich auch mittels Dialogfenstern realisiert werden. Diese Realisierung ist immer möglich, wenn die speziellen Eigenschaften eines Unterfensters nicht benötigt werden. Um die Steuerbarkeit des Dialogs möglichst wenig einzuengen, sollten diese Dialogfenster nicht-modal sein.

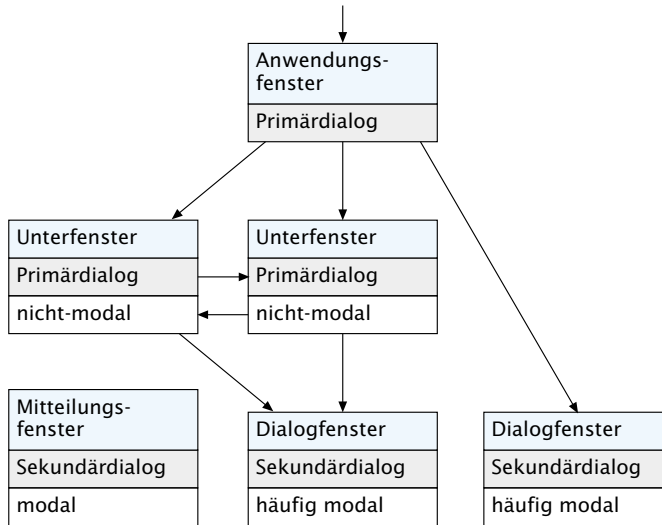
Ein **Mitteilungsfenster** ist ein spezialisiertes Dialogfenster. Der Benutzer kann mit einer Aktion auf die Mitteilung reagieren. Das Fenster enthält keine Interaktionselemente zur Datenselektion oder -manipulation. Mitteilungsfenster sind als modale Dialoge realisiert. Der Benutzer kann erst fortfahren, wenn er auf die Mitteilung reagiert hat. Mitteilungsfenster

Bei Auswahl eines nicht verfügbaren Druckers erscheint eine entsprechende Meldung, die vom Benutzer bestätigt werden muß. Beispiel

Abb. 5.3-2 zeigt den Zusammenhang zwischen den verschiedenen Fenstertypen.

## LE 9 5 Benutzungsoberflächen

Abb. 5.3-2:  
Fensterarten und  
Dialogarten



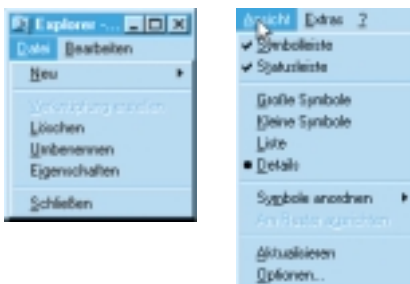
→ Öffnen des Fensters durch Benutzer möglich

## 5.4 Menüs

Menü Ein **Menü** besteht aus einer überschaubaren und meist vordefinierten Menge von Menüoptionen, aus der ein Benutzer eine oder mehrere auswählen kann. Ein **Aktionsmenü** kann Anwendungsfunktionen auslösen oder in andere Menüs verzweigen. Im zweiten Fall sprechen wir von einem Kaskadenmenü. In einem Aktionsmenü kann nur eine Option gewählt werden. Ein **Eigenschaftsmenü** kann Parameter einstellen, die das Verhalten der Anwendung bestimmen. In diesem Menü können oft mehrere Optionen selektiert werden. Auch hier ist ein Kaskadenmenü möglich.

Beispiel Die Abb. 5.4-1 zeigt das Aktionsmenü *Datei*. Die Menüoption *Neu* enthält ein Kaskadenmenü, das durch ein Dreieck markiert wird. Unter *Ansicht* wird ein Eigenschaftsmenü angeboten, in dem mehrere Parameter eingestellt werden können. Verschiedene Parameter-Gruppen sind durch Linien getrennt. Auch hier ist ein Kaskadenmenü enthalten.

Abb. 5.4-1:  
Aktions- und  
Eigenschaftsmenü



Es lassen sich zwei Menüarten unterscheiden:

- Menübalken mit *drop-down*-Menüs und
- *pop-up*-Menüs.

Der Menübalken enthält alle Menütitel. Er ist ständig sichtbar und belegt damit auch ständig Platz. Das Anwendungsfenster besitzt stets einen Menübalken. Dialogfenster und Mitteilungsfenster haben keinen Menübalken. Das Unterfenster einer MDI-Anwendung kann einen Menübalken besitzen, der sich jedoch nicht im Unterfenster befindet, sondern dynamisch den Menübalken des Anwendungsfensters überlagert. Gleichnamige Operationen können bei einer MDI-Anwendung unterschiedliche Wirkungen besitzen, die vom jeweils aktiven Unterfenster abhängen. Besitzen Unterfenster keine eigenen Menübalken, dann wirken die Menüoptionen des Anwendungsfensters auf die gesamte Anwendung.

Menübalken

Das *drop-down*-Menü erscheint nach dem Selektieren des zugehörigen Menütitels im Menübalken. Die Menüoptionen besitzen einen globalen Geltungsbereich. Im aktuellen Kontext nicht selektierbare Menüoptionen sind grau dargestellt. Der Mauszeiger muß vor der Auswahl im *drop-down*-Menü immer zum Menübalken bewegt werden. Beispiele für *drop-down*-Menüs finden Sie in Abb. 5.4-1.

*drop-down*-Menü

Das ***pop-up*-Menü** erscheint an der aktuellen Position des Mauszeigers. Dieses Menü bezieht sich immer auf das Objekt oder die Objektgruppe, für die es aktiviert wurde. Das *pop-up*-Menü ist unsichtbar, wenn es nicht geöffnet ist. Seine Menüoptionen besitzen einen lokalen Geltungsbereich. Es werden im allgemeinen nur diejenigen Menüoptionen angezeigt, die auf das selektierte Objekt angewendet werden können. Der Mauszeiger bleibt hier immer im Arbeitsbereich. Abb. 5.2-1 zeigt ein Beispiel für ein *pop-up*-Menü.

*pop-up*-Menü

Für beide Menüarten gilt:

- Das Menü wird angezeigt, bis eine Menüoption durch Anklicken selektiert wird, ein Klicken außerhalb des Menüs erfolgt oder die *Esc*-Taste gedrückt wird.
- Menüoptionen können dynamisch von der Anwendung geändert werden.
- Durch Kaskadenmenüs lassen sich eine oder mehrere Hierarchiestufen hinzufügen.

**Windows style  
guide für drop-  
down-Menüs**

Die Menübalken sind immer mit *drop-down*-Menüs belegt. Das Windows-GUI-System setzt also eine zweistufige Hierarchie voraus. Durch den Einsatz eines Symbolbalkens ist zusätzlich eine einstufige Hierarchie möglich.

■ Datei-Menü

Die Anwendung sollte Menüoptionen für Öffnen, Speichern, Speichern unter und Drucken enthalten. Enthält die Anwendung eine Beenden-Funktion, dann ist diese Funktion hier als letzte Menüoption aufzuführen.

■ Bearbeiten-Menü

Hier sind die Menüoptionen Ausschneiden, Kopieren und Einfügen aufzuführen. Desweiteren enthält dieses Menü die Optionen Rückgängig, Wiederholen, Suchen, Ersetzen, Löschen und Duplizieren, sofern sie bei der jeweiligen Anwendung relevant sind.

■ Ansicht-Menü

Hierher gehören alle Menüoptionen, mit denen die Benutzersicht auf das zu bearbeitende Objekt beeinflusst werden kann.

■ Fenster-Menü

Hier sind alle Fenster, die im Rahmen einer MDI-Anwendung gleichzeitig geöffnet sind, aufzuführen.

■ Hilfe-Menü

Hier wird der Zugriff auf Hilfe in verschiedener Form angeboten.

Beschleunigung  
der Menüauswahl

Geübte Benutzer werden durch die Menüauswahl oft in ihrem Arbeitsfluß gehemmt. Um zügig arbeiten zu können, ist es oft notwendig, die Menüauswahl zu beschleunigen. Zur Beschleunigung der Menüauswahl gibt es unter anderem folgende Möglichkeiten:

- mnemonische Auswahl über die Tastatur,
- Auswahl über Tastaturkürzel (*accelerator key, short-cut key*),
- Symbolbalken mit Symbolen außerhalb des Menübalkens (*tool-bar*),
- Aufführung der jeweils zuletzt benutzten Objekte,
- Aufführung der häufigsten zuletzt benutzten Objekte,
- Auslagerung von Menüoptionen auf Arbeitsbereiche.

mnemonisches  
Kürzel

Im Menütitel bzw. in der Menüoption wird jeweils ein alphanumerisches Zeichen ausgewählt (im allgemeinen die Anfangsbuchstaben). Dieses Zeichen (Kürzel) wird unterstrichen dargestellt. Die Menütitel im Menübalken werden durch das gleichzeitige Drücken einer Funktionstaste (ALT-Taste bei Windows) und des Kürzels ausgewählt. Menüoptionen werden im heruntergeklappten Menü nur durch das Kürzel ausgewählt. Die Kürzel müssen nur innerhalb eines *drop-down*-Menüs eindeutig sein. Buchstaben können bei der Auswahl in Klein- und in Großschreibung eingegeben werden.



In *Winword* kann das Menü unter dem Titel *Datei* mit dem mnemonischen Kürzel ALT + »D« ausgeklappt werden. Dann kann mit »N« ein neues Dokument angelegt werden. Beispiel

Tastaturkürzel (*accelerator keys, short-cut keys*) sind Tastenkombinationen zur Beschleunigung der Auswahl innerhalb von *drop-down*-Menüs. Im Unterschied zu mnemonischen Kürzeln ist mindestens eine Taste eine Funktionstaste, die durch weitere Tasten ergänzt wird. Tastaturkürzel müssen über alle Menüoptionen des aktiven Fensters hinweg eindeutig sein. *Drop-down*-Menüs werden zuvor nicht ausgeklappt. Tastaturkürzel

In *Winword* kann mit dem Tastaturkürzel STRG + »A« der komplette Text markiert werden, ohne daß zuvor der Menütitel *Bearbeiten* selektiert wird, in dem sich die Menüoption *Alles markieren* befindet. Beispiel

Der Symbolbalken (*tool bar*) kann Schaltflächen mit Mini-Piktogrammen (*icons*) enthalten, die auf Mausklick eine zugeordnete Funktion aktivieren. Oft werden die am häufigsten benutzten Menüoptionen zusätzlich im Symbolbalken dargestellt. Der Symbolbalken kann ein geschlossenes Menü (*drop-down list box*) enthalten, das als Eigenschaftsmenü geeignet ist. Geschlossene Menüs sind platzsparend und zeigen permanent die aktuelle Option an. In manchen Anwendungen kann der Benutzer den Symbolbalken durch eigene Symbole individualisieren. Symbolbalken

Der Symbolbalken von *Winword* enthält die wichtigsten Menüoptionen – z.B. *Speichern, Drucken* – als Mini-Piktogramme. Die Schriftart wird über eine *drop-down-list-box* eingestellt. Beispiel

Die Objekte, die zuletzt benutzt wurden, werden mit ihren Pfadnamen im Menü aufgelistet. Das zuletzt benutzte Objekt steht oben usw. Die Anzahl der Objekte ist begrenzt. Die Objekte werden automatisch mit Ziffern durchnummeriert und können per Tastatur über diese Ziffern ausgewählt werden. zuletzt benutzte Objekte

Der Menütitel *Datei* von *Winword* enthält eine Liste der zuletzt bearbeiteten Dokumente. Beispiel

Die Objekte, die am häufigsten zuletzt benutzt wurden, werden als abgetrennte obere Menügruppe automatisch angeordnet. Das am häufigsten zuletzt benutzte Objekt steht oben. Die Anzahl der Objekte ist begrenzt. häufigste, zuletzt benutzte Objekte

Menüoptionen können auch als Schaltflächen (*buttons*) auf Arbeitsbereiche ausgelagert werden. Dabei kann die Menüoption erhalten bleiben oder entfallen. Menüoptionen in Arbeitsbereichen

In den Dialogfenstern von *Winword* sind zahlreiche Aktionen und Einstellungen von Parametern durch Schaltflächen realisiert. Beispiel

### Gestaltungsregeln für Menüs

#### Menütitel (von *drop-down*-Menüs)

- Einheitliche Bezeichnung und Anordnung in allen Anwendungen und Fenstern.
- Kurz, prägnant, selbsterklärend.
- Einheitlicher grammatikalischer Stil.

#### Benennung der Menüoptionen

- Kurz und prägnant.
- Einheitlich in allen Anwendungen und Fenstern.
- Dem Benutzer vertraut.

#### Gestaltung der Menüoptionen

- Linksbündig anordnen.
- Zufällige Anordnung vermeiden, stattdessen eine alphabetische Anordnung oder eine funktionale Gruppierung verwenden.
- Wenn möglich, statt einer rein sprachlichen Darstellung, zusätzlich bildhaft darstellen (z.B. Formatvorlage bei *Winword*).

#### Kaskadenmenüs

- Maximal zweistufig (in Ausnahmefällen auch dreistufig).
- Breite, flache Bäume mit etwa 8 bis 16 Gruppen.
- Aussagefähige Gruppennamen wählen, aus denen man auf die darunterliegenden Menüoptionen schließen kann.
- Gruppen möglichst disjunkt.

#### Positionierung von *pop-up*-Menüs

- Rechts, nahe dem aktiven Objekt, ohne dieses zu überdecken.

#### Mnemotechnische Kürzel

- Gleichen Menüoptionen – in mehreren Menüs – die gleichen Kürzel zuordnen.
- Leicht merkbare Kürzel wählen.

#### Abkürzungsregeln

- Abkürzungen möglichst vermeiden.
- Streichen einzelner Buchstaben (meistens von rechts nach links und meistens Vokale), z.B. Zchn für Zeichen.
- Abschneiden der letzten Buchstaben des Wortes, z.B. Dir für Directory.

### Anwendungstypen und ihre Menüs

Büro-  
kommunikation

Eine Anwendung der Bürokommunikation (z.B. Textverarbeitung, Tabellenkalkulation, Grafik) ist dadurch gekennzeichnet, daß es nur einen einzigen Objekttyp gibt (z.B. Textdokument, Rechenblatt, Grafik). Jeder Benutzer erstellt im allgemeinen mehrere Exemplare dieses Typ. Der Menübalken ist funktionsorientiert angelegt. Zur Verwaltung der Dokumente gibt es das Standardmenü *Datei* mit den Optionen *Neu*, *Öffnen*, *Schließen* etc.

Bei einer kaufmännisch-administrativen Anwendung müssen mehrere Objekttypen (z.B. Kunde, Lieferant, Auftrag) bearbeitet werden. Außerdem gibt es im allgemeinen nur eine Datenbasis, in der alle Objekte gespeichert werden. Bei diesen Anwendungen gibt es oft genauso viele Objekttypen wie Funktionen. Ein weiterer Unterschied zur Bürokommunikation ist, daß es nicht nur Fenster gibt, um ein einzelnes Objekt zu bearbeiten, sondern für jeden Objekttyp gibt es im allgemeinen ein Erfassungsfenster und ein Listenfenster.



**Dialog (dialog)** Ein Dialog ist eine Interaktion zwischen dem Benutzer und einem Dialogsystem, um ein bestimmtes Ziel zu erreichen. Ein Benutzer ist ein Mensch, der mit dem Dialogsystem arbeitet /ISO 9241-10 : 1996/. Arbeitsschritte, die zur direkten Aufgabenerfüllung dienen, bezeichnet man als Primärdialog. Benötigt der Benutzer situationsabhängig zusätzliche Informationen, dann werden diese Hilfsdienste durch Sekundärdialoge erledigt.

**Dialogmodus** Ein modaler Dialog (*modal dialog*) muß beendet sein, bevor eine andere Aufgabe der Anwendung durchgeführt werden kann. Ein nicht-modaler Dialog (*modeless dialog*) ermöglicht es dem Benutzer, den aktuellen →Dialog zu unterbrechen, während das ursprüngliche Fenster geöffnet bleibt.

**Fenstertypen** Es lassen sich folgende Fenstertypen unterscheiden: Anwendungsfenster, Unterfenster, Dialogfenster und Mitteilungsfenster. Das Anwendungsfenster erscheint nach dem Aufruf der Anwendung, Unterfenster unterstützen die Primärdialoge, Dialogfenster werden für Sekundärdialoge benötigt und Mitteilungsfenster sind spezialisierte Dialogfenster.

**Gestaltungsregelwerk (style guide)** Ein Gestaltungsregelwerk schreibt vor, wie die Benutzungsoberfläche von Anwendungen gestaltet wird. Es soll si-

cherstellen, daß das *look and feel* über verschiedene Anwendungen hinweg gleich bleibt. *Style guides* können sowohl Regelwerke des GUI-Herstellers oder auch unternehmenseigene Gestaltungsregelwerke sein.

**GUI** Ein GUI (*graphical user interface*) ist eine grafische Benutzungsoberfläche. Sie besteht aus einer Dialogkomponente (Bedienungsabläufe) und einer E/A-Komponente (Gestaltung der Informationen).

**Menü** Ein Menü besteht aus einer überschaubaren und meist vordefinierten Menge von Menüoptionen, aus der ein Benutzer eine oder mehrere auswählen kann. Bei einem Aktionsmenü lösen die Menüoptionen Anwendungsfunktion aus, bei einem Eigenschaftsmenü lassen sich Parameter einstellen. Es lassen sich *pop-up*-Menüs und Menübalken mit *drop-down*-Menüs unterscheiden.

**Prototyp** Ein Prototyp dient dazu, bestimmte Aspekte vor der Realisierung des Softwaresystems zu überprüfen. Der Prototyp der Benutzungsoberfläche zeigt die vollständige Oberfläche des zukünftigen Systems, ohne daß bereits Funktionalität realisiert ist.

**Software-Ergonomie** Die Software-Ergonomie befaßt sich mit der menschengerechten Gestaltung von Softwaresystemen. Sie verfolgt das Ziel, die Software an die Eigenschaften und Bedürfnisse der Benutzer anzupassen.



Bei der Gestaltung von Benutzungsoberflächen lassen sich Primär- und Sekundärdialoge unterscheiden. Nicht-modale Dialoge erlauben mehr Flexibilität als modale. Weiterhin gibt es verschiedene **Fenstertypen**. Zu den Primärfenstern gehören Anwendungsfenster und MDI-Unterfenster. Sekundärfenster sind Dialogfenster und Mitteilungsfenster. Bei **Menüs** lassen sich zwei Arten unterscheiden: Menübalken mit *drop-down*-Menü sowie *pop-up*-Menüs. Während

## LE 9 Zusammenhänge/Aufgaben

Anwendungen der Bürokommunikation nur einen Objekttyp verwenden, müssen kaufmännisch-administrative Anwendungen mit vielen Objekttypen arbeiten.

Aufgabe  
5–10 Minuten

- 1 Lernziel: Möglichkeiten zur Auswahl einer Aktion im Dialog.**  
Beschreiben Sie, welche Möglichkeiten es gibt, eine Aktion (Operation) im Dialog über das Menü auszuwählen. Geben Sie an, wie schnell diese Auswahl jeweils erfolgen kann (z.B. Anzahl der Mausklicks, Länge der Mausbewegung, Menge der Eingaben über die Tastatur).

Aufgabe  
10–15 Minuten

- 2 Lernziel: Grundsätzliche Dialoggestaltung erkennen können.**  
Analysieren Sie Ihr bevorzugtes Textverarbeitungssystem (in der Musterlösung: *Winword*) oder ein anderes Anwendungsprogramm und bearbeiten Sie folgende Teilaufgaben:
- a** Nennen einiger Primärdialoge.
  - b** Nennen einiger Sekundärdialoge.
  - c** Nennen einiger modaler Dialoge.
  - d** Nennen einiger nicht-modaler Dialoge.
  - e** Angeben, wo die eine objektorientierte Bedienung mit Menüs vorliegt.
  - f** Angeben, wo eine funktionsorientierte Bedienung vorliegt.
  - g** Angeben, wo die direkte Manipulation verwendet wurde.

Aufgabe  
5–10 Minuten

- 3 Lernziel: Die verschiedenen Fenstertypen und ihren Einsatzbereich kennen.**  
Analysieren Sie Ihr bevorzugtes Textverarbeitungssystem (in der Musterlösung: *Winword*) oder ein anderes Anwendungsprogramm und bearbeiten Sie folgende Teilaufgaben:
- a** Für jeden Fenstertyp ein Beispiel angeben.
  - b** Angeben, ob es sich um eine SDI oder eine MDI-Anwendung handelt.

Aufgabe  
5–10 Minuten

- 4 Lernziel: Menüs und ihre Einsatzbereiche kennen.**  
Analysieren Sie Ihr bevorzugtes Textverarbeitungssystem (in der Musterlösung: *Winword*) oder ein anderes Anwendungsprogramm und geben Sie – sofern vorhanden – folgendes an:
- a** Aktionsmenü.
  - b** Eigenschaftsmenü.
  - c** *pop up*-Menü.
  - d** Benutzte Möglichkeiten zur Beschleunigung der Menüauswahl.

## 5 Gestaltung von Benutzungsoberflächen (Teil 2)



- Erklären können, welche Interaktionselemente es gibt und wie sie eingesetzt werden.
- Erklären können, wie Gruppierung und Hervorhebung bei der Gestaltung von Fenstern eingesetzt werden.
- Ein Klassendiagramm in eine Dialogstruktur transformieren können.
- Fenster mittels Interaktionselementen gestalten können.

verstehen

anwenden



- Die Inhalte der Kapitel 5.1 bis 5.4 müssen bekannt sein.
- Für das Verständnis von Kapitel 5.5 sind die objektorientierten Konzepte und die UML-Notation, wie sie in Kapitel 2 beschrieben wurden, Voraussetzung.



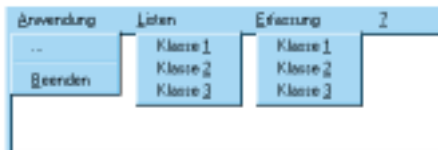
- 5.5 Vom Klassendiagramm zur Dialogstruktur 210
- 5.6 Interaktionselemente 215
- 5.7 Gestaltung von Fenstern 221

## 5.5 Vom Klassendiagramm zur Dialogstruktur

**Definition** Aus dem Klassendiagramm kann systematisch eine objektorientierte Dialogstruktur abgeleitet werden. Wir verwenden die Transformationsregeln in Anlehnung an /Balzert 96/. Die grundlegende Idee der Transformation ist, daß jede Klasse des Analysemodells auf ein Erfassungsfenster und ein Listenfenster abgebildet wird.

**Menübalken** Der Menübalken enthält je ein *drop-down*-Menü für Listenfenster und Erfassungsfenster (Abb.5.5-1). Für jede Klasse des OOA-Modells ist zu prüfen, ob für die betreffenden Daten eine Listenausgabe sinnvoll ist und ob die Daten über einen separaten Dialog erfaßt und geändert werden sollen. Die ermittelten Klassen werden in den Menüs *Listen* bzw. *Erfassung* aufgeführt. Wenn zu viele Klassen vorliegen, dann werden sie zusätzlich – z.B. mittels Paketen – gruppiert. Natürlich sind auch andere Anordnungen der Klassen möglich.

Abb. 5.5-1:  
Abbildung der  
Klassen auf Menüs



**Dialogstruktur Klasse**

Das **Erfassungsfenster** bezieht sich auf ein einzelnes Objekt der Klasse (Abb. 5.5-2). Jedes Attribut der Klasse wird – entsprechend seines Typs – auf ein grafisches Interaktionselement im Erfassungsfenster abgebildet. Jede Operation der Klasse wird auf eine Menüoption innerhalb eines *pop-up*-Menüs oder auf eine Schaltfläche (*button*) abgebildet. Das Erfassungsfenster dient zum Erfassen und zum Ändern eines Objekts. Die Schaltflächen besitzen folgende semantische Bedeutung:

- Ok: Speichern des Objekts und Schließen des Fensters.
- Übernehmen: Speichern des Objekts ohne das Fenster zu schließen. Da im allgemeinen anschließend ein anderes Objekt erfaßt wird, werden alle Felder des Fensters neu initialisiert.
- Abbrechen: Schließen des Fensters und Verwerfen der Eingabe.
- Liste: Öffnen des zugehörigen Listenfensters, während das Erfassungsfenster geöffnet bleibt.

Das **Listenfenster** zeigt alle Objekte der Klasse an (Abb. 5.5-2). Meistens werden die Objekte im Listenfenster nur durch einen Teil der Attribute beschrieben. Der Bediener soll die wichtigsten Attribute auf einen Blick sehen und kann bei Bedarf das Erfassungsfenster des entsprechenden Objekts öffnen.

Klassenattribute und -operationen beziehen sich auf alle Objekte der Klasse und werden daher im Listenfenster dargestellt. Klassenattribute werden auf Interaktionselemente, Klassenoperationen auf Menüoptionen bzw. Schaltflächen abgebildet.

## 5.5 Vom Klassendiagramm zur Dialogstruktur LE 10

Artikel
Nummer
Bezeichnung
Preis

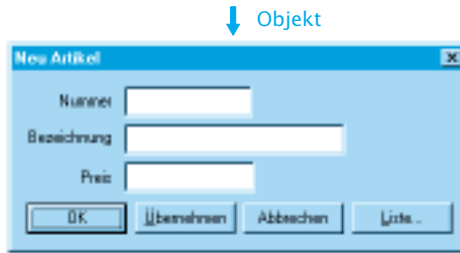


Abb. 5.5-2:  
Abbildung einer Klasse auf Erfassungs- und Listenfenster

Die abgebildeten Schaltflächen besitzen folgende semantische Bedeutung:

- Neu: Öffnen eines leeren Erfassungsfensters.
- Ändern: Öffnen des Erfassungsfensters für das selektierte Objekt.
- Löschen: Löschen des selektierten Objekts.
- Schließen: Schließen des Listenfensters.

Jedes Fenster ist über eine entsprechende Menüoption erreichbar und der Benutzer kann jederzeit zwischen allen geöffneten Erfassungs- und Listenfenstern wechseln (Abb. 5.5-3).

Assoziationen erlauben es den Benutzern, durch ein Netz von Objekten zu traversieren. Bei einer fertigen Anwendung werden viele Objektverbindungen durch die implementierten Operationen auf-

Dialogstruktur  
Assoziation

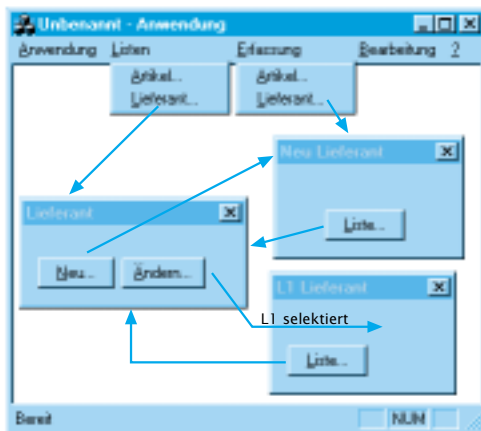


Abb. 5.5-3:  
Erreichbarkeit von Erfassungs- und Listenfenster

## LE 10 5 Benutzungsoberflächen

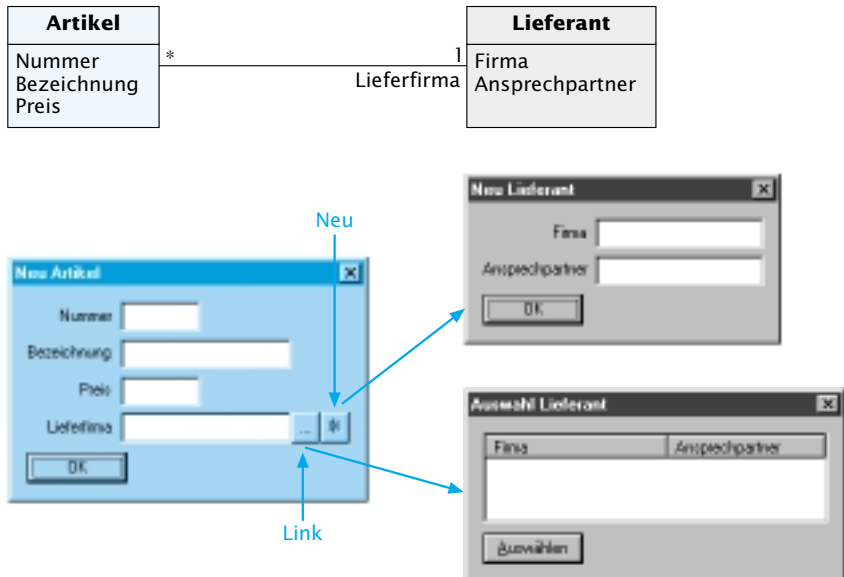
gebaut und geändert. Einige Verbindungen werden aber auch weiterhin über den Dialog erstellt. Im OOA-Modell sind alle Assoziationen inhärent bidirektional. Sie werden daher im folgenden auch bidirektional auf den Prototyp abgebildet. Es kann jedoch durchaus reichen, wenn einige Assoziationen nur in einer Richtung realisiert werden. Dann kann auch die entsprechende Richtung in der Dialogstruktur entfallen.

Das Erstellen und Entfernen von Objektverbindungen wird in das Erfassungsfenster der betreffenden Klassen integriert. Für jedes Erfassungsfenster ist darzustellen, zu welchen Klassen Verbindungen möglich sind, welche Objekte dieser Klassen existieren und mit welchen Objekten bereits eine Verbindung besteht. Verbindungen zu anderen Objekten können aufgebaut und auch wieder getrennt werden.

### 1-Assoziation

Im Erfassungsfenster der Abb. 5.5-4 ist jeder Artikel mit genau einem Lieferanten verbunden. Die Verbindung wird für den Benutzer sichtbar durch das Attribut Lieferfirma in einem Textfeld des Erfassungsfensters von Artikel dargestellt. Bei einer Muß-Verbindung handelt es sich um ein Muß-Feld, d.h. das Erfassungsfenster kann erst dann geschlossen werden, wenn die Verbindung existiert. Dieses Textfeld wird um zwei Schaltflächen ergänzt. Die Neu-Schaltfläche »\*« öffnet ein leeres Erfassungsfenster des Lieferanten. Ein neu erfaßter Lieferant wird automatisch mit dem Artikel verbunden. Die Link-Schaltfläche »...« öffnet ein Auswahlfenster, das alle Lieferanten anzeigt. Der selektierte Lieferant wird dem Artikel zugeordnet. In beiden Fällen wird in das Textfeld Lieferfirma die Firma

Abb. 5.5-4:  
Darstellung einer  
1-Assoziation





des zugehörigen Lieferanten automatisch eingetragen. Dabei ist wichtig, daß es sich bei Firma um ein Schlüsselattribut handelt.

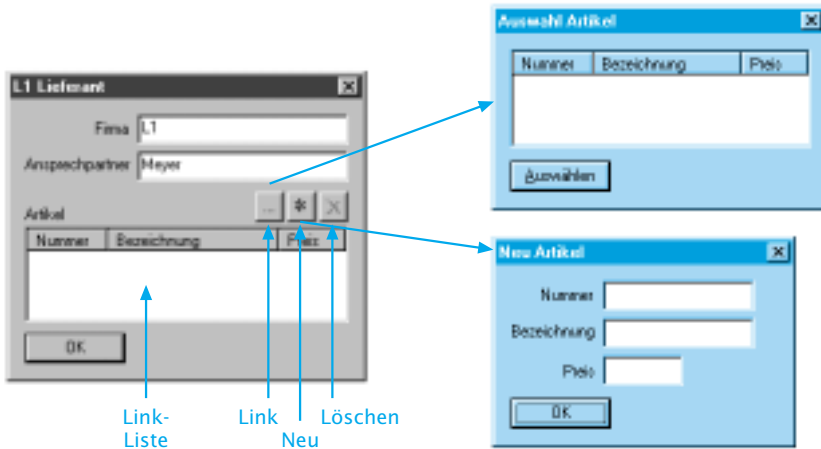
Etwas komplizierter ist der Fall für *many*-Assoziationen. In der Abb. 5.5-5 gibt es zu einem Lieferanten mehrere Artikel. Daher wird das Erfassungsfenster des Lieferanten um eine *Link*-Liste der zugehörigen Artikel erweitert. Analog zu oben wird mit der Neu-Schaltfläche »\*« das Erfassungsfenster von Artikel geöffnet und der neu erfaßte Artikel in die *Link*-Liste eingetragen. Mit der *Link*-Schaltfläche »...« wird ein vorhandener Artikel ausgewählt und in die *Link*-Liste eingetragen. Die Löschen-Schaltfläche »X« ermöglicht es, einen Artikel in der *Link*-Liste zu löschen. Damit wird jedoch nur die Verbindung zu diesem Artikel gelöscht, nicht der Artikel selbst. Wenn von einer Klasse mehrere *many*-Assoziationen ausgehen, kann jede *Link*-Liste platzsparend auf einer Seite eines Registers (siehe Kapitel 5.6) dargestellt werden.

*many*-Assoziation

Kapitel 5.6



Abb. 5.5-5: Darstellung einer *many*-Assoziation

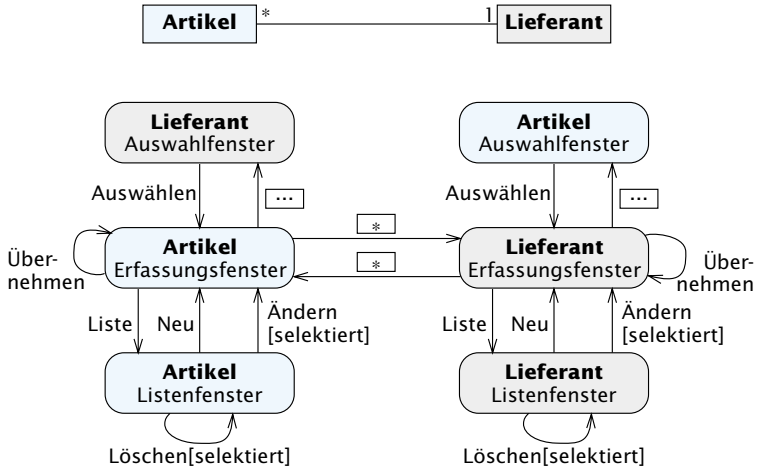


Für die Klassen Artikel und Lieferant ergibt sich die im Zustandsdiagramm der Abb. 5.5-6 dargestellte Dialogstruktur. Jedes Ereignis entspricht einer Schaltfläche, die auf dem jeweiligen Fenster angeboten wird. Der Vorteil dieser systematischen Transformation liegt insbesondere darin, daß eine konsistente Dialogstruktur entsteht, die der Benutzer schnell erlernen kann.

Dialogstruktur

## LE 10 5 Benutzungsoberflächen

Abb. 5.5-6:  
Dialogstruktur

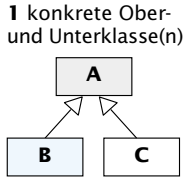


Dialogstruktur  
Einfachvererbung

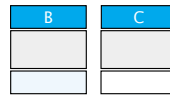
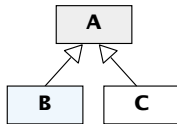
Um eine Vererbungsstruktur auf den Dialog abzubilden, gibt es verschiedene Möglichkeiten, die in Abb. 5.5-7 dargestellt sind.

- Bei einer konkreten Oberklasse wird außer den Unterklassen auch die Oberklasse auf ein Fenster abgebildet. Die Fenster der Unterklassen enthalten zusätzlich zu den eigenen Eigenschaften und Operationen alle Elemente der Oberklasse. In der Abb. 5.5-7 besteht das Fenster der Klasse B aus eigenen – blau dargestellten

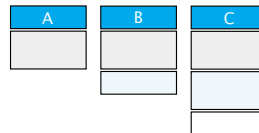
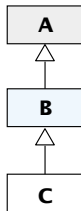
Abb. 5.5-7:  
Transformation  
der Einfach-  
vererbung



2 abstrakte Oberklasse, konkrete Unterklasse(n)



3 mehrstufige Vererbung



Elementen – und aus den von der Klasse A geerbten – grau dargestellten – Elementen.

- 2 Ist die Oberklasse abstrakt, dann taucht sie *nicht* als eigenständiges Fenster auf. Wie bei 1 enthalten die Fenster der Unterklassen zusätzlich zu den eigenen die geerbten Elemente.
- 3 Bei einer mehrstufigen Vererbungsstruktur ist analog zu 1 bzw. 2 zu verfahren.

Die ererbten Attribute sollten in den Fenstern der Unterklassen einheitlich präsentiert werden, damit der Benutzer erkennt, daß es sich um dieselben Elemente handelt. Reicht der Platz in einem Erfassungsfenster nicht aus, dann ist ein Register (siehe Kapitel 5.6) zu verwenden. Alle Muß-Attribute sollten möglichst auf der ersten Seite stehen.



Kapitel 5.6

Viele Anwendungen enthalten eine Reihe von anwendungsneutralen Funktionen, die noch hinzugefügt werden müssen. Das könnten beispielsweise sein:

anwendungs-  
neutrale  
Funktionen

- eine Funktion zur Änderung von Kennwörtern,
- Funktionen zum Initialisieren und zur Definition von Voreinstellungen,
- Funktionen zum Hinzufügen bzw. Entfernen von Benutzern oder Benutzerprivilegien.

## 5.6 Interaktionselemente

Jedes GUI-System verfügt über eine Menge von **Interaktionselementen** bzw. Steuerelemente (*controls*), wobei es zwischen verschiedenen GUI-Systemen einige Unterschiede gibt. Abb. 5.6-1 gibt einen Überblick über die wichtigsten Interaktionselemente von Windows /MS 95/. Die deutsche Übersetzung der einzelnen Steuerelemente lehnt sich an die internationale Terminologie für Windows-Oberflächen an /GUI Guide 93/. Es wurden jedoch auch andere übliche Begriffe eingeführt /Balzert 96/.

Das **Textfeld** bzw. **Eingabefeld** (*text box, edit control*) dient zur Ein- und Ausgabe von numerischen Daten oder Texten in einer einzigen Zeile.

Textfeld

- Bei Zahlenwerten und Daten sollte das Feld alle Zeichen darstellen können. Ist die Anzahl der eingegebenen Zeichen variabel – wie bei fast allen Texten – dann ist das Feld so zu dimensionieren, daß die Mehrzahl der Eingaben komplett in das Feld paßt. Für extrem lange Eingaben ist der Eingabebereich zu verschieben.
- Der Benutzer soll obligatorische und optionale Eingaben (Muß- und Kann-Felder) unterscheiden können. Das kann beispielsweise durch unterschiedliche Untergrundtönungen erreicht werden.

## LE 10 5 Benutzungsoberflächen

- Häufig vorkommende Eingaben sollen als Standardvorbelegung im Feld stehen. Es muß erkennbar sein, daß sie geändert werden können.
  - Zahlen werden rechtsbündig angeordnet. Alle nicht-numerischen Eingaben sollen bei diesen Feldern abgewiesen werden.
  - Texte werden linksbündig angeordnet.
  - Felder, die nur zur Ausgabe dienen, sind zu kennzeichnen. Außerdem sind sie für Eingaben zu sperren.
- mehrzeilige Textfelder **Mehrzeilige Textfelder** (*multi-line edit field*) dienen zur Ein- und Ausgabe von Texten.
- In einem Textfenster sollen mindestens vier Zeilen Text sichtbar sein.
  - Um längere Texte platzsparend einzugeben, werden Rollbalken verwendet. Vertikale Rollbalken sind horizontalen vorzuziehen.
  - Texteingaben werden grundsätzlich linksbündig angeordnet.
  - Die Anzahl der Zeichen pro Zeile sollte zwischen 40 und 60 liegen.
- Drehfeld Das **Drehfeld** (*spin box, spin button*) ist die Kombination eines Textfeldes mit einem *up-down control*. Es bietet eine Menge von geordneten Eingabewerten, wobei der gewählte Wert im Textfeld sichtbar ist. Der Bediener kann mit den Auf- und Ab-Pfeilen die Alternativen traversieren oder direkt einen Wert eingeben.
- Schaltfläche Mit der **Schaltfläche** (*command button, push button*) wird eine Aktion ausgelöst oder eine Bestätigung durchgeführt. Sie wird nur kurzzeitig aktiviert. Anschließend kehrt sie in den inaktiven Zustand zurück.
- Jede Schaltfläche muß eine Beschriftung oder ein Symbol (Piktogramm) enthalten.
  - Die Beschriftung soll aus einem Wort bestehen, mit Großbuchstaben beginnen und die entsprechende Funktionalität genau beschreiben.
  - Eine Gruppe von Schaltflächen soll möglichst horizontal als Leiste dargestellt werden, kann aber auch vertikal angeordnet werden.
  - Eine Schaltfläche innerhalb einer Gruppe kann als Standardvorgabe (*default*) gekennzeichnet sein. Sie wird dann durch die *Enter*-Taste ausgelöst.
  - Eine mnemonische Auswahl (Auswahlzeichen unterstrichen) sollte nur bei Schaltflächen möglich sein, die nicht bereits einer Funktionstaste zugeordnet sind.
- Optionsfeld Das **Optionsfeld** (*option button, radio button*) ermöglicht eine Einfachauswahl unter mehreren Alternativen. In einer Gruppe von Optionsfeldern kann also nur eines gewählt werden. Optionsfelder werden als kleine Kreise dargestellt und die gewählte Alternative durch einen Punkt markiert. Die Bedeutung eines jeden Optionsfeldes wird durch eine Beschriftung rechts vom Kreis erläutert.

- Eine spaltenweise Anordnung der Alternativen ist einer zeilenweisen oft vorzuziehen.
- Bei einer zeilenweisen Anordnung sind drei Zeichen Abstand einzuhalten.
- Eine Spalte sollte maximal acht Alternativen enthalten.
- Kann eine Alternative in einer bestimmten Situation nicht gewählt werden, dann wird sie grau dargestellt (*disabled*).
- Dieses Interaktionselement ist nur einzusetzen, wenn die Alternativen bereits zum Zeitpunkt der Oberflächengestaltung bekannt sind und stabil bleiben.

Das **Kontrollkästchen** (*check box*) erlaubt eine Mehrfachauswahl, d.h. eine n-aus-m-Auswahl. Dabei kann n zwischen 0 und m liegen. Als Sonderfall kann auch die 0-aus-1-Auswahl vorkommen. Das Kontrollkästchen besteht aus einem Quadrat mit einer nebenstehenden Beschriftung. Ausgewählte Möglichkeiten werden markiert. Im Gegensatz zur Einfachauswahl müssen sich die Möglichkeiten nicht gegenseitig ausschließen.

Kontrollkästchen

- Eine spaltenweise Anordnung der Auswahlmöglichkeiten ist einer zeilenweisen Darstellung oft vorzuziehen.
- Bei einer zeilenweisen Anordnung sind drei Zeichen Abstand einzuhalten.
- Eine Spalte sollte maximal acht Auswahlmöglichkeiten enthalten.
- Kann eine Möglichkeit in einer bestimmten Situation nicht gewählt werden, dann wird sie grau dargestellt (*disabled*).
- Dieses Interaktionselement ist nur einzusetzen, wenn die Auswahlmöglichkeiten bereits zum Zeitpunkt der Oberflächengestaltung bekannt sind und stabil bleiben.

Das **Listenfeld** bzw. die **Auswahlliste** (*list box*) dient zur Darstellung mehrerer vertikal angeordneter alphanumerischer oder grafischer Listeneinträge. Bei einer *single selection list box* kann maximal ein Element gewählt werden. Eine *multiple selection list box* ermöglicht es, mehrere Einträge zu selektieren. In der *extended selection list box* können zusammenhängende Bereiche gewählt werden.

Listefeld

- Vertikale Rollbalken ermöglichen das Blättern in einer Liste mit vielen Einträgen. Auf horizontale Rollbalken ist zu verzichten.
- Um das Lesen der Listeneinträge nicht zu stören, sollten mindestens vier Zeilen gleichzeitig sichtbar sein.
- Die Einträge werden von der Anwendung gefüllt. Die Anzahl der Einträge ist in der Regel umfangreich und variabel.
- Das Listenfeld wird auch dann benutzt, wenn die Anzahl der Elemente eine Darstellung durch Optionsfelder nicht mehr zulässt. Das ist bei mehr als acht Elementen der Fall.

Das **Kombinationsfeld** (*combo box*) kombiniert die Eigenschaften des Textfeldes mit dem Listenfeld. Die Information kann entweder direkt eingetippt oder in der Liste selektiert werden. Es lassen sich zwei Varianten unterscheiden. Bei der einen wird jedes neu eingege-

Kombinationsfeld

## LE 10 5 Benutzungsoberflächen

bene Element in die Liste aufgenommen und steht bei der nächsten Benutzung zur Selektion zur Verfügung. Im zweiten Fall kann ein Element zwar eingetippt werden, wird aber nicht in der Liste gespeichert.

Dropdown-  
Listenfeld

Das **Dropdown-Listenfeld** bzw. die **Klappliste** (*drop-down list box*) ist die platzsparende Variante des Listenfeldes. Vor dem Selektieren muß die Liste aufgeklappt werden, danach ist sie wieder unsichtbar. Das vom Benutzer selektierte Element wird ständig angezeigt. Die aufgeklappte Liste kann zeitweilig andere Interaktionselemente überdecken.

- Es gelten die Gestaltungsregeln des Listenfeldes.
- Wegen seines ähnlichen Aufbaus kann das *Dropdown-Listenfeld* gut mit Textfeldern kombiniert werden.
- Gegebenenfalls kann eine Voreinstellung gewählt werden.

Dropdown-  
Kombinationsfeld

Wird das *Dropdown-Listenfeld* mit einem Textfeld kombiniert, dann entsteht das **Dropdown-Kombinationsfeld** (*drop-down combo box*). Der Benutzer kann Daten entweder direkt eingeben oder aus dem *Dropdown-Listenfeld* auswählen. Wie beim Kombinationsfeld kann ein neu eingegebenes Element entweder in die Liste aufgenommen werden und später zur Selektion zur Verfügung stehen oder nur eingetippt, aber nicht gespeichert werden.

Wie Abb. 5.6-1 zeigt, sind das *Dropdown-Listenfeld* und das *Dropdown-Kombinationsfeld* optisch nicht zu unterscheiden.

Listenelement

Das **Listenelement** bzw. die **Tabelle** (*list view control*) ist eine Erweiterung des Listenfeldes (*list box*). Für die Darstellung der Einträge gibt es vier Varianten:

- Piktogramme mit Beschriftung, die in dem Listenelement beliebig positioniert werden können.
- Mini-Piktogramme mit Beschriftung, die in dem Listenelement beliebig positioniert werden können.
- Liste aller Einträge, von denen jedes aus einem Mini-Piktogramm und der Beschriftung besteht, die spaltenweise sortiert sind.
- Als »Report« benötigt jeder Eintrag eine Zeile, wobei jede Zeile aus mehreren Spalten besteht. Die Breite der Spalten ist durch den Benutzer variabel einstellbar. Wie im Kapitel 5.5 gezeigt wird, kann jeder Eintrag ein Objekt repräsentieren, wobei die Spalten den Attributen der Klasse entsprechen.





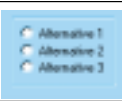





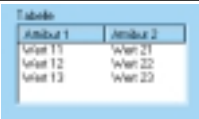


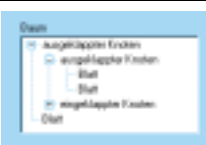
Kapitel 5.5

Regler

Der **Regler** bzw. **Schieberegler** (*slider*) zeigt den Wert, die Größe oder die Position von etwas auf einer Skala an. Oft kann der Regler vom Benutzer verstellt werden. Der Regler sollte dann benutzt werden, wenn es nicht darum geht, einen genauen, sondern nur einen relativen Wert einzugeben (z.B. Doppelklick-Geschwindigkeit der Maus).



Abb. 5.6-1:  
Interaktions-  
elemente

Bezeichnung	engl. Bezeichnung	Beispiel
Textfeld Eingabefeld	<i>text box, edit control</i>	
Mehrzeiliges Textfeld	<i>multi-line text box</i>	
Drehfeld	<i>spin box</i>	
Schaltfläche, Druckknopf	<i>command button, push button</i>	
Optionsfeld, Einfach- auswahlknopf	<i>option button, radio button</i>	
Kontrollkästchen, Mehrfach- auswahlknopf	<i>check box</i>	
Listenfeld, Auswahlliste	<i>list box</i>	
Kombinationsfeld	<i>combo box</i>	
Dropdown-Listenfeld, Klappliste	<i>drop-down list box</i>	
Dropdown- Kombinationsfeld	<i>drop-down combo box</i>	
Listenelement, Tabelle	<i>list view control</i>	
Regler, Schieberegler	<i>slider</i>	
Register	<i>tab control, property sheet</i>	
Strukturansicht, Baum	<i>tree view control</i>	

Ein **Register** (*tab control, property sheet, notebook*) besteht aus mehreren Seiten, von denen zu einem Zeitpunkt immer nur eine Seite angezeigt wird. Alle Seiten müssen gleich groß sein. Es lassen sich drei Varianten unterscheiden:

Register

## LE 10 5 Benutzungsoberflächen

- Ganzseitiges Register mit Steuerelementen (z.B. Ok- und Abbrechen-Schaltfläche). Diese Steuerelemente wirken nur auf diejenige Seite, auf der sie angeordnet sind.
- Ganzseitiges Register. Die Steuerelemente befinden sich außerhalb des Registers im gleichen Fenster und wirken daher auf alle Seiten.
- Register als Interaktionselement in einem Fenster.

Das Register kann immer dann eingesetzt werden, wenn viele Informationen dargestellt werden müssen, die ein einziges Fenster überladen würden und zu einem Zeitpunkt nur ein Teil der Informationen benötigt wird.

Strukturansicht In der **Strukturansicht** bzw. dem **Baum** (*tree view control*) sind die Einträge hierarchisch angeordnet. Er enthält Schaltflächen, die es erlauben, die nächste Ebene eines Knotens anzuzeigen (*expand*) oder zu verbergen (*collapse*). Jeder Knoten des Baums wird durch einen Text und ein optionales Mini-Piktogramm dargestellt. Damit können die Daten auf verschiedenen Abstraktionsebenen dargestellt werden und der Benutzer kann schneller in der Baumstruktur navigieren als es in einer Liste möglich wäre.

Führungstext Die meisten Interaktionselemente benötigen einen **Führungstext** (*static text*), der erklärt, welche Bedeutung das Element hat und was eingetragen werden soll.

- Er soll kurz, aussagekräftig, eindeutig und präzise sein.
- Er soll nicht aus mehreren Worten bestehen und – außer den allgemein üblichen – keine Abkürzungen enthalten.
- Bei einzeiligen Interaktionselementen steht der Führungstext links davon, wobei beide Elemente horizontal zu zentrieren sind.
- Bei mehrzeiligen Interaktionselementen steht der Führungstext links ausgerichtet darüber.
- Ist die Länge der verschiedenen Führungstexte fast gleich (weniger als 6 Zeichen), dann sind sie linksbündig auszurichten, ansonsten rechtsbündig.
- Jeder Führungstext soll durch räumliche Nähe mit dem Element assoziiert sein, wobei der minimale Abstand ein Zeichen breit ist.
- Auf ein Trennzeichen (z.B. Doppelpunkt) zwischen Führungstext und Element ist zu verzichten.

Kombination Bei der Gestaltung von Fenstern können auch mehrere Interaktionselemente kombiniert werden.

Beispiel Für ein Projekt werden alle Rollen aufgeführt, in denen die Mitarbeiter aktiv werden können. Ein Mitarbeiter kann in einem Projekt mehrere Rollen ausfüllen, z.B. Analytiker und Projektleiter, wobei sich diese Zuordnung ändern kann. Diese Problemstellung kann durch zwei Listenfelder und zwei Schaltflächen realisiert werden (Abb. 5.6-2). Das linke Listenfeld enthält alle Rollen in einem Projekt, das rechte die Rollen des jeweiligen Mitarbeiters. Mit den



Schaltflächen »>>« und »<<<« werden Rollen für einen Mitarbeiter hinzugefügt bzw. entfernt.



Abb. 5.6-2:  
Kombination von  
Interaktions-  
elementen

## 5.7 Gestaltung von Fenstern

Der Benutzer muß optisch durch ein Fenster geführt werden. Zwei wichtige Gestaltungsmittel sind Gruppierung und Hervorhebung. Dabei ist zu berücksichtigen, daß alle Gestaltungsmittel sowohl innerhalb eines Fensters als auch über alle Fenster der Anwendung hinweg konsistent verwendet werden.

Semantisch zusammengehörende Elemente sollen gruppiert werden, denn durch geeignete Gruppierung kann die Suchzeit in einem Fenster reduziert werden. Der Benutzer orientiert sich zuerst an den Gruppen, dann an deren Inhalten.

Gruppierung

- 1 Informationen im oberen Bereich einer Gruppe werden schneller entdeckt als im unteren Bereich.
- 2 Die Elemente werden in der Gruppe so angeordnet, wie es der Arbeitsablauf des Benutzers erfordert.
- 3 Für das Suchen und Vergleichen von Elementen innerhalb einer Gruppe ist es günstiger, die Elemente spaltenweise statt zeilenweise anzuordnen.
- 4 Gruppenüberschriften erhöhen zwar die Übersichtlichkeit, sie vergrößern jedoch die dargestellte Informationsmenge und den für ihre Darstellung nötigen Raumbedarf.
- 5 Eine Gruppe sollte nicht mehr als vier oder fünf Elemente enthalten, damit das gesuchte Element unmittelbar in dieser Gruppe identifiziert werden kann.
- 6 Bei fachlicher Notwendigkeit müssen natürlich größere Gruppen gebildet werden. Zur besseren Orientierung sollten dann Gruppenüberschriften angegeben werden.
- 7 Um einen umfassenden Überblick über mehrere Gruppierungen zu ermöglichen, sollte die Anzahl der Gruppen nicht größer als vier oder fünf sein.
- 8 Wenn es nicht auf den umfassenden Überblick über alle Gruppierungen ankommt, können bis zu 15 Gruppen gewählt werden.
- 9 Gruppen sollten deutlich voneinander getrennt werden.

**Gestaltungs-  
regeln  
Gruppierung**

Manchmal ist es vorteilhaft, die Aufmerksamkeit des Benutzers auf bestimmte Bereiche zu lenken. Dies sollte in der Regel durch

Hervorhebung

## LE 10 5 Benutzungsoberflächen

eine Hervorhebung geschehen. Prinzipielle Möglichkeiten für eine Hervorhebung sind Größe, Farbe/Kontrast, Isolierung/Einzelstellung oder Umrandung/abweichende Orientierung.

### **Gestaltungsregeln Hervorhebung**

- 1 Maximal 10 bis 20 Prozent aller Einzelinformationen hervorheben.
- 2 Farben sparsam verwenden, maximal fünf Farben.
- 3 Von den verschiedenen Arten der Hervorhebung sparsam Gebrauch machen.
- 4 Eine Hervorhebung durch Blinken ist zu vermeiden, da Blinken in der Regel zu einer unnötigen Ablenkung des Benutzers führt.

Farbe Die Verwendung von Farbe kann die visuelle Informationsverarbeitung wirksam unterstützen. Unterschiedliche Farben werden schneller erkannt als verschiedene Größen oder Helligkeiten. Die wichtigsten Bildschirmfarben besitzen folgende Helligkeitsrangfolge:  
Weiß – Gelb – Cyan – Grün – Magenta – Rot – Blau – Schwarz.

### **Gestaltungsregeln Farbe**

- 1 Gestalten Sie farbig und nicht bunt.
- 2 Vor einem dunklen Hintergrund sind Weiß, Gelb, Cyan und Grün am besten geeignet, vor einem hellen Hintergrund jedoch Magenta, Rot, Blau und Schwarz.
- 3 Unterschiedliche Farben sind sparsam einzusetzen, da Farben die Aufmerksamkeit stark lenken.
- 4 Über die verschiedenen Bildschirmseiten hinweg sollen maximal sieben Farben verwendet sein. Eine Ausnahme bilden graduelle Abstufungen des Farbtons.
- 5 Farben sind konsistent zu verwenden.
- 6 Konventionelle Farbkodierungen sind einzuhalten: Rot für halt, heiß, Gefahr; Grün für weiter, sicher; Gelb für Vorsicht; Blau für kalt und Beruhigung.
- 7 Farbtonunterschiede im Rot- und Purpurbereich sind schwieriger zu erkennen als im Gelb- und Blaubereich.

### harmonische Gestaltung

Ein Dialogfenster soll nicht nur so gestaltet werden, daß der Benutzer seine Aufgaben schnell durchführen kann, sondern es soll auch ästhetisch ansprechend sein /Kruschinski 99/.

#### **1 Proportionen**

Einem Betrachter erscheinen Flächen angenehmer, wenn diese eher breit als hoch sind. Daher sollten Fenster ein Seitenverhältnis von 1:1 bis 1:2 (Höhe zu Breite) besitzen. Diese Forderung läßt sich meistens durch eine Verteilung der Informationen in zwei Spalten verwirklichen.

Abb. 5.7-1 zeigt zwei Fenster, die exakt die gleichen Informationen über ein Auto enthalten. Das obere Fenster unterstützt im Gegensatz zum unteren die harmonische Gestaltung durch Spaltenbildung. Weitere Gestaltungselemente wurden hier bewußt noch nicht eingesetzt.

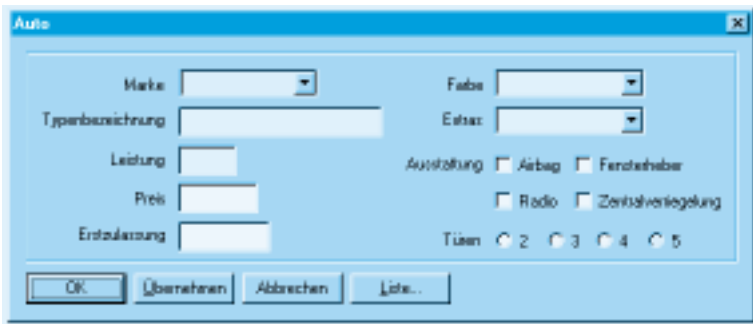


Abb. 5.7-1:  
Verschiedene  
Proportionen durch  
Spaltenbildung



## 2 Balance

Wenn ein Fenster durch eine vertikale Linie in der Mitte geteilt würde, dann soll die Informationsdichte auf beiden Seiten gleich groß sein.

Die Forderung der Balance wird durch das Fenster des Hilfsassistenten in Abb. 5.7-2 erfüllt. Außerdem wurden hier Gruppen gebildet, um zu zeigen, daß bestimmte Textfelder zusammengehören.

## 3 Symmetrie

Die Symmetrie ist eine Verstärkung der Balance. Hier wird zusätzlich gefordert, daß horizontal gegenüberliegende Elemente gleichartig sind. Diese Gleichartigkeit kann durch identische Interaktionselemente oder durch gleich große Elemente erreicht werden. In der Praxis läßt sich diese Forderung jedoch nicht immer erfüllen.

Während das Fenster der Abb. 5.7-2 nur die Forderung der Balance erfüllt, ist das Fenster des Hilfsassistenten in Abb. 5.7-3 symmetrisch gestaltet, weil die beiden Gruppen gegenüberliegen.

## LE 10 5 Benutzungsoberflächen

Abb. 5.7-2:  
Balanciertes  
Fenster

Hilfsassistent

Name

Vorname

Studienadresse

Straße

PLZ

Ort

Telefon

Geburtsdatum

Heimatadresse

Straße

PLZ

Ort

Telefon

Matrikel-Nr.

Arbeitszeugnis

Datum Arbeitszeugnis

OK Übernehmen Abbrechen Liste...

Abb. 5.7-3:  
Symmetrisches  
Fenster

Hilfsassistent

Name

Vorname

Geburtsdatum

Studienadresse

Straße

PLZ

Ort

Telefon

Matrikel-Nr.

Arbeitszeugnis

Datum Arbeitszeugnis

Heimatadresse

Straße

PLZ

Ort

Telefon

Matrikel-Nr.

OK Übernehmen Abbrechen Liste...

### 4 Sequenz

Das Auge des Benutzers soll sequentiell durch das Fenster geführt werden und keine unnötigen Sprünge machen müssen. Die wichtigsten Informationen sollten oben links zu finden sein, denn auf diesen Bereich schaut der Benutzer zuerst.

### 5 Einfachheit

Gestalten Sie jedes Fenster so einfach wie möglich. Verwenden Sie verschiedene Schriftarten oder Farben sehr zurückhaltend. Verwenden Sie Interaktionselemente nicht nur deswegen, weil sie existieren.

### 6 Virtuelle Linien minimieren

Außer den gezeichneten Linien gibt es in einem Fenster auch virtuelle Linien, die durch die Kanten der Interaktionselemente gebildet werden. Der Einfluß dieser Linien auf die harmonische Gestaltung darf nicht unterschätzt werden. Der Benutzer bildet intuitiv diese Linien, wenn genügend Fangpunkte – hier Kanten – vorhanden sind. Bei der Bildung von virtuellen Linien spielen

große Elemente eine dominantere Rolle als kleine. Rechteckige Elemente werden stärker bewertet als Elemente ohne festen Umriß (z.B. Führungstexte). Für eine harmonische Gestaltung ist es wichtig, daß ein Dialogfenster eine möglichst geringe Anzahl von virtuellen Linien enthält. Auch die waagrechten virtuellen Linien müssen berücksichtigt werden. Die Erfahrung hat allerdings gezeigt, daß bei den waagrechten Linien weniger Fehler gemacht werden. Um die virtuellen Linien zu minimieren, sollten die Textfelder jedoch nicht willkürlich verlängert werden. Der fachliche Verwendungszweck der Elemente sollte immer Vorrang haben.

Im linken Fenster der Abb. 5.7-4 wurden die virtuellen Linien minimiert. Es wirkt dadurch harmonischer als das rechte Fenster, in dem für die Textfelder willkürlich unterschiedliche Längen gewählt wurden.

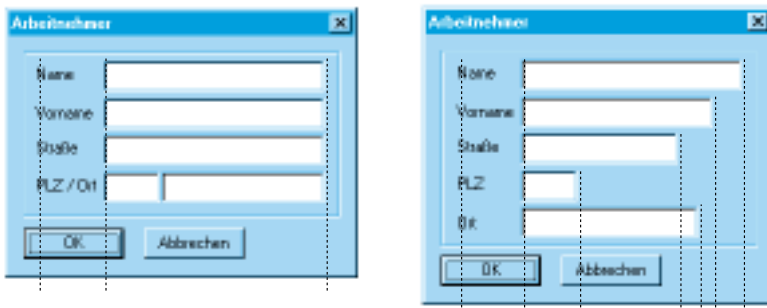


Abb. 5.7-4: Fenster mit wenigen und vielen virtuellen Linien



**Erfassungsfenster** Das Erfassungsfenster bezieht sich auf ein einzelnes Objekt einer Klasse. Jedes Attribut der Klasse wird auf ein →Interaktionselement des Fensters abgebildet. Das Erfassungsfenster dient zum Erfassen und Ändern von Objekten und zum Erstellen und Entfernen von Verbindungen zu anderen Objekten.

**Interaktionselement (control)** Ein Interaktionselement dient zur Ein-

und/oder zur Ausgabe von Informationen. Das sind beispielsweise Textfelder, Schaltflächen und Listenfelder.

**Listenfenster** Das Listenfenster zeigt alle Objekte der Klasse an. Im allgemeinen enthält es von einem Objekt nur dessen wichtigste Attribute.

**Steuerelement (control)**

→ Interaktionselement



Ein OOA-Modell kann systematisch in eine Dialogstruktur abgebildet werden. Für jede Klasse des Analysemodells werden ein **Erfassungsfenster** und ein **Listenfenster** erstellt. Assoziationen zwischen den Klassen werden mittels eines Auswahl Fensters realisiert. Jedes GUI-System verfügt über eine Reihe von **Interaktionselementen**. Bei der Gestaltung der Fenster müssen nicht nur geeignete Interaktionselemente ausgewählt werden, sondern sie müssen auch geeignet gruppiert und wichtige Informationen hervorgehoben werden. Außerdem ist die harmonische Gestaltung der Fenster von großer Bedeutung.

## LE 10 Aufgaben

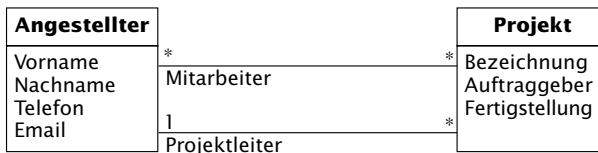
Aufgabe  
5–10 Minuten

- 1 Lernziel:** Geeignete Interaktionselemente auswählen können.
- Geben Sie an, welche Interaktionselemente in den folgenden Situationen zu wählen sind.
- Ein Fachhochschulstudent kann unterschiedliche Voraussetzungen besitzen: Abitur, Fachabitur, abgeschlossene Ausbildung. Alle gültigen Voraussetzungen sind fest definiert. Dabei ist auch eine Kombination verschiedener Voraussetzungen möglich.
  - Bei der Angabe der Adresse für einen weltweiten Versand ist das Land einzugeben. Um eine einheitliche Schreibweise zu erreichen, sind alle Ländernamen vorgegeben.
  - Bei der Anrede in Briefen sollen für die Anrede alternativ *Herr*, *Frau* oder *Firma* gewählt werden.
  - Der Benutzer wählt die Größe, in der Daten ausgedruckt werden können. Die Angabe kann kontinuierlich in Prozenten erfolgen.

Aufgabe  
10–15 Minuten

- 2 Lernziel:** Fenster mit Interaktionselementen gestalten können.
- Für das Klassendiagramm der Abb. LE10-A2 ist für die Klasse Projekt das Fenster zum Erfassen eines neuen Projekts möglichst optimal zu gestalten. Dabei sind auch alle Verbindungen zu den Angestellten zu berücksichtigen. Begründen Sie Ihre Gestaltung.

Abb. LE10-A2:  
Klassendiagramm  
zur Verwaltung  
von Projekten und  
Angestellten



Aufgabe  
15–20 Minuten

- 3 Lernziel:** Aus einem Klassendiagramm eine Dialogstruktur ableiten können.
- Aus dem Klassendiagramm der Abb. LE10-A2 ist die komplette Dialogstruktur abzuleiten und als Zustandsdiagramm zu modellieren.

## 6 Konzepte und Notation des objektorientierten Entwurfs (Teil 1)



- Erklären können, was eine generische Klasse ist.
- Erklären können, wie eine *Container*-Klasse verwendet wird.
- Sichtbarkeit für Attribute und Operationen erklären können.
- Erklären können, wie die Notation von Attributen, Operationen und Assoziationen im Entwurf erweitert wird.
- Erklären können, wie die Konzepte Objekt, Klasse, Attribut, Operation und Assoziation in C++ und Java umgesetzt werden.
- Objektverwaltung im Entwurf realisieren können.
- Assoziationen in den Entwurf transformieren können.
- Generische Klassen entwerfen können.
- Signaturen spezifizieren können.

verstehen

anwenden



- Die objektorientierten Konzepte der Analyse und die UML-Notation aus Kapitel 2 müssen bekannt sein.
- In diesem Kapitel werden grundlegende Programmierkenntnisse vorausgesetzt.
- Basiswissen in einer objektorientierten Programmiersprache erleichtert das Verständnis.

- i 6.1 Objekt/Klasse 228
- 6.2 Attribut 235
- 6.3 Operation 238
- 6.4 Assoziation 244

## 6 Konzepte und Notation des objektorientierten Entwurfs

Definition In diesem Kapitel werden die objektorientierten Konzepte der Analyse um die Konzepte für den objektorientierten Entwurf ergänzt und die UML-Notation (siehe /UML 97/ und /Booch et al. 98/) für deren Beschreibung eingeführt. Aktuelle Informationen zur vollständigen UML sind unter der angegebenen Internet-Adresse zu finden.

[www.rational.com/uml](http://www.rational.com/uml)



Das OOD-Modell soll ein Spiegelbild des Programms sein. Daher müssen alle Namen des OOD-Modells der Syntax der jeweiligen Programmiersprache entsprechen. Um die Durchgängigkeit zu den objektorientierten Programmiersprachen zu zeigen und um Programmierern das Verstehen der Entwurfskonzepte zu erleichtern, werden alle objektorientierten Konzepte auf C++ /Stroustrup 98/ und Java /Balzert 99/ abgebildet.

### 6.1 Objekt/Klasse

Klassenname Die Notation der Klasse gilt auch für den Entwurf. Lediglich der Name der Klasse muß nun der Syntax der verwendeten Programmiersprache folgen. Klassennamen beginnen immer mit einem Großbuchstaben.

deutsch vs. englisch In der Systemanalyse verwende ich in diesem Buch konsequent die deutsche Sprache und lege großen Wert auf die jeweilige Fachterminologie. In der Entwurfsphase und in der Implementierung ist es dagegen üblich, die englische Sprache zu verwenden, einerseits weil sie kürzere Bezeichnungen ermöglicht und andererseits, weil sie bei Klassenbibliotheken durchgängig verwendet wird.

Stereotyp Der Stereotyp kann sinnvoll verwendet werden, um die Zugehörigkeit der Klasse zu einer bestimmten Entwurfskomponente zu zeigen.

Beispiel Alle Klassen, die zur Datenverwaltung gehören, werden mit «DB» (*data base*) und alle Klassen zur Realisierung der Benutzungsoberfläche mit «UI» (*user interface*) gekennzeichnet.

Eine wesentliche konzeptionelle Erweiterung bieten die generischen Klassen und die *Container*-Klassen.

#### Generische Klasse

Definition Eine **generische Klasse** (*parameterized class, template*) ist eine Beschreibung einer Klasse mit einem oder mehreren formalen Parametern. Sie definiert daher eine Familie von Klassen. Ein Parameter besteht aus dem Namen und dem Typ. Der Typ entfällt, wenn der Name bereits einen Typ beschreibt. Die Parameterliste darf nicht



leer sein. Mehrere Parameter in der Liste werden durch Kommata getrennt. Damit eine generische Klasse benutzt werden kann, müssen deren formale Parameter an aktuelle Parameter gebunden werden.

Es wird eine generische Klasse `Queue` deklariert (Abb. 6.1-1), welche die üblichen Operationen `insert()` und `delete()` besitzt. Welche und wie viele Elemente die `Queue` verwalten soll, wird (noch) nicht bestimmt. Der Parameter `Element` beschreibt einen Typ. Daher sind für diesen Parameter keine weiteren Angaben notwendig. Der Parameter `n` vom Typ `int` gibt die maximale Größe der `Queue` an. Diese generische Klasse bildet die Vorlage für die »normalen« Klassen `Queue<int, 100>`, in der maximal 100 `int`-Werte gespeichert werden können und `FloatQueue`, die maximal 20 `float`-Werte enthalten kann. Von diesen beiden Klassen können dann entsprechende Objekte erzeugt werden. Wie Abb. 6.1-1 zeigt, kann das Binden einer Klasse an eine generische Klasse entweder durch Eintragen der aktuellen Parameter bei der Klasse oder mit Hilfe des Stereotypen »bind« erfolgen.

Beispiel

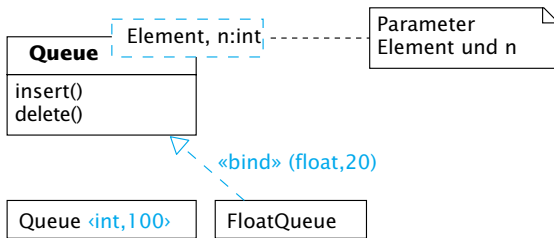


Abb. 6.1-1:  
Generische Klasse  
`Queue`

Eine generische Klasse kann nicht Oberklasse einer »normalen« Klasse sein. Umgekehrt kann sie jedoch die Unterklasse einer »normalen« Klasse *Super* sein. Das bedeutet, daß alle »normalen« Klassen, die durch *binding* gebildet werden, Unterklassen von *Super* sind.

Außerdem ist eine unidirektionale Assoziation von einer generischen Klasse zu einer »normalen« Klasse erlaubt (Abb. 6.1-2), d.h. die generische Klasse kennt die »normale« Klasse, jedoch nicht umgekehrt. Die unidirektionale Assoziation wird in Kapitel 6.4 eingeführt.



Kapitel 6.4

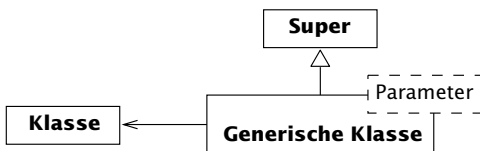


Abb. 6.1-2:  
Eigenschaften  
generischer  
Klassen

**Container-Klasse**

In der Analyse sollten Sie keine Klassen bilden, um Mengen von Objekten zu verwalten, denn jede Klasse der Analysephase besitzt inhärent eine Objektverwaltung. Im Entwurf wird die Objektverwaltung mittels *Container*-Klassen realisiert. Viele Container-Klassen können aus entsprechenden Bibliotheken genommen werden. Bei der Realisierung von Container-Klassen lassen sich besonders vorteilhaft generische Klassen verwenden.

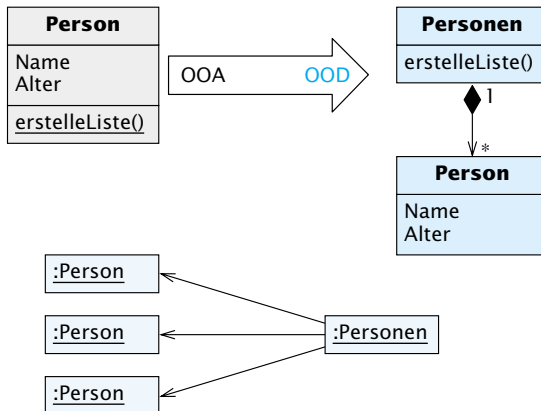
Definition Eine **Container-Klasse** ist eine Klasse, die dazu dient, eine Menge von Objekten einer anderen Klasse zu verwalten. Sie stellt Operationen bereit, um auf die verwalteten Objekte zuzugreifen. Ein Objekt der *Container*-Klasse wird als *Container* bezeichnet. Typische *Container* sind beispielsweise Felder (*arrays*) und Mengen (*sets*). *Container*-Klassen werden in Kapitel 10 für die Realisierung der Entwurfsarchitektur verwendet.

Beispiel Die Objekte der Klasse *Person* werden in einem *Container*, d.h. einem Objekt der Klasse *Personen* verwaltet (Abb. 6.1.-3). Das bedeutet, daß das *Personen*-Objekt die Objektidentitäten (OIDs) aller Objekte der Klasse *Person* kennt. Wird eine neue *Person* erzeugt, dann wird die entsprechende OID in das *Personen*-Objekt eingefügt. Die Klassenoperation `erstelleListe()` wird zu einer Objektoperation der Klasse *Personen*. Sie liest mit `getLink()` die Objektidentität jeder *Person* und holt sich dann vom jeweiligen Objekt der Klasse *Person* die gewünschten Daten (Abb. 6.1-4). Die Pfeilspitzen in Abb. 6.1-3 geben die Navigationsrichtung der Assoziation an, die in Kapitel 6.4 erläutert wird.



Kapitel 6.4

Abb. 6.1-3:  
Container für die  
Verwaltung von  
Personen



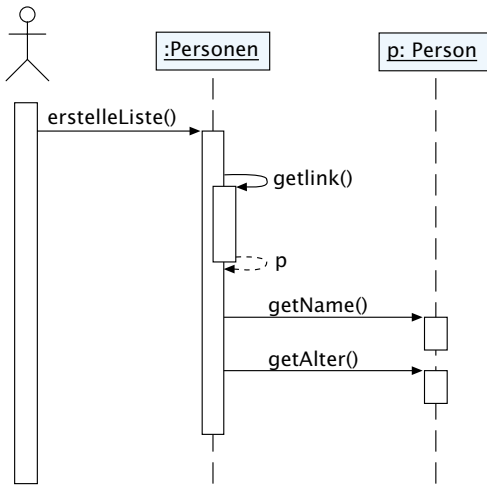


Abb. 6.1-4:  
Sequenzdiagramm  
für die Verwaltung  
von Personen im  
Container

### Schnittstelle

Eine **Schnittstelle** (*interface*) spezifiziert einen Ausschnitt aus dem Verhalten einer Klasse. Eine Schnittstelle besteht nur aus den Signaturen von Operationen, d.h. sie besitzt keine Implementierung, keine Attribute, Zustände oder Assoziationen. Eine Schnittstelle ist äquivalent zu einer abstrakten Klasse, die ausschließlich abstrakte Operationen besitzt. Die Realisierung bzw. Implementierung einer Schnittstelle durch eine Klasse wird durch den gestrichelten Vererbungspfeil gekennzeichnet.

Definition

Die Schnittstelle `ClassInfo` wird mit dem Stereotypen «interface» gekennzeichnet (Abb. 6.1-5). Da das Attributfeld einer Schnittstelle immer leer ist, kann es entfallen. Die Klassen `MyClass` und `OtherClass` implementieren diese Schnittstelle und die darin enthaltenen Operationen (ausgedrückt durch die gestrichelten »Vererbungspfeile«). Daher können alle Objekte von `MyClass` und `OtherClass` mit `getClassName()` ihren Klassennamen zurückliefern. In diesem Fall dient die Schnittstelle der Vereinheitlichung.

Beispiel

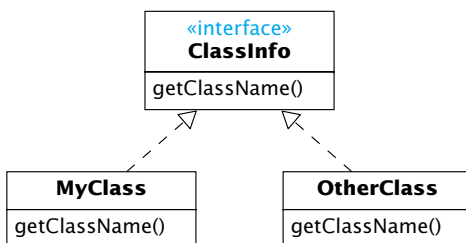


Abb. 6.1-5:  
Schnittstelle  
(*interface*)

Objekt/Klasse in  
C++**Klasse**

- Die Klasse wird durch das *class*-Konzept realisiert.
- Es ist üblich, die Spezifikation der Klasse in eine *Header*-Datei (.h) und die Implementierung in eine Quelldatei (.cpp) zu schreiben. Eine Ausnahme bilden *templates*. Hier muß der vollständige Quellcode in der *Header*-Datei zugänglich sein.
- Jede Klasse ist ein (abstrakter) Datentyp. Die Umkehrung – ein Datentyp ist eine Klasse – gilt nicht, weil die primitiven Typen nicht als Klasse implementiert sind.

```
//zaehler.h
class Zaehler
{private:
    unsigned int Zaehlerstand;
public:
    void inkrementieren();
    void initialisieren();
    int getZaehlerstand() const;
};
//zaehler.cpp
void Zaehler::inkrementieren ()
{ ... }
void Zaehler::initialisieren ()
{ ... }
int Zaehler::getZaehlerstand () const
{ ... }
```

**Objekt**

- Objekte werden wie normale Variablen behandelt, die entweder (statisch) deklariert (*stack*) oder dynamisch erzeugt (*heap*) werden können.
- Der dynamisch erzeugte Speicherplatz von Objekten muß explizit freigegeben werden.
- Jedes Objekt wird über seine Adresse eindeutig identifiziert.

```
Zaehler einZaehler;           //stack-Variablen;
Zaehler* pZaehler;           //Zeiger-Variablen
pZaehler = new Zaehler;      //heap-Variablen erzeugen, auf die
                             //pZaehler verweist
```

**Abstrakte Klasse**

- Abstrakte Klassen können in C++ nicht speziell gekennzeichnet werden.
- Besitzt eine Klasse jedoch mindestens eine abstrakte Operation (*pure virtual member function*), dann ist sie abstrakt, d.h. der Compiler meldet einen Fehler, wenn ein Objekt der Klasse erzeugt werden soll.

```
class Abstract
{public:
    void operation() = 0;
    ...
};
```

## Generische Klasse

- Generische Klassen können durch *templates* realisiert werden.

```
template <class ElementType, int max>
class Queue
{protected:
    struct ListElement
    { ElementType value;
      ListElement* next;
    }; ...
public:
    void insert (ElementType element); ...
};
typedef Queue<int, 100> IntegerQueue;
//Besetzung mit dem aktuellen Elementtyp »int« und der Größe
//100
IntegerQueue aQueue;
aQueue.insert(5);
```

## Schnittstelle

Das Konzept der Schnittstelle (*interface*) existiert in C++ nicht, kann jedoch mit dem Klassenkonzept nachgebildet werden.

## Klasse

- Die Klasse wird durch das *class*-Konzept realisiert.
- In Java wird *nicht* zwischen der Spezifikation und der Implementierung einer Klasse unterschieden.
- Jede Klasse ist ein (abstrakter) Datentyp. Die Umkehrung – ein Datentyp ist eine Klasse – gilt nicht, weil die primitiven Typen nicht als Klasse implementiert sind.

```
class Zaehler
{private int Zaehlerstand;
public void inkrementieren()
{ ... }
public void initialisieren()
{ ... }
public int getZaehlerstand()
{ ... }
}
```

## Sichtbarkeit der Klasse

Eine Klasse kann normalerweise nur innerhalb ihres Pakets benutzt werden. Wird sie als `public` deklariert, dann kann sie überall benutzt werden.

```
public class Zaehler {}
```

## Objekt

- Objekte können in Java grundsätzlich nur dynamisch – also als *heap*-Variable – erzeugt werden. Daher ist keine Unterscheidung bei der Namenswahl notwendig.

**Objekt/Klasse in  
Java**

- Wenn ein Objekt nicht mehr referenziert wird, dann wird dessen Speicherplatz durch den *garbage collector* automatisch freigegeben.

```
Zaehler einZaehler;           //Objektvariable einZaehler
einZaehler = new Zaehler(); //Objekt der Klasse Zaehler erzeugen,
                             //auf das einZaehler verweist
```

### Abstrakte Klasse

- Abstrakte Klassen werden mit `abstract` gekennzeichnet.
- Sobald eine Klasse eine abstrakte Operation besitzt, muß sie als `abstract` deklariert werden.

```
abstract class Abstract1
{ public void operation1() {...}
  public void operation2() {...}
}
abstract class Abstract2
{ public abstract void operation1();
  ...
}
```

### Generische Klasse

Generische Klassen können in Java *nicht* realisiert werden. Dennoch ist es nicht notwendig, für jeden Typ eine eigene *Container*-Klasse zu entwickeln, weil alle Klassen implizit von der Klasse `Object` abgeleitet sind. Dann kann beispielsweise eine Klasse `Queue` gebildet werden, die Objekte der Klasse `Object` oder ihrer Unterklassen verwalten kann. Um den Typ der Objekte nach dem Zugriff über den *Container* wieder zu restaurieren, sind allerdings Typwandlungen (*type casts*) erforderlich.

### Schnittstelle

- Eine Schnittstelle kann in Java aus Konstanten und abstrakten Operationen bestehen.
- Schnittstellen werden mit dem Schlüsselwort `interface` deklariert und von Klassen mit dem Schlüsselwort `implements` benutzt.

```
interface ClassInfo
{ public abstract String getClassname();
}
class MyClass implements ClassInfo
{ public String getClassname()
  { return "MyClass";
  }
  ...
}
```

### Besonderes

- In Java können Klassen mit `final` gekennzeichnet werden. Von einer solchen Klasse dürfen keine Unterklassen abgeleitet werden. Eine Klasse darf nicht gleichzeitig als `abstract` und als `final` deklariert sein.

## 6.2 Attribut

Der Attributname muß an die Konventionen der Programmiersprache angepaßt werden. Im Gegensatz zur UML, die fordert, daß ein Attributname immer mit einem Kleinbuchstaben beginnt, verwende ich sowohl Großbuchstaben (bei deutschen Substantiven) als auch Kleinbuchstaben.

Attributname

In der Systemanalyse haben wir alle Attribute außerhalb der Klasse verborgen, jedoch für ihre Unterklassen sichtbar gemacht. Diesen Modus bezeichnet man als *protected*. Im Entwurf müssen wir die **Sichtbarkeit** (*visibility*) nun feiner differenzieren zwischen

Sichtbarkeit

- *public*: sichtbar für alle anderen Klassen,
- *protected*: sichtbar innerhalb der Klasse und in deren Unterklassen,
- *private*: sichtbar nur innerhalb der Klasse.

Abb. 6.2-1 zeigt die Notation der UML.

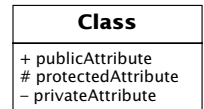


Abb. 6.2-1:  
Notation für die  
Sichtbarkeit von  
Attributen

Attribute sollten prinzipiell als *protected* oder *private* vereinbart werden.

Bei einem *protected*-Attribut sieht jedes Objekt alle Attribute seiner Oberklassen und kann direkt darauf zugreifen. Sind die Attribute *private*, dann sehen Objekte die Attribute ihrer Oberklassen nicht, sondern dürfen nur über entsprechende Operationen zugreifen. Der Vorteil liegt darin, daß Veränderungen der Attribute sich nicht auf die Unterklassen auswirken. Dadurch wird die Realisierung der Unterklasse unabhängig von der Darstellung der Attribute der Oberklasse. Dem steht der Nachteil zusätzlicher Lese-/Schreiboperationen gegenüber. Bei einem *public*-Attribut wird das Geheimnisprinzip verletzt.

Die Notation eines Attributs in der UML lautet:

Notation

Sichtbarkeit Attribut: Typ = Anfangswert {Merkmalsliste}

Die verwendeten Typen müssen in der jeweiligen Programmiersprache spezifiziert werden können. Viele der in Kapitel 2.3 aufgeführten Standardtypen und zahlreiche elementare Klassen können aus entsprechenden Klassenbibliotheken übernommen werden. In der optionalen Merkmalsliste können die Eigenschaften des Attributs angegeben werden, z.B. {mandatory, frozen}.

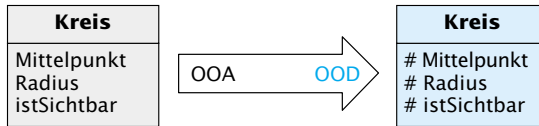
Kapitel 2.3

Um kompakte Klassendiagramme zu erhalten, trage ich bei »echten« OOD-Klassendiagrammen außer dem Attributnamen nur die Sichtbarkeit ein. Der Typ und der optionale Anfangswert werden mit allen anderen Angaben zur Spezifikation des Attributs separat dokumentiert.

Die Abb. 6.2-2 zeigt, wie die Klasse Kreis aus dem OOA-Modell in eine Klasse Kreis des OOD-Modells transformiert wird. In diesem Buch wird in OOD-Klassendiagrammen auf die Angabe der Sichtbarkeit verzichtet, wenn sie für die jeweilige Problemstellung nicht relevant ist.

## LE 11 6 Konzepte und Notation für OOD

Abb. 6.2-2:  
Abbildung der  
Attribute in den  
Entwurf



Klassenattribut Ein Klassenattribut kann auf zwei Arten im OOD-Modell realisiert werden:

- ebenfalls als Klassenattribut,
- als Objektattribut einer separaten Klasse. Diese Klasse besitzt dann nur ein einziges Objekt mit dem Wert des Klassenattributs.

abgeleitetes Attribut Abgeleitete Attribute sind ein wichtiges Entwurfskonzept. Sie können im Programm entweder als Attribut – mit entsprechender Konsistenzprüfung – oder durch eine Operation realisiert werden, die stets den aktuellen Wert ermittelt.

Abstraktion, Verkapselung, Geheimnisprinzip Die Begriffe Abstraktion, Verkapselung und Geheimnisprinzip werden oft verwechselt, weil sie sehr eng miteinander in Beziehung stehen. Wir wollen sie daher näher betrachten und gegeneinander abgrenzen /Berard 93/.

- **Abstraktion** (*abstraction*), als ein Prozeß betrachtet, bezeichnet die Vorgehensweise, die wesentlichen Aspekte über etwas zu ermitteln und die unwesentlichen Details zu ignorieren. Auch ein Modell oder ein bestimmter Blickwinkel wird als Abstraktion bezeichnet.
- Die Einhaltung des **Geheimnisprinzips** (*information hiding*) bedeutet, daß der Zustand eines Objekts und die Implementierung der Operationen außerhalb der Klasse nicht sichtbar sind.
- Die **Verkapselung** (*encapsulation*) sagt aus, daß zusammengehörende Attribute und Operationen, in einer Einheit – der Klasse – verkapselt sind. Im Unterschied zum Geheimnisprinzip können die Attribute und die Realisierung der Operationen durchaus nach außen sichtbar sein. Beispielsweise erlaubt C++ mit seinen verschiedenen Sichtbarkeiten *public*, *protected* und *private* die Verkapselung ohne und mit Einhaltung des Geheimnisprinzips.

In der Analysephase gehen wir davon aus, daß alle Attribute generell für andere Objekte unsichtbar sind. Wir können daher Verkapselung und Geheimnisprinzip in der Analyse gleichsetzen. Eine Unterscheidung beider Konzepte ist jedoch für Entwurf und Implementierung sinnvoll und notwendig, weil wir hier die Angabe der Sichtbarkeit berücksichtigen.

### Attribute in C++

Attribute werden in C++ als *Member-Variablen* bezeichnet.

#### Sichtbarkeit

Die Sichtbarkeit ist in C++ genauso definiert wie in der UML, d.h.

- *public*: sichtbar für alle,
- *protected*: sichtbar innerhalb der Klasse und ihrer Unterklassen,



- *private*: sichtbar innerhalb der Klasse, Die Voreinstellung, d.h. wenn keine Sichtbarkeit angegeben wird, ist *private*.

### Initialisierung der Attribute

Attribute werden am elegantesten mittels der Konstruktoren (siehe Kapitel 6.3) initialisiert. Im Vergleich zu einer Operation `init()` ist dadurch sichergestellt, daß jedes Objekt definierte Werte besitzt und alle *default*-Werte gesetzt sind. Ein Konstruktor wird aktiviert, wenn ein entsprechendes Objekt erzeugt wird.

Kapitel 6.3

```
Kreis::Kreis ()
{ istSichtbar = true;
  ...
}
Kreis* pKreis = new Kreis;
```

### Klassenattribute

- Klassenattribute werden mittels `static` gekennzeichnet.
- Da die Deklaration der Klasse selbst keinen Speicherplatz belegt, müssen alle Klassenattribute explizit deklariert und ggf. initialisiert werden.

```
class Kreis
{protected:
  bool istSichtbar;
  static unsigned int Anzahl; //Klassenattribut
  ....
};
unsigned int Kreis::Anzahl = 0; //Deklaration und
//Initialisierung
```

### Typen

C++ bietet eine Reihe von elementaren Datentypen (z.B. *int*, *float*) an. Viele der anderen – in Kapitel 2.3 aufgeführten Standardtypen – werden in C++ als (elementare) Klassen realisiert.

Kapitel 2.3

Attribute werden in Java als Elemente (*fields*) bezeichnet.

### Sichtbarkeit

In Java ist die Sichtbarkeit geringfügig anders definiert:

- *public*:  
Diese Attribute sind für alle Klassen sichtbar.
- *protected*:  
Innerhalb des gleichen Pakets sind diese Attribute für alle Klassen sichtbar. Außerhalb des Pakets sind sie nur für die Unterklassen sichtbar. Natürlich sind sie auch in der Klasse selbst sichtbar.
- *private*:  
Diese Attribute sind nur innerhalb der Klasse sichtbar.

---

**Attribute in Java**

Ohne eine explizite Angabe der Sichtbarkeit (d.h. Voreinstellung) ist ein Attribut grundsätzlich innerhalb des gesamten Pakets sichtbar, in dem die Klasse definiert ist. Außerhalb des Pakets ist es für alle Klassen unsichtbar.

### Initialisierung der Attribute

Die Initialisierung der Attribute ist über Konstruktoren möglich. Ein Konstruktor wird aktiviert, wenn ein entsprechendes Objekt deklariert wird.

```
public Kreis ()
{ istSichtbar = false;
}
Kreis einKreis;
einKreis = new Kreis();
```

### Klassenattribute

- Klassenattribute werden als `static` gekennzeichnet.
- Sie erhalten durch eine statische Initialisierung den Anfangswert zugewiesen. Eine statische Initialisierung wird genau einmal durchgeführt, nämlich dann, wenn die Klasse vom Laufzeitsystem geladen wird.

```
class Kreis
{ static int Anzahl = 0;
  ...
}
```

### Typen

Java bietet eine Reihe von einfachen Datentypen (z.B. `int`, `float`) an. Der Typ `String` ist im Gegensatz zu C++ standardmäßig als (elementare) Klasse definiert.

## 6.3 Operation

**Operationsname** Der Operationsname muß an die Konventionen der Programmiersprache angepaßt werden. Wie in der UML gefordert, beginne ich alle Operationsnamen mit einem Kleinbuchstaben.

**Sichtbarkeit** Analog zu den Attributen sind in der UML folgende Sichtbarkeiten definiert (Abb. 6.3-1):

Class
+ publicOperation()
# protectedOperation()
- privateOperation()

Abb. 6.3-1:  
Notation der  
Sichtbarkeiten von  
Operationen

- Eine `private` Operation (*private*) kann nur von Operationen derselben Klasse aus aufgerufen werden. Sie ist für alle anderen Klassen bzw. deren Objekte unsichtbar.
- Eine geschützte Operation (*protected*) kann von Operationen der eigenen Klasse und ihren Unterklassen aus aufgerufen werden.
- Eine öffentliche Operation (*public*) kann von Operationen aller Klassen bzw. deren Objekten aufgerufen werden.

Im Entwurf ist für jede Operation deren vollständige Signatur anzugeben. Die **Signatur** (*signature*) einer Operation besteht aus dem

Namen der Operation, den Namen und Typen aller Parameter, und dem Ergebnistyp der Operation. Die Menge aller Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die **Schnittstelle** (*interface*) der Klasse bzw. des Objekts.

Normalerweise geht die Entwicklung von Entwurfsmodell und Programmierung in C++ oder Java Hand in Hand. Außerdem führt die objektorientierte Modellierung zu kompakten Operationen. Daher erübrigt es sich meistens, die Wirkung einer Operation im Entwurfsmodell separat zu dokumentieren, sondern sie wird sofort in der entsprechenden Programmiersprache implementiert und durch umgangssprachliche Kommentare ergänzt. Bei Bedarf kann auch eine Beschreibung mittels Vor- und Nachbedingungen erfolgen. Die **Vorbedingung** (*precondition*) beschreibt, welche Bedingungen vor dem Aktivieren einer Operation erfüllt sein müssen. Die **Nachbedingung** (*postcondition*) beschreibt die Änderung, die durch die Operation bewirkt wird. Für die Implementierung einer Operation wird auch der Begriff »Methode« (*method*) verwendet.

Implementierung  
einer Operation

Eine **abstrakte Operation** besteht nur aus der Signatur, d.h. dem Namen der Operation, den Namen und Typen aller Parameter und dem Ergebnistyp. Sie besitzt keine Implementierung. Abstrakte Operationen werden verwendet, um für Unterklassen eine gemeinsame Schnittstelle zu definieren. In der UML werden abstrakte Operationen kursiv eingetragen oder – z.B. bei handschriftlicher Modellierung – mittels {abstract} gekennzeichnet.

abstrakte  
Operation

In der Abb. 6.3-2 definiert die abstrakte Operation *zeichnen()*, daß jedes Grafikobjekt, in diesem Fall also Kreise und Rechtecke, mit dem gleichen Operationsaufruf gezeichnet werden kann, während sich die Implementierungen beider Operationen erheblich unterscheiden können.

Beispiel

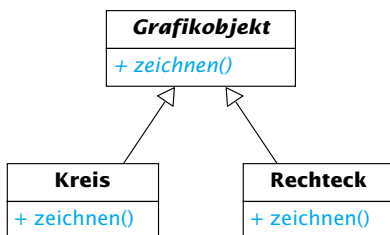


Abb. 6.3-2:  
Abstrakte  
Operation definiert  
einheitliche  
Schnittstelle

Im Entwurf fordert die UML für jede Operation außer der Sichtbarkeit und dem Namen folgende Angaben:

Notation

- Die Parameterliste (*parameter list*) enthält formale Parameter, die jeweils durch Komma getrennt sind. Für jeden Parameter gilt die Syntax:

Art Name : Typ = Anfangswert oder

Art Name : Typ.

## LE 11 6 Konzepte und Notation für OOD

Die Art gibt an, ob es sich um einen Eingabe- (*in*), einen Ausgabe- (*out*) oder einen transienten Parameter (*inout*) handelt.

- Der Ergebnistyp (*return type expression*) beschreibt den Typ des Ergebnisparameters. Wenn der Ergebnistyp fehlt, dann gibt die Operation keinen Wert zurück.
- In einer optionalen Merkmalsliste können spezielle Eigenschaften der Operation angegeben werden.

Damit ergibt sich die folgende – vollständige – Syntax einer Operation in UML:

Sichtbarkeit Operation (Parameterliste) : Ergebnistyp {Merkmalsliste}

Für jeden Parameter der Parameterliste gilt:

[in | out | inout] Name: Typ = Anfangswert

»Name« steht für den formalen Parameter. Mehrere Parameter in der Liste werden durch Kommata getrennt.

Die UML erlaubt es, auf die Angabe der Parameterliste und des Ergebnistyps zu verzichten. Dadurch ergeben sich bei »echten« OOD-Klassendiagrammen kompaktere Diagramme.

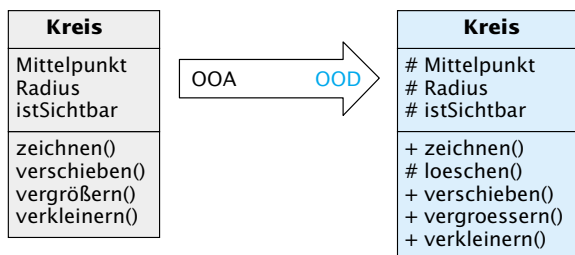
Außer in der angegebenen Form erlaubt die UML, die Signatur einer Operation in der Notation einer Programmiersprache, wie z.B. C++ oder Java, zu spezifizieren.

Abb. 6.3-3 zeigt, wie die Klasse *Kreis* aus dem OOA-Modell in eine Klasse *Kreis* des OOD-Modells transformiert wird. Während im Analysemodell nur diejenigen Operationen eingetragen werden, die für das Verständnis des Fachkonzepts wichtig sind, enthält das Entwurfsmodell – mit Ausnahme der Verwaltungsoperationen (Kapitel 2.4) – alle Operationen, die im C++ bzw. Java-Programm enthalten sind. Die Abfrage-Operationen (*select*), die in der Analysephase im allgemeinen nicht alle aufgeführt werden, sind jedoch im Entwurf vollständig einzutragen.

Kapitel 2.4



Abb. 6.3-3:  
Abbildung von  
Operationen  
in den Entwurf



*overloading*

In der Analyse wird gefordert, daß der Operationsname innerhalb einer Klasse eindeutig ist. In Entwurf und Programmierung darf der gleiche Operationsname innerhalb einer Klasse mehrfach verwendet werden, wobei sich die entsprechenden Operationen in ihrer Parameterliste unterscheiden müssen. Man spricht vom Überladen (*overloading*) der Operation.

Operationen werden in C++ durch *Member*-Funktionen realisiert.

### Sichtbarkeit

Die Sichtbarkeit ist in C++ genauso definiert wie in der UML, d.h.

- *public*: sichtbar für alle,
- *protected*: sichtbar innerhalb der Klasse und in ihren Unterklassen,
- *private*: sichtbar innerhalb der Klasse.

Die Voreinstellung, d.h. wenn keine Sichtbarkeit angegeben wird, ist *private*.

```
class Kreis
{protected:
    Punkt Mittelpunkt;
    unsigned int Radius;
    bool istSichtbar;
    static unsigned int Anzahl;           //Klassenattribut
protected:
    void Loeschen();
public:
    Kreis ();                             //Konstruktor 1
    Kreis (int x, int y);                 //Konstruktor 2
    void zeichnen();
    void verschieben (Punkt neu);
    void vergroessern (int faktor);
    void verkleinern (int faktor);
    static unsigned int getAnzahl ();    //Klassenoperation
};
```

Beispiel

**Operationen in C++**

### Parameterkonzept

Eingabeparameter können in C++ als *call by value* oder *call by reference* übergeben werden. Ausgabeparameter können entweder als *call by reference* in der Parameterliste oder als Ergebnistyp realisiert werden.

### Abstrakte Operation

Abstrakte Operationen werden in C++ als *pure virtual member functions* (nicht zu verwechseln mit den *virtual member functions*!) realisiert. Eine abstrakte Operation wird durch folgende Notation realisiert:

```
void zeichnen() = 0;
```

### frozen-Operation

Operationen, die keine Veränderungen der Attributwerte herbeiführen, sondern nur zum Lesen dienen, werden als *const* deklariert.

```
bool getIstSichtbar() const;
```

### Klassenoperation

Klassenoperationen (*static member functions*) werden analog zu den Attributen mit `static` gekennzeichnet. Beim Aufruf einer Klassenoperation wird der Name der Klasse angegeben.

```
static unsigned int getAnzahl (); //Klassenoperation
i = Kreis::getAnzahl ();        //Aufruf Klassenoperation
```

### Konstruktoren und Destruktoren

- Die Basisoperation `new()` wird mittels Konstruktoren realisiert. Ein Konstruktor ist eine spezielle *Member-Funktion*, die den gleichen Namen wie die Klasse hat. Sie definiert, wie ein Objekt der Klasse initialisiert wird. Bei dynamisch erzeugten Objekten kann der benötigte Speicherplatz im Konstruktor angefordert werden.
- Ein Konstruktor wird aktiviert, wenn ein entsprechendes Objekt erzeugt wird. Fehlt der Konstruktor, dann ruft C++ einen impliziten Konstruktor auf.
- Eine Klasse kann mehrere Konstruktoren enthalten, die sich jedoch in ihrer Parameterschnittstelle unterscheiden müssen (*overloading*).
- Die Basisoperation `delete()` wird durch den Destruktor realisiert.
- Jede Klasse besitzt nur einen Destruktor. Wurde für ein Objekt dynamisch Speicherplatz erzeugt, dann muß dieser spätestens im Destruktor wieder freigegeben werden.

```
class Kreis
{protected:
    bool istSichtbar;
    ...
public:
    Kreis()                //Konstruktor 1
    {istSichtbar = false;
     ...
    }
    Kreis (int x, int y)   //Konstruktor 2
    {istSichtbar = false;
     Mittelpunkt.setX(x);
     Mittelpunkt.setY(y);
    }
    ...
};
Kreis einKreis;
```

Operationen werden in Java als Methoden (*methods*) bezeichnet.

### Sichtbarkeit

In Java ist die Sichtbarkeit geringfügig anders als in C++ definiert:

- *public*: Sichtbar für alle.
- *protected*:  
Innerhalb des gleichen Pakets sind diese Operationen für alle Klassen sichtbar. Außerhalb des Pakets sind sie nur für die Unterklassen sichtbar. Natürlich sind sie auch in der Klasse selbst sichtbar.
- *private*: Sichtbar nur innerhalb der Klasse.

Ohne eine explizite Angabe der Sichtbarkeit (d.h. Voreinstellung) ist eine Operation innerhalb des gesamten Pakets sichtbar, in dem die Klasse definiert ist. Außerhalb des Pakets ist sie für alle Klassen unsichtbar.

```
class Kreis
{
    protected Punkt Mittelpunkt;
    protected int Radius;
    protected boolean istSichtbar;
    protected static int Anzahl;           //Klassenattribut

    public Kreis(){...}                   //Konstruktor 1
    public Kreis(int x, int y) {...}     //Konstruktor 2
    public void zeichnen() {...}
    public void verschieben (Punkt neu) {...}
    public void vergroessern (int faktor) {...}
    public void verkleinern (int faktor) {...}
    protected void loeschen() {...}
    public static int getAnzahl () {...} //Klassenoperation
}

```

Beispiel

### Parameterkonzept

In Java werden Parameter vom primitiven Typ als *call by value* übergeben. Das bedeutet, daß deren Ausgabe nur über den Ergebnistyp erfolgt. Für alle Objekttypen wird *call by reference* verwendet.

### Abstrakte Operation

Abstrakte Operationen werden mit `abstract` gekennzeichnet. Sie dürfen nur in Klassen deklariert werden, die ebenfalls `abstract` sind und müssen in einer Unterklasse implementiert werden. Abstrakte Operationen dürfen nicht die Sichtbarkeit *private* besitzen.

```
abstract void zeichnen();
```

**Operationen in Java**

**Klassenoperation**

Klassenoperationen werden mit `static` gekennzeichnet. Innerhalb einer Klassenoperation darf nur auf Elemente zugegriffen werden, die ebenfalls `static` sind. Sie gelten implizit als `final` und können nicht überschrieben werden. Klassenoperationen können nicht gleichzeitig abstrakt sein.

```
static int getAnzahl ();
i = Kreis.getAnzahl ();
```

**Konstruktoren und Destruktoren**

- Konstruktoren sind eine elegante Möglichkeit, Objekte zu initialisieren. Fehlt der Konstruktor, dann ruft Java einen impliziten Konstruktor auf.
- Eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Parametern enthalten (*overloading*).
- Destruktoren im Sinne von C++ gibt es in Java nicht. Anstelle der expliziten Freigabe von Speicherplatz besitzt Java einen Automatismus: den *garbage collector*. Er läuft im Hintergrund und ermittelt diejenigen Objekte, die nicht mehr referenziert werden. Diese Objekte werden markiert und dann entfernt.

```
class Kreis
{ protected boolean istSichtbar;
  ...
  public Kreis() //Konstruktor 1
  { istSichtbar = false;
    ...
  }
  public Kreis (int x, int y) //Konstruktor 2
  { istSichtbar = false;
    Mittelpunkt.setX(x);
    Mittelpunkt.setY(y);
  }
  ...
}
einKreis = new Kreis();
zweiterKreis = new Kreis (10, 20);
```

**6.4 Assoziation**

Navigation

Während in der Analyse alle Assoziationen inhärent bidirektional sind, wird im Entwurf festgelegt, ob sie uni- oder bidirektional implementiert werden. Wir sprechen von der **Navigation** (*navigability*) der Assoziation. In der Abb. 6.4-1 muß im grauen Modell nur von einer Abteilung auf ihre Angestellten zugegriffen werden. Daher reicht es aus, die Assoziation in dieser Richtung zu implementieren. Im blauen Modell soll für einen Mitarbeiter zusätzlich ein Ausweis gedruckt werden. Dafür ist ein Zugriff auf die zugehörige



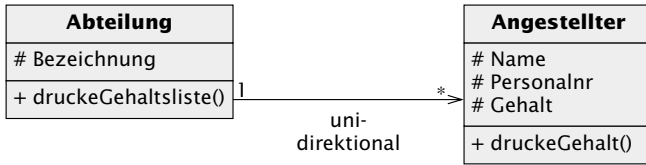
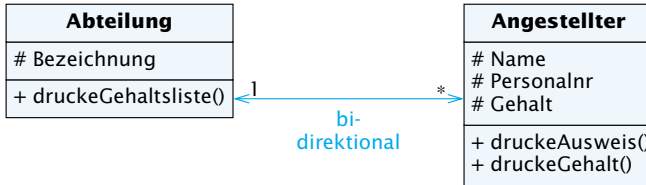


Abb. 6.4-1:  
Unidirektionale vs.  
bidirektionale  
Assoziation



Abteilung erforderlich. In diesem Fall muß die Assoziation in beiden Richtungen implementiert werden.

Die Richtung, in der die Assoziation realisiert werden muß, wird im Klassendiagramm mit einer Pfeilspitze gekennzeichnet. Eine Assoziation kann keinen, einen oder zwei Pfeile besitzen.

Notation  
Navigation

Eine der folgenden Konventionen sollte im gesamten OOD-Modell eingehalten werden:

- 1 Alle Pfeile werden eingetragen, d.h. eine Assoziation ohne Pfeile wird in diesem Fall nicht traversiert.
- 2 Soll eine Assoziation in beiden Richtungen traversiert werden, dann werden keine Pfeile eingetragen. Andernfalls wird die Richtung durch eine Pfeilspitze kenntlich gemacht. Dieser Fall ist nur dann sinnvoll, wenn alle Assoziationen des Diagramms auch traversiert werden.

Nach dem WYSIWYG-Prinzip (*what you see is what you get*) kennzeichnen wir alle Navigationsrichtungen, die im Programm realisiert werden, durch einen Pfeil. Diese Information ist für die spätere Wartung von großer Bedeutung. Eine bidirektionale Implementierung ist komplexer als eine unidirektionale Realisierung. Beispielsweise muß beim Löschen eines Objekts darauf geachtet werden, daß auch der Zeiger auf dieses Objekt im assoziierten Objekt entfernt wird.

Werden im Entwurf Objektdiagramme erstellt, so können die Pfeile zum Anzeigen der Navigationsrichtung einer Objektverbindung (*link*) auch hier eingezeichnet werden.

Objektdiagramm

Im OOD-Modell kann die Angabe der Kardinalität auf einer Seite fehlen, wenn in dieser Richtung keine Navigation stattfindet. Das bedeutet, daß sie in diesem Fall irrelevant ist.

Kardinalität

Jede Richtung einer Assoziation kann mittels Zeigern zwischen Objekten realisiert werden. Dann kennt jedes Objekt seine assoziierten Objekte. Durch die Operationen muß sichergestellt werden,

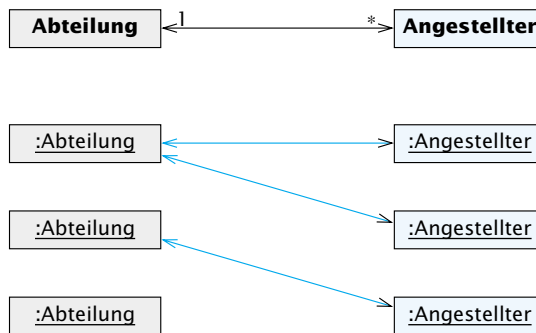
Realisierung  
mittels Zeigern

## LE 11 6 Konzepte und Notation für OOD

daß alle Verbindungen konsistent auf- und abgebaut werden. Eine Kardinalität von 0..1 oder 1 wird dabei durch einen einzelnen Zeiger realisiert. Liegt eine Kardinalität größer 1 vor, dann muß eine Menge von Zeigern gespeichert werden. Wenn keine Ordnung definiert sein muß, dann können *Container*-Klassen wie *Set*, *Bag* etc. verwendet werden.

**Beispiel** In der Abb. 6.4-2 realisieren wir die Assoziation zwischen *Abteilung* und *Angestellter* als bidirektionale Assoziation mittels Zeigern. Jeder *Angestellter* referenziert genau eine *Abteilung*. Jede *Abteilung* besitzt eine Menge von Zeigern auf *Angestellte*. Beim Erfassen eines neuen *Angestellten* muß die Verbindung zu einer *Abteilung* aufgebaut werden. Existiert die gewünschte *Abteilung* nicht, dann muß das Objekt erst erzeugt werden. Außerdem muß in der zugewiesenen *Abteilung* ein Zeiger auf den neuen *Angestellten* gespeichert werden. Beim Löschen eines *Angestellten* ist nicht nur das betreffende Objekt zu löschen, sondern in der zugehörigen *Abteilung* muß der Zeiger auf diesen *Angestellten* entfernt werden.

Abb. 6.4-2:  
Realisierung einer  
bidirektionalen  
Assoziation mittels  
Zeigern



Realisierung  
mittels Klasse

Eine weitere Möglichkeit besteht darin, die Assoziation mittels einer eigenen Klasse zu realisieren, die jedoch nicht im Klassendiagramm dargestellt wird. Dann kennen sich die assoziierten Objekte nicht mehr direkt. Das Wissen, wer mit wem verbunden ist, ist nur in dem Assoziationsobjekt (vergleiche /Rumbaugh et al. 91/) verborgen. Diese Möglichkeit ist vor allem dann einzusetzen, wenn die Assoziation nachträglich hinzugefügt werden soll und die Klassen nicht verändert werden sollen. Das gilt insbesondere bei vordefinierten Klassen aus Bibliotheken, weil hier keine Veränderung der bestehenden Klassen möglich ist.

**Beispiel** In der Abb. 6.4-3 wird die bidirektionale Assoziation zwischen *Angestellter* und *Abteilung* durch ein Assoziationsobjekt realisiert. Dazu fügen wir die Klasse *Abteilung/Angestellter* hinzu, von der nur ein einziges Objekt existiert. Dieses Objekt besitzt eine Liste von Einträgen. Jeder Eintrag besteht aus je einem Zeiger auf eine

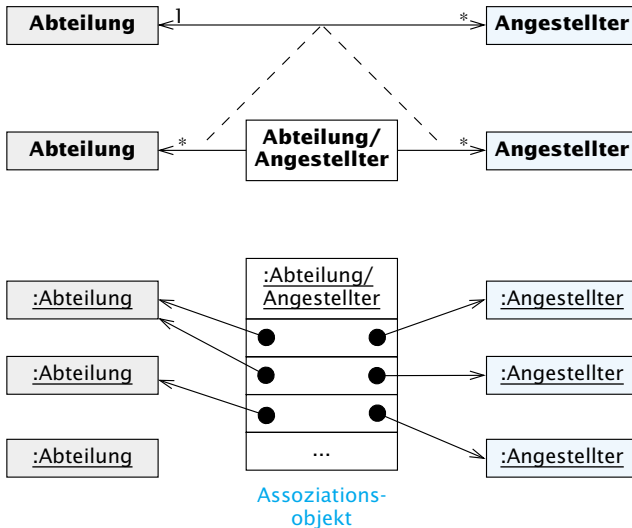


Abb. 6.4-3:  
Realisierung einer  
bidirektionalen  
Assoziation mittels  
Assoziationsobjekt

Abteilung und einen Angestellten. Ein Objekt der Klasse Abteilung kann unabhängig von Angestellten existieren. Daher ist noch kein Eintrag im Assoziationsobjekt notwendig. Beim Erfassen eines neuen Angestellten muß dagegen ein neuer Eintrag erfolgen. Wenn das zugehörige Abteilungsobjekt noch nicht existiert, dann muß es erzeugt werden. Beim Löschen eines Angestellten muß der zugehörige Eintrag im Assoziationsobjekt ebenfalls entfernt werden. Beim Auflösen einer Abteilung müssen alle betroffenen Einträge im Assoziationsobjekt modifiziert, d.h. die Angestellten einer anderen Abteilung zugewiesen werden.

Die Aggregation wird im Prinzip genauso realisiert wie die »normale« Assoziation. Jedoch muß ein Ganzes stets seine Teile kennen, d.h. es muß eine Navigation vom Ganzen (Aggregatobjekt) zu den Teilen möglich sein.

Aggregation

Auch bei der Komposition muß eine Navigation vom Ganzen zu den Teilen existieren. Desweiteren ist darauf zu achten, daß Operationen, die das Ganze betreffen, sich auch auf seine Teile auswirken. Das Ganze und die Teile einer Komposition sind als Einheit zu betrachten. Dazu gehört z.B. das Sperren/Entsperren und die Autorisierung. Der Zugriff im Dialog und das Erzeugen der Teile erfolgen immer über das Aggregatobjekt.

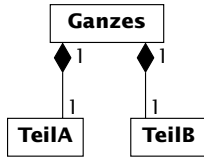
Komposition

Eine Komposition kann außer über Zeiger (*by reference*) auch über echtes physisches Enthaltensein (*by value*) realisiert werden. Die Abb. 6.4-4 zeigt beide Möglichkeiten in C++. Eine *by value*-Realisierung besitzt den Vorteil, daß die Teile automatisch mit dem Ganzen erzeugt bzw. gelöscht werden. Auch das Kopieren des Aggregatobjekts bezieht sich immer automatisch auf seine Teile. Bei einer *by reference*-Realisierung muß das Erzeugen und Löschen der Teile

Realisierung  
Komposition

## LE 11 6 Konzepte und Notation für OOD

Abb. 6.4-4:  
Realisierung der  
Komposition



```

class Ganzes
{
    Teil A einTeil A;
    Teil B* einTeil B;
}

public:
    Ganzes()
    {
        einTeil B = new Teil B;
    }
    ~Ganzes()
    {
        delete einTeil B;
    }
};
    
```

durch den Konstruktor bzw. Destruktor durchgeführt werden. Beim Kopieren muß die entsprechende Operation auch das Kopieren der Teile realisieren.

ordered  
sorted

Bereits in der Analyse wurden Assoziationen, deren Objektverbindungen (*links*) geordnet sind, mit {ordered} gekennzeichnet. Genauere Informationen über die Art der Ordnung müssen – wenn notwendig – separat beschrieben werden. Im Entwurf kann die Angabe {sorted} verwendet werden. Sie sagt aus, daß als Ordnungskriterium die Elementwerte verwendet werden. Beispielsweise können alle Kundenobjekte nach der Kundennummer sortiert sein. Genauere Informationen zur Sortierung müssen durch eine separate Restriktion formuliert werden. Bei einer geordneten Assoziation muß zur Realisierung eine *Container*-Klasse verwendet werden, die eine Ordnung ihrer Elemente ermöglicht (z.B. *Array*, *Vector*).

Merkmale

Die UML ermöglicht es, auch für Assoziationen eine Reihe von Merkmalen zu definieren, die auf jeder Seite einer Assoziation angegeben werden können.

- {frozen}

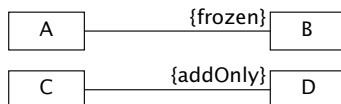
In Abb. 6.4-5 wird spezifiziert, daß eine Objektverbindung weder hinzugefügt noch gelöscht oder geändert werden kann, nachdem ein Objekt der Klasse B erzeugt und initialisiert wurde.

- {addOnly}

Dieses Merkmal gibt an, daß für ein Objekt der Klasse D – bei einer *many*-Kardinalität – zwar weitere Verbindungen eingetragen, vorhandene Verbindungen aber nicht entfernt werden dürfen (Abb. 6.4-5).

- Wird *kein* Merkmal angegeben, dann können Objektverbindungen beliebig hinzugefügt und entfernt werden.

Abb. 6.4-5:  
Merkmale einer  
Assoziation



Sichtbarkeit

Für Assoziationen können in der UML zusätzlich die Sichtbarkeiten angegeben werden. Analog zu Attributen und Operationen werden +, #, – oder ein explizites Schlüsselwort (z.B. {public}) als Präfix des Rollennamens verwendet.

Als Beispiel wird die bidirektionale Assoziation der Abb. 6.4-1 realisiert. Die Basisoperationen `link()`, `unlink()` und `getlink()` müssen in C++ implementiert werden. Die Operation `getlink()` gibt hier die ermittelte OID als Referenzparameter statt als Ergebnisparameter zurück (d.h. `Abteilung* getlink()`). Dadurch kann in C++ das Konzept des Überladens ausgenutzt werden, wenn Assoziationen zu mehreren Klassen bestehen.

### Kardinalität maximal 1

```
class Angestellter
{ protected:
    Abteilung* arbeitIn;
public:
    void link (Abteilung* abt);    //Erstellen einer Verbindung
    void unlink (Abteilung* abt); //Löschen einer Verbindung
    void getlink (Abteilung* &abt); //Lesen einer Verbindung
};
void Angestellter::link (Abteilung* abt)
{ arbeitIn = abt;
}
void Angestellter::unlink (Abteilung* abt)
{ ...
  arbeitIn = NULL;
}
void Angestellter::getlink (Abteilung* &abt)
{ abt = arbeitIn;
}
```

### Kardinalität größer als 1

Eine Menge von Objekten läßt sich elegant mit einer generischen Klasse verwalten. Wir verwenden hier die generische Klasse `VectorBag`. Sie verwaltet eine Menge von Zeigern auf Objekte einer noch zu bestimmenden Klasse und stellt unter anderem die Operationen `addElement()`, `delElement()` und `getElement()` zur Verfügung.

```
template <class VObject>
class VectorBag
{//verwaltet Referenzen (Zeiger) auf Objekte von VObject
    ...
    virtual void addElement (VObject* Obj);
    void delElement (int pos);
    VObject* getElement (int pos);
};
```

Die Basisoperationen `link()`, `unlink()` und `getlink()` müssen in C++ implementiert werden. Die Operation `getlink()` wurde hier so implementiert, daß sie immer nur eine Objektverbindung liest. Alternativ wäre eine Realisierung möglich, die alle Objektverbindungen einer Assoziation in einem Aufruf ermittelt.

```
class Abteilung
{protected:
    VectorBag <Angestellter> Mitarbeiter;
public:
    void link (Angestellter* ang);
    void unlink (Angestellter* ang);
    int getlink (Angestellter* &ang, int pos); // =0, wenn ok
};
void Abteilung::link (Angestellter* ang)
{ Mitarbeiter.addElement (ang);
}
void Abteilung::unlink (Angestellter* ang)
{ ...
  Mitarbeiter.delElement (ang);
}
int Abteilung::getlink (Angestellter* &ang, int pos)
{ ...
  ang = Mitarbeiter.getElement (pos);
  ...
}
```

### Assoziation in Java

Analog zu C++ realisieren wir als Beispiel die bidirektionale Assoziation der Abb. 6.4-1. Die Operation `getlink()` gibt bei dieser Implementierung die Referenz eines Objekts als Ergebnisparameter zurück. Gehen von der Klasse `Angestellter` mehrere Assoziationen aus, dann sind für die *getlink*-Operationen Namen wie `getlinkAbteilung()` zu verwenden, da der Ergebnisparameter beim Überladen von Operationsnamen nicht berücksichtigt wird.

#### Kardinalität maximal 1

```
class Angestellter
{ protected Abteilung arbeitIn;

    public void link (Abteilung abt)
    { arbeitIn = abt;
    }
    public void unlink (Abteilung abt)
    { ...
      arbeitIn = null;
    }
    public Abteilung getlink ()
    { return arbeitIn;
    }
}
```

## Kardinalität größer als 1

In Java gibt es keine generischen Klassen. Die Sprache bietet jedoch geeignete Klassen an, um eine Menge von Objekten zu verwalten. Wir verwenden die vorhandene Klasse `Vector`, die unter anderem die Operationen `addElement()`, `removeElement()` und `elementAt()` zur Verfügung stellt.

```
class Abteilung
{
    protected Vector Mitarbeiter; //verwaltet eine Menge von
                                //Referenzen

    public void link (Angestellter ang)
    {Mitarbeiter.addElement (ang);
    }
    void unlink (Angestellter ang)
    {Mitarbeiter.removeElement (ang);
    }
    Angestellter getlink (int pos)
    {Angestellter ang;
    ...
    ang = (Angestellter) Mitarbeiter.elementAt (pos);
    return ang;
    }
}
```

**Abstrakte Operation (*abstract operation*)** Eine Operation, für die nur die Signatur angegeben ist, die aber nicht spezifiziert bzw. implementiert ist (*pure virtual member function* in C++). Enthält eine Klasse mindestens eine abstrakte Operation, dann handelt es sich um eine abstrakte Klasse. Die zugehörige Spezifikation bzw. Implementierung wird erst in den Unterklassen angegeben.

**Abstraktion (*abstraction*)** Abstraktion, als ein Prozeß betrachtet, bezeichnet die Vorgehensweise, die wesentlichen Informationen über etwas zu ermitteln und die unwesentlichen Informationen zu ignorieren.

Abstraktion, als Ergebnis betrachtet, bezeichnet ein Modell oder einen bestimmten Blickwinkel.

**Assoziation (*association*)** Eine Assoziation modelliert Verbindungen zwischen  $\rightarrow$ Objekten einer oder mehrerer  $\rightarrow$ Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwi-

schen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch Kardinalitäten und einen optionalen Assoziationsnamen oder Rollenamen. Sie kann um Restriktionen ergänzt werden. Besitzt eine Assoziation selbst wieder Attribute und ggf. Operationen und Assoziationen zu anderen Klassen, dann wird sie zur assoziativen Klasse. Die Qualifikationsangabe (*qualifier*) zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Eine abgeleitete Assoziation liegt vor, wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sonderfälle der Assoziation sind die Aggregation und die Komposition. In der Analyse ist jede Assoziation inhärent bidirektional. Im Entwurf wird die gewünschte  $\rightarrow$  Navigationsrichtung angegeben.

**Attribut (*attribute*)** Attribute beschreiben Daten, die von den  $\rightarrow$ Objekten der  $\rightarrow$ Klasse angenommen werden können. Alle Objekte einer Klasse besit-

zen dieselben Attribute, jedoch im allgemeinen unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten Typ und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muß jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig. Abgeleitete Attribute lassen sich aus anderen Attributen berechnen.

**Container-Klasse** Eine *Container-Klasse* ist eine →Klasse, deren Objekte Mengen von →Objekten (anderer) Klassen sind. Sie können homogene Mengen verwalten, d.h. alle Objekte einer Menge gehören zur selben Klasse, oder auch heterogene Mengen, d.h. die Objekte einer Menge gehören zu unterschiedlichen Unterklassen einer gemeinsamen Oberklasse. *Container-Klassen* werden oft mittels →generischer Klassen realisiert.

**Geheimnisprinzip (*information hiding*)** Die Einhaltung des Geheimnisprinzips bedeutet, daß die →Attribute und die Implementierung der →Operationen außerhalb der →Klasse nicht sichtbar sind.

**Generische Klasse (*parameterized class, template*)** Eine generische Klasse ist eine Beschreibung einer →Klasse mit einem oder mehreren formalen Parametern. Sie definiert daher eine Familie von Klassen. →*Container-Klassen* werden häufig als generische Klassen realisiert.

**Klasse (*class*)** Eine Klasse definiert für eine Kollektion von →Objekten deren Struktur (→Attribute), das Verhalten (→Operationen) und Beziehungen (→Assoziationen, Vererbungsstrukturen). Klassen besitzen – mit Ausnahme von abstrakten Klassen – einen Mechanismus, um neue Objekte zu erzeugen. Der Klassenname muß mindestens im Paket, besser im gesamten System eindeutig sein.

**Nachbedingung (*postcondition*)** Die Nachbedingung beschreibt die Änderung, die durch eine Verarbeitung bewirkt wird, unter der Voraussetzung, daß vor ihrer Ausführung die →Vorbedingung erfüllt war.

**Navigation (*navigability*)** Die Navigation legt im Entwurf fest, ob eine →Assoziation uni- oder bidirektional implementiert wird.

**Objekt (*object*)** Ein Objekt besitzt einen Zustand (Attributwerte und Verbindungen zu anderen Objekten), reagiert mit einem definierten Verhalten (Operationen) auf seine Umgebung und besitzt eine Objektidentität, die es von allen anderen Objekten unterscheidet. Jedes Objekt ist Exemplar einer →Klasse.

**Operation (*operation*)** Eine Operation ist eine Funktion, die auf die internen Daten (Attributwerte) eines →Objekts Zugriff hat. Sie kann Botschaften an andere Objekte senden. Auf alle Objekte einer Klasse sind dieselben Operationen anwendbar. Für Operationen gibt es im allgemeinen in der Analyse eine fachliche Beschreibung. Sie wird in einer objektorientierten Programmiersprache durch eine Implementierung (Methode) realisiert. Abstrakte Operationen besitzen nur eine →Signatur. Externe Operationen werden vom späteren Bediener des Systems aktiviert. Interne Operationen werden dagegen immer von anderen Operationen aufgerufen.

**Parametrisierte Klasse** →generische Klasse

**Schnittstelle (*interface*)** In der UML besteht eine Schnittstelle nur aus Operationen, die keine Implementierung besitzen. Sie ist äquivalent zu einer →Klasse, die keine →Attribute, Zustände oder →Assoziationen und ausschließlich →abstrakte Operationen besitzt. Die Menge aller →Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die Schnittstelle der →Klasse bzw. des →Objekts. Eine Schnittstelle kann in Java aus Konstanten und abstrakten →Operationen bestehen.

**Sichtbarkeit (*visibility*)** Die Sichtbarkeit legt fest, ob auf →Attribute und →Operationen außerhalb ihrer →Klasse zugegriffen werden kann. Auch für →Assoziationen kann die Sichtbarkeit definiert werden. Die UML unterscheidet die folgenden Sichtbarkeiten: *public*



= sichtbar für alle Klassen, *protected* = sichtbar innerhalb der Klasse und für alle ihre Unterklassen, *private* = sichtbar nur innerhalb der Klasse.

**Signatur (signature)** Die Signatur einer →Operation besteht aus dem Namen der Operation, den Namen und Typen aller Parameter, und dem Ergebnistyp der Operation.

**Verkapselung (encapsulation)** Die Verkapselung sagt aus, daß zusammengehörende →Attribute und →Operationen in einer Einheit zusammengefügt sind.

**Vorbedingung (precondition)** Die Vorbedingung beschreibt, welche Bedingungen vor dem Ausführen einer Verarbeitung erfüllt sein müssen, damit die Verarbeitung definiert ausgeführt werden kann.



Das Konzept der Klasse wird im Entwurf um generische Klassen, Container-Klassen und Schnittstellen erweitert. Für Attribute und Operationen wird die Notation um Sichtbarkeiten erweitert. Bei Operationen ist außerdem die komplette Signatur anzugeben. Die Notation von Assoziationen wird um die Navigation und die Sichtbarkeit erweitert. Assoziationen können auf verschiedene Arten (z.B. mittels Zeigern) realisiert werden.

**1 Lernziel: Objektorientierte Konzepte des Entwurfs kennen.**

Aufgabe  
10 Minuten

- a** Welche Vorteile besitzt eine generische Klasse?
- b** Was ist der Unterschied zwischen einer Schnittstelle und einer abstrakten Klasse in der UML?
- c** Geben Sie ein Programm in Java oder C++ an, das die Verkapselung erfüllt, aber nicht das Geheimnisprinzip.
- d** Wann verwenden Sie eine abstrakte Operation?
- e** Welche Vorteile besitzt die Realisierung einer Assoziation als Klasse?

**2 Lernziel: Objektorientierte Notation des Entwurfs anwenden können.**

Aufgabe  
10 Minuten

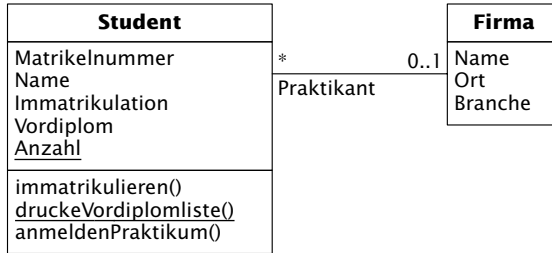
Spezifizieren Sie folgende Problemstellungen als OOD-Klassendiagramm und als Objektdiagramm.

- a** Wenn ein Patient (in einer Arztpraxis) behandelt wird, dann sollen die letzten 10 Behandlungen angezeigt werden.
- b** Ein Artikel wird von einem oder mehreren Lieferanten bezogen. Ein Lieferant liefert im Normalfall mehrere Artikel. Es soll eine Liste aller Artikel erstellt werden, die für jeden Artikel alle zugehörigen Lieferanten enthält. Außerdem soll für einen beliebigen Lieferanten eine Liste aller von im gelieferten Artikel erstellt werden.

## LE 11 Aufgaben

- Aufgabe 3 *Lernziele: Klassen und Assoziationen des Analysemodells in den Entwurf abbilden und die Objektverwaltung realisieren können.*  
 10–15 Minuten  
 Bilden Sie das Klassendiagramm der Abb. LE11-A3 in den Entwurf ab. Erstellen Sie ein Klassendiagramm und ein Objektdiagramm.

Abb. LE11-A3:  
 Klassendiagramm  
 für Studenten und  
 Praktikumsfirmen



- Aufgabe 4 *Lernziele: Generische Klassen entwerfen und Signaturen in UML spezifizieren können.*  
 15 Minuten  
 Entwerfen Sie generische Klassen für folgende Probleme und geben Sie die grafische Darstellung der Klassen einschließlich der vollständigen Signaturen in UML-Notation an:
- Liste (geordnete Kollektion) mit folgenden Operationen:
    - Einfügen (*insert*) eines Elements am Ende der Liste,
    - Entfernen (*remove*) eines Elements an der angegebenen Position,
    - Lesen (*retrieve*) eines Elements an der angegebenen Position.
  - Menge (ungeordnete Kollektion ohne Duplikate) mit folgenden Operationen:
    - Einfügen (*insert*) eines Elements,
    - Prüfung auf Enthaltensein (*containsElement*) eines Elements,
    - Bilden des Durchschnitts zweier Mengen (*createIntersection*),
    - Prüfen, ob eine Menge echte Teilmenge einer anderen Menge ist (*isProperSubset*).
- Berücksichtigen Sie bei dieser Aufgabe keine Fehlerbehandlung.

## 6 Konzepte und Notation des objektorientierten Entwurfs (Teil 2)



- Erklären können, was Polymorphismus ist und welche Vorteile er mit sich bringt.
- Erklären können, was Mehrfachvererbung ist.
- Verschiedene Vererbungsstrukturen unterscheiden können.
- Erklären können, wie Interaktionsdiagramme eingesetzt werden.
- Erklären können, wie Zustandsdiagramme eingesetzt werden.
- Verstehen, wie die Konzepte Polymorphismus, Vererbung, Paket, Szenario und Zustandsautomat in C++ und Java umgesetzt werden.
- Polymorphismus anwenden können.
- Interaktionsdiagramme zur Modellierung von Programmabläufen einsetzen können.
- Den Objekt-Lebenszyklus mit dem Zustandsmuster in den Entwurf transformieren können.

verstehen

anwenden



- Die objektorientierten Konzepte der Analyse und die UML-Notation aus Kapitel 2 müssen bekannt sein.
- Sie sollten Kapitel 6.1 bis 6.4 durchgearbeitet haben.
- In diesem Kapitel werden grundlegende Programmierkenntnisse vorausgesetzt.
- Basiswissen in einer objektorientierten Programmiersprache erleichtert das Verständnis.

- i 6.5 Polymorphismus 256
- 6.6 Vererbung 261
- 6.7 Paket 267
- 6.8 Szenario 269
- 6.9 Zustandsautomat 274

## 6.5 Polymorphismus

**Definition** Ein wichtiges Konzept des objektorientierten Paradigmas ist der Polymorphismus (bzw. die Polymorphie). Das Konzept des Polymorphismus ist in der Analyse von untergeordneter Bedeutung und kann erst im Entwurf und in der Implementierung richtig genutzt werden.

Mit kaum einem Konzept der objektorientierten Softwareentwicklung haben Lernende so viele Probleme wie mit dem Polymorphismus. Das hat meines Erachtens mehrere Ursachen. Zum einen wird der Begriff von verschiedenen Autoren oft unterschiedlich beschrieben, zum anderen spielen hier mehrere Konzepte zusammen. Und schließlich spielt auch die verwendete Programmiersprache dabei eine Rolle, wie Polymorphismus verstanden wird.

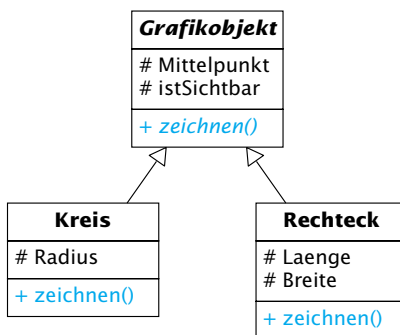
**Definition** Der **Polymorphismus** (*polymorphism*) ermöglicht es, den gleichen Namen für gleichartige Operationen zu verwenden, die auf Objekten verschiedener Klassen auszuführen sind. Der Sender muß nur wissen, daß ein Empfängerobjekt das gewünschte Verhalten besitzt; er muß nicht wissen, zu welcher Klasse das Objekt gehört. Dieser Mechanismus ermöglicht es, flexible und leicht änderbare Softwaresysteme zu entwickeln. Das Gegenteil von Polymorphismus ist Monomorphismus. Er ist in allen Programmiersprachen vorhanden, die sowohl eine strenge Typisierung als auch eine frühe Bindung (zur Übersetzungszeit) besitzen.

**Beispiel** Wir gehen von der Vererbungsstruktur der Abb. 6.5-1 aus. Desweiteren deklarieren wir einen Zeiger `pGrafik`.

`Grafik objekt* pGrafik;`

Dann kann der Operationsaufruf `pGrafik->zeichnen()` völlig unterschiedliche Wirkungsweisen besitzen. Gilt `pGrafik=new Kreis`, dann wird die Operation `Kreis.zeichnen()` aktiviert. Gilt `pGrafik=new Rechteck`, dann wird `Rechteck.zeichnen()` ausgeführt.

Abb. 6.5-1:  
Beispiel zum  
Polymorphismus



**spätes Binden** Wie das obige Beispiel zeigt, wird erst zur Laufzeit des Programms bestimmt, ob der Zeiger `pGrafik` auf ein Kreis- oder ein Rechteck-

Objekt zeigt. Man spricht daher von später oder dynamischer Bindung. Polymorphismus und **spätes Binden** (*late binding*) sind untrennbar verbunden. Ist zur Übersetzungszeit die Klasse des Objekts nicht bekannt, dann kann zu diesem Zeitpunkt noch nicht bestimmt werden, welche Operation ausgeführt wird. Spätes Binden bedeutet, daß eine Operation erst zur Ausführungszeit an ein bestimmtes Objekt gebunden wird. Wir sprechen auch von einer **polymorphen Operation**. Das späte Binden ist in den objektorientierten Programmiersprachen unterschiedlich realisiert. Während in C++ Operationen explizit polymorph deklariert werden müssen, sind in Java und Smalltalk alle Operationen automatisch polymorph.

Polymorphismus bedeutet, daß dieselbe Botschaft an Objekte verschiedener Klassen gesendet werden kann und daß die Empfängerobjekte jeder Klasse auf ihre eigene – evtl. ganz unterschiedliche – Art darauf reagieren. Das bedeutet, daß der Sender einer Botschaft nicht wissen muß, zu welcher Klasse das Empfängerobjekt gehört /Wirfs-Brock 90/.

### Polymorphismus-Definitionen

Wirfs-Brock

Ein Name (z.B. eine Variablendeklaration) kann Objekte verschiedener Klassen bezeichnen, die durch eine gemeinsame Oberklasse verbunden sind. Jedes Objekt, das durch diesen Namen bezeichnet wird, kann auf die gleiche Botschaft auf seine eigene Art und Weise reagieren /Booch 94b/.

Booch

Nach /Eisenecker 95/ bedeutet Polymorphismus, »daß ein Objekt in unterschiedlichen Erscheinungsformen auftreten und eine Variable verschiedenartige Objekte aufnehmen kann.«

Eisenecker

Wird eine Botschaft an ein Objekt gesendet, dann wird sowohl durch die Botschaft als auch durch das Empfänger-Objekt bestimmt, welche Operation ausgeführt wird. Verschiedene Objekte, die auf die gleiche Botschaft reagieren, können daher unterschiedliche Implementierungen der Operationen, die diese Botschaft realisieren, besitzen. Dynamisches Binden (*dynamic binding*) bedeutet, daß einer Botschaft erst zur Laufzeit eine entsprechende Klasse zugeordnet wird. Dynamisches Binden ermöglicht es Ihnen, Objekte mit identischen Schnittstellen während der Laufzeit beliebig auszutauschen. Diese Austauschbarkeit wird als Polymorphismus bezeichnet /Gamma et al. 95/.

Gamma

Die Anwendung des Polymorphismus macht umfangreiche *switch*-Anweisungen, in denen entsprechend dem Objekttyp eine Aktion ausgelöst wird, überflüssig. Das Vorhandensein solcher *switch*-Anweisungen ist ein Indiz dafür, daß der Polymorphismus nicht angewendet wurde. Bei herkömmlicher strukturierter Programmierung (z.B. in C) wäre für obiges Beispiel folgende Konstruktion notwendig:

## LE 12 6 Konzepte und Notation für OOD

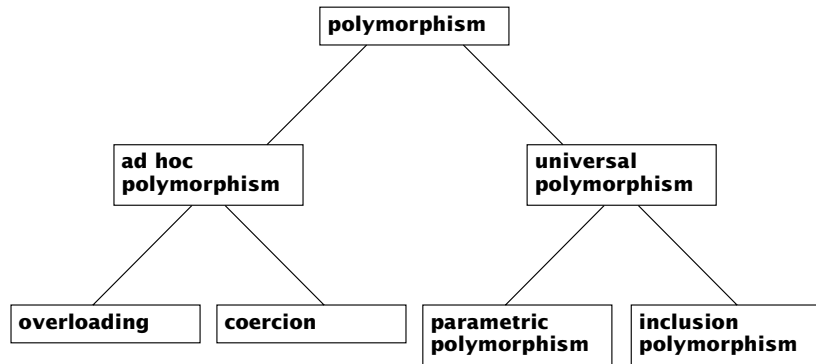
```
Beispiel enum Grafi kart {istRechteck, istKreis};  
voi d zeichnenGrafi k (Grafi k Grafi kdaten)  
swi tch (Grafi kdaten. Art)  
  
    { case i stRechteck: zeichnenRechteck(Grafi kdaten); break;  
      case i stKreis: zeichnenKreis(Grafi kdaten); break;  
    }
```

### Interpretationen des Begriffs Polymorphismus

Cardelli und Wegner haben 1985 für die verschiedenen Formen des Polymorphismus die in der Abb. 6.5-2 beschriebene Systematik aufgestellt /Blair et al. 91/.



Abb. 6.5-2:  
Formen des  
Polymorphismus



**Überladen** (*overloading*) erlaubt Operationen mit demselben Namen – jedoch unterschiedlicher Semantik und Implementierung – für Objekte verschiedener Klassen zu verwenden. Beispielsweise kann die Addiere-Operation sowohl für ganze Zahlen ( $5 + 2 \rightarrow 7$ ) als auch für Zeichenketten («alpha» + »bet«  $\rightarrow$  »alphabet«) definiert sein. Diese Form des Polymorphismus wurde schon in den frühen Programmiersprachen in beschränktem Umfang angewendet. In den objekt-orientierten Sprachen wurde dieses Konzept systematisch weiterentwickelt und allgemein verfügbar gemacht. **Coercion** ist seit langem bekannt. Sprachen, die diese Form des Polymorphismus unterstützen, erlauben bestimmte Typumwandlungen. Ist beispielsweise die Addition für zwei reelle Zahlen definiert und werden bei deren Anwendung eine ganze und eine reelle Zahl verwendet, dann wird die ganze Zahl automatisch in eine reelle umgewandelt. Die beiden genannten Formen werden von Cardelli und Wagner als **ad hoc-Polymorphismus** bezeichnet, denn sie sind immer nur für bestimmte Typen der Programmiersprache definiert. Wir wollen das polymorphe Verhalten jedoch auf beliebig viele Typen anwenden können. Die beiden nächsten Formen des Polymorphismus werden daher auch als **universeller Polymorphismus** bezeichnet. **Parametrischer Polymorphismus** (generischer Polymorphismus, *parametric polymorphism*) liegt vor, wenn eine Funktion, die nur einmal programmiert

ist, mit verschiedenen Typen arbeiten kann. Die Funktion besitzt Typen als Parameter, d.h. die Argumente der Funktion können von unterschiedlichem Typ sein. Wir sprechen von einer parametrischen oder generischen Funktion. Allgemein ausgedrückt bedeutet parametrischer Polymorphismus, die Fähigkeit ein »Stück Software« mit einem oder mehreren Typen zu parametrisieren. /Eisenecker 95/ spricht hier auch von signaturgebundenem Polymorphismus. Es ist die einzige Art des Polymorphismus in Smalltalk (hier sind alle Operationen generisch) und entspricht in C++ den generischen Klassen (*templates*). Diese Art des Polymorphismus bietet maximale Flexibilität. Sie birgt jedoch die Gefahr, Fehlermeldungen der Art »*message not understood*« zu erhalten. Auch beim **inklusionsbasierten Polymorphismus** (*inclusion polymorphism*) kann eine Funktion mit mehreren Typen arbeiten, wobei allerdings eine Einschränkung auf Untertypen (*subtypes*) besteht. Eine Funktion, die für einen bestimmten Typ definiert ist, kann auch mit jedem Untertyp dieses Typs arbeiten. /Eisenecker 95/ spricht hier von vererbungsgebundenem Polymorphismus. Diese Form des Polymorphismus schließt aus, daß ein Objekt eine Botschaft zur Laufzeit unerwarteterweise nicht versteht. Sie erzwingt jedoch die Definition von Klassen, die lediglich als Schnittstellen für abgeleitete Klassen dienen.

Das folgende Beispiel von Eisenecker /Eisenecker 98/ zeigt sehr anschaulich den Unterschied zwischen vererbungs- und signaturgebundenem Polymorphismus. Obwohl in der Signatur der Funktion `hallo()` nur die Klasse A aufgeführt ist, wird beim vererbungsgebundenen Polymorphismus die Operation `gruesse()` auch auf die Unterklasse B angewendet, wenn der Aufruf `hallo(B)` erfolgt. Beim signaturgebundenen Polymorphismus wird der formale Parameter wahlweise an die Klasse A bzw. an die Klasse B gebunden. Beide Programme erzeugen die Bildschirmausgabe: »Gruesse von A« und »Gruesse von B«.

Beispiel

```
//Vererbungsgebundener Polymorphismus in C++
#include <iostream.h>

class A
{public:
    virtual void gruesse () const;
};
class B: public A
{public:
    virtual void gruesse () const;
};

void A::gruesse () const
{ cout << "Gruesse von A" <<endl;
}
void B::gruesse () const
{ cout <<"Gruesse von B" << endl;
}
```

## LE 12 6 Konzepte und Notation für OOD

```
void hallo (const A& partner) //normale Funktion
{ partner.gruesse ();
}
void main()
{ hallo (A());
  hallo (B());
}
```

Und nun das analoge Beispiel zum signaturgebundenen Polymorphismus.

```
//Signaturgebundener Polymorphismus in C++
#include <iostream.h>

class A
{public:
  void gruesse () const;
};
class B
{public:
  void gruesse () const;
};

void A::gruesse () const
{ cout << "Gruesse von A" <<endl;
}
void B::gruesse () const
{ cout <<"Gruesse von B" << endl;
}
template <class T>           //normales Funktionstemplate
void hallo (const T& partner)
{ partner.gruesse ();
}
void main()
{ hallo (A());
  hallo (B());
}
```

### Polymorphismus in C++

#### Polymorphe Operation

■ Operationen sind nur polymorph, wenn sie als `virtual` deklariert werden. Dann werden die Referenzen erst zur Laufzeit aufgelöst. Ohne dieses Schlüsselwort findet ein frühes Binden statt, d.h. eine Bindung zur Übersetzungszeit.

■ Für die Ausnutzung des Polymorphismus ist es wichtig, Zeiger oder Referenzen zu verwenden. Ein Zeiger vom Typ einer Oberklasse kann auf Objekte aller Unterklassen verweisen.

```
class Grafikobjekt
{public:
  virtual void zeichnen() = 0;
  ...
};
```



```

class Kreis: public Grafikobjekt
{public:
    virtual void zeichnen();
    ...
};
class Rechteck: public Grafikobjekt
{public:
    void zeichnen();
    ...
};

Grafikobjekt* pGrafik;
pGrafik = new Kreis;
pGrafik->zeichnen(); //zeichnet einen Kreis
delete pGrafik;

pGrafik = new Rechteck;
pGrafik->zeichnen(); //zeichnet ein Rechteck
delete pGrafik;

```

### Polymorphe Operation

- In Java sind Operationen automatisch polymorph, d.h. die Referenzen werden erst zur Laufzeit aufgelöst.
- Im Gegensatz zu C++ muß eine Operation – durch die Angabe von `final` – explizit als nicht virtuell deklariert werden. Dann kann keine Klasse diese Operation überschreiben.

```

class Grafikobjekt
{ public void zeichnen();
  ...
}
class Kreis extends Grafikobjekt
{ public void zeichnen();
  ...
}
class Rechteck extends Grafikobjekt
{ public void zeichnen();
  ...
}

Grafikobjekt einGrafik;
einGrafik = new Kreis();
einGrafik.zeichnen(); //zeichnet einen Kreis

einGrafik = new Rechteck();
einGrafik.zeichnen(); //zeichnet ein Rechteck

```

### Polymorphismus in Java

## 6.6 Vererbung

Bei der Erstellung des Analysemodells haben wir uns auf die Einfachvererbung beschränkt. In Entwurf und Implementierung wird neben der Einfachvererbung auch die Mehrfachvererbung benötigt. Definition

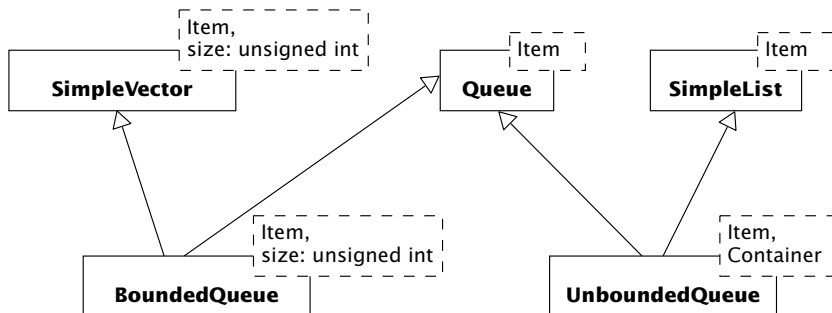
## LE 12 6 Konzepte und Notation für OOD

Die **Einfachvererbung** ist eine Vererbungsstruktur, in der jede Klasse – mit Ausnahme der Wurzel – genau eine direkte Oberklasse besitzt. Es entsteht eine Baumhierarchie. Die **Mehrfachvererbung** ist eine Vererbungsstruktur, in der jede Klasse mehrere direkte Oberklassen besitzen kann. Sie bildet einen azyklischen Graphen, der mehr als eine Wurzel haben kann. Bei der Mehrfachvererbung kann der Fall auftreten, daß eine Klasse von ihren Oberklassen zwei Attribute oder Operationen gleichen Namens – aber unterschiedlichen Inhalts – erbt. Es muß daher festgelegt werden, wie diese Konflikte zu lösen sind.

**Beispiel** Ein interessantes Beispiel für Mehrfachvererbung enthält die Booch-Klassenbibliothek in C++ /Booch 93/. Diese Bibliothek bietet unter anderem *Container*-Klassen wie *Queue*, *Set*, *Stack* an. Abb. 6.6-1 zeigt deren Struktur am Beispiel der Warteschlange. Die Klasse *Queue* kapselt die reinen Eigenschaften der Warteschlange, bei der Elemente an einem Ende eingefügt und am anderen Ende entfernt werden. Die Klasse *SimpleVector* bietet die Speicherfunktionalität eines Arrays, die Klasse *SimpleList* die Speichermöglichkeiten der verketteten Liste an.

```
template <class Item> class Queue { ... };
template <class Item, unsigned int size> class SimpleVector
{ ... };
template <class Item> class SimpleList { ... };
template <class Item, unsigned int size> class BoundedQueue:
    private SimpleVector <Item, size>,
    public Queue <Item> { ... };
template <class Item, class Container> class UnboundedQueue:
    private SimpleList <Item>,
    public Queue <Item> { ... };
```

Abb. 6.6-1:  
Beispiel einer  
Mehrfach-  
vererbung



**Probleme** Die Mehrfachvererbung birgt noch viel stärker als die Einfachvererbung die Gefahr der Spaghetti-Vererbung (*spaghetti inheritance*). Darunter ist eine Vererbungsstruktur zu verstehen, die ähnlich einer Spaghetti-Programmierung nur sehr schwierig zu verstehen und für die Wartung eine Katastrophe ist. Trotzdem kann die

Mehrfachvererbung in der Entwurfsphase oft vorteilhaft verwendet werden. Ich betrachte sie in Übereinstimmung mit /Booch 94/ als eine Art Fallschirm. Sie wird nur selten benötigt, aber wenn, dann ist man heilfroh, sie zu haben.



Bei der Mehrfachvererbung können Namenskonflikte auftreten. Das ist immer dann der Fall, wenn zwei oder mehr Oberklassen den gleichen Namen für ein Attribut oder eine Operation verwenden. Es gibt drei grundlegende Ansätze, um diesen Konflikt zu lösen /Booch 94/:

Namenskonflikte

- Treten Namenskonflikte auf, dann müssen die Attribute und Operationen der Oberklassen entsprechend umbenannt werden, oder es muß auf die Mehrfachvererbung verzichtet werden.
- Attribute, die in den Oberklassen namentlich identisch sind, müssen auch vom selben Typ sein.
- Bei der Unterklasse wird durch Angabe der jeweiligen Oberklasse festgelegt, welches Attribut bzw. welche Operation geerbt werden soll. Dieser Ansatz ist in C++ realisiert.

### Restriktionen

Für Vererbungsstrukturen können Restriktionen angegeben werden. Die UML kennt folgende vordefinierte Restriktionen.

#### ■ *overlapping*

In der Abb. 6.6-2 besitzt ein Segelboot sowohl Eigenschaften von Fahrzeugen mit Windantrieb als auch von Wasserfahrzeugen. Die Klasse Segelboot wird daher von beiden Klassen abgeleitet, d.h. als Mehrfachvererbung modelliert.

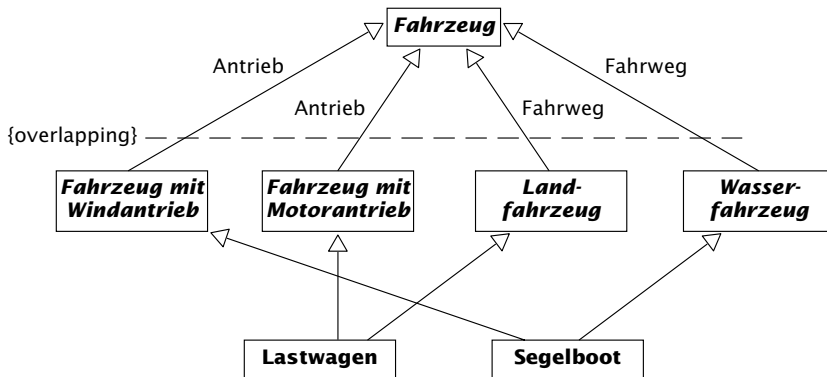


Abb. 6.6-2:  
Restriktion  
*overlapping*  
/UML 97/

#### ■ *disjoint*

Die Eigenschaften der Unterklassen überschneiden sich nicht. In der Abb. 6.6-3 werden unterschiedliche Spezies von Bäumen modelliert.

#### ■ *complete*

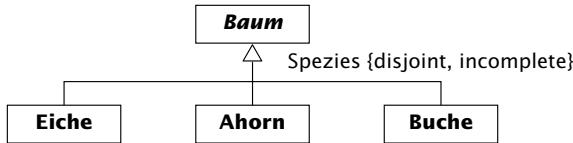
Die Menge der Unterklassen ist vollständig. Weitere Unterklassen werden aufgrund der Problemstellung nicht erwartet.

## LE 12 6 Konzepte und Notation für OOD

### ■ *incomplete*

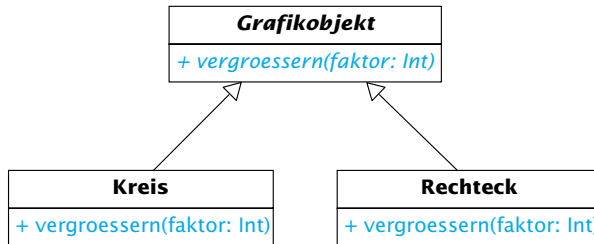
Die betreffende Vererbungsstruktur enthält einen Teil der Unterklassen. Es gibt weitere Unterklassen, die das Modell noch nicht enthält. Beispielsweise modelliert die Vererbungsstruktur der Abb. 6.6-3 nur einige Spezies von Bäumen. Verwechseln Sie diese Restriktion nicht mit der Angabe »...«. Die durch Punkte angedeutete Auslassung bedeutet, daß im aktuellen Diagramm einige Klassen weggelassen wurden, die jedoch im Modell enthalten sind.

Abb. 6.6-3:  
Restriktionen  
disjoint und  
incomplete  
/UML 97/



Überschreiben Von **Überschreiben** (*overriding*) bzw. **Redefinition** (*redefinition*) spricht man, wenn eine Unterklasse eine Operation der Oberklasse – unter dem gleichen Namen – neu implementiert. Dieses Konzept hat den Vorteil, daß ein Programmierer, der eine Vererbungsstruktur benutzt, die verschiedenen (Unter-)Klassen verwenden kann und sich keine Gedanken darüber machen muß, zu welcher Unterklasse ein spezielles Objekt gehört. Diese Eigenschaft erfordert spätes Binden bzw. die Verwendung polymorpher Operationen. In der Abb. 6.6-4 verwendet der Programmierer für alle Grafikobjekte die Operation `vergroessern()`.

Abb. 6.6-4:  
Überschreiben von  
Operationen



Beim Überschreiben einer Operation müssen die Anzahl und Typen der Ein-/Ausgabeparameter gleichbleiben. Wir sprechen von kompatiblen Operationen. In der Systemanalyse gilt diese Forderung übrigens nicht, da wir die Operationen im Analysemodell ohne Parameterschnittstellen modellieren. Wurden für die Operation der Oberklasse *pre-* und *postconditions* spezifiziert, so müssen die *pre-* und *postconditions* der neuen Operation (Unterklasse) dazu konform sein.

Überschreiben vs. Überladen Verwechseln Sie das Überschreiben nicht mit dem Überladen einer Operation. Man spricht von Überladen (*overloading*), wenn der-

selbe Operationsname innerhalb einer Klasse mit verschiedenen Parameterschnittstellen verwendet wird (Abb. 6.6-5). Beim Überschreiben muß die Operation der Unterklasse kompatibel mit derjenigen der Oberklasse sein. Dabei kommt es häufig vor, daß bei der Implementierung von `DerivedClass.doSomething()` die gleichnamige Operation der Oberklasse aufgerufen wird. Hierdurch zeigt sich das typische Verhalten der Unterklasse als Erweiterung der Oberklasse.

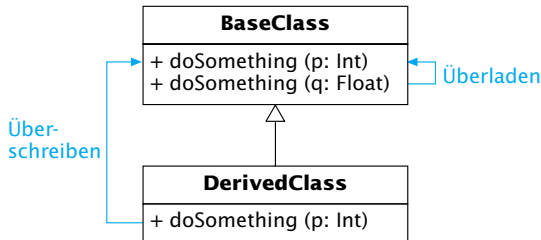


Abb. 6.6-5  
Überschreiben und  
Überladen von  
Operationen



Einige Autoren differenzieren zwischen mehreren Formen der Vererbung. /Bertino, Martino 93/ und /Gamma et al. 95/ unterscheiden beispielsweise Spezifikations- und Implementierungshierarchie.

verschiedene  
Formen  
der Vererbung

Die **Spezifikationshierarchie** (*specification hierarchy, subtype hierarchy, subtyping*) beschreibt das externe Verhalten der Objekte. Sie schreibt vor, daß ein Exemplar der Unterklasse überall dort verwendet werden kann, wo ein Exemplar der Oberklasse erwartet wird. Das hat für die Unterklasse zur Folge, daß nur neue Attribute und Operationen hinzugefügt werden dürfen. Attribute und Operationen dürfen nur dann redefiniert werden, wenn sie mit den Attributen und Operationen der Oberklasse kompatibel sind. Das gilt nur für Attribute und Operationen, die für das externe Verhalten wichtig sind.

Bei der **Implementierungshierarchie** (*implementation inheritance, class inheritance*) dürfen geerbte Attribute und Operationen beliebig redefiniert werden. Diese Hierarchie muß daher nicht mit der Spezifikationshierarchie übereinstimmen.

Die meisten Programmiersprachen unterscheiden nicht zwischen diesen beiden Konzepten. In den Sprachen C++ und Eiffel bedeutet Vererbung sowohl *interface* als auch *implementation inheritance*. In C++ kann die reine *implementation inheritance* durch die *private*-Vererbung erreicht werden. In Smalltalk bedeutet Vererbung immer *implementation inheritance*.

Eine feinere Differenzierung der Vererbungsstrukturen erfolgt bei /Atkinson et al. 89/.

#### ■ *substitution inheritance*

Eine Klasse `Derived` ist Unterklasse der Klasse `Base`, wenn auf den Objekten von `Derived` mehr Operationen ausgeführt werden können als auf den Objekten von `Base`. Dann kann jedes Objekt von

Base durch ein Objekt von *Deri ved* ersetzt werden. Diese Art der Vererbung basiert auf dem Verhalten und nicht auf den Daten.

- *inclusion inheritance*

Eine Klasse *Deri ved* ist Unterklasse von *Base*, wenn jedes Objekt von *Deri ved* ebenfalls ein Objekt von *Base* ist. Dieser Typ von Vererbung basiert auf der Struktur und nicht auf den Operationen.

- *constraint inheritance*

Diese Form der Vererbung ist ein Sonderfall der *inclusion inheritance*. Eine Klasse *Deri ved* ist Unterklasse der Klasse *Base*, wenn sie zwar keine zusätzlichen Attribute und/oder Operationen besitzt, aber alle Objekte von *Deri ved* eine oder mehrere zusätzliche Restriktionen erfüllen.

- *specialization inheritance*

Eine Klasse *Deri ved* ist Unterklasse der Klasse *Base*, wenn Objekte der Klasse *Deri ved* auch Objekte der Klasse *Base* sind, die zusätzliche Informationen enthalten.

abstrakte Klasse

Im Entwurf enthalten viele Vererbungsstrukturen abstrakte Klassen. Der Hauptzweck einer **abstrakten Klasse** ist, gemeinsame Eigenschaften und Operationen für ihre Unterklassen verfügbar zu machen. Abstrakte Klassen können auf zwei Arten konzipiert werden /Wirfs-Brock 90/:

- Sie enthält »normale« Operationen, die durch eine Implementierung realisiert werden und von den Unterklassen geerbt werden.
- Sie enthält eine oder mehrere – abstrakte – Operationen, die in den Unterklassen redefiniert und implementiert werden.

**Vererbung in C++**

### Sichtbarkeit der Oberklasse

- Eine Oberklasse bzw. Basisklasse kann in C++ als *public*, *protected* oder *private* deklariert werden.

- `class D: public B{}`

Die Sichtbarkeit aller Attribute und Operationen von *B* wird unverändert von *D* geerbt.

- `class D: protected B {}`

Die Sichtbarkeit von *public*-Attributen bzw. -Operationen aus *B* wird in *D* zu *protected*. Alle anderen Sichtbarkeiten gelten analog zu *public*-Vererbung.

- `class D: private B {}`

Die Sichtbarkeit von *public*- und *protected*-Attributen bzw. Operationen wird in *D* zu *private*.

### Mehrfachvererbung

- C++ realisiert die Mehrfachvererbung.

```
class D: public B1, private B2
{...}
```

### Sichtbarkeit der Oberklasse

- In Java bleiben die Sichtbarkeiten der Oberklasse in der Unterklasse stets erhalten.
- Die Vererbung wird mittels `extends` definiert.

```
class B
{ ... }
class D extends B
{ ... }
```

### Mehrfachvererbung

- In Java ist das Konzept der Mehrfachvererbung für Klassen *nicht* realisiert.
- Als Ersatzmechanismus dienen die Schnittstellen (*interfaces*), wobei es jedoch signifikante Unterschiede gibt. Eine Schnittstelle kann nur abstrakte Operationen und Konstanten weitergeben, nicht aber Attribute und Implementierungen von Operationen. Deshalb ist sie mit der Mehrfachvererbung *nicht* gleichzusetzen.

```
class B1 { ... }
interface B2 { ... }
interface B3 { ... }
class D extends B1 implements B2, B3
{ ... }
```

### Vererbung in Java

## 6.7 Paket

Das Entwurfmodell ist wesentlich umfangreicher als das Analysemodell. Dementsprechend ist das Konzept des Pakets besonders wichtig für den Entwurf. Pakete dienen dazu (Modell-) Elemente – insbesondere Klassen – zu Gruppen zusammenzufassen und als Ganzes zu behandeln. Desweiteren unterstützen Pakete die Darstellung von alternativen Entwürfen oder von Entwürfen für verschiedene Plattformen. Pakete können ineinandergeschachtelt werden, was eine Modellierung des Systems auf verschiedenen Abstraktionsebenen ermöglicht /Booch et al. 98/.

Die in einem Paket enthaltenen Elemente können entweder in Textform oder als grafisches Symbol eingetragen werden (Abb. 6.7-1).

Sichtbarkeiten

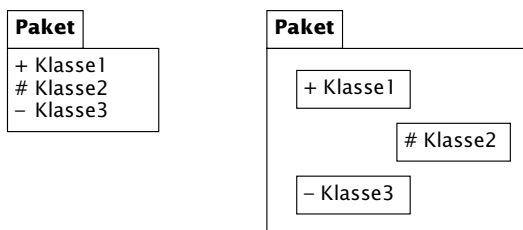


Abb. 6.7-1:  
Notation für Pakete  
und darin enthaltene Klassen

## LE 12 6 Konzepte und Notation für OOD

Für die Elemente kann die Sichtbarkeit analog zu den Attributen und Operationen einer Klasse definiert werden. Es gelten:

+ Das *public*-Element ist für alle Pakete sichtbar, die das betreffende Paket importieren.

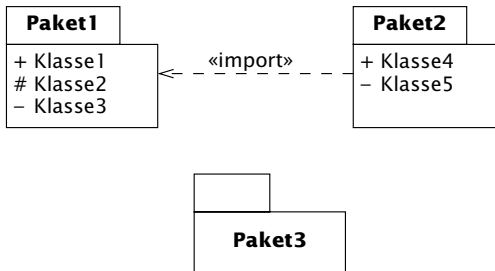
# Das *protected*-Element ist für alle Pakete sichtbar, die das betreffende Paket spezialisieren.

- Das *private*-Element ist nur in dem betreffenden Paket sichtbar.

Die Bedeutung dieser Sichtbarkeiten wird im folgenden noch genauer erläutert. Wenn ein Element in einem Paket A sichtbar ist, dann ist es auch in allen Paketen A1, A2 sichtbar, die in A enthalten sind.

*import* Zwischen zwei Paketen kann eine *import*-Beziehung definiert werden. In Abb. 6.7-2 importiert Paket2 das Paket1. Das bedeutet, daß Paket2 die *public*-Klasse Klasse1 sieht. Für Paket3 sind sie unsichtbar, weil keine *import*-Beziehung besteht. Die *import*-Beziehung ist *nicht* transitiv.

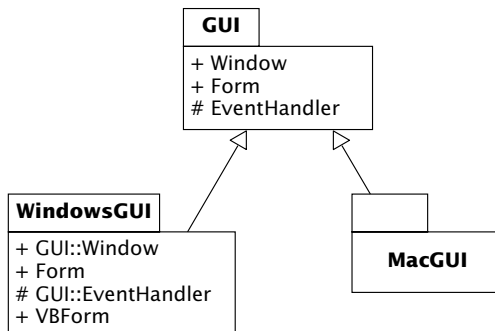
Abb. 6.7-2:  
*import*-Beziehung  
zwischen Paketen



Paketvarianten

Das Vererbungssymbol wird benutzt, um Paket-Varianten darzustellen. In Abb. 6.7-3 »erben« die Pakete *WindowsGUI* und *MacGUI* die *protected*- und *public*-Elemente des allgemeineren Pakets *GUI*. Analog zur Vererbung bei Klassen kann das spezialisierte Paket geerbte Elemente neu definieren und zusätzliche Elemente hinzufügen. Beispielsweise erbt das Paket *WindowsGUI* von *GUI* die Klassen *GUI::Window* und *GUI::EventHandler*, überschreibt die Klasse *Form* und fügt die Klasse *VBForm* hinzu. Ein spezialisiertes Paket kann

Abb. 6.7-3:  
Spezialisierung  
eines Pakets  
/Booch et al. 98/





überall dort benutzt werden, wo das allgemeinere Paket verwendet werden kann.

Der Paketname kann durch einen darüberstehenden Stereotypen ergänzt werden, um die Bedeutung des Pakets im System deutlich zu machen. Die UML definiert für Pakete eine Reihe von Standard-Stereotypen. Dazu gehören unter anderem:

Stereotypen

- *subsystem*: Das betreffende Paket modelliert ein unabhängiges Teilsystem.
- *system*: Das betreffende Paket repräsentiert das gesamte System.

### ... C++

In C++ gibt es das Konzept des Pakets nicht.

### ... Java

- In Java ist das Paket (*package*) standardmäßig implementiert.
- Um ein Paket zu erstellen, muß im Projektverzeichnis ein Unterverzeichnis erstellt werden, dessen Name mit dem Paketnamen übereinstimmt.
- In dieses Verzeichnis kommen alle Klassen (je als separate Datei) des Pakets. Jede Datei beginnt mit der Zeile `package Paketname; .`
- Ein vorhandenes Paket wird mit `import Paketname.*;` in die gewünschte Datei eingebunden.
- Soll nur eine Klasse eines Pakets eingebunden werden, benötigt man den Befehl `import Paketname.Klassenname;`



**Paket in ...**

## 6.8 Szenario

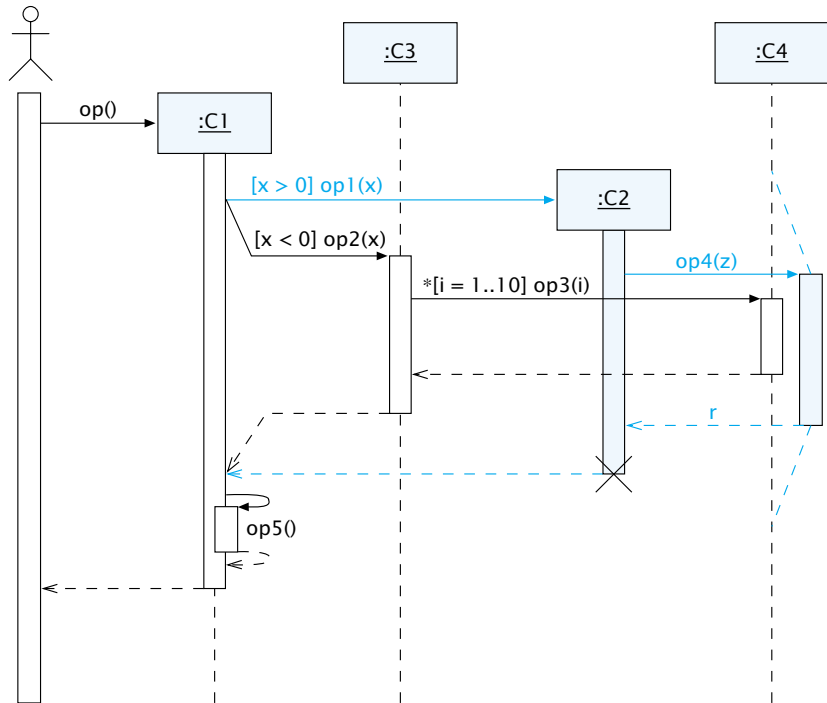
Die Zusammenarbeit von Objekten ist anhand von C++ und Java-Programmen schwer nachzuvollziehen, da oft eine ganze Kette von Operationen aus verschiedenen Klassen ausgeführt wird. Das Klassendiagramm zeigt nur eine Menge von Schnittstellen, nicht jedoch das dynamische Verhalten. Die Zusammenarbeit der Objekte kann deshalb nur mittels geeigneter Szenarios beschrieben werden. Setzen Sie daher für komplexe dynamische Zusammenhänge konsequent Sequenz- und Kollaborationsdiagramme ein. Ich halte Sequenzdiagramme für besser geeignet, wenn es darum geht, komplette Szenarios darzustellen und die Reihenfolge der Operationsaufrufe deutlich gemacht werden soll. Das Kollaborationsdiagramm wähle ich, wenn es um Ausschnitte von Szenarios geht, wenn die direkte Zusammenarbeit von Objekten über ihre Verbindungen ausgedrückt wird und wenn die Reihenfolge der Operationsaufrufe erst später festgelegt werden soll.

Die Notation des Sequenzdiagramms wird im Entwurf geringfügig erweitert (Abb. 6.8-1). Eine Verzweigung des Kontrollflusses tritt auf, wenn mehrere Botschaftspfeile vom selben Punkt ausgehen. Je-

Notation  
Sequenz-  
diagramme

## LE 12 6 Konzepte und Notation für OOD

Abb. 6.8-1:  
Notation des  
Sequenz-  
diagramms



der Pfeil ist mit einer Bedingung (*guard condition*) beschriftet. Die Objektlinie kann in zwei oder mehrere Linien verzweigen, die zu einem späteren Zeitpunkt wieder zusammengeführt werden.

Desweiteren kann ein Botschaftspfeil – wie in der Analyse – mit einer Bedingung beschriftet werden. Die Botschaft wird nur dann gesendet, wenn die Bedingung erfüllt ist. Bei einer Iteration, die mit \* oder mit \*[Bedingung] gekennzeichnet ist, wird die Botschaft wiederholt gesendet.

Die Verzweigung des Kontrollflusses ermöglicht es, mehrere Abläufe in einem Sequenzdiagramm zusammenzufassen und die Anzahl der Diagramme zu reduzieren. Ein Nachteil ist, daß leicht unübersichtliche Diagramme entstehen. Es ist daher die Aufgabe des Entwerfers, dieses Element sinnvoll einzusetzen.

Die Operationen des Sequenzdiagramms entsprechen den Operationen des zugehörigen C++ oder Java-Programms. Zusätzlich zu den Operationsnamen können bei Bedarf deren (aktuelle) Parameter angegeben werden. Die gestrichelten Rückgabepfeile sollten im Entwurf immer angegeben werden, wenn es zur Verständlichkeit des Diagramms – insbesondere bei verzweigten Lebenslinien – notwendig ist. Ich beschrifte sie optional mit dem Ergebnisparameter. Da wir hier nur sequentielle Systeme betrachten, wird eine Operation niemals unterbrochen. Jede Operation, d.h. die Reaktion auf eine Botschaft, wird beendet und liefert ggf. eine Antwort zurück. Sen-

det eine Operation  $op()$  die Botschaften  $op1()$  und  $op2()$  an andere Objekte, so müssen die aktivierten Operationen beendet sein, bevor die Operation  $op()$  terminiert werden kann.

Wie in der Analyse wird die Aktivierungsdauer einer Operation (*focus of control*) durch ein schmales Rechteck angegeben. Auch für das Erzeugen und Löschen von Objekten und den rekursiven Aufruf von Operationen wird die Notation der Analyse unverändert übernommen.

Abb. 6.8-1 zeigt die vollständige Notation für ein Sequenzdiagramm /UML 97/. Die Operation  $op()$  erzeugt ein Objekt der Klasse C1. Bei der Ausführung von  $op()$  verzweigt der Kontrollfluß. Gilt  $x > 0$  (blau eingetragener Pfad), dann erzeugt die Botschaft  $op1(x)$  ein Objekt der Klasse C2. Die Operation  $op1(x)$  schickt dann die Botschaft  $op4(z)$  an ein vorhandenes Objekt von C4 und erhält den Ergebniswert  $r$ . Zum Schluß löscht die Operation  $op1(x)$  das erzeugte Objekt von C2 wieder (transientes Objekt).

Wenn  $x < 0$  ist, dann sendet die Operation  $op()$  die Botschaft  $op2(x)$  an ein bereits existierendes Objekt von C3. Innerhalb dieser Operation wird zehn mal  $op3(i)$  – für  $i = 1$  bis 10 – auf ein Objekt von C4 angewendet.

Zum Schluß ruft die Operation  $op()$  die Operation  $op5()$  auf, die ebenfalls zur Klasse C1 gehört.



In den folgenden Lehreinheiten finden Sie mehrere – teilweise komplexe – Beispiele zu Sequenzdiagrammen. Kapitel 10.3 zeigt mittels Sequenzdiagrammen, wie die Kommunikation zwischen der GUI- und der Fachkonzept-Schicht abläuft. In den Kapiteln 10.4 und 10.5 werden diese Diagramme um die Anbindung an eine objektorientierte Datenbank bzw. flache Dateien erweitert. In Kapitel 10.6 modellieren Sequenzdiagramme den Entwurf der Datenhaltung mit einer relationalen Datenbank.

Kapitel  
10.3 bis 10.6

Um die Architektur des Systems und dessen Strukturierung in Pakete transparent zu machen, ist es manchmal sinnvoll, diese Paket-Grenzen in die Sequenzdiagramme einzutragen /IBM 97/. Das können zum Beispiel die Grenzen zwischen Client und Server sein oder zwischen GUI-, Fachkonzept- und Datenhaltungsschicht oder allgemein zwischen verschiedenen Paketen (Abb. 6.8-2).

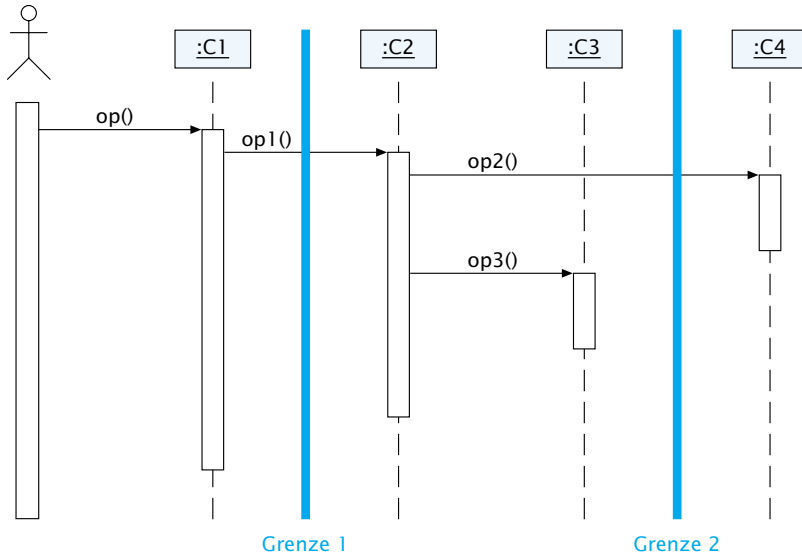
Die dynamische Modellierung mittels Sequenzdiagrammen ist eine der wichtigsten Aufgaben im Entwurf, weil sie das dynamische Verhalten von Anfang bis Ende transparent macht. Bei komplexen Anwendungen können – theoretisch – Hunderte von Sequenzdiagrammen erstellt werden. Für die Praxis – mit dem üblichen Termin- druck – ist es daher wichtig, die primären Szenarios mittels Sequenzdiagrammen zu modellieren und die Sonderfälle ggf. durch Kommentare zu beschreiben.

Während Sie bei der Erstellung des Sequenzdiagramms von vorn- herein über die Reihenfolge der Operationsaufrufe nachdenken

Kollaborations-  
diagramm

## LE 12 6 Konzepte und Notation für OOD

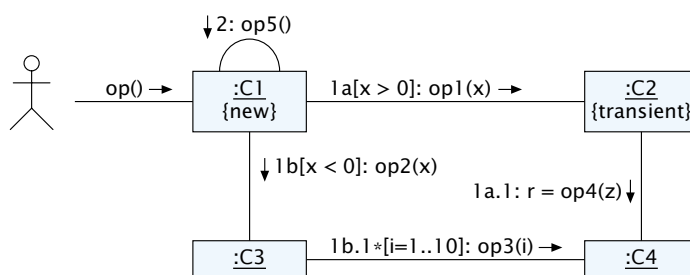
Abb.6.8-2: Teil-Systemgrenzen im Sequenzdiagramm



müssen, haben Sie beim Kollaborationsdiagramm größeren Freiraum. Die Ausgangsbasis bilden die Objekte und ihre Verbindungen untereinander. Die Reihenfolge der Operationen können Sie zum Schluß durch entsprechende Nummern hinzufügen. Diese Diagramme eignen sich daher besser, um neue Ideen zu skizzieren. Wie beim Sequenzdiagramm werden die Operationsnamen aus dem Programmcode eingetragen, ggf. mit Parametern und Ergebniswerten. Im Gegensatz zum Sequenzdiagramm beschreibt jedes Kollaborationsdiagramm immer nur eine externe Operation, die dann weitere interne Operationen aktiviert. Es beschreibt zusätzlich die Verbindungen zwischen den Objekten, die durch die Navigationsrichtungen ergänzt werden können.

Abb. 6.8-3 zeigt die Notationselemente von Kollaborationsdiagrammen. Zum besseren Vergleich beschreibt dieses Diagramm den gleichen Sachverhalt wie das Sequenzdiagramm der Abb. 6.8-1. Die externe Operation  $op()$  erhält keine Nummer. Alle Botschaften zwischen den Objekten werden nummeriert. Die Nummern 1a und 1b beschreiben die Verzweigung des Kontrollflusses. Die Nummer 1a.1

Abb. 6.8-3: Notation des Kollaborationsdiagramms



gibt an, daß diese Operation innerhalb von 1a aufgerufen wird. Innerhalb von 1b wird 10 mal die Operation 1b.1 aktiviert. Zum Schluß wird die Operation mit der Nummer 2 aktiviert. Das Objekt der Klasse C1 wird im Rahmen dieses Szenarios erzeugt und daher mit {new} gekennzeichnet. Das Objekt der Klasse C2 ist {transient}, weil es innerhalb der beschriebenen Operation sowohl erzeugt als auch gelöscht wird.

Für Kollaborationsdiagramme definiert die UML das Multi-Objekt-Symbol (*multi object*). Es repräsentiert die Menge der Objekte am *many*-Ende einer Assoziation. Das Klassendiagramm wird durch diese Objektmenge nicht verändert.

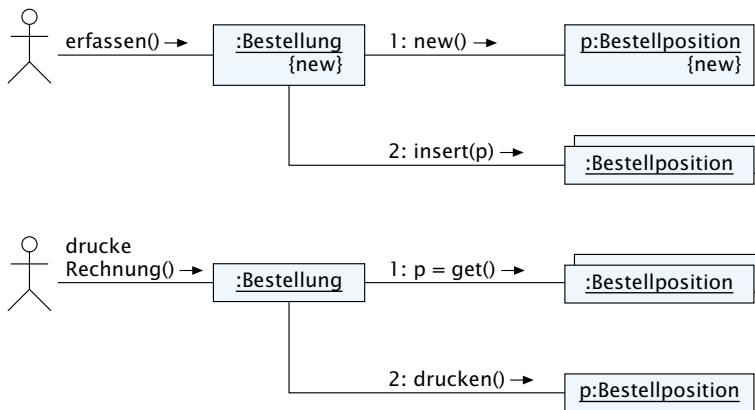
*multi object*

Damit für das *Bestellung*-Objekt in Abb. 6.8-4 die Operation *druckeRechnung()* ausgeführt werden kann, ist zuerst eine Botschaft an das Multi-Objekt zu senden, um die Objektidentität (OID) einer *Bestellposition* zu ermitteln. Dann wird die entsprechende Operation auf das *Bestellposition*-Objekt angewendet. Analog wird beim Erfassen einer neuen *Bestellung* vorgegangen. Zuerst wird ein neues *Bestellposition*-Objekt erzeugt und anschließend in die Objektmenge eingefügt.

Beispiel



Abb. 6.8-4: *multi object*-Symbol



In den Programmiersprachen gibt es keine Notation, um die Zusammenarbeit der Objekte zu dokumentieren, sondern ein Szenario kann nur aus der Reihenfolge der Operationsaufrufe gewonnen werden.

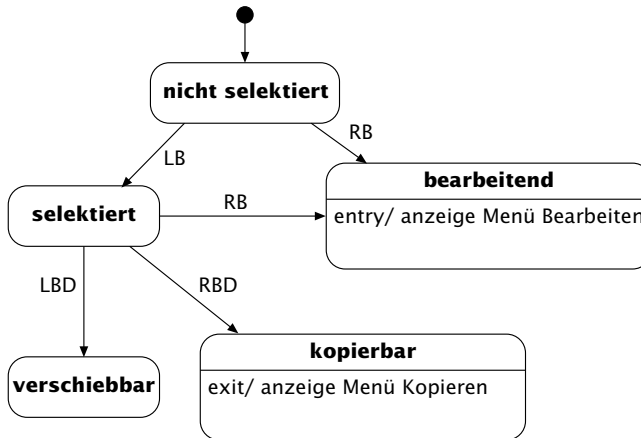
**Szenarios in C++ und Java**

## 6.9 Zustandsautomat

Benutzungs-  
oberfläche

Der Zustandsautomat kann im Entwurf besonders vorteilhaft zur Modellierung der Benutzungsoberfläche eingesetzt werden. Abb. 6.9-1 zeigt, wie ein Dokument – z.B. innerhalb einer Baumstruktur – mit der Maus manipuliert werden kann. In diesem Zustandsdiagramm wird jede Manipulation mit der Maus (z.B. linke Maustaste gedrückt) als Ereignis dargestellt, das einen Zustandsüber-

Abb. 6.9-1:  
Zustandsautomat  
zur Manipulation  
eines Dokuments



LB : left (mouse) button  
 RB : right (mouse) button  
 LBD : left (mouse) button down  
 RBD : right (mouse) button down

gang auslöst. Diese Form des Zustandsautomaten ist eine präzise Vorgabe für die Programmierung und erleichtert außerdem den Test und die spätere Wartung.

Lebenszyklus

Die Operationen einer Klasse werden im Entwurf gegenüber der Analyse normalerweise erweitert. Daher sind die entsprechenden Lebenszyklen nun aus Entwurfssicht zu überarbeiten und mit den entsprechenden Operationen zu ergänzen. Dazu ist die aus der Analyse bekannte Notation einzusetzen.

einfache  
Realisierung

Ein Zustandsautomat kann nicht direkt in eine Programmiersprache umgesetzt werden. Für einfache Automaten bietet sich folgende Realisierung an. Jede Klasse mit einem Lebenszyklus erhält im Entwurf ein *private*-Attribut `classState`. In diesem Attribut wird der aktuelle Zustand des Objekts gespeichert. Jede Operation, die im Lebenszyklus aufgeführt ist, muß dieses Attribut abfragen, bevor sie ihre Verarbeitung durchführt. Ist mit dieser Operation ein Zustandswechsel verbunden, dann muß sie das Zustandsattribut aktualisieren. Jede Klasse, die einen Objekt-Lebenszyklus besitzt, stellt eine ereignis-interpretierende Operation zur Verfügung, die eintreffende Ereignisse interpretiert und ggf. eine entsprechende

Verarbeitung auslöst /Booch 94/. Eine Verarbeitung kann die Ausführung einer Operation, das Auslösen eines anderen Ereignisses, das Starten einer Aktivität oder das Beenden einer Aktivität sein.



Für die Realisierung eines komplexen Lebenszyklus bietet sich das **Zustandsmuster** (*state pattern*) von /Gamma et al. 95/ (vergleiche Kapitel 7) an. Es ist insbesondere dann sinnvoll, wenn die Operationen in Abhängigkeit vom jeweiligen Zustand verschiedene Teilaufgaben ausführen, d.h. wenn sie große Auswahlanweisungen enthalten. Abb. 6.9-2 zeigt die prinzipielle Realisierung eines Lebenszyklus durch dieses Muster. Wenn ein Objekt der – grauen – OOA-Klasse auf die Botschaft bearbeiten mit der gleichnamigen Operation reagiert, dann hängt sein aktuelles Verhalten vom jeweiligen Zustand ab. Der Zustandsautomat zeigt, daß es nur in den Zuständen 1 und 2 darauf reagieren kann, während die Botschaft im Zustand3 ignoriert wird. Im Zustand1 führt bearbeiten() Schritte aus, die unter bearbeiten1 zusammengefaßt sind, im Zustand2 die Verarbeitung bearbeiten2. Dabei kann es sich um einen Operationsaufruf oder um mehrere Arbeitsschritte handeln.

Zustandsmuster  
Kapitel 7

Im – blauen – Entwurfsmodell erhält jede Klasse mit einem Objekt-Lebenszyklus eine abstrakte Klasse. Jede ihrer Unterklassen realisiert einen Zustand und implementiert alle Operationen und Attribute, die für diesen Zustand relevant sind. Bei allen anderen tritt eine entsprechende Ausnahmebehandlung auf. Das entsprechende OOD-Objekt besitzt eine Verbindung zu seinem aktuellen

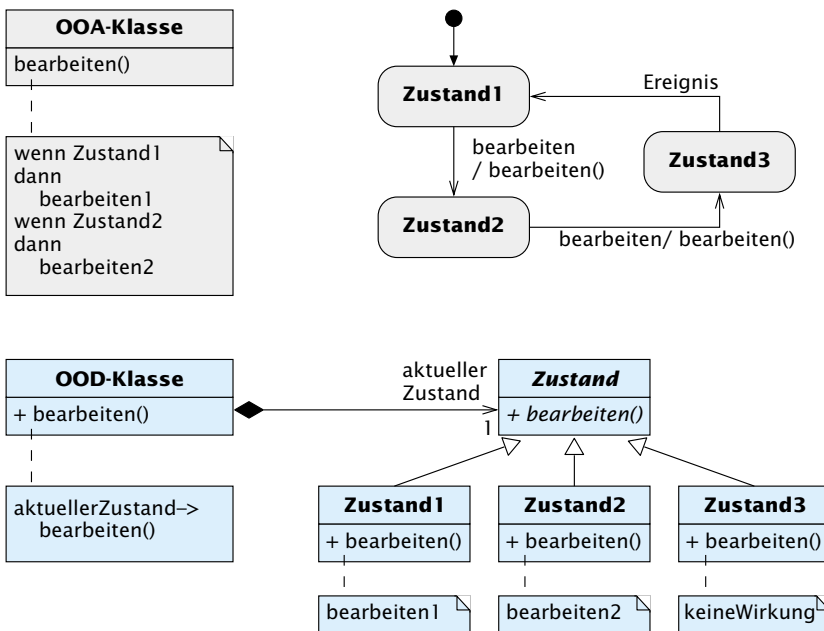


Abb. 6.9-2:  
Realisierung  
des Lebenszyklus  
durch das  
Zustandsmuster

Zustandsobjekt. Der Vorteil dieses Musters ist, daß das komplette Verhalten eines Zustands in einer Klasse gekapselt ist. Das erlaubt es, neue Zustände und Zustandsübergänge leicht hinzuzufügen. Nachteilig ist dagegen, daß sich die Anzahl der Klassen erhöht.

Aktivitätsdiagramm

Kapitel 2.11

Das **Aktivitätsdiagramm** (*activity diagram*) – als Sonderfall des Zustandsdiagramms – dient dazu, die interne Verarbeitung zu spezifizieren, wobei jeder Zustand einen Schritt eines Algorithmus beschreibt. Im Entwurf kann es sinnvoll zur Beschreibung von komplexen Operationen eingesetzt werden. Achten Sie bei der Verwendung von Aktivitätsdiagrammen immer darauf, daß sie nicht zu Programmablaufplänen »entarten«.



### Zustandsautomat in C++

#### Einfache Realisierung

- Der Zustand wird durch einen Aufzählungstyp realisiert.

```
class Schublade
{private:
    enum State {offen, zuUnverschlossen, zuVerschlossen};
    State classState;
public:
    void oeffnen();
    ...
};
void Schublade::oeffnen()
{if (classState == zuUnverschlossen)
    classState = offen; ...}
```

### Zustandsautomat in Java

#### Einfache Realisierung

- In Java gibt es keinen benutzerdefinierbaren Aufzählungstyp.
- Die Zustände werden daher als Integer-Konstanten definiert.

```
class Schublade
{final static int OFFEN = 1;
  final static int ZU_UNVERSCHLOSSEN = 2;
  final static int ZU_VERSCHLOSSEN = 3;
  private int classState;
  public void oeffnen()
  {if (classState == ZU_UNVERSCHLOSSEN)
      classState = OFFEN;
    ...
  }
}
```

#### Abstrakte Klasse (*abstract class*)

Von einer abstrakten Klasse können keine Objekte erzeugt werden. Die abstrakte Klasse spielt eine wichtige Rolle in →Vererbungsstrukturen, wo sie die Gemeinsamkeiten einer Gruppe von Unterklassen definiert. Damit eine abstrakte Klasse verwendet werden kann, muß von ihr zunächst eine Unterklasse abgeleitet werden. Eine abstrakte Klas-

se kann auf zwei verschiedene Arten konzipiert werden: Mindestens eine Operation wird nicht spezifiziert bzw. implementiert, d.h. der Rumpf ist leer. Es wird nur die Signatur dieser Operation angegeben. Man spricht dann von einer abstrakten Operation. Alle Operationen können – wie auch bei einer konkreten Klasse – vollständig spezifiziert bzw. implementiert wer-





den. Es ist jedoch nicht beabsichtigt, von dieser Klasse Objekte zu erzeugen.

**Aktivitätsdiagramm (*activity diagram*)** Ein Aktivitätsdiagramm ist der Sonderfall eines Zustandsdiagramms, bei dem – fast – alle Zustände mit einer Verarbeitung verbunden sind. Ein Zustand wird verlassen, wenn die Verarbeitung beendet ist. Außerdem ist es möglich, eine Verzweigung des Kontrollflusses zu spezifizieren und zu beschreiben, ob die Verarbeitungsschritte in festgelegter oder beliebiger Reihenfolge ausgeführt werden können.

**Dynamisches Binden (*dynamic binding*)** →spätes Binden

**Einfachvererbung (*single inheritance*)** Bei der Einfachvererbung besitzt jede Unterklasse genau eine direkte Oberklasse. Es entsteht eine Baumstruktur.

**Kollaborationsdiagramm (*collaboration diagram*)** Ein Kollaborationsdiagramm beschreibt die Objekte und die Verbindungen zwischen diesen Objekten. An jede Verbindung (*link*) kann eine Botschaft in Form eines Pfeiles angetragen werden. Die Reihenfolge und Verschachtelung der Operationen wird durch eine hierarchische Numerierung angegeben.

**Mehrfachvererbung (*multiple inheritance*)** Bei der Mehrfachvererbung kann jede Klasse mehrere direkte Oberklassen besitzen. Sie bildet einen azyklischen Graphen, der mehr als eine Wurzel haben kann (Netzstruktur). Bei der Mehrfachvererbung können Namenskonflikte auftreten.

**Paket (*package*)** Ein Paket faßt Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene auszudrücken. Pakete können im Paketdiagramm dargestellt werden.

**Polymorphe Operation** Eine polymorphe Operation ist eine Operation, die erst zur Ausführungszeit an ein bestimmtes Objekt gebunden wird. Man spricht vom →späten Binden (*late binding*) bzw. vom dynamischen Binden.

**Polymorphismus (*polymorphism*)** Ein Name kann Objekte verschiedener Klassen bezeichnen. Jedes Objekt, das

durch diesen Namen bezeichnet wird, kann auf die gleiche Botschaft auf seine eigene Art und Weise reagieren. Polymorphismus und →spätes Binden sind untrennbar verbunden.

**Szenario (*scenario*)** Ein Szenario ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind. Diese Schritte sollen das Hauptziel des Akteurs realisieren und ein entsprechendes Ergebnis liefern. Szenarios werden mit Hilfe von →Sequenz- und →Kollaborationsdiagrammen dokumentiert.

**Sequenzdiagramm (*sequence diagram*)** Ein Sequenzdiagramm besitzt zwei Dimensionen. Die Vertikale repräsentiert die Zeit und auf der Horizontalen werden die Objekte angetragen. In das Diagramm werden die Botschaften eingetragen, die zum Aktivieren der Operationen dienen.

**Spätes Binden (*late binding*)** Beim späten Binden wird erst zur Ausführungszeit bestimmt, welche →polymorphe Operation auf ein Objekt angewendet wird. Man spricht auch von dynamischem Binden. Das Gegenstück zum späten Binden ist das frühe Binden, das zur Übersetzungszeit stattfindet.

**Überschreiben (*overriding*)** Von Überschreiben bzw. Redefinition spricht man, wenn eine Unterklasse eine geerbte Operation der Oberklasse – unter dem gleichen Namen – neu implementiert. Beim Überschreiben müssen die Anzahl und Typen der Ein-/Ausgabeparameter gleichbleiben. Bei der Implementierung der überschriebenen Operation wird im allgemeinen die entsprechende Operation der Oberklasse aufgerufen.

**Vererbung (*generalization, inheritance*)** Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und Assoziationen der Basisklasse. Operationen der Basisklasse dürfen redefiniert werden. Es entsteht eine Klassenhierarchie oder Vererbungsstruktur. Außer der →Einfachvererbung, bei der Klassen eine Baumstruktur bilden, gibt es die →Mehrfachvererbung (Netzstruktur).

## LE 12 Glossar/Zusammenhänge/Aufgaben

**Zustandsautomat (finite state machine)** Ein Zustandsautomat besteht aus Zuständen und Transitionen. Er hat einen Anfangszustand und kann einen Endzustand besitzen.

**Zustandsmuster (state pattern)** Das Zustandsmuster ist ein Entwurfsmu-

ster, mit dem Objekt-Lebenszyklen des OOA-Modells systematisch in ein OOD-Klassendiagramm umgesetzt werden können. Es ist insbesondere für die Realisierung komplexer →Zustandsautomaten gedacht.

Der **Polymorphismus** ermöglicht es, flexible Programme zu entwickeln. Im Gegensatz zur Analyse tritt im Entwurf häufig Vererbung auf, wobei außer der **Einfachvererbung** auch die **Mehrfachvererbung** vorkommen kann. Zusammenhängende Funktionsabläufe können nur mittels **Szenarios** dokumentiert werden. Daher sind **Sequenz-** und **Kollaborationsdiagramme** im Entwurf von besonderer Bedeutung. Der **Zustandsautomat** kann vorteilhaft zur Spezifikation der Benutzungsoberfläche eingesetzt werden. Komplexe Lebenszyklen aus der Analyse können gut mit dem **Zustandsmuster** in den Entwurf transformiert werden.



Aufgabe  
15 Minuten

**1 Lernziel: Anwenden des Polymorphismus.**

Erstellen Sie für folgende Problemstellung ein Klassendiagramm und skizzieren Sie die Funktionalität mittels Programmcode oder Pseudocode.

Literaturstellen (Bücher und Zeitschriftenartikel) sind nach Autoren sortiert in einer gemeinsamen Liste abzulegen. Für jedes Buch sind der Autor, der Titel, der Ort und das Erscheinungsjahr zu speichern, für jeden Artikel der Autor, der Titel, die Zeitschrift, die Ausgabe und Seitenangaben. Folgende Funktionalität ist zu realisieren. Jede Literaturstelle ist zu erfassen und nach Autoren sortiert in der Liste zu speichern. Alle Literaturstellen sollen nach Autoren sortiert ausgegeben werden.

Aufgabe  
20 Minuten

**2 Lernziele: Kollaborationsdiagramme und Sequenzdiagramme erstellen können.**

Erstellen Sie Kollaborationsdiagramme und Sequenzdiagramme für folgende Problemstellungen:

B\* ei nB;

C\* ei nC;

D\* ei nD;

```
//Teil a
void A::ausgeben()
{ for (i=1; i<=10; i++)
  ei nB->ausgeben();
  for (i=1; i<=10; i++)
    ei nC->ausgeben();
}

//Teil b
void A::verarbeiten()
{ ei nB->verarbeiten1();
  ei nC->verarbeiten2();
}
void B::verarbeiten()
{ ei nC->verarbeiten1();
  D* noch einD = new D;
}
void C::verarbeiten2()
{ if (Bedingung)
  ei nB->verarbeiten()
  else
  ei nD->verarbeiten();
}
```

**3 Lernziel: Anwenden des Zustandsmusters.**

Realisieren Sie den Lebenszyklus der Abb. LE12-A3 mit dem Zustandsmuster.

Aufgabe  
15 Minuten

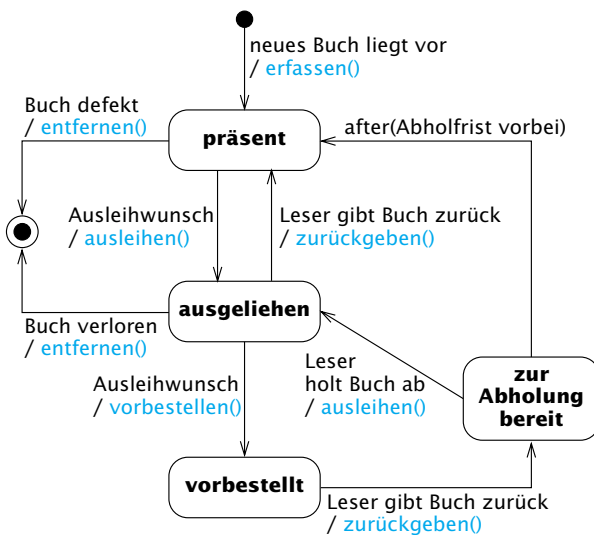


Abb. LE12-A3:  
Zustandsautomat  
der Klasse Buch

Buch
erfassen() ausleihen() zurückgeben() vorbestellen() entfernen()

# 7 Entwurfsmuster



- Entwurfsmuster, *Frameworks* und Klassenbibliotheken unterscheiden können.
- Wichtige Entwurfsmuster kennen und erklären können, wo sie eingesetzt werden.
- Entwurfsmuster bei der Modellierung einsetzen können.
- Entwurfsmuster in einem OOD-Modell erkennen können.

verstehen

anwenden



Die Voraussetzungen für diese Lehreinheit sind die objektorientierten Konzepte und die UML-Notation, wie sie in den Kapiteln 2 und 6 beschrieben sind.

- i 7.1 Entwurfsmuster, *Frameworks*, Klassenbibliotheken 282
- 7.2 Fabrikmethode-Muster 286
- 7.3 *Singleton*-Muster 287
- 7.4 Kompositum-Muster 289
- 7.5 Proxy-Muster 291
- 7.6 Fassaden-Muster 293
- 7.7 Beobachter-Muster 295
- 7.8 Schablonenmethode-Muster 297

## 7.1 Entwurfsmuster, *Frameworks*, Klassenbibliotheken

Definition Ein **Entwurfsmuster** (*design pattern*) gibt eine bewährte, generische Lösung für ein immer wiederkehrendes Entwurfsproblem an, das in bestimmten Situationen auftritt.

Das Standardwerk über Entwurfsmuster wurde von E. Gamma und drei weiteren Autoren /Gamma et al. 95/ geschrieben. In der Literatur werden die Autoren dieses Werks auch als GoF (*Gang of Four*) bezeichnet. In diesem Kapitel werden einige Entwurfsmuster aus /Gamma et al. 95/ vorgestellt, die in späteren Lehreinheiten beim objektorientierten Entwurf verwendet werden. Weitere Informationen zu Entwurfsmustern finden sich in /Buschmann et al. 96/, /Larman 98/ und den PLoP-Bänden (*Pattern Languages of Program Design*)/Coplien et al. 95/, /Vlissides et al. 96/ und /Martin et al. 97/.

Beschreibung eines Musters Allgemein betrachtet besitzt ein Muster vier grundlegende Elemente:

- **Name des Musters**

Er beschreibt ein Entwurfsproblem, seine Lösung und Konsequenzen mit einem oder zwei Wörtern. Die Namen der Muster erweitern das Entwurfsvokabular.

- **Problembeschreibung**

Sie gibt an, wann das Muster anwendbar ist. Das Problem und der Kontext werden erklärt. Es können auch spezifische Entwurfsprobleme beschrieben werden.

- **Lösungsbeschreibung**

Die Lösung gibt keinen konkreten Entwurf und keine konkrete Implementierung an, da ein Muster wie eine Schablone in verschiedenen Situationen angewendet werden kann. Ein Muster stellt eine abstrakte Beschreibung des Entwurfsproblems dar und beschreibt, wie eine allgemeine Anordnung der Klassen bzw. Objekte aussehen kann, um das Problem zu lösen.

- **Konsequenzen**

Die Kenntnis der Konsequenzen ist wichtig, um Entwurfsalternativen zu evaluieren und um das Kosten-Nutzen-Verhältnis von Mustern abzuwägen. Die Konsequenzen beziehen sich oft auf Zeit- versus Speichereffizienz. Sie können sich aber auch auf Sprach- und Implementierungseigenschaften beziehen. Weitere Konsequenzen sind die Auswirkungen auf Flexibilität, Erweiterbarkeit und Portabilität.

Klassifikation von Mustern /Gamma et al.95/ klassifizieren Entwurfsmuster nach den Aufgaben in Erzeugungs-, Struktur- und Verhaltensmuster. Des weiteren werden klassenbasierte und objektbasierte Muster unterschieden. **Klassenbasierte Muster** behandeln Beziehungen zwischen Klassen. Sie werden durch Vererbungen ausgedrückt und zur Über-

setzungszeit festgelegt. **Objektbasierte Muster** beschreiben Beziehungen zwischen Objekten, die zur Laufzeit geändert werden können. Auch objektbasierte Muster benutzen bis zu einem gewissen Grad die Vererbung.

**Erzeugungsmuster** (*creational patterns*) helfen, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. Ein klassenbasiertes Erzeugungsmuster verwendet Vererbung, um die Klasse des zu erzeugenden Objekts zu variieren. Ein objektbasiertes Erzeugungsmuster delegiert die Erzeugung an ein anderes Objekt.

Erzeugungsmuster

**Strukturmuster** (*structural patterns*) befassen sich damit, wie Klassen und Objekte zu größeren Strukturen zusammengesetzt werden. Ein klassenbasiertes Strukturmuster benutzt Vererbungen, um Schnittstellen und Implementierungen zusammenzuführen. Ein einfaches Beispiel ist die Mehrfachvererbung, die zwei oder mehr Klassen in einer neuen Klasse vereint. Dieses Muster ist besonders hilfreich, um unabhängig voneinander entwickelte Bibliotheken zusammenarbeiten zu lassen. Objektbasierte Strukturmuster beschreiben dagegen Wege, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen. Diese Muster ermöglichen eine zusätzliche Flexibilität gegenüber den Klassenmustern, weil sich bei Objektstrukturen die Struktur zur Laufzeit ändern kann.

Strukturmuster

**Verhaltensmuster** (*behavioral patterns*) befassen sich mit der Interaktion zwischen Objekten und Klassen. Sie beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind. Sie lenken die Aufmerksamkeit weg vom Kontrollfluß hin zu der Art und Weise, wie die Objekte interagieren. Klassenbasierte Verhaltensmuster verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen. Objektbasierte Verhaltensmuster verwenden Aggregation bzw. Komposition anstelle von Vererbung.

Verhaltensmuster

### Klassenbibliotheken

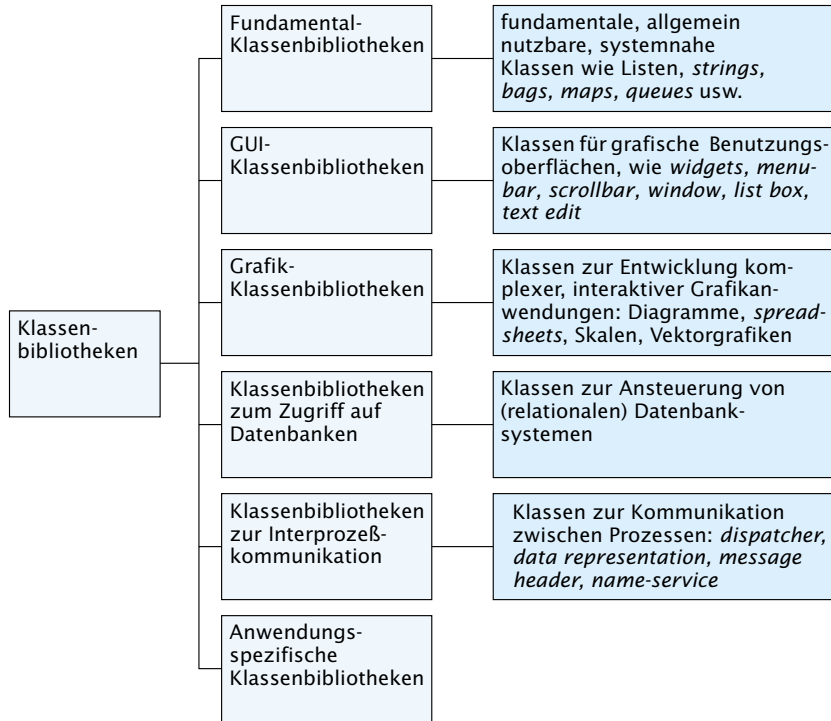
Eine **Klassenbibliothek** ist eine organisierte Sammlung von Klassen, aus denen der Entwickler nach Bedarf Einheiten verwendet, d.h. Objekte dieser Klassen definiert und Operationen darauf anwendet oder Unterklassen bildet. Klassenbibliotheken können unterschiedliche Topologien besitzen. Es sind zahlreiche Klassenbibliotheken für unterschiedliche Anwendungsgebiete erhältlich (Abb. 7.1-1). Auch die Compilerhersteller statten ihre Programmierumgebungen immer stärker mit Klassenbibliotheken aus.

Definition

Klassenbibliotheken können unterschiedliche Topologien besitzen. Bei einer Baum-Topologie existiert eine gemeinsame Wurzelklasse (Smalltalk-ähnliche Klassenstruktur). Sie wird insbesondere bei GUI-Bibliotheken verwendet. Eine Wald-Topologie liegt vor, wenn die Bibliothek aus mehreren Baumhierarchien besteht. Ihr Vorteil liegt in einer flacheren Vererbungshierarchie im Vergleich

Topologien

Abb. 7.1-1:  
Anwendungsgebiete  
von Klassen-  
bibliotheken  
/Balzert 96/



zur Baum-Topologie. Sie wird bei Fundamentalklassen und bei Klassen zur Steuerung von GUIs verwendet. Bei einer Baustein-Topologie handelt es sich um unabhängige Klassen. Hier wird anstelle der Vererbung das Konzept der generischen Klasse zur spezifischen Anpassung verwendet. Bei den Klassenbibliotheken lassen sich zwei Arten unterscheiden:

- »Einfache« Klassenbibliotheken und
- *Frameworks*.

»Einfache« Klassenbibliotheken erzwingen keine bestimmte Anwendungsarchitektur. Sie ermöglichen Code-Wiederverwendung und sind das objektorientierte Äquivalent zu Funktionsbibliotheken. Das Gegenstück sind die *Frameworks*, auf die wir noch genauer eingehen.

### Frameworks

Definition Ein **Framework** besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und – insbesondere – aus abstrakten Klassen, die Schnittstellen definieren. Die abstrakten Klassen enthalten sowohl abstrakte als auch konkrete Operationen. Im allgemeinen wird vom Anwender des *Frameworks* erwartet, daß er Unterklassen definiert, um das

*Framework* zu verwenden und anzupassen. Diese selbstdefinierten Unterklassen empfangen Botschaften von den vordefinierten *Framework*-Klassen nach dem Hollywood-Prinzip »Don't call us, we'll call you« /Gamma et al. 95/.

*Frameworks* sind immer spezifisch auf einen Anwendungsbe-  
reich ausgelegt. Beispielsweise kann ein *Framework* die Erstellung  
grafischer Editoren unterstützen. Ein anderes *Framework* kann Sie  
bei der Erstellung von Finanzsoftware unterstützen. Sie spezialisie-  
ren ein *Framework* für eine konkrete Anwendung, indem Sie Unter-  
klassen von den abstrakten *Framework*-Klassen ableiten. *Frame-  
works* werden mittels Programmiersprachen realisiert. Sie können  
also ausgeführt und direkt wiederverwendet werden. Die Bedeu-  
tung von *Frameworks* nimmt immer mehr zu, da mit ihrer Hilfe eine  
hohe Wiederverwendbarkeit erreicht werden kann.

Eigenschaften von  
*Frameworks*

Ein *Framework* bestimmt die Architektur der Anwendung. Es de-  
finiert die Struktur der Klassen und Objekte und deren Verantwor-  
tlichkeiten, legt fest, wie Klassen und Objekte zusammenarbeiten  
und wie der Kontrollfluß aussieht. Das *Framework* legt alle diese  
Entwurfsparameter fest, damit sich der Anwendungsprogrammierer  
auf die Details der Anwendung konzentrieren kann.

Der Zweck von *Frameworks* ist die Entwurfs-Wiederverwendung,  
nicht die Code-Wiederverwendung, obwohl ein *Framework* im allge-  
meinen konkrete Unterklassen enthält, die direkt verwendet werden  
können. Diese Wiederverwendung führt zu einer Umkehrung in der  
Steuerung der Software. Wenn Sie eine »einfache« Klassenbibliothek  
verwenden, dann entwerfen Sie Ihre Anwendung und rufen aus der  
Anwendung den Code auf, den Sie wiederverwenden wollen. Bei der  
Benutzung eines *Frameworks* schreiben Sie Programmcode, der von  
den Operationen des *Frameworks* aufgerufen wird.

Wenn Sie *Frameworks* verwenden, dann liegen die Konsequenzen  
nicht nur in der schnelleren Entwicklung, sondern auch darin, daß  
alle Anwendungen ähnliche Strukturen besitzen, wodurch sie einfa-  
cher zu warten sind.

Entwurfsmuster und *Frameworks* besitzen einige Ähnlichkeiten,  
unterscheiden sich jedoch in drei Aspekten /Gamma et al. 95/:

Muster vs.  
*Framework*

- 1** Entwurfsmuster sind abstrakter als *Frameworks*. Sie werden im  
Gegensatz zu *Frameworks* nur beispielhaft durch Programmcode  
repräsentiert. Eine Anwendung von Entwurfsmustern ist daher  
keine Wiederverwendung von Programmcode, sondern ist mit ei-  
ner neuen Implementierung verbunden.
- 2** Entwurfsmuster sind kleiner als *Frameworks*. Ein typisches  
*Framework* enthält mehrere Entwurfsmuster, während die Um-  
kehrung niemals zutrifft.



- 3 Entwurfsmuster sind weniger spezialisiert als *Frameworks*. Sie sind also im Gegensatz zu *Frameworks* nicht auf einen bestimmten Anwendungsbereich beschränkt, sondern können in nahezu allen Anwendungsbereichen verwendet werden.

## 7.2 Fabrikmethode-Muster

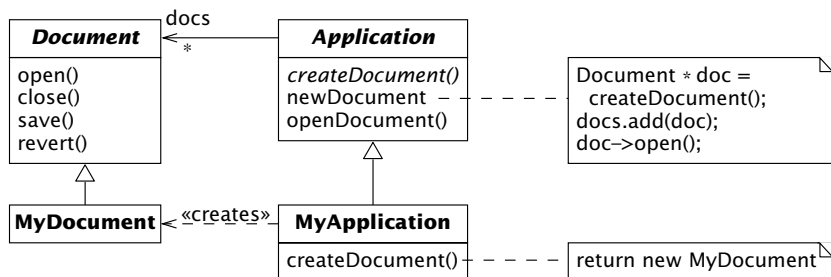
### Zweck

Das **Fabrikmethode-Muster** (*factory method*) ist ein klassenbasiertes Erzeugungsmuster. Es bietet eine Schnittstelle zum Erzeugen eines Objekts an, wobei die Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Dieses Muster wird auch als virtueller Konstruktor (*virtual constructor*) bezeichnet.

### Motivation

Wir benutzen ein *Framework* für Anwendungen, die mehrere Dokumente gleichzeitig anzeigen können. Es verwendet die beiden abstrakten Klassen `Application` und `Document` und modelliert eine Assoziation zwischen ihren Objekten (Abb. 7.2-1). Außerdem ist die Klasse `Application` für die Erzeugung neuer Dokumente zuständig. Der Softwarekonstrukteur leitet von diesen beiden Klassen seine anwendungsspezifischen Klassen ab. Wenn nun aus der Klasse `MyApplication` ein neues Objekt von `MyDocument` erzeugt werden soll, so tritt folgendes Problem auf. Das *Framework* muß Objekte erzeugen, kennt aber nur die abstrakte Oberklasse, von der es keine Objekte erzeugen darf. Dieses Muster bietet dazu folgende Lösung. Die Unterklassen von `Application` überschreiben die abstrakte Operation `createDocument()`, so daß sie ein Exemplar von `MyDocument` zurückgibt. Sobald ein Objekt von `MyApplication` erzeugt ist, kann dieses anwendungsspezifische Dokumente erzeugen, ohne deren exakte Klasse zu kennen. Die Operation `createDocument()` heißt Fabrikmethode, weil sie für die »Fabrikation« eines Objekts verantwortlich ist.

Abb. 7.2-1: Beispiel für Fabrikmethode



## Anwendbarkeit

Verwenden Sie dieses Muster, wenn

- eine Klasse die von ihr zu erzeugenden Objekte *nicht* im voraus kennen kann,
- eine Klasse benötigt wird, deren Unterklassen selber festlegen, welche Objekte sie erzeugen.

## Struktur

Abb. 7.2-2 zeigt die Struktur des Fabrikmethode-Musters, dessen Klassen folgende Bedeutung haben:

**Product:** Die Klasse definiert die Schnittstelle der Objekte, die von der Fabrikmethode erzeugt werden.

**ConcreteProduct:** Diese Unterklasse implementiert die Schnittstelle des Produkts.

**Creator:** Die Klasse deklariert die abstrakte Fabrikmethode.

**ConcreteCreator:** Diese Unterklasse überschreibt die Fabrikmethode, so daß sie ein Objekt von ConcreteProduct zurückgibt.

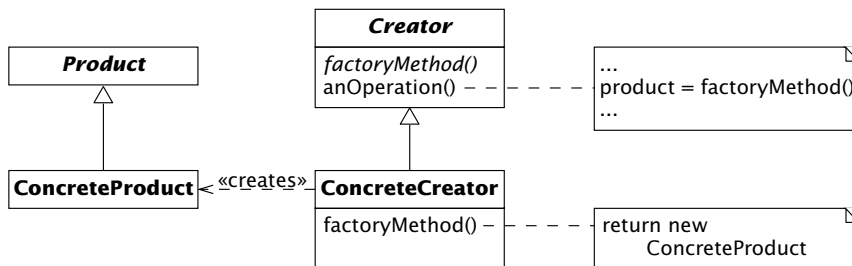


Abb. 7.2-2:  
Fabrikmethode-  
Muster

## Interaktionen

Der Creator verläßt sich darauf, daß Unterklassen die Fabrikmethode korrekt implementieren.

## Konsequenzen

Fabrikmethoden verhindern es, daß Sie anwendungsspezifische Klassen in den Code des *Framework* einbinden müssen.

## 7.3 Singleton-Muster

### Zweck

Das **Singleton-Muster** (*Singleton*) ist ein objektbasiertes Erzeugungsmuster. Es stellt sicher, daß eine Klasse genau ein Objekt besitzt und ermöglicht einen globalen Zugriff auf dieses Objekt.

### Motivation

Bei manchen Klassen ist es notwendig, daß es genau ein Objekt gibt. Auf dieses Objekt muß oft von mehreren anderen Objekten zu-

## LE 13 7 Entwurfsmuster

gegriffen werden. Daher muß der Zugriff einfach sein. Die *Singleton*-Klasse muß garantieren, daß nur ein Exemplar erzeugt werden kann und einen einfachen Zugriff auf dieses Exemplar ermöglichen.

### Anwendbarkeit

Verwenden Sie dieses Muster, wenn

- es genau ein Objekt einer Klasse geben und ein einfacher Zugriff darauf bestehen soll,
- das einzige Exemplar durch Spezialisierung mittels Unterklassen erweitert wird und Klienten das erweiterte Exemplar verwenden können, ohne ihren Code zu ändern.

### Struktur

Abb. 7.3-1 zeigt das *Singleton*-Muster. Die Klasse *Singleton* definiert die Klassenoperation *instance()*, die es dem Klienten ermöglicht, auf das einzige Exemplar zuzugreifen.

Die *Singleton*-Klasse wird folgendermaßen deklariert:

```
class Singleton
{
    public:
        static Singleton* instance();
    protected:
        Singleton();
    private:
        static Singleton* uniqueInstance;
};
```

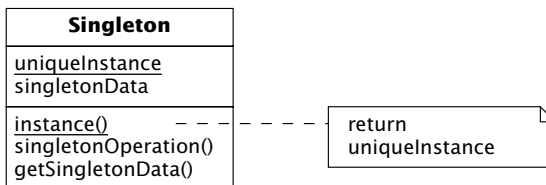
Das Klassenattribut wird definiert durch:

```
Singleton* Singleton : uniqueInstance = 0;
```

Die Implementierung der Operation *instance()* lautet:

```
Singleton* Singleton : instance()
{
    if (uniqueInstance == 0) //es existiert noch kein Exemplar
    {
        uniqueInstance = new Singleton;
    }
    return uniqueInstance;
}
```

Abb. 7.3-1:  
Singleton-Muster



### Interaktionen

Klienten holen sich ausschließlich über die Klassenoperation *instance()* eine Referenz auf das einzige Objekt.

### Konsequenzen

- Das *Singleton*-Muster ist eine Verbesserung gegenüber globalen Variablen.
- Die *Singleton*-Klasse kann durch Unterklassen spezialisiert werden.
- Werden später mehrere Exemplare benötigt, dann kann diese Änderung leicht durchgeführt werden.

## 7.4 Kompositum-Muster

### Zweck

Das **Kompositum-Muster** (*composite*) ist ein objektbasiertes Strukturmuster. Es setzt Objekte zu Baumstrukturen zusammen, um *whole-part*-Hierarchien darzustellen. Dieses Muster ermöglicht es, sowohl einzelne Objekte als auch einen Baum von Objekten einheitlich zu behandeln.

### Motivation

Grafische Editoren ermöglichen es, einzelne Grafikelemente zu komplexen Grafiken zusammenzusetzen, welche wiederum zu noch komplexeren Grafiken zusammengefügt werden können (Abb. 7.4-1).

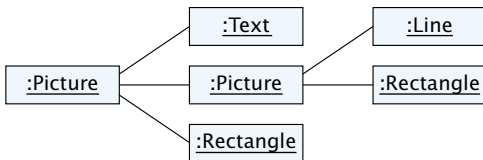
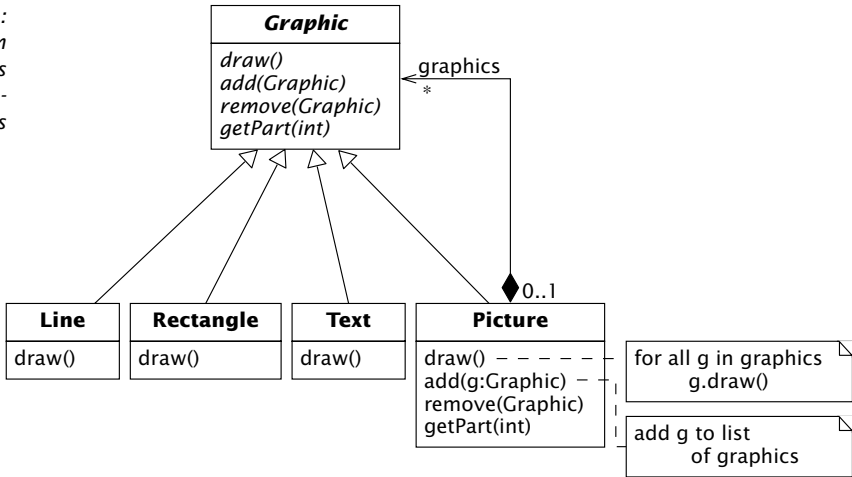


Abb. 7.4-1:  
Objektdiagramm  
für das Beispiel  
des Kompositum-  
Musters

Dabei müssen primitive und zusammengesetzte Objekte durch verschiedenen Programmcode manipuliert werden. Dieses Muster zeigt, wie eine reflexive Komposition benutzt wird, damit ein Klient diese Unterscheidung nicht treffen muß. Das Muster verwendet die abstrakte Klasse *Graphic*, die sowohl primitive als auch zusammengesetzte Objekte repräsentiert. Diese Klasse deklariert Operationen zur Manipulation elementarer Objekte (z.B. `draw()`) und zusammengesetzter graphischer Objekte (Zugriff auf Teilobjekte und Verwaltung von Teilobjekten). Die Klassen *Line*, *Rectangle* und *Text* implementieren elementare grafische Objekte. *Picture* ist eine Aggregatklasse, die das zusammengesetzte Objekt repräsentiert (Abb. 7.4-2).

LE 13 7 Entwurfsmuster

Abb. 7.4-2:  
Klassendiagramm  
für das Beispiel des  
Kompositum-  
Musters



**Anwendbarkeit**

Verwenden Sie dieses Muster, wenn

- Sie *whole-part*-Hierarchien von Objekten darstellen wollen,
- die Klienten keinen Unterschied zwischen elementaren und zusammengesetzten Objekten wahrnehmen und alle Objekte gleich behandeln sollen.

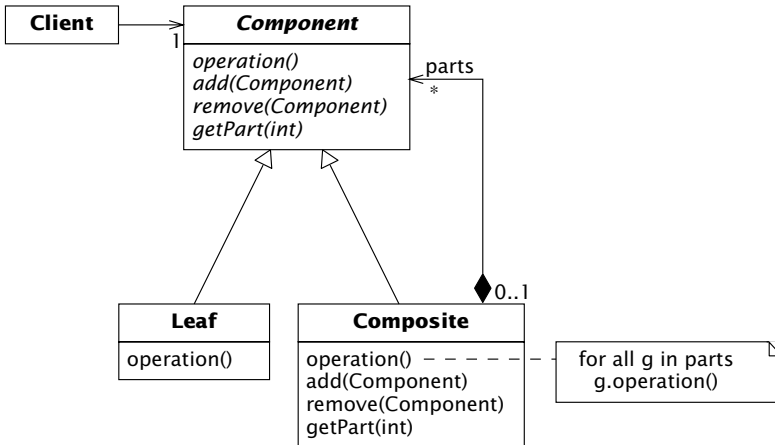
Dieses Muster läßt sich in vielen Klassenbibliotheken und speziell in *Frameworks* finden.

**Struktur**

Abb. 7.4-3 zeigt die Struktur des Kompositum-Musters, an dem folgende Klassen beteiligt sind:

**Component:** Diese Klasse deklariert die Schnittstelle für alle Objekte und implementiert eventuell ein *default*-Verhalten für die gemeinsame Schnittstelle. Sie deklariert eine Schnittstelle zum Zugriff und zum Verwalten von Teilobjekten in der Aggregatstruktur.

Abb. 7.4-3:  
Kompositum-Muster



Leaf: Diese Klasse repräsentiert elementare Objekte.

Composite: Diese Aggregatklasse definiert das Verhalten von zusammengesetzten Objekten, speichert Teilobjekte, und implementiert Operationen, die sich auf Teilobjekte beziehen.

Client: Diese Klasse repräsentiert die Klienten.

### Interaktionen

Alle Klienten verwenden nur die Schnittstelle von Component. Ist der Empfänger ein elementares Objekt, dann wird die Botschaft direkt bearbeitet. Ist der Empfänger ein zusammengesetztes Objekt, dann leitet es die Botschaft an seine Teilobjekte weiter.

### Konsequenzen

- Der Klient wird einfacher, da er zusammengesetzte und elementare Objekte gleich behandeln kann.
- Es ist einfach, neue Arten von Komponenten einzufügen.

## 7.5 Proxy-Muster

### Zweck

Das **Proxy-Muster** (*proxy*) ist ein objektbasiertes Strukturmuster. Es kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreter-Objekts. Dieses Muster wird auch als Surrogat (*surrogate*) bezeichnet.

### Motivation

Wir gehen von einem Textdokument aus, das grafische Objekte in den Text integrieren kann. Die Darstellung großer Bilder benötigt jedoch sehr viel Computerleistung. Wir sprechen daher von »teuren« Objekten. Wir wollen diese »teuren« Objekte nicht auf einmal, sondern erst »auf Verlangen« anzeigen. Dieses Problem lässt sich lösen, indem wir anstelle des Bilds einen Platzhalter – den Proxy – verwenden. Der Bild-Proxy erzeugt das tatsächliche Bild nur dann, wenn das Textdokument dessen Anzeige befiehlt. Abb. 7.5-1 zeigt das Objektdiagramm und Abb. 7.5-2 das Klassendiagramm für das beschriebene Beispiel.

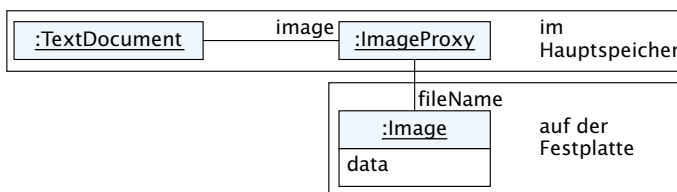


Abb. 7.5-1:  
Objektdiagramm  
für das Beispiel des  
Proxy-Musters

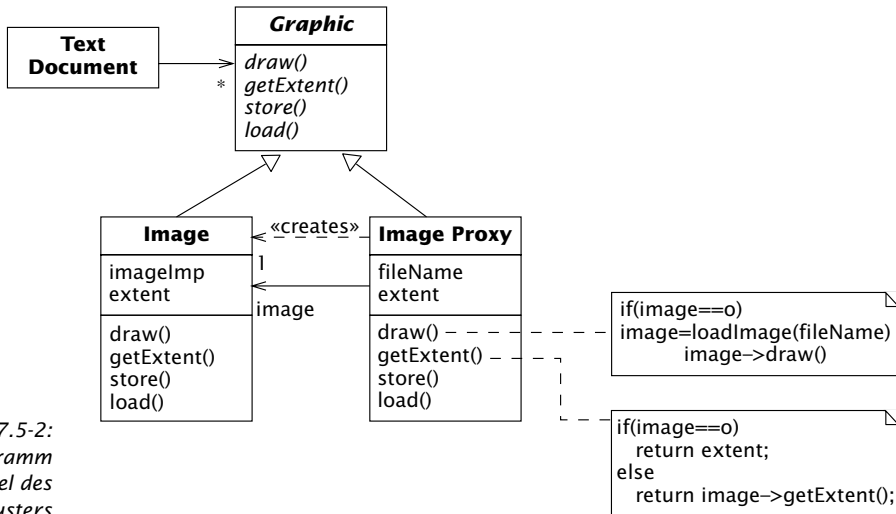


Abb. 7.5-2:  
Klassendiagramm  
für das Beispiel des  
Proxy-Musters

### Anwendbarkeit

Dieses weitverbreitete Muster ist in folgenden Situationen anwendbar:

- Ein *remote*-Proxy stellt einen lokalen Vertreter für ein Objekt auf einem anderen Computer dar.
- Ein virtuelles Proxy erzeugt »teure« Objekte auf Verlangen (siehe obiges Beispiel).
- Ein Schutz-Proxy kontrolliert den Zugriff auf das Originalobjekt.
- Ein *smart reference* ist ein Ersatz für einen einfachen Zeiger, der zusätzliche Aktionen ausführt. Dazu gehört das Zählen der Referenzen auf das eigentliche Objekt, so daß es automatisch freigegeben wird, wenn es keine Referenzen mehr besitzt (*smart pointer*). Weiterhin sorgt er dafür, daß ein persistentes Objekt beim erstmaligen Referenzieren in den Speicher geladen wird. Eine weitere Aufgabe besteht darin, daß getestet wird, ob das eigentliche Objekt gesperrt (*locked*) ist, bevor darauf zugegriffen wird.

### Struktur

Abb. 7.5-3 zeigt die allgemeine Struktur des Proxy-Musters, an dem die beteiligten Klassen folgende Aufgaben erfüllen:

**Proxy:** Diese Klasse kontrolliert den Zugriff auf das eigentliche Objekt und ist dafür zuständig, es zu erzeugen und zu löschen. Sie bietet eine Schnittstelle an, die mit der von Subject identisch ist, so daß ein Proxy für Subject eingesetzt werden kann. *Remote*-Proxies kodieren eine Botschaft und senden sie an das Real Subject in einem anderen Adreßraum. Virtuelle Proxies können zusätzliche Informationen über das Real Subject speichern, damit wegen dieser Infor-

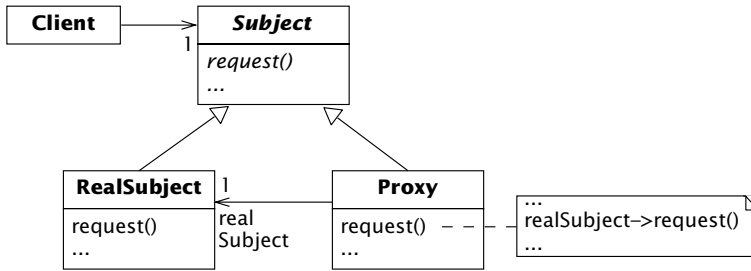


Abb. 7.5-3:  
Proxy-Muster

mationen nicht auf das echte Objekt zugegriffen werden muß. Schutz-Proxies prüfen, ob der Aufrufer die notwendigen Zugriffsrechte besitzt.

**Subject:** Diese Klasse definiert die gemeinsame Schnittstelle des echten Objekts und des Proxy-Objekts.

**Real Subject:** Die Klasse definiert das echte Objekt.

### Interaktionen

Der Proxy leitet Befehle an das echte Objekt weiter.

### Konsequenzen

- Ein *remote*-Proxy verbirgt die Tatsache, daß sich ein Objekt in einem anderen Adreßraum befindet.
- Ein virtuelles Proxy dient der Optimierung.
- Schutz-Proxies und *smart references* ermöglichen die Durchführung zusätzlicher Verwaltungsaufgaben beim Zugriff auf das Objekt.

## 7.6 Fassaden-Muster

### Zweck

Das **Fassaden-Muster** (*facade*) ist ein objektbasiertes Strukturmuster. Es bietet eine einfache Schnittstelle zu einer Menge von Schnittstellen (Pakete) an. Die Fassadenklasse definiert eine abstrakte Schnittstelle, um die Benutzung des Pakets zu vereinfachen.

### Motivation

Ein wichtiges Entwurfsziel ist es, Pakete möglichst lose zu koppeln. Das kann beispielsweise durch die Einführung einer Fassadenklasse erreicht werden, die eine vereinfachte Schnittstelle für die – umfangreichere – Funktionalität des Pakets zur Verfügung stellt. Den meisten Klienten genügt diese vereinfachte Sicht. Klienten, denen diese Schnittstelle nicht reicht, müssen hinter die Fassade schauen.



**Anwendbarkeit**

Verwenden Sie dieses Muster, wenn

- Sie eine einfache Schnittstelle zu einem komplexen Paket anbieten wollen.
- es zahlreiche Abhängigkeiten zwischen Klienten und einem Paket gibt. Dann entkoppelt die Fassade beide Komponenten und fördert damit Unabhängigkeit und Portabilität des Pakets.
- Sie die Pakete in Schichten organisieren wollen. Dann definiert eine Fassade den Eintritt für jede Schicht. Die Fassade vereinfacht den Zugriff auf die Schichten.

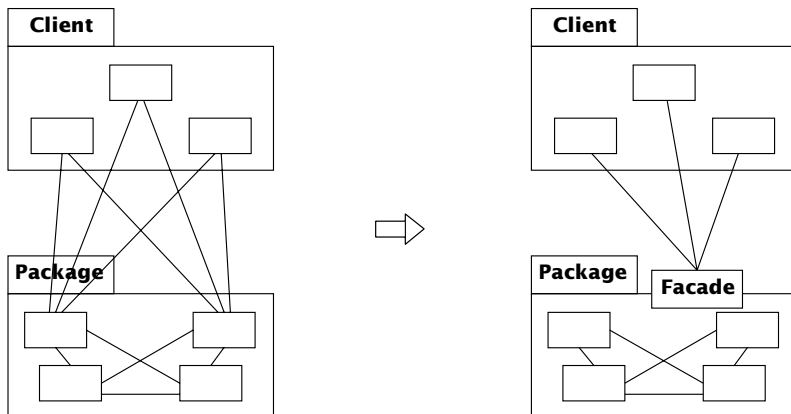
**Struktur**

Abb. 7.6-1 zeigt die allgemeine Struktur des Fassaden-Musters, an der folgende Klassen beteiligt sind:

*Facade*: Die Klasse weiß, welche Klassen des Pakets für die Bearbeitung einer Botschaft zuständig sind und delegiert Botschaften vom Klienten an die zuständige Klasse. Sie definiert keine neue Funktionalität. Oft wird nur ein Objekt der Fassadenklasse benötigt.

*Package*-Klassen: Sie führen die von der Fassade zugewiesenen Aufgaben durch und wissen nichts von der Fassade.

Abb. 7.6-1:  
Fassaden-Muster

**Interaktionen**

Die Klienten kommunizieren mit dem Paket, indem sie Botschaften an die Fassade schicken, welche diese dann an das zuständige Objekt innerhalb des Pakets weiterleitet.

**Konsequenzen**

- Das Fassaden-Muster reduziert die Anzahl der Klassen, welche den Klienten bekannt sein müssen und vereinfacht die Benutzung des Systems.
- Die lose Kopplung erleichtert es, Pakete auszutauschen und erleichtert deren unabhängige Implementierung.

- Bei Bedarf können Klienten die Fassade umgehen und direkt auf Klassen des Pakets zugreifen.

## 7.7 Beobachter-Muster

### Zweck

Das **Beobachter-Muster** (*observer*) ist ein objektbasiertes Verhaltensmuster. Es sorgt dafür, daß bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

### Motivation

Wir gehen von einem Objekt aus, das Anwendungsdaten enthält. Diese Daten sollen auf verschiedene Arten angezeigt werden, z.B. als Tabelle und als Kreisdiagramm (Abb. 7.7-1). Die beiden Anzeige-Objekte kennen einander nicht, was es erleichtert, nur eines der beiden wiederzuverwenden. Das Kreisdiagramm soll sich jedoch ändern, wenn die Daten in der Tabelle verändert werden und umgekehrt. Das Anwendungsobjekt (*subject*) kennt alle seine Anzeige-Objekte (Beobachter, *observer*) und informiert sie über alle Änderungen. Als Reaktion darauf synchronisiert sich jeder Beobachter mit dem Zustand des *subjects*.

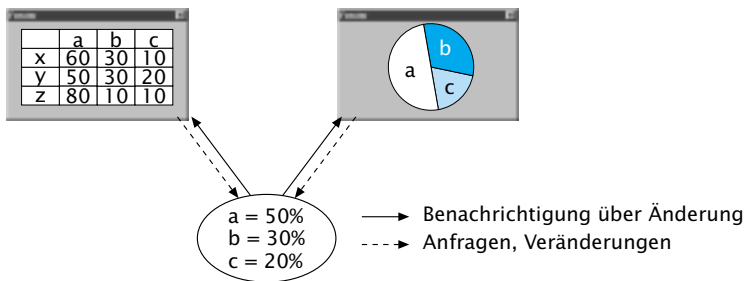


Abb. 7.7-1: Beispiel für das Beobachter-Muster

### Anwendbarkeit

Wenden Sie dieses Muster an, wenn gilt:

- Eine Abstraktion besitzt zwei Aspekte, die wechselseitig voneinander abhängen. Die Kapselung in zwei Objekte ermöglicht es, sie unabhängig voneinander wiederzuverwenden oder zu modifizieren.
- Die Änderung eines Objekts impliziert die Änderung anderer Objekte und es ist nicht bekannt, wie viele Objekte geändert werden müssen.
- Ein Objekt soll andere Objekte benachrichtigen und diese Objekte sind nur lose gekoppelt.

**Struktur**

Abb. 7.7-2 zeigt die Struktur des Beobachter-Musters, dessen Klassen für folgende Aufgaben verantwortlich sind:

**Subject:** Die Klasse kennt eine beliebige Anzahl von Beobachtern.

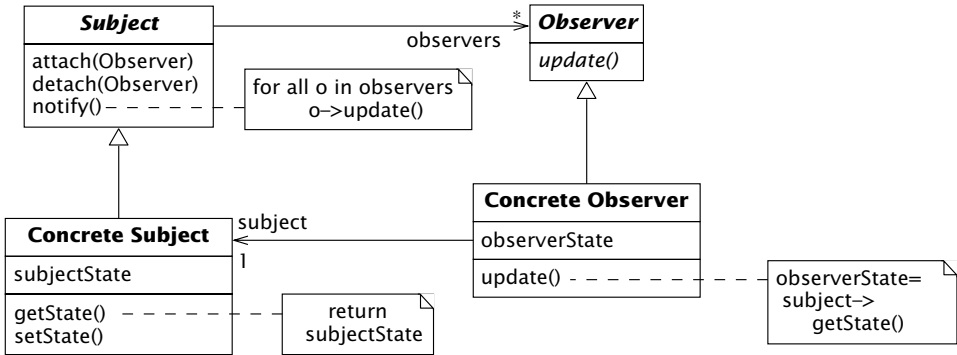
**Observer:** Diese Klasse definiert die Schnittstelle für alle konkreten *observer*, d.h. für alle Objekte, die über Änderungen eines *subjects* informiert werden müssen.

**ConcreteSubject:** Die Objekte dieser Klasse speichern die Daten, die für die konkreten Beobachter relevant sind.

**ConcreteObserver:** Die Objekte dieser Klasse kennen das konkrete Subjekt und merken sich den Zustand, der mit dem des Subjekts konsistent sein soll. Sie implementiert die Schnittstelle der

*Observer*-Klasse, um die Konsistenz zum Subjekt sicherzustellen.

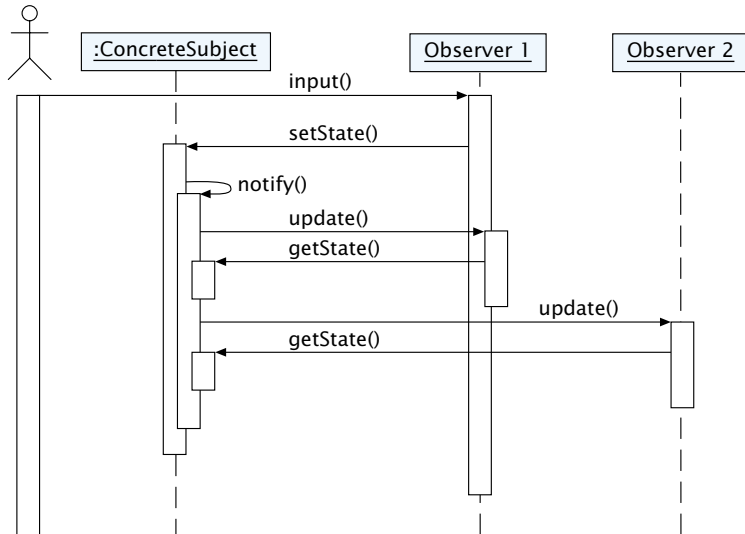
Abb. 7.7-2: Beobachter-Muster



**Interaktionen**

Abb. 7.7-3 beschreibt die Kommunikation der Objekte. Wird das Objekt der Klasse ConcreteSubject geändert, dann benachrichtigt es

Abb. 7.7-3: Interaktionen des Beobachter-Muster



alle seine Beobachter mittels `notify()`. Jedes benachrichtigte Objekt der Klasse `ConcreteObserver` bringt dann seinen Zustand mit dem des Subjekts in Einklang.

### Konsequenzen

- Das Beobachter-Muster ermöglicht es, Subjekte und Beobachter unabhängig voneinander zu modifizieren.
- Beobachter und Subjekte können einzeln wiederverwendet werden.
- Neue Beobachter können ohne Änderung des Subjekts hinzugefügt werden.

## 7.8 Schablonenmethode-Muster

### Zweck

Das **Schablonenmethode-Muster** (*template method*) ist ein objektbasiertes Verhaltensmuster. Es definiert den Rahmen eines Algorithmus in einer Operation und delegiert Teilschritte an Unterklassen.

### Motivation

Wir gehen von einem *Framework* aus, das die Klassen `Document` und `Application` bereitstellt (Abb. 7.8-1). Die Anwendung ist für das Öffnen der Dokumente zuständig und verwendet folgenden Algorithmus:

```
void Application::openDocument (const char* name)
{ if (!canOpenDocument (name) )
  return;
  Document* doc = doCreateDocument();
  if (doc)
  { docs->addDocument (doc);
    aboutToOpenDocument (doc);
    doc->open();
    doc->doRead();
  }
}
```

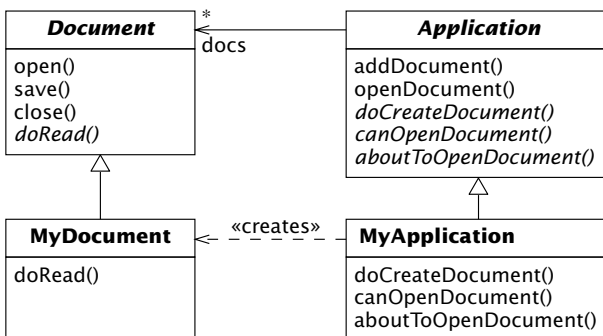


Abb. 7.8-1: Beispiel für Schablonenmethode-Muster

Bei den kursiven Namen handelt es sich um abstrakte Operationen, die von den Unterklassen überschrieben werden. Wir nennen `openDocument()` eine Schablonenmethode.

**Anwendbarkeit**

Verwenden Sie dieses Muster,

- um die invarianten Teile eines Algorithmus ein einziges mal festzulegen und die konkrete Ausführung der variierenden Teile den Unterklassen zu überlassen,
- wenn gemeinsames Verhalten von Unterklassen in einer Oberklasse realisiert werden soll, um die Duplikation von Code zu vermeiden.

Schablonenmethoden sind in vielen abstrakten Klassen zu finden.

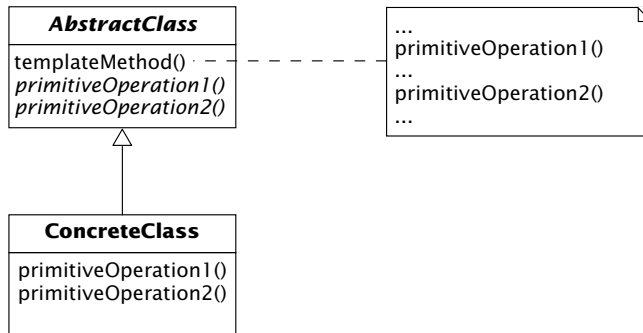
**Struktur**

Abb. 7.8-2 zeigt die allgemeine Struktur des Schablonenmethode-Musters, deren beteiligte Klassen für folgende Aufgaben verantwortlich sind:

*AbstractClass*: Diese Klasse definiert abstrakte primitive Operationen und implementiert die Schablonenmethode.

*ConcreteClass*: Sie implementiert die primitiven Operationen der abstrakten Oberklasse.

Abb. 7.8-2:  
Schablonenmethode-Muster



**Interaktionen**

Die *ConcreteClass* setzt voraus, daß die *AbstractClass* die invarianten Teile implementiert.

**Konsequenzen**

Schablonenmethoden bilden eine grundlegende Technik zur Wiederverwendung von Code. Insbesondere für Klassenbibliotheken bilden sie einen wichtigen Mechanismus, um das gemeinsame Verhalten in Bibliotheksklassen darzustellen. Sie realisieren das Hollywood-Prinzip »Don't call us, we'll call you«.



**Beobachter-Muster (*observer pattern*)** Das Beobachter-Muster ist ein objektbasiertes →Verhaltensmuster. Es sorgt dafür, daß bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

**Entwurfsmuster (*design pattern*)** Ein Entwurfsmuster gibt eine bewährte, generische Lösung für ein immer wiederkehrendes Entwurfsproblem an, das in bestimmten Situationen auftritt. Es lassen sich klassen- und objektbasierte Muster unterscheiden. Klassenbasierte Muster werden durch Vererbungen ausgedrückt. Objektbasierte Muster beschreiben in erster Linie Beziehungen zwischen Objekten.

**Erzeugungsmuster (*creational pattern*)** Erzeugungsmuster helfen dabei, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden.

**Fabrikmethode-Muster (*factory method*)** Das Fabrikmethode-Muster ist ein klassenbasiertes →Erzeugungsmuster. Es bietet eine Schnittstelle zum Erzeugen eines Objekts an, wobei die Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist.

**Fassaden-Muster (*facade pattern*)** Das Fassaden-Muster ist ein objektbasiertes →Strukturmuster. Es bietet eine einfache Schnittstelle zu einer Menge von Schnittstellen (Paket) an. Die Fassadenklasse definiert eine Schnittstelle, um die Benutzung des Pakets zu vereinfachen.

**Framework** Ein *Framework* besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und insbesondere aus abstrakten Klassen, die Schnittstellen definieren. Die abstrakten Klassen enthalten sowohl abstrakte als auch konkrete Operationen. Im allgemeinen wird vom Anwender (=Programmierer) des *Frameworks* erwartet, daß er Unterklassen definiert, um das

*Framework* zu verwenden und anzupassen.

**Klassenbibliothek** Eine Klassenbibliothek ist eine organisierte Sammlung von Klassen, aus denen der Entwickler nach Bedarf Einheiten verwendet, d.h. Objekte dieser Klassen definiert und Operationen darauf anwendet oder Unterklassen bildet. Klassenbibliotheken können unterschiedliche Topologien besitzen.

**Kompositum-Muster (*composite pattern*)** Das Kompositum-Muster ist ein objektbasiertes →Strukturmuster. Es setzt Objekte zu Baumstrukturen zusammen, um *whole-part*-Hierarchien darzustellen. Dieses Muster ermöglicht es, sowohl einzelne Objekte als auch einen Baum von Objekten einheitlich zu behandeln.

**Proxy-Muster (*proxy pattern*)** Das Proxy-Muster ist ein objektbasiertes →Strukturmuster. Es kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreter-Objekts.

**Schablonenmethode-Muster (*template method pattern*)** Das Schablonenmethode-Muster ist ein objektbasiertes →Verhaltensmuster. Es definiert den Rahmen eines Algorithmus in einer Operation und delegiert Teilschritte an Unterklassen.

**Singleton-Muster (*singleton pattern*)** Das *Singleton*-Muster ist ein objektbasiertes →Erzeugungsmuster. Es stellt sicher, daß eine Klasse genau ein Objekt besitzt und ermöglicht einen globalen Zugriff auf dieses Objekt.

**Strukturmuster (*structural pattern*)** Strukturmuster befassen sich damit, wie Klassen und Objekte zu größeren Strukturen zusammengesetzt werden.

**Verhaltensmuster (*behavioral pattern*)** Verhaltensmuster befassen sich mit der Interaktion zwischen Objekten und Klassen. Sie beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind. Sie lenken die Aufmerksamkeit weg vom Kontrollfluß hin zu der Art und Weise, wie die Objekte interagieren.

## LE 13 Zusammenhänge/Aufgaben

**Entwurfsmuster** beschreiben Lösungen für immer wiederkehrende Entwurfsprobleme. Aus dem Standardwerk von Gamma werden folgende Muster vorgestellt: **Fabrikmethode**, **Singleton**, **Kompositum**, **Proxy**, **Fassade**, **Beobachter** und **Schablonenmethode**. Während Muster nur abstrakte Lösungen bieten, stellen **Frameworks** Klassen bereit, die als Basisklassen für neu zu erstellende Anwendungen verwendet werden.



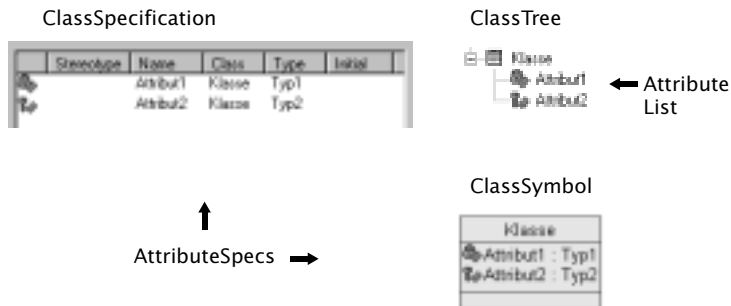
Aufgabe  
20 Minuten

### 1 Lernziele: Modellieren und Erkennen von Mustern.

Erstellen Sie für folgende Problemstellung ein OOD-Modell (Klassen- und Sequenzdiagramm). Prüfen Sie, welches der beschriebenen Muster vorliegt und wenden Sie es bei der Modellierung an. Abb. LE13-A1 zeigt Ausschnitte aus der Benutzungsoberfläche eines Werkzeugs für die objektorientierte Modellierung. Attribute werden in einem Spezifikationsfenster eingegeben, das durch die Klasse `ClassSpecification` realisiert wird. Die eingegebenen Daten sind ebenfalls im Klassensymbol (Klasse `ClassSymbol`) und in der Baumstruktur (Klasse `ClassTree`) sichtbar. Während im Spezifikationsfenster und im Klassensymbol die gleichen Informationen (*AttributeSpecs*) dargestellt werden, enthält der Baum nur die *AttributeList*.



Abb. LE13-A1:  
Benutzungsoberfläche eines Case-Werkzeugs



**2 Lernziele: Systematisches Identifizieren von Entwurfsmustern.**

Geben Sie an, ob und gegebenenfalls welche Muster in den Klassendiagrammen der Abb. LE13-A2a, LE13-A2b und LE13-A2c beschrieben sind.

Aufgabe  
10 Minuten

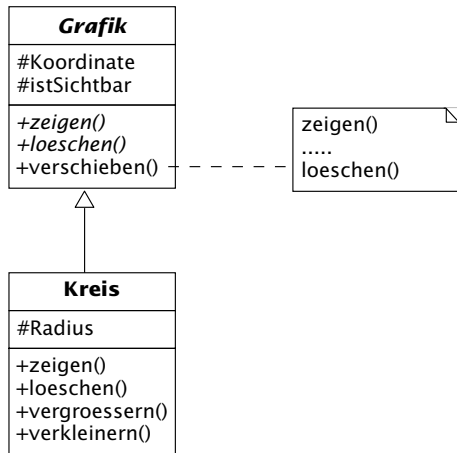


Abb. LE13-A2a:  
Klassendiagramm  
zum Identifizieren  
von Mustern  
(Teilaufgabe a)

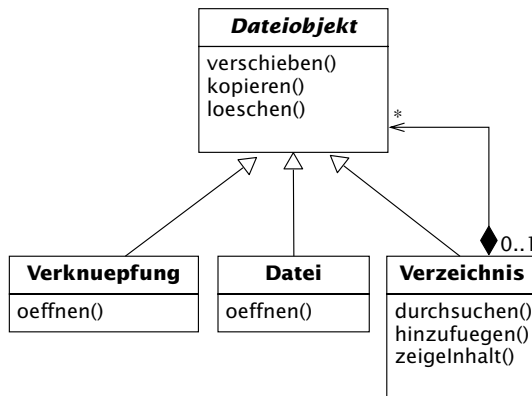


Abb. LE13-A2b:  
Klassendiagramm  
zum Identifizieren  
von Mustern  
(Teilaufgabe b)

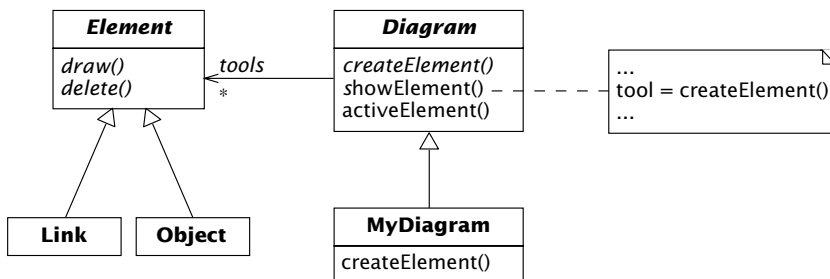


Abb. LE13-A2c:  
Klassendiagramm  
zum Identifizieren  
von Mustern  
(Teilaufgabe c)



### LE 13 Aufgaben

Aufgabe 3 *Lernziele: Gemeinsamkeiten und Unterschiede zwischen Muster, Framework und Klassenbibliothek kennen.*  
5–10 Minuten

Erläutern Sie

**a** was ein Muster, ein *Framework* und eine Klassenbibliothek gemeinsam haben und

**b** worin sich ein Muster, ein *Framework* und eine Klassenbibliothek unterscheiden.

# 8 Datenbanken

## Relationale Datenbanken und objekt-relationale Abbildung



- Wissen, was ein Datenbanksystem ist.
- Unterschiede des relationalen und objektorientierten Modells kennen.
- Erklären können, was ein relationales Datenbanksystem ist.
- DDL und DML anwenden können.
- Ein objektorientiertes Klassendiagramm systematisch auf Tabellen einer relationalen Datenbank abbilden können.

wissen

verstehen  
anwenden

Die objektorientierten Konzepte und die UML-Notation, wie sie in Kapitel 2 beschrieben sind, müssen bekannt sein.



- 8.1 Was ist ein Datenbanksystem? 304
- 8.2 Relationale Datenbanksysteme 306
- 8.3 Abbildung des objektorientierten Modells auf Tabellen 314

## 8 Datenbanken

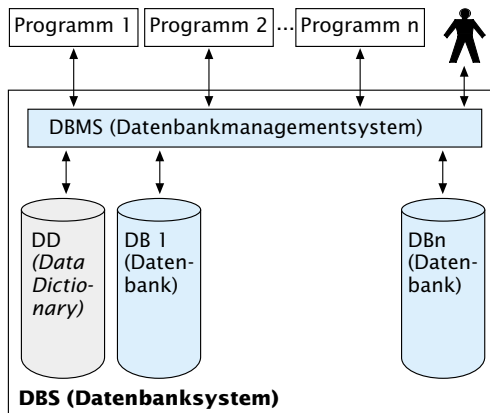
### 8.1 Was ist ein Datenbanksystem?

**Motivation** Oft benötigen die verschiedenen Programme in einer Organisation (Unternehmen, Behörden) gemeinsame Daten. Wenn jedes Programm seine eigene Datenhaltung besitzt, dann führt dies zu Mehrfacherfassungen, redundanter Speicherung und mit hoher Wahrscheinlichkeit zu inkonsistenten Datenbeständen. Datenbanksysteme ermöglichen die integrierte Verwaltung aller Daten.

**Definition** Ein **Datenbanksystem** (DBS) besteht aus einer oder mehreren Datenbanken, einem *Data Dictionary* und einem Datenbankmanagementsystem.

In der Datenbank (DB) sind alle Daten gespeichert. Das *Data Dictionary* (DD) enthält das Datenbankschema, das den Aufbau der Daten der Datenbank(en) beschreibt. Die Verwaltung und zentrale Kontrolle der Daten ist Aufgabe des Datenbankmanagementsystems (DBMS). Abb. 8.1-1 zeigt das Zusammenwirken dieser Komponenten.

Abb. 8.1-1:  
Architektur eines  
Datenbanksystems



**Eigenschaften** Ein Datenbanksystem muß eine Reihe von Eigenschaften besitzen (aus /Balzert 96/, in Anlehnung an /Dittrich, Geppert 95/):

- **Persistente Speicherung** der Daten, d.h. die Daten gehen nicht bei Programmende verloren, sondern sie stehen solange zur Verfügung, bis sie explizit gelöscht oder überschrieben werden.
- **Zuverlässige Verwaltung** der Daten, d.h. das Datenbanksystem stellt die Konsistenz, Integrität und Unversehrtheit der Daten sicher. Im Falle eines Hardware- oder Softwareausfalls ermöglicht das Datenbanksystem einen Wiederanlauf (*recovery*).
- **Unabhängige Verwaltung** der Daten, d.h. die in der Datenbank gespeicherten Daten werden einheitlich beschrieben. Dadurch werden die Anwendungsprogramme, die ein Datenbanksystem benutzen und das Datenbanksystem selbst weitgehend unabhängig voneinander.

- **Komfortable Verwendung** der Daten, d.h. der Benutzer muß sich nicht um Details – z.B. Speicherung der Daten – kümmern, sondern kommuniziert über eine höhere, abstrakte Schnittstelle mit der Datenbank.
- **Flexibler Zugang** zu den Daten, d.h. mit Hilfe geeigneter Anfragesprachen und anderer Hilfsmittel kann der Benutzer ohne prozedurale Programmierung *ad hoc* auf die Daten zugreifen.
- **Datenschutz**, d.h. Daten können vor unberechtigtem Zugriff geschützt werden.
- Verwaltung **großer Datenbestände**, d.h. die Datenbank kann nicht vollständig im Arbeitsspeicher gehalten werden.
- **Integrierte Datenbank**, d.h. alle Daten werden redundanzarm gespeichert, selbst wenn sie von verschiedenen Anwendungen stammen bzw. von verschiedenen Anwendungen verwendet werden. Das hat zur Folge, daß nicht jedes Anwendungsprogramm alle Daten benötigt, sondern nur bestimmte Ausschnitte.
- **Mehrfachbenutzung der Datenbank**, d.h. auf die Daten können mehrere Anwendungen – u.U. auch gleichzeitig – zugreifen. Der parallele Zugriff mehrerer Anwendungsprogramme oder Benutzer muß koordiniert werden.

Jedem Datenbanksystem liegt ein **Datenmodell** zugrunde, in dem festgelegt wird, Datenmodell

- welche Eigenschaften die Datenelemente besitzen,
- welche Struktur die Datenelemente besitzen dürfen,
- welche Konsistenzbedingungen einzuhalten sind und
- welche Operationen zum Speichern, Suchen, Ändern und Löschen von Datenelementen existieren.

Ein relationales Datenbanksystem (RDBS) liegt vor, wenn dem Datenbanksystem ein relationales Datenmodell zugrunde liegt. Es ist heute das am meisten verwendete Datenmodell. Analog liegt ein objektorientiertes Datenbanksystem (ODBS) vor, wenn ein objektorientiertes Datenmodell zugrunde liegt. Objektorientierte Datenbanksysteme ermöglichen eine homogene objektorientierte Entwicklung. Alle Objekte der Anwendung lassen sich direkt in der Datenbank speichern (Abb. 8.1-2). Bei Verwendung einer relationalen relational vs. objektorientiert

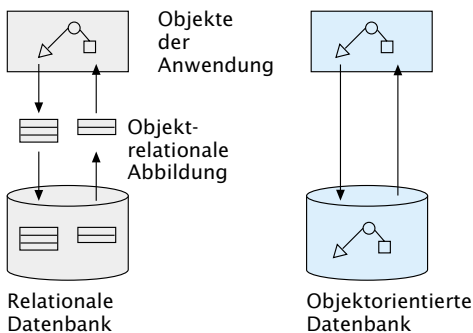


Abb. 8.1-2: Vergleich von relationalen und objektorientierten Datenbanksystemen

Datenbank in objekt-orientierten Anwendungen ist dagegen eine **objekt-relationale Abbildung** (*object relational mapping*) durchzuführen. Wegen der großen Verbreitung der relationalen Datenbanksysteme ist dieser Prozeß von besonderer Bedeutung.

## 8.2 Relationale Datenbanksysteme

Für eine »gute« objekt-relationale Abbildung ist nicht nur ein Verständnis des Objektmodells, sondern auch des relationalen Modells notwendig. Das Ziel dieses Kapitel ist es, Ihnen die grundlegenden Kenntnisse über relationale Datenbanksysteme zu vermitteln, damit Sie eine objektorientierte Anwendung mit einer relationalen Datenbank verbinden können. Viele wichtige Aspekte von **relationalen Datenbanksystemen** werden nur am Rande gestreift oder entfallen ganz, weil es sich um Aufgaben des Datenbank-Entwurfs handelt. Um den Praxisbezug zu betonen, wurde auf die zugrundeliegende Theorie nahezu völlig verzichtet und zur Formulierung aller Beispiele Oracle/SQL verwendet. Ausführlichere Informationen zu relationalen Datenbanken finden Sie in /Balzert 96/ und /Vossen 94/.

Relationale Datenbanken speichern Daten in Form von Relationen. Eine **Relation** kann anschaulich als **Tabelle** verstanden werden. Im einfachsten Fall lassen sich die Attribute einer Klasse auf eine Tabelle abbilden. Wir betrachten als Beispiel die Klasse **Artikel** (Abb. 8.2-1). Jede Zeile der Tabelle repräsentiert ein Objekt der Klasse. Bei relationalen Datenbanken spricht man von einem **Tupel**. Jedes Attribut der Klasse wird auf ein Attribut der Tabelle abgebildet. Das ist in diesem Fall möglich, weil alle Attribute von einem Typ sind, der in relationalen Datenbanken realisiert werden kann. Alle Tupel einer Tabelle müssen gleich lang sein. Die Reihenfolge der Attribute spielt keine Rolle.

Abb. 8.2-1:  
Abbildung einer Klasse auf eine Tabelle

	Schlüssel	Attribut	
<b>Artikel</b>	<u>Nummer</u>	Bezeichnung	Preis
<b>Tupel</b>			

**Schlüsselattribut** In einer relationalen Datenbank muß jedes Tupel durch einen eindeutigen Schlüssel identifizierbar sein. Der Schlüssel (auch als Primärschlüssel bezeichnet) kann aus einem oder mehreren Attributen bestehen. Wenn kein fachliches Attribut als Schlüsselattribut verwendet werden kann, dann muß ein künstliches Schlüssel-

attribut (z.B. eine Nummer) hinzugefügt werden. In der grafischen Darstellung einer Tabelle wird der Schlüssel unterstrichen.

In Abb. 8.2-2 wird bei der Abbildung der Klasse Lieferant auf die gleichnamige Tabelle das Schlüsselattribut Nummer hinzugefügt, da davon ausgegangen wird, daß die Klasse kein Schlüsselattribut (key) enthält. Beispiel

Nachteilig an diesem Konzept ist, daß sich Schlüsselattribute in der Tabelle äußerlich nicht von anderen Attributen unterscheiden. Schlüsselattribute müssen explizit verwaltet werden. Im objektorientierten Modell besitzt dagegen jedes Objekt implizit eine Objektidentität (*object identifier, OID*). Schlüssel vs. Objektidentität

Die Assoziationen bzw. die Objektverbindungen (*links*) im objektorientierten Modell werden bei relationalen Datenbanken durch Schlüssel-Fremdschlüssel-Beziehungen realisiert. Fremdschlüssel

Wie die Abb. 8.2-2 zeigt, gehört jeder Artikel zu genau einem Lieferanten, der mehrere Artikel liefern kann. Die Assoziation zwischen den Tabellen Lieferant und Artikel wird mittels Fremdschlüssel realisiert. Dadurch kann zu jedem Artikel der jeweilige Lieferant ermittelt werden. Wegen der konstanten Satzlänge ist es nicht möglich, daß zu einem Lieferanten die Nummern aller von ihm gelieferten Artikel eingetragen werden. Im Gegensatz zum objektorientierten Modell wird die Verbindung zwischen den Artikel- und Lieferanten-Objekten durch den Fremdschlüssel nur in einer Richtung realisiert. Beispiel

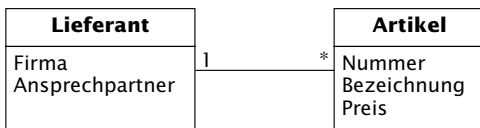


Abb. 8.2-2: Abbildung einer Assoziation auf Tabellen

**Schlüssel**

Lieferant	<u>Nummer</u>	Firma	Ansprechpartner

**Fremdschlüssel**

Artikel	<u>Nummer</u>	Bezeichnung	Preis	L-Nummer

Beim Relationenmodell sind zwei fundamentale Integritätsregeln einzuhalten. **Entitäts-Integrität** bedeutet, daß Schlüsselattribute immer einen Wert besitzen müssen. **Referentielle Integrität** bedeutet: Korrespondiert ein Fremdschlüssel FS einer Tabelle T<sub>1</sub> mit Integritätsregeln

dem Primärschlüssel PS einer Tabelle  $T_2$ , dann muß jeder Wert von FS als Wert von PS vorkommen oder FS besitzt einen Nullwert. Einfach ausgedrückt bedeutet die referentielle Integrität: Wenn in einer Tabelle ein Fremdschlüssel vorhanden ist, dann muß der Fremdschlüsselwert auch als Primärschlüsselwert in der korrespondierenden Tabelle vorkommen.

### Relationales Datenmodell

formale Definition

Das relationale Datenmodell wurde 1970 von E. F. Codd veröffentlicht. Das relationale Modell basiert auf dem (mathematischen) Konzept der Relation und ist daher exakt formulierbar.

Sind  $W(A_1), W(A_2), \dots, W(A_n)$  endliche Mengen, so heißt die Menge aller Kombinationen ihrer Elemente (Vektoren) ihr kartesisches Produkt  $[W(A_1) \times W(A_2) \times \dots \times W(A_n)]$ . Die Elemente von kartesischen Produkten heißen Tupel. Jede Teilmenge  $R$  eines kartesischen Produkts  $W(A_1) \times W(A_2) \times \dots \times W(A_n)$  heißt eine ( $n$ -stellige) Relation über  $W(A_1) \times W(A_2) \times \dots \times W(A_n)$ .

Jede Relation  $R \subseteq W(A_1) \times W(A_2) \times \dots \times W(A_n)$  kann als Tabelle mit dem Namen  $R$  dargestellt werden. Die Spalten tragen die Namen der Attribute. In den Zeilen sind die Elemente von  $R$  – die Tupel – aufgeführt. Da Relationen Mengen sind, ist das mehrfache Vorkommen eines Tupels ausgeschlossen.

Beispiel

Es seien  $W(\text{Nummer}) = \{1; 2; 3\}$ ,  $W(\text{Anrede}) = \{\text{Herr; Frau}\}$  und  $W(\text{Titel}) = \{\text{Dr; Prof}\}$ .

Dann gilt:

$$W(\text{Nummer}) \times W(\text{Anrede}) \times W(\text{Titel}) =$$

$$\{ (1, \text{Herr}, \text{Dr}), (1, \text{Herr}, \text{Prof}), (1, \text{Frau}, \text{Dr}), (1, \text{Frau}, \text{Prof}),$$

$$(2, \text{Herr}, \text{Dr}), (2, \text{Herr}, \text{Prof}), (2, \text{Frau}, \text{Dr}), (2, \text{Frau}, \text{Prof}),$$

$$(3, \text{Herr}, \text{Dr}), (3, \text{Herr}, \text{Prof}), (3, \text{Frau}, \text{Dr}), (3, \text{Frau}, \text{Prof}) \}$$

↑Tupel

Normalformen

Im Zusammenhang mit relationalen Datenbanken wird oft von Normalisierung bzw. von **Normalformen** gesprochen. Es gibt fünf Normalformen, wobei in der Praxis die ersten drei Normalformen von Bedeutung sind. Damit Daten in einer relationalen Datenbank gespeichert werden können, müssen sie sich mindestens in der ersten Normalform befinden, da Grundlage für den Aufbau der Tabellen eine konstante Länge der Einträge ist. Die erste Normalform fordert, daß alle Attribute einer Tabelle keine Wiederholung von Werten (*array*) und keine internen Datenstrukturen (*struct*) enthalten dürfen. Mit anderen Worten: Eine Tabelle befindet sich in der ersten Normalform, wenn alle Attribute von einem elementaren Typ sind. Normalerweise sind die Daten, die in einer relationalen Datenbank gespeichert sind, in der ersten, zweiten oder dritten Normalform. Während die erste Normalform Datenredundanz aufweist, besitzt die dritte Normalform einen hohen Grad an Schlüsselredundanz. In der Praxis ist daher sorgfältig abzuwägen »wieviel Normalisierung« sinnvoll ist. Da die Wahl der richtigen Normalform eine Aufgabe des

Datenbank-Entwurfs darstellt, gehe ich hier nicht näher darauf ein. Bei einem objektorientierten Modell spielt die Normalisierung keine Rolle. Die Bildung von Attributtypen erfolgt in der objektorientierten Analyse ausschließlich unter problemadäquaten Gesichtspunkten.

Die Menge aller Tabellen bildet die relationale Datenbank. Sie werden als **logisches Schema** bezeichnet, das im *Data Dictionary* eingetragen wird. Alle persistenten Daten werden in der Datenbank abgelegt und die Funktionalität wird durch die Anwendungsprogramme realisiert. Daten und Funktionen sind im relationalen Modell nicht gekapselt. Alle Attribute des logischen Schemas sind für den Benutzer und die Anwendungsprogramme sichtbar. Das Geheimnisprinzip wird von den relationalen Datenbanksystemen ebenfalls nicht realisiert.

logisches Schema

Die formale Definition des logischen Schemas erfolgt durch die **Datendefinitionssprache** bzw. die **DDL** (*Data Definition Language*), die das Datenbanksystem zur Verfügung stellt. Als Standard hat sich die Sprache **SQL** (*Structured Query Language*) etabliert. In SQL wird eine Tabelle durch den *create table*-Befehl erzeugt. Für jedes Attribut sind dessen Name und der Typ anzugeben. Attributnamen müssen innerhalb einer Tabelle eindeutig sein. Bei einer tabellenübergreifenden Betrachtung wird das Attribut durch Tabelle.Attribut eindeutig bezeichnet. Schlüssel- und Fremdschlüsselattribute werden in SQL nicht speziell gekennzeichnet, sondern sehen wie »normale« Attribute aus. Ein Attribut, das mit *not null* gekennzeichnet ist, muß bereits beim Erzeugen des Tupels einen Wert besitzen. Es handelt sich um ein Mußattribut. Schlüsselattribute und Fremdschlüssel einer Muß-Assoziation sind immer mit *not null* einzutragen.

DDL

Die Menge aller *create table*-Befehle richtet die leere relationale Datenbank ein. Jede definierte Tabelle wird in das *Data Dictionary* eingetragen. Mit dem *drop table*-Befehl wird eine Tabelle wieder gelöscht.

Die in Abb. 8.2-2 angegebenen Tabellen werden durch folgende SQL-Befehle erzeugt: Beispiel

```
create table Lieferant
  (Nummer      number(5)      not null ,
   Firma       char(30)       not null ,
   Ansprechpartner char(30));
create table Artikel
  (Nummer      number(5)      not null ,
   Bezeichnung char(30)       not null ,
   Preis       number(8,2),
   L_Nummer    number(5)      not null );
```



**Zur Historie von SQL**

**SQL** (*Structured Query Language*) ist eine Sprache der 4. Generation. Sie ist eine deklarative Programmiersprache, d.h. sie besitzt im Unterschied zu den klassischen Programmiersprachen keine Schleifen, keine Prozeduren, keine Rekursion und keine ausreichenden mathematischen Operationen.

SQL wurde in den 70ern in den Forschungslaboratorien von IBM im Zusammenhang mit der Prototypentwicklung des relationalen Datenbanksystems SYSTEM R (R wie relational) entworfen. Die Grundlagen bildeten die Arbeiten von E. F. Codd. 1979 brachte die Firma Oracle die erste SQL-Datenbank auf den Markt. Viele Datenbankhersteller zogen mit ihren SQL-Entwicklungen nach. 1983 wurde von ANSI und ISO ein SQL-Standard definiert. Heute gilt SQL als die Standard-Abfragesprache für relationale Datenbanksysteme. Weiterentwicklungen führten zum derzeitigen Standard SQL2, der 1992 veröffentlicht wurde, und zu SQL3 (noch nicht verabschiedet).

Typen Eine wesentliche Eigenschaft von relationalen Datenbanksystemen ist, daß nur elementare Typen realisiert werden können. Der ANSI-SQL-Standard /ANSI 92/ definiert für relationale Datenbanken folgende Typen, die jedoch nicht von allen Datenbanksystemen unterstützt werden:

- |                     |             |
|---------------------|-------------|
| ■ character         | ■ date      |
| ■ character varying | ■ interval  |
| ■ bit               | ■ time      |
| ■ bit varying       | ■ timestamp |
| ■ numeric           | ■ float     |
| ■ decimal           | ■ real      |
| ■ integer           | ■ double    |
| ■ smallint          | ■ precision |

Strukturen oder Listen können in einer Tabelle nicht direkt dargestellt werden. Einige Datenbanksysteme erlauben zusätzlich sogenannte BLOBs (*binary large objects*). Dieser Typ ermöglicht das Speichern von Bildern, Texten, Sprache usw. Die interne Struktur dieser Daten geht jedoch verloren und muß im Anwendungsprogramm dargestellt werden.

Wenn die leere Datenbank eingerichtet ist, kann sie von den Benutzern und Anwendungsprogrammen gefüllt und verändert werden. Für diese Aufgaben stellt das DBMS die **Datenmanipulations-sprache** bzw. die **DML** (*Data Manipulation Language*) zur Verfügung. Auch hier gilt SQL als Standard. Weil die DML weder Kontrollstrukturen noch Prozedurkonzepte enthält, ist die Erstellung umfangreicher Programme problematisch. Daher ist es sinnvoll, die DML mit einer klassischen Programmiersprache (z.B. C++) zu kombinieren.

Das Relationenmodell stellt einen Satz von generischen Operationen zur Verfügung, d.h. die Semantik dieser Operationen ist nicht anwendungsspezifisch, sondern gehört zur Konzeption des Relationenmodells. Wir betrachten im folgenden generische Operationen zum Verändern der Datenbank und zur Formulierung von Anfragen.

generische  
Operationen

Mit dem *insert*-Befehl werden neue Tupel in eine Tabelle eingetragen, mit dem *update*-Befehl vorhandene Tupel verändert und mit *delete* gelöscht.

*insert, update, delete*

```
insert into Lieferant values
(0815, 'SchreibmühtMüller', null, 'Hans Müller');
insert into Artikel values
(4711, 'Notizblock', null);
update Artikel
set Preis = 4.95
where Nummer = 4711;
update Artikel
set Preis = Preis + 1.5
where Nummer = 4711;
delete from Artikel
where Nummer = 4711;
```

Beispiel

SQL enthält den *select*-Befehl, mit dem Anfragen (*queries*) flexibel und komfortabel realisiert werden können. Das Ergebnis eines *select*-Befehls besteht aus einem oder mehreren Tupeln. Wir betrachten im folgenden einige einfache *select*-Operationen.

*select*

Die Selektion wählt Tupel, d.h. Zeilen, aus einer Tabelle aus. Das Ergebnis einer Selektion können alle Zeilen der Tabelle sein. Im allgemeinen werden die Tupel entsprechend einer Bedingung gefiltert. In dieser Bedingung können Werte abgefragt oder das Vorhandensein von Werten geprüft werden.

Selektion

Die folgende Selektion gibt die vollständige Tabelle aus. Die Angabe »\*« bedeutet, daß alle Attribute der Tabelle Artikel ausgegeben werden.

Beispiele

```
select * from Artikel;
```

Das Ergebnis der folgenden Selektion ist eine Teilmenge aller Tupel von Artikel, d.h. alle Artikel, die mindestens 100 DM kosten.

```
select *
from Artikel
where Preis >= 100;
```

Die folgende Selektion ermittelt alle Artikel, für die noch kein Preis eingetragen wurde.

```
select *
from Artikel
where Preis is null;
```

Eine Projektion wählt bestimmte Spalten einer Tabelle aus, während alle Tupel der Tabelle angezeigt werden. Die Angabe von *distinct* sorgt dafür, daß keine Duplikate von Datensätzen erzeugt werden. Projektion und Selektion werden häufig kombiniert.

Projektion

## LE 14 8 Datenbanken

Beispiele Nachfolgende Projektion gibt für jeden Artikel der Tabelle die Bezeichnung und den Preis aus. Wenn mehrere Artikel die gleiche Bezeichnung und den gleichen Preis besitzen, dann werden sie wegen der Angabe von *distinct* auf ein einziges Tupel abgebildet.

```
select distinct Bezeichnung, Preis  
from Artikel;
```

Die folgende *select*-Anweisung ermittelt nur diejenigen Artikel, für die ein Preis eingegeben wurde. Für alle selektierten Tupel werden Bezeichnung und Preis ausgegeben (Kombination von Selektion und Projektion).

```
select distinct Bezeichnung, Preis  
from Artikel  
where Preis is not null;
```

natürlicher Verbund Der natürliche Verbund (*natural join*) verknüpft zwei oder mehrere Tabellen über gemeinsame Attribute, wobei diese Attribute in der Ergebnistabelle nur einmal aufgeführt werden. Mehrere Bedingungen in der *where*-Klausel werden mittels *and* verknüpft.

Beispiele Die folgende *select*-Anweisung ermittelt für alle Lieferanten deren Nummer und Firma sowie die Bezeichnungen und Preise ihrer gelieferten Artikel. Sind zu einem Lieferanten mehrere Artikel eingetragen, dann ergeben sich mehrere Tupel.

```
select Lieferant.Nummer, Firma, Bezeichnung, Preis  
from Lieferant, Artikel  
where Artikel.L_Nummer = Lieferant.Nummer;
```

Die folgende *select*-Anweisung ermittelt alle Lieferanten, die einen Billigartikel unter 100 DM liefern.

```
select Lieferant.Nummer, Firma, Bezeichnung, Preis  
from Lieferant, Artikel  
where Artikel.L_Nummer = Lieferant.Nummer  
and Artikel.Preis < 100;
```

externe Schemata Bestimmte Benutzergruppen oder Anwendungsprogramme sollen oft nur einen definierten Ausschnitt des logischen Schemas sehen. Daher werden aus dem logischen Schema die **externen Schemata** bzw. **Sichten** (*views*) abgeleitet. Externe Schemata werden ebenfalls im *Data Dictionary* abgelegt. Einem externen Schema können *keine* Daten zugeordnet werden. Die Definition einer Sicht bedeutet daher nicht, daß die Daten mehrfach in der Datenbank abgelegt werden, sondern sie werden stets bei einer Abfrage neu aufgebaut. Dadurch ist sichergestellt, daß sich jede Sicht (*view*) stets auf die aktuellen Daten bezieht. Eine Sicht wird in SQL durch den *create view*-Befehl erzeugt. Er kann aus einer oder mehreren Basistabellen – und auch aus vorhandenen Sichten – abgeleitet werden. Sichten können analog zu den Tabellen mit dem *drop view*-Befehl wieder gelöscht werden. Dieser Befehl besitzt keinen Einfluß auf die Originaltabelle und die darin enthaltenen Daten.

Die Sicht `Billigartikel` enthält alle Artikel, die weniger als 100 DM kosten. Die Sicht `ArtikelOhnePreis` erstellt eine Liste aller Artikel, jedoch ohne deren Preis (unabhängig davon, ob in der Originaltabelle ein Preis eingetragen ist oder nicht). Beispiele

```
create view Billigartikel
as select *
from Artikel
where Preis < 100;
create view ArtikelOhnePreis
as select Nummer, Bezeichnung
from Artikel;
```

Indizes können für zwei Aufgaben eingesetzt werden: Zum einen steigern sie die *Performance* und zum anderen stellen sie die Eindeutigkeit von Schlüsselattributen sicher. Wenn für eine Tabelle ein Index existiert, dann benutzt das Datenbanksystem bei allen Anfragen diesen Index. Ist kein Index für eine Tabelle definiert, dann durchsucht das Datenbanksystem die Tabelle von Anfang bis Ende, um die gewünschten Tupel zu finden. Aus diesem Grund sollten Sie Indizes für alle Attribute anlegen, die häufig in *where*-Klauseln von *select*-Befehlen auftreten. Außerdem empfiehlt sich ein Index für alle Schlüsselattribute. Ein Index kann auch für mehrere Attribute einer Tabelle definiert werden (zusammengesetzter Index). Index

In der Tabelle `Artikel` stellt der Index `Artikelnummer` sicher, daß jede Nummer nur einmal vergeben wird. Der Index `Lieferantenfirmen` optimiert den Zugriff über das Attribut `Firma`. Beispiele

```
create unique index Artikelnummer on Artikel (Nummer);
create index Lieferantenfirmen on Lieferant (Firma);
```

Eine kompakte Modellierung der Tabellenstrukturen ermöglicht die in Abb. 8.2-3 dargestellte Notation. Die Primärschlüssel werden unterstrichen. Eine Schlüssel-Fremdschlüssel-Beziehung wird durch den Eintrag des Fremdschlüssel-Attributs und durch einen Pfeil zwischen den jeweiligen Tabellen dargestellt. Um Tabellen und Klassen leicht unterscheiden zu können, werden alle Tabellennamen blau eingetragten. grafische Notation

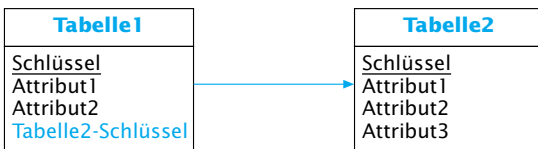


Abb. 8.2-3:  
Notation zur  
Darstellung der  
Tabellenstruktur

## 8.3 Abbildung des objektorientierten Modells auf Tabellen

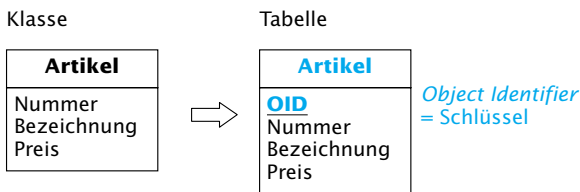
In Kapitel 8.2 haben wir den Fall betrachtet, daß eine Klasse auf eine Tabelle abgebildet wird. Das ist nicht immer so einfach. Prinzipiell gibt es folgende Abbildungsmöglichkeiten (siehe /Rumbaugh et al. 91/, /Brown, Whitenack 95/ und /Ambler 97/):

- 1 Klasse → 1 Tabelle,
- 1 Klasse → n Tabellen,
- n Klassen → 1 Tabelle.

### Abbildung einer Klasse

Eine Klasse, deren Attribute alle vom elementaren Typ sind, erfüllt die erste Normalform. Sie wird auf eine einzige Tabelle abgebildet (Abb. 8.3-1).

Abb. 8.3-1:  
Abbildung einer  
einfachen Klasse



OID-Attribut Jede Tabelle wird – unabhängig davon, ob ein fachliches Schlüsselattribut vorhanden ist – um ein **OID-Attribut** erweitert, das die Rolle des Schlüsselattributs spielt. Ein OID-Attribut darf *keinesfalls* eine semantische Bedeutung besitzen, denn erfahrungsgemäß ändert sich diese Semantik. Würde beispielsweise als OID-Attribut die Kundennummer gewählt und ist eine Erweiterung des Nummernkreises notwendig, dann müssen alle Tupel, in denen diese Kundennummer als Primär- und Fremdschlüssel vorkommt, aktualisiert werden. Gegebenenfalls muß auch der entsprechende Datentyp in den Tabellen geändert werden.

Beim OID-Attribut werden drei Stufen der Eindeutigkeit unterschieden:

- Eindeutigkeit innerhalb einer Klasse,
- Eindeutigkeit innerhalb einer Vererbungshierarchie und
- Eindeutigkeit innerhalb der Datenbank.

Die größte Flexibilität bietet die dritte Stufe. Sie bedeutet, daß jedes Tupel einer jeden Tabelle in der Datenbank einen eindeutigen OID-Wert besitzt.

OID-Attribut Realisierung Das OID-Attribut kann durch eine sehr große ganze Zahl realisiert werden. Einige Datenbanken generieren entsprechende OID-Attribute, während bei anderen die OID-Verwaltung vom Anwendungsprogramm realisiert werden muß. Im einfachsten Fall kann eine OID-Tabelle verwendet werden, welche die jeweils die zuletzt vergebene

OID jeder Klasse enthält. Diese Tabelle ist zu aktualisieren, wann immer eine neue OID vergeben wird. Um die *Performance* zu steigern, können die OIDs von der Anwendung blockweise angefordert werden /Brown, Whitenack 95/. Eine weitere Verbesserung der *Performance* ergibt sich durch den *high/low*-Ansatz von /Ambler 97/. Die OID besteht aus einem *high*-Wert, der beispielsweise aus einer OID-Tabelle entnommen wird und einem *low*-Wert, der von der Anwendung verwaltet wird. Der *low*-Wert wird mit Null initialisiert und jedesmal inkrementiert, wenn eine neue OID benötigt wird.

Im einfachsten Fall wird ein Attribut einer Klasse auf ein Attribut einer Tabelle abgebildet. Vor der Abbildung muß jedoch zunächst geprüft werden, welche Attribute einer Klasse überhaupt persistent sein sollen. Insbesondere abgeleitete Attribute werden meistens berechnet und sind daher nicht dauerhaft zu speichern.

persistente Attribute

Ist ein Attribut von einem **Struktur-Typ**, dann muß es in Komponenten zerlegt werden, die vom einfachen Typ sind (Abb. 8.3-2). Diese Attribute können entweder in die Tabelle der Klasse integriert (z.B. Name) oder in einer eigenen Tabelle dargestellt werden (z.B. Adresse). Eine separate Tabelle besitzt die Vorteile, daß der Zusammenhang der Komponenten zu einem Ganzen erhalten bleibt. Ist das strukturierte Attribut ein Kann-Attribut, das nur selten einen Wert erhält, dann wird in der separaten Tabelle kein Speicherplatz benötigt. Der Nachteil dieser zusätzlichen Tabelle besteht jedoch darin, daß zur »Konstruktion« des Objekts aus mehreren Tabellen zusätzliche *join*-Operationen notwendig sind.

komplexer Attributtyp

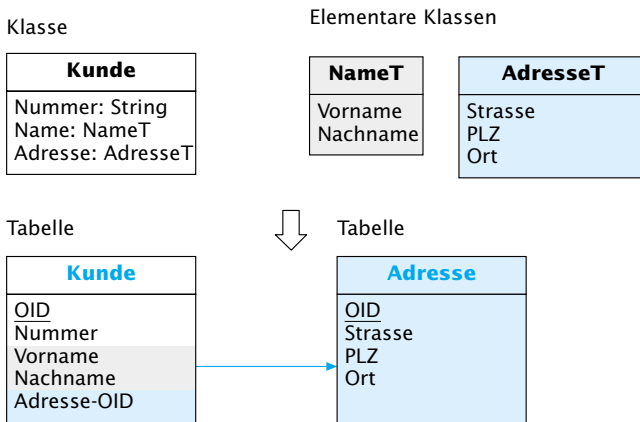
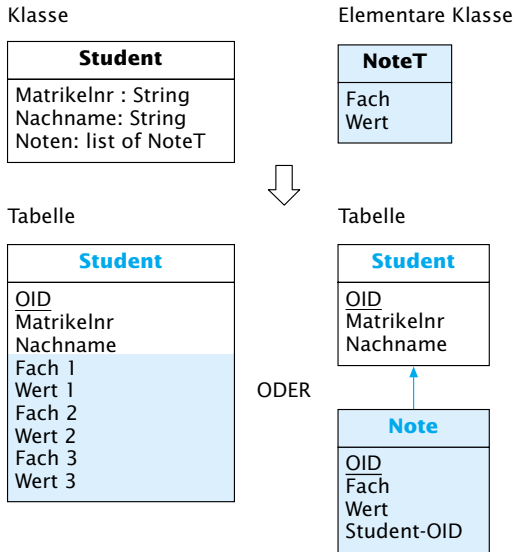


Abb. 8.3-2: Abbildung eines strukturierten Attributs

Enthält die Klasse Attribute vom **Listen-Typ** (Abb. 8.3-3) und ist für die Liste eine feste Obergrenze bekannt, dann können diese Attribute in die Tabelle der Klasse eingetragen werden. Ist die Obergrenze der Liste variabel oder besitzen meist nur wenige Elemente der Liste Werte, dann ist die Liste auf eine eigene Tabelle abzubilden.

## LE 14 8 Datenbanken

Abb. 8.3-3:  
Abbildung eines  
Listenattributs

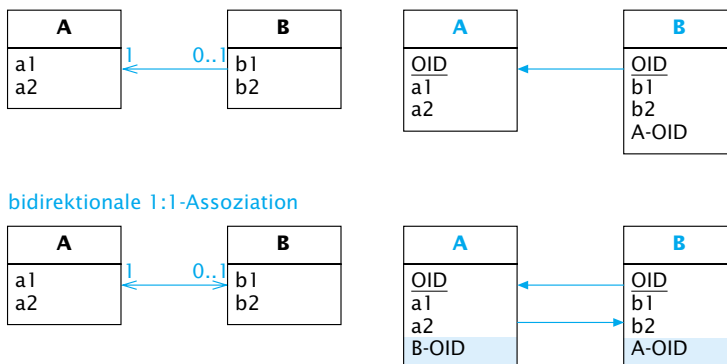


**Klassenattribut** Klassenattribute sind nur einmal für alle Objekte einer Klasse zu speichern. Daher ist es nicht sinnvoll, sie in »normale« Tupel einer Tabelle zu integrieren, sondern sie werden in eine separate Tabelle eingetragen.

### Abbildung einer Assoziation

**1:1-Assoziation** Eine **1:1-Assoziation** kann je nach benötigter Navigation in eine oder beide Tabellen als Fremdschlüssel eingetragen werden. Bei einer Muß-Verbindung ist das entsprechende Attribut mit *not null* zu kennzeichnen. Bei der unidirektionalen Muß-Assoziation der Abb. 8.3-4 ist der Fremdschlüssel A-OID als *not null* einzutragen. Bei der bidirektionalen Assoziation ist nur A-OID als *not null* zu kennzeichnen, während B-OID ein Kann-Attribut ist.

Abb. 8.3-4:  
Abbildung der 1:1-  
Assoziation



### 8.3 Abbildung des objektorientierten Modells auf Tabellen LE 14

Bei einer **1:m-Assoziation** (Abb. 8.3 -5) können wir die Assoziation mit der Tabelle auf der *many*-Seite »verschmelzen«, wenn die Navigation zur *one*-Tabelle benötigt wird. Auch hier ist bei einer Muß-Verbindung das entsprechende Fremdschlüssel-Attribut mit *not null* zu kennzeichnen. Wenn beide Navigationsrichtungen erforderlich sind, dann ist die Assoziation auf eine separate Tabelle abzubilden. Das Verschmelzen der Assoziation mit einer Tabelle besitzt folgende Vorteile. Es gibt weniger Tabellen und der Zugriff auf Objekte erfolgt schneller, weil weniger Tabellen durchlaufen werden müssen. Demgegenüber besitzt eine separate Tabelle folgende Vorteile. Das Wissen, welche Objekte einander kennen, ist nicht mit den Objekten selbst verwoben und entspricht daher besser dem objektorientierten Ansatz. Wird aus der 1:m-Assoziation später eine m:m-Assoziation, dann ist diese Änderung einfach durchzuführen. Sie können auch beide Möglichkeiten kombinieren, wie die grau dargestellte Erweiterung der Abb. 8.3-5 zeigt. Diese Lösung reduziert bei Navigationen von B nach A die Anzahl der *joins*, weil in diesem Fall kein Zugriff auf die Tabelle AB erforderlich ist. Sie ist daher zu wählen, wenn viele Zugriffe dieser Art zu erwarten sind. Nachteilig wirkt sich bei dieser Kombination aus, daß zusätzliche Konsistenzmaßnahmen im Falle von Änderungen notwendig sind.

1:m-Assoziation

#### »Verschmelzen« der Assoziation

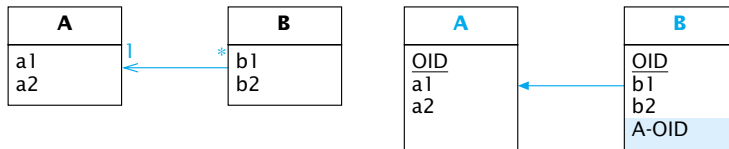
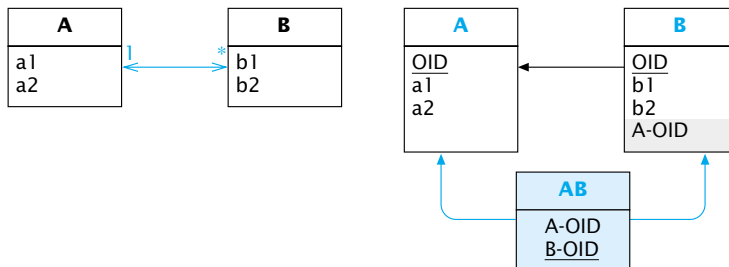


Abb. 8.3-5:  
Abbildung der  
1:m-Assoziation

#### Eigene Tabelle für Assoziation



Bei einer **m:m-Assoziation** wird die Assoziation immer auf eine eigene Tabelle abgebildet (Abb.8.3-6). Der Primärschlüssel dieser Tabelle setzt sich aus den Schlüsseln der beteiligten Tabellen zusammen. Es ist oft vorteilhaft, wenn diese Tabelle ein eigenes OID-Attribut erhält. Dann werden alle Tabellen gleich behandelt, was ihre Implementierung vereinfacht. Ein weiterer Vorteil liegt in der Laufzeit-Effizienz, da bei *joins* von Tabellen mit zusammengesetz-

m:m-Assoziation

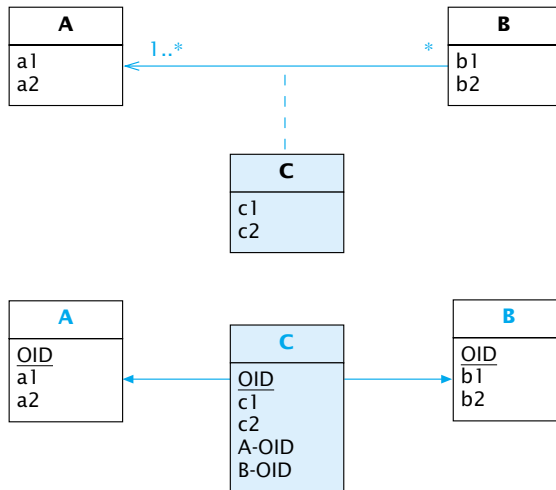


## LE 14 8 Datenbanken

ten Schlüsseln bei einigen Datenbanken Probleme auftreten /Am-  
bler 97/.

assoziative Klasse Existiert eine assoziative Klasse, dann werden deren Attribute ebenfalls in diese Tabelle eingetragen (Abb. 8.3-6). Auch bei einer 1:1- oder einer 1:m-Assoziation sollten Sie eine assoziative Klasse auf eine eigene Tabelle abbilden, sofern sie nicht nur aus einem oder zwei Attributen besteht.

Abb. 8.3-6:  
Abbildung einer  
m:m-Assoziation  
bzw. einer assozia-  
tiven Klasse



Komposition Aus Sicht einer Datenbank besteht der einzige Unterschied zwischen einer einfachen Assoziation und einer Komposition darin, wie hoch die Kopplung zwischen den beteiligten Objekten ist. Bei einer Komposition wirkt sich die Funktionalität des Ganzen auch auf seine Teile aus. Gleiches gilt für das Speichern und Löschen von Objekten in der Datenbank. Daraus resultiert für die Abbildung einer Komposition, daß jedes Ganze seine Teile kennen muß.

### Abbildung der Einfachvererbung

Es gibt drei Möglichkeiten, um eine Vererbungshierarchie auf Tabellen abzubilden (Abb. 8.3-7).

1 Tabelle für  
Hierarchie

Bei der **ersten Variante** werden alle Attribute aus allen Klassen einer Hierarchie in einer einzigen Tabelle gespeichert. Alle Attribute, die ein Objekt bzw. ein Tupel nicht annehmen kann, müssen auf *null* gesetzt werden. Der Vorteil dieses Ansatzes liegt in seiner Einfachheit. *Ad hoc*-Anfragen sind einfach, weil alle Daten in einer einzigen Tabelle liegen und keine *joins* notwendig sind. Dem stehen folgende Nachteile gegenüber: Die Kopplung innerhalb der Vererbungshierarchie wird erhöht. Wird eine Klasse der Hierarchie um ein neues Attribut erweitert, dann sind alle Objekte der Hierarchie davon betroffen. Außerdem verschwendet dieser Ansatz Speicher-

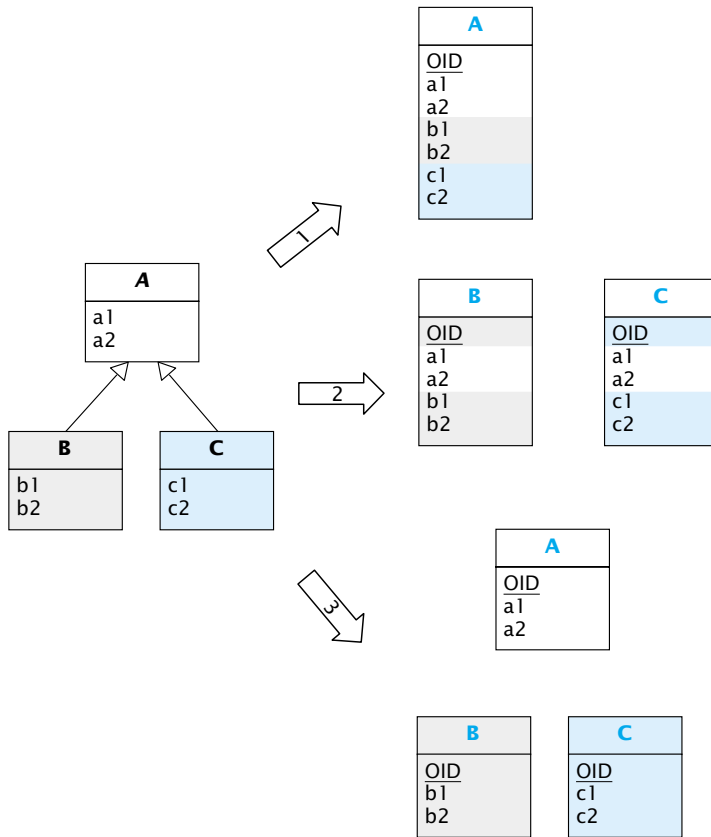


Abb. 8.3-7:  
Abbildung von  
Vererbungs-  
strukturen

platz, weil die Tabelle viele *null*-Werte enthält. Dieser Nachteil ist jedoch bei Vererbungsstrukturen von geringem Umfang vernachlässigbar.

Bei der **zweiten Variante** wird jede konkrete Klasse auf eine Tabelle abgebildet. Sie enthält außer ihren eigenen auch alle Attribute ihrer Oberklassen. Auch bei dieser Form sind *ad hoc*-Anfragen relativ einfach. Nachteilig ist jedoch, daß die Attribute der abstrakten Oberklasse in mehreren Tabellen vorhanden sind. Wenn diese Attribute modifiziert werden, dann sind alle betroffenen Tabellen zu aktualisieren.

1 Tabelle für jede  
konkrete Klasse

Bei der **dritten Variante** wird jede Klasse – auch eine abstrakte – auf eine Tabelle abgebildet. Die Identität eines Objekts in der Vererbung wird durch die Verwendung eines gemeinsamen OID-Attributs sichergestellt. Der Hauptvorteil dieses Ansatzes ist, daß er am besten dem objektorientierten Konzept entspricht. Änderungen in der Oberklasse sind mit minimalem Aufwand durchführbar und neue Attribute können in allen Klassen einfach ergänzt werden. Dem stehen jedoch mehrere Nachteile gegenüber. Es entstehen viele Tabel-

1 Tabelle für jede  
Klasse

len in der Datenbank, die Zugriffe auf Objekte dauern länger, weil mehrere Tabellen betroffen sind (*joins*). Sofern keine *views* aufgebaut werden, sind *ad hoc*-Anfragen schwieriger zu formulieren.

Die Tabelle 8.3-1 zeigt die Vor- und Nachteile dieser Alternativen im Überblick.

Tab. 8.3.-1:  
Vergleich von  
Abbildungs-  
strategien der  
Einfachvererbung  
/Amber 97/

Betrachter Faktor	1 Tabelle für gesamte Hierarchie	1 Tabelle für jede konkrete Klasse	1 Tabelle für jede Klasse
Einfachheit der Implementierung	einfach	mittel	schwierig
Einfachheit des Datenzugriffs	einfach	einfach	mittel/einfach
Kopplung	sehr hoch	hoch	gering
Geschwindigkeit des Datenzugriffs	schnell	schnell	mittel/schnell
Unterstützung des Polymorphismus	mittel	gering	hoch

Wie Sie anhand der obigen Ausführungen erkennen, ist eine objekt-relationale Abbildung ein Prozeß, bei dem viele Faktoren zu berücksichtigen sind. In jedem Fall ist zu überlegen, wie die spätere Nutzung aussieht und mit welchen Datenmengen zu rechnen ist. Separate Tabellen erfordern immer zusätzliche *join*-Operationen und verschlechtern damit die *Performance*. Eine intensive Verschmelzung von Klassen und Assoziationen in eine einzige Tabelle erschwert die Nachvollziehbarkeit vom Analysemodell zum relationalen Datenmodell und kann zu aufwendigen Änderungen führen.

Die objekt-relationale Abbildung ist nur ein Aspekt, der beim Benutzen von relationalen Datenbanken in objektorientierten Anwendungen wichtig ist. In Kapitel 10.6 gehen wir darauf ein, wie die softwaretechnische Anbindung der objektorientierten Anwendung an eine relationale Datenbank entworfen wird.

Kapitel 10.6



**Datenbanksystem** Ein Datenbanksystem besteht aus einer oder mehreren Datenbanken, einem *Data Dictionary* und einem Datenbankmanagementsystem. In der Datenbank (DB) sind alle Daten gespeichert. Das *Data Dictionary* (DD) enthält das Datenbankschema, das den Aufbau der Daten der Datenbank(en) beschreibt. Die Verwaltung und zentrale Kontrolle der Daten ist Aufgabe des Datenbankmanagementsystems (DBMS).

**Datendefinitionssprache (data definition language)** Die Datendefinitionssprache (DDL) ist eine Sprache, die ein  $\rightarrow$ relationales Datenbanksystem zur Verfügung stellt und die zur forma-

len Definition des logischen Schemas – d.h. den leeren  $\rightarrow$ Tabellen der relationalen Datenbank – dient. Als Standard hat sich die Sprache  $\rightarrow$ SQL etabliert.

**Datenmanipulationssprache (data manipulation language)** Die Datenmanipulationssprache (DML) dient dazu, die leeren  $\rightarrow$ Tabellen einer relationalen Datenbank mit Daten zu füllen und diese Daten zu ändern. Eine DML enthält keine Kontrollstrukturen und Prozedurkonzepte. Als Standard hat sich die Sprache  $\rightarrow$ SQL etabliert.

**Datenmodell** Jedem Datenbanksystem liegt ein Datenmodell zugrunde, in dem festgelegt wird, welche Eigenschaften und Strukturen die Datenele-



mente besitzen dürfen, welche Konsistenzbedingungen einzuhalten sind und welche Operationen zum Speichern, Suchen, Ändern und Löschen von Datenelementen existieren. Es lassen sich relationale und objektorientierte Datenmodelle unterscheiden.

**DLL (Data Definition Language)**

→Datendefinitionssprache

**DML (Data Manipulation Language)**

→Datenmanipulationssprache

**Objektrelationale Abbildung (object relational mapping)** Die objektrelationale Abbildung gibt an, wie ein Klassendiagramm auf

→Tabellen einer relationalen Datenbank abgebildet wird. Sie enthält Abbildungsvorschläge für Klassen, Assoziationen und Vererbungsstrukturen. Ein weiterer Aspekt ist die Realisierung der Objektidentität in relationalen Datenbanken.

**Relation** →Tabelle

**Relationales Datenbanksystem (relational database system)** Ein relationales Datenbanksystem (RDBS) ist ein

Datenbanksystem, dem ein relationales →Datenmodell zugrunde liegt. Die Daten werden in Form von →Tabellen gespeichert.

**SQL (Structured Query Language)**

SQL ist eine deklarative Programmiersprache, d.h. sie besitzt im Unterschied zu den klassischen Programmiersprachen keine Schleifen, keine Prozeduren, keine Rekursion und keine ausreichenden mathematischen Operationen. Sie dient der Definition und Manipulation relationaler Datenbanken. 1983 wurde von ANSI und ISO ein SQL-Standard definiert. Weiterentwicklungen führten zum derzeitigen Standard SQL2, der 1992 veröffentlicht wurde, und zu SQL3 (noch nicht verabschiedet).

**Tabelle (table)** →Relationale Datenbanksysteme speichern Daten in Form von Tabellen (Relationen). Jede Zeile der Tabelle wird als Tupel bezeichnet. Alle Tupel einer Tabelle müssen gleich lang sein. Jedes Tupel muß durch einen eindeutigen Schlüssel identifizierbar sein. Der Schlüssel (auch als Primärschlüssel bezeichnet) kann aus einem oder mehreren Attributen bestehen. Beziehungen zwischen Tabellen werden mittels Fremdschlüsseln realisiert.



**Datenbanksysteme** dienen der persistenten Speicherung von Daten. Je nach dem zugrundeliegenden **Datenmodell** spricht man von einem objektorientierten oder **relationalen Datenbanksystem**. Auch bei einer objektorientierten Entwicklung werden häufig relationale Datenbanksysteme eingesetzt, bei denen alle Daten in **Tabellen** gespeichert werden. Als **Datendefinitions-** und **Datenmanipulations-**sprache wird im allgemeinen **SQL** verwendet. Um die Daten einer objektorientierten Anwendung in einer relationalen Datenbank zu speichern, ist eine **objektrelationale Abbildung** durchzuführen.

**1 Lernziel: Unterschiede des relationalen und objektorientierten Modells erkennen.**

- a** Erläutern Sie den Unterschied zwischen dem Schlüsselattribut einer relationalen Datenbank und der Objektidentität der objektorientierten Modellierung.
- b** Erläutern Sie den Unterschied zwischen einer Assoziation und der Schlüssel-Fremdschlüssel-Beziehung.
- c** Erläutern Sie, warum bei einem objektorientierten Modell die erste Normalform nicht eingehalten werden muß.
- d** Was ist ein OID-Attribut und welche Vorteile ergeben sich durch dessen Verwendung bei einer relationalen Datenbank?

Aufgabe  
5–10 Minuten

## LE 14 Aufgaben

Aufgabe 2 *Lernziel: Wichtige Begriffe von relationalen Datenbanksystemen kennen.*  
5 Minuten

Erläutern Sie die folgenden Begriffe:

- a** logisches Schema,      **b** externes Schema,      **c** DDL,  
**d** DML,                      **e** SQL.

Aufgabe 3 *Lernziel: Objekt-relationale Abbildung durchführen können.*  
10 Minuten

Bilden Sie die folgende Klasse Studentische Hilfskraft auf eine oder mehrere Tabellen einer relationalen Datenbank ab. Stellen Sie die Tabellen grafisch dar.

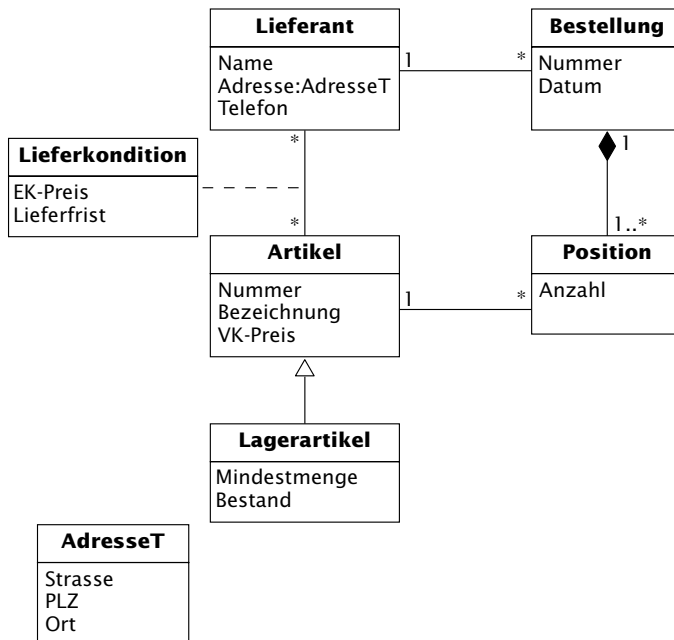
- 7-stellige Matrikelnummer
- Name bestehend aus Vorname und Nachname
- Adresse bestehend aus Straße, PLZ und Ort
- Liste aller Arbeitsverträge, wobei für jeden Arbeitsvertrag Beginn, Ende und vereinbarte Stundenzahl gespeichert wird.
- Aktueller Stundenlohn, der für alle studentischen Hilfskräfte gleich ist (Klassenattribut).

Die Menge der Arbeitsverträge darf nicht begrenzt werden. Der Stundenlohn soll nur einmal gespeichert werden.

Aufgabe 4 *Lernziel: Objekt-relationale Abbildung durchführen können.*  
15–20 Minuten

Bilden Sie das Klassendiagramm der Abb. LE14-A4 auf Tabellen einer relationalen Datenbank ab. Stellen Sie die Tabellen grafisch dar.

Abb. LE14-A4:  
Klassendiagramm  
Bestellwesen



**5 Lernziele: DDL und DML anwenden können.**

Aufgabe  
15–20 Minuten

Verwenden Sie als DDL und DML die Sprache SQL.

- a** Definieren Sie für die Aufgabe 4 das logische Schema. Gehen Sie davon aus, daß Bestell- und Artikelnummern jeweils eindeutig sind.
- b** Erstellen Sie folgendes externe Schema:  
Liste aller Artikel, auch der Lagerartikel, die in höchstens einer Woche geliefert werden können. Das Schema soll enthalten: Artikelnummer, Artikelbezeichnung, Lieferantename und EK-Preis für diesen Lieferanten.
- c** Formulieren Sie folgende Anfrage:  
Benötigt wird eine Liste aller Lagerartikel, bei denen die Mindestmenge unterschritten ist. Die Liste soll enthalten: Nummer, Bezeichnung, Bestand, Mindestmenge.
- d** Formulieren Sie folgende Anfrage:  
Für jeden Lieferanten ist eine Liste der von ihm gelieferten Artikel mit EK-Preis und Lieferfrist zu erstellen. Die Liste soll folgende Angaben enthalten: Lieferantename, Artikelbezeichnung, EK-Preis und Lieferfrist.

# 8 Datenbanken

## Objektorientierte Datenbanken



- Wissen, was der ODMG-Standard ist und was seine wichtigsten Komponenten sind. wissen
- Wissen, was ein objekt-relationales Datenbanksystem ist. verstehen
- Erklären können, was die wesentlichen Eigenschaften von objektorientierten Datenbanksystemen sind. verstehen
- Erklären können, wie die Anbindung an die objektorientierte Programmiersprache C++ funktioniert. anwenden
- Klassendiagramme in ODL spezifizieren können. anwenden
- Anfragen in OQL formulieren können. anwenden



- Die objektorientierten Konzepte und die UML-Notation, wie sie in Kapitel 2 und Kapitel 6 beschrieben werden, müssen bekannt sein.
- Die Kapitel 8.1 und 8.3 sollten bekannt sein, um die Unterschiede zwischen objektorientierten und relationalen Datenbanken zu verstehen.

- i 8.4 Objektorientierte Datenbanksysteme 326
- 8.5 ODL 332
- 8.6 OQL 335
- 8.7 Anbindung an C++ 339
- 8.8 Objekt-relationale Datenbanksysteme 345

## 8.4 Objektorientierte Datenbanksysteme

Definition Ein **objektorientiertes Datenbanksystem** integriert die Eigenschaften einer Datenbank mit den Möglichkeiten von objektorientierten Programmiersprachen. Nach *object-oriented data-base system manifesto* (siehe /Dittrich, Geppert 95/ und /Heuer 97/) muß ein objektorientiertes Datenbanksystem alle funktionalen Eigenschaften eines klassischen Datenbanksystems besitzen. Dazu kommt die Forderung, daß ihm ein **objektorientiertes Datenmodell** zugrunde liegt, was die Unterstützung folgender Konzepte bedeutet:

- Objektidentität.
- direkte Repräsentation zusammengesetzter Objekte.
- Klassen.
- Definierbarkeit von Klassen durch den Systembenutzer, d.h. es müssen Mechanismen angeboten werden, mit denen der Benutzer nach der Installation des Systems weitere Klassen – insbesondere Unterklassen – hinzufügen kann.
- Berechnungsvollständige Sprache, welche die Formulierung beliebiger Algorithmen gestattet.
- Geheimnisprinzip, d.h. Zustand und Implementierung der Objekte sind für den Objektbenutzer unsichtbar.

Um das Geheimnisprinzip zu realisieren, erfolgt der Zugriff auf die Attribute eines Objekts – genau genommen – nur mittels Operationen. Dadurch werden aber flexible Anfragen (*queries*) erschwert. Ein Kompromiß ist dadurch entstanden, daß im Falle von *ad hoc queries* auf das Geheimnisprinzip verzichtet wird, was zu einer bedeutenden Vereinfachung führt.

- Klassenhierarchien und Vererbung.
- Überladen, Überschreiben und spätes Binden.

Der Standard für objektorientierte Datenbanksysteme wird von der ODMG (*Object Database Management Group*) definiert. Um von den Eigenschaften eines bestimmten Datenbanksystems zu abstrahieren, erläutere ich die Eigenschaften objektorientierter Datenbanksysteme anhand des ODMG-Standards. Im Exkurs 2 wird eine einfache Realisierung mit dem objektorientierten Datenbanksystem Poet beschrieben.

Exkurs 2

www.odmg.org

1991 gründeten Hersteller und Anwender von objektorientierten Datenbanksystemen die **ODMG** (*Object Database Management Group*), die der OMG (*Object Management Group*) beigeordnet ist. 1993 wurde von dieser Gruppe ein Standard für objektorientierte Datenbanksysteme vorgeschlagen, ODMG-93 genannt. Der aktuelle **ODMG-Standard** 2.0 vom Juni 1997 /Cattell, Barry 97/ besteht aus:

- Objektmodell.  
Es spezifiziert die Konzepte, die von einem objektorientierten Datenbanksystem unterstützt werden.



- ODL (*Object Definition Language*).  
Die ODL ist eine Schema-Beschreibungssprache, die alle Elemente des Objektmodells abdeckt.
- OIF (*Object Interchange Format*).  
OIF ist ein ASCII-Austauschformat für die Inhalte von objektorientierten Datenbanksystemen.
- OQL (*Object Query Language*).  
Diese deklarative Sprache, die SQL2 ähnlich ist, ermöglicht Anfragen (*queries*) in der Datenbank.
- Sprachanbindung an C++.  
Hier wird festgelegt, wie portabler Quellcode in C++ geschrieben wird, der persistente Objekte verwendet.
- Sprachanbindung an Smalltalk (analog zu C++).
- Sprachanbindung an Java (analog zu C++).

Es gibt zwei Stufen der Einhaltung des Standards: ODMG-konform (*ODMG compliant*) und ODMG-zertifiziert (*ODMG certified*). Zertifizierte Datenbanksysteme haben vorgegebene Testprogramme (*test suites*) erfolgreich durchlaufen. Konforme Datenbanksysteme behaupten lediglich, dem Standard zu entsprechen. Aussagen über Konformität und Zertifizierung können auch nur für eine Komponente abgegeben werden, z.B. für die Sprachanbindung an C++.

Das **ODMG-Objektmodell** (*object model*) spezifiziert, welche Konzepte von einem objektorientierten Datenbanksystem unterstützt werden müssen. Die grundlegenden Elemente sind Objekte und Literale. Desweiteren definiert das Objektmodell bereits bekannte Konzepte wie Typen (*types*), Operationen (*operations*), Attribute (*attributes*) und Assoziationen (*relationships*).

ODMG-Objektmodell

**Objekte** (*objects*) besitzen eine Objektidentität und können separat in der Datenbank gespeichert werden. **Literale** (*literals*) sind Daten, die keine Objektidentität besitzen. Sie können nur als Teil eines Objekts in der Datenbank gespeichert werden. Der ODMG-Standard fordert strenge Typisierung, d.h. für jedes Objekt und jedes Literal muß dessen Typ angegeben werden.

Objekt  
Literal

Jedes Objekt besitzt eine eindeutige **Objektidentität** (OID, *object identity*), die es von allen anderen Objekten unterscheidet. Sie ändert sich während der Lebensdauer des Objekts nicht. Objektidentitäten werden vom Datenbanksystem generiert und verwaltet. Sie besitzen keine (verwendbare) Semantik und sind dem Programmierer nicht bekannt. Im ODMG-Standard wird nicht festgelegt, wie Objektidentitäten zu implementieren sind. Die Objektidentität ist eine notwendige Voraussetzung dafür, daß in der Datenbank Verbindungen zwischen Objekten dargestellt werden können.

Objektidentität

Es gibt mehrere Arten, die Objektidentität in objektorientierten Datenbanksystemen zu realisieren /Bertino, Martino 93/. Dazu gehören Surrogate und typisierte Surrogate.

Realisierung der  
Objektidentität

## LE 15 8 Datenbanken

Ein Surrogat (*surrogate*) wird durch einen Algorithmus erzeugt, der Eindeutigkeit garantiert (z.B. ein Zeitstempel bestehend aus Datum und Uhrzeit oder ein Zähler, der kontinuierlich erhöht wird). Ein typisiertes Surrogat (*typed surrogate*) besteht aus einem Klassen-Identifikator (*type ID*) und einem Objekt-Identifikator, der für jede Klasse individuell inkrementiert wird.

**Objektname** Zusätzlich zur Objektidentität, deren Wert der Programmierer nicht kennt, kann ein Objekt einen oder mehrere Namen besitzen, die der Programmierer zur Identifikation des Objekts benutzt. Objektnamen sind mit den globalen Variablen einer Programmiersprache vergleichbar. Jeder Objektname muß innerhalb einer Datenbank eindeutig sein.

**Literaltyp** Der ODMG-Standard unterscheidet vier Literaltypen:

- atomare Literale,
- Kollektionen von Literalen,
- strukturierte Literale und das
- Null-Literal, das dem Null-Wert bei SQL2 entspricht.

Typen für **atomare Literale** sind:

- long, short, unsigned long, unsigned short (für ganze Zahlen),
- float, double (für reelle Zahlen),
- boolean,
- octet,
- char, string und
- enum (für Aufzählungstypen).

### Beispiel long Artikelnummer

Bei einer **Kollektion** sind alle Elemente vom selben Typ. Als Elementtyp sind Objekt- und Literaltypen erlaubt. Für Kollektionen von Literalen sind definiert:

- set<t> (ungeordnete Kollektion von Elementen, in der keine Duplikate erlaubt sind),
- bag<t> (ungeordnete Kollektion von Elementen, die Duplikate besitzen kann),
- list<t> (geordnete Kollektion von Elementen),
- array<t> (geordnete Kollektion von Elementen, in der jedes Element durch seine Position lokalisiert werden kann),
- dictionary<t, value> (ungeordnete Sequenz von Paaren <Schlüssel, Wert>, wobei keine Duplikate des Schlüssels erlaubt sind).

**Beispiele** set <Artikel> gelieferte Artikel  
list <Person> allePersonen

Eine **Struktur** besteht aus einer festen Anzahl von Elementen, von denen jedes einen Namen besitzt und entweder einen Literalwert oder ein Objekt enthalten kann. Für strukturierte Literale sind folgende Typen definiert:

- date,
- time (gibt die aktuelle Zeit in der jeweiligen Zeitzone an),
- interval (beschreibt eine Zeitdauer),
- timestamp (besteht aus Datum und Zeit).

date Rechnungsdatum

Beispiel

Außerdem ist die Bildung beliebiger Datenstrukturen durch den Programmierer möglich.

```
struct AdresseT
{
    string Gebaeude;
    string Raumnummer;
};
```

Beispiel

AdresseT Bueroadresse;

Auch alle Objekte der Datenbank werden durch ihren Typ spezifiziert. Der ODMG-Standard unterscheidet:

Objektyp

- atomare Objekte, die vom Programmierer definiert werden,
- Kollektionen von Objekten und
- strukturierte Objekte.

Die Typen für Kollektionen und Strukturen sind analog zu den Literalypen definiert, wobei Objekttypen – im Unterschied zu den Literalypen – immer mit einen Großbuchstaben beginnen.

### Klassen und Schnittstellen

Der ODMG-Standard unterscheidet Klassen (*classes*) und Schnittstellen (*interfaces*).

Eine Klasse (*class*) definiert für ihre Objekte das Verhalten (Operationen) und den Zustand (Attribute, Assoziationen). Für jede Klasse können optional die Klassenextension (*extent*) und ein Schlüssel (*key*) angegeben werden.

class

```
class Klassenname
    (extent Klassenextension
    key Schlüsselattribut)
```

Beispiel

Unter der **Klassenextension** (*extent*) ist die Menge aller Objekte einer Klasse zu verstehen. Der Programmierer kann entscheiden, ob eine Klassenextension notwendig ist. Falls der *extent* angelegt wird, wird neu erzeugtes Objekt automatisch eingefügt und beim Löschen wieder entfernt. Das Konzept der Klassenextension realisiert bei objektorientierten Datenbanksystemen die Objektverwaltung und ermöglicht die Durchführung von Operationen (z.B. Selektionen) auf der Menge aller Objekte einer Klasse. Wenn also für eine Klasse keine Extension definiert wurde, dann können deren Objekte nicht selektiert werden. Die Klassenextension wird dementsprechend im *select*-Befehl von OQL verwendet werden.

extent

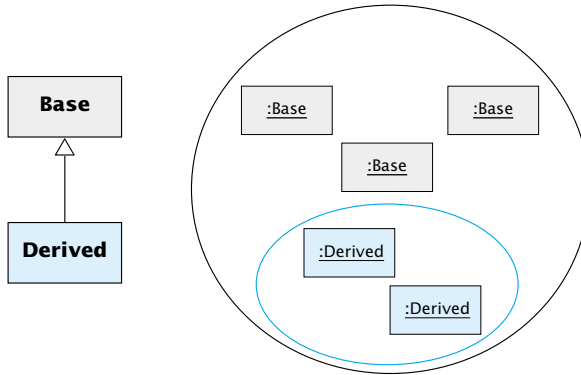
Zwischen der Klassenextension und der Vererbung besteht folgender Zusammenhang: Gehört ein Objekt zur Klasse Base, dann ist

Klassenextension und Vererbung

## LE 15 8 Datenbanken

es auch ein Element der Klassenextension von Base. Ist *Derived* eine Unterklasse von *Base*, dann ist die Klassenextension von *Derived* eine Teilmenge der Klassenextension von *Base*, d.h. jedes Objekt von *Derived* ist auch in der Klassenextension von *Base* enthalten (Abb. 8.4-1).

Abb. 8.4-1:  
Klassenextension  
und Vererbung



*key* Schlüssel werden für einen schnellen Zugriff auf die Objekte verwendet. Der Schlüssel besteht aus einem (*key*) oder mehreren (*keys*) Attributen. Jeder Schlüssel muß ein Objekt der Klasse eindeutig identifizieren. Die Definition der Klassenextension ist Voraussetzung für die Verwendung eines Schlüssels.

*interface* Außer der Klasse gibt es die Schnittstelle (*interface*), die nur das Verhalten spezifiziert. Von einer Schnittstelle können – im Gegensatz zur Klasse – keine Objekte erzeugt werden können.

Beispiel `interface Schnittstellenname`

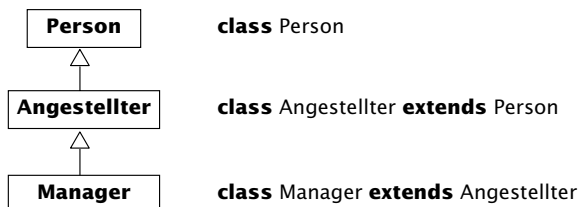
### Vererbung

Der ODMG-Standard definiert zwei Arten von Vererbungsstrukturen:

- *extends* und
- *subtyping* (auch *is a*- oder *ISA*-Vererbung genannt).

*extends* Die *extends*-Vererbung ist eine Einfachvererbung zwischen zwei Klassen, wobei die Unterklasse die Eigenschaften und das Verhalten der Oberklasse erbt. Die *extends*-Beziehung ist transitiv. Abb. 8.4-2 spezifiziert die *extends*-Vererbung in UML- und in ODL-Notation.

Abb. 8.4-2:  
*extends*-Vererbung



Die *subtyping*- oder *ISA*-Vererbung (*is a*) bezieht sich auf die Vererbung von Verhalten. Sie wird in ODL durch einen Doppelpunkt spezifiziert. Bei der *ISA*-Vererbung dürfen sowohl Klassen als auch Schnittstellen von einer Schnittstelle (*interface*) abgeleitet werden. Mittels *subtyping* ist auch die Mehrfachvererbung möglich. Abb. 8.4-3 spezifiziert beide Formen der Vererbung in UML- und in ODL-Notation.

*subtyping*

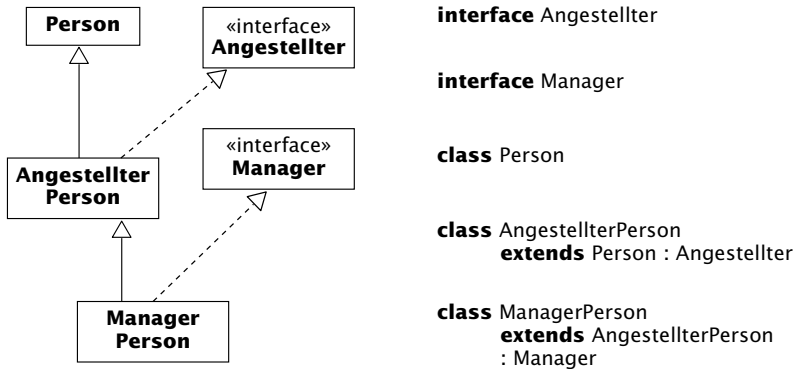


Abb. 8.4-3: subtyping-Vererbung

Bei objektorientierten Datenbanken wurde auf eine enge Anbindung an Programmiersprachen geachtet. Abb. 8.4-4 zeigt den typischen Umgang mit einem objektorientierten Datenbanksystem. Der Programmierer schreibt zunächst eine Schemadeklaration, die vom

Anbindung an Programmiersprachen

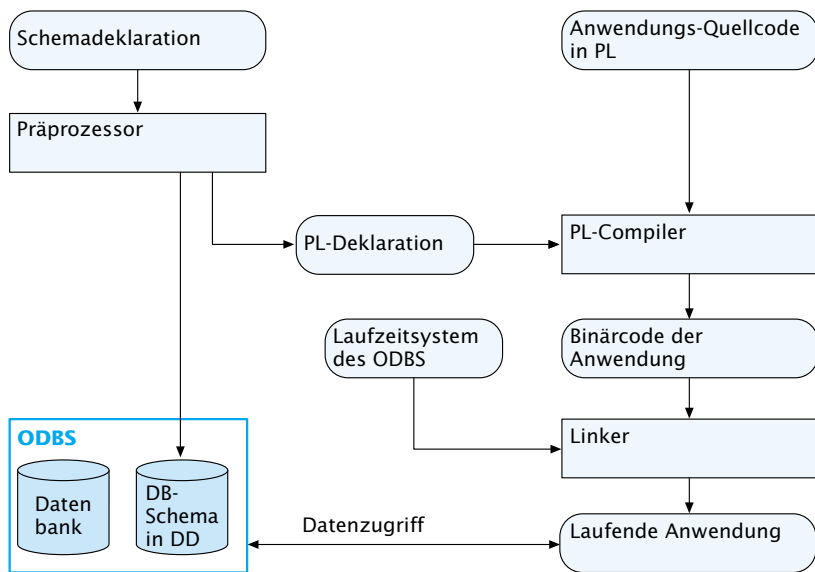



Abb. 8.4-4: Funktionsweise eines objektorientierten Datenbanksystems

Legende: PL = programming language, z.B. C++ oder Java  
DD = data dictionary

Präprozessor in das Datenbankschema und einen Deklarationsteil der Programmiersprache (PL-Deklaration) übersetzt wird. Diese Schemadeklaration kann entweder in ODL oder in PL-ODL (z.B. C++ ODL) erfolgen. Desweiteren erstellt ein Programmierer das Anwendungsprogramm in einer objektorientierten Programmiersprache. Dabei müssen Klassenbibliotheken benutzt werden, die eine Manipulation der Datenbank erlauben. Die PL-Deklaration und das Anwendungs-Quellprogramm werden übersetzt und mit dem Laufzeitsystem der Datenbank gebunden, damit eine lauffähige Anwendung entsteht.

## 8.5 ODL

Zur Schemadeklaration kann die **Objektdefinitionssprache ODL** (*Object Definition Language*) eingesetzt werden. ODL unterstützt alle Konzepte des ODMG-Objektmodells. Es handelt sich nicht um eine vollwertige Programmiersprache, sondern diese Sprache dient ausschließlich zur Spezifikation von Klassen und Schnittstellen. Dabei ist jede Spezifikation vollkommen losgelöst von ihrer Implementierung.

Kapitel 9 ODL ist syntaktisch eine Erweiterung von IDL (*Interface Definition Language*), der Sprache, die von der OMG als Teil des CORBA-Standards entwickelt wurde (vergleiche Kapitel 9). Von ihrer Intension unterscheiden sich beide Sprachen jedoch. Während die IDL das Verhalten von Objekten spezifiziert, dient die ODL zur Beschreibung von persistenten Daten. 

Zu einer ODL-Spezifikation gibt es im Normalfall eine Implementierung in C++, Java oder Smalltalk. Es können auch mehrere Implementierungen – beispielsweise für verschiedene Plattformen – existieren.

Attribut in ODL Ein Attribut ist entweder ein Literal (*literal*) oder die Identität (OID) eines Objekts. Es kann niemals ein Objekt sein. Attribute, die nur gelesen werden dürfen, sind als *readonly* gekennzeichnet. Klassenattribute können in ODL *nicht* spezifiziert werden.

Beispiel Die Attribute *Nummer*, *Name*, *Geburtsdatum* und *Alter* sind Literale der Klasse *Person*, d.h. die entsprechenden Werte werden direkt in jedem Objekt der Klasse *Person* gespeichert. Das Attribut *Wohnung* ist ein »Zeiger« auf ein Objekt der Klasse *AdresseT* bzw. die OID dieses Objekts. Dagegen ist das Attribut *Name* eine selbst definierte Literalstruktur, deren Elementwerte direkt im Objekt der Klasse *Person* gespeichert sind (Abb.8.5-1).

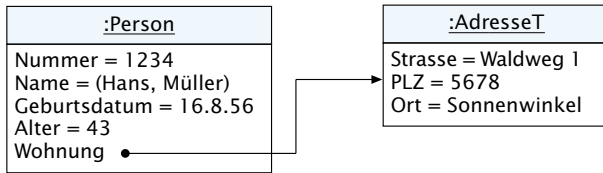


Abb. 8.5-1:  
Realisierung von  
komplexen  
Objekten

```
class AdresseT
{ attribute string Strasse;
  attribute string PLZ;
  attribute string Ort;
};
class Person
{ struct NameT
  { string Vorname,
    string Nachname
  };
  attribute long Nummer;
  attribute NameT Name;           //Datenstruktur (Literal)
  attribute date Geburtsdatum;
  readonly attribute short Alter; //readonly-Attribut
  attribute AdresseT Wohnung;    //OID von AdresseT-Objekt
};
```

Der ODMG-Standard unterstützt nur binäre Assoziationen, d.h. Assoziationen zwischen zwei Objekten. Sie werden in der ODL stets bidirektional dargestellt. Die Angabe *inverse* sorgt dafür, daß die Assoziation in beiden Richtungen navigiert werden kann.

Assoziation in  
ODL

Eine Unterscheidung zwischen einfacher Assoziation, Aggregation und Komposition erfolgt nicht. Es lassen sich nur binäre Assoziationen mit den Kardinalitäten 1:1, 1:m und m:m spezifizieren, wobei nicht zwischen Muß- und Kann-Assoziationen unterschieden wird. Die Spezifikation von assoziativen Klassen ist nicht möglich.

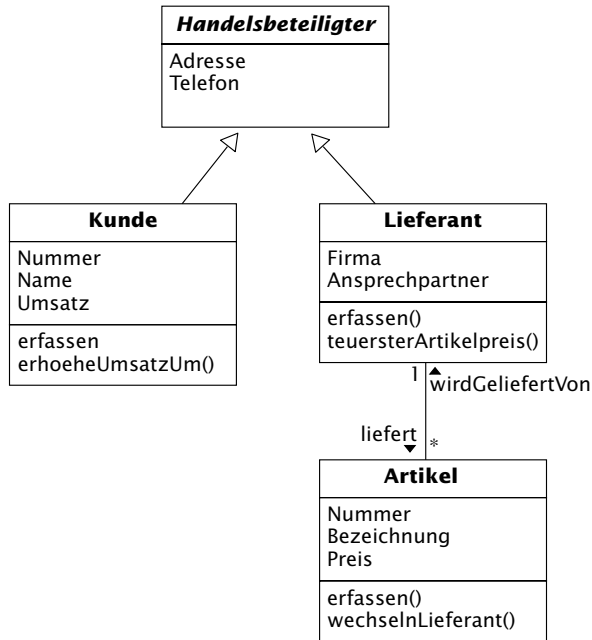
Die Assoziation zwischen Artikel und Lieferant (Abb. 8.5-2) wird in ODL sowohl in der Klasse Artikel als auch in Lieferant spezifiziert. Anstelle der Assoziationsnamen können auch Rollennamen verwendet werden.

Beispiel

```
class Artikel
{ ...
  relationship Lieferant wirdGeliefertVon //one-Richtung
    inverse Lieferant::liefert;
}
class Lieferant extends Handelsteiliger
{ ...
  relationship set <Artikel> liefert //many-Richtung
    inverse Artikel::wirdGeliefertVon;
}
```

## LE 15 8 Datenbanken

Abb. 8.5-2:  
Klassendiagramm  
zum Beispiel



Operation in ODL Für jede Operation ist die **Signatur** anzugeben. Sie enthält die Ein-/Ausgabeparameter einschließlich ihrer Typen und mögliche Ausnahmebehandlungen (*exceptions*). Der prinzipielle Aufbau lautet:

```

ErgebnisTyp OperationName( in Typ1 Par1,
                             out Typ2 Par2,
                             inout Typ3 Par3, ... )
raises (Ausnahme1, Ausnahme2, ...)
    
```

Gibt die Operation kein Ergebnis zurück, dann wird void angegeben. Klassenoperationen können in ODL nicht spezifiziert werden.

**Beispiel** Beim Wechsel des Lieferanten wird der neue Lieferant als *in*-Parameter übergeben. Stimmt der neue mit dem alten Lieferanten überein, dann wird ein Fehler (*exception*) gemeldet.

```

class Artikel
{
    ...
    void wechselnLieferant(in Lieferant neuerLieferant)
        raises (gleicherLieferant);
}
    
```

**Beispiel** Für die Klassen Artikel, Handelsbeteiligter und Lieferant (Abb. 8.5-2) ergeben sich folgende Spezifikationen in ODL:

```

class Artikel
( extent ArtikelListe
  key Nummer)
    
```



```

{ attribute long Nummer;
  attribute string Bezeichnung;
  attribute float Preis;
  relationship Lieferant_wirdGeliefertVon
    inverse Lieferant::liefert;
  void erfassen()
    raises (schonVorhanden);
  void wechsel_nLieferant(in Lieferant neuerLieferant)
    raises (gleicherLieferant);
}

class Handel sbeteiligter
{ struct AdresseT
  { string Strasse,
    string PLZ,
    string Ort
  };
  attribute AdresseT Adresse;
  attribute string Telefon;
}

class Lieferant extends Handel sbeteiligter
( extent Lieferanten)
{ attribute string Firma;
  attribute string Ansprechpartner;
  relationship set <Artikel> liefert
    inverse Artikel::wirdGeliefertVon;
  void erfassen()
    raises (schonVorhanden);
  float teuersterArtikelpreis() //ermittelt den Preis des teuersten
    raises (liefertNichts); //Artikel s dieses Lieferanten
}

class Kunde extends Handel sbeteiligter
(extent Kunden)
{ attribute long Nummer;
  attribute string Name;
  attribute float Umsatz;
  void erfassen()
    raises (schonVorhanden);
  void erhoeheUmsatzUm(in float Erhoehung);
}

```

## 8.6 OQL

Anfragen an eine objektorientierte Datenbank können – außer in der verwendeten Programmiersprache – auch mit der **Anfragesprache OQL** (*Object Query Language*) durchgeführt werden. Sie baut auf dem *select-from-where*-Block von SQL2 auf. OQL kann sowohl als eigenständige, interaktive – aber nicht berechnungsvollständige – Datenbanksprache als auch eingebettet in verschiedene Programmiersprachen benutzt werden. OQL stammt von den Sprachen des O<sub>2</sub>-Systems RELOOP, O<sub>2</sub>Query und O<sub>2</sub>SQL ab und wurde

## LE 15 8 Datenbanken

von der Firma O<sub>2</sub>-Technology in den ODMG-Standard eingebracht. OQL erlaubt Anfragen auf die Attributwerte analog zu SQL. Im Gegensatz zu relationalen Datenbanken kann aber auch mit komplexen Datenstrukturen gearbeitet werden.

**Klassenextension** Die Menge aller Objekte einer Klasse läßt sich einfach durch Angabe der Klassenextension oder mittels einer *select*-Anfrage ermitteln. Diese Befehle entsprechen der Anweisung `select * from Tabellenname in SQL2`.

**Beispiele** ■ `ArtikelListe`  
■ `select a`  
`from ArtikelListe a`

**einfache Anfragen** Einfache Anfragen lassen sich sehr leicht analog zu SQL formulieren, wobei OQL – statt der Tabellen in SQL – die Klassenextensionen referenziert. Mehrere Bedingungen werden mittels *and* verknüpft.

**Beispiele** Das Ergebnis der folgenden Anfrage ist die Menge aller Artikel, deren Bezeichnung mit »S« beginnt:

```
select a
from ArtikelListe a
where a.Bezeichnung = "S"
```

Alle Artikel im Nummernkreis von 1000 bis 1999 ermittelt die folgende Anfrage:

```
select a
from ArtikelListe a
where a.Nummer >= 1000 and a.Nummer <= 1999
```

Sollen nicht vollständige Artikelobjekte, sondern nur deren Nummern selektiert werden, dann ist das Ergebnis vom Typ `set<long>`. Die Angabe *distinct* sorgt dafür, daß im Ergebnis jede Nummer maximal einmal vorkommt. Ohne *distinct* wäre das Ergebnis vom Typ `bag<long>`:

```
select distinct a.Nummer
from ArtikelListe a
where a.Nummer >= 1000 and a.Nummer <= 1999
```

Sollen nur die Nummer und die Bezeichnung der selektierten Artikel ausgegeben werden, dann ist das Ergebnis vom Typ `set<struct>`:

```
select distinct struct (Nr: a.Nummer, Bezeichnung:
a.Bezeichnung)
from ArtikelListe a
where a.Nummer >= 1000 and a.Nummer <= 1999
```

**Zugriff in Struktur** Da eine Klasse nicht nur Standardtypen, sondern Datenstrukturen (Literele oder Objekte) enthalten kann, muß OQL auch einen Zugriff innerhalb dieser Datenstrukturen ermöglichen. Dazu verwendet OQL analog zu objektorientierten Programmiersprachen den ».«-Operator.

Bei der folgenden Anfrage wird die Klassenextension `Lieferanten` nach dem Ort »Dortmund« gefiltert. Beachten Sie, daß das Attribut `Adresse` von der Klasse `Handelsbetriebe` vererbt wird. Der Ort ist eine Komponente des strukturierten Typs `AdresseT` – der als Klasse realisiert ist – und wird über einen Pfadausdruck erreicht. Das Ergebnis ist vom Typ `set<struct>`.

```
select distinct struct (Name: l.Ansprechpartner,
                       Tel: l.Telefon)
from Lieferanten l
  where l.Adresse.Ort = "Dortmund"
```

Objektorientierte Datenbanksysteme verwenden keine Schlüssel-Fremdschlüssel-Beziehungen, sondern definieren stattdessen Assoziationen (*relationships*), um von einem Objekt über seine Objektverbindung zu einem damit assoziierten Objekt zu navigieren. Von der Notation her ist kein Unterschied, ob über die Objektverbindung navigiert wird oder ein Zugriff innerhalb einer Struktur erfolgt.

Beispiel Navigieren in OQL

Es sind alle Artikel zu ermitteln, die von den Dortmunder Lieferanten geliefert werden. Dazu gehen wir von einem Objekt der Klasse `Artikel` aus, navigieren zu seinem (genau einem) Lieferanten und greifen in dem strukturierten Attribut `Adresse` auf den Ort zu (Abb. 8.6-1):

Beispiel

```
select a
from ArtikelListe a, a.wirdGeliefertVon l
where l.Adresse.Ort = "Dortmund"
```

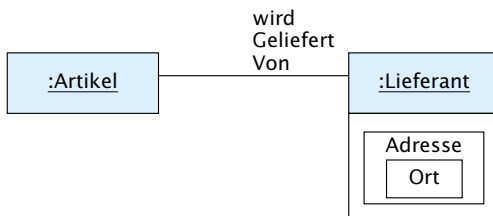


Abb. 8.6-1:  
Navigieren von  
Artikel zu Lieferant

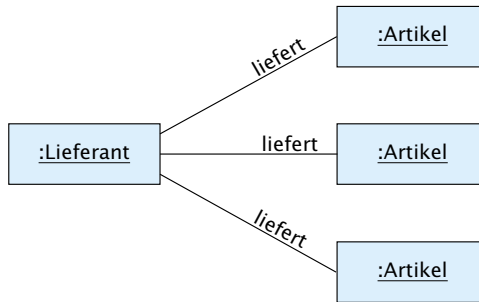
Auch die folgende Anfrage navigiert über diese Assoziationsrichtung und gibt eine Liste aller Artikelbezeichnungen und deren Lieferfirmen aus:

```
select struct ( Bezeichnung: a.Bezeichnung,
               Lieferfirma: a.wirdGeliefertVon.Firma)
from ArtikelListe a
```

Nun betrachten wir die inverse Richtung der Assoziation. Wir wollen alle Lieferanten ermitteln, die einen Artikel mit der Bezeichnung »Diskette« liefern. Dazu gehen wir von einem Objekt der Klasse `Lieferant` aus, navigieren zu jedem seiner `Artikel` und lesen dessen Attribut `Bezeichnung` (Abb. 8.6-2):

## LE 15 8 Datenbanken

Abb. 8.6-2:  
Navigieren vom  
Lieferanten zu  
seinen Artikeln



```
select l
from Lieferanten l, l.liefert a
where a.Bezeichnung = "Diskette"
```

Wir formulieren nun eine Anfrage, die eine Liste aller Firmennamen und der gelieferten Artikel im Nummernkreis von 1000 bis 1999 erstellt. Um die gelieferten Artikel zu ermitteln, traversieren wir die Assoziation `liefert` und selektieren in dieser Teilmenge alle gewünschten Artikel. Das Ergebnis ist vom Typ

```
set <struct(Name:string, gelieferteArtikel: bag<Artikel >)>:
select distinct struct (Lieferfirma: l.Firma, gelieferteArtikel:
  (select a
   from l.liefert a
   where a.Nummer >= 1000 and a.Nummer <= 1999))
from Lieferanten l
```

Operation in OQL OQL erlaubt den Aufruf einer Operation (mit oder ohne Parameter) überall dort, wo der Ergebnistyp der Operation in der Anfrage erlaubt ist, d.h. wo ein Attribut des gleichen Typs stehen kann. Der Ergebniswert einer Operation muß nicht von einem einfachen Typ sein. Ein Objekt oder eine Kollektion von Objekten ist ebenfalls zulässig. Auch das **späte Binden** von Operationen wird von OQL unterstützt. Besitzt eine Operation keine Parameter, dann ist anhand der Notation nicht ersichtlich, ob es sich um ein Attribut, einen Operationsaufruf oder um die Navigation einer Assoziation handelt. Parameter einer Operation werden in Klammern angegeben.

Beispiel Die Operation `teuersterArtikelpreis()` liest für einen Lieferanten die Preise der von ihm gelieferten Artikel und ermittelt den Maximalwert. Die folgende Anfrage gibt für alle Lieferanten in Dortmund den Preis ihres teuersten Artikels aus:

```
select l.teuersterArtikelpreis()
from Lieferanten l
where l.Adresse.Ort = "Dortmund"
```

Probleme können entstehen, wenn in *select*-Anweisungen auf undefinierte Daten zugegriffen wird. OQL ermöglicht daher die Arbeit mit Nullwerten. Nullwerte

Die folgende Anfrage führt zu einem Laufzeitfehler, wenn für einen Lieferanten kein Ort eingetragen ist. Beispiel

```
select l . Adresse . Ort
from Lieferanten l
```

Die korrekte Anfrage bei möglichen Nullwerten ist:

```
select l . Adresse . Ort
from Lieferanten l
where is_defined (l . Adresse . Ort)
```

Umgekehrt gibt die folgende Anfrage alle Lieferanten aus, für die noch kein Ort eingetragen wurde.

```
select l
from Lieferanten l
where is_undefined (l . Adresse . Ort);
```

Ein neues Objekt kann mit Hilfe des Objektkonstruktors erzeugt und über die Parameterliste mit bestimmten Attributwerten initialisiert werden: Konstruktor

Artikel (Nummer: 4711, Bezeichnung: "SuperMan") Beispiel

Mit dem *define*-Operator werden vordefinierte Anfragen (*named queries*) – ähnlich den Sichten bei relationalen Datenbanken – erstellt. Diese vordefinierten Anfragen können dann in der *from*-Klausel von *select*-Anweisungen benutzt werden. *define*

```
define DortmundLieferanten() as
select l
from Lieferanten l
where l . Adresse . Ort = "Dortmund" Beispiel
```

## 8.7 Anbindung an C++

Das Grundprinzip der Sprachanbindung bei objektorientierten Datenbanksystemen ist, daß sich der Programmierer in einer einzigen Sprache ausdrücken kann und nicht mit zwei Sprachen und deren Schnittstellen umgehen muß.

Die Sprachanbindung für C++ wird definiert durch

- C++ ODL (*Object Definition Language*),
- C++ OML (*Object Manipulation Language*) und
- C++ OQL (*Object Query Language*).

C++ ODL ist eine Klassenbibliothek, die es ermöglicht, die Konzepte des ODMG-Objektmodells zu implementieren. C++ ODL

Klassen werden in C++ persistent, indem sie von der Klasse `d_Object` abgeleitet werden. Eine Extension des Typs `t` wird auf `d_Extent<t>` abgebildet. Das Schlüsselkonzept wird in C++ ODL

## LE 15 8 Datenbanken

nicht unterstützt. Zusätzlich zu den in C++ vorhandenen Standardtypen bietet C++ ODL eigene Typen, z.B. `d_Long`, `d_ULong`, `d_Float`, die im Gegensatz zu den Typen von C++ auf allen Plattformen denselben Wertebereich besitzen. Für die Speicherung von *Strings* in der Datenbank sollte der Typ `d_String` verwendet werden, während der »normale« *String* bei allen transienten Objekten angewandt wird. Attribute und Operationen werden – unter Benutzung der zusätzlichen Typen – wie in C++ deklariert.

Assoziationen in C++ ODL

Verbindungen zwischen Objekten werden in C++ ODL als Objektreferenzen spezifiziert. Assoziationen können wie folgt deklariert werden:

- `d_Rel_Ref <T, const char*>`,
- `d_Rel_Set <T, const char*>` und
- `d_Rel_List <T, const char*>`.

»T« ist eine persistente Klasse, der zweite Parameter spezifiziert die inverse Richtung der Assoziation.

**Beispiel** Das Klassendiagramm der Abb. 8.7-1 wird in C++ ODL wie folgt spezifiziert:

```
extern const char _wirdGeliefertVon[], _liefert[];
class Lieferant: public d_Object
{public:
    d_String Firma;
    d_String Ansprechpartner;
    d_Rel_Set <Artikel, _wirdGeliefertVon> liefert;
    void erfassen();
    d_Float teuersterArtikelpreis();
};
class Artikel: public d_Object
{public:
    d_ULong Nummer;
    d_String Bezeichnung;
    d_Float Preis;
    d_Rel_Ref <Lieferant, _liefert> wirdGeliefertVon;
    void erfassen();
    void wechsellieferant (Lieferant &neuerLieferant);
};
const char _wirdGeliefertVon[] = "wirdGeliefertVon";
const char _liefert[] = "liefert";
```

Abb.8.7-1:  
Klassendiagramm  
Lieferant und  
Artikel



Jede Assoziation muß bidirektional spezifiziert werden. Das bedeutet, daß in jeder Klasse (z.B. Lieferant) der Name der assoziierten Klasse (z.B. Artikel) und die inverse Richtung der Assoziation (z.B.

wird (geliefert von) angegeben werden muß. Dadurch muß der Programmierer den Aufbau einer Verbindung nur in einer Richtung spezifizieren, während die andere Richtung vom Datenbanksystem automatisch erstellt wird.

Ein grundlegendes Entwurfskonzept von C++ OML ist, daß sich die Syntax zur Manipulation der Objekte so wenig wie möglich von der Manipulation transienter Objekte unterscheidet. Zur Erzeugung neuer Objekte wird `new()` verwendet und über den Parameter die Lebensdauer (transient oder permanent) bestimmt. Objekte können mittels `delete_object()` gelöscht werden. Handelt es sich um ein persistentes Objekt, dann wird es nicht nur aus dem Speicher, sondern auch aus der Datenbank entfernt.

Auf Objekte wird über ihre Objektreferenz zugegriffen. In C++ OML sind Objektreferenzen Exemplare der generischen Klasse `d_Ref<T>`. Ist ein referenziertes Objekt nicht im Speicher, dann wird es automatisch von der Datenbank in den Speicher geladen. Objektreferenzen sind in vielen Aspekten den Zeigern in C++ vergleichbar, besitzen jedoch zusätzliche Mechanismen, um die referentielle Integrität der persistenten Objekte zu gewährleisten. Man spricht daher von intelligenten Zeigern (*smart pointers*).

Es wird ein persistentes Objekt der Klasse `Lieferant` angelegt.

```
d_Ref<Lieferant> l_lieferant;
l_lieferant = new (&database, "Lieferant") Lieferant;
```

Beispiel

Objektnamen ermöglichen dem Programmierer einen bequemen Zugriff auf Objekte. In einer Datenbank müssen alle Objektnamen eindeutig sein. Die Operation `set_object_name()` weist dem jeweiligen Objekt einen String zu.

Objektnamen in C++ OML

```
database->set_object_name (l_lieferant, "SuperBillig");
```

Beispiel

Auf Attribute wird in C++ OML wie in C++ zugegriffen. In C++ ist es erlaubt, daß ein Attribut eines Objekts selbst wieder ein Objekt ist, nicht nur die OID eines anderen Objekts, wie beim ODMG-Standard. Diese *embedded objects* besitzen nach dem ODMG-Standard keine eigene Objektidentität und werden wie »einfache« Attribute behandelt. Auf diese Objekte darf daher nicht mittels `d_Ref` zugegriffen werden.

Attribut in C++ OML

Von dem persistenten Objekt der Klasse `Artikel` wird dessen Bezeichnung ausgegeben.

```
d_Ref <Artikel > artikel ;
...
cout << artikel ->Bezeichnung;
```

Beispiel

Operationen werden in C++ OML wie in C++ verwendet. Wird ein persistentes Objekt geändert, dann muß diese Änderung der Datenbank sichtbar gemacht werden. Daher müssen alle modifizierenden

Operation in C++ OML

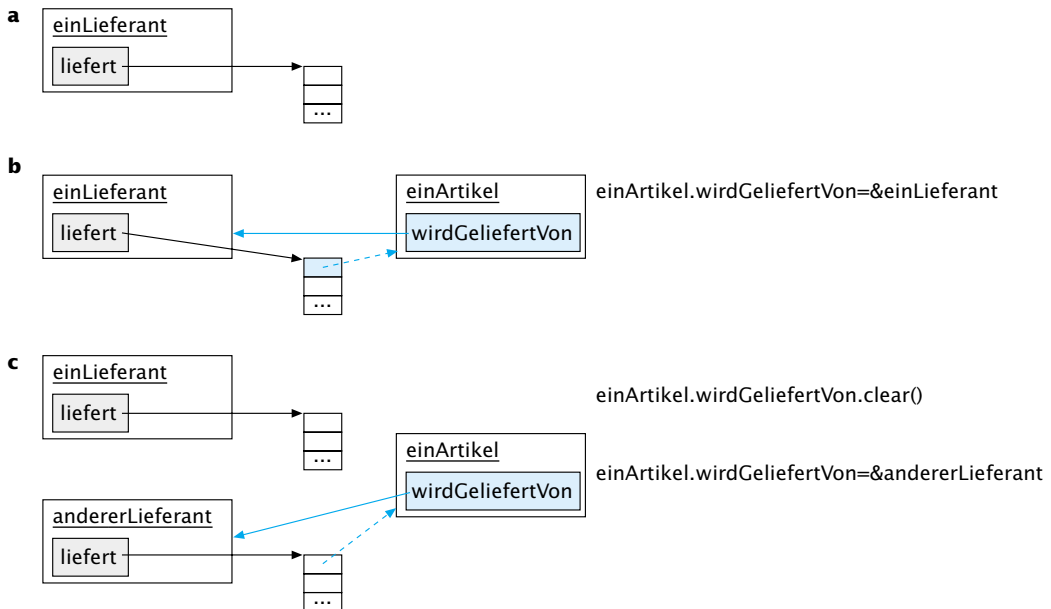
## LE 15 8 Datenbanken

Operationen den Operationsaufruf `obj_ref->mark_modifiziert()` enthalten.

Assoziation in C++ OML Wir betrachten das Aufbauen und Trennen von Objektverbindungen in C++ OML anhand des Beispiels von Lieferanten und Artikeln, dessen Klassendiagramm in Abb. 8.7-1 dargestellt ist.

**Beispiel** Zunächst gehen wir darauf ein, wie eine Objektverbindung von einem Artikel zu seinen Lieferanten aufgebaut wird (*one*-Richtung der Assoziation). In Abb. 8.7-2 wird zunächst ein neues Objekt der Klasse `Lieferant` erzeugt (Teil a). Da es zu einem Lieferanten viele Artikel geben kann, verbirgt sich hinter `liefert` eine Liste von Referenzen auf Artikelobjekte, die zunächst leer ist. Durch die Anweisung (Teil b) `einArtikel.wirdGeliefertVon = &einLieferant` wird die durchgezogene blaue Linie vom Programmierer aufgebaut und die gestrichelte blaue Linie – d.h. die inverse Richtung der Assoziation – automatisch durch das Datenbanksystem erstellt. Ändert sich der Lieferant des Artikels (Teil c), dann wird zuerst die Verbindung zum aktuellen Lieferanten durch die Anweisung `einArtikel.wirdGeliefertVon.clear()` gelöst, wobei die inverse Richtung automatisch vom Datenbanksystem entfernt wird. Anschließend wird analog zu oben eine Verbindung zum Objekt andererLieferant aufgebaut.

Abb. 8.7-2:  
Aufbau von  
Verbindungen vom  
Artikel-Objekt zum  
Lieferant Objekt



Nun betrachten wir, wie Objektverbindungen vom Lieferanten zu seinen Artikelobjekten aufgebaut werden (Abb.8.7-3). Da es sich in dieser Richtung um eine *many*-Assoziation handelt, müssen die



Operationen `insert_element()` und `remove_element()` verwendet werden. Die Ausgangsbasis ist wieder ein vorhandenes Objekt der Klasse `Lieferant` (Teil a). In Teil b der Abb. 8.7-3 wird dargestellt, wie die bidirektionale Verbindung zwischen `einLieferant` und `einArtikel` aufgebaut wird. Der Programmierer erstellt mit der Anweisung `einLieferant.liefert.insert_element(&einArtikel)` die Verbindung vom Lieferanten zum Artikel (durchgezogene blaue Linie), während das Datenbanksystem die inverse Richtung (gestrichelte blaue Linie) automatisch aufbaut. Es wird also das gleiche Ergebnis erreicht, wie im Teil b der Abb. 8.7-2. Analoges gilt für den Teil c der beiden Abbildungen.

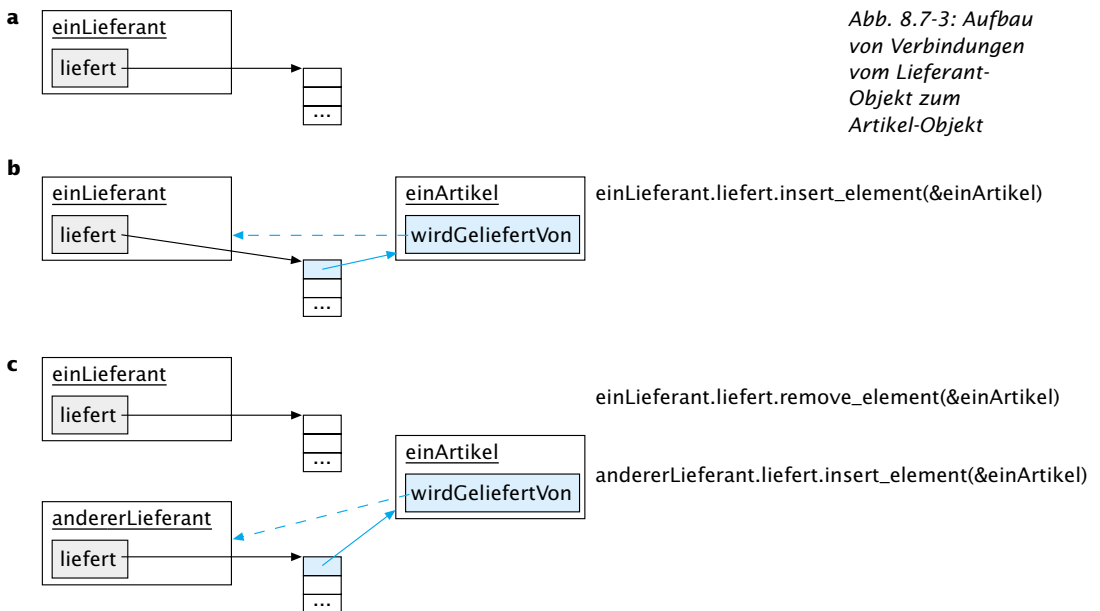


Abb. 8.7-3: Aufbau von Verbindungen vom Lieferant-Objekt zum Artikel-Objekt

Wie obige Ausführungen zeigen, wird die **referentielle Integrität** der Objektreferenzen automatisch durch das Datenbanksystem sichergestellt. Wenn eine Verbindung zwischen zwei Objekten existiert und eines der Objekte gelöscht wird, dann entfernt das Datenbanksystem automatisch die inverse Verbindung.

```
d_Ref <Lieferant> Lieferant;
d_Ref <Artikel> artikel;
//Aufbauen einer Verbindung aus one-Richtung und Löschen dieser
//Verbindung
artikel->wirdGeliefertvon = Lieferant;
artikel->wirdGeliefertvon.clear();
//Aufbauen der Verbindung aus der many-Richtung und Löschen dieser
//Verbindung
Lieferant->liefert.insert_element(artikel);
Lieferant->liefert.remove_element(artikel);
```

Beispiel

## LE 15 8 Datenbanken

Lesen einer Kollektion Mit der generischen Klasse `d_Iterator` kann in C++ OML auf die einzelnen Elemente einer Kollektion zugegriffen werden.

**Beispiel** Die folgenden Anweisungen ermittelt alle Objekte in der Klassenextension der Klasse `Lieferant`.

```
d_Iterator <d_Ref<Lieferant>> iterator=
    Lieferanten.create_iterator();
d_Ref<Lieferant> l;
while (iterator.next(l))
{ l = iterator.get_element();
  ...
};
```

**Transaktion** Transaktionen sind ein wichtiges Konzept von Datenbanksystemen. Ist eine Transaktion erfolgreich (*commits*), dann werden alle im Rahmen der Transaktion durchgeführten Änderungen permanent in die Datenbank eingetragen und sind für alle anderen Benutzer sichtbar. Wenn eine Transaktion fehlschlägt (*aborts*), dann darf diese Transaktion keinerlei Veränderungen in der Datenbank zur Folge haben. Alle bereits im Rahmen der Transaktion durchgeführten Verarbeitungen müssen rückgängig gemacht werden. Transaktionen werden in C++ OML über Objekte der Klasse `d_Transaction` gesteuert.

**Beispiel** `d_Transaction transaction;`  
`transaction.begin();`  
`... //Operationen auf persistenten Objekten`  
`transaction.commit();`

**Beispiel C++ OML** Die folgenden Anweisungen suchen einen Lieferanten nach Namen und geben die von ihm gelieferten Artikel aus.

```
d_Database database;
database.open("ArtikelLieferantenDB");
d_Transaction transaction;
transaction.begin();

d_Ref<Lieferant> lieferant;
lieferant = database->lookup_object("SuperBillig");

d_Iterator<d_Ref<Artikel>> iterator;
d_Ref<Artikel> artikel;
iterator = lieferant->Liefert.create_iterator();
cout << "Lieferant " << lieferant->Firma
    << " Liefert folgende Artikel " << endl;
while (iterator.next(artikel))
{ cout << artikel->Nummer << " " << artikel->Bezeichnung
    << " " << artikel->Preis << endl;
}
transaction.commit();
database.close();
```

C++ OQL bildet die Semantik von OQL in C++ ab. Die Klasse `d_Collection` besitzt eine *query*-Operation mit folgender Signatur: C++ OQL

```
int query (d_Collection<T> &result, const char* predicate) const;
```

Diese Operation filtert eine Kollektion von Objekten, auf die das Prädikat angewendet wird, und stellt das Ergebnis im ersten Parameter zur Verfügung. Das Prädikat wird als *String* mit der Syntax der *where*-Klausel von OQL übergeben.

Eine weitere Möglichkeit besteht darin, eine Anfrage als Objekt der Klasse `d_OQL_Query` zu erzeugen. Danach kann die Anfrage beliebig oft ausgeführt werden. Dazu wird die freie generische Funktion

```
template<class T> void d_oql_execute(d_OQL_Query &query,
                                T &result)
```

benutzt. Der erste Parameter der Funktion ist eine Referenz auf das zuvor erzeugte Objekt von `d_OQL_Query`. Im zweiten Parameter wird das Ergebnis der Anfrage zurückgegeben.

## 8.8 Objekt-relationale Datenbanksysteme

**Objekt-relationale Datenbanksysteme** verfolgen das Ziel, die besten Ideen aus der relationalen und der objektorientierten Welt zu verbinden. Das grundlegende Konzept bleibt weiterhin die Tabelle bzw. die Relation. Die Sprache SQL3, die eine Weiterentwicklung von SQL2 ist, wird von ANSI und ISO genormt. Wichtige objekt-orientierte Erweiterungen sind /Heuer 97/:

- Abstrakte Datentypen (ADTs),
- Objektidentitäten,
- Definition von Operationen für ADTs,
- Überschreiben des Operationsnamens mit der Möglichkeit zur dynamischen Auswahl der Operationsimplementierung,
- ADT-Hierarchien,
- Tabellenhierarchien.

Ein Abstrakter Datentyp (ADT) kann für Objekte (Objekt-ADT – mit Objektidentität) und für Werte (Wert-ADT – ohne Objektidentität) definiert werden. Mit der Definition eines Objekt-ADT wird gleichzeitig ein Surrogat-Attribut für diesen ADT erzeugt, das die Eigenschaften der Objektidentität erfüllt. Die Verkapselung und das Geheimnisprinzip können durch Angabe der Sichtbarkeiten *public*, *protected* und *private* gesteuert werden. ADT in SQL3

Für jedes Attribut eines ADT werden automatisch zwei Operationen generiert:

- Eine *Observer*-Operation ermöglicht den lesenden Zugriff.
- Eine *Mutator*-Operation ermöglicht Änderungen des Attributwertes.

Außerdem ist ein Konstruktor mit dem Namen des ADT und ein Destruktor mit dem Namen *destroy* vordefiniert.

## LE 15 8 Datenbanken

Es werden Funktionen und Prozeduren unterschieden. Prozeduren verändern Objekte oder Werte und haben keinen Ergebniswert. Ihre Parameter können mit *in*, *out*, und *inout* unterschieden werden. Funktionen realisieren im allgemeinen Anfragen und besitzen einen Ergebniswert. Sie dürfen nur *in*-Parameter besitzen.

Beispiel 

```
create type Artikel
(
  public
    Nummer integer,
    Bezeichnung char (30),
    Verkaufspreis real,
    Einkaufspreis real,
    ...
  public function
    ermittlePreisdifferenzVerkaufEinkauf(a Artikel)
    returns real
  begin
    ...
  end
)
```

ADT-Hierarchie Ein spezialisierter ADT wird mit der *under*-Klausel definiert. Hierbei kann der spezialisierte ADT auch Operationen des allgemeineren ADT überschreiben. Die dynamische Auswahl einer Implementierung entspricht dem dynamischen Binden in einer objekt-orientierten Programmiersprache.

Beispiel 

```
create type Lieferant under Handelsbetriebe
(...)
```

Tabellenhierarchie Die Tabellenhierarchie bezieht sich auf die Extensionen (Tabellen, Relationen) der beteiligten ADTs. Die Untertabelle enthält alle Attribute der Obertabelle. Jede *insert*-Operation in der Untertabelle wirkt auch auf die Obertabelle, jede *delete*-Operation in der Obertabelle auch auf die Untertabelle.

Beispiel 

```
Lieferanten und Handelsbetriebe bezeichnen die Extensionen der
ADTs Lieferant und Handelsbetriebe.
create table Lieferanten of Lieferant
under Handelsbetriebe of Handelsbetriebe
(...)
```

**Anfragesprache** →OQL

**Klassenextension** (*extent*) Unter der Klassenextension ist die Menge aller Objekte einer Klasse zu verstehen. Die Klassenextension wird im Entwurf durch *Container*-Klassen realisiert, während in der Analyse jede Klasse die Eigenschaft der Objektverwaltung besitzt. Bei →objektorientierten Datenbanksystemen kann der Programmierer entscheiden, ob eine Klassenextension

erzeugt werden soll. Falls der *extent* angelegt wird, wird ein neu erzeugtes Objekt automatisch eingefügt, beim Löschen wieder entfernt. Das Konzept der Klassenextension ermöglicht die Durchführung von Operationen (z.B. Selektionen) auf der Menge aller Objekte einer Klasse.

**Literal** (*literal*) Literale sind Daten, die im Gegensatz zu Objekten in einer objektorientierten Datenbank keine



→Objektidentität besitzen. Sie können daher nur als Teil eines →Objekts in einer Datenbank gespeichert werden.

**Objekt (*object*)** Objekte besitzen eine →Objektidentität und können – im Gegensatz zu →Literalen – separat in einer objektorientierten Datenbank gespeichert werden.

**Objektdefinitionssprache** →ODL

**Objektidentität (*object identity*)**

**1** Jedes Objekt besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Selbst wenn zwei →Objekte zufällig dieselben Attributwerte besitzen, haben sie eine unterschiedliche Identität.

**2** In objektorientierten Datenbanksystemen werden Objektidentitäten automatisch vom System generiert und verwaltet. Sie besitzen keine (verwendbare) Semantik und sind dem Programmierer nicht bekannt. Objektidentitäten können in objektorientierten Datenbanksystemen beispielsweise als Surrogate realisiert werden.

**Objektorientiertes Datenbanksystem (*object database systems*)** Ein objektorientiertes Datenbanksystem (ODBS) ist ein Datenbanksystem, dem ein objektorientiertes Datenmodell zugrunde liegt. Es integriert die Eigenschaften einer Datenbank mit den Möglichkeiten von objektorientierten Programmiersprachen.

**Objekt-relationales Datenbanksystem (*object-relational database system*)** Objekt-relationale Datenbanksysteme verfolgen das Ziel, die besten Ideen aus der relationalen und der objektorientierten Welt zu verbinden. Das grundlegende Konzept bleibt weiterhin die Tabelle. Es wird um objektorientierte Konzepte wie Abstrakte Datentypen, →Objektidentität, Operationen und Vererbung erweitert.

**ODL (*Object Definition Language*)** Die Objektdefinitionssprache ODL ist eine Sprache, die ausschließlich zur Spezifikation von Klassen und Schnitt-

stellen dient. Diese Spezifikation erfolgt dadurch unabhängig von der Implementierung in einer Programmiersprache. ODL unterstützt alle Konzepte des →ODMG-Objektmodells.

**ODMG (*Object Database Management Group*)** Die ODMG ist eine Gruppe von Herstellern und Anwendern →objektorientierter Datenbanksysteme. 1993 wurde von dieser Gruppe die erste Version eines Standards für objektorientierte Datenbanksysteme vorgeschlagen: ODMG-93 genannt.

**ODMG-Standard (*object database standard ODMG*)** Der ODMG-Standard 2.0 besteht aus dem →Objektmodell, der →ODL (*Object Definition Language*), dem ASCII-Austauschformat OIF (*Object Interchange Format*), der deklarativen Sprache →OQL (*Object Query Language*) und Sprachanbindungen zu C++, Smalltalk und Java. Es gibt zwei Stufen der Einhaltung des Standards: ODMG-konform (*ODMG compliant*) und ODMG-zertifiziert (*ODMG certified*).

**ODMG-Objektmodell (*object model*)** Das ODMG-Objektmodell spezifiziert die Konzepte, die von einem →objektorientierten Datenbanksystem unterstützt werden. Es bildet die Grundlage für den →ODMG-Standard.

**OID** →Objektidentität

**OQL (*Object Query Language*)** Die Anfragesprache OQL dient zur Formulierung von Selektionen in einer objektorientierten Datenbank. OQL baut auf dem *select-from-where*-Block von SQL2 auf. OQL kann sowohl als eigenständige – Datenbanksprache als auch eingebettet in verschiedene Programmiersprachen benutzt werden. OQL wurde im →ODMG-Standard definiert.

**Schnittstelle (*interface*)** Der →ODMG-Standard verwendet außer der Klasse das Konzept der Schnittstelle, die nur das Verhalten spezifiziert. Von einer Schnittstelle können – im Gegensatz zur Klasse – keine Objekte erzeugt werden.

## LE 15 Zusammenhänge/Aufgaben

**Objektorientierte Datenbanksysteme** kombinieren die Eigenschaften einer Datenbank mit den Möglichkeiten von objektorientierten Programmiersprachen. Wesentliche Eigenschaften sind die automatische Vergabe einer **Objektidentität** und die Fähigkeit, **Objekte** beliebigen Typs zu speichern. Der **ODMG-Standard** definiert u.a. die **Objektdefinitionssprache ODL**, die **Anfragesprache OQL** und eine Anbindung an C++ (C++ ODL, C++ OML, C++ OQL), Java und Smalltalk. **Objekt-relationale Datenbanksysteme** sollen die besten Ideen aus der relationalen und der objektorientierten Welt verbinden.



Aufgabe 1 **Lernziel:** *Grundbegriffe des ODMG-Objektmodells erläutern können.* 5–10 Minuten

Erläutern Sie folgende Begriffe:

- a** Literal und Objekt.
- b** Klasse (*class*) und Schnittstelle (*interface*).
- c** *extends*- und *subtyping*-Vererbung.

Aufgabe 2 **Lernziele:** *Assoziationen in ODL und Anfragen in OQL spezifizieren können.* 10–15 Minuten

Erweitern Sie das Beispiel der Abb. 8.5-2 um eine m:m-Assoziation zwischen Kunde und Artikel, die folgende semantische Bedeutung besitzt: Ein Kunde kauft 1 bis viele Artikel. Ein Artikel kann von 0 bis vielen Kunden gekauft werden. Dabei soll nicht festgehalten werden, ob ein bestimmter Artikel mehrmals vom gleichen Kunden gekauft wurde.

- a** Ergänzen Sie die Spezifikation in ODL.
- b** Formulieren Sie eine OQL-Anfrage, die den Namen und den Umsatz aller Kunden ausgibt, deren Umsatz größer als 10.000 ist.
- c** Ermitteln Sie mittels OQL alle Kunden, die einen Artikel mit der Nummer 4711 gekauft haben.

**3 Lernziel:** Klassendiagramme in ODL spezifizieren können.

Erstellen Sie für das Klassendiagramm der Abb. LE15-A3 eine Spezifikation mittels ODL.

Aufgabe  
20–25 Minuten

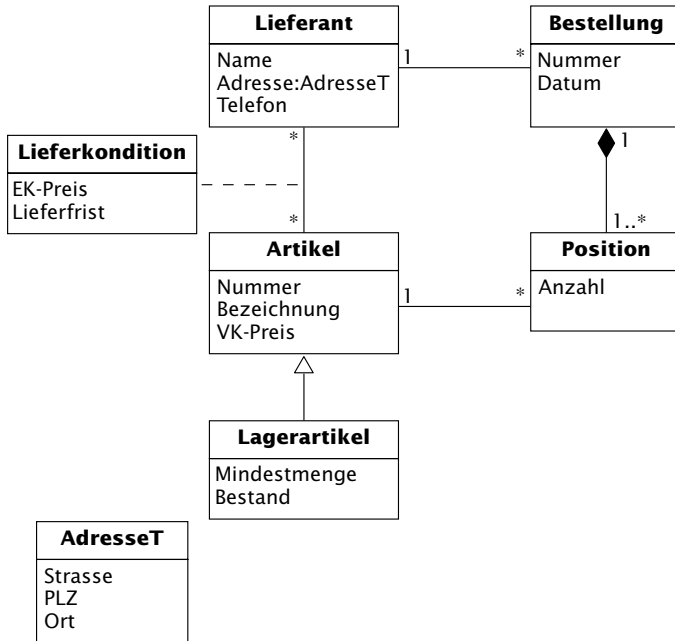


Abb. LE15-A3:  
Klassendiagramm  
Bestellwesen

**4 Lernziel:** Anfragen in OQL spezifizieren können.

Formulieren Sie für das in der Aufgabe 3 erstellte Schema folgende Anfragen in OQL:

Aufgabe  
10 Minuten

- a** Erstellen einer Liste aller Lagerartikel, bei denen der Mindestbestand unterschritten ist. Die Liste soll enthalten: Nummer, Bezeichnung, Bestand, Mindestmenge.
- b** Für jeden Dortmunder Lieferanten ist eine Liste der ihm erteilten Bestellungen zu erstellen. Die Liste soll folgende Angaben enthalten: Lieferantename, Bestelldatum.

## 9 Verteilte objektorientierte Anwendungen



- Erklären können, wie die Kommunikation zwischen entfernten Objekten stattfindet.
- Erklären können, was der ORB ist und aus welchen Komponenten er besteht.
- Erklären können, aus welchen Komponenten OMA besteht.
- Klassen mittels IDL spezifizieren können.

verstehen

anwenden



Sie sollten mit den objektorientierten Konzepten, wie sie in den Kapiteln 2 und 6 beschrieben werden, vertraut sein.

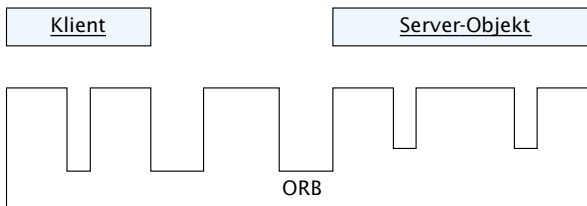
- i 9.1 Kommunikation von verteilten Objekten 352
- 9.2 ORB-Architektur 353
- 9.3 OMA 356
- 9.4 IDL 357
- 9.5 Entwicklung eines verteilten Systems 361



## 9.1 Kommunikation von verteilten Objekten

Viele Anwendungen sind heute auf Client-Server-Architekturen verteilt. Eine Aufgabe des Entwurfs ist es daher, für verteilte objektorientierte Anwendungen eine geeignete Architektur für heterogene und vernetzte Systeme zu entwickeln (siehe /Redlich 96/, /Ben-Natan 95/, /Balzert 96/). Objekte sollen miteinander kommunizieren können und zwar unabhängig von der verwendeten Programmiersprache und ebenfalls unabhängig von der Systemplattform, auf der sie sich befinden. Beispielsweise kann sich auf einem MVS-Rechner ein in Cobol programmiertes Objekt befinden, das von einem Java-Programm benutzt wird, das auf einem Windows-Netzwerk-Client läuft. Allgemein ausgedrückt bietet ein Server-Objekt eine Dienstleistung an, die vom Klienten benötigt wird. Klient und Server-Objekt müssen nicht wissen, wo sich jeweils der andere Partner befindet. Die Kommunikation zwischen beiden wird vom **ORB** (*Object Request Broker*) durchgeführt (Abb. 9.1-1). Der ORB ist vergleichbar mit einer Telefonvermittlung, die das Anrufen anderer Teilnehmer (Server-Objekte) und das Entgegennehmen von Anrufen (Operationsaufrufen) realisiert. Seine Hauptaufgabe ist es, Operationsaufrufe vom Klienten an das entfernte Server-Objekt zu übermitteln und die Ergebnisse zurückzugeben.

Abb. 9.1-1:  
Kommunikation  
mittels ORB



Operationsaufruf  
eines entfernten  
Objekts

Der **Klient** (*client*) ist eine Softwareeinheit, die eine Operation eines Objekts auf einem entfernten Server benutzen möchte. Beim Klienten kann es sich um ein einfaches Programm oder um ein Objekt handeln. Mit einem *request* fordert der Klient das Objekt zur Ausführung einer Operation auf. Dabei soll der Klient auf die vom entfernten Objekt bereitgestellten Operationen zugreifen können, als ob es sich um ein lokales Objekt handeln würde. Der ORB muß nun das Server-Objekt ermitteln und ihm den *request* übermitteln. Ein *request* ist ein Ereignis, mit dem folgende Informationen verbunden sind: der Name der angeforderten Operation, das Zielobjekt, die aktuellen Parameter und ein optionaler Kontext. Auf der Server-Seite ist das Verhalten des Objekts beschrieben, d.h. festgelegt, welche Operationen es ausführen kann. Aus objektorientierter Sichtweise gehört jedes Server-Objekt zu einer Server-Klasse. Aus Sicht des Betriebssystems werden mehrere Server-Objekte zu einem

Programm – dem Server – zusammengefaßt. Der Klient identifiziert das Server-Objekt über seine systemweit eindeutige **Objektreferenz** (*object reference*). Dabei ist zu beachten, daß sich das Server-Objekt außerhalb des Prozeßraums des Operationsaufrufs oder auf einem anderen Rechner befinden kann. Probleme entstehen insbesondere dann, wenn es sich um eine heterogene Netzwerkumgebung handelt. Der ORB ist dafür verantwortlich, daß das gewünschte Objekt gefunden wird und den Operationsaufruf empfangen kann.

Um die Unabhängigkeit von Programmiersprachen zu gewährleisten muß zwischen der Schnittstelle und der Implementierung der Server-Klassen unterschieden werden. Dazu werden die Schnittstellen der Server-Klassen in der Schnittstellensprache **IDL** (*Interface Definition Language*) beschrieben. Die Implementierung erfolgt in einer Programmiersprache, z.B. C++. Ein Server-Objekt besteht daher aus der IDL-Schnittstelle und aus der Objekt-Implementierung (*object implementation*). Die IDL-Schnittstelle befindet sich auf dem Klienten. Sie beschreibt, welche Operationen die Objekt-Implementierung zur Verfügung stellt und wie sie aufgerufen werden. Auf dem Server befindet sich genau genommen nur die Objekt-Implementierung. In der CORBA-Literatur wird daher vom Klienten und von der Objekt-Implementierung gesprochen. In den folgenden Kapiteln verwende ich ebenfalls diese Terminologie. Die **Objekt-Implementierung** (*object implementation*) definiert das Verhalten eines Objekts, in dem sie festlegt, welche Verarbeitung beim Aufruf einer Operation auszuführen ist. Außerdem legt sie fest, welche Daten benötigt werden, um den Zustand eines konkreten Objekts zu repräsentieren.

Server-Klasse  
= IDL-Schnittstelle  
+ Objekt-  
Implementierung

## 9.2 ORB-Architektur

Aus der IDL-Definition erstellt ein IDL-Compiler die in Abb. 9.2-1 dargestellten Informationen.

Die *IDL Stubs* bilden die lokalen Vertreter der entfernten Objekte. Für jede Server-Klasse ist eine IDL-Schnittstelle zu erstellen, aus der

*IDL Stub*

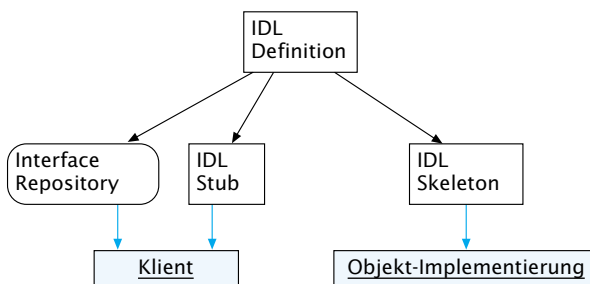


Abb. 9.2-1:  
Aus der IDL-  
Schnittstelle  
abgeleitete  
Informationen

## LE 16 9 Verteilte objektorientierte Anwendungen

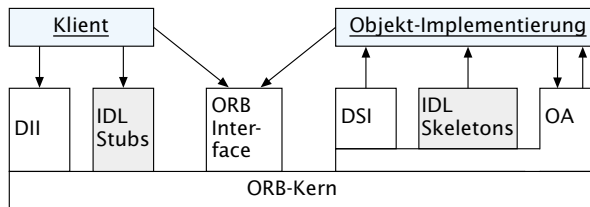
ein *IDL Stub* erzeugt wird. Er besitzt für jede IDL-Operation eine gleichlautende Operation in der jeweiligen Programmiersprache.

*IDL Skeleton* Parallel zum *IDL Stub* wird aus der IDL-Schnittstelle ein *IDL Skeleton* erzeugt. Das ist ein Rahmen in der gewünschten Programmiersprache, der vom Programmierer mit Code gefüllt werden muß. Ein *Skeleton* ist dafür zuständig, einen Operationsaufruf (*request*) mit der Implementierung dieser Operation zu verbinden.

*Interface Repository* Das *Interface Repository* ist eine systemweit zugängliche Datenbank, in der alle Informationen der IDL-Definitionen abgelegt werden.

*Implementation Repository* Das *Implementation Repository* verwaltet Informationen, die der ORB benötigt, um Objekt-Implementierungen zu lokalisieren und zu starten. Beim Aufruf einer Operation identifiziert der Klient das gewünschte Objekt durch seine Objektreferenz. Diese enthält jedoch keine Informationen über den physischen Aufenthaltsort der zugehörigen Objekt-Implementierung. Die Abbildung der Objektreferenz auf die physische Adresse erfolgt mit Hilfe des *Implementation Repository*. Diese Trennung ermöglicht es, alle Operationsaufrufe für ein Objekt an eine äquivalente Objekt-Implementierung auf einem anderen Rechner umzuleiten, wenn der entsprechende Eintrag im *Implementation Repository* geändert wird. Es ist ebenfalls denkbar, daß für ein Objekt auf mehreren Rechnern Objekt-Implementierungen bereitgestellt werden und der ORB in Abhängigkeit von der momentanen Auslastung den Operationsaufruf an den am wenigsten belasteten Rechner weitergibt. Aus der Sicht des Klienten gibt es dabei nur ein Objekt und eine Objektreferenz.

Abb. 9.2-2:  
Schnittstellen  
des ORB



Der ORB besitzt mehrere Schnittstellen zum Klienten und zur Objekt-Implementierung (Abb. 9.2-2), die zur Kommunikation zwischen Klient und Objekt-Implementierung beitragen. Ein Klient kann den *request* statisch oder dynamisch an ein Objekt übermitteln (Abb. 9.2-3).

statische  
Operationsaufrufe

Statische Operationsaufrufe werden immer dann verwendet, wenn die Definition des Server-Objekts vor der Übersetzung des Klienten zur Verfügung steht. Dann wird für die Server-Klasse eine IDL-Definition erstellt und daraus ein *IDL Stub* erzeugt. Der Operationsaufruf eines entfernten Objekts wird zunächst an die entsprechende Operation des *IDL Stubs* übergeben. Hier wird der

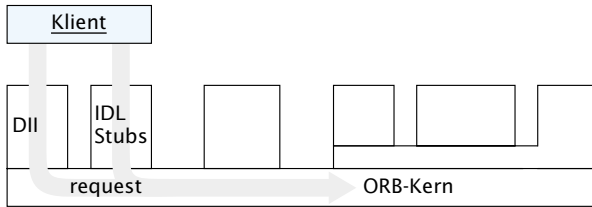


Abb. 9.2-3:  
Operationsaufruf  
(request) durch  
den Klienten

Operationsaufruf verpackt und über den ORB an das entfernte Objekt weitergeleitet.

Das *Dynamic Invocation Interface* (DII) ermöglicht es dem Klienten, einen Operationsaufruf zur Laufzeit zu generieren. Dazu liest er die Beschreibung der gewünschten Operationen aus dem *Interface Repository*. Anhand dieser Beschreibung baut der Klient die Parameterliste der Operation auf, erzeugt und verpackt den Operationsaufruf und sendet ihn über den ORB an das entfernte Objekt. Da die Benutzung von *IDL Stubs* für den Programmierer wesentlich einfacher ist, sollte die dynamische Form nur gewählt werden, wenn die Schnittstelle zur Übersetzungszeit des Klienten nicht bekannt ist.

*Dynamic  
Invocation  
Interface*

Auf der Empfängerseite ist nicht bekannt, ob ein Operationsaufruf statisch oder dynamisch erstellt wurde. In beiden Fällen empfängt der *Object Adapter* (OA) den Operationsaufruf und sorgt dafür, daß die entsprechende Operation der Objekt-Implementierung aufgerufen wird (Abb. 9.2-4). Für die Durchführung des Operationsaufrufs ist ein zur entsprechenden Schnittstelle gehörendes *Skeleton* notwendig. Das *IDL Skeleton* ist speziell für eine IDL-Schnittstelle zuständig und wird vom IDL-Compiler aus der entsprechenden IDL-Definition generiert. Ist für ein Objekt kein *IDL Skeleton* vorhanden, dann ist das *Dynamic Skeleton Interface* (DSI) zu verwenden, das beliebige *requests* entgegennehmen kann. *Dynamic Skeletons* können sowohl durch *IDL Stubs* als auch durch das *Dynamic Invocation Interface* aktiviert werden. Sie kommen hauptsächlich bei Operationsaufrufen zwischen ORBs verschiedener Hersteller zum Einsatz.

*Object Adapter*

Der *Object Adapter* trägt alle Schnittstellendefinitionen und alle erzeugten Objekte einschließlich ihrer Objektreferenzen in das *Implementation Repository* ein. Jeder CORBA-konforme ORB muß min-

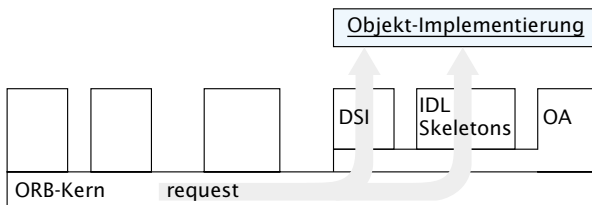


Abb. 9.2-4:  
Empfang eines  
Operationsaufrufs  
(request) durch  
die Objekt-  
Implementierung

destens den *Basic Object Adapter* (BOA) unterstützen. Er ist für die üblichen verteilten Anwendungen entworfen worden, in denen über mehrere Rechner hinweg verteilte Objekte miteinander kommunizieren müssen. Für die Zusammenarbeit mit Datenbanken ist der *Object-Oriented Database Adapter* (ODA) vorgesehen. Er übernimmt die automatische Rekonstruktion eines Objekts aus der Datenbank und das Zurückschreiben des neuen Objektzustandes nach der Ausführung der Operation. Der ODA ist im Gegensatz zum BOA auf die Arbeit mit großen Objektmengen abgestimmt.

Objektreferenz

Jedes Objekt wird durch eine systemweit eindeutige Objektreferenz identifiziert, die vom OA in Zusammenarbeit mit dem ORB vergeben wird. CORBA stellt sicher, daß jede Objektreferenz einen Server und innerhalb des Servers ein Exemplar einer Objekt-Implementierung identifiziert. Jeder ORB muß für jede von ihm unterstützte Programmiersprache eine Repräsentation für Objektreferenzen bereitstellen. Beispielsweise ist in C++ jeder *Stub* als Klasse realisiert, deren Exemplare jeweils eine konkrete Objektreferenz verkörpern. Auch dem dynamischen Operationsaufruf mittels DII muß eine Objektreferenz zur Identifikation des Objekts übergeben werden.

### 9.3 OMA

OMG Die **OMG** (*Object Management Group*) ist ein internationales Konsortium von Hardwareherstellern, Softwareentwicklern, Netzwerkbetreibern und kommerziellen Nutzern. Sie wurde 1989 von acht Firmen gegründet und umfaßte Anfang 1999 mehr als 800 Mitglieder. Die OMG verfolgt das Ziel, Standards und Spezifikationen für verteilte objektorientierte Anwendungen zu schaffen. Unter der angegebenen Internet-Adresse sind umfangreiche Informationen über die OMG und die von ihr erstellten Standards zu finden.

[www.omg.org](http://www.omg.org)



CORBA **CORBA** (*Common Object Request Broker Architecture*) ist der OMG-Standard, der spezifiziert, wie Objekte in einer verteilten, heterogenen Umgebung kommunizieren. Er beschreibt den Aufbau des ORB, seine Bestandteile sowie deren Verhalten und Schnittstellen.

OMG-Objektmodell

Allen OMG-Spezifikationen liegt das **OMG-Objektmodell** zugrunde. Es beschreibt einerseits die zugrundeliegenden objektorientierten Konzepte, die für Klienten wichtig sind: Objekte (*objects*), Operationsaufrufe (*requests*), Typen (*types*), Schnittstellen von Klassen (*interfaces*), Operationen (*operations*) und Attribute (*attributes*). Desweiteren beschreibt es alle Konzepte für die Ausführung von Operationen auf der Server-Seite.

OMA Die Grundlage aller Standardisierungsaktivitäten der OMG ist die **OMA** (*Object Management Architecture*). Sie unterteilt die Bestand-

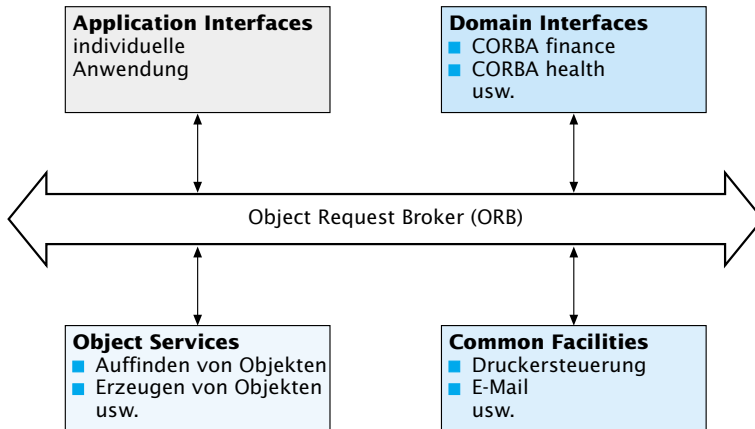


Abb. 9.3-1: OMA  
(Object Mangement  
Architecture)

teile einer verteilten Anwendung in mehrere Komponenten (9.3-1). Den Kern bildet der bereits beschriebene ORB, der als Kommunikationszentrale im Mittelpunkt der Architektur steht.

*Object services* sind elementare, betriebssystemähnliche Funktionen, die für die Entwicklung von verteilten Anwendungen benötigt werden. Beim Vergleich des ORBs mit einer Telefonvermittlung entsprechen die *object services* den elementaren Diensten wie z.B. Auskunft, Notrufe, Rufweiterleitung. Von der OMG werden u.a. folgende *object services* definiert: Auffinden von Objekten im Netz (*naming services*), Erzeugen und Löschen von Objekten (*lifecycle services*).

*object services*

*Common facilities* sind höhere Dienste, die einen wesentlichen Teil einer Anwendung ausmachen und für viele Anwendungsbereiche relevant sind. Sie bauen auf den *object services* auf und sind stärker spezialisiert. Zu diesen Diensten gehören beispielsweise grafische Benutzungsoberflächen, Dienste zur Druckersteuerung, Datenbanken und E-Mail.

*common facilities*

*Domain interfaces* bieten Lösungen für bestimmte Anwendungsbereiche. Das sind beispielsweise Softwarepakete für Finanzen (CORBA *finance*) oder für das Gesundheitswesen (CORBA *health*).

*domain interfaces*

*Application objects* bilden die eigentlichen Anwendungen, mit deren Hilfe der Benutzer bestimmte Aufgaben durchführt. Aus diesem Grund werden sie von der OMG nicht standardisiert.

*application objects*

## 9.4 IDL

**IDL** (*Interface Definition Language*) ist die Sprache zur Spezifikation der Schnittstellen aller Objekte, die von den Klienten verwendet werden. IDL ist eine rein beschreibende Sprache. Objekte können daher nicht in IDL implementiert oder aus einem IDL-Quelltext heraus aufgerufen werden. Daher muß die IDL-Definition in äquivalen-

te Konstrukte der jeweiligen Programmiersprache übersetzt werden (*language mapping*).

Kapitel 8.5

Die ODL des ODMG-Standards wurde auf der Grundlage von IDL entwickelt. ODL ohne *extends*, *keys* und *relationships* ist weitgehend mit IDL identisch. Trotz ihrer syntaktischen Ähnlichkeit unterscheiden sich beide Sprachen von ihrer Intension. Während die ODL zur Beschreibung von persistenten Daten dient, beschreibt die IDL die Schnittstellen, d.h. das Verhalten von entfernten Objekten.



IDL-Syntax

Die IDL-Syntax ist ähnlich der Syntax von C++. Eine IDL-Definition besteht aus Definitionen von IDL-Schnittstellen (*interfaces*), Typen (*types*), Modulen (*modules*), Konstanten und Ausnahmen (*exceptions*), die in beliebiger Reihenfolge und Anzahl auftreten können.

```
//IDL-Definition
const <Typ> <Name> = ...;
typedef ...;
enum <Name> { ... };
struct <Name> { ... };
exception <Name> { ... };
interface <Name> { ... };
module <Name> { ... };
```

*interface*

Die **Schnittstelle** (*interface*) spezifiziert die Signaturen von Operationen, die ein Klient aufrufen kann. Sie stellt die wichtigste Komponente einer IDL-Definition dar.

Operation

Die **Signatur** einer Operation wird folgendermaßen definiert:

```
[oneway] <Ergebnistyp> <Operationsname> (in Typ1 Parameter1,
                                         out Typ2 Parameter2,
                                         inout Typ3 Parameter3, ...)
    [raises (Ausnahme1, ..., AusnahmeM)]
    [context (Kontextname1, ..., KontextnameK)]
```

Ein *in*-Parameter wird vom Klienten zum Server übertragen. Er darf vom Server nicht geändert werden. *Out*-Parameter werden vom Server zum Klienten übertragen. Der *inout*-Parameter wird zunächst vom Klienten zum Server und anschließend vom Server zum Klienten übertragen. Besitzt die Operation einen Ergebnistyp, dann ist der Ergebniswert ein *out*-Parameter, andernfalls wird – wie in C++ – *void* angegeben.

Eine Operation kann optional eine *raises*-Klausel besitzen. Sie spezifiziert, welche Ausnahmen innerhalb eines Operationsaufrufes ausgelöst werden. Eine Ausnahme tritt auf, wenn die aufgerufene Operation nicht ordnungsgemäß zu Ende geführt werden kann. Beispielsweise kann eine Operation zu einem ungeeigneten Zeitpunkt oder mit sinnlosen Eingabeparametern aufgerufen werden. In diesem Fall muß das Server-Objekt die Ausführung der Operation vorzeitig beenden und den Klienten darüber informieren. Tritt eine Ausnahme auf, dann sind die Werte der *out*- und *inout*-Parameter

undefiniert. Ebenfalls optional ist die Angabe eines Kontextausdrucks (*context*). Er kann zusätzliche Informationen bereitstellen, z.B. den Aufenthaltsort des Klienten für die Rückmeldung der Ergebnisse. Der Standardfall ist eine synchrone Datenübertragung. Der Klient wartet nach dem Operationsaufruf bis die Ergebnisse oder eine Ausnahmemeldung zurückgegeben werden. Das optionale Schlüsselwort *oneway* kennzeichnet eine asynchrone Kommunikation. Der Klient wartet dann nicht auf das Ende der Operation, sondern arbeitet sofort weiter. Voraussetzung für asynchrone Übermittlung ist, daß keine Ergebnisse zurückgeliefert werden. Asynchrone Operationen dürfen daher nur *in*-Parameter besitzen. Obwohl es vom CORBA-Standard nicht gefordert wird, arbeiten viele CORBA-Implementierungen so, daß der *oneway*-Operationsaufruf vom Server quittiert wird. Nur dann kann der Klient sicher sein, daß er nicht im Netz verloren ging.

Klassenoperationen können in IDL *nicht* spezifiziert werden. IDL unterstützt elementare und komplexe Datentypen:

Klassenoperation  
IDL-Typen

- |                  |                                    |
|------------------|------------------------------------|
| ■ long           | ■ struct                           |
| ■ short          | ■ union                            |
| ■ unsigned long  | ■ enum                             |
| ■ unsigned short | ■ array (konstante Größe)          |
| ■ float          | ■ string                           |
| ■ double         | ■ octet                            |
| ■ char           | ■ sequence                         |
| ■ boolean        | ■ any (kann jeden Wert darstellen) |

Ein *sequence*-Typ ist ein eindimensionales Feld, das begrenzt (mit maximaler Länge) oder unbegrenzt sein kann. Die aktuelle Länge darf die maximale Länge nicht überschreiten.

Eine IDL-Schnittstelle (*interface*) kann außerdem Attribute besitzen. Jedes Attribut einer Schnittstelle ist logisch äquivalent zu einem Paar von Zugriffsoptionen, je eine zum Schreiben und eine zum Lesen des Attributwertes. Ein Attribut kann als *readonly* deklariert werden. Dann gibt es nur eine Operation zum Lesen des Werts. Das bedeutet jedoch nicht, daß der Attributwert konstant ist. Beispielsweise kann er durch eine Operation der IDL-Schnittstelle modifiziert werden.

Attribut

Eine IDL-Schnittstelle kann von beliebig vielen anderen Schnittstellen erben. Die Angabe `interface D: B1, B2` bedeutet, daß alle in den Basis-Schnittstellen B1 und B2 enthaltenen Definitionen auch in der spezialisierten Schnittstelle D verfügbar sind, als wären sie direkt in D eingetragen worden. In D können neue Deklarationen hinzugefügt werden. Geerbte Bezeichner können in der spezialisierten Schnittstelle redefiniert (überschrieben) werden. Sie bleiben unter dem qualifizierenden Namen `<Basis-Schnittstelle> : <Bezeichner>` weiterhin verfügbar. Unzulässig ist, wenn ein Bezeichner von mehreren Basis-Schnittstellen geerbt wird.

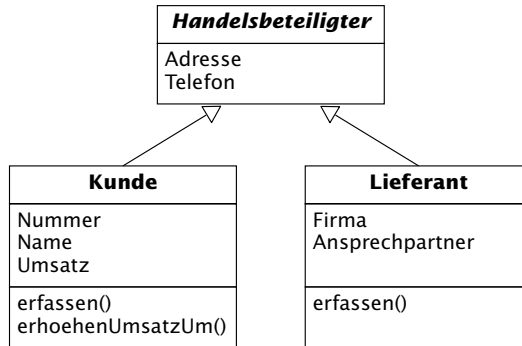
Vererbung



## LE 16 9 Verteilte objektorientierte Anwendungen

Beispiel Für das Klassendiagramm der Abb. 9.4-1 ergibt sich nachfolgende IDL-Definition. Das Attribut `Nummer` soll nur beim Erfassen eines Kunden geschrieben werden. Der Umsatz darf nur durch die Operation `erhoehenUmsatzUm()` aktualisiert werden. Weil für diese beiden Attribute keine `set`-Operation generiert werden soll, werden sie als `readonly` spezifiziert.

Abb. 9.4-1:  
Klassendiagramm  
für Kunden und  
Lieferanten



```
struct AdresseT
{ string Strasse;
  string PLZ;
  string Ort;
};
interface Handelsbeteiligter
{ attribute AdresseT Adresse;
  attribute string Telefon;
};
interface Kunde: Handelsbeteiligter
{ readonly attribute long Nummer;
  attribute string Name;
  readonly attribute float Umsatz;
  void erfassen()
    raises (schonVorhanden);
  void erhoehenUmsatzUm (in float Erhoehung);
};
interface Lieferant: Handelsbeteiligter
{ attribute string Firma;
  attribute string Ansprechpartner;
  void erfassen()
    raises (schonVorhanden);
};
```

Modul Mit Hilfe der Modul-Definition können mehrere IDL-Schnittstellen und andere IDL-Definitionen zu einer logischen Einheit gruppiert werden. Außerhalb eines Moduls werden dann alle Bezeichner von Deklarationen um den Modulnamen ergänzt, d.h. `<Modul name> :: <Bezeichner>`. Dadurch ist es möglich, Schnittstellen, die beispielsweise von verschiedenen Programmierern erstellt wurden, gleich zu benennen, ohne daß Konflikte auftreten.

## Der Zugriff auf die Schnittstelle Kunde erfolgt mittels

Beispiel

```

Personen: : Kunde.
module Personen
{ interface Kunde {...};
  interface Lieferant {...};
};

```

Die IDL-Definition kann auf zahlreiche Programmiersprachen (z.B. C++, C, Smalltalk) abgebildet werden. Um die Transformation der IDL-Definition in eine Programmiersprache zu zeigen, verwenden wir hier als Zielsprache C++. Der Programmierer einer verteilten Anwendung muß außer den IDL-Definitionen das **Schema** kennen, nach dem der *Precompiler* eine IDL-Definition in C++ transformiert. Dazu gehören folgende Abbildungsregeln:

C++ mapping

- Jede Schnittstelle (*interface*) wird zu einer Klasse. Jede IDL-Operation wird auf eine *Member*-Funktion abgebildet. Beim Aufruf der Operation können für *in*-Parameter Konstanten oder Variablen angegeben werden. Ihr Wert bleibt unverändert. Bei *out*- und *inout*-Parametern sind als Argumente nur Variablen erlaubt. Daher werden diese Parameter in C++ als Referenz deklariert. Aus der IDL-Signatur:  
 $T \text{ op } (in \ T \ p1, \ inout \ T \ p2, \ out \ T \ p3)$   
wird in C++:  
 $T \text{ op } (T \ p1, \ T\& \ p2, \ T\& \ p3)$
- Jedes IDL-Attribut wird auf zwei *Member*-Funktionen abgebildet, je eine zum Lesen und Schreiben des Attributs. *ReadOnly*-Attribute werden nur auf Lese-Operationen abgebildet.
- Jedes Modul wird auf ein *namespace*-Konstrukt in C++ abgebildet.
- Die IDL-Basistypen werden äquivalenten Typen von C++ zugeordnet, z.B. `short` wird zu `CORBA::Short`.
- Analog gibt es für alle anderen IDL-Typen eine Transformationsvorschrift.

## 9.5 Entwicklung eines verteilten Systems

Eine verteilte Softwareentwicklung erfordert mehrere Schritte, die wir hier im einzelnen betrachten. Dieses einfache Beispiel zeigt, wie der Entwicklungsprozeß grundsätzlich durchgeführt wird. Es basiert auf *Orbix*. Die hier verwendete *Orbix*-Version ist CORBA 2.0-konform und verwendet C++. Wir gehen in folgenden Schritten vor:

- 1 Server-Klassen in IDL spezifizieren.
- 2 IDL-Schnittstellen mit dem *Precompiler* übersetzen.
- 3 Server-Objekte implementieren.
- 4 Klient-Anwendung implementieren.
- 5 Server-Hauptprogramm erstellen.

**1 Server-Klassen in IDL spezifizieren**

Dieser Schritt gliedert sich in folgende Teilschritte:

- Identifizieren der Server-Klassen im OOA-Modell. Das sind diejenigen Klassen, die Dienstleistungen für andere Klassen zur Verfügung stellen und im Netz auf einem Server-Rechner liegen sollen.
- Ermitteln, welche Attribute und Operationen dieser Klassen netzweit und welche lokal zur Verfügung gestellt werden müssen. Derartige Klassen sind entsprechend zu zerlegen.
- Erstellen von IDL-Spezifikationen für alle Server-Klassen.

**Beispiel** Für die einfache Klasse der Abb. 9.5-1 ergibt sich folgende IDL-Definition:

```
interface SimpleObject
{
    attribute short val ue1;
    attribute short val ue2;
    short calcSum();
};
```

SimpleObject
Value1: Short
Value2: Short
calcSum(): Short

Abb. 9.5-1:  
Einfache Klasse

**2 IDL-Schnittstellen mit dem Precompiler übersetzen**

Aus der IDL-Definition generiert der *Precompiler* vier Dateien, deren Inhalt sich der Programmierer *nicht* ansehen muß.

- Die *Client-Header*-Datei muß ins *Client*-Hauptprogramm eingebunden werden.
- Die *Client*-Implementierungsdatei enthält Funktionen für den Zugriff auf die Klassen-Schnittstelle des entfernten Objekts.
- Die *Server-Header*-Datei muß in die Implementierung der Klassen-Schnittstelle (Objekt-Implementierung) eingebunden werden.
- Die *Server*-Implementierungsdatei enthält den Programmcode, der im Server die Verbindung zwischen dem ORB und der Objekt-Implementierung herstellt.

**Beispiel** Für unsere Beispielklasse *SimpleObject* erstellt *Orbix* folgende Dateien:

- *SimpleObject.hh* (*Client*- und *Server-Header*-Datei),
- *SimpleObjectC.cpp* (*Client*-Implementierungsdatei) und
- *SimpleObjectS.cpp* (*Server*-Implementierungsdatei).

Weil *Orbix* dieselbe Header-Datei für den Klienten und den Server benutzt, werden nur drei Dateien benötigt.

*Orbix* führt folgende Transformationen durch:

- Der Typ *short* der IDL-Definition wird in C++ auf den Typ *CORBA\_Short* abgebildet.
- Das Attribut *val ue1* – und analog *val ue2* – wird abgebildet auf:  

```
//setVal ue1()
virtual void val ue1(CORBA_Short val ue1, ...);
//getVal ue1()
virtual CORBA_Short val ue1( ... );
```
- Die Operation *calcSum()* wird folgendermaßen transformiert:  

```
virtual CORBA_Short calcSum( ... );
```

### 3 Objekt-Implementierung erstellen

Um die Objekt-Implementierung auf dem Server zu erstellen, muß der Programmierer das allgemeine Transformationsschema des IDL-*Precompilers* kennen. Bei Bedarf werden Konstruktoren und Destruktoren hinzugefügt. Die Implementierung steht in einer Unterklasse, die von der BOAImpl-Klasse abgeleitet wird, die vom IDL-Compiler erzeugt wurde.

Die Implementierung des Server-Objekts erfolgt mit der Klasse `SimpleObject_i`. Sie wird in diesem Beispiel von der Klasse `SimpleObjectBOAImpl` abgeleitet, die vom IDL-Compiler generiert wird. Beispiel

```
//SimpleObject_i.h : Header-Datei für die Implementierung
#include "SimpleObject.hh" //vom IDL-Compiler generiert
class SimpleObject_i : public SimpleObjectBOAImpl
{
    CORBA_Short m_value1;
    CORBA_Short m_value2;
public:
    //Lesen und Schreiben der Attribute
    virtual CORBA_Short value1(CORBA_Environment &env);
    virtual CORBA_Short value2(CORBA_Environment &env);
    virtual void value1(CORBA_Short value, CORBA_Environment &env);
    virtual void value2(CORBA_Short value, CORBA_Environment &env);
    //Operation
    virtual CORBA_Short calcSum(CORBA_Environment &env);
};

// SimpleObject_i.cpp : Objekt-Implementierung
#include "simpleobject_i.h"
CORBA_Short SimpleObject_i::value1(CORBA_Environment &env)
{ return m_value1; }
CORBA_Short SimpleObject_i::value2(CORBA_Environment &env)
{ return m_value2; }
void SimpleObject_i::value1(CORBA_Short value, CORBA_Environment
    &env)
{ m_value1 = value; }
void SimpleObject_i::value2(CORBA_Short value, CORBA_Environment
    &env)
{ m_value2 = value; }
// Implementierung der Operation
CORBA_Short SimpleObject_i::calcSum (CORBA_Environment &env)
{ return m_value1 + m_value2; }
```

### 4 Client-Anwendung implementieren

Ein Klient besitzt einen Zeiger auf das Objekt, über den er das Objekt eindeutig identifizieren kann. Mit Hilfe dieses Zeigers können die Operationen des Objekts aufgerufen werden. Gewöhnlich erfolgt der Operationsaufruf durch den Aufruf der entsprechenden Operation im lokalen *Stub*. Da dieser in der verwendeten Programmiersprache erstellt wurde, gibt es keinen Unterschied beim Zugriff auf entfernte und lokale Operationen.

## LE 16 9 Verteilte objektorientierte Anwendungen

Beispiel Bei unserem kleinen Beispiel besteht die *Client*-Anwendung nur aus einem Hauptprogramm. Normalerweise wird sie jedoch aus mehreren Klassen bestehen.

```
// Client.Cpp : Client-Anwendung
#include "SimpleObject.hh"
int main (int argc, char** argv)
{ ...
  //Zeiger auf das Server-Objekt; zuweisungskompatibel mit
  //SimpleObject*
  SimpleObject_var SimpleObjectVar;
  //Eingabe des Servernamens (Rechner)
  ...
  //Binden des Servernamens(Rechner) an das Server-Objekt
  ...
  char txt[10];
  int auswahl, val;
  do
  { cout << "(1) Neue Werte setzen" << endl ;
    cout << "(2) Summe berechnen" << endl ;
    cout << "(0) Programm verlassen" << endl ;
    //Eingabe der gewünschten Auswahl
    ...
    try
    { switch (auswahl)
      {
        case 1 :   cout << "Wert 1: " ; cin >> txt ;
                  val = atoi (txt);
                  SimpleObjectVar->value1 (val);
                  cout << "Wert 2: " ; cin >> txt ;
                  val = atoi (txt);
                  SimpleObjectVar->value2 (val); break ;

        case 2 :   CORBA_Short sum;
                  sum = SimpleObjectVar->calculateSum();
                  cout << sum << endl; break ;

      }
    } //end try
    ...
  } //end do
  return 0 ;
}
```

### 5 Server-Hauptprogramm erstellen

Nun kann das Server-Hauptprogramm erstellt werden, das zwei Aufgaben erledigen muß:

- Für jede Server-Klasse ist mindestens ein Objekt zu erzeugen, damit der Klient darauf zugreifen kann.
- CORBA muß mitgeteilt werden, daß der Server bereit ist, Operationsaufrufe vom Klienten entgegenzunehmen. Das erfolgt durch die Operation `impl_i_s_ready()`.

Jeder Server besitzt einen Namen, der auf der jeweiligen Plattform eindeutig ist. Im *Implementation Repository* wird die Abbildung des Server-Namens auf die Prozeß-Identifikation und den Computer-

namen des ausführbaren Codes verwaltet, der den Server implementiert. Der Programmierer eines Servers muß ihn daher im *Implementation Repository* registrieren lassen. Der Code wird dann automatisch gestartet – sofern er noch nicht ausgeführt wird – wenn ein *request* für ein Objekt eintrifft, dessen Objektreferenz zu einem speziellen Server gehört.

```
//Srv_Main.C: Hauptprogramm Server-Implementierung
#include "Si mpl eObj ect_i . h"
int main()
{ //Erzeugen ei nes Objekts
  Si mpl eObj ect_i aSi mpl eObj ect;
  try
  { //Ini tiali sierung des Server-Objekts ist abgeschl ossen
    CORBA_Orbi x. i mpl _i s_ready("Si mpl eObj ect");
  }
  ...
  return 0;
}
```

Beispiel



**CORBA (Common Object Request Broker Architecture)** CORBA ist der OMG-Standard, der spezifiziert, wie Objekte in einer verteilten, heterogenen Umgebung kommunizieren. Er beschreibt den Aufbau des →ORB, seine Bestandteile sowie deren Verhalten und Schnittstellen.

**IDL (Interface Definition Language)** Die Schnittstellensprache IDL ist eine Sprache zur Spezifikation der Schnittstellen aller Objekte, die von den →Klienten verwendet werden. IDL ist eine rein beschreibende Sprache. Die Implementierung erfolgt in einer Programmiersprache, z.B. C++.

**Klient (client)** Der Klient ist eine Softwareeinheit, die eine Operation eines Objekts auf einem entfernten Server benutzen möchte.

**Objekt-Implementierung (object implementation)** Die Objekt-Implementierung definiert bei verteilten Systemen das Verhalten eines Objekts auf dem Server, in dem sie festlegt, welche Verarbeitung beim Aufruf einer Operation auszuführen ist. Außerdem legt sie fest, welche Daten benötigt werden, um den Zustand eines konkreten Objekts zu repräsentieren.

**Objektreferenz (object reference)** Bei verteilten Systemen identifiziert der →Klient ein Objekt auf dem Server

über seine systemweit eindeutige Objektreferenz, die später auf die physische Adresse des Objekts abgebildet wird.

**OMA (Object Management Architecture)** Die Grundlage aller Standardisierungsaktivitäten der →OMG ist die OMA. Diese Architektur unterteilt die Bestandteile einer verteilten Anwendung in mehrere Komponenten. Den Kern bildet der →ORB (*Object Request Broker*), der als Kommunikationszentrale im Mittelpunkt der Architektur steht. Weitere Komponenten sind die *application interfaces*, die *domain interfaces*, die *object services* und die *common facilities*.

**OMG (Object Management Group)** Systemanbieter und Anwender objektorientierter Techniken haben sich 1989 zur OMG (*Object Management Group*) zusammengeschlossen. Die OMG verfolgt das Ziel, Standards und Spezifikationen für verteilte objektorientierte Anwendungen zu schaffen.

**OMG-Objektmodell (OMG object model)** Das OMG-Objektmodell liegt allen Spezifikationen der →OMG zugrunde. Es beschreibt alle objektorientierten Konzepte, die für Klienten wichtig sind und die Konzepte für die Ausführung der Operationen auf dem Server.

## LE 16 Glossar/Zusammenhänge/Aufgaben

**ORB (Object Request Broker)** In verteilten System wird die Kommunikation zwischen →Klient und Server vom ORB durchgeführt. Er ist vergleichbar mit einer Telefonvermittlung, der das Anrufen anderer Teilnehmer und das Entgegennehmen von Anrufen realisiert.

**Request** Mit einem *request* fordert der →Klient ein Objekt auf dem Server zur Ausführung einer Operation auf.

### **Schnittstelle (interface)**

**1** Die Menge aller Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die Schnittstelle der Klasse bzw. des Objekts.

**2** Die IDL-Schnittstelle spezifiziert die Signaturen von Operationen, die ein →Klient aufrufen kann. Sie stellt die wichtigste Komponente einer IDL-Definition dar.

### **Signatur (signature)**

**1** Die Signatur einer Operation besteht aus dem Namen der Operation, den Namen und Typen aller Parameter, und dem Ergebnistyp der Operation.

**2** Die Signatur einer Operation definiert den Namen der Operation, die Namen und Typen aller Parameter, den Ergebnistyp und die Bezeichnungen aller Ausnahmebehandlungen (*exceptions*) im Fehlerfall.

Die **OMG** hat mit der **OMA** einen Architektur-Standard für verteilte objektorientierte Anwendungen spezifiziert. **CORBA** ist der OMG-Standard, der spezifiziert, wie Objekte in einer verteilten, heterogenen Umgebung kommunizieren. Er beschreibt den Aufbau des **ORB**, seine Bestandteile sowie deren Verhalten und Schnittstellen. Der ORB identifiziert entfernte Objekte, gibt den Operationsaufruf eines Klienten an die **Objekt-Implementierung** auf einem entfernten Server weiter und liefert die Ergebnisse zurück. Die **IDL** ermöglicht eine programmiersprachenunabhängige Definition aller Server-Klassen. Die **Schnittstelle** ist die wichtigste Komponente einer IDL-Definition.



Aufgabe  
10 Minuten

**1 Lernziel: Kommunikation zwischen verteilten Objekten erläutern können.**

Erklären Sie,

- a** wie ein Objekt auf einem entfernten Server identifiziert wird,
- b** wann ein Operationsaufruf eine Ausnahmebehandlung auf der Server-Seite auslöst,
- c** den Unterschied zwischen dem »normalen« Operationsaufruf innerhalb eines Programms und dem Operationsaufruf eines entfernten Objekts.



Aufgabe  
10–15 Minuten

**2 Lernziel: ORB erklären können.**

Erläutern Sie,

- a** wofür der ORB verantwortlich ist,
- b** aus welchen Komponenten er besteht und wofür diese Komponenten verantwortlich sind,
- c** den Unterschied zwischen statischen und dynamischen Operationsaufrufen.

## Aufgaben LE 16

### 3 Lernziel: OMA erläutern können.

Beschreiben Sie, aus welchen Komponenten die OMA-Architektur besteht und für welche Aufgaben diese Komponenten verantwortlich sind.

Aufgabe  
5–10 Minuten

### 4 Lernziel: IDL-Definitionen für Klassen erstellen können.

Erstellen Sie für das Klassendiagramm der Abb. LE16-A4 die IDL-Definition.

Aufgabe  
15 Minuten

Bei einem Girokonto werden Zinsen quartalsweise gutgeschrieben bzw. abgebucht. Bei einem Sparkonto erfolgt die Gutschrift der Zinsen jährlich. Gehen Sie bei der Berechnung der Zinsen davon aus, daß der Betrag aus gespeicherten Daten – die hier nicht modelliert sind – errechnet werden kann.

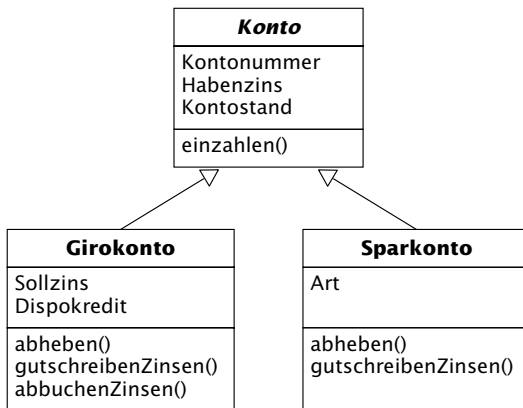


Abb. LE16-A4:  
Klassendiagramm  
Konten



## 10 Erstellen eines Entwurfsmodells mittels Drei-Schichten-Architektur (Teil 1)



- Wissen, wie die grundlegende Architektur beim objektorientierten Entwurf aussieht. wissen
- Zwei-, Drei- und Mehr-Schichten-Architekturen unterscheiden können. verstehen
- MVC-Architektur erklären können.
- Erklären können, welche Änderungen am OOA-Modell für die Transformation in den Entwurf notwendig sind.
- GUI-Klassen für Erfassungs- und Listenfenster entwerfen können. anwenden
- GUI-Klassen mit den Fachkonzeptklassen verbinden können.
- *Singleton*- und Beobachter-Muster anwenden können.



- Voraussetzungen für diese Lehreinheit sind die objektorientierten Konzepte der Analyse und des Entwurfs sowie der UML-Notation, wie sie in den Kapiteln 2 und 6 beschrieben sind.
- Außerdem sollte das Kapitel 5 bekannt sein, in dem wichtige Grundlagen für die Gestaltung der Benutzungsoberfläche eingeführt werden.
- Grundlagen in C++ oder Java erleichtern das Verstehen dieser Lehreinheit.

- i 10.1 Architekturentwurf 370
- 10.2 Entwurf der Fachkonzeptschicht 377
- 10.3 Entwurf der GUI-Schicht und Anbindung an die Fachkonzept-Klassen 382

## 10.1 Architekturentwurf

### Grundlegende Entwurfsentscheidungen

Eine fundamentale Entscheidung für den Entwurf ist die Wahl der Programmiersprache. Wir gehen davon aus, daß eine objektorientierte Programmiersprache (OOP) wie C++, Java oder Smalltalk gewählt wird. Prinzipiell ist auch die Verwendung einer prozeduralen Sprache möglich, wobei bei der Transformation der objektorientierten Konzepte stets starke Qualitätsverluste hinsichtlich Wartbarkeit und Änderbarkeit auftreten.

GUI-System Desweiteren sind die Entscheidungen zu treffen, welches **GUI-System** für die Realisierung der Benutzungsoberfläche verwendet wird und mit welchen Datenbanksystemen die Datenhaltung zu realisieren ist. Eventuell ist auch eine einfache Datenhaltung mittels flacher Dateien zu wählen. Im allgemeinen wird als GUI-System dasjenige System verwendet, mit dem bereits der Prototyp erstellt wurde. Jedoch können geänderte Anforderungen, wie beispielsweise die geforderte Portabilität des Systems, ein anderes GUI-System notwendig machen.

Datenhaltung Die Anbindung an ein bestimmtes Datenbanksystem ist häufig bereits vorgegeben. Der Entwurf der Datenhaltung hängt ganz wesentlich davon ab, ob eine objektorientierte Datenbank, eine relationale Datenbank oder ein flaches Dateisystem verwendet wird. Der Entwurf dieser Anbindung wird in den Kapiteln 10.4 bis 10.6 erläutert. ↕

Kapitel 10.4  
bis 10.6

Einsatz eines  
Datenbanksystems

Wann ist der Einsatz einer Datenbank erforderlich und wann reicht ein einfaches Dateisystem aus?

Treffen mehrere der folgenden Kriterien zu, dann spricht viel für die Verwendung eines Datenbanksystems:

- 1 Mehrere Benutzer und/oder Anwendungsprogramme müssen parallel mit dem Datenbestand arbeiten.
- 2 Alle Daten müssen redundanzarm gespeichert werden, auch wenn sie aus verschiedenen Anwendungen stammen.
- 3 Die Menge der Daten ist sehr umfangreich.
- 4 Nach technischen Fehlern muß der automatische Wiederanlauf des Systems einschließlich des Herstellens der Datenkonsistenz gewährleistet sein.
- 5 An die Benutzer müssen dedizierte Zugriffsrechte vergeben und überwacht werden können.
- 6 In der Nutzungsphase müssen *ad hoc*-Abfragen durch den Benutzer möglich sein.

relational oder  
objektorientiert

Für **relationale Datenbanksysteme** sprechen folgende Kriterien:

- 1 Es sind relativ einfache, formatierte Datenbestände zu verwalten. Dies trifft in der Regel auf betriebswirtschaftlich-administrative Anwendungen zu.

2 Im Fehlerfall kann auf die Tabellen immer direkt zugegriffen werden.

3 Sie sind in der Industrie weit verbreitet.

Für den Einsatz **objektorientierter Datenbanksysteme** sprechen folgende Kriterien:

1 Es müssen komplexe graphenartige Strukturen verwaltet werden. Das trifft beispielsweise auf CASE-, CAD- und CAM-Anwendungen sowie auf Multimedia-Anwendungen zu.

2 Bei objektorientierten Anwendungsprogrammen ist die Anbindung an das objektorientierte Datenbanksystem einfach und die Schnittstelle wird vom Hersteller der Datenbank angeboten.

Um die Wartbarkeit zu optimieren, sollte bei der Bildung der Komponenten das folgende allgemeine **Entwurfsprinzip** realisiert werden: Notwendige Änderungen in einer Komponente sollten keine oder möglichst geringe Auswirkungen auf andere Komponenten haben.

Entwurfsziel

Das bedeutet im einzelnen:

- Die Schnittstellen zwischen den Komponenten sollen möglichst schmal sein.
- Die Verteilung im Netz soll möglichst flexibel sein.
- Jede Komponente soll ihre eigenen Aufgaben vollständig allein durchführen und nicht Teile davon an andere Komponenten delegieren. Beispielsweise sollte die Benutzungsoberfläche keine Plausibilitätsprüfungen durchführen, die Bestandteil des Fachkonzepts sind.

Während des Architekturentwurfs wird die Systemarchitektur erstellt, d.h. es werden die grundlegenden Komponenten des Softwaresystems und die Abhängigkeiten zwischen ihnen dargestellt. Im Grunde handelt es sich hier um ein Architektur-Muster. Dieses Muster unterteilt die Anwendung in Schichten (*layers, tiers*). Weiterführende Literatur zu den Schichten-Architekturen finden Sie in /Larman 98/, /Fowler 97a/ und /Balzert 96/.

Die meisten Informationssysteme werden heute – mehr oder weniger – in einer **Zwei-Schichten-Architektur** (*two-tier architecture*) entworfen. Sie besteht aus einer Anwendungsschicht, in der die Benutzungsoberfläche und das Fachkonzept in einer einzigen Schicht fest verzahnt sind, und einer Datenhaltungsschicht (Abb. 10.1-1). Bei einer Client-Server-Anwendung befindet sich die An-

2-Schichten-Architektur

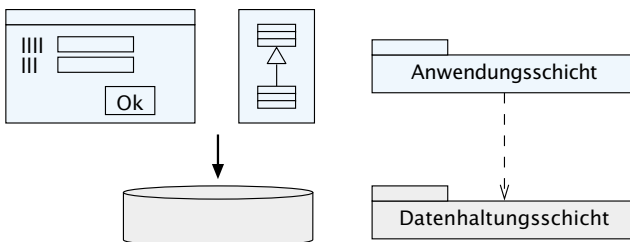


Abb. 10.1-1: Zwei-Schichten-Architektur

## LE 17 10 Entwurf einer Drei-Schichten-Architektur

wendungsschicht auf dem *Client* und die Datenhaltung auf dem Server.

**Beispiel** Eine Zwei-Schichten-Architektur wird häufig bei Visual-Basic-Programmen verwendet. In der Anwendungsschicht wird die Benutzungsoberfläche mittels *forms* realisiert und mit der Fachkonzeptlogik verbunden. Die Datenhaltung kann beispielsweise in einer Datenbank erfolgen.

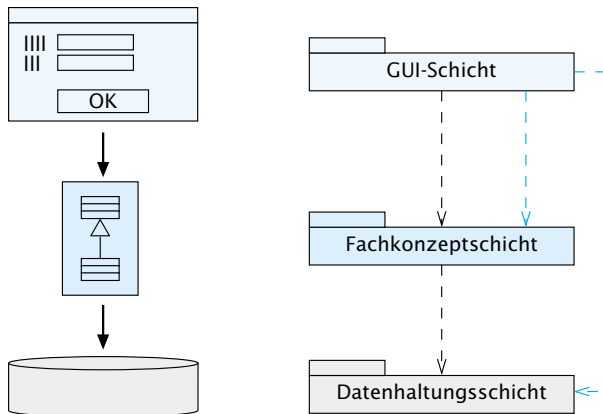
3-Schichten-Architektur Die **Drei-Schichten-Architektur** (*three-tier architecture*) besteht aus (Abb. 10.1-2)

- der GUI-Schicht,
- der Fachkonzeptschicht und
- der Datenhaltungsschicht.

Die **GUI-Schicht** realisiert die Benutzungsoberfläche einer Anwendung. Dazu gehören die Dialogführung und die Präsentation aller Daten in Fenstern, Berichten usw. Die **Fachkonzeptschicht** modelliert den funktionalen Kern der Anwendung. Außerdem enthält sie die Zugriffe auf die **Datenhaltungsschicht**, in der die jeweilige Form der Datenspeicherung realisiert wird, z.B. mit einem objekt-orientierten Datenbanksystem.

Bei der Drei-Schichten-Architektur sind zwei Ausprägungen möglich. Bei einer **strengen Drei-Schichten-Architektur** kann die GUI-Schicht nur auf die Fachkonzeptschicht und letztere nur auf die Datenhaltungsschicht zugreifen. Der Vorteil dieser Architektur ist, daß die GUI-Schicht ausschließlich von der Fachkonzeptschicht abhängt und nicht von der gewählten Speicherung der Daten. Eine **flexible Drei-Schichten-Architektur** ergibt sich, wenn die GUI-Schicht nicht nur auf die Fachkonzeptschicht, sondern zusätzlich auf die Datenhaltungsschicht zugreifen darf. Den Vorteilen der größeren Flexibilität und besseren *Performance* stehen bei dieser Ausprägung jedoch die Nachteile von geringerer Wartbarkeit, Änderbarkeit und Portabilität gegenüber (vergleiche /Balzert 96/).

Abb. 10.1-1:  
Zwei-Schichten-  
Architektur



Eine grundlegende Idee der Schichten-Architektur ist, daß keine andere Schicht direkt auf die Benutzungsoberfläche zugreifen kann. Das bedeutet insbesondere für die Fachkonzeptschicht, daß sie keinerlei Wissen über ihre Benutzungsoberfläche besitzen darf. Oft spricht man auch von einer *Model-View*-Architektur. Diese konsequente Trennung ermöglicht eine getrennte Entwicklung von Benutzungsoberfläche und Fachkonzept und einen leichteren Austausch der Benutzungsoberfläche.

Model-View-Architektur

Es gibt zwei Möglichkeiten, damit ein Fenster die Informationen aus dem Fachkonzept erhalten kann:

- Beim *polling* sendet die Benutzungsoberfläche in regelmäßigen Intervallen Botschaften an die Fachkonzeptobjekte, um Änderungen, die sich auf die Oberfläche auswirken, abzufragen. Dieses Verfahren kann sich jedoch ungünstig auf die *Performance* auswirken.
- Die indirekte Kommunikation wird mittels des Beobachter-Musters (*observer*) realisiert (Kapitel 7.7). Das Fachkonzeptobjekt schickt der Benutzungsoberfläche lediglich eine Botschaft, die das Vorliegen von Änderungen signalisiert. Man spricht auch von Benachrichtigung (*notify*). Die Oberfläche holt sich daraufhin selbständig die notwendigen Daten.

polling

indirekte Kommunikation  
Kapitel 7.7



Die Entkopplung von Benutzungsoberfläche und Fachkonzept ist heute ein Grundprinzip des Softwareentwurfs. Viele dieser Ideen haben ihren Ursprung in der Architektur **MVC** (*Model/View/Controller*), die erstmalig für Smalltalk-80 verwendet wurde.

MVC

Die MVC-Architektur /Krasner, Pope 88/ besteht aus den drei Klassen *Model*, *View* und *Controller* (Abb. 10.1-3). Das *Model*-Objekt repräsentiert das Fachkonzeptobjekt. Oft gibt es mehrere Möglichkeiten, die fachlichen Daten zu präsentieren. Für jede Präsentation gibt es ein *View*-Objekt. Das *Controller*-Objekt bestimmt, wie die Benutzungsoberfläche auf Eingaben reagiert. Jedes *View*-Objekt besitzt ein zugehöriges *Controller*-Objekt, das diese Darstellung mit der Eingabe verbindet. Das impliziert, daß es zu jedem *Model*-Objekt eine beliebige Anzahl von Paaren (*View*, *Controller*) geben kann,

MVC-Struktur

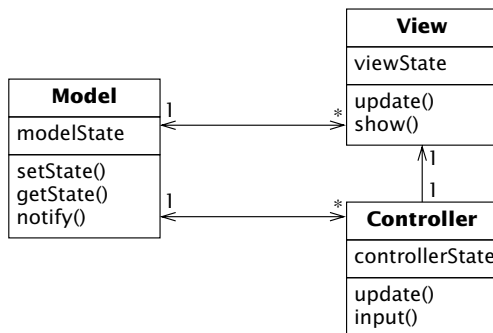


Abb.10.1-3:  
Architektur des  
MVC

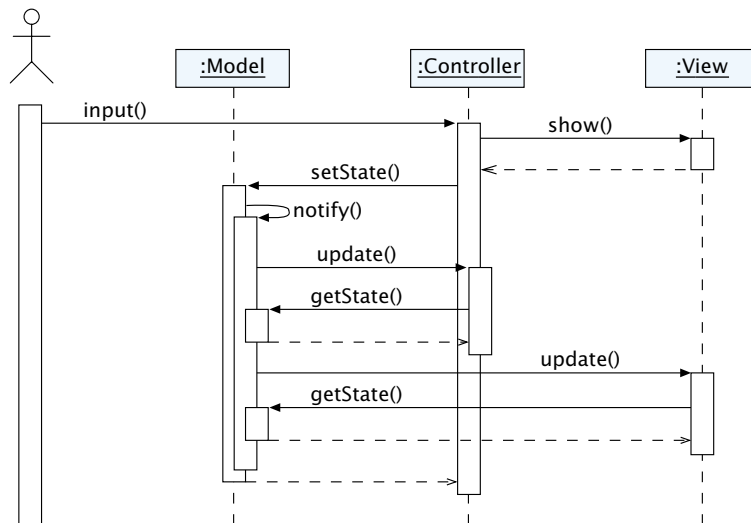
## LE 17 10 Entwurf einer Drei-Schichten-Architektur

jedoch mindestens eines. Wir sprechen auch von *dependents of the model*.

MVC-Interaktionen

Eine typische Interaktion sieht in der MVC-Architektur wie folgt aus. Wenn der Benutzer eine Eingabe durchführt, wird diese Eingabe gegebenenfalls – mit der *show*-Operation – im zugehörigen *View*-Objekt dargestellt. In jedem Fall sendet der *Controller* die Botschaft *setState()* an das *Model*-Objekt. Das *Model*-Objekt ändert gegebenenfalls seine Daten und ruft die *notify*-Operation auf, die alle assoziierten *View*- und *Controller*-Objekte mittels einer *update*-Botschaft informiert. Daraufhin können sich die benachrichtigten Objekte selbst in einen konsistenten Zustand mit dem *Model*-Objekt (Abb. 10.1-4) bringen, indem sie sich mittels der Botschaft *getState()* die notwendigen Daten vom *Model*-Objekt holen.

Abb. 10.1-4:  
Kommunikation  
zwischen Model,  
View und Controller



Die MVC-Architektur realisiert folgende Ideen: Das *Model*-Objekt (Fachkonzept) weiß nicht, *wie* seine Daten auf der Oberfläche dargestellt werden. Es darf nicht direkt auf seine assoziierten *View*- und *Controller*-Objekte zugreifen und dort Daten ändern. Es besitzt lediglich eine Liste aller von ihm abhängigen Objekte. Bei einer Aktualisierung muß es alle seine abhängigen Objekte über diese Änderung informieren. Wenn *View*- oder *Controller*-Objekte geändert oder ausgetauscht werden, ist das *Model*-Objekt nicht davon betroffen.

In Smalltalk-80 bildet MVC ein *Framework*, das durch die abstrakten Klassen *Model*, *View* und *Controller* sowie zahlreiche konkrete Unterklassen realisiert ist. Die Trennung von *Controller* und *View* gilt inzwischen als überholt. In modernen GUI-*Frameworks* werden *Controller* und *View* zusammengefaßt. In der MFC-Klassenbibliothek

(*Microsoft Foundation Classes*) wird diese Struktur als *Document View* bezeichnet, wobei das *Document*-Teilsystem dem *Model* und das *View*-Teilsystem der Kombination von *View* und *Controller* entspricht.

Die GUI-Schicht erfüllt in einer Drei-Schichten-Architektur zwei unterschiedliche Aufgaben. Das ist einerseits die Präsentation der Information und andererseits die Kommunikation mit der Fachkonzeptschicht. Entsprechend diesen Aufgaben kann eine separate Zugriffsschicht zur Fachkonzeptschicht gebildet werden /Fowler 97a/. Man spricht dann von einer **Mehr-Schichten-Architektur** (*multi-tier architecture*). Die GUI-Schicht befaßt sich dann nur noch mit der Präsentation der Informationen. Normalerweise löst sie ihre Aufgaben mit Hilfe eines *GUI-Frameworks*. Die **Fachkonzept-Zugriffsschicht** ist verantwortlich für alle Zugriffe auf die Fachkonzeptschicht. Die GUI-Schicht arbeitet normalerweise mit einer kleinen Menge von relativ einfachen Typen, während die Fachkonzeptschicht Typen beliebiger Komplexität besitzen kann. Die Zugriffsschicht paßt die Daten der Fachkonzeptschicht für die Präsentation durch die GUI-Schicht an. Auf diese Weise verbirgt sie die komplexen Beziehungen der Fachkonzeptschicht vor der GUI-Schicht.

Schichten der Benutzeroberfläche

Eine optimale Schichten-Architektur realisiert folgende Entwurfsziele:

Entwurfsziele

■ Wiederverwendbarkeit

Jede Schicht besitzt eine präzise definierte Aufgabe und Schnittstelle. Eine vorhandene Schicht kann entweder direkt wiederverwendet werden oder sie wird um eine weitere Schicht ergänzt, die zusätzlich benötigte Aufgaben durchführt.

■ Änderbarkeit/Wartbarkeit

Solange die Schnittstelle einer Schicht unverändert bleibt, kann deren interne Organisation beliebig verändert werden, ohne daß sich die Änderungen außerhalb der Schicht auswirken.

■ Portabilität

Hardwareabhängigkeiten können in einer Schicht isoliert und modifiziert werden, ohne den Rest des Systems zu tangieren.

Diesen Vorteilen steht der Nachteil gegenüber, daß sich die Schichten-Architektur negativ auf die *Performance* auswirkt.

Die Datenhaltung ist am einfachsten zu realisieren, wenn ein objektorientiertes Datenbanksystem verwendet wird. In diesem Fall wird der Datenbankzugriff in die Fachkonzeptschicht integriert. Viele Unternehmen verwenden heute jedoch relationale Datenbanken. Manchmal reichen auch flache Dateien für die Datenhaltung aus. Prinzipiell muß eine objektorientierte Anwendung mit jeder Form der Datenhaltung zurechtkommen. Dem Zugriff auf die Datenhaltung liegt dabei folgende Grundidee zugrunde. Eine Klasse der Fachkonzeptschicht weiß, wie sich ihre Objekte in der Datenbank

Datenhaltung

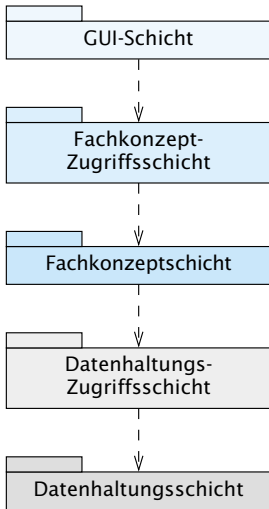
speichern und daraus laden lassen. Wenn ein Objekt auf der Benutzungsoberfläche dargestellt werden soll, dann muß die Fachkonzeptschicht prüfen, ob es bereits im Speicher ist. Falls nicht, dann lädt sie es aus der Datenbank. Aus Sicht der Benutzungsoberfläche ist nicht festzustellen, ob ein Datenbankzugriff stattfindet.

Datenhaltungs-  
Zugriffsschicht

Eine direkte Verbindung zwischen der Fachkonzeptschicht und der Datenhaltung kann zu Problemen führen. Die Fachkonzeptschicht muß dann neben ihrer eigentlichen Aufgabe – der Modellierung des Problembereichs – die Zugriffe auf die Datenhaltung durchführen. Die Lösung liegt in einer separaten **Datenhaltungs-**

**Zugriffsschicht**. Sie füllt die Fachkonzeptobjekte mit Daten aus der Datenbank und aktualisiert die Datenbank bei Änderungen der Fachkonzeptobjekte. Diese Schicht ist ähnlich der Fachkonzept-Zugriffsschicht. Auch hier müssen die Typen der Fachkonzeptschicht gegebenenfalls für die jeweilige Datenhaltung konvertiert werden (Abb. 10.1-5).

Abb. 10.1-5:  
Mehr-Schichten-  
Architektur



verteilte Systeme

Bei Client-Server-Architekturen ist das System zusätzlich auf das Netz zu verteilen. Das bedeutet, daß die Softwarekomponenten den verfügbaren Hardwaresystemen zugewiesen werden. Dieser Prozeß unterscheidet sich völlig von der objektorientierten Modellierung. Im Fall einer Verteilung muß das OOD-Modell unterteilt werden in ein

*Client*-OOD-Modell und ein *Server*-OOD-Modell. Die Benutzungsoberfläche befindet sich immer auf dem *Client*, die Datenhaltung immer auf dem *Server*. Es stellt sich jedoch die Frage, wo sich die Fachkonzeptschicht befinden soll. Bei einem *Client*-basierten Ansatz befindet sie sich auf den *Clients*, was jedoch leistungsfähige Computer auf der *Client*-Seite erfordert. Außerdem müssen Änderungen in der Fachkonzeptschicht eines *Clients* mit allen anderen *Clients* konsistent gehalten werden. Wenn sich die Fachkonzeptschicht auf dem *Server* befindet, kann die Konsistenz leichter sichergestellt werden. Dabei ist es auch möglich, daß sich Fachkonzeptschicht und Datenbank auf unterschiedlichen Servern befinden.

Für viele der beschriebenen Aufgaben stehen heute Werkzeuge und Klassenbibliotheken zur Verfügung. Beispielsweise kann der Anschluß an relationale Datenbanken durch solche Werkzeuge mehr oder weniger stark unterstützt werden. Andere Werkzeuge unterstützen den Entwerfer bei der Verteilung der Anwendung auf das Netz.



- 1 Entscheidungen der Programmierumgebung
  - Welche Programmiersprache?
  - Welches GUI-System?
  - Soll ein Datenbanksystem verwendet werden?
  - Wenn ja, ein relationales oder ein objektorientiertes?
  
- 2 Architektur festlegen
  - Zwei-Schichten-Architektur?
  - Drei-Schichten-Architektur?
  - Mehr-Schichten-Architektur?
  - Client-Server-Verteilung?
  - Welche Werkzeuge sind einzusetzen?
  - Welche Klassenbibliotheken sind einzusetzen?

**Checkliste:  
Entwurf der  
Gesamt-  
architektur**

## 10.2 Entwurf der Fachkonzeptschicht

Das OOA-Modell bildet die erste Version der **Fachkonzeptschicht**, die unter den Aspekten des Entwurfs verfeinert und überarbeitet wird. Diese Aufgabe wird erheblich dadurch vereinfacht, daß von der Analyse zum Entwurf *kein* Paradigmenwechsel stattfindet.

Zur Erinnerung: Das OOA-Modell hat die Aufgabe, das zukünftige System aus Benutzersicht fachlich korrekt zu modellieren, wobei die Effizienz keine Rolle spielt. Mit anderen Worten: es ist ein Modell des Problems. Das OOD-Modell hat dagegen das Ziel, eine effiziente Anwendung zu modellieren, die auf einem Computer ausgeführt werden kann. Das OOD-Modell ist also ein Modell des Lösungsraums. Viele Verfeinerungen erfolgen daher unter dem Gesichtspunkt der Effizienz. Wie bei der klassischen Entwicklung sollte sich die Effizienz der guten Struktur unterordnen. Außer der Effizienz müssen Aspekte der Wiederverwendbarkeit berücksichtigt werden.

OOA-Modell vs.  
OOD-Modell

Während in der Analyse die Objektverwaltung als inhärente Eigenschaft einer Klasse angesehen wurde, muß diese Eigenschaft im Entwurf durch *Container*-Klassen (Kapitel 6.1) implementiert werden.

1 Modifikation der  
Klassenstruktur  
Kapitel 6.1

In den meisten Fällen werden die Analyseklassen 1:1 in die Fachkonzeptschicht übernommen. Wird jedoch die funktionale Komplexität einer vorhandenen Klasse zu hoch, dann sollten Teilaufgaben an detailliertere Klassen delegiert werden. Um die geforderte *Performance* zu erreichen, ist es auch gerechtfertigt, Klassen mit starker Interaktion – d.h. mit einer hohen Kopplung – zusammenzufassen. Ebenso können weitere Klassen hinzugefügt werden, um Zwischenergebnisse zu modellieren, d.h. mehrere abgeleitete Attribute in einer neuen Klasse zu »bündeln«.

Assoziative Klassen können in den objektorientierten Programmiersprachen nicht realisiert werden und sind daher in »normale« Klassen aufzulösen.

**2 Verfeinern der Attribute**

Für Attribute, die im OOA-Klassendiagramm als »abgeleitet« gekennzeichnet wurden, ist zu prüfen, ob diese abgeleiteten Werte zu speichern sind oder ob sie jeweils aktuell berechnet werden sollen. Redundante Attribute sind im allgemeinen nur dann sinnvoll, wenn sie komplizierte oder umfangreiche Berechnungen einsparen und sich die Ursprungsdaten nicht sehr häufig ändern. Bei abgeleiteten Daten ist darauf zu achten, daß sie nur über die Basisdaten geändert werden und mit diesen immer aktuell gehalten werden. Klassen können auch um weitere abgeleitete Attribute erweitert werden, die Zwischenergebnisse speichern.

Oft können die Attribute einer Klasse ganz unterschiedlich ausgedrückt werden. Beispielsweise kann die Position eines Punkts durch seine Polar- oder seine kartesischen Koordinaten beschrieben werden. In der Analyse kann eine dieser Formen unter problemadäquaten Gesichtspunkten gewählt werden. Im Entwurf werden bei Bedarf Operationen geschrieben, um beispielsweise die kartesischen Koordinaten in die Polarkoordinaten umzurechnen.

**3 Verfeinern der Operationen**

Die spezifizierten Operationen sind aus Entwurfssicht zu beschreiben. Sie können von stark unterschiedlicher Komplexität sein. Wird der Algorithmus zu umfangreich, dann muß eine komplexe Operation in einfachere, interne Operationen zerlegt werden. Gegebenenfalls müssen neue Klassen identifiziert werden.

Besitzt die Klasse einen Lebenszyklus, so ist eine auszuführende Operation von dem jeweiligen Objektzustand abhängig. Dann muß der Algorithmus entsprechende Abfragen enthalten oder es ist das Zustandsmuster (siehe Kapitel 6.9) anzuwenden.

**Kapitel 6.9**  
**4 Verfeinern von Assoziationen**  
**Kapitel 6.4**

Während im OOA-Modell alle Assoziationen inhärent bidirektional sind, muß im Entwurf für jede Assoziation die zu implementierende Navigationsrichtung eingetragen werden (Kapitel 6.4).

In der Analyse sollten Sie redundante Assoziationen vermeiden. Im Entwurf sind die Assoziationen unter dem Gesichtspunkt des optimalen Zugriffs auf Objekte zu modellieren. Eventuell sind nach dem Hinzufügen neuer Assoziationen jetzt Assoziationen aus der Analyse überflüssig und können entfernt werden. Es ist auch möglich, daß Assoziationen im OOA-Modell nicht für die Kommunikation der Objekte benötigt werden und im Entwurf entfallen. Gehen Sie dazu folgendermaßen vor: Prüfen Sie für jede Operation, welche Assoziationen sie »durchlaufen« muß, um an die benötigten Informationen zu gelangen. Prüfen Sie, wie viele Objekte jeweils »betrachtet« werden müssen, um die gewünschten Daten zu erhalten.

Prüfen Sie insbesondere, ob eine Assoziation bidirektional entworfen werden muß. Ein relativ einfacher Fall liegt vor, wenn nur eine Richtung implementiert werden muß. Aber auch wenn in beiden Richtungen eine Navigation stattfindet, muß die Assoziation nicht unbedingt bidirektional implementiert werden.



Eine Assoziation zwischen den Klassen A und B wird nur in der Richtung von A nach B als Zeiger in A implementiert. Bei einem Zugriff in der Gegenrichtung müssen alle Objekte von A betrachtet und mit der Bedingung in B gefiltert werden. Dieser Ansatz ist dann sinnvoll, wenn die Gegenrichtung nur sehr selten benötigt wird und dieser Zugriff nicht zeitkritisch ist. Die unidirektionale Realisierung der Assoziation besitzt den Vorteil, daß keine Inkonsistenzen auftreten können. Beispiel

Auch die Vererbungsstruktur muß im Entwurf überarbeitet werden. In der Analyse haben wir Operationen, die für mehrere Unterklassen gelten, so »hoch wie möglich« in die Vererbungsstruktur eingefügt, sofern sie eine gemeinsame Beschreibung besitzen. 5 Verfeinern der Vererbung

Im OOA-Modell der Abb. 10.2-1 gilt die Operation erfassen() für alle Objekte ihrer Unterklassen. Da das Erfassen bei beiden Unterklassen unterschiedlich implementiert wird, erhält im OOD-Modell jede Unterklasse diese Operation. Die abstrakte Operation in der Klasse Person sorgt für einheitliche Schnittstellen. Beispiel

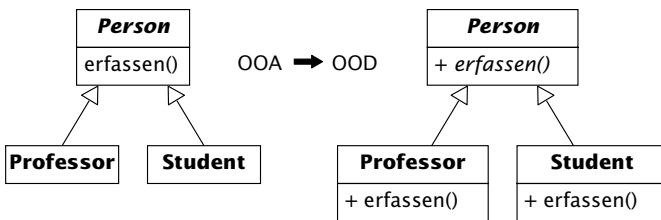


Abb. 10.2-1:  
Einführen von  
abstrakten  
Operationen

Enthält ein OOA-Modell die Klassen Rechteck, Ellipse und Linie (mit entsprechenden Unterklassen), die eine Reihe von gleichartigen Operationen besitzen (z.B. verschieben, duplizieren, vergrößern/verkleinern) dann beschreiben wir diese Gemeinsamkeiten durch eine abstrakte Oberklasse Grafikobjekt, sofern dies noch nicht im OOA-Modell durchgeführt wurde. abstrakte Klassen

Abstrakte Klassen dienen vor allem dazu, das Konzept der Vererbung voll auszunutzen. Sie werden stets künstlich in das Modell eingefügt. /Wirfs-Brock 90/ empfiehlt, so viele abstrakte Klassen wie möglich zu schaffen, weil dadurch das Hinzufügen neuer Klassen erleichtert wird.

Der Vererbungsmechanismus birgt die Gefahr, daß Attribute und Operationen in einer Klasse nur zu dem Zweck »aufgesammelt« werden, um dem Programmierer Schreibaufwand zu sparen. Jacobson spricht hier von *spaghetti inheritance*. Diese willkürlich geschaffenen Klassen lassen sich leicht erkennen. Der Klassenname besitzt keine Aussagefähigkeit oder steht in keiner Beziehung zu den Attributen und/oder Operationen der Klasse. Besonders wenn eine neue Klasse nachträglich in eine Vererbungsstruktur eingefügt keine spaghetti inheritance

## LE 17 10 Entwurf einer Drei-Schichten-Architektur

wird, muß sorgfältig auf diese Kriterien geachtet werden. Demgegenüber steht oft ein beträchtlicher Aufwand für die Restrukturierung.

maximaler Polymorphismus

Ein wichtiges Entwurfsziel ist die Maximierung des Polymorphismus. Gehen Sie dazu folgendermaßen vor:

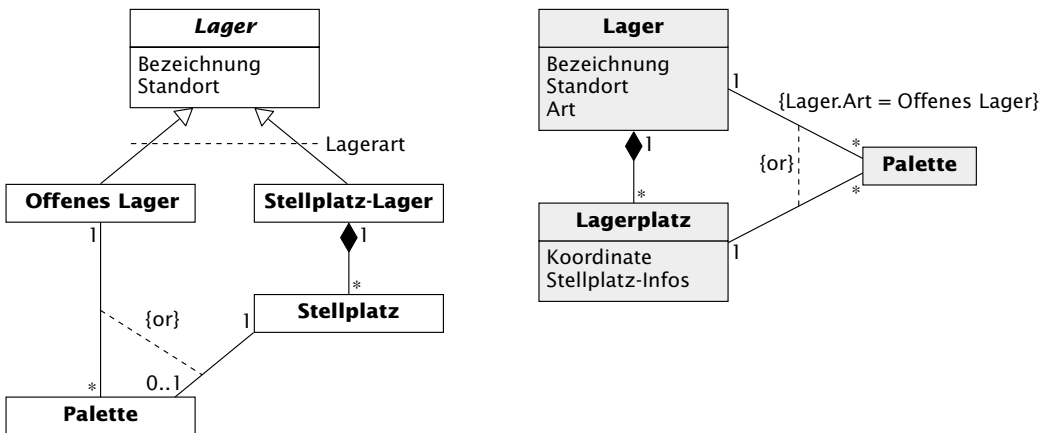
- 1 Alle Operationen von Unterklassen sind so hoch wie irgend möglich in der Vererbungshierarchie einzuordnen.
- 2 Die Namen von Operationen sind so zu wählen, daß man immer einen einzigen Namen für konzeptionell gleiche Operationen verwendet, z.B. drucken() oder erfassen().
- 3 Alle Operationen sind in der Schnittstelle so allgemein wie irgend möglich zu halten. Dazu ist zu überlegen, welche Änderungen evtl. an dem System vorgenommen werden können.

Komprimieren

Es kann auch sinnvoll sein, eine Vererbungsstruktur wieder zu einer Klasse zusammenzufassen. Dadurch wird ein Teil der Semantik, die im statischen Modell spezifiziert ist, in das dynamische Modell übernommen.

**Beispiel** Abb. 10.2-2 zeigt im weißen Modell eine Lagerverwaltung. Eine Palette kann sich demnach entweder in einem offenen Lager (d.h. ein Lagerraum ohne weitere Strukturierung) oder auf einem Stellplatz befinden. Dieser Sachverhalt wird durch das weiße Modell exakt wiedergegeben (vgl. Kapitel 3.2). Nach Beseitigung der Vererbungsstrukturen ergibt sich das wesentlich einfachere graue Modell. Allerdings darf hier nur dann eine Verbindung zwischen einer Palette und einem Lager erstellt werden, wenn gilt: Lager.Art = OffenesLager.

Abb. 10.2.-2:  
Komprimieren von Vererbungsstrukturen



Während die Wiederverwendung (*reuse*) in der Systemanalyse eine untergeordnete Rolle spielt, ist sie im Entwurf von sehr großer Bedeutung. Unter diesem Aspekt ist zu berücksichtigen, daß die Kriterien für eine »gute« Vererbungsstruktur nicht so streng sind wie in der Analyse. Beispielsweise müssen im Entwurf nicht alle Attribute und Operationen einer Oberklasse auch von der Unterklasse benötigt werden.

Wieder-  
verwendung

### 1 Modifikation der Klassenstruktur

- Hinzufügen von *Container*-Klassen.
- Zerlegen komplexer Klassen.
- Zusammenfassen von Klassen mit starker Interaktion.
- Hinzufügen von Klassen zum Modellieren von Zwischenergebnissen.
- Transformation von assoziativen Klassen in »normale« Klassen.

### 2 Verfeinern der Attribute

- Abgeleitete Attribute des OOA-Modells übernehmen oder in Operationen wandeln.
- Neue abgeleitete Attribute einführen.

### 3 Verfeinern der Operationen

- Operationen präzisieren.
- Komplexe Operationen in einfachere, interne Operationen zerlegen.
- Transformieren einfacher Lebenszyklen in Algorithmen.
- Transformieren komplexer Lebenszyklen mittels Zustandsmuster.

### 4 Verfeinern von Assoziationen

- Navigationsrichtung bei allen Assoziationen eintragen.
- Zugriffspfade optimieren.
- Prüfen, welche Assoziationen unidirektional modelliert werden können.

### 5 Verfeinern der Vererbungsstruktur

- Abstrakte Operationen für einheitliche Schnittstellen hinzufügen.
- Hinzufügen von abstrakten Oberklassen.
- Maximierung des Polymorphismus.
- Komprimieren von Vererbungsstrukturen.
- Wiederverwenden existierender Klassen.

**Checkliste:**  
**Entwurf der**  
**Fachkonzept-**  
**schicht**

## 10.3 Entwurf der GUI-Schicht und Anbindung an die Fachkonzept-Klassen

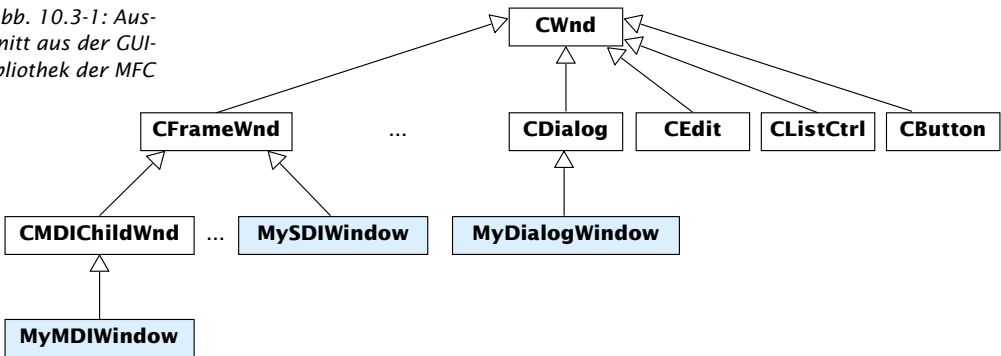
Bei Anwendungen, die überwiegend grafisch orientiert sind, wie z.B. ein Editor für grafische Objekte, fällt die genaue Abgrenzung zwischen dem Fachkonzept und der Benutzungsoberfläche oft schwer. Hier verfahren wir so, daß diese grafischen Objekte – z.B. Rechtecke, Kreise, Linien – bereits in der Analyse (Fachkonzept) modelliert werden, während Objekte wie Maus, Fenster, Auswahlbox der Entwurfsphase (GUI-Schicht) vorbehalten bleiben. Als Faustregel gilt, daß alle Objekte, die unabhängig von der verwendeten GUI dargestellt werden müssen, zum Fachkonzept gezählt werden.

Die Architektur der Benutzungsoberfläche bzw. der **GUI-Schicht** wird entscheidend durch das verwendete GUI-System geprägt.

GUI-Bibliothek Eine GUI-Bibliothek besteht meistens aus einem oder zwei größeren Bäumen. Für jedes Interaktionselement (*widget*) gibt es eine Blattklasse. Für die Dialogfenster gibt es eine Basisklasse, von der dann die individuellen Dialogfenster abgeleitet werden. Analog gibt es Basisklassen für die Ableitung von Klassen zur Realisierung von MDI-Unterfenstern (vergleiche Kapitel 5.3).

Beispiel **Abb. 10.3-1 zeigt einen Ausschnitt aus der MFC-Bibliothek (*Microsoft Foundation Class Library*). Die MFC-Bibliothek enthält außer den GUI-Klassen noch zahlreiche andere Klassen, z.B. für die Dateiverwaltung, und auch elementare Klassen. Sie bietet sowohl Klassen für eine einfache Wiederverwendung – analog zur einer Funktionsbibliothek – als auch *Frameworks*. In der Abb. 10.3-1 sind die Klassen, die der Benutzer durch Ableitung von *Framework*-Klassen bildet, blau markiert.**

Abb. 10.3-1: Ausschnitt aus der GUI-Bibliothek der MFC



elementare Klassen

Um ein einigermaßen kompaktes und gut lesbares OOD-Klassendiagramm zu erhalten, ist es wichtig, die Grenze zwischen Architekturklassen und elementaren Klassen sorgfältig zu ziehen.

Zur Erinnerung: Architekturklassen werden in das Klassendiagramm eingetragen, während elementare Klassen im Sinne von

Attributtypen behandelt werden. Als Architekturklassen der GUI-Schicht definieren wir alle Klassen, die Fenster realisieren. Dagegen betrachten wir Klassen, die Interaktionselemente implementieren, als elementare Klassen.

**Anbindung der GUI-Schicht an die Fachkonzeptschicht**

Im folgenden zeige ich anhand der Klasse `Artikel`, wie die systematische Anbindung der GUI-Klassen an die Fachkonzept-Klassen erfolgen kann. In der nächsten Lehreinheit wird dieses Beispiel für diverse Datenhaltungen weiterentwickelt.

Die hier entwickelten Entwurfsmodelle sind bewußt allgemein gehalten, um die wesentliche Struktur und nicht eine bestimmte Form der Implementierung zu zeigen. Leser, die ein Programm erstellen wollen, seien daher auf die Beispielprogramme auf der CD verwiesen.

Im einfachsten Fall existiert für eine Klasse des Fachkonzepts genau ein Erfassungsfenster, das durch eine GUI-Klasse realisiert wird. Die Verbindung zwischen dieser GUI-Klasse und der Fachkonzeptklasse wird durch eine Assoziation hergestellt. Sie wird innerhalb des GUI-Objekts durch einen Zeiger auf das Fachkonzeptobjekt realisiert.

Erfassungsfenster

Die Anbindung erfolgt nach folgenden Regeln:

- 1 Jedes Fensterobjekt ist mit seinem Fachkonzeptobjekt über einen Zeiger verbunden und kann so dessen Operationen aufrufen. In diesem Zusammenhang wird häufig der Begriff `subject` für das assoziierte Fachobjekt verwendet.
- 2 Die GUI-Klasse besitzt die Operation `update()`, die Attributwerte aus dem zugehörigen Fachkonzeptobjekt liest. Die Operation `save()` übergibt die Eingaben aus dem Fensterobjekt an das Fachkonzeptobjekt.

Abb. 10.3-2 zeigt, wie aus der OOA-Klasse `Artikel` zunächst ein Erfassungsfenster erstellt und durch eine Fensterklasse realisiert wird. Dazu leiten wir aus der Klasse `View` des verwendeten GUI-Systems die Klasse `ArtikelView` ab, die eine Assoziation zur Klasse `Artikel` besitzt. Beim Erzeugen eines neuen `ArtikelView`-Objekts wird immer ein `Artikel`-Objekt zugeordnet. Die Attribute der Klasse `ArtikelView` beschreiben die Eingabefelder des Erfassungsfensters.

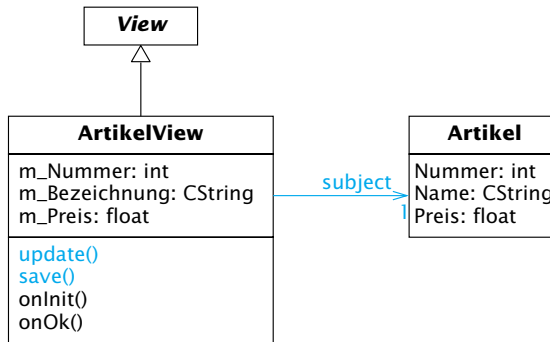
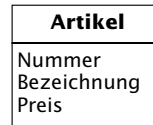
Beispiel

```
class ArtikelView: public View
{private:
    int          m_Nummer;
    CString      m_Bezeichnung;
    float        m_Preis;
    Artikel*     subject;           //Zeiger auf aktuelles Fachkonzept-
                                   //objekt
    ...
}
```

## LE 17 10 Entwurf einer Drei-Schichten-Architektur

Abb. 10.3-2: Erfassungsfenster  
Klassendiagramm  
zur Erfassung  
eines Artikels

OOA



Das Drücken der OK-Schaltfläche im Erfassungsfenster löst die Operation `save()` aus, d.h. alle Daten werden vom GUI-Objekt zum assoziierten Fachkonzeptobjekt übertragen.

```

void ArtikelView::save()
{ //precondition: ArtikelView-Objekt kennt seinen Artikel-Objekt
  subject->setNummer(m_Nummer);
  subject->setBezeichnung(m_Bezeichnung);
  subject->setPreis(m_Preis);
  ...
}
    
```

Wenn das Erfassungsfenster für einen vorhandenen Artikel geöffnet und initialisiert wird, dann werden mittels der Operation `update()` die Attributwerte des assoziierten Artikels angezeigt.

```

void ArtikelView::update()
{ //precondition: ArtikelView-Objekt kennt seinen Artikel-Objekt
  m_Nummer = subject->getNummer();
  m_Bezeichnung = subject->getBezeichnung();
  m_Preis = subject->getPreis();
  ...
}
    
```

Typkonvertierung In obigem Beispiel arbeiten die GUI-Klasse und die Fachkonzeptklasse mit den gleichen Typen. Das ist beispielsweise in der MFC-Klassenbibliothek möglich, in der die Interaktionselemente nicht nur als *Strings*, sondern auch als *int*, *float* usw. definiert werden können. Bei manchen GUI-Bibliotheken sind Interaktionselemente



grundsätzlich als *Strings* zu deklarieren. Dann ist eine Konvertierung zwischen den Datentypen notwendig. Diese Konvertierung wird meistens in der GUI-Klasse durchgeführt.

Eine andere Möglichkeit besteht darin, daß alle Eingaben – unabhängig von ihrem fachlichen Typ – als *Strings* an die Fachkonzeptklassen übergeben werden und erst dort konvertiert werden. Diese Realisierung besitzt den Vorteil, daß die Benutzungsoberfläche sehr leicht ausgetauscht werden kann.

Wir erweitern nun die Problemstellung um das Listenfenster. Wenn das Listenfenster geöffnet wird, dann müssen alle Artikel in der Liste angezeigt werden. Auch hier muß gegebenenfalls eine Typkonvertierung durchgeführt werden. Bei einer »echten« Anwendung umfaßt die Klasse `Artikel` sehr viele Attribute, die unter Umständen auf mehrere Seiten verteilt werden müssen. In der Liste werden für jeden Artikel meistens nur die wichtigsten Attribute eingetragen.

Listenfenster

Für das Listenfenster wird – analog zum Erfassungsfenster – eine eigene Klasse entworfen. Die Verwaltung der Fachkonzeptobjekte wird mittels einer geeigneten *Container*-Klasse realisiert. Für jede *Container*-Klasse existiert genau ein Objekt. Wir können daher das **Singleton-Muster** anwenden (Kapitel 7.3).

Kapitel 7.3



Abb. 10.3-3 zeigt, wie aus der OOA-Klasse `Artikel` zunächst ein Listenfenster abgeleitet wird. Das Attribut `m_Liste` realisiert das Listenelement (*list view control*). Die Fachkonzeptklasse `Artikel` wird um die *Container*-Klasse `ArtikelListe` ergänzt, die alle Artikel verwaltet. Das Klassenattribut `uniqueInstance` enthält den Zeiger auf das einzige Objekt dieser Klasse. Mit der Klassenoperation `instance()` kann auf diese Adresse überall zugegriffen werden. Beim ersten Aufruf von `instance()` wird zusätzlich die leere Artikel-liste erzeugt. Der Aufruf der *instance*-Klassenoperation erfolgt im Konstruktor und die erhaltene Referenz wird als Objektverbindung `all` festgehalten.

Beispiel

```
class ArtikelListeView: public View
{
    CListCtrl m_Liste;
    ArtikelListe* all;
    ...
}
```

Beim Öffnen des Listenfensters wird ein neues Objekt der Klasse `ArtikelListeView` erzeugt und alle Artikel werden angezeigt. Dazu holt sich die Operation `update()` aus der `Artikelliste` die Adressen aller Artikel und kann dann für jedes Artikelobjekt dessen benötigte Attributwerte lesen und darstellen.

Die `Artikelliste` wird mittels einer *for*-Scheife traversiert. Die Operation `getArtikel()` gibt für jede gültige Listenposition *i* die Adresse des Artikels im Parameter `pArtikel` zurück. Das Ende der Artikel-

zur Implementierung

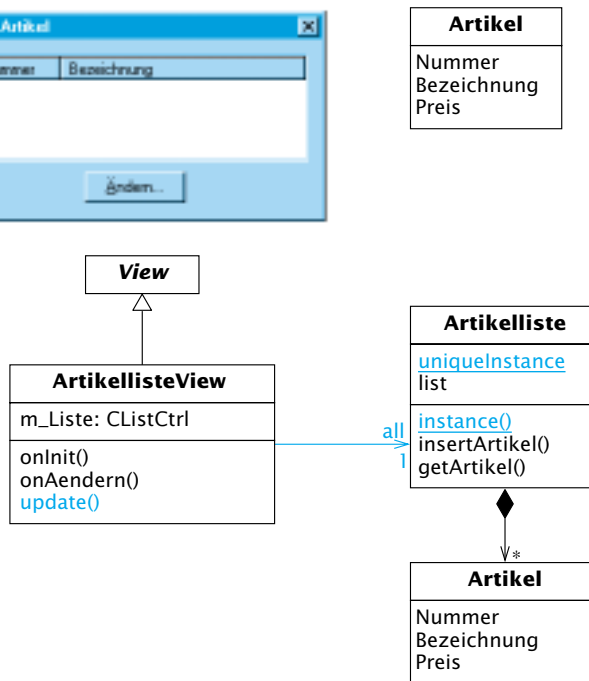
## LE 17 10 Entwurf einer Drei-Schichten-Architektur

Abb. 10.3-3:  
Klassendiagramm  
für die Liste  
aller Artikel

Listenfenster



OOA



liste zeigt die Operation durch einen Ergebniswert ungleich 0 an. Die Operation `getNumber()` liest die aktuelle Artikelnummer, konvertiert die Zahl in einen *String* und trägt diesen mit `InsertItem()` in die entsprechende Spalte im *list view control* ein.

```

void ArtikelListeView::update()
{ //Inhalt des list view control neu füllen
  Artikel * pArtikel = new Artikel ;
  CString strNumber;

  m_Liste.DeleteAllItems();
  for (int i = 0; all->getArtikel(i, pArtikel) == 0; i++)
  { strNumber.Format("%d", pArtikel->getNumber());
    m_Artikelliste.InsertItem(i, strNumber);
    ...
  }
}

```

Szenario  
Erfassen eines  
Artikels

Wenn im Erfassungsfenster ein neuer Artikel erfasst und die OK-Schaltfläche betätigt wird, so läuft das in Abb. 10.3-4 dargestellte Szenario ab. Die Operation trägt zunächst mit `save()` alle Attributwerte in das erzeugte Fachkonzept-Objekt ein. Anschließend fügt sie dieses Objekt mit `insertArtikel()` in die Liste aller Artikel ein.

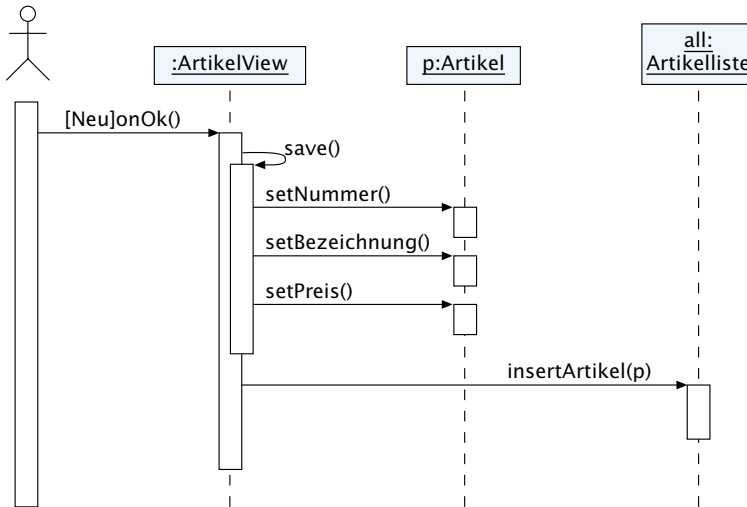


Abb. 10.3-4:  
Sequenzdiagramm  
zum Erfassen eines  
neuen Artikels

Abb. 10.3-5 beschreibt das Szenario zum Ändern eines vorhandenen Artikels. Der Benutzer öffnet ein Listenfenster. Das bedeutet, daß ein neues Objekt der Klasse `ArtikelListView` mit der Operation `new()` erzeugt und anschließend mit `onInit()` mit der `Artikelliste` initialisiert wird. Wenn der Benutzer einen Artikel selektiert hat, betätigt er die Ändern-Schaltfläche. Nun soll sich das Erfassungsfenster mit den aktuellen Daten des selektierten Artikels öffnen. Dazu wird mittels `getArtikel()` die Adresse des selektierten Artikels ermittelt, damit dessen Attributwerte ausgelesen werden können.

Szenario  
Ändern eines  
Artikels

Das Öffnen des Erfassungsfensters wird durch das Erzeugen eines entsprechenden Objekts realisiert, das anschließend initialisiert wird. Mit Auslösen der OK-Schaltfläche werden die geänderten Daten übernommen. Dazu wird die Operation `save()` aufgerufen. Wie sich hier deutlich zeigt, ist das Wissen, wie die Attributwerte gelesen und geschrieben werden, ausschließlich in den `save-` und `update-`Operationen verborgen.

Es können ein oder mehrere Listenfenster geöffnet sein, wenn im Erfassungsfenster ein neuer Artikel eingetragen wird. Um alle Daten konsistent anzuzeigen, sollen alle geöffneten Listenfenster automatisch aktualisiert werden. Für die Lösung dieser Problemstellung verwenden wir das **Beobachter-Muster** (Kapitel 7.7), wobei die Fachkonzeptklasse dem Subjekt und die Listenfenster-Klassen den Beobachtern entsprechen.

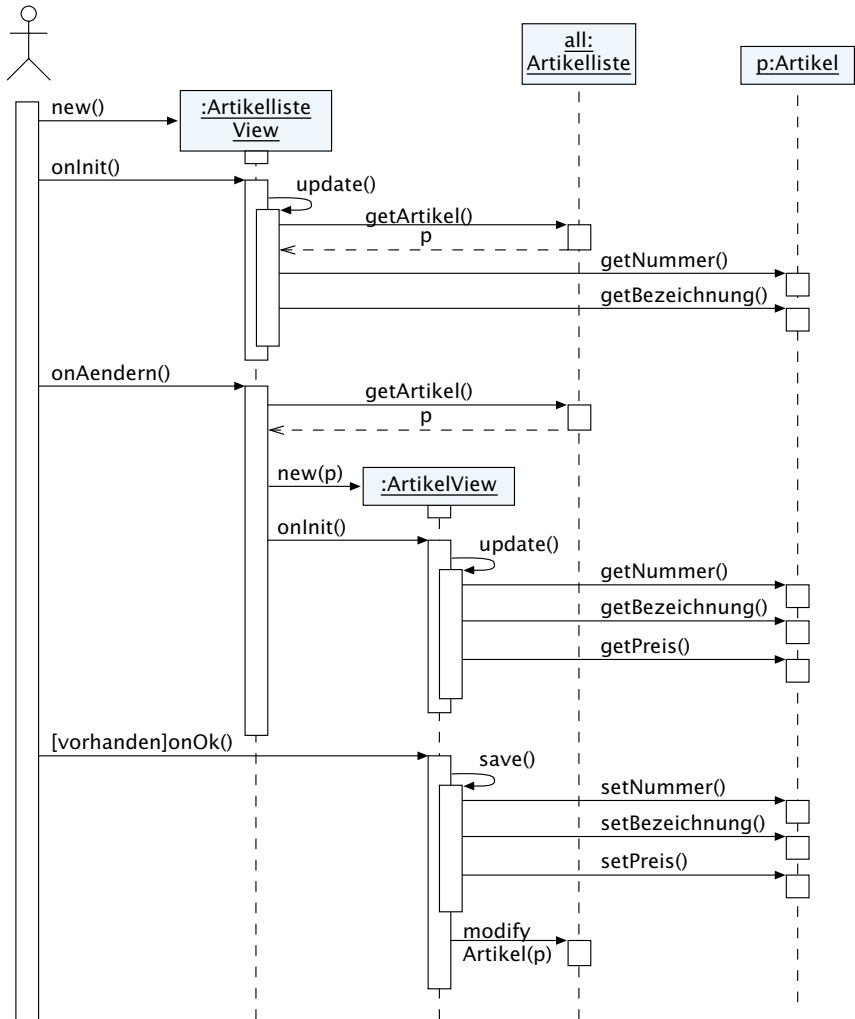
Aktualisierung  
der Listenfenster

Kapitel 7.7

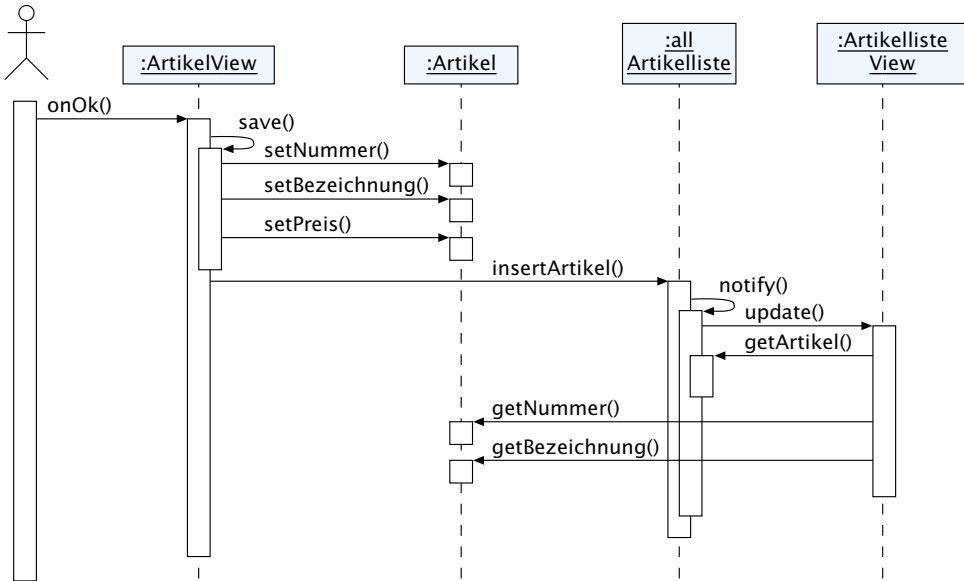


## LE 17 10 Entwurf einer Drei-Schichten-Architektur

Abb. 10.3-5:  
Sequenzdiagramm  
zum Ändern  
eines Artikels



Beispiel Abb. 10.3-6 zeigt das Szenario, das nach dem Betätigen der OK-Schaltfläche im Erfassungsfenster abläuft. Die Operation `save()` überträgt die Eingabefelder an das Fachkonzeptobjekt `Artikel`. Wurde ein neuer Artikel erfasst, dann muß das `Artikel`-Objekt mit `insert Artikel()` in die `Artikelliste` eingetragen werden. Das `ArtikelView`-Objekt kann auf die `Artikelliste` über seine Objektverbindung `all` zugreifen. Diese Verbindung wird analog zur `Listensterkerklasse` mittels `Singleton`-Muster aufgebaut. Das Fachkonzeptobjekt `Artikelliste` informiert alle `Listenster` – die `Beobachter` – mittels `notify()` darüber, daß eine Veränderung stattgefunden hat. Die `notify`-Operation sendet die Botschaft `update()` an alle `Listenster`, die sich daraufhin selbst aktualisieren, d.h. sie beschaffen sich mittels `getArtikel()` das betreffende `Artikel`-Objekt und holen sich an-



schließlich mittels der *get*-Operationen die benötigten Attributwerte. Abb. 10.3-7 zeigt für das beschriebene Szenario den zugehörigen Ausschnitt aus dem Klassendiagramm, aus dem zu ersehen ist, welche Objekte sich kennen. Die Operation *attach()* baut eine Verbindung zu einem Beobachter auf, die Operation *detach()* löst diese Verbindung wieder.

Abb. 10.3-6: Szenario zum Aktualisieren aller Listenfenster mittels Beobachter-Muster

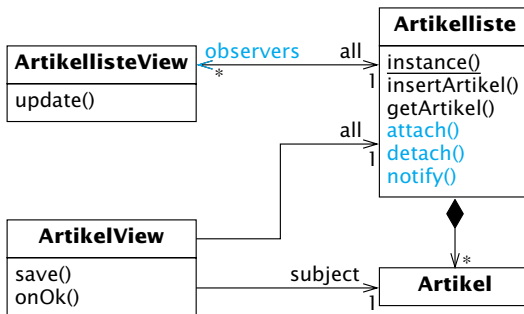


Abb. 10.3-7: Klassendiagramm zum Beobachter-Muster

Abb. 10.3-8 zeigt das vollständige Klassendiagramm für den Entwurf der GUI-Klassen und ihre Anbindung an die Fachkonzept-Klassen *Artikel* und *ArtikelListe*. Auf der CD befindet sich eine in C++ erstellte Implementierung, die der Einfachheit halber von einer temporären Datenhaltung im Arbeitsspeicher ausgeht.

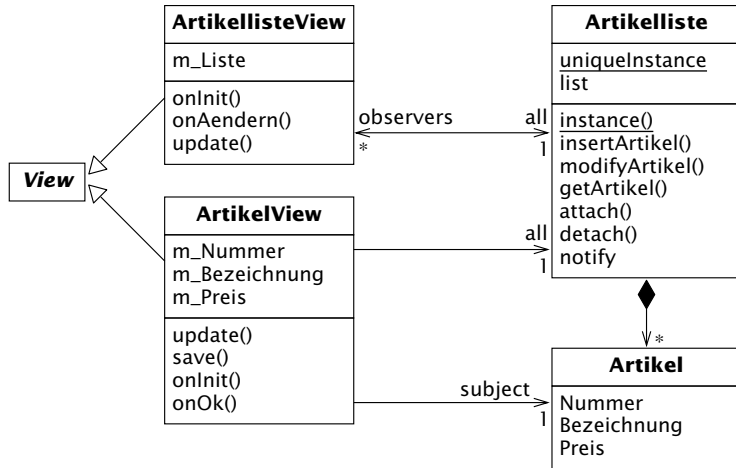
ArtikelGUI-FK

Bei einer umfangreichen Anwendung kann es sinnvoll sein, die GUI- und die Fachkonzeptschicht durch eine Zugriffsschicht zu trennen. Das kann beispielsweise durch eine **Fassade** geschehen.

Fassade

## LE 17 10 Entwurf einer Drei-Schichten-Architektur

Abb. 10.3-8:  
Vollständiges  
Klassendiagramm  
für die Anbindung  
der GUI-Schicht an  
die Fachkonzept-  
schicht



Das ist eine Klasse, die zwischen der GUI-Schicht und der Fachkonzeptschicht vermittelt. Dann kennt die GUI-Klasse die Typen der Fachkonzeptklasse *nicht*. Auf diese Weise ist die Fachkonzeptschicht vollständig von der GUI-Schicht isoliert.

Eine Fassade hat folgende Aufgaben:

- Sie reagiert auf Eingaben und andere Ereignisse.
- Sie öffnet Fenster, die Informationen der Fachkonzeptobjekte darstellen.
- Sie verwaltet Transaktionen, z.B. *commit* und *rollback*.

Wenn die Informationen der Anwendung in verschiedenen Fenstern angezeigt werden sollen (*dependent windows*), dann führt die Fassade noch weitere Aufgaben durch:

- Sie ermöglicht es, in mehreren Fenstern gleichzeitig die Informationen eines Fachkonzeptobjekts anzuzeigen.
- Sie informiert abhängige Fenster, wenn sich die Information ihres Fachkonzeptobjekts geändert hat und eine Aktualisierung der Darstellung nötig ist.

/Larman 98/ spricht von einem *application coordinator*, /Fowler 97a/ verwendet den Begriff *facade* (vergleiche Fassaden-Muster in Kapitel 7.6). Der Begriff »Fassade« macht deutlich, daß diese Klasse das komplexe Fachkonzeptobjekt für das GUI-Objekt vereinfacht. ↕

Kapitel 7.6

Der Nachteil dieser Modellierung ist offensichtlich: Es sind zusätzliche Aufrufe notwendig. Dem stehen jedoch eine Reihe von Vorteilen gegenüber:

- Die getrennte Entwicklung von GUI-Schicht und Fachkonzeptschicht durch unterschiedliche Teams wird vereinfacht.
- Es ist einfacher, mehrere Alternativen der GUI-Schicht zu entwickeln.
- Für die Fassaden kann eine spezielle Testschnittstelle entwickelt werden, die beispielsweise einen automatischen Test ermöglicht.

Handelt es sich um eine Client-Server-Anwendung und liegt die Fachkonzeptschicht auf dem Server, dann wird die Fassade auf *Client* und Server aufgeteilt. Die GUI-Schicht kommuniziert mit der *Client*-Fassade. Diese reicht die Botschaft an die Server-Fassade weiter, die dann wiederum mit der Fachkonzeptschicht kommuniziert.

- GUI-System auswählen.
- Jedes GUI-Fenster durch eine Architektur-Klasse realisieren.
- Interaktionselemente durch elementare Klassen realisieren.
- GUI-Klasse für Erfassungsfenster enthält
  - eine *one*-Assoziation zum Subjekt (*subject*),
  - eine *one*-Assoziation zur *Container*-Klasse (*all*),
  - die Operation *save()* zum Übergeben der Werte an die Fachkonzeptklasse,
  - die Operation *update()* zum Anzeigen der fachlichen Werte im Erfassungsfenster.
- GUI-Klasse für Listenfenster enthält
  - eine *one*-Assoziation zur *Container*-Klasse (*all*),
  - die Operation *update()* zum Anzeigen aller Objekte dieser Klasse,
- *Container*-Klasse (Fachkonzeptklasse), von der es nur ein Exemplar gibt (*Singleton*-Muster), enthält
  - das Klassenattribut *uniqueInstance*, welches die Referenz auf das einzige Objekt enthält,
  - die Klassenoperation *instance*, die auf diese Referenz zugreifen kann und beim ersten Aufruf das Objekt erzeugt.
- *Container*-Klasse informiert ihre GUI-Listenklassen (Beobachter) mittels Beobachter-Muster und enthält
  - eine *many*-Assoziation zur GUI-Listenklasse (*observers*),
  - die Operation *attach()*, die eine Verbindung zu einem Beobachter-Objekt aufbaut,
  - die Operation *detach()*, die eine Verbindung zu einem Beobachter-Objekt abbaut,
  - die Operation *notify()*, die alle Beobachter über eine Veränderung benachrichtigt.
- Stärkere Entkopplung von GUI- und Fachkonzeptschicht durch Fassadenklassen.

**Checkliste:**  
**Entwurf der GUI-Schicht und Anbindung an die Fachkonzeptschicht**

**Beobachter-Muster (*observer pattern*)** Das Beobachter-Muster ist ein objektbasiertes Verhaltensmuster. Es sorgt dafür, daß bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

**Datenhaltungsschicht (*storage tier, database tier*)** Die Datenhaltungsschicht realisiert die jeweilige Form der Datenspeicherung, z.B. mit einem objektorientierten oder relationalen Datenbanksystem oder mit flachen Dateien.

**Drei-Schichten-Architektur (*three-tier architecture*)** Die Drei-Schichten-Architektur besteht aus der →GUI-Schicht (Schicht der Benutzungsoberfläche), der →Fachkonzeptschicht und der →Datenhaltungsschicht. Es sind zwei Ausprägungen möglich: die →strenge und die →flexible Drei-Schichten-Architektur.

**Fachkonzeptschicht (*application logic tier*)** Die Fachkonzeptschicht modelliert in einer →Drei-Schichten-Architektur die fachliche Anwendung und die Zugriffe auf die →Datenhaltungsschicht. Das OOA-Modell bildet die erste Version der Fachkonzeptschicht.

**Flexible Drei-Schichten-Architektur** Eine flexible Drei-Schichten-Architektur ergibt sich, wenn die →GUI-Schicht sowohl auf die →Fachkonzeptschicht als auch auf die →Datenhaltungsschicht zugreifen darf.

**GUI-Schicht (*presentation tier*)** Die GUI-Schicht ist in einer →Drei-Schichten-Architektur sowohl für die Dialogführung und die Präsentation der fachlichen Daten (z.B. in Fenstern) als auch für die Kommunikation mit der →Fachkonzeptschicht und ggf. mit der →Datenhaltungsschicht zuständig.

**GUI-System (*GUI system*)** Das GUI-System ist ein Softwaresystem, das die graphische Oberfläche verwaltet und

die Kommunikation mit den Anwendungen abwickelt. Ein GUI-System wird vereinfachend auch Fenstersystem genannt.

**Mehr-Schichten-Architektur (*multi-tier architecture*)** Eine Mehr-Schichten-Architektur entsteht, wenn die →Drei-Schichten-Architektur um weitere Schichten erweitert wird bzw. die vorhandenen Schichten feiner zerlegt werden.

**MVC (*Model/View/Controller*)** MVC besteht aus den drei Klassen *Model*, *View* und *Controller*. Das *Model*-Objekt repräsentiert das Fachkonzeptobjekt. Oft gibt es mehrere Möglichkeiten, die fachlichen Daten zu präsentieren. Für jede Präsentation gibt es ein *View*-Objekt. Das *Controller*-Objekt bestimmt, wie die Benutzungsoberfläche auf Eingaben reagiert. Jedes *View*-Objekt besitzt ein zugehöriges *Controller*-Objekt, das diese Darstellung mit der Eingabe verbindet. Das impliziert, daß es zu jedem *Model*-Objekt eine beliebige Anzahl von Paaren (*View*, *Controller*) geben kann, jedoch mindestens eines.

**Singleton-Muster (*singleton pattern*)** Das *Singleton*-Muster ist ein objektbasiertes Erzeugungsmuster. Es stellt sicher, daß eine Klasse genau ein Objekt besitzt und ermöglicht einen globalen Zugriff auf dieses Objekt.

**Strenge Drei-Schichten-Architektur** Bei einer strengen Drei-Schichten-Architektur kann die →GUI-Schicht nur auf die →Fachkonzeptschicht und letztere nur auf die →Datenhaltungsschicht zugreifen.

**Zwei-Schichten-Architektur (*two-tier architecture*)** Bei einer Zwei-Schichten-Architektur sind die Benutzungsoberfläche und das Fachkonzept fest in einer Schicht verzahnt. Die zweite Schicht realisiert die Datenhaltung.



Wichtige Entwurfsentscheidungen sind die Wahl der Programmiersprache, des **GUI-Systems** und einer geeigneten Datenhaltung. Ein System kann als **Zwei-Schichten**-, **Drei-Schichten**- oder **Mehr-Schichten-Architektur** entworfen werden. Aus dem OOA-Modell entsteht durch Verfeinerung die **Fachkonzeptschicht**. Die Benutzungsoberfläche wird durch die **GUI-Schicht** realisiert, die systematisch an die Fachkonzeptschicht angebunden wird.





- 1 Lernziel: Entwurfsarchitekturen unterscheiden können.**  
 Erläutern Sie die Eigenschaften der folgenden Schichten:
- Zwei-Schichten-Architektur.
  - Drei-Schichten-Architektur.
  - Mehr-Schichten-Architektur.

Aufgabe  
5-10 Minuten

- 2 Lernziel: MVC-Architektur und Beobachter-Muster unterscheiden können.**  
 Erklären Sie Gemeinsamkeiten und Unterschiede von MVC-Architektur und Beobachter-Muster.

Aufgabe  
5 Minuten

- 3 Lernziel: GUI-Schicht an die Fachkonzeptschicht anbinden können.**  
 Entwerfen Sie für die Klasse Person und die abgebildeten Fenster (Abb. LE17-A3) die GUI-Klassen und verbinden Sie diese mit den Fachkonzeptklassen. Beschreiben Sie ein Szenario (Sequenzdiagramm), in dem eine vorhandene Person geändert und gleichzeitig alle geöffneten Listenfenster automatisch aktualisiert werden. Tragen Sie alle Operationen aus dem Szenario in das Klassendiagramm ein. Erstellen Sie zusätzlich ein Objektdiagramm mit zwei Objekten der Klasse Person und zwei geöffneten Listenfenstern, aus dem die Verbindungen der beteiligten Objekte hervorgehen.

Aufgabe  
35 Minuten

Hinweis: Diese Aufgabe wird in der nächsten Lehrinheit weiterentwickelt.

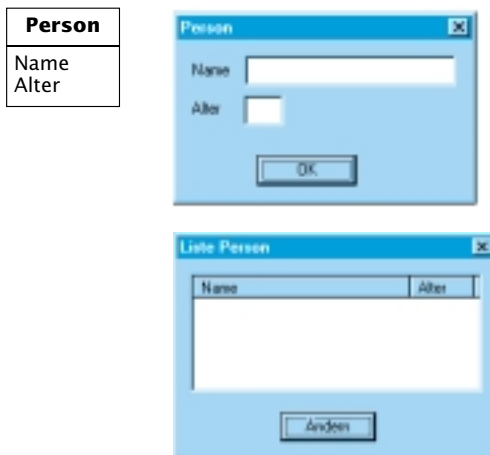


Abb. Le17-A3:  
Klasse Person mit Fenstern

## 10 Erstellen eines Entwurfsmodells mittels Drei-Schichten-Architektur (Teil 2)



- Erklären können, wie ein objektorientiertes Datenbanksystem angebunden wird.
- Erklären können, wie eine objektorientierte Anwendung mit einer einfachen Dateiverwaltung realisiert wird.
- Erklären können, wie ein relationales Datenbanksystem angebunden wird.
- Erklären können, wie ein *Framework* verwendet wird.
- Eine Drei-Schichten-Architektur mit Anbindung an ein objektorientiertes Datenbanksystem entwerfen können.

verstehen

anwenden



- Voraussetzungen für diese Lehreinheit sind die objektorientierten Konzepte der Analyse und des Entwurfs sowie der UML-Notation, wie sie in den Kapiteln 2 und 6 beschrieben sind.
- Außerdem sollten die Kapitel 10.1 bis 10.3 bekannt sein.
- Grundlagen in C++ oder Java erleichtern das Verstehen dieser Lehreinheit.



- 10.4 Entwurf der Datenhaltung mit einem objektorientierten Datenbanksystem 396
- 10.5 Entwurf der Datenhaltung mit flachen Dateien 401
- 10.6 Entwurf der Datenhaltung mit einem relationalen Datenbanksystem 405

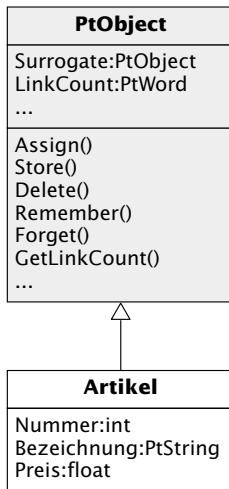
## 10.4 Entwurf der Datenhaltung mit einem objektorientierten Datenbanksystem

Wird ein **objektorientiertes Datenbanksystem** verwendet, dann ist dessen Anbindung an die Anwendung relativ einfach. Wir betrachten hier exemplarisch das objektorientierte Datenbanksystem Poet /Poet 97/. Der Programmierer macht die Objekte einer Klasse **persistent** (dauerhaft), indem er die Klasse mit `persistent` kennzeichnet. Der Präprozessor der Datenbank macht aus dieser Klasse dann eine Unterklasse von `d_Object`, die von der objektorientierten Datenbank bereitgestellt wird.

**Beispiel** Die Klasse `Artikel` wird folgendermaßen persistent:

```
//Artikel.hcd
persistent class Artikel
{protected:
    int Nummer;
    PtString Bezeichnung;
    float Preis;
    ...
};
```

Abb. 10.4-1:  
Transformation  
einer persistenten  
Klasse durch den  
Poet-Präprozessor



Aus dieser Schema-Deklaration generiert der Poet-Präprozessor unter anderem folgende C++-Datei, die dann vom Compiler übersetzt wird:

```
//Artikel.hxx
...
class Artikel: public PtObject
{...};
```

Von der Klasse `PtObject` (alias `d_Object`) erbt die Klasse `Artikel` unter anderem die Objektidentität (OID, *surrogate*) und die Fähigkeit, sich selbst in der Datenbank zu verwalten (Abb. 10.4-1).

**Typkonvertierung** Objektorientierte Datenbanken verwenden für die zu speichernden Attribute sowohl die Typen der Programmiersprache als auch ODMG-konforme oder eigene Typen. Die Benutzungsoberfläche verwendet die Typen der entsprechenden Klassenbibliothek. Daher ist in vielen Fällen eine Typkonvertierung erforderlich.

**Beispiel** Poet verwendet für Zeichenketten den Typ `PtString`. Die Konvertierung erfolgt bei diesem Beispiel innerhalb der *get-* und *set-*Operationen. Dann können die *save-* und *update-*Operationen der Klasse

ArtikelView aus Kapitel 10.3 der Lehreinheit 17 unverändert übernommen werden.

```

persistent class Artikel
{protected:
    int Nummer;
    PtString Bezeichnung;
    float Preis;
public:
    void setNummer (int n);
    int getNummer() const;
    void setBezeichnung (const CString &b);
    CString getBezeichnung () const;
    void setPreis (float p);
    float getPreis() const;
};
    
```

Die *set*- und *get*-Operationen werden folgendermaßen implementiert:

```

void Artikel::setNummer(int n)
{ Nummer = n; }
int Artikel::getNummer() const
{ return Nummer; }

void Artikel::setBezeichnung(const CString& b)
{ Bezeichnung = b; }
CString Artikel::getBezeichnung() const
{ return Bezeichnung; }

void Artikel::setPreis(float p)
{ Preis = p; }
float Artikel::getPreis() const
{ return Preis; }
    
```

Bei der **strengen Drei-Schichten-Architektur** darf die GUI-Klasse nur auf Operationen der Fachkonzeptklassen und jede Fachkonzeptklasse nur auf Operationen der Datenhaltung zugreifen.

strengere Drei-Schichten-Architektur

Wir erweitern das in Kapitel 10.3 eingeführte OOD-Modell. Abb. 10.4-2 zeigt, daß die ursprüngliche Klassenstruktur im wesentlichen beibehalten wird. Zusätzlich erbt die Klasse Artikel die persistenten Eigenschaften und es wird die von Poet generierte Klassenextension ArtikelAllSet verwendet. Die Assoziation zwischen ArtikelListe und Artikel wird *nicht* realisiert, weil bei Verwendung von Poet die entsprechenden Zugriffe auf Artikelobjekte mittels ArtikelAllSet erfolgen. Zwischen den Objekten von ArtikelListe und ArtikelAllSet besteht nur eine temporäre Beziehung für die Dauer eines Operationsaufrufs, die *nicht* ins Klassendiagramm eingetragen wird. Zur Erinnerung: Beim Erstellen eines Kollaborationsdiagramms wird diese temporäre Objektverbindung eingetragen und mit «temp» gekennzeichnet.

Beispiel

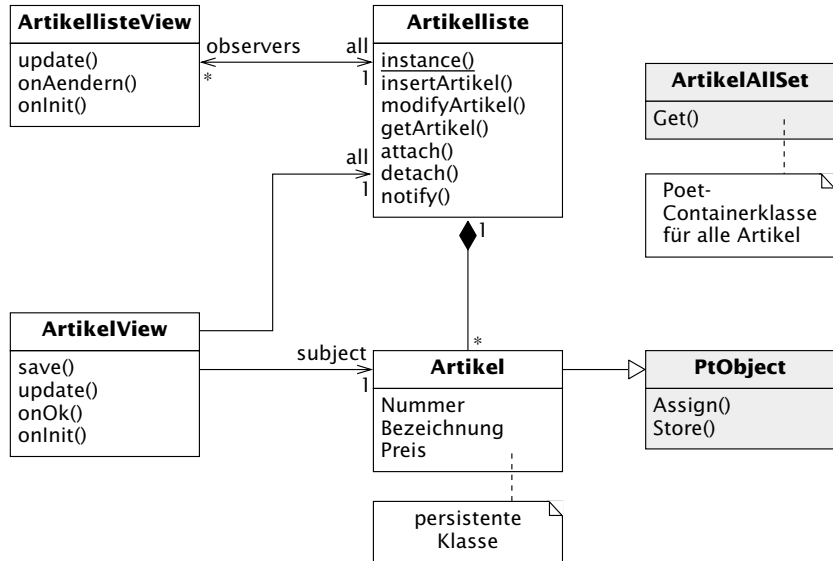


Auf der CD befindet sich die in C++ erstellte Implementierung dieses Beispiels.

ArtikelGUI-FK-Poet

## LE 18 10 Entwurf einer Drei-Schichten-Architektur

Abb. 10.4-2:  
Klassendiagramm  
für die strenge  
Drei-Schichten-  
Architektur  
mit Poet



Szenario  
Speichern eines  
neuen Artikels

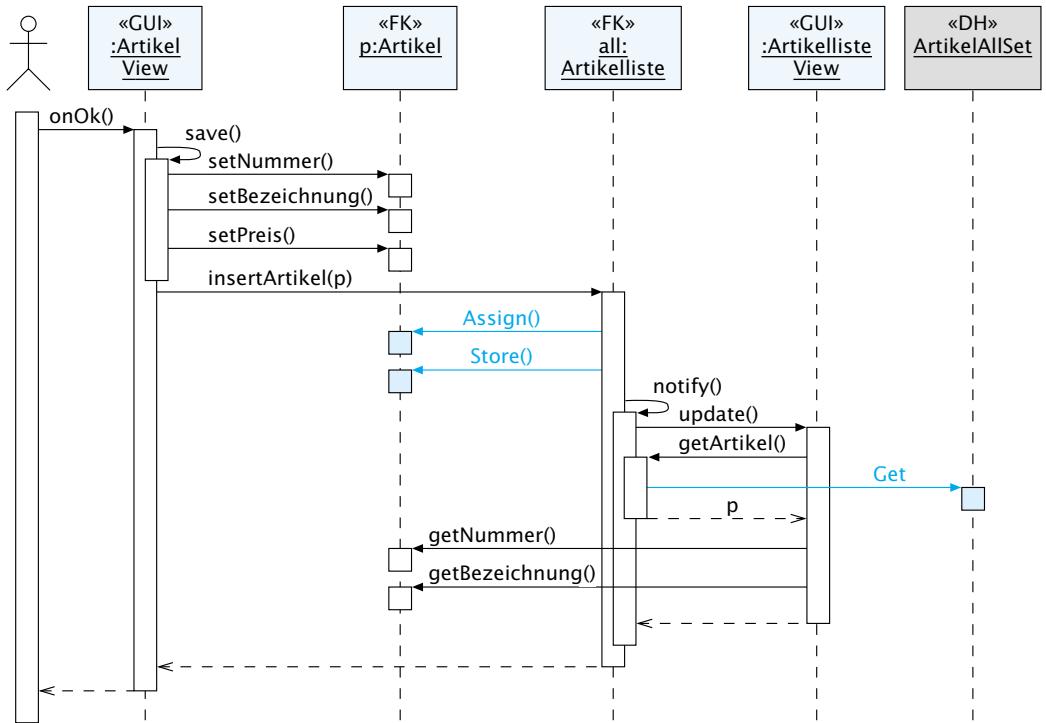
Wir betrachten als Beispiel das Szenario zum Speichern eines neuen Artikels mit Aktualisierung aller Listenfenster, das in der Abb. 10.4-3 als Sequenzdiagramm modelliert ist. Alle Poet-Operationen sind – wie bei den nachfolgenden Diagrammen – blau eingetragen.

Diese Modellierung realisiert die strenge Drei-Schichten-Architektur, weil von den GUI-Klassen aus keine direkten Zugriffe auf die Poet-Datenbank erfolgen. Um die strenge Drei-Schichten-Architektur einzuhalten, muß die Artelliste alle Zugriffe auf die Klassenextension ArtikelAllSet und die Operationsaufrufe Assign() und Store() verkapseln. Damit kann die gewählte Form der Datenhaltung relativ problemlos gegen eine andere ausgetauscht werden. Die Operation insertArtikel() speichert ein Artikel-Objekt in der Datenbank.

```
void Artelliste::insertArtikel(Artikel * pArtikel)
{ pArtikel ->Assign (theBase);
  pArtikel ->Store();
}
```

Die Operation getArtikel() liest einen Artikel aus der Datenbank und lädt ihn in den Arbeitsspeicher. Sie verwendet die Poet-Operation Get(), die als Ergebnisparameter den Wert Null zurückgibt, wenn an der angegebenen Position ein Objekt gelesen wird und der Ausgabeparameter pArtikel einen definierten Wert besitzt.

```
int Artelliste::getArtikel (int position, Artikel * &pArtikel)
{ ArtikelAllSet allArtikel (theBase); //Menge aller Artikel
  ...
  int result = allArtikel.Get (pArtikel, position, PtSTART);
  return result;
}
```



Um die grundsätzliche Vorgehensweise möglichst übersichtlich zu dokumentieren, wurde hier kein Wert auf einen möglichst effizienten Einsatz – z.B. Minimierung der zu ladenden Objekte – von Poet gelegt.

Bei der **flexiblen Drei-Schichten-Architektur** sind zwar GUI-, Fachkonzept- und Datenhaltungsklassen strikt getrennt, doch aus den GUI-Klassen heraus sind direkte Zugriffe auf die Datenhaltungsklassen möglich. Wenn die Daten im Erfassungsfenster eingegeben wurden, erfolgt von der Erfassungsklasse ein schreibender Zugriff auf die Datenbank. Der Vorteil dieses Lösungsansatzes liegt in seiner Einfachheit und dem Einsparen der Klasse `ArtikelListe` sowie entsprechender Operationsaufrufe. Nachteilig wirken sich aus, daß nun keine Beobachter-Liste für das Objekt `all: ArtikelListe` geführt wird und daß bei Austausch der Datenhaltungsschicht auch die GUI-Schicht davon betroffen ist. Dieser Lösungsansatz ist beispielsweise dann sinnvoll, wenn die geöffneten Listenfenster *nicht* automatisch aktualisiert werden sollen.

Abb. 10.4-3: Speichern eines neuen Artikels mit Poet und Aktualisierung der Listenfenster (strenge Drei-Schichten-Architektur)

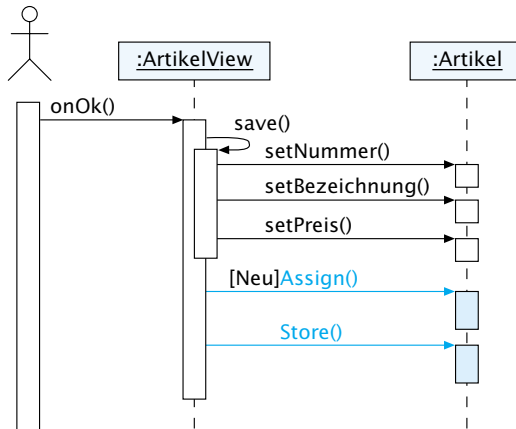
flexible Drei-Schichten-Architektur

Abb. 10.4-4 modelliert in einem Sequenzdiagramm das Erfassen eines Artikels mit der Operation `onOk()`. Die Attributwerte werden wie bisher mit der `save`-Operation an das Fachkonzept-Objekt übergeben. Anschließend ruft die Klasse `ArtikelView` die Datenbank-

Beispiel

## LE 18 10 Entwurf einer Drei-Schichten-Architektur

Abb. 10.4-4:  
Speichern eines  
Artikels mit Poet  
(flexible Drei-  
Schichten-  
Architektur)



funktionen zum Speichern auf. Analog erfolgt von der Listenfenster-Klasse ein lesender Zugriff auf die Klassenextension ArtikelAllSet, um alle Objekte der Klasse in den Speicher zu laden und im Listenfenster anzuzeigen.

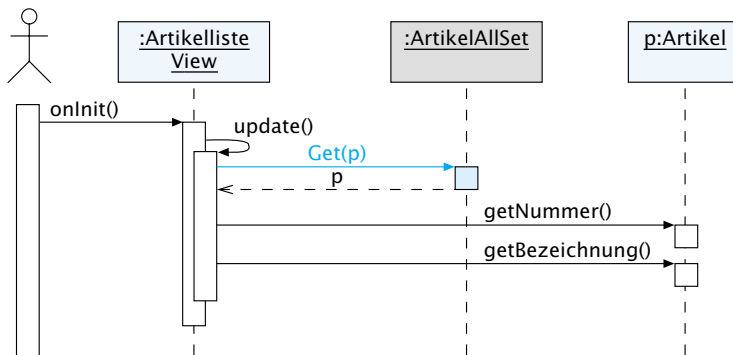
Abb.10.4-5 zeigt das Sequenzdiagramm für das Öffnen des Listenfensers mit den aktuellen Artikeln. Abb. 10.4-6 zeigt, wie das Klassendiagramm für die Realisierung der flexiblen Drei-Schichten-Architektur geändert werden muß.

ArtikelGUI-Poet

Auf der CD befindet sich die in C++ erstellte Implementierung dieses Beispiels.



Abb. 10.4-5:  
Initialisieren des  
Listenfensers mit  
Poet (flexible Drei-  
Schichten-  
Architektur)



Assoziationen

Assoziationen können in objektorientierten Datenbanksystemen direkt realisiert werden. Poet bietet außer den einfachen Zeigern die Realisierung der Assoziationen mittels *ondemand*. Bei der Verwendung von einfachen Zeigern lädt Poet ein referenziertes Objekt in den Arbeitsspeicher und trägt seine Speicheradresse in der Zeigervariablen ein. Bei dieser Realisierung wird nicht nur das jeweilige Objekt, sondern alle mit ihm assoziierten Objekte werden ebenfalls geladen. Dieser Vorgang benötigt unter Umständen sehr viel Zeit

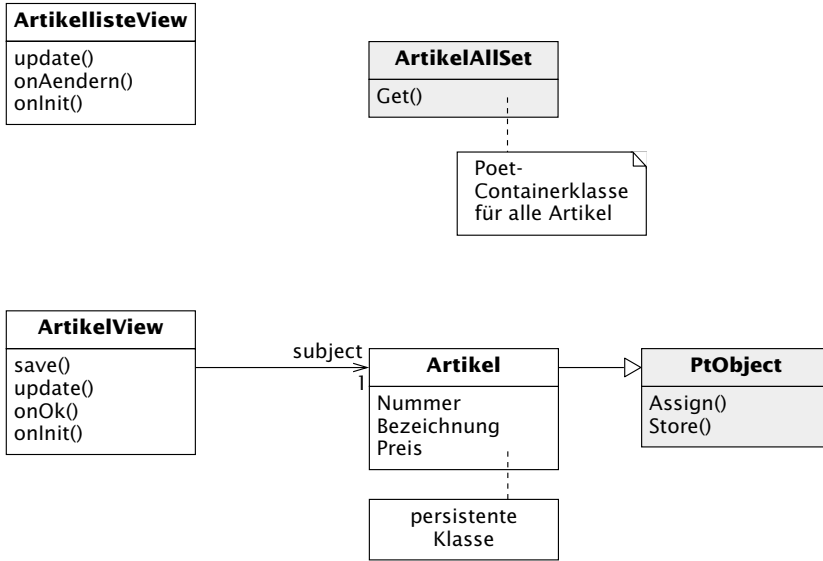


Abb. 10.4-6: Klassendiagramm für flexible Drei-Schichten-Architektur mit Poet

und viele Speicherressourcen. Daher ist er in vielen Fällen nicht erwünscht. Mittels *ondemand* kann dieser Automatismus ausgeschaltet werden. Dann werden die referenzierten Objekte nur dann in den Speicher geladen, wenn sie wirklich benötigt werden.

```

//ondemand-Assoziati on
persistent class Artikel
{ protected:
    ondemand <Lieferant> wirdGeliefertVon; //one-Assoziati on
    ...
};
persistent class Lieferant
{protected:
    cset <ondemand <Artikel>> liefert; //many-Assoziati on
    ...
};
    
```

Beispiel

## 10.5 Entwurf der Datenhaltung mit flachen Dateien

Auch die klassischen Verfahren zur Organisation flacher Dateien können zur Datenhaltung in objektorientierten Anwendungen verwendet werden. Unter **flachen Dateien** (*flat files*) sind Dateien zu verstehen, die nur rudimentäre Zugriffsoperationen anbieten. Grundlegende Verfahren sind /Hansen 96/:

- sequentielle Organisation,
- indexsequentielle Organisation,
- indizierte Organisation und
- direkte Organisation (*hash*-Verfahren).



## LE 18 10 Entwurf einer Drei-Schichten-Architektur

indizierte Organisation

Wir betrachten exemplarisch die indizierte Organisation (Abb. 10.5-1). Der Index ist entweder physisch (z.B. Array) oder logisch sortiert (z.B. Liste, Baum). Jeder Indexeintrag besteht aus einem Zugriffsschlüssel und der Adresse des Satzes, in dem die zugehörigen Daten in der Datei gespeichert sind (Abb. 10.5-1). Um einen schnellen Zugriff zu ermöglichen, sind die Schlüssel im Index sortiert. In der Datei stehen dagegen alle Sätze in der Reihenfolge, in der sie erfaßt wurden.

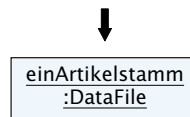
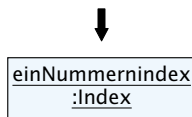
Abb.10.5-1:  
Indizierte Organisation für die Klasse Artikel

Array als Index

Nummer	Adresse
4	40
7	0
12	80
22	160
43	120

Datei

	Nummer	Bezeichnung	Preis
0	7		
40	4		
80	12		
120	43		
160	22		



Weil jede Dateiverwaltung – unabhängig vom Inhalt der verwalteten Daten – prinzipiell die gleichen Operationen benötigt, ist es sinnvoll, generische Klassen zu verwenden.

**Beispiel** Wir entwerfen eine indizierte Dateiverwaltung für die Klasse Artikel. Dazu wird ein Index benötigt, in dem die Artikel nach Nummern sortiert abgelegt sind und eine Datei, in der alle Artikel sequentiell eingetragen werden. Die generische Klasse Index verwaltet die Indextabelle, in der die Suchbegriffe – in diesem Fall die Artikelnummern – und die Adressen der zugehörigen Sätze stehen. Sie wird parametrisiert mit dem Schlüsseltyp und der maximalen Größe des Index. Es entsteht die Klasse Nummernindex. Die generische Klasse DataFile stellt die Funktionalität der Datei zur Verfügung. Aus ihr wird – zum Speichern aller Artikelattribute – die Klasse Artikelstamm abgeleitet, wobei zur Parameterisierung des Typs die Klasse Artikel verwendet wird. Bei Programmende wird der im Arbeitsspeicher gehaltene Index ebenfalls in einer Datei abgespeichert und bei einem neuen Start des Programms wieder geladen. Auch hierfür wird die generische Klasse DataFile verwendet und durch Parameterisierung – mit IndexT – die Klasse IndexFile gebildet (Abb. 10.5-2).

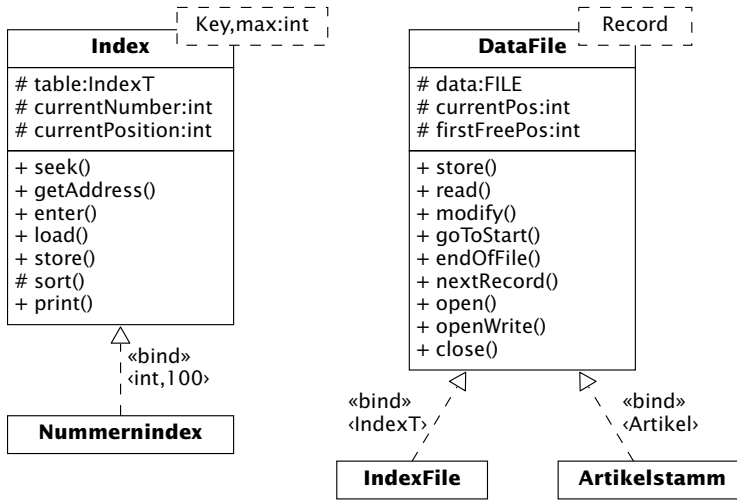


Abb.10.5-2: Generische Klassen zur Realisierung der indizierten Organisation

Bei vielen Anwendungen sind für ein Objekt eine große Anzahl von Attributen im Erfassungsfenster anzuzeigen, während das Objekt im Listenfenster nur durch wenige signifikante Attribute beschrieben wird, die zur Selektion der Objekte ausreichen. Bei Verwendung einer indizierten Organisation können diese Attribute alle im Index gespeichert werden. Dann ist beim Füllen der Tabelle im Listenfenster – im Gegensatz zur Modellierung in dieser Lehrinheit – kein Zugriff auf die Stammdatei notwendig.

Performance-Verbesserung

Die Anbindung an die Datenhaltung mit flachen Dateien erfolgt analog zu den objektorientierten Datenbanken. Wir beschränken uns hier auf die strenge Drei-Schichten-Architektur. Die GUI- und Fachkonzeptklassen können nahezu unverändert übernommen werden. Die Fachkonzeptklasse erbt jedoch ihre persistenten Eigenschaften nicht, sondern die Verwaltung der persistenten Objekte wird ausschließlich durch die Klassen der indizierten Organisation realisiert.

strenge Drei-Schichten-Architektur

Alle Zugriffe auf die Klassen `Nummernindex` und `Artikelstamm`, die in Abb. 10.5-3 grau dargestellt sind, erfolgen über die Klasse `Artikelliste`. Die Klasse `Artikelliste` erfüllt bei der strengen Drei-Schichten-Architektur also zwei Aufgaben: Einerseits die automatische Aktualisierung aller geöffneten Listenfenster und andererseits die Verkapselung der Zugriffe auf die jeweilige Form der Datenhaltung. Wie das Sequenzdiagramm der Abb.10.5-4 zeigt, ruft die Operation `insertArtikel()` nun – anstelle von `Assign()` und `Store()` – die Operationen `enter()` und `store()` auf. Analog dazu verwendet die Operation `getArtikel()` – anstelle der Poet-Operation `Get()` – nun die Operationen `getAddress()` und `read()`.

Beispiel



Auf der CD befindet sich die in C++ erstellte Implementierung dieses Beispiels.

ArtikelGUI-FK-Index

## LE 18 10 Entwurf einer Drei-Schichten-Architektur

Abb.10.5-3:  
Klassendiagramm  
für die strenge  
Drei-Schichten-  
Architektur mit  
Index

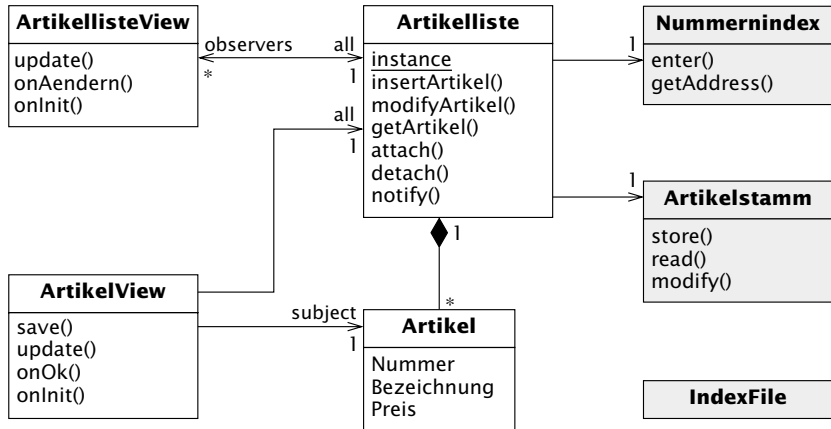
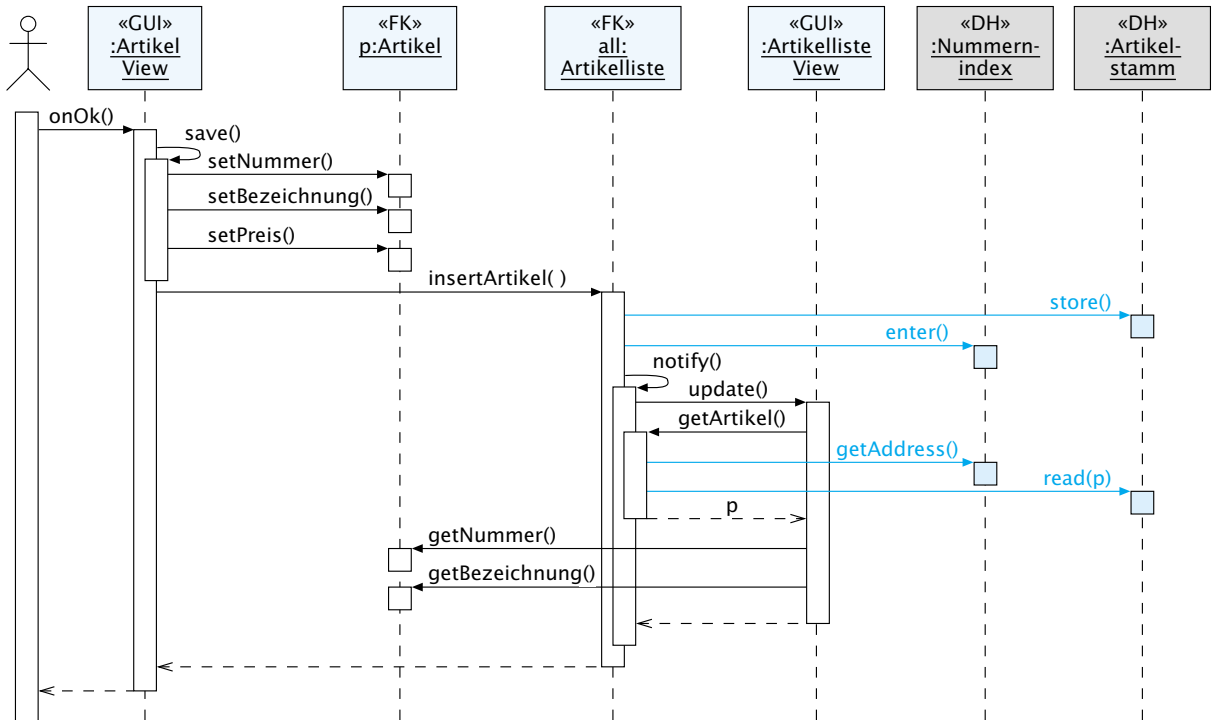


Abb.10.5-4:  
Speichern eines  
neuen Artikels mit  
Index und Aktualisie-  
rung der Listenfen-  
ster (strenge Drei-  
Schichten-Architektur)

Viele Compilerumgebungen und **Klassenbibliotheken** bieten Möglichkeiten an, Daten einfach zu speichern und wieder zu lesen. Diesem Komfort steht allerdings der Nachteil einer gewissen Abhängigkeit vom Compiler oder von der Plattform gegenüber. Beim objektorientierten Entwurf mittels einer Drei-Schichten-Architektur sind allerdings nur wenige Klassen und Operationen von notwendigen Änderungen betroffen.



Eine einfache Möglichkeit, Objekte permanent in einer Datei abzulegen, bietet die Serialisierung /Schmidberger et al.97/. Bei der **Serialisierung** werden im Gegensatz zu konventionellen Verfahren nicht nur die Attribute, sondern das gesamte Objekt unter Beibehaltung seiner Klasse, seiner Verbindungen (*links*) und seiner Attribute gespeichert. Die Serialisierung besitzt also gewisse Eigenschaften von objektorientierten Datenbanksystemen. Im Unterschied zu diesen ist die Serialisierung aber weder mehrbenutzerfähig, noch ermöglicht sie Anfragen oder Transaktionen. Außerdem müssen bei der Serialisierung immer alle Objekte des Archivs geladen werden.

Serialisierung

In der MFC-Bibliothek wird die Serialisierung durch eine logische Archivierungsdatei (Objekt der MFC-Klasse `CArchive`) und eine physische Datei (Objekt der Klasse `CFile`) realisiert (Abb. 10.5-5). Die Klasse `CArchive` generiert für jedes Objekt eine Objektidentität (OID). Dadurch wird sichergestellt,

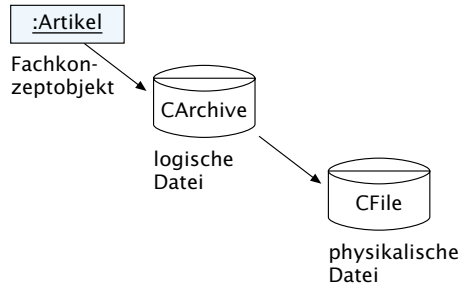


Abb.10.5-5: MFC-Konzept zur Serialisierung von Objekten

daß ein Objekt auch bei mehrfacher Serialisierung nur einmal in der physischen Datei gespeichert wird. Ein Archiv kann entweder zum Laden oder zum Speichern von Objekten verwendet werden, wobei die gewünschte Richtung im Konstruktor festgelegt werden muß.

Jede Klasse, deren Objekte serialisiert werden sollen, wird von der MFC-Klasse `CObject` abgeleitet. Desweiteren muß diese Klasse die von `CObject` geerbte polymorphe Operation `void Serialize (CArchive&)` überschreiben (Abb. 10-5-6).

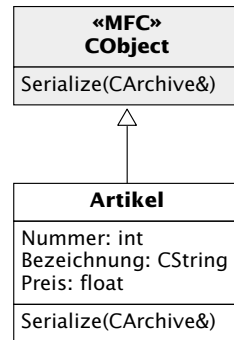


Abb.10.5-6: Serialisierung der Artikel-Objekte

## 10.6 Entwurf der Datenhaltung mit einem relationalen Datenbanksystem

Um eine objektorientierte Anwendung an ein **relationales Datenbanksystem** anzubinden, muß entweder die Schnittstelle selbst programmiert werden oder es werden entsprechende Produkte gekauft, die diese Funktionalität bereitstellen. Bei einer manuellen Anbindung müssen alle zu erledigenden Aufgaben selbst programmiert werden. Produkte, die eine Anbindung realisieren, können sehr stark in ihrer Mächtigkeit variieren. Bei Einsatz einer Klassenbibliothek wird ein Teil der Aufgaben durch die Bibliothek erledigt.

Die Vorteile einer solchen Klassenbibliothek liegen darin, daß

- durch ihre Verfügbarkeit auf verschiedenen Plattformen eine Unabhängigkeit vom Betriebssystem erreicht wird und
- eine weitgehende Unabhängigkeit von der verwendeten relationalen Datenbank möglich wird.

Eine komfortable Möglichkeit bietet der Einsatz eines Generators. Er automatisiert die Anbindung der Anwendung an die relationale Datenbank.

*Persistence Framework*

Für die Anbindung an die relationale Datenbank verwenden wir das *Persistence Framework* (PFW), das in /Larman 98/ beschrieben ist. Dieses **Framework** transformiert Objekte in Datensätze und speichert sie in der Datenbank und wandelt umgekehrt Sätze in Objekte, wenn ein lesender Zugriff erfolgt. Ein solches *Framework* kann nicht nur bei relationalen Datenbanken, sondern genauso bei flachen Dateien oder bei hierarchischen Datenbanken verwendet werden. Bei objektorientierten Datenbanken ist es überflüssig.

Das hier verwendete einfache *Framework* bietet folgende Funktionalität:

- Speichern und Holen von Objekten aus dem persistenten Speicher.
- *Commit* und *rollback*.

Darüberhinaus zeigt es sehr schön, wie Entwurfsmuster systematisch eingesetzt werden können.

Im folgenden wird beschrieben, wie mit dem *Persistence Framework* /Larman 98/ die Anbindung an ein relationales Datenbanksystem erfolgt.

### 1 Abbildung auf Tabellen

Der erste Schritt ist die **objekt-relationale Abbildung**, d.h. die Abbildung des Klassendiagramms auf Tabellen, wie sie bereits in Kapitel 8.3 beschrieben wurde. Um die Konsistenz von Datensätzen und Objekten sicherzustellen, wurden alle Sätze um eine Objektidentität (OID) erweitert. Eine gute Wahl ist ein 32-stelliger alphanumerischer Wert, der garantiert eindeutig ist. Jede Datenbanktabelle und jeder ihrer Datensätze erhält das OID-Attribut als Primärschlüssel. Dabei kann die Tabellen-OID Teil der Satz-OID sein. Dann kann jedes Objekt – das implizit eine OID besitzt – eindeutig auf einen Satz abgebildet werden.

Kapitel 8.3



Nach der Abbildung des Klassendiagramms in Tabellen kann eine Normalisierung der Tabellen notwendig sein. Da es sich hier um eine Aufgabe für einen Datenbankentwerfer handelt, gehe ich nicht weiter darauf ein. Das entstehende Datenbankschema kann sich vom ursprünglichen Klassendiagramm erheblich unterscheiden.

## 2 Database Broker-Muster

Wir benötigen eine Klasse *Broker*, die für das Zerlegen der Objekte in Datensätze (De-Materialisierung), das Wiedergewinnen der Objekte aus den Datensätzen (Materialisierung) und der *Cache*-Verwaltung verantwortlich ist. Jede persistente Klasse des Fachkonzepts kann eine eigene *Broker*-Klasse besitzen. Außerdem kann für jede Form der Speicherverwaltung ein eigener *Broker* definiert werden. Es entsteht eine Klassenhierarchie verschiedener *Broker* (Abb. 10.6-1), in die neue Unterklassen leicht eingefügt werden können. Diese Klassenhierarchie ist ein wesentlicher Teil des *Persistence Frameworks*.

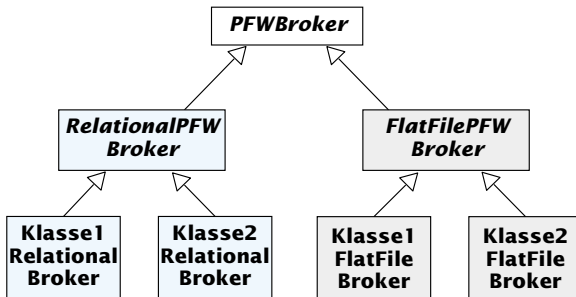


Abb. 10.6-1:  
Database Broker-  
Klassenhierarchie

## 3 Materialisierung von Objekten

Die Materialisierung wird mit Hilfe der Schablonenmethode-Musters (Kapitel 7.8) realisiert. Den prinzipiellen Entwurf zeigt Abb. 10.6-2, in der die Wirkung der wichtigsten Operationen mittels Pseudocode spezifiziert ist. Die Klasse *PFWBroker* besitzt die Operation `objectWith()`, deren Aufgabe es ist, ein Objekt zu materialisieren, d.h. aus einem oder mehreren Sätzen der relationalen Datenbank aufzubauen. Die Operation prüft, ob sich das Objekt bereits im *Cache*-Speicher befindet. Dann kann es sofort als Ergebnisparameter übergeben werden. Andernfalls wird die Operation `materializeWith()` aufgerufen, die das gewünschte Objekt aus einem oder mehreren Datensätzen mit Hilfe der Operation `currentRecordAsObject()` rekonstruiert. Die Operation `materializeWith()` ist aus der Sicht von `objectWith()` eine elementare Operation und aus der Sicht von `currentRecordAsObject()` eine Schablonenmethode.

Für jede Fachkonzeptklasse, deren Daten persistent sind, wird eine entsprechende *Broker*-Klasse definiert.

Das PFW demonstriert typische Eigenschaften eines *Frameworks*:

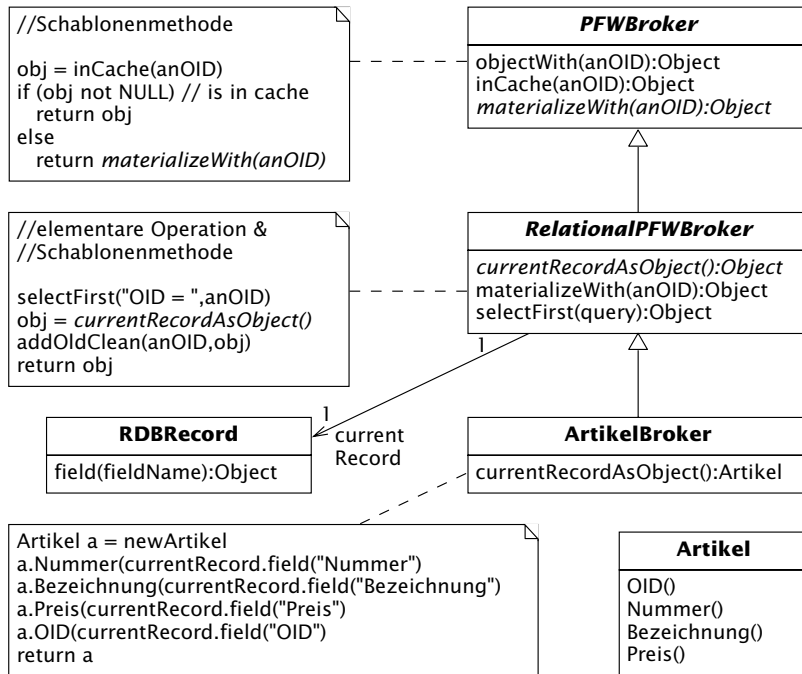
- 1 Abstrakte Oberklassen verwenden Schablonenmethoden.
- 2 Der Softwareentwickler fügt Unterklassen hinzu.
- 3 In den Unterklassen werden elementare Operationen (*primitive operations*) definiert, um die geerbten Schablonenmethoden zu vervollständigen.

Kapitel 7.8

Framework-  
Eigenschaften

**LE 18 10 Entwurf einer Drei-Schichten-Architektur**

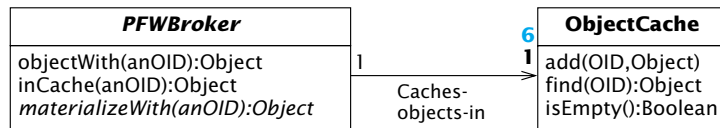
Abb. 10.6-2:  
Anwendung des  
Schablonenmusters  
für die  
Materialisierung



**4 Optimierung der Materialisierung mittels Cache**

Die Materialisierung von Objekten ist relativ langsam. Um die *Performance* zu verbessern, werden materialisierte Objekte daher in einem *Cache*-Speicher gehalten. Das *Cache-Management*-Muster beschreibt, wie der *Broker* diesen *Cache*-Speicher verwaltet (Abb. 10.6-3). Falls jede Anwendungsklasse einen eigenen *Broker* besitzt, gibt es auch für jede Klasse einen eigenen *Cache*-Speicher.

Abb. 10.6-3:  
Cache-Management-Muster



Statt eines einzelnen *Cache*-Speichers kann es zu einem *Broker* bis zu sechs *Cache*-Speicher geben. Jeder *Cache*-Speicher realisiert einen Transaktionszustand (siehe Punkt 8). Damit wird die Basis für *commit* und *rollback* von Transaktionen gelegt.

**5 Virtual Proxy-Muster**

ondemand-Materialisierung

Manchmal ist es wünschenswert, die Materialisierung eines Objekts solange hinauszuschieben, bis es wirklich benötigt wird (*on-demand materialization, lazy materialization*). Für die Lösung dieses Problems kann das bereits bekannte *Virtual Proxy*-Muster (Kapi-



tel 7.5) eingesetzt werden. Wenn der Klient den Preis eines Artikels wissen will und sich dieser Artikel nicht im Speicher befindet, dann sendet er die Botschaft `Preis()` an das `ArtikelProxy`-Objekt. Daraufhin wird die Materialisierung dieses Artikels durchgeführt (Abb. 10.6-4). In Abb. 10.6-2 wurde eine Vereinfachung vorgenommen und die `OID` bei der Fachkonzeptklasse `Artikel` eingetragen. In Wirklichkeit wird die `OID` – wie Abb. 10.6-4 – dem Proxy-Objekt zugeordnet. Die abstrakte Klasse `VirtualProxy` definiert die Eigenschaften (`OID`-Attribut und Assoziation zu `real Subject`) und die Operationen, die alle konkreten Proxy-Klassen gemeinsam haben.

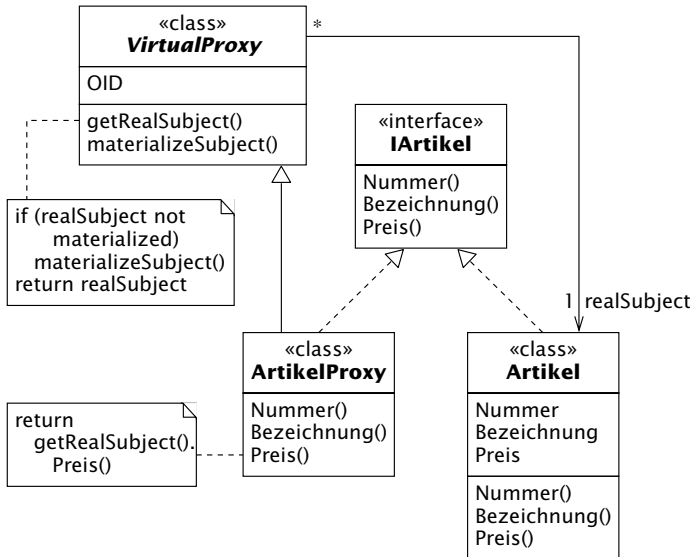


Abb. 10.6-4: Anwendung des Proxy-Musters für die Materialisierung

### 6 Kombination von Virtual Proxy und Database Broker

Wir kombinieren nun die beiden Muster *Virtual Proxy* und *Database Broker*, um ein Objekt aus der Datenbank zu materialisieren. Woher weiß ein konkretes Proxy-Objekt, welches konkrete *Broker*-Objekt es benutzen soll? Zur Lösung dieses Problems wird das bekannte Fabrikmethode-Muster (Kapitel 7.2) verwendet. Die Fabrikmethode `createBroker()` verwendet das *Singleton*-Muster, um genau ein Objekt der Klasse `ArtikelRelationalBroker` zu erzeugen (Abb. 10.6-5).

Kapitel 7.2

### 7 Materialisierung von Objektstrukturen

Bisher haben wir nur die Materialisierung einzelner Objekte betrachtet. Objekte sind jedoch nicht isoliert, sondern mit anderen Objekten verbunden. Beispielsweise besteht eine Bestellung aus mehreren Bestellpositionen, von denen jede wiederum einen Artikel referenziert, und zu jedem Artikel existiert ein Lieferant. Was soll passieren, wenn ein Artikel materialisiert wird? Wird nur das ge-



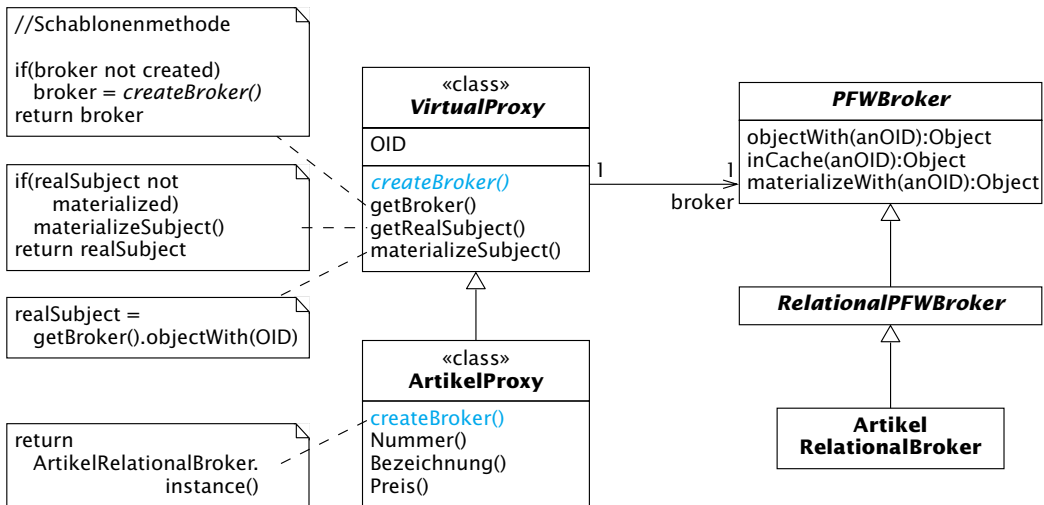


Abb. 10.6-5: Anwendung des Fabrikmethode-Musters für die Materialisierung

wünschte Artikelobjekt geladen oder werden alle mit ihm verbundenen Objekte ebenfalls geladen? Das Laden aller assoziierten Objekte ist langsam und benötigt viel Speicherplatz. Dieses Problem wird durch die *ondemand*-Materialisierung gelöst.

Wir betrachten der Einfachheit halber nur die Klassen Artikel und Lieferant, wobei jeder Artikel von genau einem Lieferanten geliefert wird. Soll für einen Artikel die Lieferfirma ermittelt werden, dann löst die Operation `ermittleLieferfirma()` die in Abb. 10.6-6 dargestellte Kommunikation aus. Wenn die Operation `currentRecordAsObject()` auf das `ArtikelRelationalBroker`-Objekt angewendet wird, dann wird das entsprechende Artikel-Objekt erzeugt und die OID des assoziierten Lieferanten ermittelt.

### 8 Transaktionen

Wenn ein *commit* auf der Datenbank durchgeführt werden soll, dann werden die Objekte in Abhängigkeit von ihrem Transaktionszustand unterschiedlich behandelt. Beispielsweise müssen geladene Objekte, die nicht verändert wurden, nicht in die Datenbank zurückgeschrieben werden, während das bei geänderten Objekten der Fall ist.

Transaktionszustände

Die möglichen Transaktionszustände sind:

- *new clean*, d.h. neu erzeugte Objekte, die nicht verändert wurden,
- *old clean*, d.h. alte materialisierte Objekte, die nicht verändert wurden,
- *new dirty*, d.h. neu erzeugte Objekte, die verändert wurden,
- *old dirty*, d.h. alte materialisierte Objekte, die verändert wurden,
- *new delete*, d.h. neu erzeugt Objekte, die gelöscht werden,
- *old delete*, d.h. alte materialisierte Objekte, die gelöscht werden.

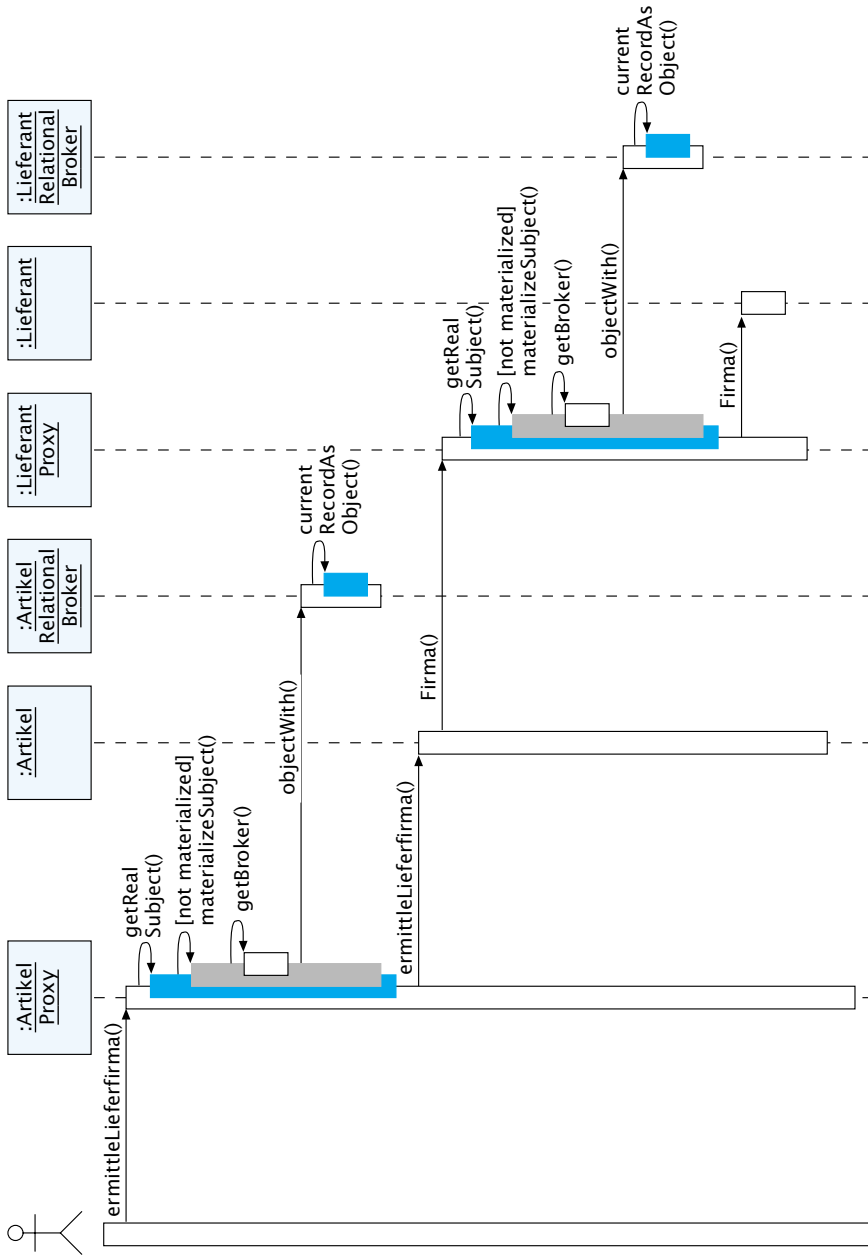


Abb. 10.6-6: Sequenzdiagramm für die Operation ermittelteLieferfirma()

Der *Broker* reserviert einen separaten *Cache* für jeden Zustand und stellt sicher, daß das Objekt im passenden *Cache* ist. Wenn ein Objekt beispielsweise erstmalig materialisiert wird, dann befindet es sich im *Old Clean Cache*. Nach einer Änderung seiner Attribute kommt es in den *Old Dirty Cache*.

## LE 18 10 Entwurf einer Drei-Schichten-Architektur

**Fassade** Die Kommunikation mit dem Broker erfolgt über die Klasse *BrokerServer*, die eine Fassade bildet. Der *BrokerServer* ermittelt dann den entsprechenden Broker. Damit wird die Kopplung zu den Broker-Klassen minimiert.

**commit** Wenn ein *commit* für eine Transaktion durchgeführt werden soll, dann wird die Botschaft *commit* an den *BrokerServer* geschickt. Der *BrokerServer* sendet dann ein *commit* an jeden Broker.

Während einer Transaktion kann ein Objekt erzeugt, geändert oder gelöscht – genauer gesagt als gelöscht markiert – worden sein. Die folgende Aufstellung faßt zusammen, wie Objekte in ihren möglichen Transaktionszuständen behandelt werden:

- new clean* → Objekt in die Datenbank einfügen,  
→ Objekt in *old clean Cache* verschieben,
- old clean* → keine Aktionen notwendig,
- new dirty* → Objekt in die Datenbank einfügen,  
→ Objekt in *old clean Cache* verschieben,
- old dirty* → Objekt in der Datenbank aktualisieren,  
→ Objekt in *old clean Cache* verschieben,
- new deleted* → Objekt aus dem *Cache* löschen,
- old deleted* → Objekt aus der Datenbank entfernen,  
→ Objekt aus dem *Cache* löschen.

**rollback** Wird eine Transaktion vollständig verworfen, dann spricht man von einem *rollback*. Dann wird dem *BrokerServer* die Botschaft *rollback* geschickt, die dann an alle Broker weitergegeben wird. Die *rollback*-Operation hat folgende Wirkung:

- old clean* → keine Aktionen notwendig,  
alle anderen Transaktionszustände → Objekt im *Cache* löschen.

Wenn ein *rollback* durchzuführen ist, zeigt sich deutlich der Vorteil, der durch die Verwendung des *Virtual Proxy* entsteht. Wenn die *Cache*-Speicher geleert sind, dann referenzieren die *Virtual Proxy*-Objekte nicht-materialisierte Objekte. Anschließend können die Objekte wieder aus der Datenbank materialisiert werden. Damit sind alle Änderungen der letzten Transaktion konsistent entfernt worden und der Zustand nach dem letzten *commit* ist wieder hergestellt worden.

**weitere Aufgaben des PFW** Ein *Persistence Framework* muß außer den beschriebenen noch zahlreiche weitere Aufgaben durchführen. Dazu gehören:

- De-Materialisierung von Objekten, d.h. Zerlegung von Objekten in Datensätze,
- Materialisierung und De-Materialisierung von Kollektionen,
- Fehlerbehandlung bei fehlerhaften Datenbank-Operationen,
- Zugriffsschutz auf die Datenbank,
- Mehrbenutzerzugriff und *locking*-Strategien.

Im allgemeinen verwenden Datenbanksysteme das pessimistische Sperrverfahren. Betroffene Sätze werden vor dem Schreibzugriff gesperrt und erst bei Transaktionsende wieder freigegeben.

ben. Der Vorteil dieses Verfahrens liegt in der Sicherheit der geänderten Daten, dem der Nachteil eines verringerten Durchsatzes gegenübersteht. Bei einem optimistischen Sperrverfahren wird ein Satz nicht vor dem Ändern, sondern erst am Ende der Transaktion gesperrt. In einem Zeitstempelverfahren wird geprüft, ob eine nebenläufige Operation den gleichen Satz geändert hat. Im Konfliktfall wird die gesamte Transaktion verworfen (*rollback*). Das Sperrvolumen nimmt hier wesentlich ab, jedoch führen häufige *rollbacks* zu einer Mehrbelastung.

**Technische Schnittstelle zu relationalen Datenbanksystemen**

Relationale Datenbanksysteme wurden ursprünglich als *stand alone* Informationssysteme realisiert. Das Ziel war, durch alleinigen Einsatz einer deklarativen Programmiersprache, z.B. durch SQL, die Datenbank zu programmieren. Damit der oben beschriebene Zugriff von einer objektorientierten Anwendung auf eine relationale Datenbank überhaupt stattfinden kann, muß das Programm eine technische Verbindung zur Datenbank aufbauen können.

Die Hersteller der diversen Datenbanken öffnen ihre Datenbanken für Anwendungsprogramme mit herstellereigenen Schnittstellen. Diese Schnittstellen besitzen den Vorteil, daß sie für das jeweilige Datenbanksystem maßgeschneidert sind. Andererseits binden sie den Softwareentwickler an das Datenbanksystem und machen die Entwicklung von Anwendungen für mehrere verschiedene Datenbanken sehr aufwendig.

proprietäre Schnittstellen

Unter *Embedded SQL* versteht man die Erweiterung einer Programmiersprache um spezielle Sprachkonstrukte. Sie ermöglichen es, direkt aus der Anwendung über SQL-Befehle auf die Datenbank zuzugreifen. Ein Präprozessor wandelt die SQL-Befehle in Aufrufe von Bibliotheksprogrammen um. Das so entstandene Programm kann dann mit einem herkömmlichen Compiler übersetzt werden. Auch *Embedded SQL* ist als eine proprietäre Schnittstelle aufzufassen, denn jeder Hersteller einer relationalen Datenbank unterstützt seine eigene *Embedded SQL*-Implementierung und es muß der zugehörige Präprozessor verwendet werden.

*Embedded SQL*

Die MFC-Klassen bieten beispielsweise die Datenzugriffsobjekte **DAO** (*Data Access Objects*) an, die es auf einfache Art ermöglichen, von einem in einer OOP geschriebenen Programm auf relationale Datenbanken zuzugreifen.

DAO

**ODBC** (*Open Database Connectivity*) ist eine standardisierte Schnittstelle für den Zugriff auf relationale Datenbanksysteme. Sie wurde ursprünglich von Microsoft spezifiziert, hat sich aber inzwischen zu einem betriebssystemübergreifenden, allgemein akzeptierten de facto-Standard entwickelt. Mittels ODBC kann auf jede Datenbank, für die ein ODBC-Treiber existiert, zugegriffen werden. ODBC stützt sich auf eine standardisierte SQL-Version. Dadurch ist es

ODBC

möglich, die Anwendung unabhängig von einer spezifischen Datenbank zu programmieren. ODBC kann jedoch ausschließlich von C- und C++-Programmen verwendet werden. Es wurde ursprünglich für C entwickelt und ist daher nicht objektorientiert. ODBC stellt dem Programm ganzzahlige Werte als *handles* und Zeiger auf Strukturen zur Verfügung, die von der Anwendung zu verwenden sind. Der Verzicht auf Objektorientierung und der Zugriff mittels Zeigern machen ODBC relativ fehleranfällig.

JDBC Mit **JDBC** (*Java Database Connectivity*) hat *Sun Microsystems* einen Standard definiert, um aus Java-Programmen heraus auf relationale Datenbanksysteme zugreifen zu können. JDBC ist durch die Verwendung von Java als Programmiersprache vollständig objektorientiert und plattformunabhängig /Klute 98/. Zwei Besonderheiten zeichnen JDBC aus:

- Der Zugriff erfolgt unabhängig vom jeweiligen Datenbanksystem. Beispielsweise kann eine Anwendung, die bisher auf eine Oracle-Datenbank zugegriffen hat, auch mit einer Informix-Datenbank arbeiten.
- Ein JDBC-Treiber sorgt für die spezifische Anpassung an das jeweilige Datenbanksystem. Er ist ebenfalls in Java geschrieben und wird von den Datenbankherstellern oder von Dritten angeboten.

**Flache Dateien (*flat files*)** Unter einer Speicherverwaltung mit flachen Dateien ist eine Organisationsform zu verstehen, die nur rudimentäre Zugriffsoperationen anbietet.

**Flexible Drei-Schichten-Architektur** Eine flexible Drei-Schichten-Architektur ergibt sich, wenn die GUI-Schicht sowohl auf die Fachkonzeptschicht als auch auf die Datenhaltungsschicht zugreifen darf.

**Framework** Ein *Framework* besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und insbesondere aus abstrakten Klassen, die Schnittstellen definieren. Die abstrakten Klassen enthalten sowohl abstrakte als auch konkrete Operationen. Im allgemeinen wird vom Anwender (=Programmierer) des *Frameworks* erwartet, daß er Unterklassen definiert, um das *Framework* zu verwenden und anzupassen.

**JDBC (*Java Database Connectivity*)**

Mit JDBC hat *Sun Microsystems* einen Standard definiert, um aus Java-Programmen heraus auf relationale Datenbanksysteme zugreifen zu können. JDBC ist durch die Verwendung von Java als Programmiersprache vollständig objektorientiert und plattformunabhängig.

**ODBC (*Open Database Connectivity*)**

ODBC ist eine standardisierte Schnittstelle für den Zugriff auf relationale Datenbanksysteme. Sie wurde ursprünglich von Microsoft spezifiziert, hat sich aber inzwischen zu einem betriebssystemübergreifenden, allgemein akzeptierten de facto-Standard entwickelt.

**Objektorientiertes Datenbanksystem (*object database system*)**

Ein objektorientiertes Datenbanksystem (ODBS) ist ein Datenbanksystem, dem ein objektorientiertes Datenmodell zugrunde liegt. Es integriert die Eigenschaften einer Datenbank mit den Möglichkeiten von objektorientierten Programmiersprachen.



**Objekt-relationale Abbildung (object relational mapping)** Die objekt-relationale Abbildung gibt an, wie ein Klassendiagramm auf Tabellen einer relationalen Datenbank abgebildet wird. Sie enthält Abbildungsvorschläge für Klassen, Assoziationen und Vererbungsstrukturen. Ein weiterer Aspekt ist die Realisierung der Objektidentität in relationalen Datenbanken.

**Persistenz** Persistenz ist die Fähigkeit eines Objekts, über die Ausführungszeit eines Programms hinaus zu leben, d.h. die Daten dieses Objekt bleiben auch nach Beendigung des Programms

erhalten und stehen bei einem Neustart wieder zur Verfügung.

**Relationales Datenbanksystem (relational database system)** Ein relationales Datenbanksystem (RDBS) ist ein Datenbanksystem, dem ein relationales Datenmodell zugrunde liegt. Die Daten werden in Form von Tabellen gespeichert.

**Strenge Drei-Schichten-Architektur** Bei einer strengen Drei-Schichten-Architektur kann die GUI-Schicht nur auf die Fachkonzeptschicht und letztere nur auf die Datenhaltungsschicht zugreifen.



Bei der **Drei-Schichten-Architektur** werden zwei Ausprägungen unterschieden. In der **flexiblen** Form darf jede Schicht auf alle darunter liegenden Schichten zugreifen, bei der **strengen** Form sieht jede Schicht nur die direkt darunter liegende. Wenn keine Datenbank verwendet wird, muß die Datenhaltung mit **flachen Dateien** und den klassischen Speicherungsverfahren realisiert werden, wobei diese objektorientiert entworfen werden. Alternativ kann die Datenverwaltung einer Klassenbibliothek benutzt werden, die eine schnelle Realisierung ermöglicht. Ein **objektorientiertes Datenbanksystem** kann relativ einfach an die Fachkonzeptschicht angebunden werden, weil alle Konzepte aufeinander abgestimmt sind. Aufwendiger ist die systematische Anbindung an ein **relationales Datenbanksystem**.

**1 Lernziel:** *Drei-Schichten-Architektur mit Anbindung an objektorientierte Datenbank erstellen können.*

Aufgabe  
20–25 Minuten

Die Ausgangsbasis für diese Aufgabe bildet die Aufgabe 3 der Lehrinheit 17 und das dort erstellte Entwurfsmodell. Binden Sie nun das objektorientierte Datenbanksystem Poet an und verwenden Sie die strenge Drei-Schichten-Architektur.

- a** Erstellen Sie das Klassendiagramm, in das Sie alle verwendeten Operationen eintragen, die Sie im Sequenzdiagramm der Teilaufgabe b benötigen.
- b** Erweitern Sie das Sequenzdiagramm um die Zugriffe auf die Datenbank.

## LE 18 Aufgaben

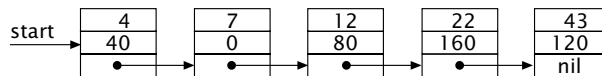
Aufgabe 2 **Lernziel:** Änderbarkeit einer objektorientierten Datenhaltung mit flachen Dateien beurteilen können.  
5–10 Minuten

Nehmen Sie die Klassendiagramme der Abb. 10.5-2 bis Abb. 10.5-4 zur Hand. Beantworten Sie folgende Fragen.

- Wie ändert sich die Spezifikation, wenn für den Index anstelle eines Arrays eine verkettete Liste (Abb. LE18-A2) verwendet wird?
- Wie ändert sich die Spezifikation, wenn im Index alle Attribute des Listenfensters gespeichert werden?

Abb. LE18-A2:  
Indizierte Datei-  
verwaltung mit  
verketteter Liste

einNummernindex



einArtikelstamm

	Nummer	Bezeichnung	Preis
0	7		
40	4		
80	12		
120	43		
160	22		

Aufgabe 3 **Lernziel:** Wissen, wie die Anbindung an eine relationale Datenbank funktioniert.  
10–15 Minuten

- Was versteht man unter Materialisierung eines Objekts?
- Durch welche Klassen wird beim *Persistence Framework* die Materialisierung von Objekten realisiert?
- Erläutern Sie, wie das Schablonenmethode-Muster bei der Materialisierung angewendet wird.
- Wie kann die Effizienz der Materialisierung gesteigert werden?
- Was versteht man unter *ondemand*-Materialisierung?
- Was ist ein *rollback* und wann wird er durchgeführt?

Aufgabe 4 **Lernziel:** Erkennen, welche Entwurfsmuster beim Erstellen des Entwurfsmodells verwendet wurden.  
10 Minuten

Prüfen Sie, welche der in Kapitel 7 beschriebenen Entwurfsmuster in den Lehreinheiten 17 und 18 verwendet wurden. Geben Sie an, welche Probleme dadurch gelöst wurden.

# Exkurs 1

## Erstellen des Prototyps der Benutzungsoberfläche mit dem Ressourcen-Editor

(Microsoft Visual Studio C++)



- Wissen, wie Interaktionselemente mit dem Ressourcen-Editor des Microsoft Visual Studios C++ erstellt werden.
- Prototyp der Benutzungsoberfläche mit Dialogfenstern erstellen können.
- Prototyp der Benutzungsoberfläche mit MDI-Unterfenstern erstellen können.

wissen

anwenden



- Sie sollten die Lehreinheiten zu den objektorientierten Konzepten der Analyse (Kapitel 2) und zur Gestaltung der Benutzungsoberflächen (Kapitel 5) durchgearbeitet haben.
- Da sich dieser Exkurs direkt auf die Fallstudie *Diploma* bezieht, sind aus dem Anhang 1 die Kapitel 1 bis 3 Voraussetzung.
- Außerdem benötigen Sie Grundkenntnisse in C++ und im Umgang mit dem Microsoft Visual Studio.
- Lesern mit geringeren Programmierkenntnissen empfehle ich die Realisierung des Prototyps mit Dialogfenstern. Erfahrene Programmierer können gleich mit der MDI-Form des Prototyps beginnen.

- 1     Interaktionselemente des Ressourcen-Editors 418
- i     2     Prototyp mit Dialogfenstern 420
  - 2.1   Arbeitsbereich anlegen 421
  - 2.2   Erstellen der Menüs 421
  - 2.3   Realisieren des Erfassungsfensters 422
  - 2.4   Realisieren des Listenfensters 424
  - 2.5   Verbinden der Menüs und der Fenster 425
  - 2.6   Programmieren der Schaltflächen in den Fenstern 426
  - 2.7   Realisieren der *one*-Richtung einer Assoziation 427
  - 2.8   Realisieren der *many*-Richtung einer Assoziation 429
  - 2.9   Programmieren der Schaltflächen für die Assoziationen 430
  - 2.10  Abbildung des vollständigen OOA-Modells auf die Benutzungsoberfläche 431
- 3     Prototyp mit MDI-Unterfenstern 434
  - 3.1   Arbeitsbereich anlegen 434
  - 3.2   Erstellen der Menüs 434
  - 3.3   Realisieren des Erfassungsfensters 435
  - 3.4   Realisieren des Listenfensters 438
  - 3.5   Verbinden der Menüs und der Fenster 439
  - 3.6   Definieren der Schaltflächen in den Fenstern 440
  - 3.7   Abbildung weiterer Klassen auf die Benutzungsoberfläche 441
  - 3.8   Realisieren der Assoziation 442



## Exkurs 1 1 Interaktionselemente des Ressourcen-Editors

Dieser Exkurs hat *nicht* das Ziel, Ihnen die Programmierung von Benutzungsoberflächen in C++ beizubringen, sondern stellt eine praktische Anweisung dar, wie ein Prototyp der Benutzungsoberfläche einfach erstellt werden kann. Lesern, die eine fundierte Einführung für die Programmierung in C++ suchen, empfehle ich /Schmidberger et al. 97/ und /Stroustrup 98/. Für spezielle Fragen zu C++ ist /Kruglinski 97/ gut geeignet.

## 1 Interaktionselemente des Ressourcen-Editors

In diesem Kapitel wird erläutert, wie die Interaktionselemente bzw. Steuerelemente der Lehreinheit 10 (Kapitel 5.6) mit dem Ressourcen-Editor erstellt werden. Die folgenden Angaben beziehen sich jeweils auf den Eigenschaften-Dialog des Interaktionselements. »Ein« bedeutet das Anklicken des entsprechenden Kontrollkästchens, »Aus« steht für Ausschalten. Es wird die deutsche Version des Visual Studios C++ 5.0 verwendet.

Symbolleiste  
Steuerelement

Die Interaktionselemente (Steuerelemente) werden standardmäßig angezeigt bzw. sind über *Extras/Anpassen* und *Symbolleisten/Steuerelemente* erhältlich.

### Eingabefeld (*text box*) für numerische Daten

Folgende Einstellung sorgt dafür, daß nur Ziffern eingegeben werden können:

■ Numerisch: Ein

Eine Zahl soll immer vollständig im Eingabefeld dargestellt werden und die Ausgabe der Zahl rechtsbündig erfolgen.

■ Text ausrichten: Rechtsbündig

■ Auto.Hor.Bildlauf: Aus

■ Schreibgeschützt: Ein (bei Ausgabefeldern)

### Eingabefeld (*text box*) für Texte

Bei einzeiligen Texten können alle Voreinstellungen übernommen werden. Bei mehrzeiligen Texten gilt:

■ Mehrzeilig: Ein

■ Vert. Bildlauf : Ein

■ Auto Hor. Bildlauf: Aus

■ Return möglich: Ein (Enter-Taste löst Übergang in nächste Zeile aus)

■ Schreibgeschützt: Ein (bei Ausgabefeldern)

### **Führungstext**

Der Führungstext ist bei einzeiligen Eingabefeldern links davor zu schreiben, bei mehrzeiligen darüber. Bei einzeiligen Eingabefeldern ist er gegebenenfalls rechtsbündig auszurichten.

### **Schaltfläche (*button*)**

- Mnemotechnisches Auswahlzeichen durch das Präfix »&« kennzeichnen, z.B. &Liste für Liste
- Ok  
ID = IDOK            Auslösen auch durch Enter-Taste vordefiniert
- Abbrechen  
ID = IDCancel        Auslösen auch durch ESC-Taste vordefiniert

### **Optionsfeld (*option button*)**

- Die Zusammengehörigkeit von Optionsfeldern kann durch ein Gruppenelement betont werden. Außer dieser optischen Gruppenbildung gibt es eine logische Gruppierung, die im Eigenschaftendialog spezifiziert wird.
- Pro Gruppe stellen Sie ein:  
Gruppe: Ein (für erstes Optionsfeld)  
Gruppe: Aus (ab dem zweiten Optionsfeld)  
Dadurch wird sichergestellt, daß pro Gruppe maximal ein Optionsknopf selektiert ist.
- Die mnemotechnische Auswahl wird durch das Präfix »&« gekennzeichnet.

### **Kontrollkästchen (*check box*)**

- Analog zu den Optionsfelder können Kontrollkästchen mit dem Gruppenelement optisch gruppiert werden.

### **Listenfeld (*list box*)**

- Auswahl: Einfach (Einfachauswahlliste)
- Auswahl: Mehrfach (Mehrfachauswahlliste)
- Auswahl: Erweitert (Mehrfachauswahlliste)
- Mehrspaltig: Ein (mehrspaltige Auswahlliste)
- Sortieren: Ein (automatische Sortierung)
- Horiz. Bildlauf: Ein
- Vert. Bildlauf: Ein

### **Listenelement (*list view control*)**

- Gewünschte Ansicht einstellen, z.B. für Tabelle: Bericht
- Einzelauswahl: Ein (Einfachauswahlliste)

### **Kombinationsfeld (*combo box*)**

Über den »Typ« wird gesteuert, welches der folgenden Interaktionselemente verwendet wird.

## Exkurs 1 1 Interaktionselemente des Ressourcen-Editors

- Typ: einfach (Eingabefeld mit Listenfeld, d.h. Kombinationsfeld laut Windows-Terminologie)
- Typ: *Dropdown* (Eingabefeld mit *Dropdown*-Listenfeld, d.h. *Dropdown*-Kombinationsfeld laut Windows-Terminologie)
- Typ: *Dropdown*-Listenfeld (*Dropdown*-Listenfeld laut Windows-Terminologie)

Im Eigenschaftsdialog können auf der Notizbuch-Seite *Daten* Listeneinträge direkt eingegeben werden. Der Übergang in die nächste Zeile erfolgt mit den Tasten *Strg + Enter*.

Die Länge der Liste ist mit der Maus auf die gewünschte Größe zu ziehen (bei den eingeklappten Formen zuvor auf das Dreieck klicken!).

### **Drehfeld (*spin box*)**

Um ein Drehfeld zu erstellen, muß das Eingabefeld mit dem *spin* kombiniert werden.

### **Piktogramm bzw. Symbol (*icon*)**

Das Symbol wird unter anderem mit folgenden Elementen kombiniert:

- Schaltfläche (Symbol: Ein)
- Optionsknopf (Symbol: Ein)
- Kontrollkästchen (Symbol: Ein)
- Listenelement (Ansicht: Symbol oder Minisymbol)

### **Regler (*slider*)**

Die Beschriftung muß durch einen Führungstext und gegebenenfalls durch eine Gruppe erstellt werden.

### **Register (*tab control*)**

Damit dieses Interaktionselement für den Prototyp sinnvoll verwendet werden kann, muß bereits bei der Erstellung des Prototyps Programmcode geschrieben werden.

### **Strukturansicht (*tree view control*)**

Folgende Einstellungen sind empfohlen:

- Mit Schaltfläche: Ein
- Mit Linien: Ein
- Linien am Ursprung: Ein

## 2 Prototyp mit Dialogfenstern

In diesem Kapitel wird ein Prototyp entwickelt, bei dem die Erfassungs- und Listenfenster mittels **nicht-modaler Dialogfenster** realisiert werden. Sie sind im Unterschied zu den MDI-Unterfenstern



nicht in der Größe veränderbar und können über das Anwendungsfenster herausgeschoben werden. Diese beiden Eigenschaften werden für den betrachteten Anwendungstyp nicht unbedingt benötigt. Der vollständige Prototyp ist im Verzeichnis *DiplomaDlg* enthalten.

DiplomaDlg

Wir transformieren zunächst eine einzige Klasse in einen Prototyp. Unsere Ausgangsbasis ist die – isolierte – Klasse *Professor*, die von der Klasse *Person* abgeleitet wird (Abb. 2-1). In Kapitel 2.9 wird die Assoziation zur Diplomarbeit hinzugefügt.

**2.1 Arbeitsbereich anlegen**

- Im Menü ist *Datei/Neu* zu wählen und unter Projekte *MFC-Anwendungs-Assistent(exe)* anzugeben. Als Projektname wird *Diploma* eingetragen. Es wird automatisch ein gleichnamiges Verzeichnis angelegt. Alle sonstigen Standardeinstellungen belassen und *Ok* drücken.
- Schritt 1: SDI wählen.
- Schritt 2: keine Datenbankunterstützung.
- Schritt 3: keine Unterstützung.
- Schritt 4: Statusleiste und 3D-Steuererelemente anklicken, den Rest ausschalten.
- Schritt 5: Statisch verknüpfte Bibliothek.
- Schritt 6: Fertigstellen.
- Dann werden folgende Klassen generiert: CDiplomaApp, CMainFrame, CDiplomaDoc, CDiplomaView.
- Das erzeugte Rahmenprogramm sollten Sie zunächst mit *Erstellen/Ausführen* laufen lassen, um sich über den Leistungsumfang des Microsoft Visual Studios klar zu werden.

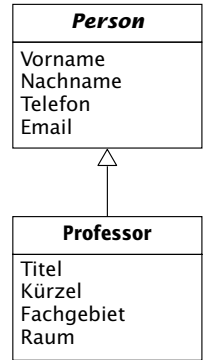


Abb. 2-1: Klasse Professor

In den nächsten Schritten werden wir die standardmäßig generierte Oberfläche an unsere Anforderungen anpassen.

**2.2 Erstellen der Menüs**

Im *Ressource View* sind folgende Änderungen vorzunehmen:

- Öffnen Sie *Menu/IDR\_MAINFRAME*.
- Ändern Sie die Menü-Hauptleiste wie in Abb. 2-2 angegeben.
- Passen Sie dann die *pull-down*-Menüs an. Selektieren Sie eine Menüoption und drücken Sie die rechte Maustaste.



Abb. 2-2: Menüs für Diploma

## Exkurs 1 2 Prototyp mit Dialogfenstern

Eigenschaften  
Menü

- Dann führen Sie die folgenden Einstellungen im Eigenschaftsdialog durch:

ID	Beschriftung
ID_Datenbank_Oeffnen	&Datenbank öffnen
ID_Liste_Professor	&Professor...
ID_Liste_Diplomarbeit	&Diplomarbeit...
ID_Liste_Student	&Student...
ID_Erfassen_Professor	&Professor...
ID_Erfassen_Diplomarbeit	&Diplomarbeit...
ID_Erfassen_Student	&Student...

Die Menüoption *Beenden* bleibt. Die restlichen Menüoptionen sind zu löschen.

Es ist wichtig, daß Sie die IDs sorgfältig benennen, denn über sie werden im Programm die Menüoptionen angesprochen. Das Zeichen »&« gibt an, daß das nachfolgende Zeichen als mnemonisches Kürzel zur Auswahl der Menüoption verwendet werden kann. Die mnemonischen Kürzel sind in den Menüs durch Unterstreichen gekennzeichnet.

### 2.3 Realisieren des Erfassungsfensters

#### Dialog-Ressource erstellen

- Für das Gestalten der Dialogmasken läßt sich sehr komfortabel der Klassen-Assistent einsetzen. Damit ist es möglich, mit relativ wenig Programmieraufwand den Prototyp zu erstellen. Wählen Sie *Dialog/Dialog einfügen...* und erstellen Sie eine neue Dialog-Ressource.
- Dieses Fenster können Sie nun mit den Interaktionselementen wie in der Abb. 2-3 gestalten. Die Attribute *Fachgebiet* und *Titel* werden auf Kombinationsfelder des Typs *Dropdown* abgebildet, alle anderen Attribute auf Eingabefelder.

Abb. 2-3:  
Erfassungsfenster  
für die Klasse  
Professor (ohne  
Assoziation)

The image shows a Windows-style dialog box titled "Neu Professor" with a close button (X) in the top right corner. The dialog is divided into two main sections. The left section is titled "Name" and contains two text input fields: "Vorname" and "Nachname". Below these are two more text input fields: "Telefon" and "Email". The right section contains a "Fachgebiet" dropdown menu, a "Titel" dropdown menu, a "Raum" text input field, and a "Kürzel" text input field. At the bottom of the dialog, there are four buttons: "OK", "Übernehmen", "Abbrechen", and "Liste ...".

Übernehmen Sie die **Schaltflächen** *Ok* und *Abbrechen* – wodurch deren vordefinierte Wirkung für Dialogfenster gültig bleibt – und geben Sie für die anderen Schaltflächen folgende IDs ein:

Eigenschaften  
ProfessorView

<b>ID</b>	<b>Beschriftung</b>
IDÜbernehmen	&Übernehmen
IDListe	&Liste...

- Die **statischen Textfelder** heißen alle IDC\_STATIC. Da diese Namen im Programm nicht verwendet werden, können wir sie so belassen.
- Benennen Sie die Eingabefelder und *Dropdown*-Kombinationsfelder mit:
 

IDC_Vorname	IDC_COMBOFachgebiet
IDC_Nachname	IDC_COMBOTitel
IDC_Telefon	IDC_Raum
IDC_Email	IDC_Kuerzel
- Das Gruppenelement wird im Programmcode ebenfalls nicht verwendet. Daher belassen wir den Namen IDC\_STATIC.
- Definieren Sie folgende Eigenschaften des Fensters:
  - ID = IDD\_ProfessorView, Beschreibung = Neu Professor
  - Kontrollkästchen Sichtbar: Ein (notwendig für nicht-modale Dialoge)
- Im Hauptmenü können Sie in *Layout/Tabulatorreihenfolge* durch einfaches Anklicken der Felder die Reihenfolge festlegen, in der die Elemente später mit der Tabulator-Taste angesprochen werden.
- Sie können das Fenster anschließend mit dem Test-Schalter überprüfen.

Tabulatoren

Fenster testen

### Realisierung der Dialog-Ressource durch eine Klasse

Jedes Fenster wird durch eine Klasse realisiert. In der MFC-Bibliothek stellt die Klasse *CWnd* die Funktionalität für alle Fensterklassen zur Verfügung, von der dann spezialisierte Fensterklassen abgeleitet werden. Dabei wird zwischen dem Fensterobjekt und dem Fenster unterschieden, die beide eng zusammenwirken. Das Fensterobjekt wird durch »normale« Konstruktoren und Destruktoren erzeugt und gelöscht. Das Fenster wird mittels *Create()* erzeugt und mittels *DestroyWindow()* gelöscht.

Da es in diesem Exkurs darum geht, die Abbildung des OOA-Modells auf eine objektorientierte Benutzungsoberfläche zu demonstrieren, verwende ich der Einfachheit halber die vordefinierte Funktionalität der MFC-Klassen und verzichte auf eine Speicherbereinigung.

Für die Benennung der Klassen wird folgende Systematik verwendet, damit alle Klassen, die mit Professor »zu tun haben«, im *Class View* untereinander stehen:

Benennung der  
Klassen

## Exkurs 1 2 Prototyp mit Dialogfenstern

Klassen-Assistent  
ProfessorView

- Professor = OOA-Klasse,
  - ProfessorView = Klasse, die das Erfassungsfenster realisiert,
  - ProfessorenView = Klasse, die das Listenfenster realisiert.
- Mit Hilfe des Klassen-Assistenten definieren Sie mittels *Klasse hinzufügen/Neu* für das Erfassungsfenster die Klasse ProfessorView. Da es sich um ein Dialogfenster handelt, wird Dialog als Basisklasse gewählt.
- Um auf Interaktionselemente im Programm einfach zugreifen zu können, sollten Sie mit dem Klassen-Assistenten für die *Dropdown*-Listenfelder **Member-Variablen** erstellen. Wählen Sie folgende Namen: m\_ComboFachgebiet, m\_ComboTitel (jeweils mit der Kategorie *Control*).
- Erstellen Sie mit dem Klassen-Assistenten für die Objekt-ID = ProfessorView und die Nachricht = WM\_INITDIALOG eine **Operation**. Übernehmen Sie den vorgeschlagenen Namen OnInitDialog(). Diese Operation wird vom System automatisch beim Öffnen des Fensters aufgerufen.
- Die beiden *Dropdown*-Kombinationsfelder können Sie folgendermaßen mit Testdaten füllen, sofern die Listeneinträge nicht bereits beim Erstellen der Dialog-Ressource eingegeben wurden.

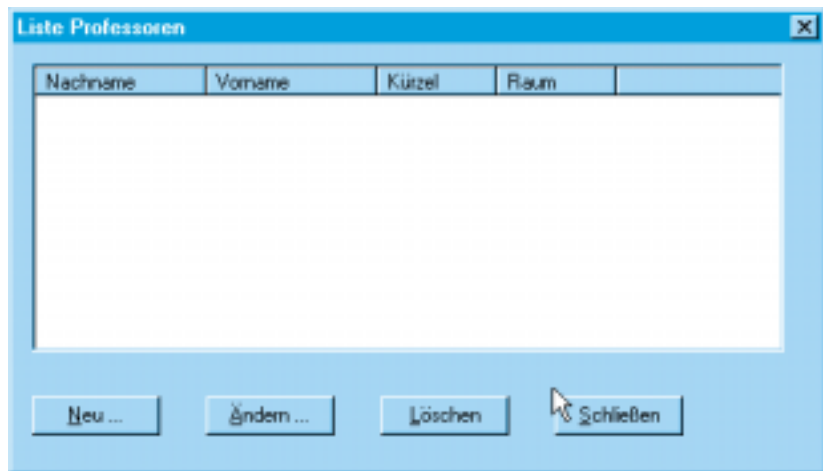
```
BOOL ProfessorView::OnInitDialog()  
{ m_ComboFachgebiet.AddString("Softwaretechnik");  
  ...  
  m_ComboTitel.AddString("Dr.");  
  ...  
}
```

### 2.4 Realisieren des Listenfensters

#### Dialog-Ressource erstellen

Erstellen Sie nun analog das Listenfenster wie in Abb. 2-4 gezeigt.

Abb. 2-4:  
Listenfenster  
für die Klasse  
Professor



- Erzeugen Sie eine zweite Dialog-Ressource und tragen Sie folgende Eigenschaften ein:
 

<b>ID</b>	<b>Beschriftung</b>
IDC_Neu	&Neu
IDC_Aendern	&Ändern
IDC_Loeschen	&Löschen
IDC_Schliessen	&Schließen
- Fügen Sie das Interaktionselement *Listenelement* ein:  
ID= IDC\_LISTProfessor, Ansicht = Bericht.
- Definieren Sie folgende Eigenschaften des Fensters:
  - ID = IDD\_ProfessorenView, Beschreibung = Liste Professor
  - Kontrollkästchen Sichtbar: Ein (notwendig für nicht-modale Dialoge)

**Realisierung der Dialog-Ressource durch eine Klasse**

- Erstellen Sie mit dem Klassen-Assistenten die Klasse *ProfessorenView*, die ebenfalls von *CDialog* abgeleitet wird. Klassen-Assistent  
ProfessorenView
- Definieren Sie die **Member-Variable** *m\_Liste* für das Listenelement.
- Erstellen Sie mit dem Klassen-Assistenten die **Operation** *OnInitDialog()*. Gehen Sie dabei analog zum Erfassungsfenster vor (Objekt-ID = *ProfessorenView*, Nachricht = *WM\_INITDIALOG*).
- Damit das Fenster wie in Abb. 2-4 aussieht, müssen Sie folgenden Programmcode in C++ einfügen:

```

BOOL ProfessorenView::OnInitDialog()
{
    ...
    m_Liste.InsertColumn(0, "Nachname", LVCFMT_LEFT, 100);
    m_Liste.InsertColumn(1, "Vorname", LVCFMT_LEFT, 100);
    m_Liste.InsertColumn(2, "Kürzel", LVCFMT_LEFT, 70);
    m_Liste.InsertColumn(3, "Raum", LVCFMT_LEFT, 70);
    ...
}
    
```

**2.5 Verbinden der Menüs und der Fenster**

- In diesem Schritt verbinden wir Menü und Fenster auf folgende Art:
- Die Menüoption *Erfassung/Professor* öffnet das Fenster *ProfessorView*. Die Schaltflächen *Ok* und *Abbrechen* schließen das Fenster.
  - Die Menüoption *Listen/Professor* öffnet das Fenster *ProfessorenView*. Die Schaltfläche *Schließen* schließt es wieder.
  - Rufen Sie für die Menüoption *Erfassung/Professor* den Klassen-Assistenten auf, wählen für die Klasse *CMainFrame* die Objekt-ID = *ID\_Erfassen\_Professor* und die Nachricht = *Command* und drücken dann *Funktion hinzufügen*. Klassen-Assistent  
CMainFrame
  - In der Klasse *CMainFrame* wird eine Member-Funktion mit dem Namen *OnErfassungProfessor()* generiert. Fügen Sie mittels *Code be-*



## Exkurs 1 2 Prototyp mit Dialogfenstern

*arbeiten* folgenden Code ein, durch den beim Anwählen der Menüoption ein nicht-modales Erfassungsfenster geöffnet wird:

```
void CMainFrame::OnErfassungProfessor()  
{ ProfessorView* profDlg = new ProfessorView;  
  profDlg->Create(IDD_ProfessorView);  
}
```

- Verfahren Sie analog für die Menüoption *Listen/Professor* und geben Sie hier als Code ein:

```
void CMainFrame::OnListeProfessor()  
{ ProfessorenView* profDlg = new ProfessorenView;  
  profDlg->Create(IDD_ProfessorenView);  
}
```

- Tragen Sie die notwendigen *Header*-Dateien in die Dateien ein und führen Sie dann die Anwendung aus, um sich von der Funktionsfähigkeit zu überzeugen. Sie sollten jetzt über die definierten Menüoptionen die entsprechenden Fenster öffnen können.
- Analog verbinden Sie später die anderen Menüoptionen mit den Fenstern.

### 2.6 Programmieren der Schaltflächen in den Fenstern

Die Schaltflächen sollen folgende Wirkung besitzen:

- Im Erfassungsfenster schließen *Ok* und *Abbrechen* das Fenster, *Übernehmen* hat im Prototyp keine Wirkung und mit *Liste* kann das zugehörige Listenfenster geöffnet werden.
- Im Listenfenster öffnen die Schaltflächen *Neu* und *Ändern* das Fenster *ProfessorView*. Die Schaltfläche *Schließen* schließt das Listenfenster, während *Löschen* im Prototyp keine Wirkung hat.

- Bei den Schaltflächen *Ok* und *Abbrechen* brauchen Sie nichts zu tun, sondern können die vordefinierte Funktionalität übernehmen. Beim späteren Entwurf der Benutzungsoberfläche müssen Sie die geerbten Operationen jedoch entsprechend überschreiben.

- Die Schaltfläche *Übernehmen* besitzt im Prototyp keine Wirkung.

Klassen-Assistent  
ProfessorView

- Für die Schaltfläche *Liste* ist eine **Operation** zu erstellen. Rufen Sie in der Dialog-Ressource *ProfessorView* den Klassen-Assistenten auf. Wählen Sie: Objekt-ID = IDC\_Liste und Nachricht = BN\_CLICKED. Erstellen Sie mit *Funktion hinzufügen* die neue Member-Funktion *OnListe()*, für die folgender Code einzugeben ist:

```
void ProfessorView::OnListe()  
{ ProfessorenView* profDlg = new ProfessorenView;  
  profDlg->Create(IDD_ProfessorenView);  
}
```

## 2.7 Realisieren der *one*-Richtung einer Assoziation Exkurs 1

- Auch für die Schaltfläche *Neu* ist eine **Operation** zu erstellen. Wählen Sie: Objekt-ID = IDC\_Neu und Nachricht = BN\_CLICKED und editieren die generierte Member-Funktion wie folgt:

```
void ProfessorenView::OnNeu()  
{ ProfessorView* profDlg = new ProfessorView;  
  profDlg->Create(IDD_ProfessorView);  
}
```

- Erstellen Sie analog dazu für die Schaltfläche *Ändern* eine **Operation**, die den gleichen Code besitzt wie OnNeu(). Hier hätten Sie beide Schaltflächen auch auf eine einzige Funktion abbilden können. Da wir den Prototyp aber später zur Benutzungsoberfläche weiterentwickeln wollen, nehmen wir zwei getrennte Funktionen.
- Für die Schaltfläche *Schließen* definieren Sie die folgende **Operation**:

```
void ProfessorenView::OnSchliessen()  
{ DestroyWindow(); //löscht das Fenster  
  delete this; //löscht das Fensterobjekt  
}
```

- Bei der Schaltfläche *Löschen* tragen Sie nichts ein.
- Führen Sie dann die Anwendung wieder aus und überprüfen Sie das Ergebnis.

## 2.7 Realisieren der *one*-Richtung einer Assoziation

Hier betrachten wir die Klasse *Diplomarbeit* (Abb. 2-5) und die *one*-Richtung der Assoziation von *Diplomarbeit* zu *Professor*.

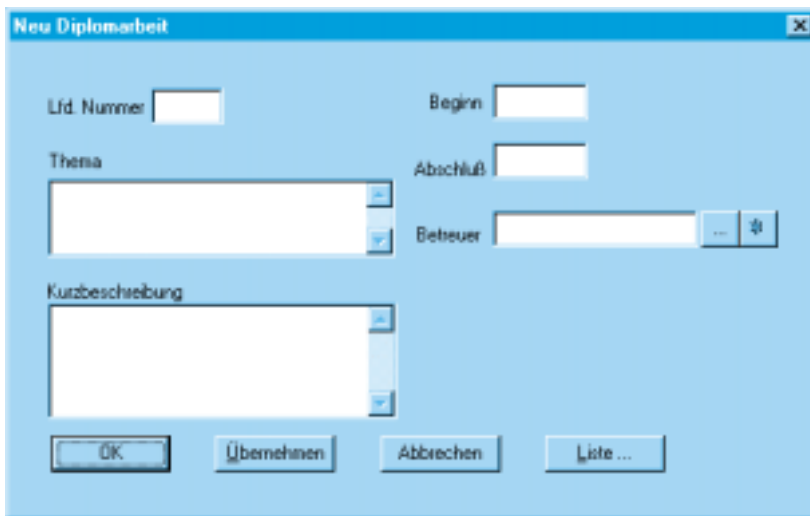


Abb. 2-5:  
Erfassungsfenster  
für Diplomarbeit  
(ohne Assoziation  
zu Student)

## Exkurs 1 2 Prototyp mit Dialogfenstern

- Erstellen Sie analog zu oben Erfassungs- und Listenfenster. Sie ersparen sich Arbeit, wenn Sie von einer vorhandenen Dialog-Resource ausgehen und diese mittels *Kopieren* und *Einfügen* duplizieren und umbenennen. Außerdem läßt sich dadurch leichter die Einheitlichkeit der Fenster erreichen.
- Anschließend verbinden Sie die Menüoptionen mit den Fenstern und programmieren die Schaltflächen in den neuen Fenstern.

### Eingabefeld und Schaltflächen definieren

Die *one*-Assoziation zu Professor wird durch ein Eingabefeld IDC\_Betreuer und zwei Schaltflächen realisiert (Abb. 2-5).

Eigenschaften  
DiplomarbeitView

- Die *Link*-Schaltfläche IDC\_AsozProfessor ist eine normale – kleine – Schaltfläche, die mit »...« beschriftet wird.
- Die *Neu*-Schaltfläche IDC\_NeuProfessor wird mit einem Piktogramm versehen (Kontrollkästchen Symbol = Ein).
- Dieses **Piktogramm** wird wie folgt realisiert: Erstellen Sie im *Resource View* mit *Icon/Einfügen.../Neu* ein neues Piktogramm (32 x 32) und zeichnen Sie in die Mitte einen kleinen gelben Stern. Öffnen Sie das Eigenschaftsfenster durch Anklicken in der Baumstruktur und benennen Sie dieses Piktogramm mit IDI\_Neulcon.

Klassen-Assistent  
DiplomarbeitView

- Definieren Sie für die *Neu*-Schaltfläche die **Member-Variable** m\_ButtonNeuProfessor.
- Damit das Piktogramm auf die Schaltfläche geschrieben wird, müssen Sie für die Klasse DiplomarbeitView die **Operation** OnInitDialog() erstellen (Objekt-ID = DiplomarbeitView, Nachricht = WM\_INITDIALOG) und folgenden Code eintragen:

```
BOOL DiplomarbeitView::OnInitDialog()
{
    ...
    HICON icon = App()->LoadIcon (IDI_Neulcon);
    m_ButtonNeuProfessor.SetIcon (icon);
    ...
}
```

### Zugriff auf das Anwendungsobjekt

Wir verwenden die **globale Funktion** App(), die einen Zeiger auf das einzige Objekt der Anwendungsklasse zurückgibt. Auf diese Weise können Sie überall auf dieses Objekt zugreifen. Fügen Sie dazu in der Datei Diploma.h ein:

```
CDiplomaApp* App();
```

In der Datei Diploma.cpp fügen Sie ein:

```
CDiplomaApp* App()
{ return (CDiplomaApp*) AfxGetApp();
}
```

### Erstellen des Auswahlfensters als Dialog-Ressource

- Erstellen Sie die Dialog-Ressource *ProfessorAuswahl* entsprechend Abb. 2-6.
- Für die beiden Schaltflächen können Sie die Voreinstellungen übernehmen. Ersetzen Sie nur die Bezeichnung *Ok* durch *Auswählen*.

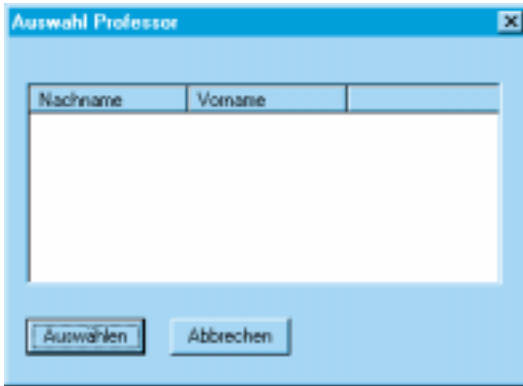


Abb. 2-6:  
Auswahlfenster für  
Professor

### Realisieren der Dialog-Ressource durch eine Klasse

- Mit dem Klassen-Assistenten wird die Klasse *ProfessorAuswahl* erstellt, die von der Basisklasse *CDiialog* abgeleitet wird.
- Definieren Sie die **Member-Variable** *m\_Liste* für das Listenelement.
- Erstellen Sie eine **Operation** *Onlinetialog()*, in die Sie folgenden Code eintragen:

Klassen-Assistent  
*ProfessorAuswahl*

```

BOOL ProfessorAuswahl::Onlinetialog()
{
    ...
    m_Liste.InsertColumn (0, "Nachname", LVCFMT_LEFT, 100);
    m_Liste.InsertColumn (1, "Vorname", LVCFMT_LEFT, 100);
    ...
}
    
```

## 2.8 Realisieren der *many*-Richtung einer Assoziation

### Listenelement und Schaltflächen definieren

Die *many*-Assoziation vom Professor zu den zugehörigen Diplomarbeiten wird durch ein Listenelement und drei Schaltflächen realisiert.

- Zuerst erweitern wir das zuvor erstellte Erfassungsfenster eines Professors um die Liste aller betreuten Diplomarbeiten (Abb. 2-7).
- Dann werden analog zur oben beschriebenen *one*-Richtung die beiden Schaltflächen »...« und »\*« verwendet.

## Exkurs 1 2 Prototyp mit Dialogfenstern

Abb. 2-7:  
Vollständiges  
Erfassungs-  
fenster für  
Professor

- Zusätzlich wird eine Lösch-Schaltfläche benötigt, für die ein zweites Piktogramm (*icon*) mit einem roten »X« in der Mitte zu zeichnen ist. Sie wird mit `IDI_LoeschlIcon` benannt.
- Erstellen Sie die **Dialog-Ressource** `DiplomarbeitAuswahl` und leiten Sie die zugehörige **Klasse** `DiplomarbeitAuswahl` von `CDialog` ab.
- Definieren Sie auch hierfür die **Member-Variable** `m_Liste` und die folgende **Operation**:

Klassen-Assistent  
ProfessorAuswahl

```
BOOL ProfessorAuswahl::OnInitDialog()  
{  
    ...  
    m_Liste.InsertColumn(0, "Nachname", LVCFMT_LEFT, 100);  
    m_Liste.InsertColumn(1, "Vorname", LVCFMT_LEFT, 100);  
}
```

### 2.9 Programmieren der Schaltflächen für die Assoziation

Klassen-Assistent  
DiplomarbeitView

- Im Erfassungsfenster von `Diplomarbeit` soll die *Link*-Schaltfläche »...« das modale Auswahlfenster von `Professor` öffnen. Definieren Sie (Objekt-ID = `IDC_AssozProfessor`, Nachricht = `BN_CLICKED`) folgende Operation:

```
void DiplomarbeitView::OnAssozProfessor()  
{  
    ProfessorAuswahl * auswahlDlg = new ProfessorAuswahl;  
    auswahlDlg->DoModal();  
}
```

- Damit die Neu-Schaltfläche »\*« das nicht-modale Erfassungs-fenster von Professor öffnet, ist folgende Operation zu erstellen (Objekt-ID = IDC\_NeuProfessor, Nachricht = BN\_CLICKED):

```
void DiplomarbeitsView::OnNeuProfessor()
{
    ProfessorView* profDlg = new ProfessorView;
    profDlg->Create(IDD_ProfessorView);
}

```

- Die Lösch-Schaltfläche besitzt im Prototyp noch keine Wirkung.

### 2.10 Abbildung des vollständigen OOA-Modells auf die Benutzungsoberfläche

- Vervollständigen Sie den Prototyp, indem Sie das Erfassungs-fenster für Diplomarbeit gemäß Abb. 2-8 erweitern.

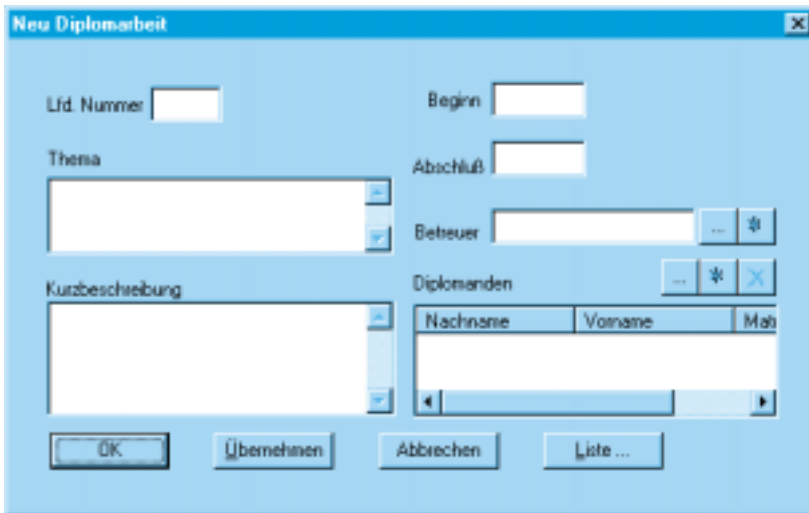


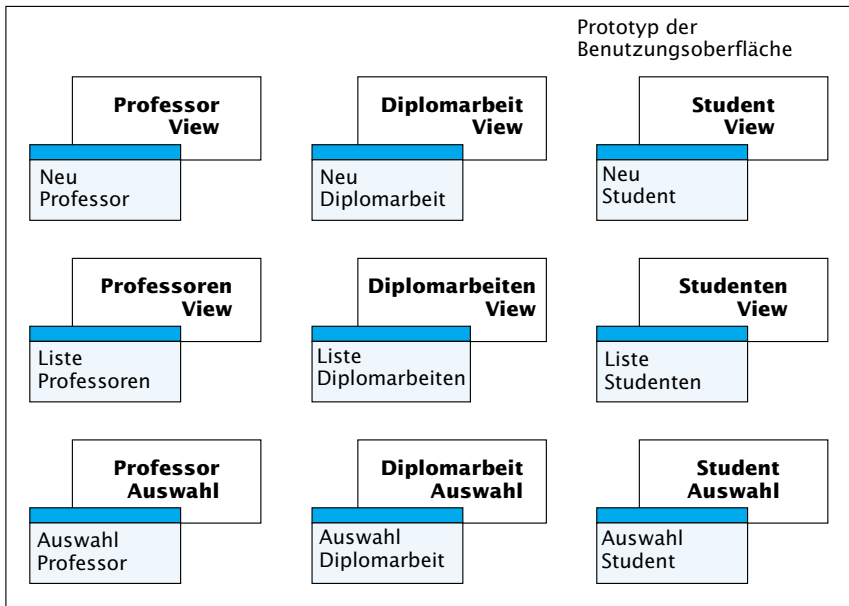
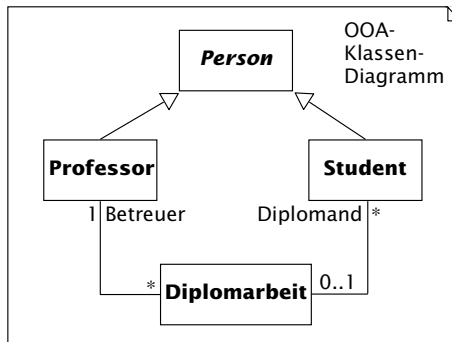
Abb. 2-8: Vollständiges Erfassungs-fenster für Diplomarbeit

- Erstellen Sie die Dialog-Ressource für das Erfassungs-fenster von Student entsprechend Abb. 2-9, desweiteren die Dialog-Ressourcen für Listen- und Auswahl-fenster.
- Dann führen Sie analog zu oben die restlichen Schritte durch. Wie Abb. 2-10 zeigt, werden für jede der drei OOA-Klassen Professor, Diplomarbeit und Student jeweils ein Erfassungs- und ein Listen-fenster erzeugt, von denen jedes durch eine Klasse realisiert wird. In diesem Prototyp wurden alle Assoziationen direkt durch Dialoge realisiert. Daher kommen drei modale Auswahl-fenster hinzu, von denen ebenfalls jedes durch eine Klasse realisiert wird. Diese Assoziationen könnten alternativ auch durch entsprechende Operationen (z.B. Anmelden eines Studenten für eine Diplomarbeit) realisiert werden.

## Exkurs 1 2 Prototyp mit Dialogfenstern

Abb. 2-9:  
Erfassungs-  
fenster für  
Student

Abb. 2-10:  
Transformation  
der OOA-Klassen in  
Fenster und GUI-  
Klassen



- 1 Menüs erstellen.
- 2 Im Ressourcen-Editor sind zu erstellen bzw. zu kopieren
  - Erfassungsfenster und
  - Listenfenster.
- 3 *one*-Assoziation durch ein Eingabefeld und die Schaltflächen »...« und »\*« darstellen, *many*-Assoziation durch ein Listenelement und die Schaltflächen »...«, »\*« und »x« darstellen.
- 4 Piktogramme für die Schaltflächen »\*« und »x« der Assoziationen erstellen oder kopieren.
- 5 Alle Interaktionselemente – mit Ausnahme von *Static* – sorgfältig benennen.
- 6 Je eine Klasse – von *CDi al og* abgeleitet – mit dem Klassen-Assistenten erstellen für
  - Erfassungsfenster und
  - Listenfenster.
- 7 Namenssystematik bei der Benennung der Klassen beachten: *Kl asse* – *Kl asseVi ew* – *Kl assenVi ew* – *Kl asseAuswahl* .
- 8 Globale Funktion *App()* für Zugriff auf Anwendungsobjekt realisieren.
- 9 Member-Variablen für diejenigen Interaktionselemente eintragen, die im Programm des Prototyps angesprochen werden, d.h. hier für
  - Kombinationsfelder,
  - Listenelemente (*list view controls*),
  - Schaltflächen »\*« und »x« der Assoziation.
- 10 Operation *Onl ni ti tDi al og()* erzeugen und Code erstellen für die
  - Bemalung der Schaltflächen »\*« und »x« mit Symbolen,
  - Initialisierung mit Testdaten und
  - Spaltenüberschriften der Listenelemente.
- 11 Menüoptionen mit der Erfassungs- und Listenklasse verbinden (C++).
- 12 Programmieren
  - der Schaltfläche *Liste* im Erfassungsfenster und
  - der Schaltflächen *Neu*, *Ändern* und *Schließen* im Listenfenster.
- 13 Auswahlfenster (modales Dialogfenster) im Ressourcen-Editor erstellen bzw. kopieren.
- 14 Für das Auswahlfenster mit dem Klassen-Assistenten von *CDi al og* eine Klasse ableiten.
- 15 Für die Klasse des Auswahlfensters
  - Member-Variable für die Liste und
  - Operation *Onl ni tDi al og()* erzeugen und
  - Spaltenüberschriften in C++ programmieren.
- 16 Programmieren der Schaltflächen der Assoziation, d.h.
  - »...« öffnet modales Auswahlfenster

**Checkliste zum Erstellen des Prototyps der Benutzungsoberfläche (Dialogfenster)**



## 3 Prototyp mit MDI-Unterfenstern

In diesem Kapitel wird ein Prototyp der Benutzungsoberfläche unter Verwendung von MDI-Unterfenstern erstellt. Diesen Prototyp DiplomaMDI finden Sie auf der CD im Verzeichnis *DiplomaMDI*.



### 3.1 Arbeitsbereich anlegen

- *Datei/Neu* wählen und unter Projekte *MFC-Anwendungs-Assistent (exe)* wählen. Als Projektname wird *Diploma* eingetragen. Es wird automatisch ein gleichnamiges Verzeichnis angelegt. Alle anderen Standardeinstellungen belassen und Ok drücken.
- Schritt 1: Mehrere Dokumente (MDI).
- Schritt 2: keine Datenbankunterstützung.
- Schritt 3: keine Unterstützung.
- Schritt 4: Statusleiste und 3D-Steuerelemente anklicken, den Rest ausschalten.
- Schritt 5: Statisch verknüpfte Bibliothek anklicken.
- Schritt 6: Fertigstellen.
- Es werden automatisch folgende Klassen generiert: CDiplomaApp, CMainFrame, CChildFrame, CDiplomaDoc, CDiplomaView.
- Das generierte Rahmenprogramm sollten Sie zunächst mit *Erstellen/Ausführen* laufen lassen, um sich über den Leistungsumfang des Microsoft Visual Studios klar zu werden.

In den nächsten Schritten werden wir die standardmäßig generierte Oberfläche an unsere Anforderungen anpassen.

### 3.2 Erstellen der Menüs

Bei der MDI-Anwendung sind zwei Menübalken zu erstellen:

- IDR\_MAINFRAME für das Anwendungsfenster (Abb. 3-1) und
- IDR\_DIPLOMTYPE für die Unterfenster (Abb. 3-2).

Abb. 3-1:  
Menübalken für  
das Anwendungs-  
fenster

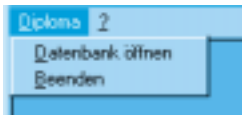


Abb. 3-2: Menü-  
balken für die  
Unterfenster



Eigenschaften  
CMainFrame

- Öffnen Sie im *ResourceView Menu/IDR\_MAINFRAME*.
- Den Menüpunkt *Datei* ändern Sie in *Diploma (&Diploma)* und löschen alle Optionen bis auf *Beenden* und die Trennlinie davor.

- Fügen Sie über der Trennlinie die folgende Menüoption ein:
 

ID	Beschriftung
ID_Datenbank_Oeffnen	&Datenbank öffnen
- Löschen Sie die anderen Menüpunkte bis auf "?".
- Öffnen Sie nun Menu/IDR\_DIPLOMTYPE. Ändern Sie den Menüpunkt *Datei* analog in *Diploma*. Löschen Sie hier die Optionen bis auf *Beenden*. Löschen Sie ebenfalls die weiteren Menüpunkte bis auf »?«.
- Erstellen Sie zwei weitere Menüpunkte : Listen (&Listen) und Erfassung (&Erfassung).
- Fügen Sie folgende Optionen in diese zwei Punkte ein:
 

ID	Beschriftung
ID_Liste_Professor	&Professor...
ID_Liste_Diplomarbeit	&Diplomarbeit...
ID_Liste_Student	&Student...
ID_Erfassen_Professor	&Professor...
ID_Erfassen_Diplomarbeit	&Diplomarbeit...
ID_Erfassen_Student	&Student...

### 3.3 Realisieren des Erfassungsfensters

#### Dialog-Ressource erstellen

- Wenn Sie zuvor den Prototyp mit Dialogfenstern erstellt haben, dann können Sie die Dialog-Ressourcen und die Piktogramme kopieren. In diesem Fall müssen Sie folgende Änderungen durchführen:
  - Schaltfläche IDCANCEL in IDC\_Abbrechen umbenennen.
  - Im allen MDI-Fenstern unter Eigenschaften *Format = untergeordnet* und *Rand = keine* einschalten, alles andere ausschalten.
- Andernfalls müssen Sie die Ressourcen neu gestalten (Abb. 3-3). Für die Schaltflächen wählen Sie folgende Beschriftung:
 

ID	Beschriftung	Optionen
IDOK	OK	
IDC_Uebernehmen	&Übernehmen	
IDC_Abbrechen	Abbrechen	
IDC_Liste	&Liste...	
IDC_AssozDiplomarbeit	...	
IDC_NeuDiplomarbeit	(ohne)	Symbol anklicken
IDC_LoeschDiplomarbeit	(ohne)	Symbol anklicken
- Für die Editierfelder, *Dropdown*-Kombinationsfelder und das Listenelement (*list view control*) wählen Sie folgende Namen: IDC\_Vorname, IDC\_Nachname, IDC\_Telefon, IDC\_Email, IDC\_COMBOFachgebiet, IDC\_COMBOTitel, IDC\_Raum, IDC\_Kuerzel, IDC\_LISTDiplomarbeit.
- Für das Listenelement wählen Sie die Ansicht *Bericht*.

Dialog-Ressourcen kopieren

Eigenschaften ProfessorView

## Exkurs 1 3 Prototyp mit MDI-Unterfenstern

Abb. 3-3:  
Erfassungsfenster für  
Professor

- Die statischen Textfelder und Gruppfelder haben alle standardmäßig die ID = IDC\_STATIC, die Sie übernehmen.
- Ändern Sie nun die Eigenschaften des Fensters: *IDD\_Professor View*, Format = untergeordnet, Rand = keine, alles andere ausgeschaltet.
- Erstellen Sie im *Resource View* die beiden **Piktogramme** für die Neu- und Lösch-Schaltflächen, die Sie *IDI\_NeuIcon* und *IDI\_LoeschIcon* nennen.

### Realisierung der Dialog-Ressource durch eine Klasse

Klassen-Assistent  
ProfessorView

- Mit Hilfe des Klassen-Assistenten erstellen Sie eine neue Klasse für diese Dialog-Ressource, die Sie ebenfalls mit *ProfessorView* benennen. Da es sich um ein MDI-Unterfenster handelt, wird *CFormView* als Basisklasse gewählt.
- Erstellen Sie mit Hilfe des Klassen-Assistenten **Member-Variablen** für die Interaktionselemente. Wählen Sie folgende Namen:
  - *Dropdown-Kombinationsfeld*: *m\_ComboTitel*, *m\_ComboFachgebiet* (Kategorie *Control*)
  - Schaltflächen: *m\_ButtonLoeschDA*, *m\_ButtonNeuDA*
  - Listenelement: *m\_ListeDA*
- Erstellen Sie mit dem Klassen-Assistenten für Objekt-ID = *ProfessorView* und Nachricht = *OnInitialUpdate* die **Operation** *OnInitialUpdate()* und geben Sie folgendem Code ein:

```

void ProfessorView::OnInitDialog()
{
    ...
    //Anpassung der Fenstergröße an vorgegebenen Inhalt
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit(FALSE);

    //Titel im MDI-Fenster setzen
    GetParentFrame()->SetWindowText ("Neu Professor");

    //Dropdown-Listefeld mit Testdaten füllen
    m_ComboFachgebiet.AddString ("Softwaretechnik");
    ...
    //Dropdown-Listefeld mit Testdaten füllen
    m_ComboTitel.AddString ("Dr. ");
    ...
    //Zuweisen der Piktogramme
    HICON icon = App()->LoadIcon (IDI_NeuIcon);
    m_ButtonNeuDA.SetIcon (icon);
    icon = App()->LoadIcon (IDI_LoeschIcon);
    m_ButtonLoeschDA.SetIcon (icon);

    //Überschriften für Listenelement der Diplomarbeiten
    m_ListeDA.InsertColumn (0, "Nr.", LVCFMT_LEFT, 40);
    m_ListeDA.InsertColumn (1, "Thema", LVCFMT_LEFT, 400);
}

```

- In einem MDI-Fenster kann der **Fenstertitel** nicht im Eigenschaftsdialog gesetzt werden, sondern diese Angabe muß über das Programm erfolgen, z.B.:  
Fenstertitel

```
GetParentFrame()->SetWindowText ("Neu Professor");
```
- Die automatisch generierte Klasse `CChildFrame` müssen Sie um folgende Member-Funktion ergänzen, um die geerbte Operation, die den Fenstertitel automatisch setzt, zu überschreiben.

```

void CChildFrame::OnUpdateFrameTitle (BOOL b)
{ //leer
}

```

#### Zugriff auf das Anwendungsobjekt

Wir verwenden die **globale Funktion** `App()`, die einen Zeiger auf das einzige Objekt der Anwendungsklasse zurückgibt. Auf diese Weise können Sie überall auf dieses Objekt zugreifen. Fügen Sie dazu in der Datei `Diploma.h` ein: Anwendungsobjekt

```
CDiplomaApp* App();
```

In der Datei `Diploma.cpp` fügen Sie ein:

```

CDiplomaApp* App()
{ return (CDiplomaApp*) AfxGetApp(); }

```

## Exkurs 1 3 Prototyp mit MDI-Unterfenstern

### 3.4 Realisieren des Listenfensters

#### Dialog-Ressource erstellen

Eigenschaften  
ProfessorenView

- Erstellen Sie nun analog das Listenfenster (Abb. 3-4). Wählen Sie für die Schaltflächen:

ID **Beschriftung**

IDNeu &Neu...

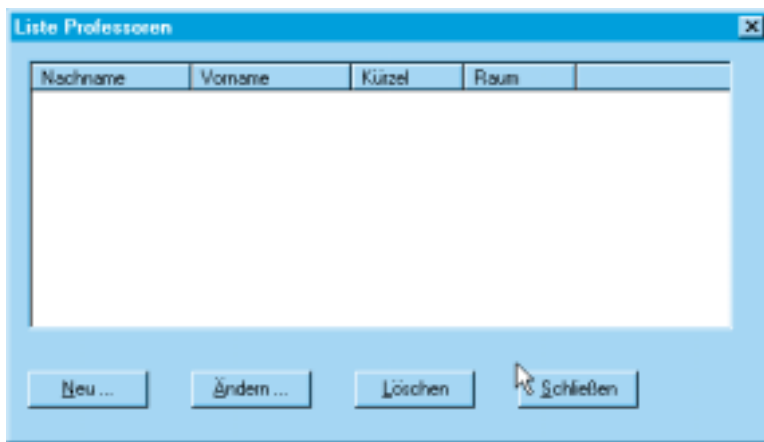
IDÄndern &Ändern...

IDLoeschen &Löschen

IDSchliessen &Schließen

- Das Listenelement erhält als ID = IDC\_LISTProfessor, als Ansicht wird *Bericht* eingestellt.
- Ändern Sie analog zum Erfassungsfenster die Eigenschaften des Fensters.

Abb. 3-4:  
Listenfenster für  
Professor



#### Realisieren der Dialog-Ressource durch eine Klasse

Klassen-Assistent  
ProfessorenView

- Erstellen Sie mit Hilfe des Klassen-Assistenten eine Klasse, die Sie ProfessorenView nennen und wählen Sie als Basisklasse CFormView.
- Definieren Sie die **Member-Variable** m\_Liste für das Listenelement.
- Erzeugen Sie die **Operation** OnInitUpdate(). Gehen Sie dabei analog zum Erfassungsfenster vor (Objekt-ID = ProfessorenView, Nachricht = OnInitUpdate). Fügen Sie folgenden Code ein:

```
void CProfessorenView::OnInitUpdate()  
{ ...  
    GetParentFrame()->RecalcLayout();  
    ResizeParentToFit(FALSE);  
  
    //Titel im MDI-Fenster setzen  
    GetParentFrame()->SetWindowText ("Liste Professoren");
```

```

m_Li ste.InsertColumn (0, "Nachname", LVCFMT_LEFT, 100);
m_Li ste.InsertColumn (1, "Vorname", LVCFMT_LEFT, 100);
m_Li ste.InsertColumn (2, "Kürzel", LVCFMT_LEFT, 70);
m_Li ste.InsertColumn (3, "Raum", LVCFMT_LEFT, 70);
}

```

### 3.5 Verbinden der Menüs und der Fenster

In diesem Schritt verbinden wir Menü und Fenster auf folgende Art:

- Die Menüoption *Erfassung/Professor* öffnet das Fenster *ProfessorView*.
- Die Schaltflächen *Ok* und *Abbrechen* schließen das Fenster.
- Die Menüoption *Listen/Professor* öffnet das Fenster *ProfessorenView*.
- Die Schaltfläche *Schließen* schließt es wieder.

#### Vorlagen anlegen

- Für die beiden erstellten Fenster müssen **Vorlagen** (*templates*) angelegt werden. Fügen Sie in der Klasse `CDiplomaApp` folgende `public`-Attribute (*Member-Variablen*) hinzu: Vorlagen deklarieren

```

CMultiDocTemplate* pProfessorenDocT;
CMultiDocTemplate* pProfessorDocT;

```

- Fügen Sie weiterhin für die Klasse `CDiplomaApp` folgende `public`-Operation (*Member-Funktionen*) hinzu:

```

CMainFrame* getMainFrame()

```

In der `cpp`-Datei implementieren Sie die Operation wie folgt:

```

CMainFrame* CDiplomaApp::getMainFrame()
{ return (CMainFrame*) m_pMainWnd;
}

```

- Die Vorlagen (*templates*) werden in der Operation `CDiplomaApp::InitInstance()` initialisiert. Fügen Sie im Anschluß an die generierten Zeile `»AddDocTemplate(pDocTemplate)«` ein: Vorlagen initialisieren

```

pProfessorenDocT = new CMultiDocTemplate(
    IDR_DIPLOMTYPE,
    RUNTIME_CLASS(CDiplomaDoc),
    RUNTIME_CLASS(CChildFrame),
    RUNTIME_CLASS(ProfessorenView));

```

```

pProfessorDocT = new CMultiDocTemplate(
    IDR_DIPLOMTYPE,
    RUNTIME_CLASS(CDiplomaDoc),
    RUNTIME_CLASS(CChildFrame),
    RUNTIME_CLASS(ProfessorView));

```

- Um zu vermeiden, daß das – leere – Standardfenster *CDiplomaView* geöffnet wird, löschen Sie in der gleichen Datei die `if`-Anweisung hinter dem Kommentar `»Verteilung der in der Befehlszeile angegebenen Befehle«`. Standardfenster unterdrücken

## Exkurs 1 3 Prototyp mit MDI-Unterfenstern

### Definieren der Menüoptionen

Klassen-Assistent  
CMainFrame

- Rufen Sie den Klassen-Assistenten für die Klasse CMainFrame auf.
- Wählen Sie für ID = ID\_Datenbank\_Oeffnen die Nachricht = Command und fügen Sie die entsprechende Funktion hinzu. Übernehmen Sie die vorgeschlagenen Namen. Tragen Sie in diese Operationen folgenden Code ein:

```
void CMainFrame::OnDatenbankOeffnen()  
{ //legt fest, daß beim Öffnen der Datenbank das Listenfenster  
  //von Professor angezeigt wird  
  App()->pProfessorenDocT->OpenDocumentFile (NULL);  
}
```

- Fügen Sie für die ID = ID\_Erfassen\_Professor (Nachricht = Command) die folgende Funktion hinzu:

```
void CMainFrame::OnErfassenProfessor()  
{ createViewToDoc (App()->pProfessorDocT); }
```

- Fügen Sie für ID = ID\_Liste\_Professor (Nachricht = Command) die folgende Funktion hinzu:

```
void CMainFrame::OnListeProfessor()  
{ createViewToDoc (App()->pProfessorenDocT); }
```

- Erweitern Sie die Klasse CMainFrame um die **public-Member-Funktion** createViewToDoc(). Diese Operation öffnet ein MDI-Fenster und liefert einen Zeiger auf ein CView-Objekt zurück. So können Sie später im Entwurf mit den CFormView-Klassen kommunizieren. Momentan spielt der Rückgabewert noch keine Rolle. Den exakten C++-Programmcode dieser Operation entnehmen Sie bitte der Klasse CMainFrame im Projekt von *DiplomaMDI*.

### 3.6 Programmieren der Schaltflächen in den Fenstern

- Im **Erfassungsfenster** schließen die Schaltflächen *OK* und *Abbrechen* das Fenster, *Übernehmen* hat im Prototyp keine Wirkung und mit *Liste* kann das Listenfenster geöffnet werden.
- Im **Listenfenster** öffnen die Schaltflächen *Neu* und *Ändern* das Fenster *ProfessorView*, die Schaltfläche *Schließen* schließt das Listenfenster, die Schaltfläche *Löschen* hat im Prototyp keine Wirkung.

Klassen-Assistent  
ProfessorView

- Erstellen Sie mit dem Klassen-Assistenten für die jeweilige Schaltflächen-ID und die Nachricht BN\_CLICKED neue Member-Funktionen, wobei Sie die unten angegebenen Namen wählen.
- Die Funktionen der Klasse **ProfessorView** werden wie hier abgedruckt mit Leben gefüllt:

```

void ProfessorView::OnOk()
{
    GetParent()->DestroyWindow();
}
void ProfessorView::OnAbbrechen()
{
    GetParent()->DestroyWindow();
}
void ProfessorView::OnListe()
{
    CMainFrame* mf = App()->getMainFrame();
    mf->createViewToDoc (App()->pProfessorenDocT);
}
    
```

- Für die Klasse **ProfessorenView** werden die Funktionen wie folgt mit Leben gefüllt:

Klassen-Assistent  
ProfessorenView

```

void ProfessorenView::OnNeu()
{
    CMainFrame* mf = App()->getMainFrame();
    mf->createViewToDoc (App()->pProfessorDocT);
}
void ProfessorenView::OnSchliessen()
{
    GetParent()->DestroyWindow();
}
void ProfessorenView::OnAendern()
{
    CMainFrame* mf = App()->getMainFrame();
    mf->createViewToDoc (App()->pProfessorDocT);
}
    
```

### 3.7 Abbildung weiterer Klassen auf die Benutzungsoberfläche

- Erstellen Sie analog zu oben Erfassungsfenster für Diplomarbeit (Abb. 3-5) und Student (Abb. 3-6). Desweiteren sind entsprechende Listenfenster zu erstellen.

Abb. 3-5: Erfassungsfenster für Diplomarbeit



## Exkurs 1 3 Prototyp mit MDI-Unterfenstern

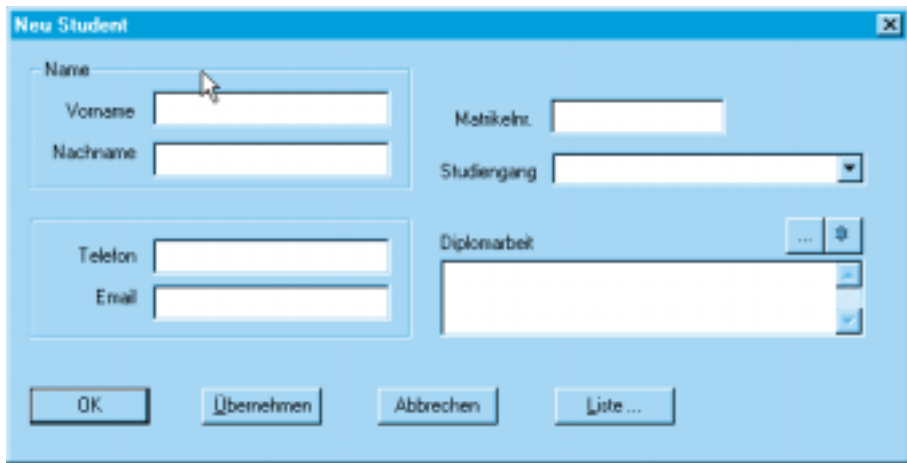


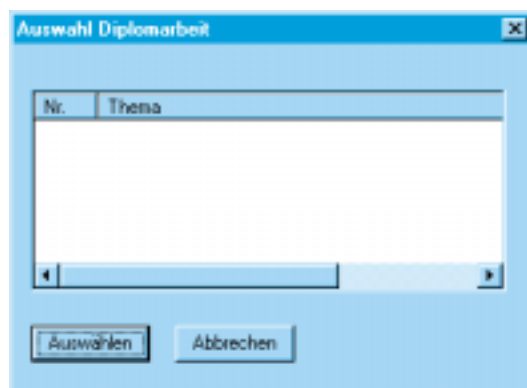
Abb. 3-6:  
Erfassungsfenster  
für Student

- Für jede Dialog-Ressource ist analog zu den Kapiteln 3.3 und 3.4 eine Klasse von `CFormView` abzuleiten.
- Erstellen Sie analog zu Kapitel 3.5 Vorlagen (*templates*) für die Fenster von Diplomarbeit(en) und Student(en) und verbinden Sie die Menüoptionen mit den Fenstern.

### 3.8 Realisieren der Assoziation

- Für die Realisierung der Assoziation zwischen Professor und Diplomarbeit benötigen Sie ein Auswahlfenster. Dieses Fenster realisieren Sie auch bei einer MDI-Anwendung als modales Dialogfenster.
- Gehen Sie vor, wie in den Kapiteln 2.8 bis 2.10 beschrieben wurde.

Abb. 3-7:  
Auswahlfenster für  
Diplomarbeit



- Damit die Schaltflächen »...« und »\*« im Erfassungsfenster von *ProfessorView* die gewünschte Wirkung haben, muß folgender Code eingegeben werden:

```
void ProfessorView::OnAssozDiplomarbeit()
{
    DiplomarbeitAuswahl * auswahl = new DiplomarbeitAuswahl();
    auswahl->DoModal();
}
void ProfessorView::OnNeuDiplomarbeit()
{
    CMainFrame* mf = App()->getMainFrame();
    mf->createViewToDoc(App()->pDiplomarbeitDoc());
}
```

- Erstellen Sie für die Klasse *DiplomarbeitView* analog dazu die Operationen *OnAssozProfessor()* und *OnNeuProfessor()*.
- Analog realisieren Sie die Assoziation zwischen *Diplomarbeit* und *Student*.

- 1 Menüs für *IDR\_MAINFRAME* und *IDR\_DIPLOMTYPE* erstellen.
- 2 Im Ressourcen-Editor sind zu erstellen bzw. zu kopieren
  - Erfassungsfenster und
  - Listenfenster.
- 3 *one*-Assoziation durch ein Eingabefeld und die Schaltflächen »...« und »\*« darstellen, *many*-Assoziation durch ein Listenelement und die Schaltflächen »...«, »\*« und »x« darstellen.
- 4 Piktogramme für die Schaltflächen »\*« und »x« von Assoziationen erstellen oder kopieren.
- 5 Alle Interaktionselemente – mit Ausnahme von *Static* – sorgfältig benennen.
- 6 Mit dem Klassen-Assistenten ist von *CFormView* je eine Klasse abzuleiten für
  - Erfassungsfenster und
  - Listenfenster.
- 7 Namenssystematik bei der Benennung der Klassen beachten: Klasse – KlasseView – KlassenView – KlasseAuswahl.
- 8 Globale Funktion *App()* für Zugriff auf Anwendungsobjekt realisieren.
- 9 Member-Variablen für diejenigen Interaktionselemente eintragen, die im Programm des Prototyps angesprochen werden, d.h. hier
  - Kombinationsfelder,
  - Listenelemente,
  - Schaltflächen »\*« und »x« der Assoziation.
- 10 Operation *OnInitUpdate()* erzeugen und Code für die
  - Bemalung der Schaltflächen »\*« und »x« mit Piktogrammen,
  - Initialisierung mit Testdaten und
  - Spaltenüberschriften der Listenelemente.

**Checkliste zum Erstellen des Prototyps der Benutzungsoberfläche (MDI-Unterfenster)**

## Exkurs 1 3 Prototyp mit MDI-Unterfenstern

- 11** Menüoptionen mit der Erfassungs- und Listenklasse verbinden:
  - Erstellen von Vorlagen (*templates*),
  - Definieren der Menüoptionen,
  - Operation `CMai nFrame:: createVi ewTodoc()` einfügen.
- 12** Programmieren
  - der Schaltfläche *Liste* im Erfassungsfenster und
  - der Schaltflächen *Neu*, *Ändern* und *Schließen* im Listenfenster.
- 13** Auswahlfenster (modales Dialogfenster) im Ressourcen-Editor erstellen bzw. kopieren.
- 14** Für Auswahlfenster mit dem Klassen-Assistenten von `CDi al og` eine Klasse ableiten.
- 15** Für die Klasse des Auswahlfensters
  - Member-Variable für Liste und
  - Operation `Onl ni ti al Update()` erzeugen und
  - Spaltenüberschriften in C++ programmieren.
- 16** Programmieren der Schaltflächen der Assoziation, d.h.
  - »...« öffnet Auswahlfenster
  - »\*« öffnet Erfassungsfenster.

## Exkurs 2

### Realisierung der Datenhaltung mit dem objektorientierten Datenbanksystem Poet für C++



- Klassen mittels Poet persistent machen können.
- Vererbungsstrukturen mit Poet realisieren können.
- Assoziationen mit Poet realisieren können.
- Für die Fallstudie Diploma die Datenhaltung mit Poet realisieren können.

anwenden



- Um diesen Exkurs erfolgreich zu bearbeiten, müssen Sie die Lehreinheiten zu den objektorientierten Konzepten der Analyse (Kapitel 2) und des Entwurfs (Kapitel 6) und die Grundlagen objektorientierter Datenbanksysteme (Kapitel 8.4 bis 8.7) durchgearbeitet haben.
- Da sich dieser Exkurs direkt auf die Fallstudie bezieht, sind aus dem Anhang 1 die Kapitel 1, 2 und 5 Voraussetzung.
- Weiterhin müssen Sie Grundkenntnisse in C++ besitzen.
- Für diesen Exkurs werden das Microsoft Developer Studio C++ 5.0 und die objektorientierte Datenbank Poet 5.0 verwendet.

- i 1 [Verwalten von Projekten](#) 446
- 2 [Erstellen einer einfachen Klasse](#) 448
- 2.1 [Schemadeklaration für eine Klasse](#) 448
- 2.2 [Speichern eines Objekts](#) 449
- 2.3 [Zugriff auf die Klassenextension](#) 450
- 2.4 [Selektion von Objekten](#) 451
- 2.5 [Öffnen und Schließen der Datenbank](#) 451
- 3 [Realisieren der Vererbung](#) 452
- 4 [Realisieren von Assoziationen](#) 454
- 4.1 [Realisierung mit \*pointer\*-Referenzen](#) 454
- 4.2 [Realisierung mit \*ondemand\*-Referenzen](#) 457
- 4.3 [Realisierung von \*many\*-Assoziationen](#) 459

Dieser Exkurs hat *nicht* das Ziel, Ihnen eine Einführung in die objektorientierte Datenbank Poet zu geben, sondern soll anhand von Poet zeigen, wie die objektorientierten Konzepte durchgängig von der Analyse bis zur Implementierung angewendet werden können. Lesern, die sich intensiver mit Poet beschäftigen wollen, empfehle ich die Originalliteratur /Poet 97/.

### 1 Verwalten von Projekten

Ein Projekt wird sowohl mit der Poet- als auch mit der Compiler-Entwicklungsumgebung bearbeitet.

- Erstellen Sie mit Hilfe des Explorers ein **Unterverzeichnis**. Nennen Sie es z.B. `BENUTZER\DIPLOMA`. Dieses Verzeichnis benutzen Sie sowohl bei Poet als auch beim Compiler als Projektverzeichnis.
- Die **Schemadeklaration** erfolgt in einer erweiterten Form von C++. Sie besteht aus einer oder mehreren Dateien mit der Erweiterung `hcd`. Diese **hcd-Dateien** können Sie mit einem beliebigen Editor erstellen.

#### Neues Projekt in Poet anlegen

- Mit *File/New Workbook* einen neuen Arbeitsbereich anlegen. Hier müssen Sie den Pfad zum Projektverzeichnis angeben. Nennen Sie die Datei `workbook.ptw`. (z.B. `BENUTZER\DIPLOMA\workbook.ptw`)
- Menüoption *Project/New* aufrufen. In der erscheinenden Dialogmaske müssen Sie den vorgeschlagenen Pfad nur bestätigen.
- Mit *Project/Add Item* die vorhandene `hcd`-Datei in das Projekt einfügen. Anschließend den Dialog mit *Abbrechen* verlassen.
- Unter *Options/Directories* die Pfade für die *Include*-Dateien des Compilers und von Poet einstellen, z.B. `».;COMPILER\include;POET\inc«`. Geben Sie einen existierenden Pfad für das temporäre Verzeichnis an.
- Unter *Options/PTXX* evtl. Compiler und Verzeichnis für Datenbank angeben.
- Menüoption *Project/Save* aufrufen.
- Mit *Builder/Build All* den Poet-Präprozessor (PTXX) starten. Wenn kein Fehler auftrat, erscheint die Meldung `»created database«` bzw. – bei vorhandener Datenbank – `»updated data-base«`.

#### Existierende Datenbank in Poet manipulieren

- Mit *File/Open Database* kann die vom Präprozessor generierte Datenbank geöffnet werden. Im Verzeichnis `DIPLOMA` wird das Unterverzeichnis `base` generiert, in dem sich die Datei `objects.dat` befindet. Wählen Sie dann z.B.

Host: LOCAL (bedeutet *single user mode*)

Database: BENUTZER\DIPLOMA\base\objects.dat

- Mit *Objects/Browse Extent* können Sie den Inhalt der Datenbank ansehen und Attribute verändern.
- Klicken Sie dazu in der Baumstruktur auf eine Klasse Ihrer Anwendung. Dann sehen Sie rechts die Objekte dieser Klasse, falls welche vorhanden sind.
- Durch Klicken mit der rechten Maustaste erscheint ein *pop-up*-Menü, das unter anderem eine Funktion zum Anlegen neuer Objekte für die gewählte Klasse enthält. Ein neues Objekt der Klasse wird am Ende der Liste angefügt.
- Ändern können Sie ein Objekt durch einen Doppelklick auf das Objekt in der Liste, worauf sich ein Fenster mit der Objektansicht öffnet. Klicken Sie mit der rechten Maustaste auf ein Attribut und wählen Sie *Edit*. Wenn alle Änderungen durchgeführt sind, speichern Sie das Objekt mit *Store*.
- Zum Löschen klicken Sie in der Listenansicht der Objekte mit der rechten Maustaste auf ein Objekt und wählen Sie *Delete Object*.

### Vorhandenes Projekt in Poet öffnen

Existiert das Projekt, dann kann es mit *File/Open Workbook* jederzeit geöffnet werden.

### Weiterbearbeiten des Projekts mit dem Compiler

Anschließend wird das Projekt mit dem entsprechenden Compiler bearbeitet. Wir verwenden hier den Microsoft-Developer C++ 5.0.

Legen Sie ein neues Projekt (Win32 Console Application) an. Achten Sie auf folgende Einstellungen:

- In *Extras/Optionen/Verzeichnisse* fügen Sie unter *Include-Dateien* den Pfad zu den Include-Dateien von Poet hinzu, z.B.: POET\inc.
- Analog fügen Sie unter *Bibliotheksdateien* den Pfad zu den *lib*-Dateien von Poet hinzu, z.B. POET\lib.
- In *Projekt/Einstellungen/C++* wählen Sie Kategorie = Präprozessor und geben unter *Zusätzliche Include-Verzeichnisse* den Punkt (.) als aktuelles Verzeichnis an.
- Desweiteren müssen Sie in *Projekt/Einstellungen/Linker* in den Optionen im Feld *Objekt-Bibliothek-Module* folgende Ergänzungen eintragen:  
für Release-Version: POET\lib\pt5nv5.lib (gewählte Konfiguration)  
für Debug-Version: POET\lib\pd5nv5.lib
- Wählen Sie in *Erstellen/Aktive Konfiguration festlegen: Win32 release*.
- Beachten Sie bitte auch die Hinweise in der *readme*-Dateien der Beispielprogramme.

## Exkurs 2 1 Verwalten von Projekten

Wir setzen unsere Fallstudie *Diploma* in 3 Schritten in eine Poet-Anwendung um:

- Erstellen einer einfachen Klasse *Person* (Verzeichnis *Diploma1*).
- Erstellen der Vererbungsstruktur mit den Unterklassen *Student* und *Professor* (*Diploma2*).
- Hinzufügen der Klasse *Diplomarbeit* mit den Assoziationen (*Diploma3*).

## 2 Erstellen einer einfachen Klasse

Im Rahmen des ersten Schritts wird erklärt, wie

- die Schemadeklaration für eine Klasse in erweitertem C++ erstellt wird,
- welche Dateien der Poet-Präprozessor generiert,
- wie die Datenbank geöffnet und geschlossen wird,
- wie Objekte gespeichert werden,
- wie alle Objekte einer Klasse ermittelt werden,
- wie einfache Selektionen durchgeführt werden.

### 2.1 Schemadeklaration für eine Klasse

Abb. 2-1:  
Klasse *Person*

Person	
# Vorname:	PtString
# Nachname:	PtString
# Telefon:	PtString
# Email:	PtString

Für die Klasse *Person* der Abb. 2-1 wird nachfolgende Schemadeklaration erstellt und als *hcd*-Datei gespeichert.

```
hcd-Datei für eine Klasse //Diploma.hcd
#include <ptstring.hxx>
persistent class Person
{protected:
    PtString    Vorname,
                Nachname,
                Telefon,
                Email;

public:
    Person();

    PtString getVorname () const;
    void setVorname (PtString vn);

    PtString getNachname () const;
    void setNachname (PtString nn);

    ... analoge get- und set-Operationen für alle Attribute ...
};
typedef cset <Person*> PersonCSet;
```

PtString ist der von Poet verwendete *String*-Typ. PersonCSet ist ein Typ, der später für Selektionen benötigt wird. Ein Objekt dieses Typs ist eine Menge, die Objekte der Klasse Person verwaltet ("C" steht für *compact*).

Obige *hcd*-Datei dient dem **PTXX-Präcompiler** von Poet als Eingabe. Er generiert daraus die folgenden Dateien:

■ *Diploma.hxx (class declaration header)*

Es handelt sich um die *Header*-Datei der Datenbank-Klassen für den Compiler. Das bedeutet unter anderem, daß das Schlüsselwort *persistent* entfernt wird und die Klasse Person von der Klasse PtObject abgeleitet wird. Diese Struktur ist im Datenbank-Browser sichtbar.

```
/**PERSISTENT */ class Person: public PtObject { ... };
```

■ *Diploma.cxx (class factory)*

Sie enthält für die in der *hcd*-Datei enthaltenen Klassen einige Operationen, die für die Verwaltung der Objekte benötigt werden.

■ *Diploma.ptx (class factory header)*

Sie enthält die *Query*-Klassen, *AllSet*-Klassen, *ondemand*-Klassen und *Set*-Klassen für die persistenten Klassen. Bei unserem Beispiel sind dies:

```
class PersonQuery: public PtQuery { ... };
class PersonAllSet: public PtAllSet { ... };
class PersonIndirectSet: public PtObjectSet { ... };
```

Außerdem erstellt der Präprozessor

- die Poet-Datenbank (Unterverzeichnis *base*) und
- das Datenbank-Schema im *Data Dictionary* (Unterverzeichnis *dict*).

## 2.2 Speichern eines Objekts

Jedes Objekt von Person »weiß«, wie es sich selbst in der Datenbank speichert, weil die Klasse Person dieses Wissen von der Klasse PtObject erbt. Beim Speichern wird das Objekt gleichzeitig in die Klassenextension (*extent*) PersonAllSet aufgenommen. Assign(), Store()

Folgende Operationen der Klasse PtObject werden verwendet:

- Assign()
 

Sie vergibt die Objektidentität (OID) und ist Voraussetzung für das Speichern von Objekten und bei *ondemand*-Referenzen für den Aufbau der Assoziationen.
- Store()
 

Sie speichert ein Objekt in der Datenbank. Ein Objekt kann beliebig oft gespeichert werden. POET erkennt vorhandene Objekte an der OID und legt *keine* neue Kopie an.

In diesem Exkurs geht es ausschließlich um die Realisierung der Datenhaltung mittels Poet. Daher wurde für die Kommunikation mit dem Benutzer des Programms eine einfache zeichenorientierte **Oberfläche** gewählt. Diese elementaren Ein-/Ausgabeoperationen werden in die Klasse Oberfläche integriert.



## Exkurs 2 2 Erstellen einer einfachen Klasse

```
class Oberflaeche
{protected:
    PtBase* pObjBase;           //Zeiger auf die Datenbank
    void erfassenPerson();
    void ListePersonen();
    void selektiereNachnamePersonen();
public:
    void auswahl();
};
```

Die **Operation** für das **Erfassen** von Personen implementieren wir wie folgt, wobei wegen der umständlich zu bedienenden zeichenorientierten Oberfläche nur ein Teil der Attribute eingegeben wird.

```
Erfassen einer Person void Oberflaeche::erfassenPerson()
{ Person* p = new Person;           //transientes Objekt der Klasse Person
  char name[30];
  cout <<"Nachname: "; cin >>name; p->setNachname ((PtString) name);
  cout <<"Vorname: "; cin >>name; p->setVorname ((PtString) name);
  p->Assign (pObjBase);             //OID für Poet zuweisen
  p->Store();                         //Objekt in Datenbank speichern
  delete p;                           //transientes Objekt löschen
}
```

### 2.3 Zugriff auf die Klassenextension

Für die Klasse Person wird durch Poet die Klasse PersonAllSet generiert. Das Objekt allPersons bildet den *Container*, der alle Objekte der Klasse Person enthält.

Folgende – geerbte – Operationen werden verwendet:

■ Get(p, i, PtSTART)

Die Operation liest das i-te Element aus der Menge allPersons, auf das über die Zeiger-Variable p zugegriffen werden kann. Die Operation hat den Ergebniswert = Null, wenn an der angegebenen Position ein Element gelesen wurde, d.h. der Wert von p definiert ist.

■ Unget(p)

Die Operation ist verantwortlich für das »Aufräumen« nach einem Get(), d.h. das Objekt p wird im Speicher gelöscht.

Die folgende Operation erstellt eine Liste aller Personen:

```
Liste aller Personen void Oberflaeche::ListePersonen()
{ cout <<endl << "Personen-Liste" << endl;
  PersonAllSet* allPersons = new PersonAllSet (pObjBase);
  Person* p;
  for (int i = 0; allPersons->Get (p, i, PtSTART) == 0; i++)
  { cout << (char*) p->getVorname() << " "
    << (char*) p->getNachname() <<endl;
    allPersons->Unget(p);
  };
  delete allPersons;
}
```

## 2.4 Selektion von Objekten

Objekte können nicht nur mittels OQL, sondern auch im C++-Programm selektiert werden.

Wir benötigen folgende Klassen, die alle vom Poet-Präprozessor generiert wurden:

- Klasse PersonAllSet, deren *Container* alle Personen verwaltet,
- Klasse PersonCSet, um die Menge aller Personen, auf die das Selektionskriterium zutrifft, zu speichern (Ergebnismenge),
- Klasse PersonQuery, um Selektionsbedingungen zu definieren.

Die folgende Operation ermöglicht es, Personen nach deren Nachnamen zu selektieren, wobei der Benutzer das jeweilige Selektionskriterium (z.B. B\*) eingibt:

```

void Oberflaeche::selektiereNachnamePersonen ()                               Selektieren von
{ PersonAllSet* allPersons = new PersonAllSet (pObjBase);                   Personen
  PersonCSet* result = new PersonCSet;    //Ergebnismenge
  PersonQuery query;                      //Selektion
  Person* p;

  char Suchbegriff [20];
  cout<< "Suchbegriff*: "; cin >> Suchbegriff;

  //Selektionsbedingung definieren
  query.SetNachname ((PtString) Suchbegriff, PtEQ);

  //Selektion durchführen
  allPersons->Query (&query, result);

  //Ausgabe der selektierten Ergebnisse
  int Anzahl = result->GetNum();
  cout <<"Anzahl der Objekte: " <<Anzahl <<endl;
  int nochDaten = 0;
  nochDaten = result->Seek (0, PtSTART);
  while (nochDaten == 0)
  {   result->Get(p);
      cout << (char*) p->getVorname() << " "
          << (char*) p->getNachname() <<endl;
      result->Unget(p);
      nochDaten = result->Seek (1, PtCURRENT);
  }
  delete allPersons;
  delete result;
}
    
```

## 2.5 Öffnen und Schließen der Datenbank

Die Datenbank wird vom Programm aus wie folgt geöffnet:

```

int tPOET(PtTransactionByBase, "ClientName");
int err = PtBase::POET()->GetBase("LOCAL", "base", pObjBase);
if (err < 0)
    
```

## Exkurs 2 2 Erstellen einer einfachen Klasse

```
{ PtBase: : POET() ->UngetBase(pObj Base);  
  exit (0);  
}  
// Die Datenbank ist nun geöffnet...
```

Bei Programmende muß die Datenbank geschlossen werden.

```
PtBase: : POET() ->UngetBase(pObj Base);  
Dei ni tPOET();
```

Um die Anwendung zu testen, erstellen wir eine einfache Operation `auswahl()`. Nun können Sie bereits Personen in Poet verwalten. Das Ergebnis dieses ersten Schritts finden Sie im Verzeichnis *Diploma1*.

Diploma1



```
void Oberflaeche: :auswahl ()  
{ // ... Öffnen der Datenbank ...  
  int Funktion;  
  do  
  { cout <<"1-erfassen 2-Li ste 3-sel ektieren 0-Ende: ";  
    cin >>Funktion;  
    swi tch (Funktion)  
    { case 1: erfassenPerson(); break;  
      case 2: Li stePersonen(); break;  
      case 3: sel ektiereNachnamePersonen(); break;  
      case 0: ;  
    }  
  } while (Funktion != 0);  
  // ... Schließen der Datenbank ...  
}
```

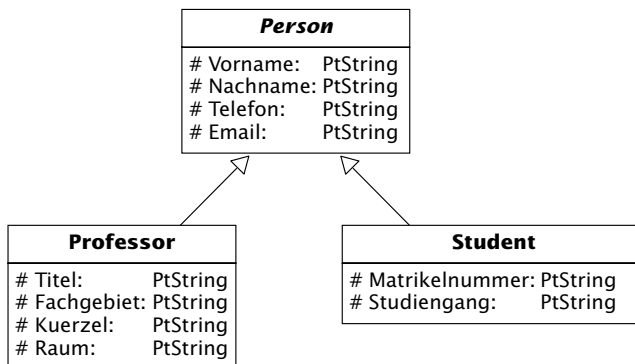
## 3 Realisieren der Vererbung

Bei unserem Fallbeispiel wollen wir Professoren und Studenten verwalten (Abb. 3-1), deren gemeinsame Eigenschaften in der abstrakten Klasse `Person` enthalten sind. Die Realisierung dieses Kapitels finden Sie im Verzeichnis *Diploma2*.

Diploma2



Abb. 3-1:  
Vererbungsstruktur



Für die Klassen der Abb. 3-1 erstellen wir folgende hcd-Datei, wobei wir der Einfachheit halber die **Schemadeklarationen** aller persistenten Klassen in eine einzige Datei schreiben. Bei dieser kleinen Fallstudie ist dies vertretbar. Normalerweise würden Sie für jede Klasse eine separate Datei erstellen.

```
//Di pl oma. hcd
#include <ptstring.hxx>
_persistent class Person //Klasse ohne extent, deswegen _persistent
{protected:
    PtString    Vorname,
                Nachname,
                Telefon,
                Email;

public:
    PtString getVorname () const;
    void setVorname (PtString vn);

    PtString getNachname () const;
    void setNachname (PtString nn);
    ...
};

persistent class Professor: public Person
{protected:
    PtString    Titel,
                Fachgebiet,
                Kuerzel,
                Raum;

public:
    PtString getKuerzel () const;
    void setKuerzel (PtString k);
    ...
};

persistent class Student: public Person
{protected:
    PtString    Matrikelnummer,
                Studiengang;

public:
    PtString getMatrikelnummer() const;
    void setMatrikelnummer (PtString mn);
    ...
};
typedef cset <Professor*> ProfessorCSet;
typedef cset <Student*> StudentCSet;
```

hcd-Datei  
für Vererbungs-  
struktur

Bei der Schemadeklaration von Person verwenden wir anstelle von persistent das Schlüsselwort `_persistent`. Dadurch wird dem Poet-Präprozessor mitgeteilt, daß für Person keine *AllSet*-Klasse zu erstellen ist. Diese Angabe kann nicht nur bei abstrakten, sondern auch bei konkreten Klassen erfolgen. Poet kann die Objekte einer solchen – konkreten – Klasse wesentlich schneller speichern. Durch die Verwendung von `persistent` bei der Klasse Person würde eine

## Exkurs 2 3 Realisieren der Vererbung

Klassenextension erzeugt, die alle Studenten und alle Professoren enthielte.

Erstellen Sie nun für die Vererbungsstruktur analog zu Kapitel 2 Operationen zum Erfassen und zur Listenausgabe von Studenten und Professoren. Das Ergebnis dieses Schritts finden Sie im Verzeichnis *Diploma2* auf der CD-Rom.



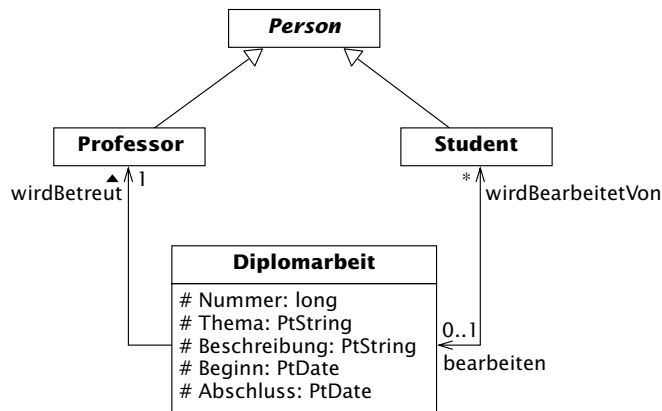
## 4 Realisieren von Assoziationen

Wir fügen nun die Klasse *Diplomarbeit* und die Assoziationen hinzu. Das Ergebnis ist die Realisierung des Klassendiagramms von *Diploma* (Abb. 4-1). Die Assoziation zwischen *Professor* und *Diplomarbeit* realisieren wir nur in einer Richtung, diejenige zwischen *Diplomarbeit* und *Student* in beiden Richtungen. Die Implementierung ist im Verzeichnis *Diploma3* enthalten.

Diploma3



Abb. 4-1:  
Klassendiagramm  
*Diploma*



Assoziationen können in Poet auf zwei Arten realisiert werden:

- als *pointer references* und
- als *ondemand references*.

Poet konvertiert die Zeiger von C++ automatisch in eine Form, die in der Datenbank gespeichert werden kann.

### 4.1 Realisierung mit *pointer*-Referenzen

Wir betrachten zuerst die Assoziation von *Diplomarbeit* zu *Professor*. Jede *Diplomarbeit* wird von genau einem *Professor* betreut. Wir nennen diese Assoziation *wirdBetreut* und realisieren sie durch einen Zeiger (*pointer*) auf *Professor*. Zur Manipulation dieser Assoziation benötigen wir Operationen zum Aufbauen (*link*) und Lesen (*getlink*) der Verbindung. Die Operation *unlink()* zum Entfernen einer existierenden Verbindung wird bei diesem Beispiel nicht benötigt, weil keine Löschoptionen implementiert werden.

```
//diploma.hcd
... Deklarationen von Person, Student und Professor ...
persistent class Diplomarbeit
{
protected:
    long Nummer;
    PtString Thema,
             Beschreibung;
    PtDate Beginn,
           Abschluss;

    //Assoziation
    Professor* wirdBetreut; //Diplomarbeit -> 1 Professor
public:
    ...
    ~Diplomarbeit();

    //Assoziation zu 1 Professor
    void link (Professor* pProf);
    void getLink (Professor* &pProf);
};
```

hcd-Datei mit  
*pointer reference*

Die Operationen `link()` und `getLink()` werden bei der *pointer reference* wie folgt implementiert:

```
void Diplomarbeit::link (Professor* pProf)
{ wirdBetreut = pProf;
}
void Diplomarbeit::getLink (Professor* &pProf)
{ pProf = wirdBetreut;
}
```

Wenn Sie in Poet mit Assoziationen arbeiten, kommen Sie nicht umhin, sich mit dem **Referenzzähler** (*link count*) zu beschäftigen. Poet lädt ein Objekt aus der Datenbank in den Speicher, wenn es angesprochen – d.h. referenziert – wird. Diese Referenzierung kann vom Programmierer über eine Variable oder – mittels der Assoziationen – über ein anderes Objekt erfolgen. Wenn dasselbe Objekt mehrmals referenziert wird, so wird es *nicht* mehrmals in den Speicher geladen, was zu Inkonsistenzen führen könnte. Stattdessen verwendet Poet den Referenzzähler (*link count*). Das ist ein Zähler, der beim Erzeugen eines neuen Objekts auf 1 gesetzt und bei jeder neuen Referenzierung dieses Objekts inkrementiert wird. Er kann jederzeit mit der Operation `int PtObject::GetLinkCount()` abgefragt werden.

Referenzzähler

Der Referenzzähler wird benötigt, um festzustellen, ob ein Objekt sicher, d. h. ohne *dangling references*, gelöscht werden kann. In C++ existiert die Löschoperation `delete`, die ein Objekt aus dem Speicher löscht, unabhängig davon, ob es noch von anderen Objekten referenziert wird. Dadurch entsteht das Problem der *dangling references*. Poet bietet die sichere Löschoption `Forget()` an. Diese Operation dekrementiert den Referenzzähler. Falls er dann Null ist

## Exkurs 2 4 Realisieren von Assoziationen

– d.h. es gibt keine weiteren Referenzen auf dieses Objekt – wird das Objekt gelöscht.

Da es in diesem Exkurs darum geht, die Realisierung objekt-orientierter Konzepte anhand von Poet zu demonstrieren, verzichte ich der Einfachheit halber auf eine vollständige Speicherbereinigung (*cleanup*), die bei einem echten Projekt jedoch unerlässlich ist.

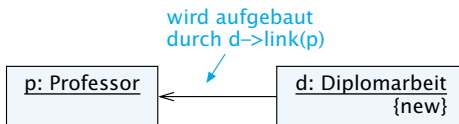
Löschen  
assoziierter  
Objekte

Wenn eine Diplomarbeit – durch eine entsprechende Operation – aus dem Speicher gelöscht wird, dann sorgt der folgende **Destruktor** dafür, daß auch der zugehörige Professor aus dem Speicher entfernt wird, sofern dieses Objekt nicht noch anderweitig referenziert wird.

```
Di pl oma rbei t : : ~Di pl oma rbei t ()  
{ if (wi rdBet reut != NULL)  
    wi rdBet reut->For get (); //jede DA wird von 1 Prof bet reut  
}
```

Jede neue **Diplomarbeit** wird bei der **Erfassung** einem Professor zugeordnet. Dieser Professor wird vom Benutzer mit der Operation `Professor* auswaehl enProfessor()` aus einer Liste ausgewählt. Anschließend wird mit `link()` die Objektverbindung von der Diplomarbeit zum Professor aufgebaut (Abb. 4-2), bevor die neue Diplomarbeit mit `Store()` in der Datenbank gespeichert wird.

Abb. 4-2: Objekte  
Diplomarbeit und  
Objektverbindung  
zu Professor



Damit ergibt sich für die Erfassung von Diplomarbeiten folgende Operation:

```
Erfassen einer  
Diplomarbeit void d Oberfl aeche : : erfassenDi pl oma rbei t ()  
{ Di pl oma rbei t* d = new Di pl oma rbei t ;  
  char name [30];  
  cout <<"Nummer : "; ci n >> name; d->setNum mer ((PtString) name);  
  cout <<"Thema : "; ci n >> name; d->setThema ((PtString) name);  
  
  Professor* p = auswaehl enProfessor ();  
  d->Assi gn (pObj Base); //OID für Poet zuweisen  
  d->li nk(p); //Verbi ndung zu Prof aufbauen  
  d->Store (); //Objekt mit Verbi ndung zu Prof in Datenbank spei chern  
  delete d; //Destruktor wirkt auch auf den referenzi erten Prof  
}  
  
Professor* Oberfl aeche : : auswaehl enProfessor ()  
{ //Li ste der Professoren anzei gen  
  ProfessorAl l Set* al l Professors = new ProfessorAl l Set  
    (pObj Base);
```

```

Professor* p;
for (int i = 0; allProfessors->Get (p, i, PtSTART) == 0; i++)
{   cout << i << ": " << (char*) p->getNachname() << endl;
    allProfessors->Unget(p);
};

//Professor auswählen
int Nummer;
cout << " Welchen Professor? (Lfd. Nummer eingeben)" << endl;
cin >> Nummer;
allProfessors->Get (p, Nummer, PtSTART);

delete allProfessors;
return p;
}

```

Bei der Ausgabe der **Liste aller Diplomarbeiten** wird außer dem Thema der jeweilige Betreuer ausgegeben. Da es sich hier um eine *pointer*-Referenz handelt, lädt die Operation `Get()` nicht nur die Diplomarbeit, sondern auch den zugehörigen Professor in den Speicher. Die Objektverbindung wird mittels `getLink()` gelesen und dann der Nachname des Professors ausgegeben. Am Ende der *for*-Schleife wird `Unget()` für »Aufräumarbeiten« aufgerufen.

Es ergibt sich folgende Operation:

```

void Oberflaeche : ListeDiplomarbeiten()
{   cout << "Liste der Diplomarbeiten" << endl;
    Diplomarbeit* d;
    Diplomarbeit* d;
    Professor* p;
    for (int i = 0; allIDA->Get (d, i, PtSTART) == 0; i++)
    {   d->getLink(p); //zugehörigen Prof (Betreuer) ermitteln
        cout << (char*) d->getThema() << " -> Betreuer: "
            << (char*) p->getNachname() << " -> Diplomanden: ";
        cout << endl;

        allIDA->Unget(d);
    };
    delete allIDA;
}

```

Liste der Diplomarbeiten und deren Betreuer

## 4.2 Realisierung mit *ondemand*-Referenzen

Wenn Sie mit Zeigern arbeiten, dann werden beim Laden eines Objekts automatisch alle referenzierten Objekte geladen. Das kann sehr leicht zu *Performance*-Problemen führen. Poet löst dieses Problem mit den *ondemand references*, d.h. jede Objektverbindung (Referenz) wird als Objekt der Klasse *ondemand* gespeichert. Um beide Techniken miteinander zu vergleichen, realisieren wir die Assoziation zwischen Student und Diplomarbeit mittels *ondemand*.

Jeder Student *bearbeitet* genau eine Diplomarbeit. Es ergibt sich folgende Schemadeklaration:



## Exkurs 2 4 Realisieren von Assoziationen

```
hcd-Datei //Di pl oma. hcd
für ondemand ... Deklarationen von Person, Professor und Di pl oma rbei t...
reference persistent class Student: public Person
{protected:
    ...
    ondemand <Di pl oma rbei t> bearbei tet; //Student -> 1 Di pl oma rbei t
public:
    ...
    //Assoziati on zu Di pl oma rbei t;
    voi d link (Di pl oma rbei t* pArbei t);
    voi d getlink (Di pl oma rbei t* &pArbei t);
};
```

Die Operationen zum Aufbauen und Lesen der *ondemand*-Referenzen werden wie folgt realisiert:

```
voi d Student::link (Di pl oma rbei t* pArbei t)
{ bearbei tet. SetReference (pArbei t);
}
voi d Student::getlink (Di pl oma rbei t* &pArbei t)
{ bearbei tet. Get (pArbei t);
}
```

Auch jedes neue Objekt der Klasse **Student** wird gleich bei der **Erfassung** einer Diplomarbeit zugeordnet. Diese Diplomarbeit wird analog zu oben vom Benutzer mit der Operation `Di pl oma rbei t* auswaehl enDi pl oma rbei t()` aus einer Liste ausgewählt.

Achten Sie beim Arbeiten mit *ondemand*-Referenzen darauf, einem Objekt *zuerst* mit `Assign()` eine OID zuzuweisen, bevor es mittels `link()` mit einem anderen Objekt verbunden wird.

Es ergibt sich folgende Operation:

```
Erfassen eines voi d Oberflaeche::erfassenStudent ()
Studenten { Student* s = new Student;
            char name [30];
            cout <<"Nachname: "; cin >> name;
            s->setNachname ((PtString) name);
            cout <<"Matrikelnummer: "; cin >> name;
            s->setMatrikelnummer ((PtString) name);

            Di pl oma rbei t* d = auswaehl enDi pl oma rbei t();
            s->Assign (pObjBase); //OID für Poet zuweisen
            s->link(d); //Verbindung von Student zu DA aufbauen
            s->Store(); //Student speichern
        }
```

Bei der **Listenausgabe** aller **Studenten** soll außer dem Namen des Studenten das Thema der betreffenden Diplomarbeit ausgegeben werden. Daher wird für jedes Objekt `s` der Klasse `Student`, das aus der Klassenextension gelesen wurde, mit `s->getlink(d)` die zugehörige Diplomarbeit ermittelt, und dann die gewünschten Daten angezeigt.

### 4.3 Realisierung von *many*-Assoziationen Exkurs 2

```
void Oberflaeche::ListeStudenten()
{ cout <<endl << "Studenten-Li ste" << endl;
  StudentAl lSet* al lStuds = new StudentAl lSet (pObj Base);
  Student* s;
  Diplomarbei t* d;
  for (int i= 0; al lStuds->Get (s, i, PtSTART) == 0; i++)
  { s->getl ink(d); //zugehöri ge DA ermi ttel n
    cout << (char*) s->getNachname() << " -> bearbei tet DA: "
          << (char*) d->getThema() <<endl;
    al lStuds->Unget(s);
  };
  delete al lStuds;
}
```

Liste der  
Studenten

### 4.3 Realisierung von *many*-Assoziationen

Wir betrachten nun die *many*-Richtung der Assoziation von Diplomarbeit zu Student, wobei zwei Probleme gleichzeitig zu behandeln sind. Zum einen sind Mengen von Referenzen zu verwalten und zum anderen wird eine bidirektionale Assoziation realisiert: Ein Student bearbeitet eine Diplomarbeit und umgekehrt wird eine Diplomarbeit – oder genauer gesagt ein Diplomarbeitsthema – von mehreren Studenten bearbeitet. Hier ist es empfehlenswert, mit *ondemand*-Referenzen zu arbeiten.

Die Schemadeklaration der Klasse `Diplomarbeit` wird wie folgt erweitert:

```
//Di pl oma. hcd
... Deklarationen von Person, Student und Professor ...
persi stent class Di pl oma rbei t
{protected:
    ...
    //Assozi ati on
    cset <ondemand <Student>> wi rdBearbei tetVon; //Di pl oma rbei t
                                                //-> * Student

public:
    ...
    //Assozi ati on zu many Studenten
    void l ink (Student* pStud);
    int getl ink (Student* &pStud, int posi ti on);
};
```

hcd-Datei mit  
*ondemand*  
references

Die *link*- und *getlink*-Operationen für die *many*-Richtung der Assoziation zur Klasse `Student` lauten wie folgt:

```
void Di pl oma rbei t::l ink (Student* pStud)
{ wi rdBearbei tetVon. Append (pStud);
}
int Di pl oma rbei t::getl ink (Student* &pStud, int posi ti on)
{ int err = wi rdBearbei tetVon. Get (pStud, posi ti on, PtSTART);
  return err; //0, wenn alles ok
}
```

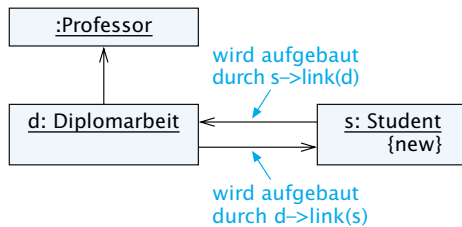
## Exkurs 2 4 Realisieren von Assoziationen

Speichern von  
Objekten

Die Operation **Store()** arbeitet mit verschiedenen Modi, die über einen Parameter eingestellt werden. Die Wahl des richtigen Modus stellt eine wichtige Entwurfsentscheidung dar. Die hier verwendete Standardform speichert das jeweilige Objekt und alle referenzierten Objekte (auch *ondemand*). Dieser Modus stellt die Konsistenz der Daten sicher und sorgt dafür, daß ein großes Netz von referenzierten Objekten schnell gespeichert wird. Nachteilig ist, daß alle referenzierten Objekte, die sich im Speicher befinden, gespeichert werden, auch wenn sie nicht verändert wurden.

Für die Operation zum **Erfassen** von **Studenten** sind geringfügige Erweiterungen notwendig, die der Übersichtlichkeit halber fett gedruckt sind. Da wir hier eine bidirektionale Assoziation realisieren, werden zwei *link*-Operationen benötigt (Abb. 4-3). Wenn anschließend die *Store*-Operation angewendet wird, dann wird nicht nur das neue Objekt der Klasse Student, sondern auch das aktualisierte Objekt von Diplomarbeit gespeichert.

Abb. 4-3: Aufbauen  
bidirektionaler  
Objekt-  
verbindungen



bidirektionale  
Objektverbindung  
erstellen

```

void Oberflaeche::erfassenStudent ()
{
    Student* s = new Student;
    char name [30];
    cout <<"Nachname: "; cin >> name;
    s->setNachname ((PtString) name);
    cout <<"Matrikelnummer: "; cin >> name;
    s->setMatrikelnummer ((PtString) name);

    Diplomarbei t* d = auswaehl enDi pl oma rbei t ();
    s->Assi gn (pObj Base); //OID für Poet zuweisen
    s->link(d); //Verbindung von Student zu DA aufbauen
    d->link(s); //Verbindung von DA zu Student aufbauen
    s->Store(); //Student speichern und DA aktualisieren
}
    
```

Die Operation für die Listenausgabe von Diplomarbeiten wird so erweitert, daß außer der Diplomarbeit der jeweilige Betreuer und die Namen vorhandener Diplomanden ausgegeben werden. Auch diese Erweiterungen sind fett gedruckt.

Durch die *pointer*-Referenz zu Professor wird beim Laden der Diplomarbeit mit `Get()` das zugehörige Professor-Objekt automatisch geladen, beim Löschen der Diplomarbeit mit `Unget()` wird über den Destruktor der Professor ebenfalls aus dem Speicher entfernt, sofern keine anderen Referenzen darauf verweisen. Die *ondemand*-Re-

### 4.3 Realisierung von *many*-Assoziationen Exkurs 2

ferenz zu den Studenten wird mittels `getlink()` gelesen und damit die assoziierten Objekte geladen.

```
void Oberflaeche::ListeDiplomarbeiten()
{ cout << "Liste der Diplomarbeiten" << endl;
  DiplomarbeitAllSet* allDA = new DiplomarbeitAllSet (pObjBase);
  Diplomarbeit* d;
  Professor* p;
  Student* s;
  for (int i = 0; allDA->Get (d, i, PtSTART) == 0; i++)
  { d->getlink(p); //zugehörigen Prof (Betreuer) ermitteln
    cout << (char*) d->getThema() <<" -> Betreuer: "
          << (char*) p->getNachname() <<" -> Diplomanden: ";

    //alle zugehörigen Studenten (Diplomanden) ermitteln
    for (int j = 0; d->getlink(s, j) == 0; j++)
      cout <<(char*) s->getNachname() <<" , ";
    cout << endl;

    allDA->Unget(d);
  };
  delete allDA;
}
```

Liste der Diplomarbeiten  
mit Betreuer  
und Diplomanden



Das Ergebnis dieses letzten Schritts finden Sie im Verzeichnis `Diploma3` *Diploma3* auf der CD-ROM.

# Anhang 1: Durchgängiges Fallbeispiel



- Ein Pflichtenheft erstellen können.
- Ein OOA-Modell erstellen können.
- Die Benutzungsoberfläche aus dem OOA-Modell ableiten können.
- Objekt-relationale Abbildung durchführen können.
- Tabellen und *queries* in SQL erstellen können.
- Datenhaltung gemäß ODMG-Standard in ODL spezifizieren können.
- Datenhaltung mit der objektorientierten Datenbank Poet spezifizieren können.

anwenden

- i 1 Pflichtenheft 464
- 2 OOA-Modell 468
- 3 Prototyp der Benutzungsoberfläche 471
- 4 Datenhaltung mittels einer relationalen Datenbank 475
- 5 Datenhaltung mittels einer objektorientierten Datenbank 477
- 6 OOD-Modell und Implementierung 481

# 1 Pflichtenheft

**Gliederungs-  
schema  
Pflichtenheft**

## 1 Zielbestimmung

Formulieren Sie Ziele (z.B. Mindestbestand von Artikeln automatisch sicherstellen) und nicht die für deren Erreichung notwendigen Funktionen (z.B. Erstellung von Bestellvorschlagslisten für Artikel, deren Mindestbestand unterschritten ist). Oft wird ein Ziel durch eine Funktion realisiert. Dann ist die Abgrenzung unter Umständen schwierig.

### 1.1 Muß-Kriterien

Nennen Sie alle Ziele, die das Softwaresystem unbedingt erfüllen muß. Kann eines der Muß-Kriterien nicht realisiert werden, dann ist das ganze System für den vorgesehenen Zweck nicht einsetzbar.

Beispiel

Bei einem Werkzeug zur Erstellung von OO-Modellen sind folgende Muß-Kriterien sinnvoll:

- Unterstützung der UML-Notation,
- Mehrbenutzerfähigkeit,
- Automatische Erstellung der Dokumentation.

### 1.2 Kann-Kriterien

Nennen Sie hier diejenigen Ziele, die das Produkt zwar erfüllen sollte, auf die aber zunächst verzichtet werden kann. Diese Abgrenzung ist ein wichtiges Instrument der Projektplanung. Bei Terminproblemen ist somit eine Konzentration auf die Muß-Kriterien möglich.

Beispiel

Bei einem Buchhaltungsprogramm ist das automatische Erstellen einer Umsatzsteuer-Voranmeldung ein Muß-Kriterium. Das Ausdrucken dieser Voranmeldung auf einem von den Finanzämtern genehmigtem Formular stellt ein Kann-Kriterium dar, weil der Benutzer das Programm auch ohne diese Funktionalität benutzen kann und nur die Programmdateien handschriftlich auf ein Formular übertragen muß.

### 1.2 Abgrenzungskriterien

Machen Sie deutlich, welche Ziele mit dem Produkt bewußt *nicht* erreicht werden sollen, die aber in vergleichbaren Anwendungen durchaus vorkommen.

Beispiel

Bei einem Werkzeug zur Erstellung von OO-Modellen erfolgt keine automatische Optimierung bei der Darstellung von Diagrammen.

## 2 Einsatz

Die Analyse des Einsatzes liefert wichtige Informationen für die Benutzungsoberfläche und die Qualitätsanforderungen des zukünftigen Systems.

### 2.1 Anwendungsbereiche

z.B. [Buchhaltung von Unternehmen](#).

### 2.2 Zielgruppen

z.B. [Buchhalter](#).

### 2.3 Betriebsbedingungen

Dazu gehören Angaben über

- die physikalische Umgebung des Softwaresystems (z.B. Büroumgebung),
- die tägliche Betriebszeit (z.B. 8 Stunden) und
- ob eine ständige Beobachtung des Softwaresystems durch den Bediener oder ein unbeaufsichtigter Betrieb vorliegt.

## 3 Umgebung

### 3.1 Software

Welche Softwaresysteme (einschließlich Versionsnummern) müssen für den Betrieb zur Verfügung stehen? Wenn das Produkt nicht als *stand-alone*-Produkt geplant ist, so sind die geplanten Schnittstellen zu anderen Softwareprodukten aufzuführen.

### 3.2 Hardware

Welche Hardware-Voraussetzungen müssen für den Betrieb erfüllt sein?

### 3.3 Orgware

Welche organisatorischen Voraussetzungen müssen für den Betrieb gegeben sein? Welche organisatorischen Schritte müssen durchgeführt werden, damit das Softwaresystem eingesetzt werden kann?

[Vor dem Einsatz eines Buchhaltungsprogramms muß zunächst von einem Buchhalter ein Kontenplan für das Unternehmens erstellt werden.](#) Beispiel

## 4 Funktionalität

Die Funktionalität des Systems ist auf oberster Abstraktionsebene zu beschreiben. Das bedeutet, daß die typischen Arbeitsabläufe, die mit dem zu erstellenden System durchgeführt werden sollen, zu nennen sind. Zu diesem frühen Zeitpunkt ist noch nicht abzu-

sehen, ob diese Arbeitsabläufe vollständig durch Software realisiert werden oder auch organisatorische Schritte beinhalten. Ein Arbeitsablauf soll immer zu einem Ergebnis für den Bediener führen. Das bedeutet, daß nicht mehrere Arbeitsabläufe kombiniert werden müssen, um ein Ergebnis zu halten. Die hier beschriebenen Arbeitsabläufe bilden die Grundlage für die im Rahmen der objektorientierten Analyse erstellten Geschäftsprozesse.

Informationssysteme enthalten im allgemeinen eine Reihe von Verwaltungsfunktionen, z.B. Erfassen eines neuen Artikels, Aktualisieren der Artikeldaten, Löschen alter Artikel aus dem System. Diese Funktionalität ist hier *nicht* aufzuführen.

Außerdem erstellen viele Informationssysteme eine Reihe von Reports, Berichten etc., von denen die wichtigsten hier aufzuführen sind. Auf Funktionen, die nur elementare Listen (z.B. Liste aller Artikel) erstellen, ist jedoch zu verzichten.

Bei der Formulierung dieses Kapitels ist zu berücksichtigen, daß hier die Basis für das spätere OOA-Modell gelegt werden soll, und daß keine vollständige textuelle Beschreibung der funktionalen Anforderungen verlangt wird.

### 5 Daten

Die langfristig zu speichernden Daten und deren voraussichtlicher Umfang sind aus Benutzersicht aufzuführen.

Beispiel

- 50.000 bis 200.000 Artikel,
- 3000 Kunden.

### 6 Leistungen

Führen Sie alle zeitlichen Anforderungen auf. Quantifizieren Sie alle Angaben, z.B. max. 2 Sekunden für das Auffinden eines Artikels. Überlegen Sie, ob die gewünschten Leistungen mit den in Kapitel 5 genannten Datenmengen und der in Kapitel 3.2 aufgeführten Hardware erreicht werden können.

### 7 Benutzungsoberfläche

Formulieren Sie grundlegende Anforderungen an die Benutzungsoberfläche. Berücksichtigen Sie die jeweiligen Eigenschaften der zukünftigen Benutzer und die voraussichtliche Art der Benutzung. Die genaue »Spezifikation« der Benutzungsoberfläche erfolgt durch den Prototyp.

Beispiel

Ein Buchhalter, der täglich mehrere hundert Buchungssätze eingibt, benötigt eine andere Benutzungsoberfläche als ein Freiberufler, der monatlich seine Umsatzsteuer-Voranmeldung mit einem Buchhaltungsprogramm erstellt.



## 8 Qualitätsziele

Welche Qualität soll das neue Softwaresystem besitzen? Beispielsweise wird eine hohe Portabilität benötigt, wenn das Softwaresystem auf verschiedenen Plattformen laufen soll. Bei den meisten Softwareprodukten sind Änderbarkeit und Wartbarkeit wichtige Qualitätsziele.

## 9 Ergänzungen

Wenn die bisherigen Kapitel nicht ausreichen, um die Anforderungen an ein Softwaresystem zu beschreiben, dann kann das Pflichtenheft individuell erweitert werden.

# Pflichtenheft Diploma

## 1 Zielbestimmung

### 1.1 Muß-Kriterien

- zentrale Verwaltung von Diplomarbeitsthemen und Diplomanden (Studenten).
- einfacher Überblick über Diplomarbeitsthemen.

### 1.2 Kann-Kriterien

keine.

### 1.3 Abgrenzungskriterien

- kein Zugriffsschutz (bei einer echten Anwendung wäre dies ein Muß-Kriterium, da jeder Professor nur für die eigenen Diplomarbeitsthemen Zugriffe durchführen darf).

## 2 Einsatz

### 2.1 Anwendungsbereiche

Hochschulen.

### 2.2 Zielgruppen

Studenten, Professoren.

## 3 Umgebung

### 3.1 Software

Windows NT 4.0.

### 3.2 Hardware

beliebiger NT 4.0-Rechner.

### **3.3 Orgware**

keine

### **4 Funktionalität**

Es läßt sich nur ein Arbeitsablauf identifizieren: Das Bearbeiten eines Diplomarbeitsthemas durch einen oder mehrere Studenten wird ins System eingetragen.

Das Erfassen, Ändern und Löschen von Professoren und Diplomarbeiten erfolgt mit Hilfe der Verwaltungsfunktionen.

Desweiteren sollen sich Studenten mittels folgender Funktionen informieren können:

- Liste der verfügbaren Diplomarbeitsthemen erstellen,
- Liste der abgeschlossenen Diplomarbeitsthemen erstellen,
- Liste der aktuell bearbeiteten Diplomarbeitsthemen erstellen.

### **5 Daten**

Es sollen die Diplomarbeitsthemen eines Fachbereichs verwaltet werden, d.h.

ca. 30 Professoren und pro Professor und Jahr ca. 10 Diplomanden, d.h.  $30 * 10 * 5 = 1500$  Studenten und 1500 Diplomarbeiten bei einer fünfjährigen Nutzungszeit der Software.

### **6 Leistungen**

Leistungen sind hier nicht relevant.

### **7 Benutzungsoberfläche**

Es wird eine objektorientierte Oberfläche mit Menüs entsprechend der Gestaltungsvorschriften in den Lehreinheiten 9 und 10 erstellt.

### **8 Qualitätsziele**

- hohe Benutzungsfreundlichkeit.
- hohe Änderbarkeit.

## **2 OOA-Modell**

Bei einfachen Fallstudien wie dieser ist es oft schwierig, sinnvolle Geschäftsprozesse zu bilden, da ein großer Teil der Funktionalität durch einfache Verwaltungsfunktionen abgedeckt wird. Es ergibt sich nur ein einziger Geschäftsprozeß (Abb. 2-1). Die im Rahmen dieses Geschäftsprozesses anstehenden Aufgaben werden vom jeweiligen Professor durchgeführt; er ist daher der alleinige Akteur des Systems.

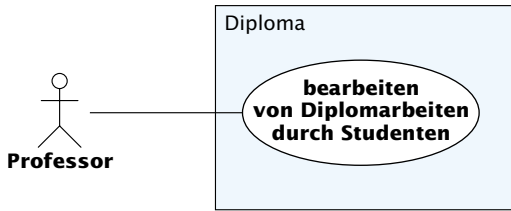


Abb. 2-1:  
Geschäftsprozess-  
diagramm für  
Diploma

Geschäftsprozeß: bearbeiten von Diplomarbeiten durch Studenten

Ziel: Student schließt Diplomarbeit erfolgreich ab

Kategorie:

Vorbedingung: Diplomarbeitsthema verfügbar

Nachbedingung Erfolg: Diplomarbeit abgeschlossen

Nachbedingung Fehlschlag: Student bricht die Arbeit ab

Akteure: Professor

Auslösendes Ereignis: Student will Diplomarbeitsthema bearbeiten

Beschreibung:

- 1 Prüfen, ob Diplomarbeitsthema noch verfügbar
- 2 Diplomarbeitsthema als vergeben kennzeichnen
- 3 Studenten erfassen und dem Diplomarbeitsthema zuweisen
- 4 Abschlußdatum eintragen

Erweiterungen:

Alternativen:

3a wenn Studenten bereits im Softwaresystem erfaßt sind, dann nur zuweisen

4a bei Abbruch der Arbeit ist die Zuordnung des Studenten zur Diplomarbeit zu lösen

Spezifikations-  
schablone

Abb. 2-2 zeigt das Klassendiagramm, dessen Attribute unten spezifiziert werden. Die Zuordnung der Operationen erfolgt nach folgendem Schema. Für *ein* Diplomarbeitsthema – der Einfachheit halber als Diplomarbeit bezeichnet – melden sich ein oder mehrere Studenten an und *eine* Diplomarbeit wird – von den zugehörigen Diplomanden – abgeschlossen. Daher werden die entsprechenden Operationen bei der Klasse Diplomarbeit eingetragen. Wenn *ein* Student – evtl. aus einer Gruppe – die Diplomarbeit abbricht, dann muß für diesen Studenten die Verbindung zur Diplomarbeit gelöst werden. Die Klassenoperationen beziehen sich jeweils auf die Menge aller Diplomarbeiten.

Operationen  
zuordnen

#### Person

Vorname: String

Nachname: String {mandatory}

Telefon: String

Email: String

#### Professor

Titel: String

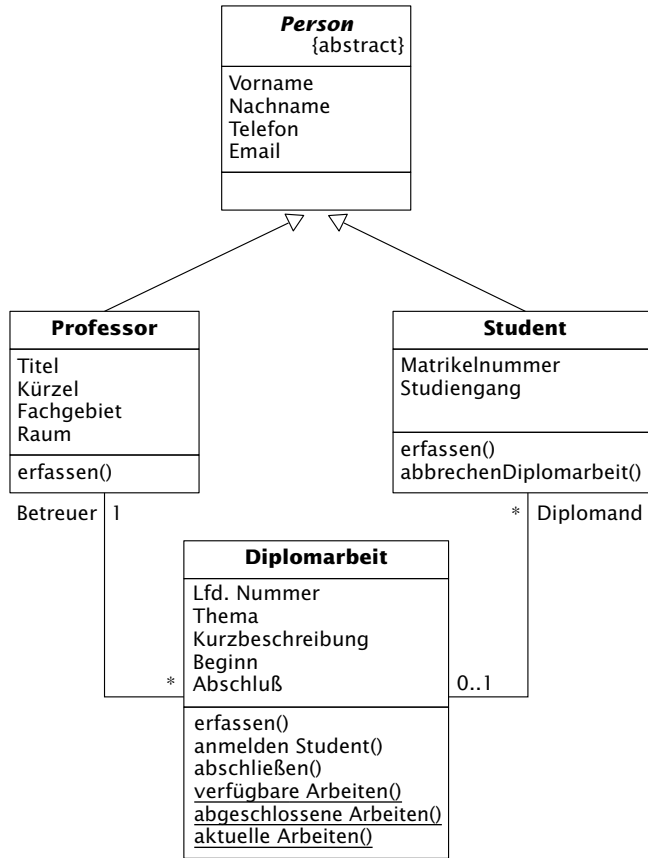
Kürzel: String {mandatory, key}

Fachgebiet: String

Raum: String

## Anhang 1 2 OOA-Modell

Abb. 2-2 Klassen-  
diagramm für  
Diploma



### Student

Matrikelnummer: String {mandatory, key}

Studiengang: String

### Diplomarbeit

Lfd. Nummer: UInt {mandatory, key}

Thema: String {mandatory}

Kurzbeschreibung: String

Beginn: Date

Abschluß: Date

{Beginn < Abschluß}

Der Übung halber erstellen wir ein Sequenzdiagramm, das bei diesem Beispiel sehr einfach ist. Abb. 2-3 modelliert den Standardfall für den oben spezifizierten Geschäftsprozeß.

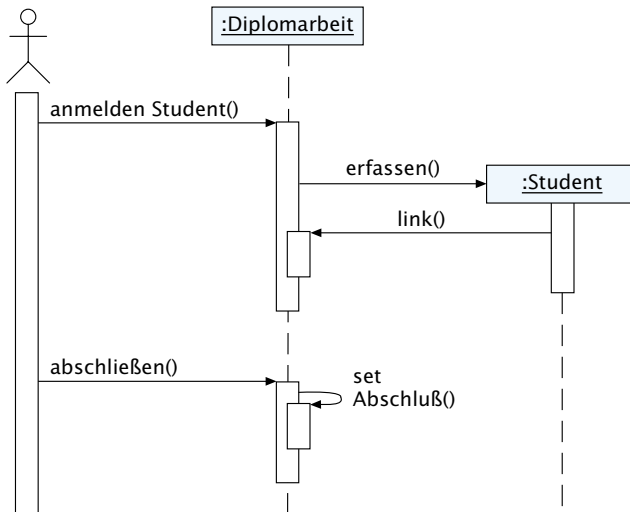


Abb. 2-3: Sequenzdiagramm für das erfolgreiche Bearbeiten von Diplomarbeiten durch Studenten

### 3 Prototyp der Benutzungsoberfläche

In diesem Kapitel erstellen wir einen Prototyp der Benutzungsoberfläche. Die Ausgangsbasis bilden das Klassendiagramm und die Spezifikation der Attribute.

Wir entwerfen den Prototyp in zwei Schritten:

- 1 Dialoggestaltung,
- 2 Gestaltung der Fenster mittels Interaktionselementen.

#### Dialoggestaltung

Wir verwenden die Standardabbildung der Analyseklassen auf den Menübalken, wie sie in Kapitel 5.5 (Lehreinheit 10) beschrieben ist. Die *drop-down*-Menüs *Listen* und *Erfassung* enthalten jeweils die Klassen Professor, Diplomarbeit und Student.

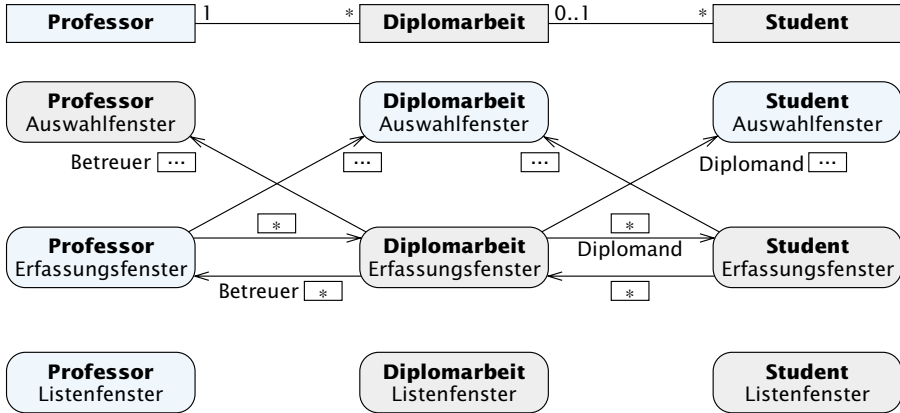
Nach den beschriebenen Transformationsregeln wird jede dieser Klassen auf ein Erfassungs- und ein Listenfenster abgebildet. Die bidirektionalen Assoziationen werden in diesem Prototyp bidirektional realisiert. Bei der späteren Anwendung, für die dieser Prototyp erstellt wird, werden viele Objektverbindungen durch entsprechende Operationen erstellt. Beispielsweise baut die Operation `anmeldenStudent()` eine Verbindung von der Diplomarbeit zum entsprechenden Studenten auf, die Operation `abbrechenDiplomarbeit()` löst diese Verbindung. Möglicherweise müssen einige Assoziationen auch nur in einer Richtung realisiert werden.

Transformations-  
schema

## Anhang 1 3 Prototyp der Benutzungsoberfläche

Für die Realisierung der drei konkreten OOA-Klassen werden neun Fenster benötigt. Abb. 3-1 zeigt in Form eines Zustandsdiagramms, wie der Dialogfluß mittels der Schaltflächen zwischen diesen Fenstern verzweigt. Aus Gründen der Übersichtlichkeit wurden hier nur die wichtigsten Transitionen eingetragen. Für alle anderen Zustandsübergänge gelten die in der Lehreinheit 10 angegebenen Konventionen.

Abb. 3-1: Zustandsdiagramm zur Modellierung der Dialogstruktur



### Gestaltung der Fenster mittels Interaktionselementen

**Erfassungsfenster** Ein Professor wird wie in Abb. 3-2 erfasst und geändert. Für die Eingabe des Fachgebiets wird ein erweiterbares *Dropdown*-Kombinationsfeld verwendet. Dadurch soll gewährleistet werden, daß das gleiche Fachgebiet nicht in verschiedenen Schreibweisen eingegeben wird. Analog wird beim Titel verfahren.

Von Professor besteht eine *many*-Assoziation zu Diplomarbeit. Sie wird im Erfassungsfenster von *Professor* durch eine Liste aller Diplomarbeiten dieses Professors realisiert. Wir verwenden hier das Interaktionselement Listenelement (*list view control*) mit Spaltenüberschriften.

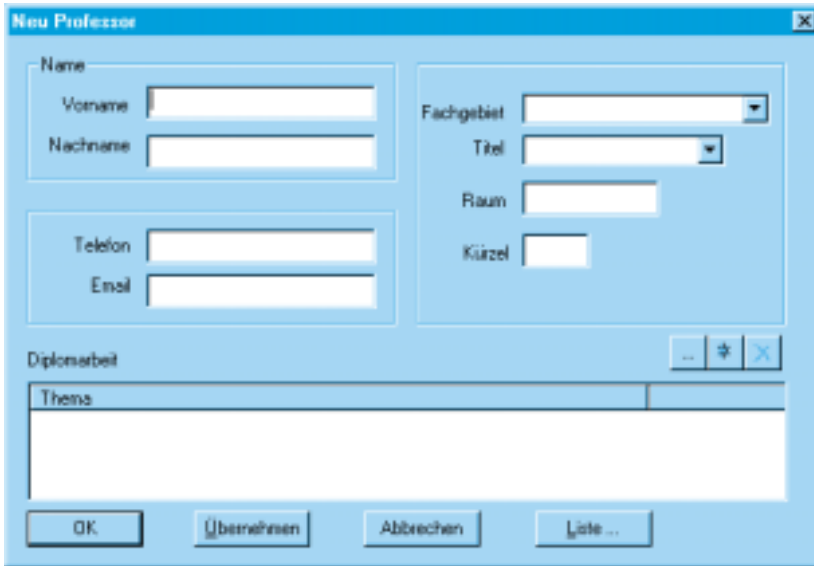


Abb. 3-2:  
Erfassungsfenster  
für Professor

In Abb. 3-3 ist das Erfassungsfenster für Diplomarbeiten dargestellt. Von hier kann zu dem betreuenden Professor und zu den Diplomanden (Studenten) verzweigt werden.

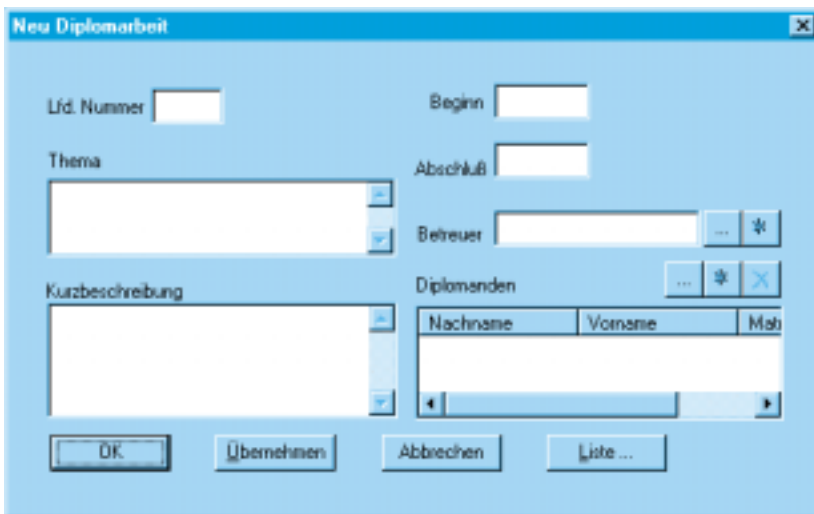


Abb. 3-3:  
Erfassungsfenster  
für Diplomarbeit

Studenten werden mit dem in Abb. 3-4 dargestellten Fenster erfaßt und geändert. Analog zu oben wurde für den Studiengang ein erweiterbares *Dropdown*-Kombinationsfeld gewählt. Beachten Sie, daß das Interaktionselement unter dem Text »Diplomarbeit« keine Auswahlliste, sondern ein mehrzeiliges Eingabefeld ist, das für lange Diplomarbeitsthemen benötigt wird.

## Anhang 1 3 Prototyp der Benutzungsoberfläche

Abb. 3-4:  
Erfassungsfenster  
für Student

The screenshot shows a window titled 'Neu Student' with a light blue background. It contains several input fields: 'Name' (subdivided into 'Vorname' and 'Nachname'), 'Matrikelnr.', 'Studiengang' (a dropdown menu), 'Telefon', 'Email', and 'Diplomarbeit' (a large text area with scrollbars). At the bottom, there are four buttons: 'OK', 'Übernehmen', 'Abbrechen', and 'Liste ...'.

Listenfenster Die Listenfenster sind alle nach dem gleichen Schema aufgebaut. Daher ist hier nur das Listenfenster für Professoren exemplarisch angegeben (Abb. 3-5). Bei der Liste der Diplomarbeiten wählen wir die Attribute Lfd. Nummer und Thema und bei der Liste der Studenten die Attribute Vorname, Nachname und Matrikelnummer.

Abb. 3-5:  
Listenfenster für  
Professoren

The screenshot shows a window titled 'Liste Professoren' with a light blue background. It features a table with the following headers: 'Nachname', 'Vorname', 'Kürzel', 'Raum', and an empty column. The table body is currently empty. At the bottom, there are four buttons: 'Neu ...', 'Ändern ...', 'Löschen', and 'Schließen'.

Auswahlfenster Auch die Auswahlfenster sind einheitlich konzipiert. Abb. 3-6 zeigt exemplarisch das Auswahlfenster für Diplomarbeiten. Hier wurden Lfd. Nummer und Thema als diejenigen Attribute ausgewählt, die der Benutzer zur Identifikation der Objekte benötigt. Bei Professoren und Studenten wählen wir in den entsprechenden Fenstern die Attribute Vorname und Nachname aus.



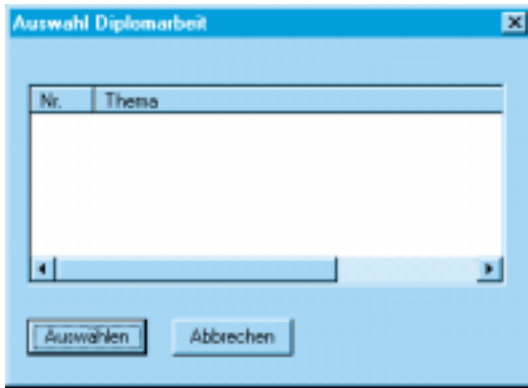


Abb. 3-6:  
Auswahlfenster für  
Diplomarbeiten

## 4 Datenhaltung mittels einer relationalen Datenbank

Die nächste Aufgabe besteht darin, die Fallstudie mittels einer relationalen Datenbank zu realisieren. Dazu gehören folgende Teilaufgaben:

- a** Abbildung des Klassendiagramms auf Tabellen.
- b** Formulieren des logischen Schemas in SQL.
- c** Realisierung eines externen Schemas.
- d** Realisierung der *Query*-Operationen in SQL.

### a Abbildung des Klassendiagramms auf Tabellen

Die Abbildung des Klassendiagramms auf Tabellen ist hier sehr einfach (Abb. 1-4-1). Bei der Vererbungsstruktur wird jede konkrete

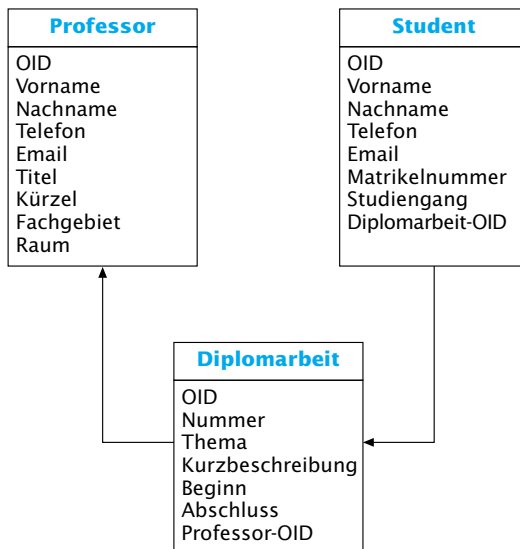


Abb. 4-1:  
Tabellenstruktur  
von Diploma

## Anhang 1 4 Datenhaltung mittels einer relationalen Datenbank

Unterklasse auf eine Tabelle abgebildet und die Attribute der abstrakten Klasse in beiden Tabellen aufgeführt. Diese Abbildung ist hier sinnvoll, weil die Klassen Professor und Student unterschiedliche Assoziationen zu Diplomarbeit besitzen. Da die Klassen nur einfache Attribute enthalten, müssen die Tabellen lediglich um die OID erweitert werden.

### b Formulieren des logischen Schemas in SQL

```
create table Professor
( OID number(10)      not null,
  Vorname char(30),
  Nachname char(30)   not null,
  Telefon char(30),
  Email char(30),
  Titel char(20),
  Kuerzel char(5)     not null,
  Fachgebiet char(50),
  Raum char(10)
);
create table Student
( OID number(10)      not null,
  Vornamechar(30),
  Nachname char(30)   not null,
  Telefon char(30),
  Email char(30),
  Matrikelnummer char(10),
  Studiengang char(30),
  Diplomarbeit_OID number(10)
);
create table Diplomarbeit
( OID number(10)      not null,
  Nummer number(4)    not null,
  Thema char(200)     not null,
  Kurzbeschreibung char(200),
  Beginn date,
  Abschluss date,
  Professor_OID number(10) not null
);
```

Folgende Indizes auf den Schlüsseln und Fremdschlüsseln – d.h. in diesem Fall den OID-Attributen – dienen der Optimierung. Beachten Sie, daß diese Indizes nur die Eindeutigkeit innerhalb einer Tabelle und nicht in der gesamten Datenbank – was eigentlich die Eigenschaft einer OID wäre – sichern. Die Werte der OID-Attribute werden normalerweise automatisch erzeugt.

```
create index ProfessorOID on Professor (OID);
create index StudentOID on Student (OID);
create index DiplomarbeitOID on Diplomarbeit (OID);
```

Folgende Indizes sorgen dafür, daß die fachlichen Schlüsselattribute eindeutig sind:

```
create unique index ProfessorKuerzel on Professor (Kuerzel);
create unique index StudentMatrikelnummer
    on Student (Matrikelnummer);
create unique index Diplomarbeitnummer on Diplomarbeit (Nummer);
```

### c Realisierung eines externen Schemas

Die Definition externer Schemata (Sichten) ist bei diesem Fallbeispiel nicht notwendig.

### d Realisierung der Operationen in SQL

Wir realisieren die folgenden Anfragen mittels SQL:

```
– verfügbare Arbeiten
select Thema, Nachname, Email
from Diplomarbeit, Professor
where (Beginnis is null);

– abgeschlossene Arbeiten
select Thema, p.Nachname, s.Nachname, d.Abschluss
from Diplomarbeit d, Professor p, Student s
where ((Beginnis not null) and
      (Abschluss is not null) and
      (s.Diplomarbeit_OID = d.OID));

– aktuelle Arbeiten
select Thema, p.Nachname, s.Nachname, d.Beginn
from Diplomarbeit d, Professor p, Student s
where ((Beginn is not null) and
      (Abschluss is null) and
      (s.Diplomarbeit_OID = d.OID));
```

## 5 Datenhaltung mittels einer objektorientierten Datenbank

Die nächste Aufgabe besteht darin, die Datenhaltung von Diploma mittels einer objektorientierten Datenbank zu spezifizieren. Wir betrachten die

**a** Schemadefinition mittels ODL gemäß ODMG-Standard 2.0 und die  
**b** Schemadefinition mit einer erweiterten Programmiersprache für Poet.

**a** Das Klassendiagramm wird in ODL entsprechend dem ODMG-Standard 2.0 wie folgt beschrieben, wobei hier auf die Angabe der Operationen verzichtet wird.

```
class Diplomarbeit
( extent Diplomarbeiten )
{ attribute Long Nummer,
  attribute string Thema;
  attribute string Kurzbeschreibung;
  attribute date Beginn;
```

## Anhang 1 5 Datenhaltung mittels einer objektorientierten Datenbank

```
attribute date Abschluss;
relationship set <Student> wirdBearbeitetVon
    inverse Student::bearbeitet;
relationship Professor wirdBetreutVon
    inverse Professor::betreut;
}
class Person
{ attribute string Vorname;
  attribute string Nachname;
  attribute string Telefon;
  attribute string Email;
}
class Professor
( extent Professoren
  key Kuerzel )
{ attribute string Titel;
  attribute string Kuerzel;
  attribute string Fachgebiet;
  attribute string Raum;
  relationship set<Diplomarbeit> betreut
    inverse Diplomarbeit::wirdBetreutVon;
}
class Student
( extent Studenten
  key Matrikelnummer )
{ attribute string Matrikelnummer;
  attribute string Studiengang;
  relationship Diplomarbeit bearbeitet
    inverse Diplomarbeit::wirdBearbeitetVon;
}
```

### b Schemadeklaration für Poet

Wird anstelle der programmiersprachen-unabhängigen ODL eine erweiterte Programmiersprache gewählt, so ergibt sich bei Verwendung des objektorientierten Datenbanksystems Poet für C++ nachfolgende Spezifikation. Diese Schemadeklaration wird dann vom Poet-Präprozessor in C++ transformiert und kann mit einer Entwicklungsumgebung weiter bearbeitet werden. Für die Realisierung der Assoziationen werden die von Poet angebotenen *ondemand*-Referenzen verwendet. Anstelle der verwendeten Poet-Typen könnten auch die ODMG-Typen (z.B. `d_String`, `d_Date`) gewählt werden. Weitere Informationen zur Realisierung der Datenhaltung mittels Poet

Exkurs 2 finden Sie im Exkurs 2.



```
_persistent class Person // abstrakte Klasse
{protected:
    PtString Vorname,
            Nachname,
            Telefon;

public:
    ... Operationen in C++Syntax...
};
```

## 5 Datenhaltung mittels einer objektorientierten Datenbank Anhang 1

```
persistent class Professor: public Person
{protected:
    PtString Titel,
                Fachgebiet,
                Raum,
                Kennwort;
    cset <ondemand <Diplomarbeit>> betreut; //Professor ->
                                           /* Diplomarbeit
public:
    //Assoziati on zu Diplomarbeit
    void link (Diplomarbeit* pDiplomarbeit);
    int getLink(Diplomarbeit* &pDiplomarbeit, int position);
    void unlink (Diplomarbeit* pDiplomarbeit);
    ... weitere Operationen in C++Syntax ...
};

persistent class Student: public Person
{protected:
    PtString Matrikelnummer,
                Studiengang;
    ondemand <Diplomarbeit> bearbeitet; //Student schreibt
                                           //0..1 Diplomarbeit
public:
    //Assoziati on zu Diplomarbeit;
    void link (Diplomarbeit* pArbeit);
    void getLink (Diplomarbeit* &pArbeit);
    void unlink (Diplomarbeit* pArbeit);
    ... weitere Operationen in C++ Syntax ...
};

persistent class Diplomarbeit
{protected:
    PtString Thema,
                Kurzbeschreibung;
    PtDate Beginn, //selbstdefinierter Datentyp
                Abschluss;

    ondemand <Professor> wIRDbetreutVon; //Diplomarbeit ->
                                           //1 Professor
    cset <ondemand <Student>> wIRDBearbeitetVon; //Diplomarbeit ->
                                           /* Student
public:
    ... Operationen in C++Syntax ...

    //Assoziati on zu 1 Professor
    void link (Professor* pProf);
    void getLink (Professor* &pProf);
    void unlink (Professor* pProf);

    //Assoziati on zu * Student
    void link (Student* pStud);
    int getLink (Student* &pStud, int position);
    void unlink (Student* pStud);
};
```

## Anhang 1 5 Datenhaltung mittels einer objektorientierten Datenbank

```
typedef cset <Professor*> ProfessorCSet;  
typedef cset <Student*> StudentCSet;  
typedef cset <Diplomarbeiter*> DiplomarbeiterCSet;
```

Indizes Analog zu den SQL-Datenbanken ist auch bei objektorientierten Datenbanken die Definition von Indizes von Bedeutung. Sie werden u.a. benötigt, um Anfragen (*queries*) zu optimieren. Beim Entwurf der Datenbank sollten Sie also sorgfältig überlegen, wo ein Index sinnvoll ist.

Um beispielsweise für den Nachnamen von Professoren und Studenten einen Index zu definieren, müssen Sie die obige Schema-deklaration für den Poet-Präprozessor wie folgt erweitern (alle Erweiterungen der Übersichtlichkeit halber fett gedruckt).

```
persistent class Diplomarbeiter  
{  
    ...  
    PtString Thema;  
    useindex ThemaIX;  
}  
indexdef ThemaIX: Diplomarbeiter  
{  
    Thema;  
};
```

Analog kann ein Index für die Nachnamen von Professoren (erbt von Person) definiert werden.

```
persistent class Professor: Person  
{  
    ...  
    useindex NachnameIX;  
}  
indexdef NachnameIX : Professor  
{  
    Nachname;  
};
```

Der folgende Index besitzt zusätzlich die Eigenschaft der Eindeutigkeit:

```
persistent class Student: Person  
    ...  
    PtString Matrikelnummer;  
    useindex MatrikelnummerIX;  
}  
unique indexdef MatrikelnummerIX: Student  
{  
    Matrikelnummer;  
};
```

## 6 OOD-Modell und Implementierung

Die hier beschriebene Fallstudie wurde von Herrn Diplom-Informatiker Andreas Schröter programmiert. Das OOD-Klassendiagramm wurde wegen seiner Größe mit einem Werkzeug erstellt. Für die Implementierung wurden Microsoft Visual Studio C++ 5.0 und die objektorientierte Datenbank Poet 5.0 gewählt.



Das OOD-Modell und der in C++ erstellte Quellcode befinden sich auf der beiliegenden CD im Verzeichnis *Anhang 1 – Fallbeispiel*. Wenn Sie zuvor die Exkurse 1 und 2 durcharbeiten, werden Sie keine Probleme haben, diese Implementierung zu verstehen.

Anhang 1 –  
Fallbeispiel

# Anhang 2: Lösungen

## LE 1

### Aufgabe 1

- a** Die gute Durchgängigkeit wird bei der objektorientierten Entwicklung dadurch erreicht, daß in allen Phasen dieselben Konzepte verwendet werden. Analyse und Entwurf können in der gleichen Notation erfolgen.
- b** Eine Klasse kann leicht verstanden und geändert werden, ohne daß andere Klassen davon stark betroffen sind. Dadurch wird die Wartbarkeit unterstützt.
- c** Eine gute Vererbungshierarchie fördert Erweiterbarkeit und Wiederverwendung.

### Aufgabe 2

Bevor wir mit der Programmierung beginnen, sind zwei vorbereitende Schritte notwendig. Zuerst kommt die sogenannte Analyse. Wir interviewen Sie und Ihre Mitarbeiter, um genau herauszufinden, was Sie von Ihrem zukünftigen Warenwirtschaftssystem erwarten. Diese Informationen schreiben wir auf. Damit wir nicht aneinander vorbeireden machen wir zunächst einen Prototyp, der alle Bildschirmmasken erhält. Sie können nun prüfen, ob alle notwendigen Daten vorhanden sind. Alle Ihre Änderungswünsche schreiben wir wieder auf und wir erstellen einen verbesserten Prototypen. Diese Schritte wiederholen sich, bis Sie zufrieden sind. Auch jetzt fangen wir noch nicht mit der Programmierung an, sondern beschäftigen uns mit dem sogenannten Entwurf. Hier überlegen wir, wie die Programme grundsätzlich zu strukturieren sind, ob wir eine Datenbank einsetzen und prüfen, welche Hilfsmittel wir noch verwenden können. Dann erst fangen wir mit der Programmierung an.

### Aufgabe 3

- a** Der Prototyp der Benutzungsoberfläche dient der Validierung der »wahren Wünsche« des Auftraggebers. Anhand des Prototyps kann er seine Wünsche besser artikulieren, die dann vom Systemanalytiker in das OOA-Modell übertragen werden.
- b** Das Pflichtenheft bildet einerseits das »Einstiegsdokument« für das Projekt in der späteren Wartung und ist andererseits die Voraussetzung für die Erstellung des OOA-Modells.
- c** In der Analyse sollen die essentielle Struktur und die Semantik des Problems dokumentiert, aber noch keine technische Lösung



## Anhang 2 Lösungen LE 1

für ein bestimmtes Computersystem erarbeitet werden. Wir können in der Analyse auch von einer fachlichen Lösung sprechen. Während man in der Analyse von einer idealen Umgebung ausgeht, ist es Aufgabe des Entwurfs die spezifizierte Anwendung auf einer Plattform unter den geforderten technischen Randbedingungen zu konstruieren, jedoch noch nicht zu programmieren.

- d** Wenn in der Analyse bereits auf Implementierungsdetails – z.B. Verwendung der Datenbank *xy* – eingegangen wird, dann besteht die Gefahr, daß das OOA-Modell speziell auf die Anforderungen dieser Datenbank abgestimmt ist. Bei späteren Anpassungen an andere Datenbanken sind dann umfangreiche Änderungen notwendig.
- e** Bei der Systemanalyse muß hochgradig abstrahiert werden. Gleichzeitig soll ein OOA-Modell spezifiziert werden, das den Anforderungen des Auftraggebers entspricht und realisierbar ist. Um beides zu garantieren muß der Systemanalytiker ein Abbild des späteren Ablaufs im Kopf haben, wenn er ein OOA-Modell erstellt. Desweiteren muß der Systemanalytiker, der die Informationen vom Auftraggeber »ungefiltert« erhält, die »Spren vom Weizen trennen«, d.h. erkennen, welche Informationen für das OOA-Modell relevant sind und welche Informationen noch fehlen, die dann beim Auftraggeber erfragt werden müssen.
- f** Oft betreffen Änderungen nur die Benutzungsoberfläche, z.B. wenn das System auf eine andere Plattform portiert werden muß, für die andere *style guides* (Gestaltungsrichtlinien der Benutzungsoberfläche) gelten. Ist die Benutzungsoberfläche nicht in einer separaten Schicht verkapselt, sind dann umfangreiche Änderungen notwendig. Im Extremfall muß das ganze System neu geschrieben werden.

### Aufgabe 4

- a** Welche Daten aus Benutzersicht zu speichern sind, wird in der Analyse im Pflichtenheft (Kapitel Daten) und im OOA-Modell festgehalten.
- b** In der Analyse wird im Pflichtenheft die gewünschte Datenbank als Information festgehalten, wenn es sich um eine Vorgabe des Auftraggebers handelt. *Wie* Informationen in der Datenbank gespeichert werden, ist erst Gegenstand des Entwurfs und der Implementierung.
- c** Hier handelt es sich um eine Benutzerfunktion, die in der Analyse im Pflichtenheft und im OOA-Modell dokumentiert wird.
- d** Erst in der Implementierung wird festgelegt, wie das logische Löschen der Sätze realisiert wird.
- e** Hier handelt es sich um eine Zielsetzung, die in der Analyse im Pflichtenheft dokumentiert wird.

- f** Wie die Benutzungsschnittstelle realisiert wird, ist Gegenstand des Entwurfs.
- g** In der Systemanalyse wird von der Netzverteilung zunächst abstrahiert. Nur die Forderung einer Client-Server-Verteilung wird im Pflichtenheft festgehalten (z.B. Zugriff auf Videodaten von mehreren Arbeitsplätzen aus). Im Entwurf wird spezifiziert, *wie* die Verteilung durchzuführen ist.

## LE 2

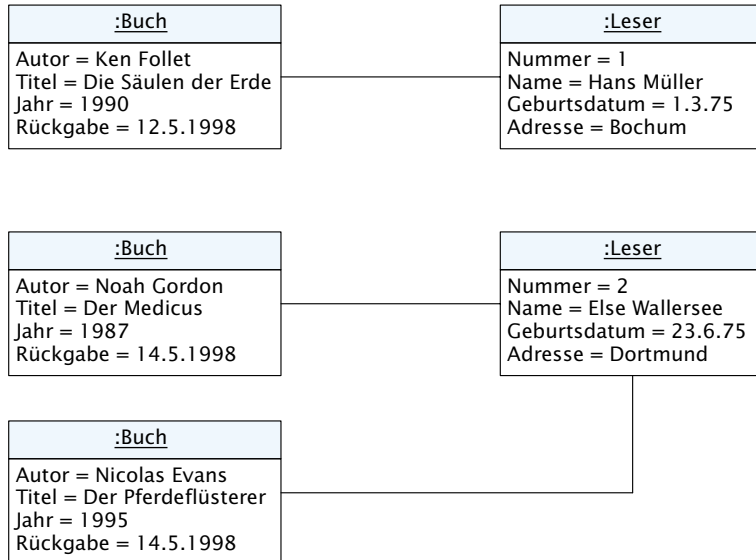
### Aufgabe 1

- a** Objektidentität bedeutet, daß alle Objekte aufgrund ihrer Existenz zu unterscheiden sind. Die Identität eines Objekts kann sich nicht ändern.
- b** Im Gegensatz zur Objektidentität identifiziert der Objektname ein Objekt innerhalb eines Objektdiagramms. Er kann in einem anderen Objektdiagramm ein anderes Objekt bezeichnen. Der Objektname besitzt eine semantische Bedeutung für den Leser.
- c** Eine Klasse definiert eine Kollektion gleichartiger Objekte und besitzt eine *object factory* zum Erzeugen neuer Objekte. Alle von dieser Klasse erzeugten Objekte bilden die Menge aller Objekte dieser Klasse.
- d** Ein Klassenattribut liegt vor, wenn nur ein Attributwert für alle Objekte der Klasse existiert.
- e** Der Wert eines abgeleiteten Attributs kann jederzeit aus anderen Attributwerten berechnet werden.
- f** Eine Klassenoperation ermöglicht die Manipulation von Klassenattributen. In der Systemanalyse nutzen wir die Eigenschaft der Objektverwaltung aus, und verwenden Klassenoperationen, um die Menge aller Objekte der Klasse zu manipulieren.
- g** Verwaltungsoperationen sind einfache Operationen zum Erzeugen und Löschen von Objekten, Lesen und Schreiben von Attributen und Erstellen, Entfernen und Lesen von Objektverbindungen. Sie werden *nicht* in das Klassendiagramm eingetragen.

## Anhang 2 Lösungen LE 2

### Aufgabe 2

Abb. LE2-2:  
Objektdiagramm



### Aufgabe 3

Abb. LE2-3 zeigt das Klassendiagramm mit den beiden Klassen `Studentische Hilfskraft` und `Angestellter`. Weil der Stundenlohn für alle studentischen Hilfskräfte gleich ist, wird er als Klassenattribut modelliert. Als Typ wurde für den Stundenlohn `Fixed(2, 2)` gewählt. Damit wird eine genaue Darstellung auf den Pfennig genau ohne Rundungsfehler gefordert. Beachten Sie, daß in vielen Beispielen für Geldbeträge der Typ `Float` verwendet wird. Das ist bei Beispielen für Geldbeträge der Typ `Float` verwendet wird. Das ist bei Beispielen für Geldbeträge der Typ `Float` verwendet wird. Für die Stundenzahl wird der Typ `Int` verwendet, weil davon ausgegangen wird, daß nur ganze Stunden vereinbart werden.

**Klassendiagramm**

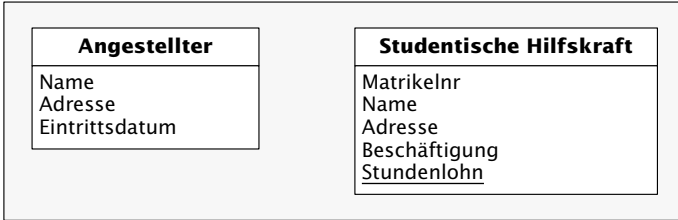


Abb. LE2-3:  
Klassendiagramm,  
Klassenbeschreibung  
und Spezifikation  
von Attributen

**Klassenbeschreibung**

Studentische Hilfskraft: Verwaltet Studenten, die zeitlich begrenzt als Hilfskraft an der Hochschule tätig sind  
 Angestellter: Verwaltet alle Angestellten einer Hochschule

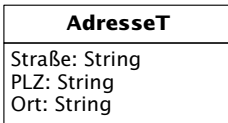
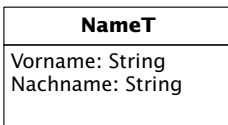
**Studentische Hilfskraft**

Matrikelnr: String { mandatory, key, frozen}  
 Name: NameT {mandatory}  
 Adresse: AdresseT  
 Beschäftigung: **list of** ArbeitsvertragT  
 Stundenlohn: Fixed (2,2)

**Angestellter**

Name: NameT {mandatory}  
 Adresse: AdresseT  
 Eintrittsdatum: Date

**Elementare Klassen**



**Aufgabe 4**



Abb. LE2-4: Objekt-  
und Klassen-  
attribute

## Anhang 2 Lösungen LE 2

### Aufgabe 5

Abb. LE2-5: Klasse Artikel

Artikel
Nummer Bezeichnung EK-Preis VK-Preis Mindestbestand Maximalbestand Bestand
erfassen () ändern Preise () buche Zugang () buche Abgang () <u>drucke Bestellvorschlag ()</u> <u>drucke Liste ()</u>

Die Attribute der Klasse Artikel werden wie folgt spezifiziert:

Nummer: UInt {mandatory, key, frozen} //alternativer Typ: String  
Bezeichnung: String {mandatory}  
EK-Preis: Fixed (5, 2)  
VK-Preis: Fixed (5, 2)  
Mindestbestand: UInt  
Maximalbestand: UInt  
Bestand: UInt = 0

## LE 3

### Aufgabe 1

Beim Anmelden eines Teilnehmers durch die Operation `anmelden()` werden die Verbindungen zu den entsprechenden Tutorium- und Rahmenprogramm-Objekten aufgebaut.

Da die Kosten für alle Tutorien gleich sind, wird dieses Attribut als Klassenattribut gespeichert. Die Operation `findetstatt()` prüft, ob von den jeweiligen Tutorium-Objekt mindestens 10 Verbindungen zu Teilnehmer-Objekten ausgehen. Wenn sich ein Referent für ein oder mehrere Tutorien anmeldet, wird die Verbindung zu diesen Tutorium-Objekten aufgebaut. Es gibt keine Klasse `Tagung`, weil hierüber keine Attribute zu speichern sind. Anhand des Objektdiagramms wird deutlich, daß die Restriktion {Vortragender ≠ Zuhörer} notwendig ist.

Klassendiagramm

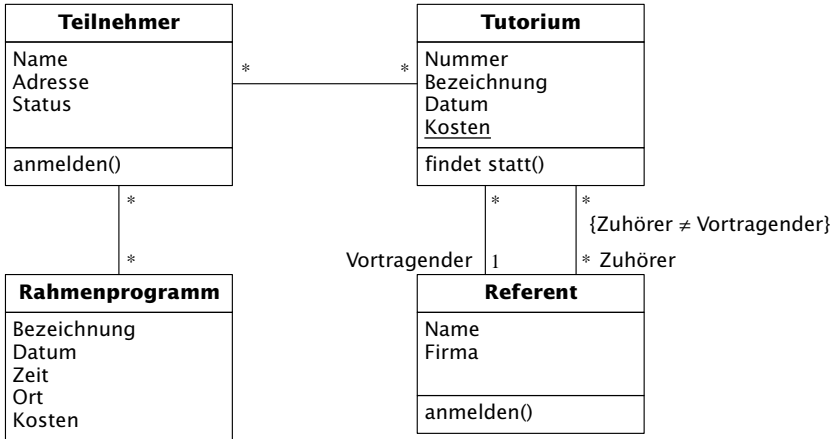
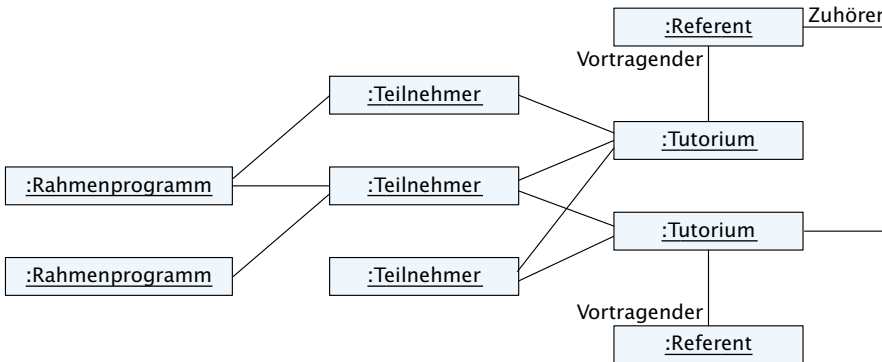


Abb. LE3-1:  
Klassendiagramm  
und Objekt-  
diagramm für  
Tagungs-  
organisation

Objektdiagramm



### Aufgabe 2

Weil bei der Eröffnung eines Kontos gleich eine Einzahlung (=Kontobewegung) vorzunehmen ist, wird bei Konto die Kardinalität 1..\* eingetragen.

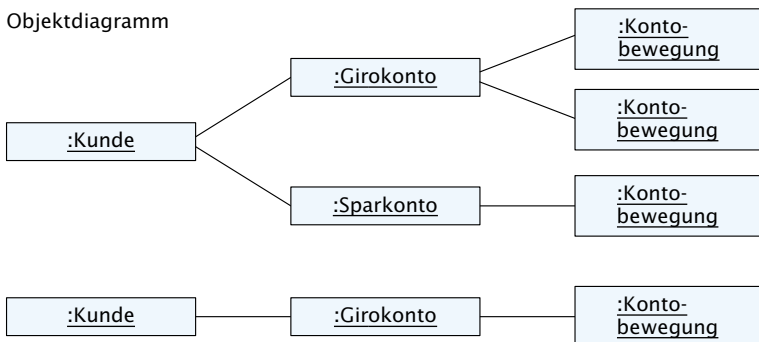
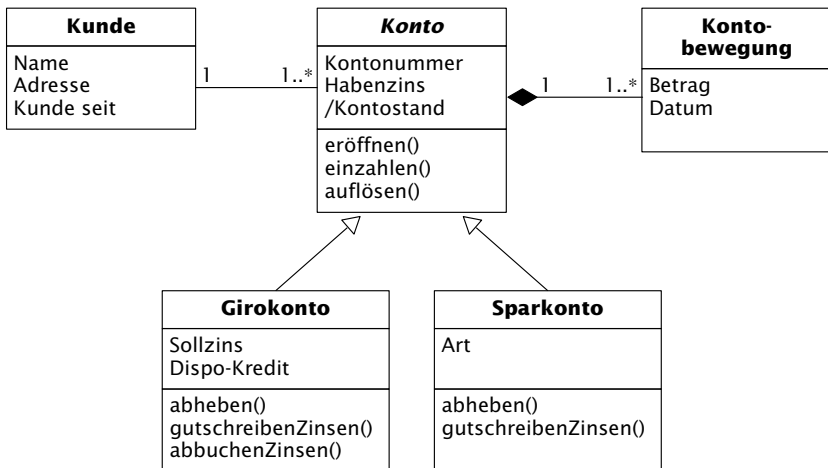
Zwischen Kunde und Konto besteht eine einfache Assoziation, weil weder der Kunde Teil eines Kontos ist, noch umgekehrt. Zwischen Konto und Kontobewegung existiert eine Komposition weil gilt: es liegt eine *whole-part*-Beziehung vor, jede Kontobewegung ist Teil genau eines Kontos und die dynamische Semantik eines Kontos gilt stets für alle Kontobewegungen. Wenn beispielsweise das Konto bei einer anderen Filiale der Bank weitergeführt werden soll, dann werden auch alle Kontobewegungen der neuen Filiale zugeordnet. Die Kontobewegungen werden für die Berechnung der Zinsen benötigt. Die Operation `einzahlen()` ist für Giro- und Sparkonten gleich. Deshalb wird sie bei der Basisklasse eingetragen und vererbt. Beim Abheben verhalten sich Giro- und Sparkonto jedoch unterschiedlich. Beim

## Anhang 2 Lösungen LE 3

Sparkonto kann nur bis zu vereinbarten Höchstbetrag auf einmal abgehoben werden, sonst fallen zusätzliche Gebühren an. Beim Girokonto steht ein Dispo-Kredit zur Verfügung. Auch das Verhalten der Operation gutschreiben Zinsen() ist unterschiedlich, da dies einmal quartalsweise und einmal jährlich erfolgt. Das Attribut Kontostand ist abgeleitet, weil es sich aus der Menge aller Kontobewegungen des jeweiligen Kontos errechnen läßt.

Beachten Sie, daß die Komposition von Konto zu Kontobewegung an die beiden Unterklassen vererbt wird, d.h. jedes Sparkonto- und jedes Girokonto-Objekt besitzt eine Menge von Kontobewegungen. Konto ist eine abstrakte Klasse, weil es keine Konto-Objekte gibt, die nicht bereits den Klassen Girokonto oder Sparkonto angehören. Analoges gilt für die Vererbung der Assoziation zu Kunde. Die Vererbung der Assoziation bedeutet, daß Verbindungen zwischen Kunde und Girokonto etc. existieren. Anhand des Objektdiagramms wird deutlich, daß es keine Objekte der Klasse Konto gibt.

Abb. LE3-2: Klassendiagramm und Objektdiagramm für die Kontoverwaltung



### Aufgabe 3

Es lassen sich drei Pakete bilden:

Paket Lieferantenverwaltung

- Lieferant
- Lieferkondition
- Bestellung an Lieferanten
- Bestellposten

Paket Artikel- und Lagerverwaltung

- Artikel
- Bestellartikel
- Lagerartikel
- Lager
- Lagerplatz
- Lagerverwalter

Paket Kundenverwaltung

- Kunde
- Kundenauftrag
- Auftragsposten

## LE 4

### Aufgabe 1

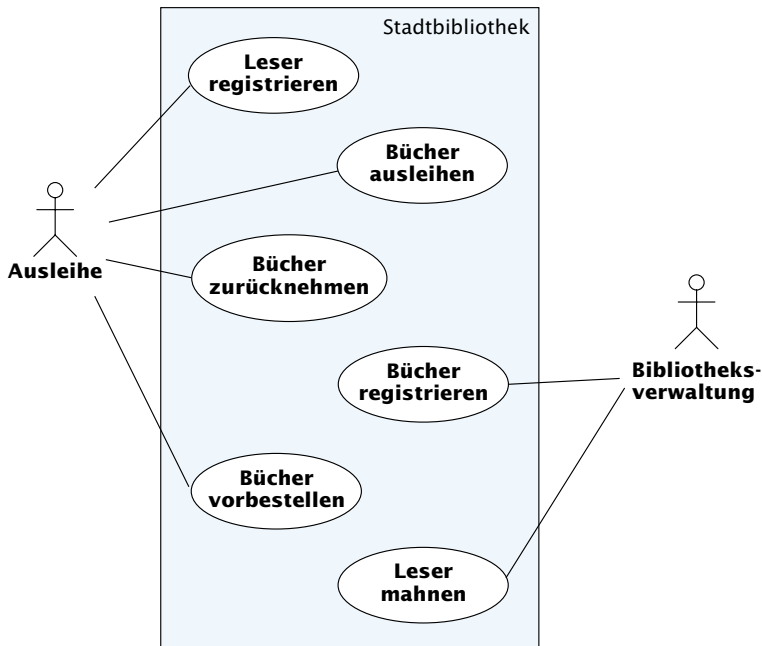


Abb. LE4-1:  
Geschäftsprozeß-  
diagramm einer  
Stadtbibliothek



## Anhang 2 Lösungen LE 4

### Aufgabe 2

Es läßt sich ein Geschäftsprozeß identifizieren, der mittels Schablone spezifiziert wird.

*Geschäftsprozeß:* Anmeldung bearbeiten

*Ziel:* Teilnehmer zum Seminar anmelden

*Vorbereitung:* –

*Nachbedingung Erfolg:* Anmeldebestätigung verschiekt

*Nachbedingung Fehlschlag:* Nachfrage beim Kunden, Absage an Kunden

*Akteure:* Kundensachbearbeiter

*Auslösendes Ereignis:* Seminaranmeldung des Kunden liegt vor

*Beschreibung:*

- 1 Neukunden erfassen
- 2 Feststellen, daß Seminar existiert und noch frei ist
- 3 Buchung durchführen und Anmeldebestätigung erstellen

*Erweiterung:*

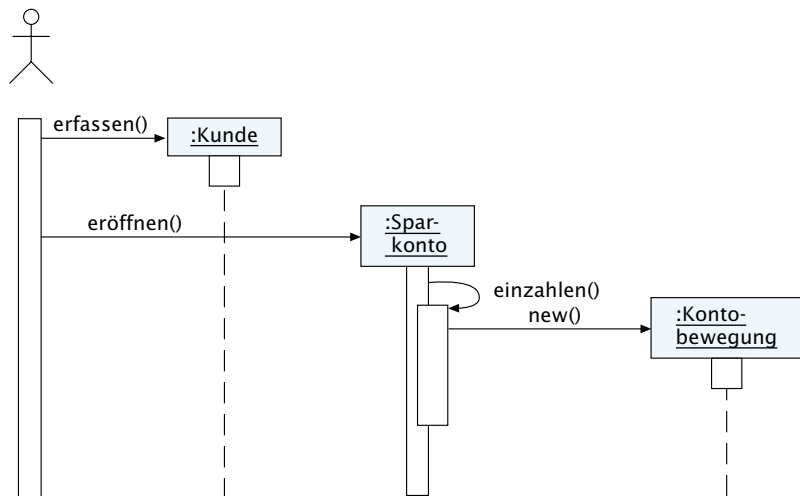
*Alternativen:*

- 1a Kundendaten abrufen
- 1b Kundendaten abrufen und aktualisieren
- 3a Absage an Kunden und Angebot neuer Termine
- 3b Nachfrage beim Kunden

### Aufgabe 3

Szenario: Neuer Kunde eröffnet ein Sparkonto

Abb. LE4-3:  
Sequenzdiagramm  
für das Eröffnen  
eines Kontos und  
zugehöriges  
Klassendiagramm



**Aufgabe 4**

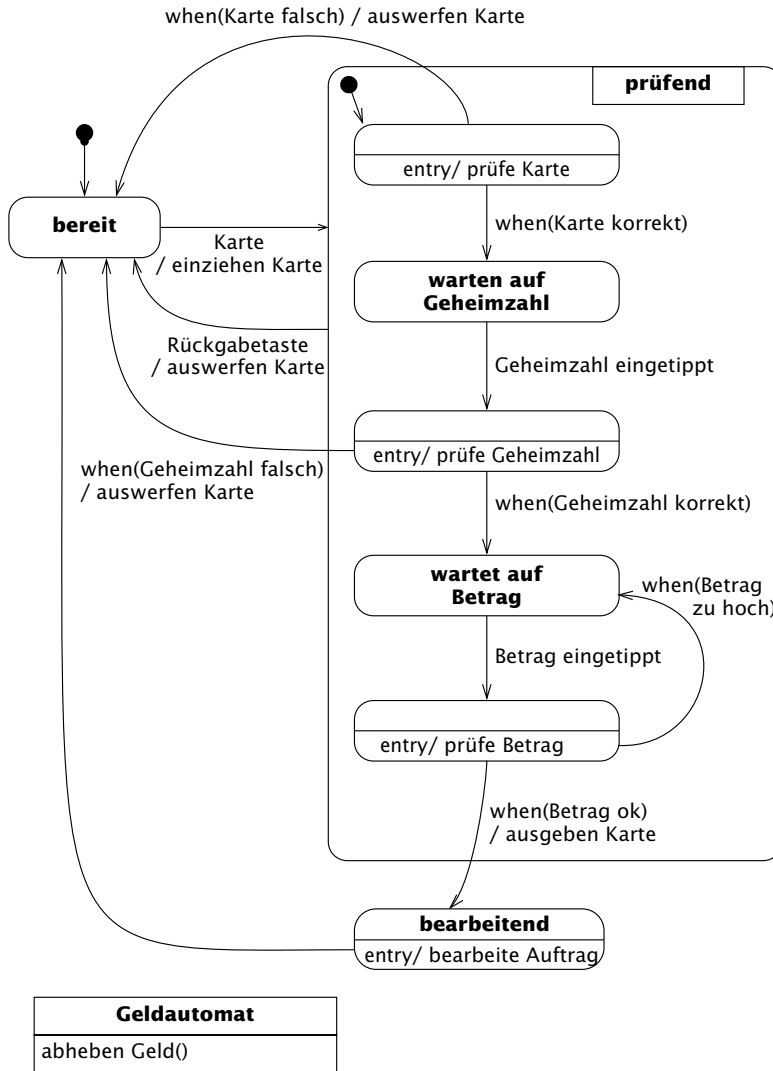
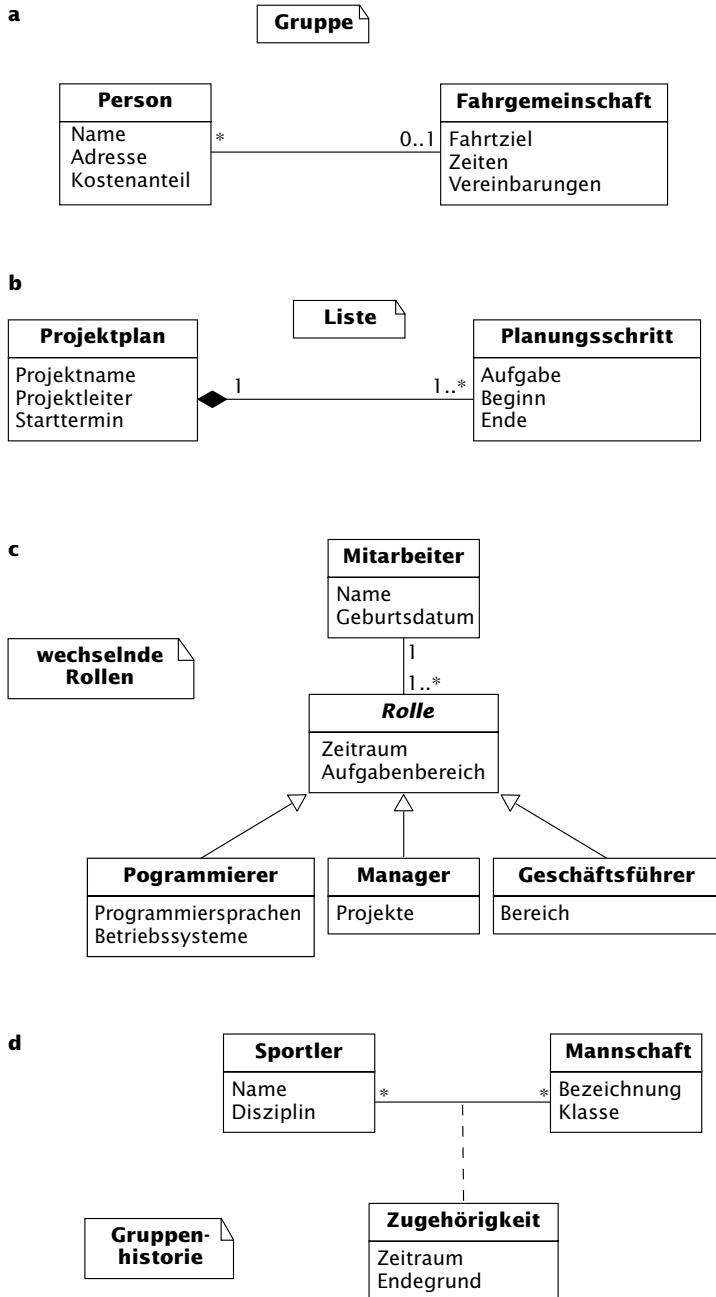


Abb. LE4-4:  
Zustandsdiagramm  
zur Spezifikation  
der Operation  
abheben Geld()

LE 5

Aufgabe 1

Abb. LE5-1a:  
Klassendiagramme  
zu den Problem-  
stellungen a bis d



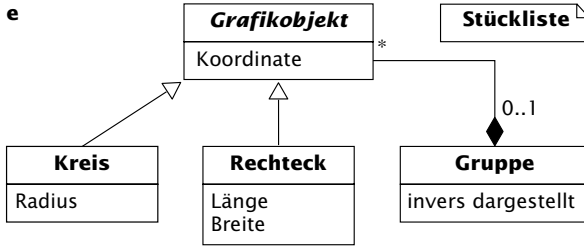
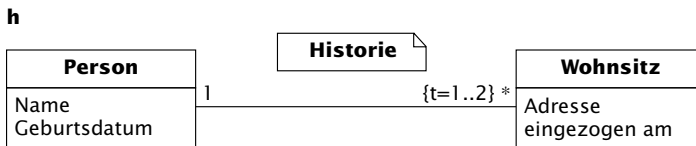
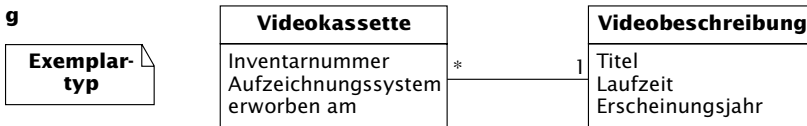
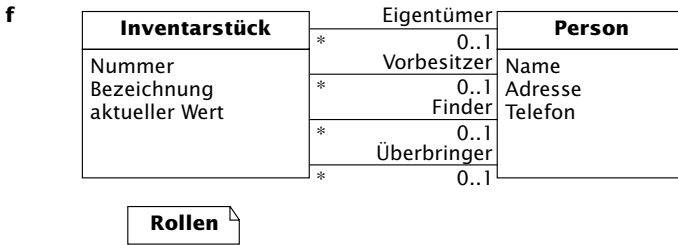


Abb. LE5-1b:  
Klassendiagramme  
zu den Problem-  
stellungen e bis h



## Aufgabe 2

a

Im Klassendiagramm der Seminarorganisation lassen sich folgende Muster aus Kapitel 3.1 identifizieren:

Exemplartyp: Seminartyp – Seminar

Koordinator: Firma – Firmenbuchung – internes Seminar

Koordinator: Kunde – Kundenbuchung – öffentliches Seminar

Gruppe: Kunde – Firma

Rollen: Kunde (Teilnehmer, Debitor) – Kundenbuchung

Rollen: Dozent (Referent, Seminarleiter) – Seminar

## Anhang 2 Lösungen LE 5

**b**

Im Klassendiagramm der Friseursalonverwaltung lassen sich folgende Muster aus Kapitel 3.1 identifizieren:

Exemplartyp: Dienstleistungsgruppe – Dienstleistung

Koordinator: Kunde – Salonbesuch (Koordinator-Klasse) – Kundenmitarbeiter – Dienstleistung

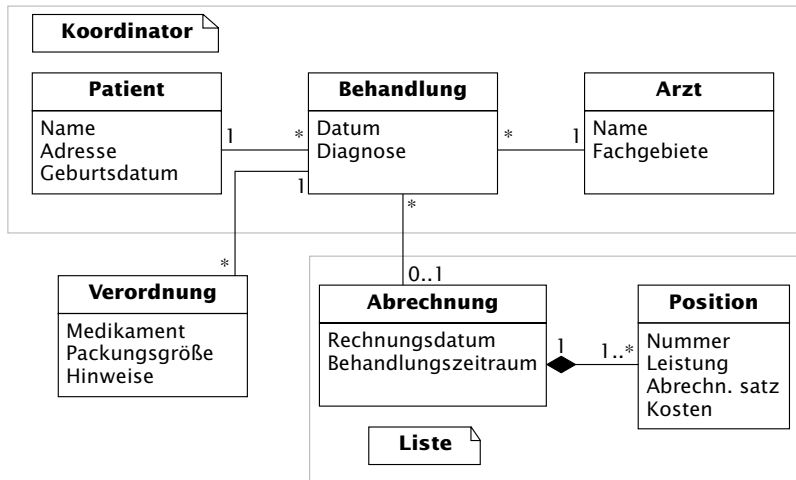
Koordinator: Artikel – Verkauf – Kundenmitarbeiter

Koordinator: Kunde – Abonnement – Dienstleistung

Historie: Mitarbeiter – Anwesenheit

### Aufgabe 3

Abb. LE5-3:  
Klassendiagramm  
mit Mustern



## LE 6

### Aufgabe 1

**a** Zuerst wird ein überschaubarer Teil des Systems modelliert (Version 0.x), der dann entworfen und implementiert wird. Durch Verfeinerung und Erweiterung von Version 0.x entsteht die Version 0.(x+1). Mit jeder Iteration wird das Verständnis des Problemereichs besser und falsche bzw. ungünstige Entscheidungen können korrigiert werden.

**b** Das Aufstellen von Geschäftsprozessen ist unabhängig von den objektorientierten Techniken, auch wenn sie gerade bei der Objektorientierung intensiv verwendet werden und die UML eine eigene Notation dafür vorsieht. Die Bildung von Paketen ist ebenfalls unabhängig von der Objektorientierung. Hier handelt es sich im Grund um die klassische Bildung von Teilsystemen.

Das Erstellen von Zustandsautomaten ist abhängig von der Objektorientierung, wenn der Lebenszyklus einer Klasse beschrieben wird. Zustandsautomaten können aber auch bei der klassi-

schen Softwareentwicklung zur Beschreibung komplexer Funktionen verwendet werden.

- c Wir können hier die vorgeschlagene Vorgehensweise gut einsetzen. Die Benutzer werden nach ihren typischen Arbeitsabläufen befragt. Aus diesen Interviews, dem laufenden System und dem Benutzerhandbuch lassen sich Geschäftsprozesse ableiten. Die Dateibesreibungen werden für den anschließenden Schritt – die Erstellung des Klassendiagramms – verwendet.
- d Die Checklisten geben Ihnen eine Hilfestellung, ohne Sie andererseits einzuengen. Außerdem sorgen Sie für einen gewissen Standard bei Erstellung und Qualitätsanalyse.

## Aufgabe 2

*Geschäftsprozess:* kassieren Laufkundschaft

*Ziel:* anonymer Kunde kauft im Salon Artikel

*Vorbereitung:* –

*Nachbereitung Erfolg:* Kassenbeleg erstellt und Betrag kassiert

*Nachbereitung Fehlschlag:* Zahlungsmittel des Kunden nicht akzeptiert

*Akteure:* Rezeptionist

*Auslösendes Ereignis:* Kunde will Artikel kaufen

*Beschreibung:*

1 alle Artikelnummern, Preise und die jeweilige Anzahl eingeben

2 Geld des Kunden annehmen und Restbetrag errechnen

3 Kassenbeleg ausdrucken und Lagerbestand aktualisieren

*Erweiterung:*

1a bei größeren Mengen und bestimmten Artikeln Sonderpreise eingeben

2a bargeldlose Zahlungsmittel prüfen. Wenn keine Akzeptanz, dann alle Eingaben stornieren

## Aufgabe 3

Art	Ereignis	Geschäftsprozess
extern	neuer Leser	registrieren des Lesers
extern	Leser leiht Buch aus	ausleihen von Büchern
extern	Leser gibt Buch zurück	zurückgeben von Büchern
zeitlich	Rückgabedatum überschritten	mahnen des Lesers
extern	Leser macht Vorbestellung	eintragen der Vorbestellung
zeitlich	Abholfrist abgelaufen	prüfen der Abholfristen
extern	Buch trifft ein	erfassen neuer Bücher

*Geschäftsprozess:* registrieren des Lesers

*Beschreibung:* Prüfen, ob der Leser bereits registriert ist. Falls nein, dann werden dessen Name und Adresse gespeichert und ein Ausweis erstellt.

*Geschäftsprozess:* ausleihen von Büchern

*Beschreibung:* Prüfen, ob es sich um einen registrierten Leser handelt. Falls ja, dann werden Ausleihdatum und Rückgabedatum gespeichert.

## Anhang 2 Lösungen LE 6

*Geschäftsprozeß:* zurückgeben von Büchern

*Beschreibung:* Wenn das Buch vorbestellt ist, dann wird der erste Leser der Liste benachrichtigt und das Buch bereitgelegt. Alle anderen Bücher werden in die Ausleihe zurückgegeben.

*Geschäftsprozeß:* eintragen Vorbestellung

*Beschreibung:* Wenn das gewünschte Buch ausgeliehen ist, dann wird der Leser auf die Warteliste gesetzt.

*Geschäftsprozeß:* prüfen der Abholfristen

*Beschreibung:* Wird ausgelöst, wenn das vorbestellte Buch eine Woche bereitliegt. Das System prüft, ob für das nicht abgeholte Buch weitere Vorbestellungen vorliegen. Falls ja, dann wird der nächste Leser der Liste benachrichtigt. Andernfalls wird das Buch in die Ausleihe zurückgegeben.

*Geschäftsprozeß:* erfassen neuer Bücher

*Beschreibung:* Jedes Buch erhält eine eindeutige Inventarnummer und kommt zunächst in die Ausleihe.

*Geschäftsprozeß:* mahnen des Lesers

*Beschreibung:* Wird ausgelöst, wenn das Rückgabedatum eines Buchs um eine Woche überschritten ist.

### Aufgabe 4

*Geschäftsprozeß:* erstellen Veranstaltungsplan

*Ziel:* Veranstaltungsplan für das nächste Semester

*Vorbereitung:* Dozenten, Räume und Veranstaltungstypen existieren

*Nachbereitung Erfolg:* Veranstaltungsplan liegt vor

*Akteure:* Fachbereich

*Auslösendes Ereignis:* festgesetzter Termin

*Beschreibung:*

- 1 trage für jeden Pflicht-Veranstaltungstyp eine Veranstaltung ein
- 2 trage ggf. für optionale Veranstaltungstypen Veranstaltungen ein
- 3 weise Dozent und Raum zu
- 4 stimme den Plan mit den Dozenten ab
- 5 erstelle endgültigen Veranstaltungsplan

*Geschäftsprozeß:* erstellen Prüfungsplan

*Ziel:* Prüfungsplan für aktuelles Semester

*Vorbereitung:* Veranstaltungsplan liegt vor

*Akteure:* Prüfungsausschuß

*Auslösendes Ereignis:* festgesetzter Termin

*Beschreibung:*

- 1 aus dem Veranstaltungsplan aktuelle Veranstaltungen und Prüfer entnehmen
- 2 Datum, Zeit und Raum jeder Prüfung eintragen
- 3 erstellen einer – noch leeren – Zulassungsliste für jede Prüfung

*Geschäftsprozeß:* anmelden zu Prüfungen

*Ziel:* Student zu Prüfungen zugelassen

*Vorbereitung:* Student ist immatrikuliert, aktueller Prüfungsplan vorhanden

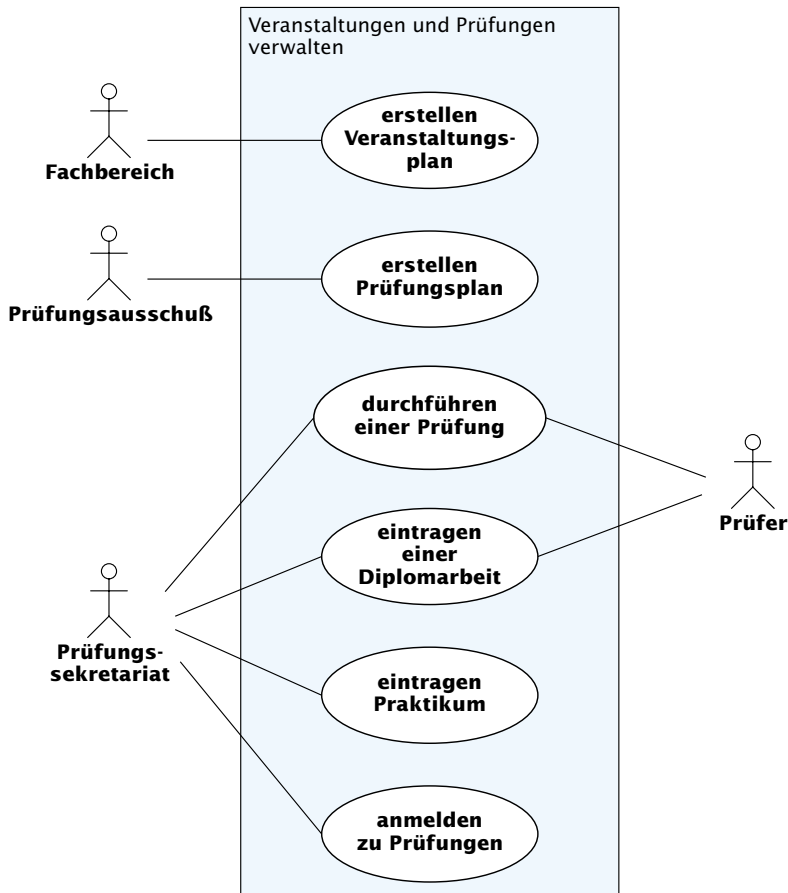


Abb. LE6-4:  
Geschäftsprozess-  
diagramm  
der Hochschul-  
verwaltung

*Nachbedingung Erfolg:* Eintrag in Zulassungsliste

*Nachbedingung Fehlschlag:* Benachrichtigung des Studenten über fehlende Voraussetzungen

*Akteure:* Prüfungssekretariat

*Auslösendes Ereignis:* Anmeldeformular des Studenten

*Beschreibung:*

- 1 Anmeldeformular auf Vollständigkeit und Korrektheit prüfen
- 2 prüfen, ob die Teilnahmevoraussetzungen erfüllt sind
- 3 ermitteln, der wievielte Versuch es ist
- 4 Eintrag in die Zulassungslisten der gewünschten Prüfungen

*Erweiterungen:*

- 1a Bei fehlenden und fehlerhaften Angaben beim Studenten nachfragen
- 4a Wenn für eine oder mehrere Prüfungen die Zulassungsvoraussetzungen fehlen, wird der Student benachrichtigt

*Geschäftsprozess:* durchführen einer Prüfung

*Ziel:* Prüfungsergebnisse den Studenten bekannt machen

*Vorbedingung:* Zulassungsliste liegt vor

*Nachbedingung Erfolg:* Prüfung ist benotet

*Akteure:* Prüfungssekretariat, Prüfer



## Anhang 2 Lösungen LE 6

*Auslösendes Ereignis:* 14 Tage vor dem festgesetzten Prüfungstermin

*Beschreibung:*

- 1 Zulassungsliste an Prüfer übermitteln
- 2 Prüfer trägt die Ergebnisse in die Zulassungsliste ein
- 3 Prüfungssekretariat validiert die Ergebnisse
- 4 veröffentlichen der Prüfungsergebnisse

*Geschäftsprozeß:* eintragen Praktikum

*Ziel:* Praktikum ist nachgewiesen

*Vorbereitung:* Student ist immatrikuliert

*Nachbedingung Erfolg:* Praktikum anerkannt

*Nachbedingung Fehlschlag:*

*Akteure:* Prüfungssekretariat

*Auslösendes Ereignis:* Student legt Praktikumsbescheinigung vor

*Beschreibung:*

- 1 prüfen, ob Firma bereits im System existiert
- 2 eintragen des Praktikums

*Erweiterung:*

1a Firma erfassen

*Geschäftsprozeß:* eintragen Diplomarbeit

*Ziel:* abschließen des Studiums

*Vorbereitung:* Student ist immatrikuliert

*Nachbedingung Erfolg:* Diplomarbeit benotet

*Nachbedingung Fehlschlag:* Anmeldung abgelehnt

*Akteure:* Prüfungssekretariat, Prüfer(=Betreuer)

*Auslösendes Ereignis:* Student gibt Anmeldeformular zur Diplomarbeit ab

*Beschreibung:*

- 1 Student beantragt Zulassung zur Diplomarbeit
- 2 Sekretariat prüft, ob für alle Pflichtveranstaltungen die Prüfungen bestanden wurden
- 3 Sekretariat teilt dem Betreuer die Anmeldung mit
- 4 Betreuer trägt das Thema der Diplomarbeit ein
- 5 Betreuer teilt die Note und das Abschlußdatum mit

### Aufgabe 5

Paket: Auftragsabwicklung

- Bearbeiten von Kundenaufträgen lt. Katalog
- Bearbeiten von Kundenaufträgen lt. Katalog mit Nachlieferungen
- Bearbeiten von Sonderwünschen der Kunden
- Weitergabe aller Aufträge an die Buchhaltung

Paket: Kundenpflege

- Informieren von Kunden über neue Produkte
- Versenden von Probesendungen an gute Kunden

Paket: Bestellabwicklung

- Erstellen von Bestellungen an Lieferanten, um gängige Artikel am Lager zu haben
- Erstellen von Bestellungen an Lieferanten, um Kundenaufträge zu erfüllen
- Weitergabe aller Bestellungen an die Buchhaltung

Paket: Marketing

- Ermittlung von Informationen für das Marketing (Penner-Renner-Liste)
- Auswertung von Sonderwünschen für das Marketing
- Auswerten von Informationen der Lieferanten, um neue Kataloge zu erstellen

## LE 7

### Aufgabe 1

Die folgende Modellierung ermöglicht, daß ein Projekt in mehreren Praktikumsveranstaltungen bearbeitet wird, d.h. in mehreren Praktikumslisten vorhanden sein kann. Die Assoziation zwischen Student und Praktikumsveranstaltung sagt aus, an welchen Veranstaltungen der Student teilnehmen soll. Die tatsächliche Teilnahme wird in der assoziativen Klasse vermerkt.

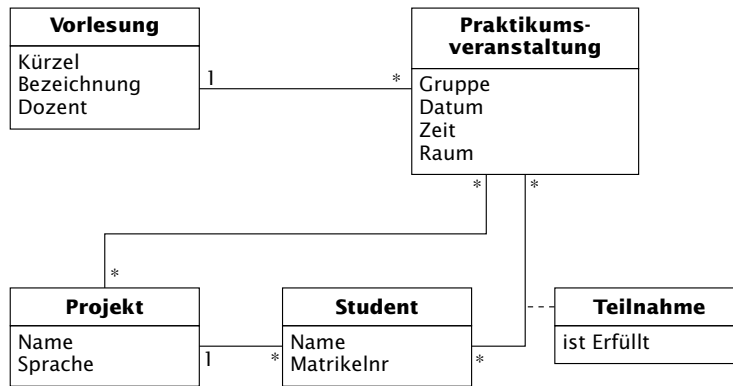


Abb. LE7-1:  
Klassendiagramm  
der Praktikumsliste

### Aufgabe 2

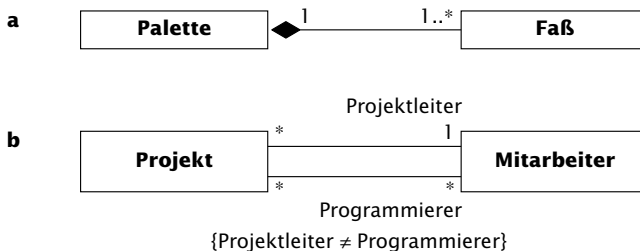
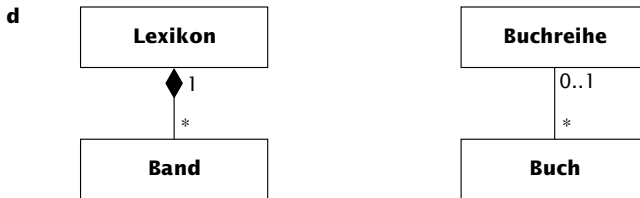
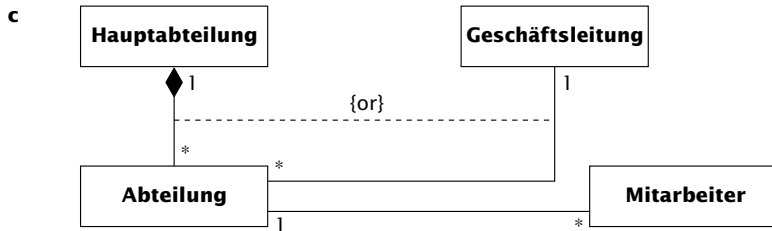


Abb. LE7-2a:  
Assoziationen

## Anhang 2 Lösungen LE 7

Abb. LE7-2b:  
Assoziationen



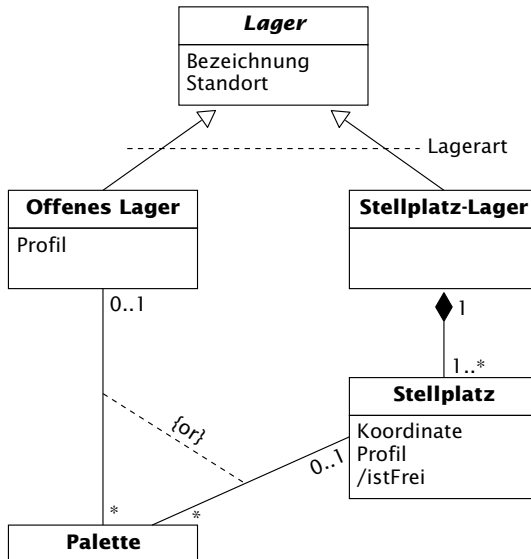
d

Das Verhalten des Lexikons wird stets auf dessen Bände übertragen, z.B. Neu-Auflage, Verkauf. Dagegen ist die Beziehung zwischen Buch und Buchreihe lose gekoppelt. Auf eine Aggregation, die Sie vielleicht aufgrund der Formulierung »ist Teil von« gewählt haben, habe ich hier verzichtet, weil sie keine Vorteile bringt.

### Aufgabe 3

Das Attribut `Stellplatz.istFrei` lässt sich aus der Anzahl der Objektverbindungen zu `Palette` ableiten.

Abb. LE7-3:  
Vererbungsstrukturen



**Aufgabe 4**

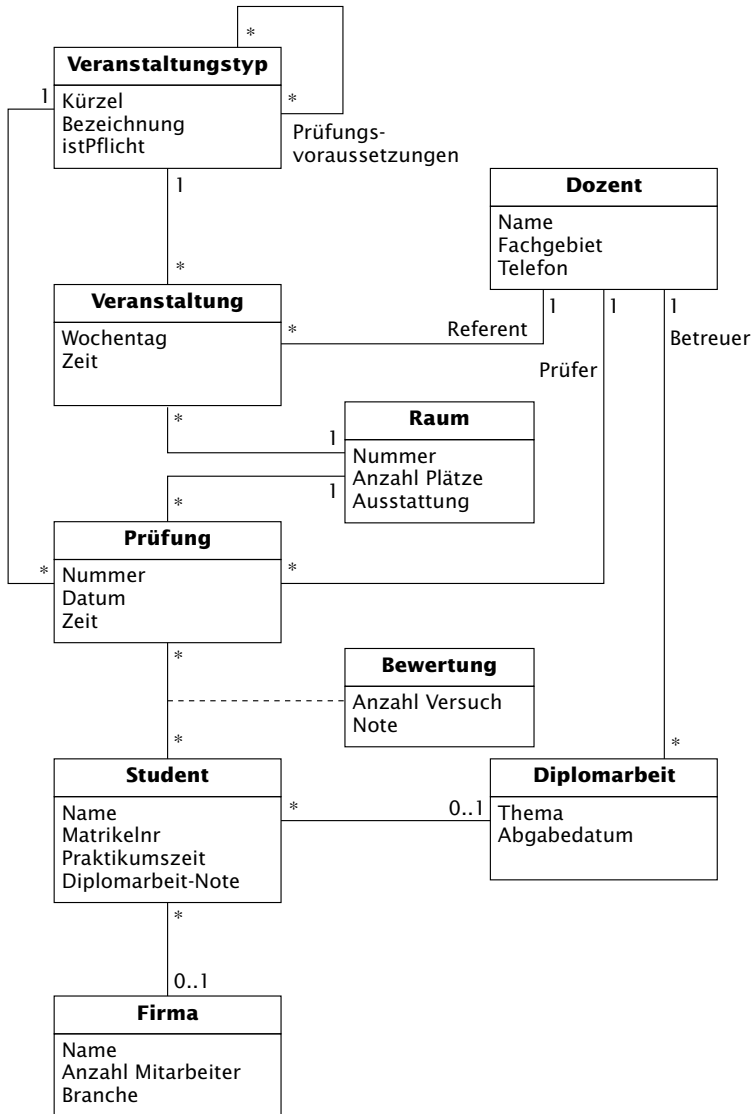


Abb. LE7-4:  
Klassendiagramm  
der Hochschul-  
verwaltung

## Anhang 2 Lösungen LE 8

### LE 8

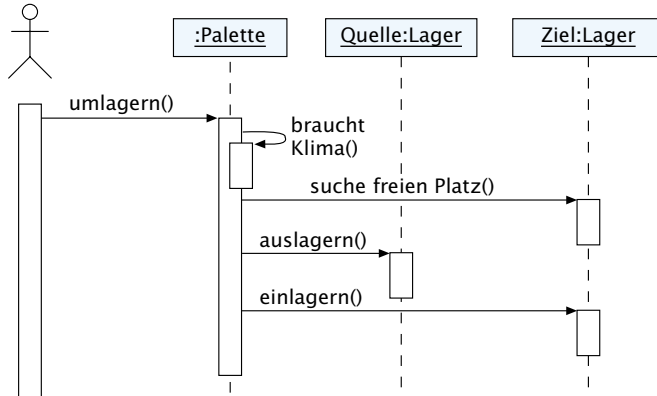
#### Aufgabe 1

Szenario: Palette umlagern (in gewünschtes Ziel-Lager)

Bedingungen: Palette ist im angegebenen Lager vorhanden

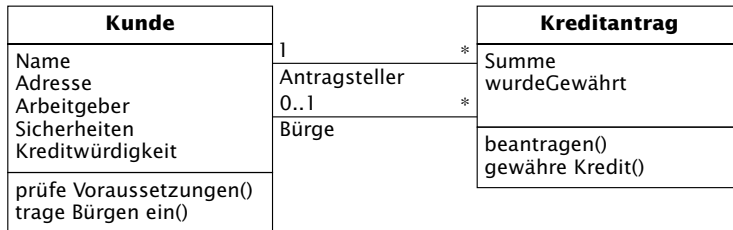
Ergebnis: Palette an neuem Ort gelagert

Abb. LE8-1:  
Szenario zum  
Umlagern einer  
Palette

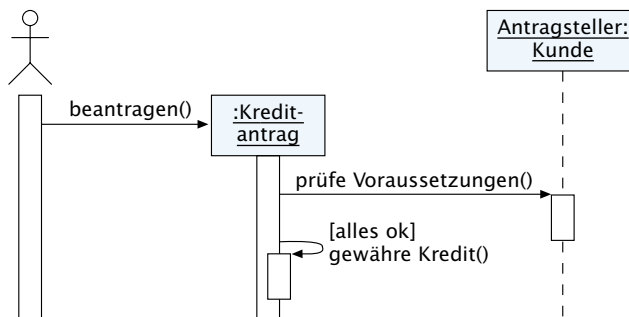


#### Aufgabe 2

Abb. LE8-2a:  
Szenarios für das  
Bearbeiten eines  
Kreditantrags

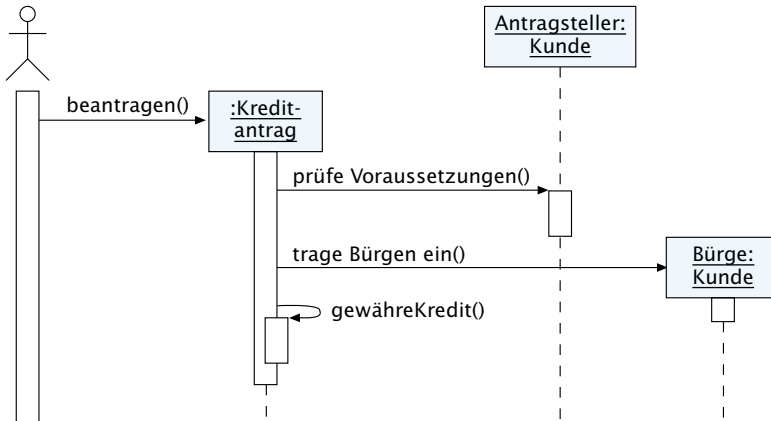


Szenario: exist. Kunden wird der Kredit gewährt



Szenario: exist. Kunden wird der Kredit nur mit einem Bürgen gewährt

Abb. LE8-2b:  
Szenarios für das  
Bearbeiten eines  
Kreditantrags



### Aufgabe 3

Für die Klasse Mietwagen ergibt sich ein nicht-trivialer Lebenszyklus.

Der Zustand verfügbar besitzt eine Transition vom Anfangszustand. Sie ist verbunden mit der Operation erfassen().

Der Zustand ausgemustert bildet den Endzustand. Befindet sich ein Mietwagen-Objekt in diesem Zustand, dann ist sein weiteres dynamisches Verhalten nicht mehr von Interesse. Die Daten dieses Mietwagens können aber für statistische Abfragen weiterhin verwendet werden.

Hier wurden alle Operationen als Aktionen dargestellt.

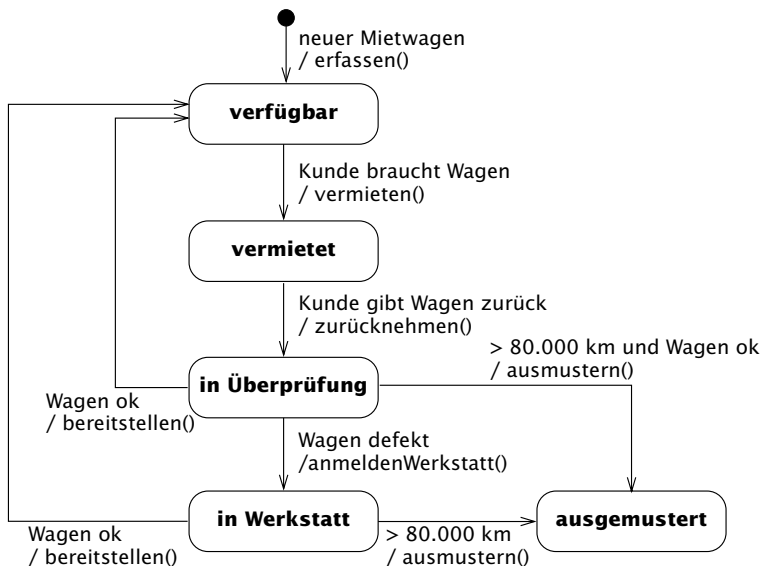


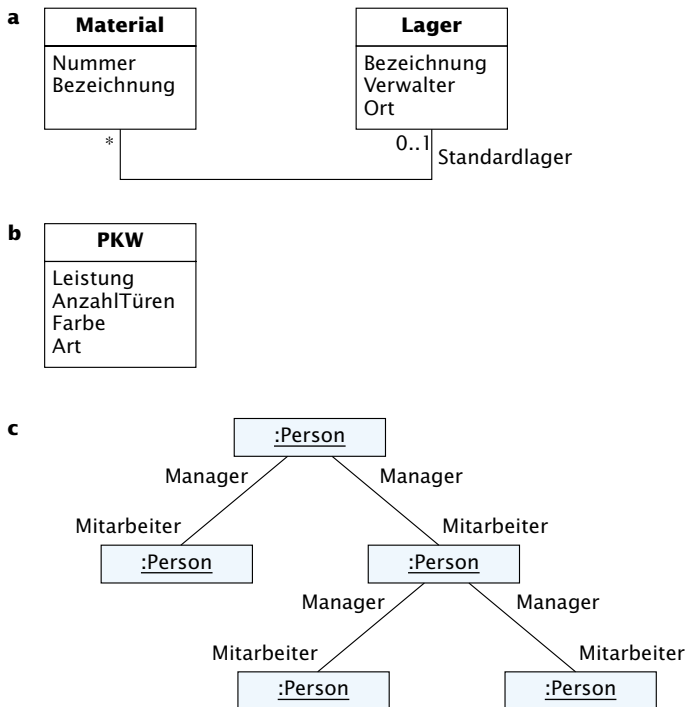
Abb. LE8-3:  
Zustandsdiagramm  
eines Mietwagens

## Anhang 2 Lösungen LE 8

### Aufgabe 4

- a** Referenzen auf andere Objekte werden durch Assoziationen dargestellt (Checkliste Attribute, Punkt 11).
- b** Das Attribut Art gibt an, ob es sich um eine Limousine oder um ein Cabrio handelt (Checkliste Vererbung, Punkt 4).
- c** Die beiden Modelle sind nicht gleichwertig. Im Modell mit der reflexiven Assoziation kann eine Person die Rollen eines Mitarbeiter und eines Managers spielen (vergleiche Abb. LE8-4, Teil c). Mitarbeiter und Manager besitzen dieselben Eigenschaften. Die Vererbungsstruktur der Aufgabenstellung modelliert Mitarbeiter- und Personenobjekte getrennt. Sie besitzen nur zu einem Teil gemeinsame Eigenschaften. In das Modell mit der reflexiven Assoziation wurde zusätzlich noch ein Fehler eingebaut, denn es muß zu jeder Person einen Manager geben, was jedoch nicht auf den »obersten« Manager zutrifft.

Abb. LE8-4: Zur Analyse von Klassendiagrammen



### Aufgabe 5

- a** Das Team der formalen Inspektion besteht außer dem Moderator und dem Autor aus ein bis vier Inspektoren. Bei einem kleinen Team führt der Moderator Protokoll. Bei einem großen Team sollte es einen Protokollführer geben.

- b** Man kann für einen Teil, der ein repräsentativer oder sogar ein besonders kritischer Ausschnitt des Prüfobjekts bildet, eine Ausschnittsprüfung durchführen. Damit kann auf das komplette Prüfobjekt geschlossen werden. Da es hier nicht darum geht, den Aufwand zu minimieren, sondern nur die benötigte Zeit, kann das Prüfobjekt auch in mehrere Prüfobjekte zerlegt und auf mehrere Inspektionsteams aufgeteilt werden.  
Hinweis: Es ist *keine* gute Idee, wenn Sie vorschlagen, das Inspektionsteam zu vergrößern, da große Teams nicht so effektiv arbeiten können wie kleine.
- c** Bei der Inspektion werden zu einem frühen Zeitpunkt Defekte gefunden, die später hohe Folgekosten verursachen. Damit können mit der Inspektion sogar Kosten eingespart werden, wenn die gesamten Entwicklungskosten betrachtet werden.
- d** Werkzeuge können nur formale Prüfungen durchführen. Dazu zählen auch viele Konsistenzprüfungen zwischen statischem und dynamischem Modell. Semantische Qualitätskriterien können nur manuell geprüft werden.

## LE 9

### Aufgabe 1

Eine Aktion kann folgendermaßen ausgelöst werden:

- Menübalken und *drop-down*-Menüs: Mausbewegung zum Menübalken, ein Mausklick auf dem Menütitel, Mausbewegung zur Menüoption, ein Mausklick auf der Menüoption, kein Schreibaufwand.  
Bei Verwendung von Kaskadenmenüs kann sich der Aufwand erhöhen.
- *pop-up*-Menü: evtl. Mausbewegung zum Objekt, ein Mausklick zum Öffnen des Menüs (auf dem Objekt), Mausbewegung zur Menüoption, ein Mausklick zur Selektion, kein Schreibaufwand.
- mnemonisches Kürzel: Menütitel durch ALT+Zeichen auswählen, Menüoption durch Eingabe des Zeichens auswählen, keine Mausbewegung.
- Tastaturkürzel: eine Funktionstaste (STRG) und ein Zeichen oder nur eine Funktionstaste (ENTF, F7) eingeben, keine Mausbewegung.
- Symbolbalken: Mausbewegung zum Symbolbalken, ein Mausklick auf dem Mini-Piktogramm.
- Menüoptionen im Arbeitsbereich: Mausbewegung zur Menüoption, ein Mausklick auf der Schaltfläche.



### Aufgabe 2

Die Analyse von *Winword 97* ergibt:

- a** Primärdialoge:
  - Erstellen eines Textes,
  - Formatieren eines Textes,
  - Bearbeiten einer Tabelle.
- b** Sekundärdialoge:
  - Datei/Speichern unter
  - Ansicht/Kopf- und Fußzeile beschreiben,
  - Einfügen/Sonderzeichen.
- c** Modale Dialoge:
  - Datei/Speichern unter,
  - Datei/Drucken.
- d** Nicht-modale Dialoge:
  - Einfügen/Sonderzeichen,
  - Bearbeiten/Suchen.
- e** Objektorientierte Bedienung: Rechtschreibvorschläge für angeklicktes Wort anzeigen lassen.
- f** Funktionsorientierte Bedienung: Öffnen eines Dokuments in *Winword*.
- g** direkte Manipulation: Verschieben eines markierten Textes an eine andere Position.

### Aufgabe 3

Die Analyse von *Winword 97* ergibt:

- a** Anwendungsfenster: *Winword*-Fenster, wird angezeigt, auch wenn kein Dokument geöffnet ist.  
Unterfenster: jedes Textdokument wird in einem Unterfenster angezeigt.  
Dialogfenster: Datei/Öffnen.  
Mitteilungsfenster: Meldung, daß die Rechtschreibprüfung des markierten Teils abgeschlossen ist.
- b** *Winword* ist eine MDI-Anwendung.

### Aufgabe 4

Die Analyse von *Winword 97* ergibt:

- a** Aktionsmenü: *drop down*-Menü »Datei«.
- b** Eigenschaftsmenü: *drop down*-Menü »Ansicht« mit Layout-Einstellung.
- c** *pop-up*-Menü: *pop-up*-Menü zur Textbearbeitung (Ausschneiden etc.) für einen markierten Text aktivierbar.
- d** Beschleunigungsmöglichkeiten: mnemonische Kürzel (z.B. ALT+»D« für »Datei« und »S« für »Speichern«, Tastaturkürzel (z.B. STRG + »P« für Drucken), Symbolbalken, zuletzt benutzte Objekte (z.B. im Datei-Menü die Anzeige der zuletzt geöffneten Textdokumente).

## LE 10

### Aufgabe 1

- a** Da für einen Studenten mehrere Möglichkeiten zutreffen können, ist die Mehrfachauswahl notwendig. Wegen der geringen Anzahl von Auswahlmöglichkeiten und deren voraussichtlicher Stabilität sind Kontrollkästchen (*check boxes*) sinnvoll.
- b** Eine Adresse kann sich nur auf ein Land beziehen. Daher kommt nur eine Einfachauswahl in Frage. Wegen der vielen Alternativen scheiden Optionsfelder (*option buttons*) aus. Es kommt nur das Listenfeld (*list box*) oder das *Dropdown*-Listenfeld (*drop-down list box*) in Frage. Der Benutzer soll nicht neue Ländernamen erfinden können. Daher ist ein Kombinationsfeld hier nicht sinnvoll.
- c** Die Anzahl der Alternativen ist fest vorgegeben und relativ gering. Außerdem ist genau eine Anrede zu wählen. Daher sind Optionsfelder (*option buttons*) zu verwenden.
- d** Die Aufgabenstellung ist hier nicht präzise genug und erlaubt zwei Lösungen. Kann die Größe kontinuierlich geändert werden (z.B. 100%, 99 %, 98%, 97% etc.), wobei immer nur ein Dekrementieren und Inkrementieren möglich ist, dann ist das Drehfeld (*spin box*) am besten geeignet. Kann die Prozentangabe aus einer Liste von Alternativen ausgewählt werden (200%, 150%, 100%, 50%), dann ist das Listenfeld (*list box*) sinnvoll. Sollen zusätzlich beliebige Prozentangaben möglich sein, dann ist sie als Kombinationsfeld (*combo box*) zu gestalten. Ist wenig Platz vorhanden, dann sollten *Dropdown*-Listenfeld bzw. *Dropdown*-Kombinationsfeld verwendet werden.

### Aufgabe 2

Das Erfassungsfenster wird wie folgt entworfen (Abb. LE10-2). Ein zweisepaltiger Dialog ergibt ein breites Fenster. Da beide Fensterhälften gleich viele Informationen erhalten, ist das Fenster balanciert. Die Anzahl der virtuellen Linien wurde minimiert, indem alle

Abb. LE10-2:  
Erfassungsfenster  
für ein Projekt

**Anhang 2 Lösungen LE 10**

Eingabefelder – mit Ausnahme des Datumsfeldes – gleich breit gewählt wurden. Weil ein Auftraggeber mehrere Projekte vergeben kann, wurde dafür ein erweiterbares *Drop-down-Kombinationsfeld (drop-down combo box)* gewählt, um unterschiedliche Schreibweisen des gleichen Auftraggebers zu vermeiden. Die Assoziationen zu Projektleiter und Mitarbeiter wurden gemäß den Transformationsregeln realisiert.

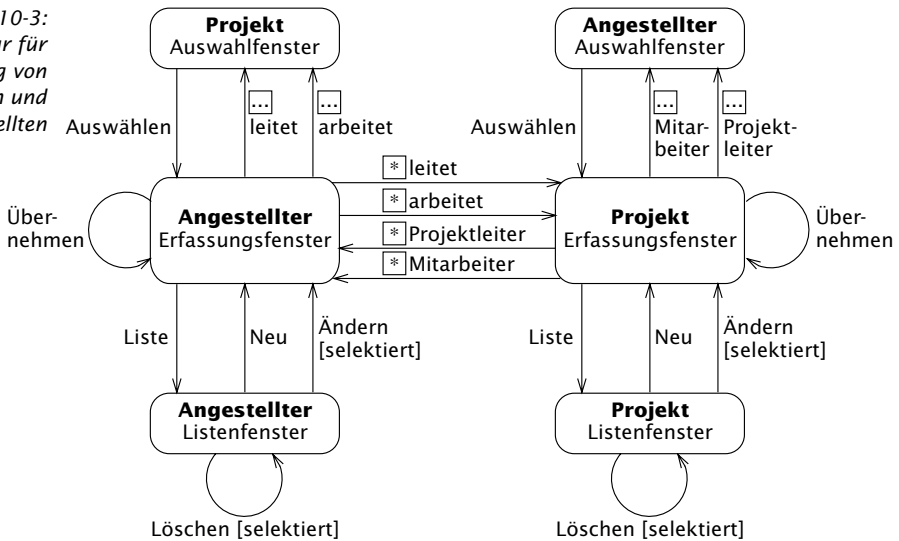
**Aufgabe 3**

Für jede Klasse werden jeweils ein Erfassungsfenster, ein Listenfenster und – wegen der Assoziationen – ein Auswahlfenster benötigt (Abb. LE10-3).

Jeder Angestellter kann mehrere Projekte leiten und in mehreren Projekten mitarbeiten. Daher muß das Erfassungsfenster für den Angestellten die beiden *Link-Listen leitet* und *arbeitet* enthalten. Mit der Neu-Schaltfläche der entsprechenden Liste kann das Projekt-Erfassungsfenster geöffnet werden. Mit der *Link*-Schaltfläche kann zum Projekt-Auswahlfenster verzweigt werden. Daher gibt es vom Zustand *Angestellter Erfassungsfenster* die beiden Transitionen *Neu-leitet* und *Neu-arbeitet* zum Zustand *Projekt Erfassungsfenster* und die zwei Zustandsübergänge *Link-leitet* und *Link-arbeitet* zum Zustand *Projekt Auswahlfenster*.

Umgekehrt muß zu jedem Projekt genau ein Projektleiter eingetragen werden, während es mehrere Projekt-Mitarbeiter gibt. Daher sind vom Zustand *Projekt Erfassungsfenster* analoge Transitionen einzutragen.

Abb. LE10-3:  
Dialogstruktur für  
die Verwaltung von  
Projekten und  
Angestellten



## LE 11

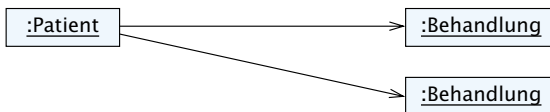
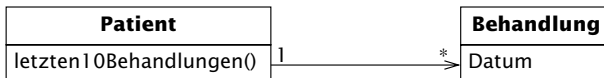
### Aufgabe 1

- a** Eine generische Klasse muß nur einmal entworfen, implementiert und getestet werden und kann durch Parametrisierung auf ähnliche Problemstellung mehrmals angewendet werden, ohne daß Änderungen notwendig sind. Dadurch sind potentielle Fehlerquellen ausgeschaltet.
- b** Eine abstrakte Klasse kann Attribute besitzen, eine Schnittstelle nicht. Eine abstrakte Klasse kann sowohl abstrakte als auch nicht abstrakte Operationen enthalten, eine Schnittstelle nur abstrakte Operationen. Von einer abstrakten Klasse können Assoziationen ausgehen, von einem *interface* nicht.
- c** Im nachfolgenden Programm sind die Attribute als *public* vereinbart. Dann ist zwar die Verkapselung erfüllt, aber nicht das Geheimnisprinzip, da der Zugriff auf die Attribute nicht mehr über die Operationen erfolgen muß.

```
class Artikel
{public:
  int Nummer;
  String Bezeichnung;
  float Preis;
  void erhoehePreisUm (float Betrag);
};
```

- d** Eine abstrakte Operation sorgt dafür, daß alle Unterklassen einer gemeinsamen Oberklasse Operationen mit den gleichen Signaturen verwenden.
- e** Wenn eine Assoziation als Klasse realisiert wird, dann ist das Wissen, welche Objekte sich kennen, nur in dieser speziellen Klasse enthalten und nicht in den Klassen, die über die Assoziation in Beziehung stehen.

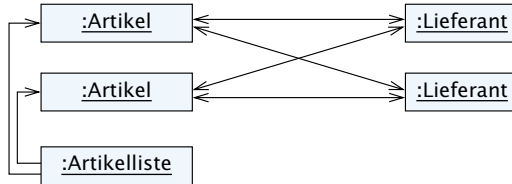
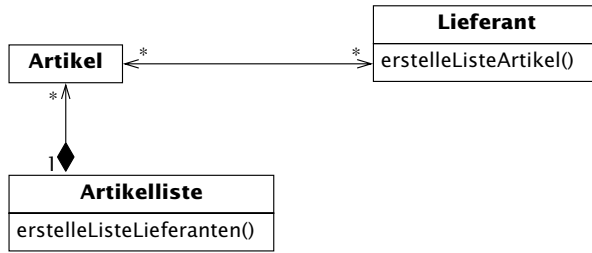
### Aufgabe 2



LE11-2a: OOD-Klassendiagramm mit Objektdiagramm

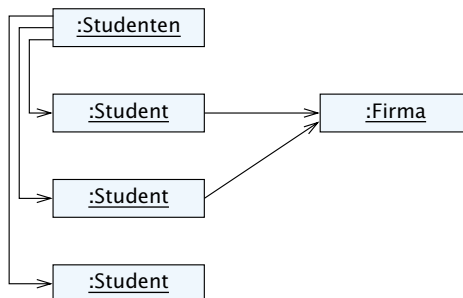
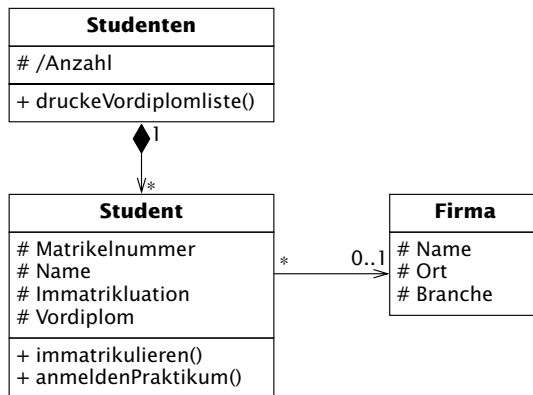
## Anhang 2 Lösungen LE 11

LE11-2b: OOD-  
Klassendiagramm  
mit Objekt-  
diagramm



### Aufgabe 3

LE11-3: OOD-  
Klassendiagramm  
und Objekt-  
diagramm



**Aufgabe 4**

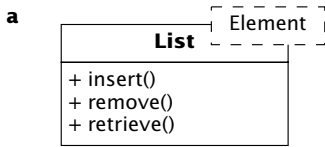
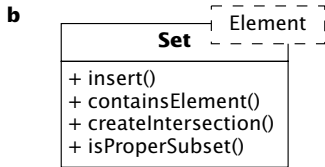


Abb. LE11-4:  
Generische Klassen



Die Signaturen lauten:

Klasse: List

insert (in element: Element)

remove (in position: Uint)

retrieve (in position: Uint): Element

Klasse: Set

insert (in element: Element)

containsElement (in element: Element) : Boolean

createIntersection (in otherSet Set): Set

isProperSubset (in otherSet Set): Boolean

**LE 12**

**Aufgabe 1**

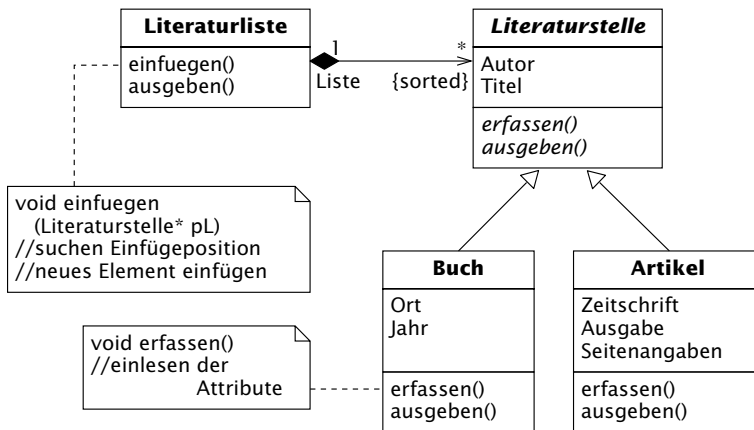


Abb. LE12-1:  
Klassendiagramm  
Literatur-  
verwaltung

## Anhang 2 Lösungen LE 12

Erfassen eines Buchs (analog dazu wird ein Artikel erfaßt)

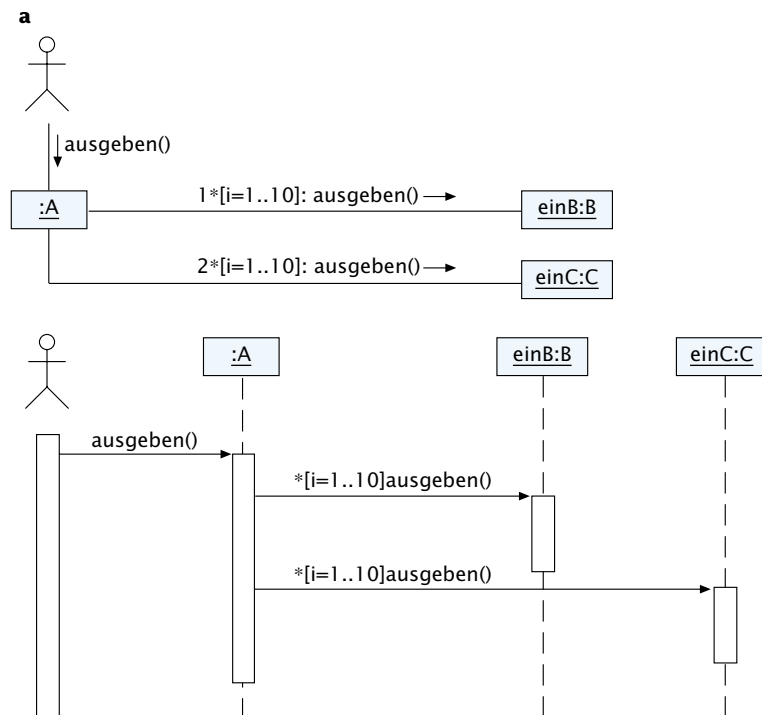
```
Li teraturstel le* pL;  
pL = new Buch;  
pL->erfassen();           //dynamische Bindung an Buch::erfassen()  
Li ste->ei nfuegen(pL)    //Buch in die Li teraturli ste ei nfuegen
```

Sortierte Ausgabe von Literaturstellen

```
voi d Li teraturli ste : : ausgeben()  
Li teraturstel le* pL  
für alle pL in der Li ste  
    pL->ausgeben();       //Buch::ausgeben() oder  
                          //Arti kel : : ausgeben()
```

### Aufgabe 2

Abb. LE12-2a:  
Kollaborations-  
und Sequenz-  
diagramm



b

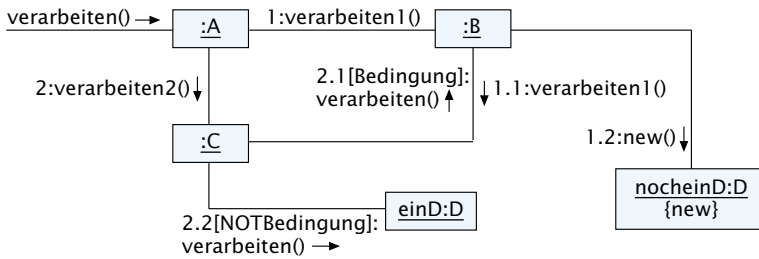
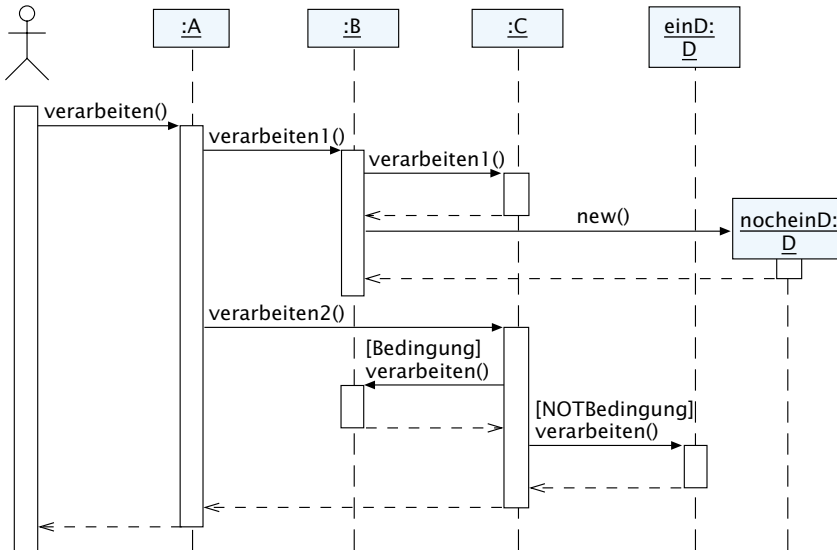


Abb. LE12-2b: Kollaborations- und Sequenzdiagramm



### Aufgabe 3

Alle blau eingetragenen Operationen besitzen eine Implementierung, da das Objekt im entsprechenden Zustand auf die jeweilige Botschaft reagieren muß. Für alle normal eingetragenen Operationen müssen nur die Ausnahmebehandlungen angegeben werden. Beispielsweise kann das Buchobjekt im Zustand präsent nicht auf die Botschaften zurueckgeben() und vorbestellen() reagieren.

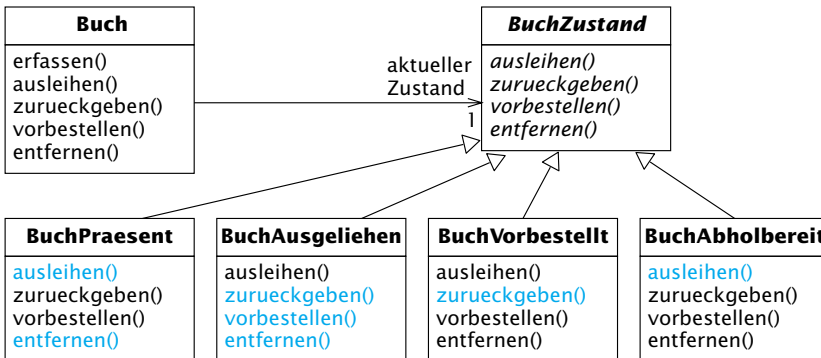


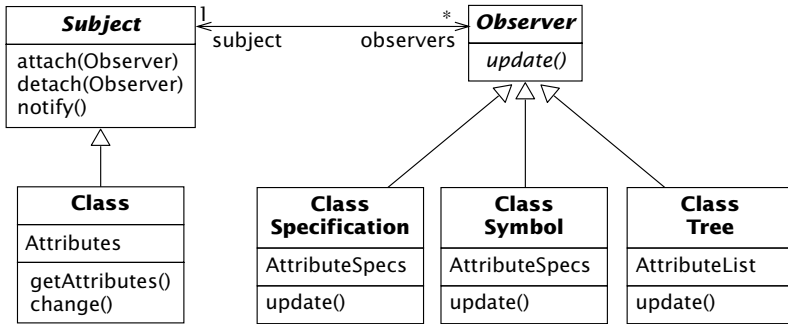
Abb. LE12-3: Zustandsmuster für den Lebenszyklus der Klasse Buch



# LE 13

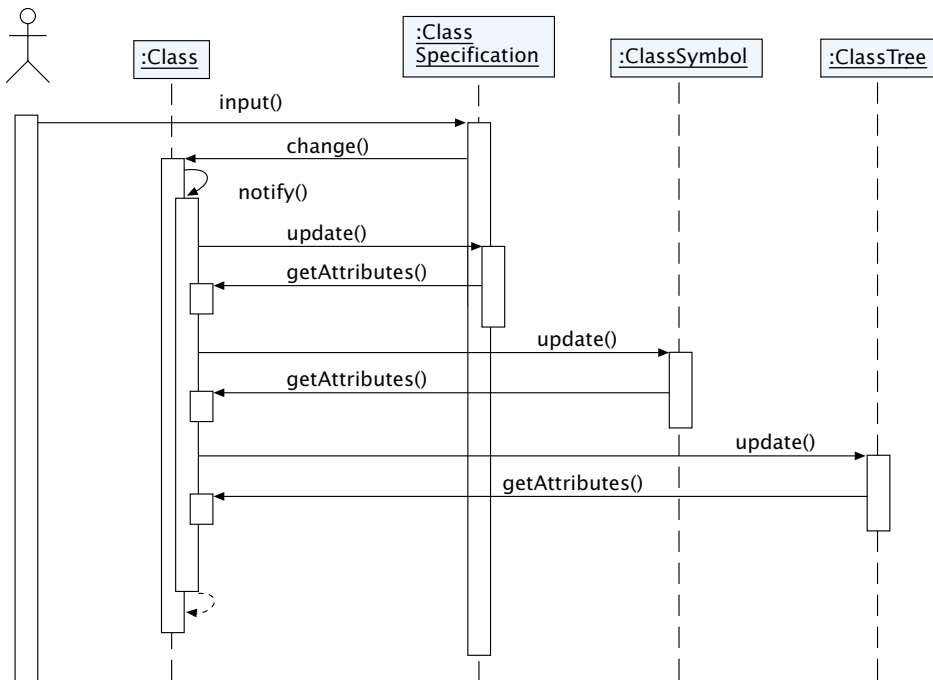
## Aufgabe 1

Abb. LE13-1a:  
OOD-Klassen-  
diagramm



Die Problemstellung lässt sich mit Hilfe des Beobachter-Musters elegant realisieren. Die Klasse `Class` bildet eine Spezialisierung von `Subject`. Jede Spezialisierung von `Subject` erbt eine Liste von Referenzen auf diverse `Observer`-Klassen. Für die Klasse `Class` gibt es drei verschiedene Darstellungsmöglichkeiten, die als Unterklassen von `Observer` realisiert werden (Abb. LE13-1a). Jede Spezialisierung von `Observer` kennt genau ein konkretes `Subject`. Abb. LE13-1b zeigt die Kommunikation zwischen den Darstellungsklassen und der Klasse `Class`.

Abb. LE13-1b:  
OOD-Sequenz-  
diagramm



### Aufgabe 2

**a** Schablonenmethode-Muster (*template method*)

Die Operation `verschieben()` ist eine Schablonenmethode. Sie ruft die abstrakten Operationen `zeigen()` und `loeschen()` auf, die in der Unterklasse `Kreis` implementiert werden.

**b** Kompositum-Muster (*composite*)

Ein Verzeichnis-Objekt kann Objekte der Klassen `Verknuepfung`, `Datei` und `Verzeichnis` enthalten. Zusammengesetzte und elementare Objekte werden weitgehend gleich behandelt.

**c** Fabrikmethode (*factory method*)

In einem Diagramm sollen verschiedene Elemente dargestellt werden, wobei jede Diagrammart andere Elemente enthalten kann. In der Klasse `MyDiagram` wird durch die Fabrikmethode `createElement()` konkret festgelegt, welche Objekte sie erzeugen soll.

### Aufgabe 3

**a** Gemeinsamkeiten

Muster, *Frameworks* und Klassenbibliotheken unterstützen alle die Wiederverwendung von Software und die Standardisierung von Entwurf und Implementierung.

**b** Unterschiede

Muster stellen die abstrakteste Form dar. Sie zeigen nur beispielhaft auf, wie ein bestimmtes Problem realisiert werden kann. Der Entwerfer muß die Lösung – unter Benutzung der Lösungsideen – vollständig selbst erstellen.

*Frameworks* bieten für bestimmte Anwendungsbereiche Klassen an, die der Entwerfer bzw. Programmierer durch Unterklassen spezialisiert.

Klassenbibliotheken sind Softwaresammlungen, deren Komponenten direkt verwendet werden können. Ein *Framework* kann als Sonderfall einer Klassenbibliothek aufgefaßt werden.

## LE 14

### Aufgabe 1

**a** Jedes Tupel einer Tabelle muß einen expliziten Schlüssel besitzen. Es besteht aus einem oder mehreren Attributen, ist äußerlich nicht von einem fachlichen Attribut zu unterscheiden und identifiziert eindeutig jedes Tupel in einer Tabelle. Auch ein fachliches Attribut kann theoretisch als Schlüsselattribut verwendet werden, obwohl dies in der Praxis vermieden werden sollte. Jedes Objekt besitzt dagegen implizit eine Objektidentität. Sie ist nicht nur innerhalb einer Klasse, sondern innerhalb des gesamten Systems eindeutig. Die Objektidentität besitzt keine semantische Bedeutung.

## Anhang 2 Lösungen LE 14

- b** Eine Assoziation ist in der Analyse inhärent bidirektional, im Entwurf wahlweise bi- oder unidirektional. Sie wird nicht durch Attribute der beteiligten Objekte ausgedrückt. Das Wissen, welche anderen Objekte ein bestimmtes Objekt kennt, ist nur in den Assoziationen enthalten. Ein Fremdschlüssel ist ein Referenz-Attribut, das dem Primärschlüssel-Attribut eines Tupels in einer anderen Tabelle entspricht. Die Schlüssel-Fremdschlüssel-Beziehung ist unidirektional.
- c** Bei einem objektorientierten Modell werden die Attribute problemadäquat beschrieben. Ein Attribut kann von jedem beliebigen Typ – auch eine Liste variabler Länge – sein. Die erste Normalform sagt dagegen aus, daß weder Strukturen noch Listen vorkommen dürfen und fordert eine konstante Länge aller Tupel, die Voraussetzung für das Speichern in einer Tabelle ist.
- d** Das OID-Attribut ist ein künstliches Attribut, um das jede Tabelle bei der objekt-relationalen Abbildung erweitert wird. Es besitzt alle Eigenschaften der Objektidentität, ist aber äußerlich von einem fachlichen Attribut nicht zu unterscheiden. Häufig handelt es sich um eine sehr große ganze Zahl, die keinerlei semantische Bedeutung besitzt. Änderungen im Fachkonzept, z.B. Erweiterung von einer 4- zu einer 6-stelligen Artikelnummer, lassen sich dann problemloser durchführen.

### Aufgabe 2

- a** Als logisches Schema wird die Menge aller Tabellen bezeichnet, welche die relationale Datenbank bilden. Es wird im *Data Dictionary* eingetragen.
- b** Ein externes Schema ist eine bestimmte Sicht (*view*) auf die Datenbank. In dieser Sicht werden jedoch keine Daten gespeichert. Mit externen Schemata kann erreicht werden, daß bestimmte Benutzergruppen nur festgelegte Daten sehen dürfen.
- c** Die DDL ist die Datendefinitionssprache zur formalen Definition des logischen Schemas.
- d** Die DML ist die Datenmanipulationssprache. Sie stellt eine Reihe von Befehlen bereit, um die Datenbank mit Daten zu füllen und damit zu arbeiten.
- e** SQL ist der Standard für DDL und DML.

### Aufgabe 3

Abb. LE14-3 zeigt die Tabellenstruktur. Der Stundenlohn muß auf eine separate Tabelle StudentK abgebildet werden, die nur ein einziges Tupel enthält. Jedes Student-Tupel enthält die OID dieses Tupels als Fremdschlüssel. Dadurch kann der Stundenlohn problemlos geändert werden. Da zu einer studentischen Hilfskraft mehrere Arbeitsverträge gespeichert werden sollen, werden deren Daten in einer separaten Tabelle gespeichert. Die Angaben zu Name

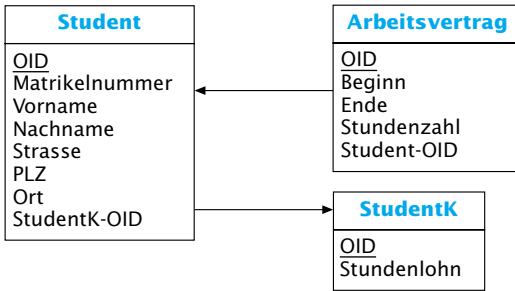


Abb. LE14-3:  
Abbildung der Klasse Studentische Hilfskraft auf Tabellen

und Adresse wurden in die Haupttabelle integriert, weil diese Angaben bei jedem Zugriff auf ein Objekt von Student benötigt werden und dadurch zwei *joins* eingespart werden.

#### Aufgabe 4

Die Vererbungsstruktur wird auf eine einzige Tabelle Artikel abgebildet, weil die Klasse Lagerartikel nur zwei Attribute hat und daher die Vorteile einer Verschmelzung beider Klassen überwiegen.

Bei der Abbildung der Klasse Lieferant wird die Datenstruktur des Attributs Adresse, das durch eine elementare Klasse beschrieben wird, in die Tabelle Lieferant integriert. Das ist hier sinnvoll,

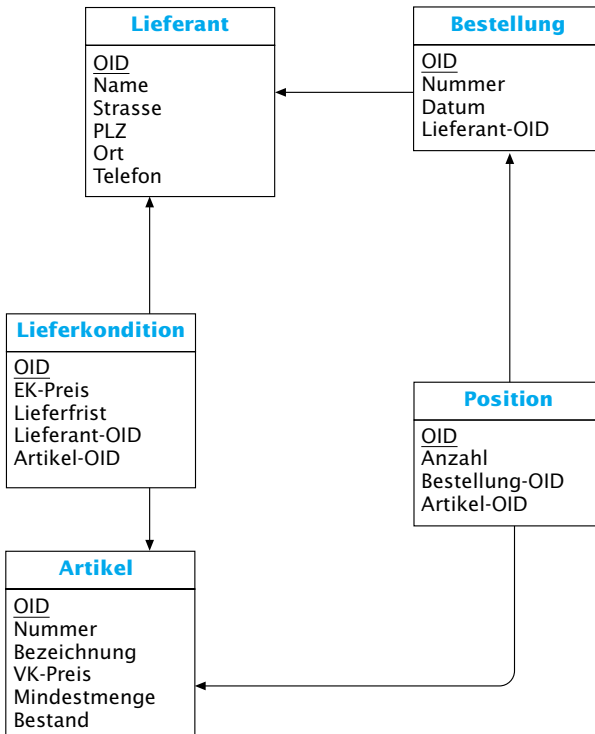


Abb. LE14-4:  
Abbildung des Klassendiagramms Bestellwesen auf Tabellen

## Anhang 2 Lösungen LE 14

weil diese Attribute für jeden Lieferanten erfaßt werden und dadurch ein *join* eingespart wird.

Die m:m-Assoziation zwischen Lieferant und Artikel wird auf eine Tabelle abgebildet, in die auch die Attribute der assoziativen Klasse Lieferkondition integriert werden.

### Aufgabe 5

a Das logische Schema wird in SQL durch folgende Tabelle definiert:

```
create table Artikel
( OID          number(10)    not null ,
  Nummer       number(7)     not null ,
  Bezeichnung  char(50)      not null ,
  VKPreis      number(8,2),
  Mindestmenge integer,
  Bestand      integer
);
create table Lieferant
( OID          number(10)    not null ,
  Name         char(30)      not null ,
  Strasse      char(30),
  PLZ          char(5),
  Ort          char(50),
  Telefon     char(20)
);
create table Bestellung
( OID          number(10)    not null ,
  Nummer       number(7)     not null ,
  Datum        date,
  Lieferant_OID number(10)   not null
);
create table Lieferkondition
( OID          number(10)    not null ,
  EKPreis      number(8,2),
  Lieferfrist  integer,
  Artikel_OID  number(10)    not null ,
  Lieferant_OID number(10)   not null
);
create table Position
( OID          number(10)    not null ,
  Anzahl       integer,
  Bestellung_OID number(10)   not null ,
  Artikel_OID  number(10)    not null
);

create unique index Artikelnummer on Artikel (Nummer);
create unique index Bestellnummer on Bestellung (Nummer);
```

b Es ergibt sich folgender *view* in SQL:

```
create view ArtikelListe
as
select Artikel.Nummer, Artikel.Bezeichnung, Lieferant.Name,
       Lieferkondition.EKPreis
```

```

from Artikel, Lieferant, Lieferkondition
where Lieferant.OID = Lieferkondition.Lieferant_OID and
      Lieferkondition.Artikel_OID = Artikel.OID and
      Lieferkondition.Lieferfrist <= 1;

```

c Es ergibt sich folgender *select*-Befehl:

```

select Nummer, Bezeichnung, Bestand, Mindestmenge
from Artikel
where Bestand is not null and
      Mindestmenge is not null and
      Bestand < Mindestmenge;

```

d Es ergibt sich folgender *select*-Befehl:

```

select Lieferant.Name, Artikel.Bezeichnung,
      Lieferkondition.EKPreis, Lieferkondition.Lieferfrist
from Artikel, Lieferkondition, Lieferant
where Artikel.OID = Lieferkondition.Artikel_OID and
      Lieferkondition.Lieferant_OID = Lieferant.OID;

```

## LE 15

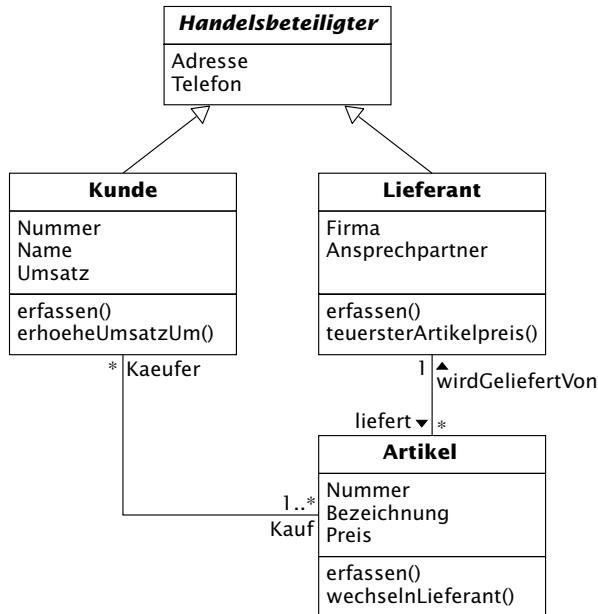
### Aufgabe 1

- a Ein **Literal** besitzt keine Objektidentität und kann nur als Komponente eines Objekts in der Datenbank gespeichert werden kann. Es wird durch einen Literaltyp beschrieben, der ein atomarer Standardtyp, eine Kollektion oder eine Struktur sein kann. Ein **Objekt** besitzt immer eine Objektidentität und kann daher für sich in der Datenbank gespeichert und wieder selektiert werden. Es wird durch den Objekttyp (atomar durch den Programmierer definiert, Kollektion oder Struktur) beschrieben.
- b Die **Klasse** definiert das Verhalten und den Zustand von Objekten. Für eine Klasse können zusätzlich die Klassenextension und der Schlüssel definiert werden. Eine **Schnittstelle** definiert nur das Verhalten. Nur von einer Klasse können Objekte erzeugt werden, von einer Schnittstelle nicht.
- c **Extends** definiert eine Einfachvererbung zwischen zwei Klassen. Bei der **subtyping**-Vererbung können Klassen und Schnittstellen von einer Schnittstelle abgeleitet werden. *Subtyping* ermöglicht auch die Mehrfachvererbung.

## Anhang 2 Lösungen LE 15

### Aufgabe 2

Abb. LE15-2:  
Klassendiagramm  
mit zusätzlicher  
Assoziation



- a Die ODL-Spezifikation muß um die blau eingetragenen Texte erweitert werden:

```

class Artikel
( extent ArtikelListe
  key Nummer)
{ attribute long Nummer;
  attribute string Bezeichnung;
  attribute float Preis;
  relationship Lieferant wirdGeliefertVon
    inverse Lieferant: liefert;
  relationship set <Kunde> Kaeufer
    inverse Kunde: Kauf;
  void erfassen()
    raises (schonVorhanden);
  void wechselnLieferant(in Lieferant NeuerLieferant)
    raises (gleicherLieferant);
}
class Handelsbeteiligter
{ struct AdresseT
  { string Strasse,
    string PLZ,
    string Ort};
  attribute AdresseT Adresse;
  attribute string Telefon;
}
class Lieferant extends Handelsbeteiligter
(extent Lieferanten)

```

```

{ attribute    string Firma;
  attribute    string Ansprechpartner;
  relationship set <Artikel> liefert
    inverse Artikel::wirdGeliefertVon;
void erfassen()
  raises (schonVorhanden);
float teuersterArtikelpreis()
  raises (liefertNichts);
}
class Kunde: Handelsteiliger
(extent      Kunden)
{ attribute   long Nummer;
  attribute   string Name;
  attribute   float Umsatz;
  relationship set <Artikel> Kauf
    inverse Artikel::Kaeufer;
void erfassen()
  raises (schonVorhanden);
void erhoeheUmsatzUm (in float Erhoehung);
}

```

**b** Name und Umsatz aller Kunden, deren Umsatz größer als 10.000

ist:

```

select distinct struct (Name: k.Name, Umsatz: k.Umsatz)
from Kunden k
where k.Umsatz > 10000

```

**c** Alle Kunden, die einen Artikel mit der Nummer 4711 gekauft haben:

```

select k
from Kunden k, k.Kauf a
where a.Nummer = 4711

```

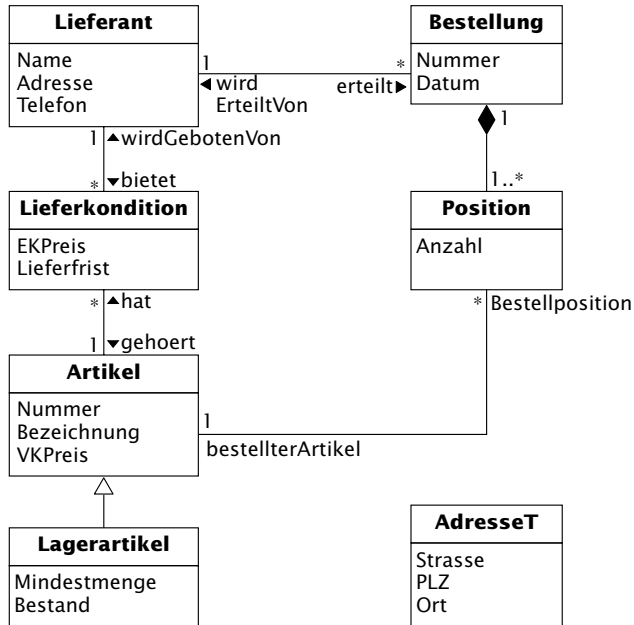
### Aufgabe 3

Die assoziative Klasse Lieferkondition kann nicht direkt in ODL spezifiziert werden, sondern muß in eine eigenständige Klasse aufgelöst werden (Abb. LE15-3). Dieses modifizierte Klassendiagramm wird dann in ODL spezifiziert.



## Anhang 2 Lösungen LE 15

Abb. LE15-3:  
Klassendiagramm  
ohne assoziative  
Klasse



```

class Artikel
(extent ArtikelListe
key Nummer)
{ attribute long Nummer;
  attribute string Bezeichnung;
  attribute float VKPreis;
  relationship set <Lieferkondition> hat
    inverse Lieferkondition::gehört;
  relationship set <Position> Bestellposition
    inverse Position::bestellterArtikel;
}
class Lagerartikel extends Artikel
(extent LagerartikelListe
key Nummer)
{ attribute long Mindestmenge;
  attribute long Bestand;
}
class Lieferant
(extent Lieferanten)
{ struct AdresseT
  { string Strasse,
    string PLZ,
    string Ort
  };
  attribute string Name;
  attribute AdresseT Adresse;
  attribute string Telefon;
  relationship set <Lieferkondition> bietet
    inverse Lieferkondition::wirdGebotenVon;
}
  
```

```

relationship set <Bestellung> erteilt
    inverse Bestellung: : wirdErteiltVon;
}
class Lieferkondition
(extent Lieferkonditionen)
{ attribute float EKPreis;
  attribute long Lieferfrist;
  relationship Lieferant wirdGebotenVon
    inverse Lieferant: : bietet;
  relationship Artikel gehoert
    inverse Artikel: : hat;
}
class Bestellung
(extent Bestellungen
key      Nummer)
{ attribute long Nummer;
  attribute date Datum;
  relationship Lieferant wirdErteiltVon
    inverse Lieferant: : erteilt;
  relationship set <Position> enthaelt
    inverse Position: : istTeilVon;
}
class Position
(extent Positionen)
{ attribute long Anzahl;
  relationship Bestellung istTeilVon
    inverse Bestellung: : enthaelt;
  relationship Artikel bestelltArtikel
    inverse Artikel: : Bestellposition;
}

```

#### Aufgabe 4

- a** Erstellen der Liste aller Lagerartikel, bei denen der Mindestbestand unterschritten ist. Die Liste soll enthalten: Nummer, Bezeichnung, Bestand, Mindestmenge.

```

select l.a.Nummer, l.a.Bezeichnung, l.a.Bestand, l.a.Mindestmenge
from Lagerartikel l
where l.a.Bestand < l.a.Mindestmenge

```

- b** Für jeden Dortmunder Lieferanten ist eine Liste der ihm erteilten Bestellungen zu erstellen. Die Liste soll folgende Angaben enthalten: Lieferantename, Bestellung.

```

select distinct struct (Lieferantename : l.Name,
    erteilteBestellungen: (select b from l.erteilt as b))
from Lieferanten l
where l.Adresse.Ort = "Dortmund"

```

## LE 16

### Aufgabe 1

- a Der Klient identifiziert ein entferntes Objekt über seine systemweit eindeutige Objektreferenz (*object reference*). Im *Implementation Repository* ist festgelegt, wie die Objektreferenz auf den physischen Aufenthaltsort der zugehörigen Objekt-Implementierung abgebildet wird.
- b Eine Ausnahme (*exception*) tritt immer dann auf, wenn die gerufene Operation nicht ordnungsgemäß ausgeführt werden kann.
- c Bei »normaler« objektorientierter Programmierung kann ein Objekt nur mit Objekten innerhalb desselben Programms kommunizieren. Der ORB ermöglicht die Kommunikation mit Objekten sowohl innerhalb desselben Programms als auch mit Objekten in anderen Programmen. Diese Programme können sich auf einem anderen Rechner mit einem anderen Betriebssystem befinden und in einer anderen Programmiersprache erstellt sein.

### Aufgabe 2

- a Der ORB ist dafür verantwortlich, die Objekt-Implementierung zu finden, den *request* an die Objekt-Implementierung weiterzugeben und eine Antwort zurückzugeben.
- b Die ORB-Architektur besteht aus folgenden Komponenten: Die *IDL Stubs* bilden die lokalen Vertreter der entfernten Objekte. Das *Dynamic Invocation Interface* macht es dem Klienten möglich, einen Operationsaufruf zur Laufzeit zu generieren. Der *Object Adapter* nimmt die Operationsaufrufe entgegen und sorgt dafür, daß die zugehörige Operation der angesprochenen Objekt-Implementierung aufgerufen wird. Das Verbinden der Operationsaufrufe mit den Implementierungen dieser Operationen erfolgt durch das *IDL Skeleton* oder das *Dynamic Skeleton Interface*.
- c Statische Operationsaufrufe werden verwendet, wenn die Klassen-Schnittstelle des entfernten Objekts bei der Übersetzung des Klienten bekannt ist. Diese Klassen-Schnittstelle wird mittels IDL definiert. Der IDL-Compiler erzeugt dann die *IDL Stubs* und die *IDL Skeletons*. Müssen die Operationsaufrufe zur Laufzeit generiert werden, dann verwendet der Klient *das Dynamic Invocation Interface* und das *Dynamic Skeleton Interface*. Auf der Empfängerseite ist nicht bekannt, ob ein Operationsaufruf statisch oder dynamisch erstellt wurde.

### Aufgabe 3

Die OMA (*Object Management Architecture*) besteht aus folgenden Komponenten: Der ORB bildet den Kern der Architektur. *Object services* sind elementare Funktionen, die für die Entwicklung verteilter Anwendungen benötigt werden. *Common facilities* stellen

eine höhere Funktionalität bereit, die in vielen Anwendungsbereichen benötigt wird. *Domain interfaces* sind Klassen-Schnittstellen, die gezielt die Funktionalität einiger Anwendungsbereiche realisieren. *Application objects* bilden die eigentliche Anwendung.

#### Aufgabe 4

```
interface Konto
{ readonly attribute long Kontonummer;
  attribute float Habenzins;
  readonly attribute float Kontostand; //keine set-Operation
  void einzahlen (in float Betrag);
}
interface Girokonto: Konto
{ attribute float Sollzins;
  attribute float Dispokredit;
  void abheben(in float Betrag)
    raises (DispokreditUeberschritten);
  void gutschreibenZinsen (in string Quartal)
    raises (Quartal SchonGutgeschrieben);
  void abbuchenZinsen (in string Quartal)
    raises (Quartal SchonAbgebucht);
}
interface Sparkonto: Konto
{ readonly attribute string Art; //keine set-Operation
  void abheben(in float Betrag)
    raises (zuwenigGeld);
  void gutschreibenZinsen (in short Jahr)
    raises (Jahr SchonGutgeschrieben);
}
```

## LE 17

#### Aufgabe 1

Die Zwei-Schichten-Architektur besteht aus einer Anwendungsschicht, in der Benutzungsoberfläche und Fachkonzept in einer Schicht fest verzahnt sind, und einer Datenhaltungsschicht.

Die Drei-Schichten-Architektur besteht aus der GUI-Schicht bzw. der Benutzungsoberfläche, der Fachkonzeptschicht und der Schicht der Datenhaltung.

Bei der Drei-Schichten-Architektur sind zwei Ausprägungen möglich: strenge Schichtung, in der jede Schicht nur auf die direkt darunterliegende zugreifen darf und flexiblere Schichten-Architektur, in der jede Schicht alle darunter liegenden benutzen darf. Die Schichten der Drei-Schichten-Architektur werden entsprechend ihrer Aufgaben in feinere Schichten zerlegt. Dann ergeben sich folgende Schichten: GUI-Schicht (Präsentationsschicht), Fachkonzept-Zugriffsschicht, Fachkonzeptschicht, Datenhaltungs-Zugriffsschicht, Datenhaltungsschicht. Je nach Anwendung können auch weitere bzw. andere Schichten gebildet werden.

**Aufgabe 2**

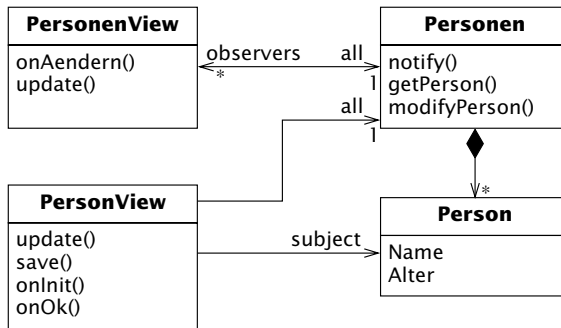
MVC-Architektur: Es handelt sich um ein *Framework*, das die Klassen *Model*, *View* und *Controller* zur Verfügung stellt und für eine Entkopplung von Fachkonzept und GUI-Präsentation sorgt.

Beim Beobachter-Muster entspricht das Subjekt dem *Model* und der Beobachter (*observer*) der Zusammenfassung von *Controller* und *View*. Dieses Muster ist nicht speziell für die Entkopplung von Benutzungsoberfläche und Fachkonzept ausgelegt, wird aber häufig dafür verwendet.

**Aufgabe 3**

Abb. LE17-3a zeigt das Klassendiagramm mit den GUI-Klassen *PersonenView* und *PersonView* und den Fachkonzeptklassen *Personen* und *Person*. Jedes Erfassungsfenster greift auf sein Fachkonzept-Objekt mittels *subject* zu. Von der *Container*-Klasse *Personen* gibt es nur ein einziges Objekt, das alle Objekte von *Person* kennt, was durch die Komposition modelliert wird. Bildlich gesprochen kann der *Container* in mehreren Objekten von *PersonenView* (*observers*) dargestellt werden.

Abb. LE17-3a:  
Klassendiagramm  
zur Erfassung und  
Listenanzeige von  
Personen



Das Szenario zum Ändern einer Person (Abb. LE17-3b) beginnt mit dem Aufruf von *onAendern()*. Das Listenfenster-Objekt beschafft sich mittels *getPerson()* die OID des zu ändernden *Person*-Objekts. Anschließend wird ein neues Erfassungsfenster für diese Person geöffnet und mittels *onInit()* initialisiert. Wenn die gewünschten Änderungen durchgeführt sind, wird *onOk()* aufgerufen und mittels *save()* das Fachkonzept-Objekt aktualisiert. Anschließend teilt das Erfassungsfenster dem *Container*-Objekt mittels *modifyPerson()* die Änderung eines Objekts mit. Der *Container* benachrichtigt mittels *notify()* alle seine Beobachter, indem er jedem Listenfenster eine *update*-Botschaft schickt. Jedes Listenfenster aktualisiert daraufhin seine Daten selbst.

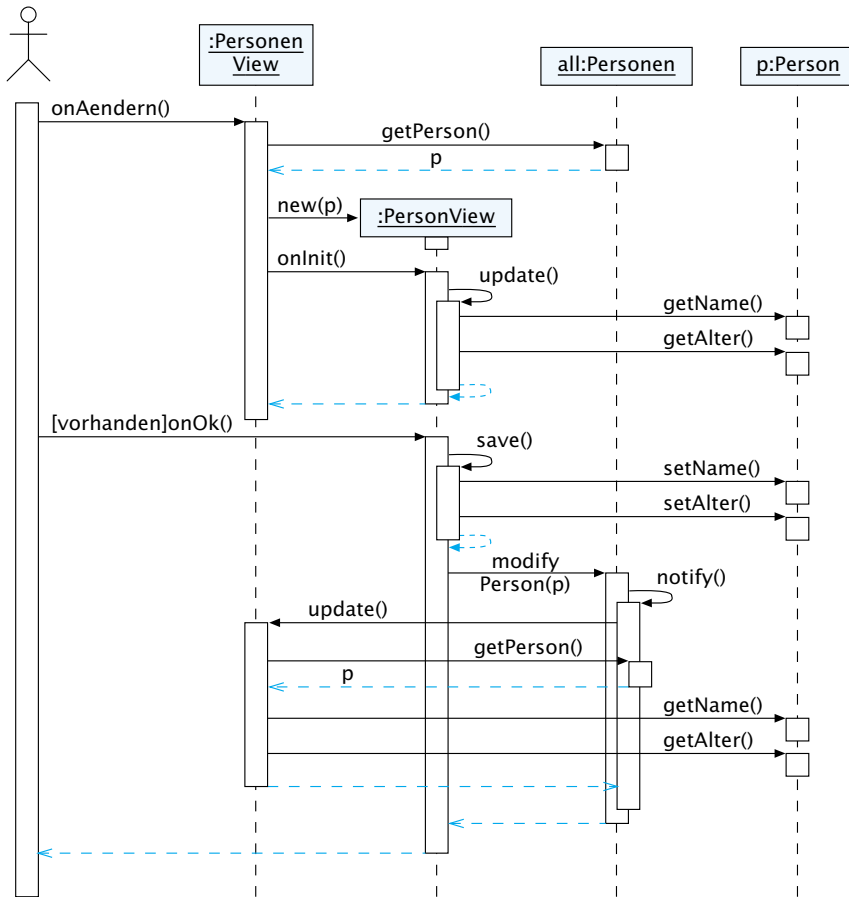


Abb. LE17-3b:  
Sequenzdiagramm zum Ändern einer Person mittels Beobachter-Muster

Das Objektdiagramm der Abb. LE17-3c zeigt, daß es nur einen *Container* `all:Personen` gibt. Er kennt die zwei Objekte von *Person* (*subject*) und zwei Listenfenster (*observers*). Jedes *View*-Objekt besitzt eine Objektverbindung zum *Container*.

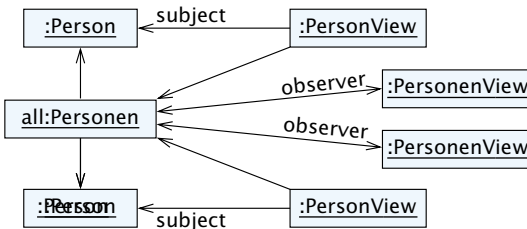


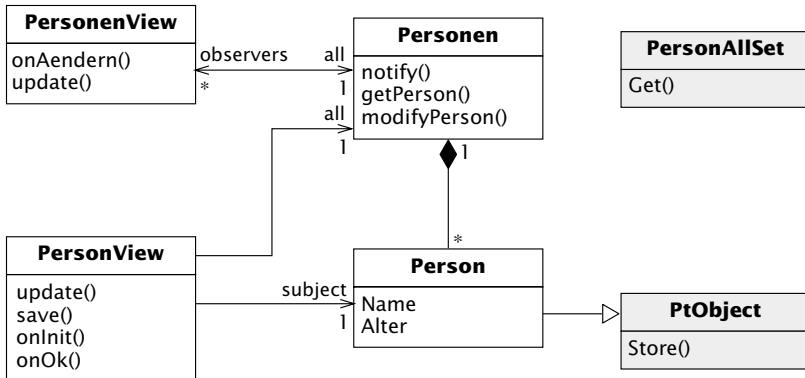
Abb. LE17-3c:  
Objektdiagramm

## LE18

### Aufgabe 1

- a** Um die Datenhaltung mittels Poet zu realisieren, muß das Klassendiagramm der Aufgabe 3 aus der Lehrinheit 17 nur um die Klasse `PersonAllSet` erweitert und die Klasse `Person` persistent gemacht werden (Abb. LE18-1a).

Abb. LE18-1a:  
Klassendiagramm  
zur Erfassung und  
Listenanzeige von  
Personen



- b** Auch das Sequenzdiagramm kann sehr leicht um die notwendigen Zugriffe erweitert werden, wobei alle bereits erstellten Interaktionen erhalten bleiben (Abb. LE18-1b).

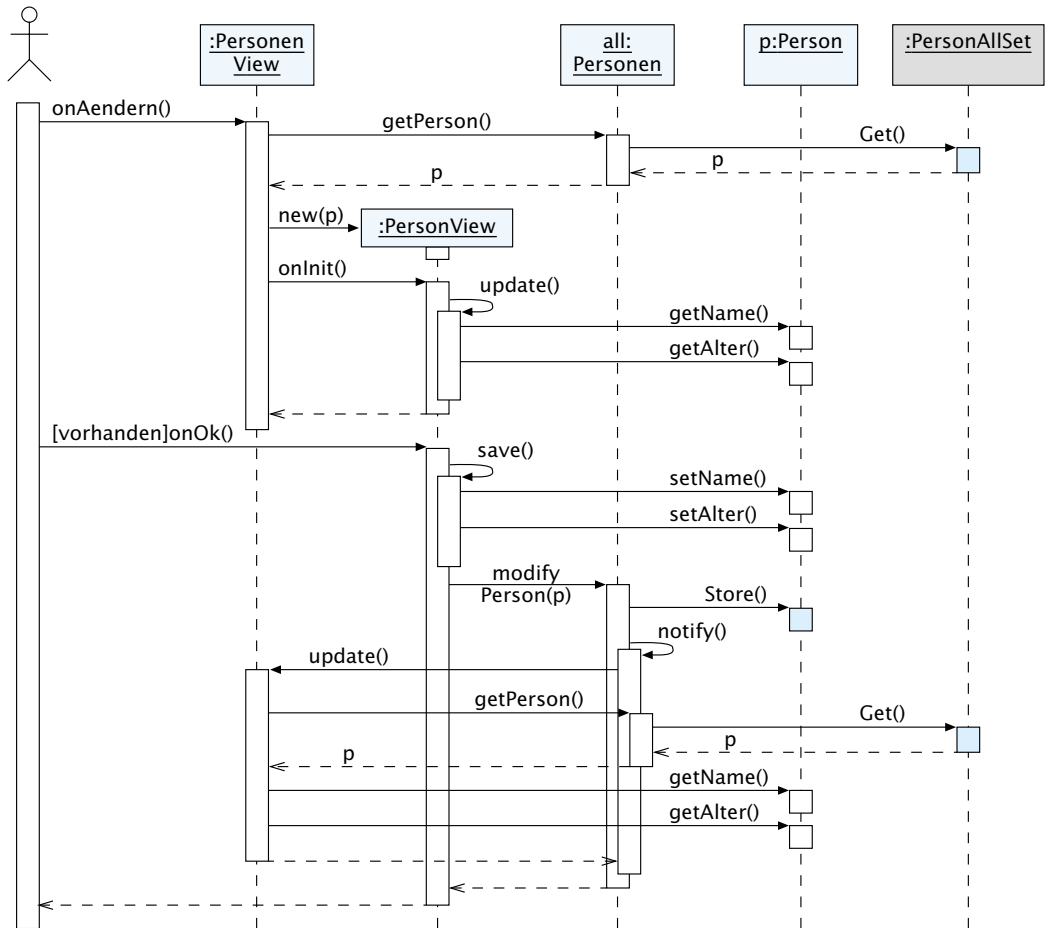


Abb. LE18-1b:  
Sequenzdiagramm  
zum Ändern einer  
Person mittels  
Beobachter-Muster

### Aufgabe 2

- a** Wird der Index mittels verketteter Liste realisiert, dann ändert sich an den *public*-Operationen der Klasse `Index` überhaupt nichts. Die *protected*-Operation `sort()`, die jedoch nur intern sichtbar ist, wird nicht mehr benötigt, wenn die Elemente in eine verkettete Liste sortiert eingefügt werden. Die *protected*-Attribute von `Index` müssen natürlich an die neue Struktur angepaßt werden.
- b** Wenn die Listenattribute im `Index` mitgeführt werden, dann ist die Parameterschnittstelle der generischen Klasse um den Parameter `fields` zu ergänzen, der eine Datenstruktur der Listenattribute – mit Ausnahme des Schlüsselattributs – darstellt. Dann erübrigt sich der Zugriff auf die Stammdatei, wenn die Liste aktualisiert wird.



### Aufgabe 3

- a** Materialisierung eines Objekts: Das Objekt wird aus einem oder mehreren Sätzen (Tupeln) der relationalen Datenbank aufgebaut.
- b** Eine Materialisierung erfolgt mit Hilfe der Proxy- und Broker-Klassen.
- c** Die abstrakte Klasse *PFWBroker* stellt die Schablonenmethode `objectWith()` zur Verfügung, in der die abstrakte Operation `materializeWith()` aufgerufen wird. Diese wird dann von allen konkreten *Broker*-Unterklassen mit einer konkreten Operation überschrieben.
- d** Um die *Performance* der Materialisierung zu steigern, werden bis zu sechs *Cache*-Speicher verwendet.
- e** Bei einer *ondemand*-Materialisierung wird die Materialisierung eines Objekts solange hinausgeschoben, bis es wirklich benötigt wird.
- f** Ein *rollback* wird immer ausgeführt, wenn eine Transaktion total verworfen wird.

### Aufgabe 4

In den Lehreinheiten 17 und 18 werden folgende Muster aus Kapitel 7 verwendet:

- *Singleton*-Muster, um sicherzustellen, daß es genau ein Objekt einer *Container*-Klasse gibt und um einfach darauf zuzugreifen.
- Beobachter-Muster, damit alle geöffneten Listenfenster nach der Neuerfassung oder Änderung im Erfassungsfenster aktualisiert werden. Dabei müssen sich Erfassungs- und Listenfenster nicht kennen.
- Schablonenmethode-Muster für die Materialisierung von Objekten einer relationalen Datenbank.
- Proxy-Muster (speziell Virtual Proxy), um eine *ondemand*-Materialisierung durchzuführen.
- Fabrikmethode-Muster, damit ein Proxy-Objekt sein zugehöriges *Broker*-Objekt erzeugt.
- *Singleton*-Muster, damit ein Proxy-Objekt genau ein zugehöriges *Broker*-Objekt erzeugt.
- Fassaden-Muster, wobei die Klasse *BrokerServer* eine Fassade für alle *Broker* bildet.

## Anhang 3: Gesamtglossar

### Abgeleitetes Attribut (*derived attribute*)

Abgeleitete Attribute lassen sich aus anderen Attributen berechnen. Sie dürfen nicht direkt geändert werden.

### Abstrakte Klasse (*abstract class*)

Von einer abstrakten Klasse können keine Objekte erzeugt werden. Die abstrakte Klasse spielt eine wichtige Rolle in Vererbungsstrukturen, wo sie die Gemeinsamkeiten einer Gruppe von Unterklassen definiert. Damit eine abstrakte Klasse verwendet werden kann, muß von ihr zunächst eine Unterklasse abgeleitet werden. Eine abstrakte Klasse kann auf zwei verschiedene Arten konzipiert werden:

- 1 Mindestens eine Operationen wird nicht spezifiziert bzw. implementiert, d.h. der Rumpf ist leer. Es wird nur die Signatur dieser Operation angegeben. Man spricht dann von einer abstrakten Operation.
- 2 Alle Operationen können – wie auch bei einer konkreten Klasse – vollständig spezifiziert bzw. implementiert werden. Es ist jedoch nicht beabsichtigt, von dieser Klasse Objekte zu erzeugen.

### Abstrakte Operation (*abstract operation*)

Eine Operation, für die nur die Signatur angegeben ist, die aber nicht spezifiziert bzw. implementiert ist (*pure virtual member function* in C++). Enthält eine Klasse mindestens eine abstrakte Operation, dann handelt es sich um eine abstrakte Klasse. Die zugehörige Spezifikation bzw. Implementierung wird erst in den Unterklassen angegeben.

### Abstrakter Datentyp (*abstract data type*)

Der abstrakte Datentyp (ADT) ist ursprünglich ein Konzept des Entwurfs. Ein abstrakter Datentyp wird ausschließlich über seine (Zugriffs-) Operationen definiert, die auf Exemplare dieses Typs angewendet werden. Die Repräsentation der Daten und die Wahl der Algorithmen zur Realisierung der Operationen sind nach außen nicht sichtbar, d.h. der ADT realisiert das Geheimnisprinzip. Von einem abstrakten Datentyp können beliebig viele Exemplare erzeugt werden. Die Klasse stellt eine Form des abstrakten Datentyps dar.

### Abstraktion (*abstraction*)

- 1 Abstraktion, als ein Prozeß betrachtet, bezeichnet die Vorgehensweise, die wesentlichen Informationen über etwas zu ermitteln und die unwesentlichen Informationen zu ignorieren.
- 2 Abstraktion, als Ergebnis betrachtet, bezeichnet ein Modell oder einen bestimmten Blickwinkel.

### **Aggregation (*aggregation*)**

Eine Aggregation ist ein Sonderfall der Assoziation. Sie liegt dann vor, wenn zwischen den Objekten der beteiligten Klassen eine Beziehung vorliegt, die sich als »ist Teil von« oder »besteht aus« beschreiben läßt.

### **Akteur (*actor*)**

Ein Akteur ist eine Rolle, die ein Benutzer des Systems spielt. Akteure befinden sich außerhalb des Systems. Akteure können Personen oder externe Systeme sein.

### **Aktion (*action*)**

Eine Aktion ist eine atomare Operation, die durch ein Ereignis ausgelöst wird und sich selbst beendet. Sie kann mit einer Transition verbunden sein. *Entry*-Aktionen werden bei Eintritt, und *exit*-Aktionen bei Verlassen des Zustandes ausgeführt.

### **Aktivität (*activity*)**

Eine Aktivität ist eine Operation, die mit einem Zustand eines Zustandsautomaten verbunden ist. Sie beginnt bei Eintritt und endet bei Verlassen des Zustandes. Sie kann alternativ durch ein Paar von Aktionen, eine zum Starten und eine zum Beenden der Aktivität, beschrieben oder durch ein weiteres Zustandsdiagramm verfeinert werden.

### **Aktivitätsdiagramm (*activity diagram*)**

Ein Aktivitätsdiagramm ist der Sonderfall eines Zustandsdiagramms, bei dem – fast – alle Zustände mit einer Verarbeitung verbunden sind. Ein Zustand wird verlassen, wenn die Verarbeitung beendet ist. Außerdem ist es möglich, eine Verzweigung des Kontrollflusses zu spezifizieren und zu beschreiben, ob die Verarbeitungsschritte in festgelegter oder beliebiger Reihenfolge ausgeführt werden können.

### **Analyse (*analysis*)**

Aufgabe der Analyse ist die Ermittlung und Beschreibung der Anforderungen eines Auftraggebers an ein Softwaresystem. Das Ergebnis soll die Anforderungen vollständig, widerspruchsfrei, eindeutig, präzise und verständlich beschreiben.

### **Analysemuster (*analysis pattern*)**

Ein Analysemuster ist eine Gruppe von Klassen mit feststehenden Verantwortlichkeiten und Interaktionen, die eine bestimmte – wiederkehrende – Problemlösung beschreiben.

### **Analyseprozeß**

Der Analyseprozeß beschreibt die methodische Vorgehensweise zur Erstellung eines objektorientierten Analysemodells. Er besteht aus einem Makroprozeß, der die grundlegenden Vorgehensschritte vorgibt und der situations- und anwendungsspezifischen Anwendung von methodischen Regeln.

### **Anfragesprache**

→OQL

**Assoziation (*association*)**

Eine Assoziation modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Binäre Assoziationen verbinden zwei Objekte. Eine Assoziation zwischen Objekten einer Klasse heißt reflexiv. Jede Assoziation wird beschrieben durch Kardinalitäten und einen optionalen Assoziationsnamen oder Rollennamen. Sie kann um Restriktionen ergänzt werden. Besitzt eine Assoziation selbst wieder Attribute und ggf. Operationen und Assoziationen zu anderen Klassen, dann wird sie zur assoziativen Klasse. Die Qualifikationsangabe (*qualifier*) zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Eine abgeleitete Assoziation liegt vor, wenn die gleichen Abhängigkeiten bereits durch andere Assoziationen beschrieben werden. Sonderfälle der Assoziation sind die Aggregation und die Komposition.

In der Analyse ist jede Assoziation inhärent bidirektional. Im Entwurf wird die gewünschte Navigationsrichtung angegeben.

**Assoziative Klasse (*association class*)**

Eine assoziative Klasse besitzt sowohl die Eigenschaften der Assoziation als auch die der Klasse.

**Attribut (*attribute*)**

Attribute beschreiben Daten, die von den Objekten der Klasse angenommen werden können. Alle Objekte einer Klasse besitzen dieselben Attribute, jedoch im allgemeinen unterschiedliche Attributwerte. Jedes Attribut ist von einem bestimmten Typ und kann einen Anfangswert (*default*) besitzen. Bei der Implementierung muß jedes Objekt Speicherplatz für alle seine Attribute reservieren. Der Attributname ist innerhalb der Klasse eindeutig. Abgeleitete Attribute lassen sich aus anderen Attributen berechnen.

**Attributspezifikation (*attribute specification*)**

Ein Attribut wird durch folgende Angaben spezifiziert:

Name: Typ = Anfangswert

{mandatory, key, frozen, Einheit: ..., Beschreibung:...}

wobei gilt:

mandatory = Muß-Attribut, key = Schlüsselattribut, frozen = Attributwert nicht änderbar.

**Balancierter Makroprozeß**

Der balancierte Makroprozeß unterstützt die Gleichgewichtigkeit von statischem und dynamischem Modell. Er beginnt mit dem Erstellen von Geschäftsprozessen und der Identifikation von Klassen. Dann werden statisches und dynamisches Modell parallel erstellt und deren Wechselwirkungen berücksichtigt.

**Beobachter-Muster (*observer pattern*)**

Das Beobachter-Muster ist ein objektbasiertes Verhaltensmuster. Es sorgt dafür, daß bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

### **Botschaft (*message*)**

Eine Botschaft ist die Aufforderung eines Senders (*client*) an einen Empfänger (*server, supplier*) eine Dienstleistung zu erbringen. Der Empfänger interpretiert diese Botschaft und führt eine Operation aus.

### **Container-Klasse**

Eine *Container*-Klasse ist eine Klasse, deren Objekte Mengen von Objekten (anderer) Klassen sind. Sie können homogene Mengen verwalten, d.h. alle Objekte einer Menge gehören zur selben Klasse, oder auch heterogene Mengen, d.h. die Objekte einer Menge gehören zu unterschiedlichen Unterklassen einer gemeinsamen Oberklasse. *Container*-Klassen werden oft mittels generischer Klassen realisiert.

### **CORBA (*Common Object Request Broker Architecture*)**

CORBA ist der OMG-Standard, der spezifiziert, wie Objekte in einer verteilten, heterogenen Umgebung kommunizieren. Er beschreibt den Aufbau des ORB, seine Bestandteile sowie deren Verhalten und Schnittstellen.

### **CRC-Karte (*Class/Responsibility/Collaboration*)**

Eine CRC-Karte ist eine Karteikarte. Oben auf der Karte wird der Name der Klasse (*class*) eingetragen. Die restliche Karte wird in zwei Hälften geteilt. Auf der einen Hälfte werden die Verantwortlichkeiten (*responsibilities*) der Klasse notiert. Darunter sind sowohl das Wissen der Klasse als auch die zur Verfügung gestellten Operationen zu verstehen. Auf der rechten Seite wird eingetragen, mit welchen anderen Klassen die beschriebene Klasse zusammenarbeiten muß (*collaborations*).

### **Datenbanksystem (*data base system*)**

Ein Datenbanksystem besteht aus einer oder mehreren Datenbanken, einem *Data Dictionary* und einem Datenbankmanagementsystem. In der Datenbank sind alle Daten gespeichert. Das *Data Dictionary* (DD) enthält das Datenbankschema, das den Aufbau der Daten der Datenbank(en) beschreibt. Die Verwaltung und zentrale Kontrolle der Daten ist Aufgabe des Datenbankmanagementsystems.

### **Daten-basierter Makroprozeß**

Beim daten-basierten Makroprozeß wird zunächst das Klassendiagramm erstellt und aufbauend darauf werden die Geschäftsprozesse und die anderen Diagramme des dynamischen Modells entwickelt.

### **Datendefinitionssprache (*data definition language*)**

Die Datendefinitionssprache (DDL) ist eine Sprache, die ein relationales Datenbanksystem zur Verfügung stellt und die zur formalen Definition des logischen Schemas – d.h. den leeren Tabellen der re-

lationalen Datenbank – dient. Als Standard hat sich die Sprache SQL etabliert.

#### **Datenhaltungsschicht (*storage tier, database tier*)**

Die Datenhaltungsschicht realisiert die jeweilige Form der Datenspeicherung, z.B. mit einem objektorientierten oder relationalen Datenbanksystem oder mit flachen Dateien.

#### **Datenmanipulationssprache (*data manipulation language*)**

Die Datenmanipulationssprache (DML) dient dazu, die leeren Tabellen einer relationalen Datenbank mit Daten zu füllen und diese Daten zu ändern. Eine DML enthält keine Kontrollstrukturen und Prozedurkonzepte. Als Standard hat sich die Sprache SQL etabliert.

#### **Datenmodell**

Jedem Datenbanksystem liegt ein Datenmodell zugrunde, in dem festgelegt wird, welche Eigenschaften und Struktur die Datenelemente besitzen dürfen, welche Konsistenzbedingungen einzuhalten sind und welche Operationen zum Speichern, Suchen, Ändern und Löschen von Datenelementen existieren. Es lassen sich relationale und objektorientierte Datenmodelle unterscheiden.

#### **DDL (*Data Definition Language*)**

→Datendefinitionssprache

#### **Destruktor (*destructor*)**

Ein Destruktor ist eine Operation, die ein Objekt löscht.

#### **Dialog (*dialog*)**

Ein Dialog ist eine Interaktion zwischen einem Benutzer und einem Dialogsystem, um ein bestimmtes Ziel zu erreichen. Ein Benutzer ist ein Mensch, der mit dem Dialogsystem arbeitet /ISO 9241-10/. Arbeitsschritte, die zur direkten Aufgabenerfüllung dienen, bezeichnet man als Primärdialog. Benötigt der Benutzer situationsabhängig zusätzliche Informationen, dann werden diese Hilfsdienste durch Sekundärdialoge erledigt.

#### **Dialogmodus**

Ein modaler Dialog (*modal dialog*) muß beendet sein, bevor eine andere Aufgabe der Anwendung durchgeführt werden kann. Ein nicht-modaler Dialog (*modeless dialog*) ermöglicht es dem Benutzer, den aktuellen Dialog zu unterbrechen, während das ursprüngliche Fenster geöffnet bleibt.

#### **DML (*Data Manipulation Language*)**

→Datenmanipulationssprache

#### **Drei-Schichten-Architektur (*three-tier architecture*)**

Die Drei-Schichten-Architektur besteht aus der GUI-Schicht (Schicht der Benutzungsoberfläche), der Fachkonzeptschicht und der Schicht der Datenhaltung. Es sind zwei Ausprägungen möglich: die strenge und die flexible Drei-Schichten-Architektur.

#### **Dynamisches Binden (*dynamic binding*)**

→spätes Binden

### Dynamisches Modell

Das dynamische Modell ist der Teil des OOA-Modells, welches das Verhalten des zu entwickelnden Systems beschreibt. Es realisiert außer den Basiskonzepten (Objekt, Klasse, Operation) die dynamischen Konzepte (Geschäftsprozeß, Szenario, Botschaft, Zustandsautomat).

### Einfachvererbung (*single inheritance*)

Bei der Einfachvererbung besitzt jede Unterklasse genau eine direkte Oberklasse. Es entsteht eine Baumstruktur.

### Elementare Klasse (*support class*)

Wird der Typ eines Attribut wieder durch eine Klasse realisiert, dann spricht man von einer elementaren Klasse. Sie wird nicht in das Klassendiagramm eingetragen.

### Entwurf (*design*)

Aufgabe des Entwurfs ist – aufbauend auf dem Ergebnis der Analyse – die Erstellung der Softwarearchitektur und die Spezifikation der Komponenten, d.h. die Festlegung von deren Schnittstellen, Funktions- und Leistungsumfang. Das Ergebnis soll die zu realisierenden Programme auf einem höheren Abstraktionsniveau widerspiegeln.

### Entwurfsmuster (*design pattern*)

Ein Entwurfsmuster gibt eine bewährte, generische Lösung für ein immer wiederkehrendes Entwurfsproblem an, das in bestimmten Situationen auftritt. Es lassen sich klassen- und objektbasierte Muster unterscheiden. Klassenbasierte Muster werden durch Vererbungen ausgedrückt. Objektbasierte Muster beschreiben in erster Linie Beziehungen zwischen Objekten. Beispiele für Entwurfsmuster sind das Beobachter-Muster und das *Singleton*-Muster.

### Ereignis (*event*)

Ein Ereignis tritt immer zu einem Zeitpunkt auf und besitzt keine Dauer. Es kann sein: eine wahr werdende Bedingung, ein Signal, eine Botschaft (Aufruf einer Operation), eine verstrichene Zeitspanne oder das Eintreten eines bestimmten Zeitpunkts. In den beiden letzten Fällen spricht man von zeitlichen Ereignissen.

### Erfassungsfenster

Das Erfassungsfenster bezieht sich auf ein einzelnes Objekt einer Klasse. Jedes Attribut der Klasse wird auf ein Interaktionselement des Fensters abgebildet. Das Erfassungsfenster dient zum Erfassen und Ändern von Objekten und zum Erstellen und Entfernen von Verbindungen zu anderen Objekten.

### Erzeugungsmuster (*creational pattern*)

Erzeugungsmuster helfen dabei, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden.

### Exemplar (*instance*)

→Objekt

**Fabrikmethode-Muster (*factory method pattern*)**

Das Fabrikmethode-Muster ist ein klassenbasiertes Erzeugungsmuster. Es bietet eine Schnittstelle zum Erzeugen eines Objekts an, wobei die Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist.

**Fachkonzeptschicht (*application logic tier*)**

Die Fachkonzeptschicht modelliert in einer Drei-Schichten-Architektur die fachliche Anwendung und die Zugriffe auf die Datenhaltungsschicht. Das OOA-Modell bildet die erste Version der Fachkonzeptschicht.

**Fassaden-Muster (*facade pattern*)**

Das Fassaden-Muster ist ein objektbasiertes Strukturmuster. Es bietet eine einfache Schnittstelle zu einer Menge von Schnittstellen (Paket) an. Die Fassadenklasse definiert eine Schnittstelle, um die Benutzung des Pakets zu vereinfachen.

**Fenstertypen**

Es lassen sich folgende Fenstertypen unterscheiden: Anwendungsfenster, Unterfenster, Dialogfenster und Mitteilungsfenster. Das Anwendungsfenster erscheint nach dem Aufruf der Anwendung, Unterfenster unterstützen die Primärdialoge, Dialogfenster werden für Sekundärdialoge benötigt und Mitteilungsfenster sind spezialisierte Dialogfenster.

**Flache Dateien (*flat files*)**

Unter einer Speicherverwaltung mit flachen Dateien ist eine Organisationsform zu verstehen, die nur rudimentäre Zugriffsoperationen anbietet.

**Flexible Drei-Schichten-Architektur**

Eine flexible Drei-Schichten-Architektur ergibt sich, wenn die GUI-Schicht sowohl auf die Fachkonzeptschicht als auch auf die Datenhaltungsschicht zugreifen darf.

**Formale Inspektion**

Die formale Inspektion ist ein formales Verfahren zur manuellen Prüfung der Dokumentation.

**Framework**

Ein *Framework* besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und insbesondere aus abstrakten Klassen, die Schnittstellen definieren. Die abstrakten Klassen enthalten sowohl abstrakte als auch konkrete Operationen. Im allgemeinen wird vom Anwender (=Programmierer) des *Frameworks* erwartet, daß er Unterklassen definiert, um das *Framework* zu verwenden und anzupassen.



### **Geheimnisprinzip (*information hiding*)**

Die Einhaltung des Geheimnisprinzips bedeutet, daß die Attribute und die Implementierung der Operationen außerhalb der Klasse nicht sichtbar sind.

### **Generalisierung (*generalization*)**

→Vererbung

### **Generische Klasse (*parameterized class, template*)**

Eine generische Klasse ist eine Beschreibung einer Klasse mit einem oder mehreren formalen Parametern. Sie definiert daher eine Familie von Klassen. *Container*-Klassen werden häufig als generische Klassen realisiert.

### **Geschäftsprozeß (*use case*)**

Ein Geschäftsprozeß (*use case*) besteht aus mehreren zusammenhängenden Aufgaben, die von einem Akteur durchgeführt werden, um ein Ziel zu erreichen bzw. ein gewünschtes Ergebnis zu erstellen.

### **Geschäftsprozeßdiagramm (*use case diagram*)**

Ein Geschäftsprozeßdiagramm beschreibt die Beziehungen zwischen Akteuren und Geschäftsprozessen in einem System. Auch Beziehungen zwischen Geschäftsprozessen können eingetragen werden. Es gibt auf einem auf hohem Abstraktionsniveau einen guten Überblick über das System und seine Schnittstellen zur Umgebung.

### **Geschäftsprozeßschablone (*use case template*)**

Die Geschäftsprozeßschablone ermöglicht eine semi-formale Spezifikation von Geschäftsprozessen. Sie enthält folgende Informationen: Name, Ziel, Kategorie, Vorbedingung, Nachbedingung Erfolg, Nachbedingung Fehlschlag, Akteure, auslösendes Ereignis, Beschreibung des Standardfalls sowie Erweiterungen und Alternativen zum Standardfall.

### **Gestaltungsregelwerk (*style guide*)**

Ein Gestaltungsregelwerk schreibt vor, wie die Benutzungsoberfläche von Anwendungen gestaltet wird. Es soll sicherstellen, daß das *look and feel* über verschiedene Anwendungen hinweg gleich bleibt. *Style guides* können sowohl Regelwerke des GUI-Herstellers oder auch unternehmenseigene Gestaltungsregelwerke sein.

### **GUI**

Ein GUI (*graphical user interface*) ist eine grafische Benutzungsoberfläche. Sie besteht aus einer Dialogkomponente (Bedienungsabläufe) und einer E/A-Komponente (Gestaltung der Informationen).

### **GUI-Schicht (*presentation tier*)**

Die GUI-Schicht ist in einer Drei-Schichten-Architektur sowohl für die Dialogführung und die Präsentation der fachlichen Daten (z.B. in Fenstern) als auch für die Kommunikation mit der Fachkonzeptschicht und ggf. mit der Datenhaltungsschicht zuständig.

**GUI-System (GUI system)**

Das GUI-System ist ein Softwaresystem, das die graphische Oberfläche verwaltet und die Kommunikation mit den Anwendungen abwickelt. Ein GUI-System wird vereinfachend auch Fenstersystem genannt.

**Identität**

→ Objektidentität

**IDL (Interface Definition Language)**

Die Schnittstellensprache IDL ist eine Sprache zur Spezifikation der Schnittstellen aller Objekte, die von den Klienten verwendet werden. IDL ist eine rein beschreibende Sprache. Die Implementierung erfolgt in einer Programmiersprache, z.B. C++.

**Instanz**

Der Begriff Instanz zur Bezeichnung eines Exemplars einer Klasse wurde aus dem Englischen übernommen (*instance*) und eingedeutscht.

→ Objekt

**Interaktionsdiagramm (interaction diagram)**

In der UML ist Interaktionsdiagramm der Oberbegriff von Sequenz- und Kollaborationsdiagramm. Bei anderen Methoden wird der Begriff Interaktionsdiagramm für das Sequenzdiagramm verwendet.

**Interaktionselement (control)**

Ein Interaktionselement dient zur Ein- und/oder zur Ausgabe von Informationen. Das sind beispielsweise Textfelder, Schaltflächen und Listfelder.

**JDBC (Java Database Connectivity)**

Mit JDBC hat *Sun Microsystems* einen Standard definiert, um aus Java-Programmen heraus auf relationale Datenbanksystem zugreifen zu können. JDBC ist durch die Verwendung von Java als Programmiersprache vollständig objektorientiert und plattformunabhängig.

**Kardinalität (multiplicity)**

Die Kardinalität bezeichnet die Wertigkeit einer Assoziation, d.h. sie spezifiziert die Anzahl der an der Assoziation beteiligten Objekte.

**Klasse (class)**

Eine Klasse definiert für eine Kollektion von Objekten deren Struktur (Attribute), das Verhalten (Operationen) und Beziehungen (Assoziationen, Vererbungsstrukturen). Klassen besitzen – mit Ausnahme von abstrakten Klassen – einen Mechanismus, um neue Objekte zu erzeugen. Der Klassenname muß mindestens im Paket, besser im gesamten System eindeutig sein.

### **Klassenattribut (*class scope attribute*)**

Ein Klassenattribut liegt vor, wenn nur ein Attributwert für alle Objekte der Klasse existiert. Klassenattribute sind von der Existenz der Objekte unabhängig.

### **Klassenbibliothek**

Eine Klassenbibliothek ist eine organisierte Sammlung von Klassen, aus denen der Entwickler nach Bedarf Einheiten verwendet, d.h. Objekte dieser Klassen definiert und Operationen darauf anwendet oder Unterklassen bildet. Klassenbibliotheken können unterschiedliche Topologien besitzen.

### **Klassendiagramm (*class diagram*)**

Das Klassendiagramm stellt die Klassen, die Vererbung und die Assoziationen zwischen Klassen dar. Zusätzlich können Pakete modelliert werden.

### **Klassenextension (*extent*)**

Unter der Klassenextension ist die Menge aller Objekte einer Klasse zu verstehen. Die Klassenextension wird im Entwurf durch *Container*-Klassen realisiert, während in der Analyse jede Klasse die Eigenschaft der Objektverwaltung besitzt. Bei objektorientierten Datenbanksystemen kann der Programmierer entscheiden, ob eine Klassenextension erzeugt werden soll. Falls der *extent* angelegt wird, dann wird ein neu erzeugtes Objekt automatisch eingefügt, beim Löschen wieder entfernt. Das Konzept der Klassenextension ermöglicht die Durchführung von Operationen (z.B. Selektionen) auf der Menge aller Objekte einer Klasse.

### **Klassenoperation (*class scope operation*)**

Eine Klassenoperation ist eine Operation, die für eine Klasse statt für ein Objekt der Klasse ausgeführt wird.

### **Klassenvariable**

→Klassenattribut

### **Klient (*client*)**

- 1** Der Klient ist eine Softwareeinheit, die eine Operation eines Objekts auf einem entfernten Server benutzen möchte.
- 2** Der Klient ist eine Softwareeinheit, die eine Operation eines Objekts benutzen möchte.

### **Kollaborationsdiagramm (*collaboration diagram*)**

Ein Kollaborationsdiagramm beschreibt die Objekte und die Verbindungen zwischen diesen Objekten. An jede Verbindung (*link*) kann eine Botschaft in Form eines Pfeiles angetragen werden. Die Reihenfolge und Verschachtelung der Operationen wird durch eine hierarchische Numerierung angegeben.

### **Komplexes Objekt (*composite object, complex object*)**

Besitzt ein Objekt Attribute, die selbst wieder Objekte sind, so wird es als komplexes Objekt bezeichnet. Ein (Unter-) Objekt kann ebenfalls komplex sein.

**Komposition (*composition*)**

Die Komposition ist eine besondere Form der Aggregation. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch einem anderen Ganzen zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile.

**Kompositum-Muster (*composite pattern*)**

Das Kompositum-Muster ist ein objektbasiertes Strukturmuster. Es setzt Objekte zu Baumstrukturen zusammen, um *whole-part*-Hierarchien zu darzustellen. Dieses Muster ermöglicht es, sowohl einzelne Objekte als auch einen Baum von Objekten einheitlich zu behandeln.

**Konstruktor (*constructor*)**

Ein Konstruktor ist eine Operation, die ein neues Objekt einer Klasse erzeugt und es initialisiert.

**Konzept (*concept*)**

Der Begriff des Konzepts wird in der Informatik im Sinne von Leitidee verwendet, z.B. Konzepte der Programmierung, Konzepte der Objektorientierung. Ein Konzept beschreibt einen definierten Sachverhalt (z.B. eine Klasse) unter einem oder mehreren Gesichtspunkten.

**Listenfenster**

Das Listenfenster zeigt alle Objekte der Klasse an. Im allgemeinen enthält es von einem Objekt nur dessen wichtigste Attribute.

**Literal (*literal*)**

Literale sind Daten, die im Gegensatz zu Objekten in einer objektorientierten Datenbank keine Objektidentität besitzen. Sie können daher nur als Teil eines Objekts in einer Datenbank gespeichert werden.

**Makroprozeß**

Der Makroprozeß beschreibt auf einem hohen Abstraktionsniveau die einzelnen Schritte, die zur systematischen Erstellung eines OOA-Modells durchzuführen sind. Der Makroprozeß kann die Gleichgewichtigkeit von statischem und dynamischem Modell (balancierter Makroprozeß) unterstützen oder daten-basiert bzw. szenario-basiert sein.

**Mehrfachvererbung (*multiple inheritance*)**

Bei der Mehrfachvererbung kann jede Klasse mehrere direkte Oberklassen besitzen. Sie bildet einen azyklischen Graphen, der mehr als eine Wurzel haben kann (Netzstruktur). Bei der Mehrfachvererbung können Namenskonflikte auftreten.

**Mehr-Schichten-Architektur (*multi-tier architecture*)**

Eine Mehr-Schichten-Architektur entsteht, wenn die Drei-Schichten-Architektur um weitere Schichten erweitert wird bzw. die vorhandenen Schichten feiner zerlegt werden.

### Menü

Ein Menü besteht aus einer überschaubaren und meist vordefinierten Menge von Menüoptionen, aus der ein Benutzer eine oder mehrere auswählen kann. Bei einem Aktionsmenü lösen die Menüoptionen Anwendungsfunktion aus, bei einem Eigenschaftsmenü lassen sich Parameter einstellen. Es lassen sich *pop-up*-Menüs und Menübalken mit *drop-down*-Menüs unterscheiden.

### Metaklasse (*meta class*)

Eine Metaklasse ist eine Klasse, deren Exemplare selbst wieder Klassen sind.

### Methode (*method*)

- 1 Der Begriff »Methode« beschreibt die planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen.
- 2 In der Softwaretechnik wird der Begriff »Methode« als Oberbegriff von Konzepten, Notation und methodischer Vorgehensweise verwendet.
- 3 Alternative Bezeichnung für die Operation einer Klasse (z.B. in Java).
- 4 Implementierung einer Operation.

### Methodische Vorgehensweise (*method*)

Eine methodische Vorgehensweise ist eine planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen. Sie wird häufig als Methode bezeichnet.

### Methodologie (*methodology*)

Methodologie (Methodenlehre) bedeutet die Lehre von den in den Einzelwissenschaften angewendeten Methoden. Oft wird dieser Begriff synonym für »Methode« benutzt.

### Muster (*pattern*)

Ein Muster ist – ganz allgemein – eine Idee, die sich in einen praktischen Kontext als nützlich erwiesen hat und es wahrscheinlich auch in anderen sein wird. Muster beschreiben Strukturen von Klassen bzw. Objekten, die sich in Softwaresystemen wiederholt finden und dienen zur Lösung bekannter Probleme. Entsprechend ihrer Anwendung in der jeweiligen Phase unterscheidet man Analyse- und Entwurfsmuster.

### MVC (*Model/View/Controller*)

MVC besteht aus den drei Klassen *Model*, *View* und *Controller*. Das *Model*-Objekt repräsentiert das Fachkonzeptobjekt. Oft gibt es mehrere Möglichkeiten, die fachlichen Daten zu präsentieren. Für jede Präsentation gibt es ein *View*-Objekt. Das *Controller*-Objekt bestimmt, wie die Benutzungsoberfläche auf Eingaben reagiert. Jedes *View*-Objekt besitzt ein zugehöriges *Controller*-Objekt, das diese Darstellung mit der Eingabe verbindet. Das impliziert, daß es zu jedem *Model*-Objekt eine beliebige Anzahl von Paaren (*View*, *Controller*) geben kann, jedoch mindestens eines.

**Nachbedingung (*postcondition*)**

Die Nachbedingung beschreibt die Änderung, die durch eine Verarbeitung bewirkt wird, unter der Voraussetzung, daß vor ihrer Ausführung die Vorbedingung erfüllt war.

**Nachricht (*message*)**

→Botschaft

**Navigation (*navigability*)**

Die Navigation legt im Entwurf fest, ob eine Assoziation uni- oder bidirektional implementiert wird.

**Notation (*notation*)**

Darstellung von Konzepten durch eine festgelegte Menge von grafischen und/oder textuellen Symbolen, zu denen eine Syntax und Semantik definiert ist.

**Oberklasse**

In einer Vererbungsstruktur heißt jede Klasse, von der eine Klasse Eigenschaften und Verhalten erbt, Oberklasse dieser Klasse. Mit anderen Worten: Eine Oberklasse ist eine Klasse, die mindestens eine Unterklasse besitzt.

**Objekt (*object*)**

- 1** Ein Objekt besitzt einen Zustand (Attributwerte und Verbindungen zu anderen Objekten), reagiert mit einem definierten Verhalten (Operationen) auf seine Umgebung und besitzt eine Objektidentität, die es von allen anderen Objekten unterscheidet. Jedes Objekt ist Exemplar einer Klasse.
- 2** Objekte besitzen eine Objektidentität und können – im Gegensatz zu Literalen – separat in einer objektorientierten Datenbank gespeichert werden.

**Objektdefinitionssprache**

→ODL

**Objektdiagramm (*object diagram*)**

Das Objektdiagramm stellt Objekte und ihre Verbindungen untereinander dar. Objektdiagramme werden im allgemeinen verwendet, um einen Ausschnitt des Systems zu einem bestimmten Zeitpunkt zu modellieren. Objekte können einen – im jeweiligen Objektdiagramm – eindeutigen Namen besitzen oder es können anonyme Objekte sein. In verschiedenen Objektdiagrammen kann der gleiche Name unterschiedliche Objekte kennzeichnen.

**Objektidentität (*object identity*)**

- 1** Jedes Objekt besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Selbst wenn zwei Objekte zufällig dieselben Attributwerte besitzen, haben sie eine unterschiedliche Identität.

2 In objektorientierten Datenbanksystemen werden Objektidentitäten automatisch vom System generiert und verwaltet. Sie besitzen keine (verwendbare) Semantik und sind dem Programmierer nicht bekannt. Objektidentitäten können in objektorientierten Datenbanksystemen beispielsweise als Surrogate realisiert werden.

### **Objekt-Implementierung (*object implementation*)**

Die Objekt-Implementierung definiert bei verteilten Systemen das Verhalten eines Objekts auf dem Server, in dem sie festlegt, welche Verarbeitung beim Aufruf einer Operation auszuführen ist. Außerdem legt sie fest, welche Daten benötigt werden, um den Zustand eines konkreten Objekts zu repräsentieren.

### **Objektorientierte Analyse (*object oriented analysis*)**

Ermittlung und Beschreibung der Anforderungen an ein Softwaresystem mittels objektorientierter Konzepte und Notationen. Das Ergebnis ist ein OOA-Modell.

### **Objektorientierte Softwareentwicklung (*object oriented software development*)**

Bei einer objektorientierten Softwareentwicklung werden die Ergebnisse der Phasen Analyse, Entwurf und Implementierung objektorientiert erstellt. Für letztere werden objektorientierte Programmiersprachen verwendet. Auch die Verteilung auf einem Netz kann objektorientiert erfolgen.

### **Objektorientierter Entwurf (*object oriented design*)**

Aufbauend auf dem OOA-Modell erfolgt die Erstellung der Softwarearchitektur und die Spezifikation der Klassen aus Sicht der Realisierung. Das Ergebnis ist das OOD-Modell, das ein Spiegelbild der objektorientierten Programme auf einem höheren Abstraktionsniveau bildet.

### **Objektorientiertes Datenbanksystem (*object database system*)**

Ein objektorientiertes Datenbanksystem (ODBS) ist ein Datenbanksystem, dem ein objektorientiertes Datenmodell zugrunde liegt. Es integriert die Eigenschaften einer Datenbank mit den Möglichkeiten von objektorientierten Programmiersprachen.

### **Objektreferenz (*object reference*)**

Bei verteilten Systemen identifiziert der Klient ein Objekt auf dem Server über seine systemweit eindeutige Objektreferenz, die später auf die physische Adresse des Objekts abgebildet wird.

### **Objekt-relationale Abbildung (*object relational mapping*)**

Die objekt-relationale Abbildung gibt an, wie ein Klassendiagramm auf Tabellen einer relationalen Datenbank abgebildet wird. Sie enthält Abbildungsvorschläge für Klassen, Assoziationen und Vererbungsstrukturen. Ein weiterer Aspekt ist die Realisierung der Objektidentität in relationalen Datenbanken.

### **Objekt-relationales Datenbanksystem (*object-relational database system*)**

Objekt-relationale Datenbanksysteme verfolgen das Ziel, die besten Ideen aus der relationalen und der objektorientierten Welt zu verbinden. Das grundlegende Konzept bleibt weiterhin die Tabelle. Es wird um objektorientierte Konzepte wie Abstrakte Datentypen, Objektidentität, Operationen und Vererbung erweitert.

### **Objektverwaltung (*class extension, object warehouse*)**

In der Systemanalyse besitzen Klassen implizit die Eigenschaft der Objektverwaltung. Das bedeutet, daß die Klasse weiß, welche Objekte von ihr erzeugt wurden. Damit erhält die Klasse die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte einer Klasse durchzuführen.

### **ODBC (*Open Database Connectivity*)**

ODBC ist eine standardisierte Schnittstelle für den Zugriff auf relationale Datenbanksysteme. Sie wurde ursprünglich von Microsoft spezifiziert, hat sich aber inzwischen zu einem betriebssystemübergreifenden, allgemein akzeptierten de facto-Standard entwickelt.

### **ODL (*Object Definition Language*)**

Die Objektdefinitionssprache ODL ist eine Sprache, die ausschließlich zur Spezifikation von Klassen und Schnittstellen dient. Diese Spezifikation erfolgt dadurch unabhängig von der Implementierung in einer Programmiersprache. ODL unterstützt alle Konzepte des ODMG-Objektmodells.

### **ODMG (*Object Database Mangement Group*)**

Die ODMG ist eine Gruppe von Herstellern und Anwendern objektorientierter Datenbanksysteme. 1993 wurde von dieser Gruppe die erste Version eines Standards für objektorientierte Datenbanksysteme vorgeschlagen: ODMG-93 genannt.

### **ODMG-Objektmodell (*object model*)**

Das ODMG-Objektmodell spezifiziert die Konzepte, die von einem objektorientierten Datenbanksystem unterstützt werden. Es bildet die Grundlage für den ODMG-Standard.

### **ODMG-Standard (*object database standard ODMG*)**

Der ODMG-Standard 2.0 besteht aus dem Objektmodell, der ODL (*Object Definition Language*), dem Austauschformat OIF (*Object Interchange Format*), der deklarativen Sprache OQL (*Object Query Language*) und Sprachanbindungen zu C++, Smalltalk und Java. Es gibt zwei Stufen der Einhaltung des Standards: ODMG-konform (*ODMG compliant*) und ODMG-zertifiziert (*ODMG certified*).

### **OID**

→Objektidentität

### **OMA (*Object Management Architecture*)**

Die Grundlage aller Standardisierungsaktivitäten der ODMG ist die OMA. Diese Architektur unterteilt die Bestandteile einer verteilten Anwendung in mehrere Komponenten. Den Kern bildet der ORB



(*Object Request Broker*), der als Kommunikationszentrale im Mittelpunkt der Architektur steht. Weitere Komponenten sind die *application interfaces*, die *domain interfaces*, die *object services* und die *common facilities*.

#### **OMG (Object Management Group)**

Systemanbieter und Anwender objektorientierter Techniken haben sich 1989 zur OMG (*Object Management Group*) zusammengeschlossen. Die OMG verfolgt das Ziel, Standards und Spezifikationen für verteilte objektorientierte Anwendungen zu schaffen.

#### **OMG-Objektmodell (OMG object model)**

Das OMG-Objektmodell liegt allen Spezifikationen der OMG zugrunde. Es beschreibt alle objektorientierten Konzepte, die für Klienten wichtig sind und die Konzepte für die Ausführung der Operationen auf dem Server.

#### **OOA**

→Objektorientierte Analyse

#### **OOA-Modell**

Fachliche Lösung des zu realisierenden Systems, die in einer objektorientierten Notation modelliert wird. Das OOA-Modell besteht aus dem statischen und dem dynamischen Modell und ist das wichtigste Ergebnis der Analyse.

#### **OOD**

→Objektorientierter Entwurf

#### **OOD-Modell**

Technische Lösung des zu realisierenden Systems, die in einer objektorientierten Notation modelliert wird. Das OOD-Modell ist ein Abbild des späteren objektorientierten Programms.

#### **Operation (operation)**

Eine Operation ist eine Funktion, die auf die internen Daten (Attributwerte) eines Objekts Zugriff hat. Sie kann Botschaften an andere Objekte senden. Auf alle Objekte einer Klasse sind dieselben Operationen anwendbar. Für Operationen gibt es im allgemeinen in der Analyse eine fachliche Beschreibung. Sie wird in einer objektorientierten Programmiersprache durch eine Implementierung (Methode) realisiert. Abstrakte Operationen besitzen nur eine Signatur. Externe Operationen werden vom späteren Bediener des Systems aktiviert. Interne Operationen werden dagegen immer von anderen Operationen aufgerufen.

#### **OQL (Object Query Language)**

Die Anfragesprache OQL dient zur Formulierung von Selektionen in einer objektorientierten Datenbank. OQL baut auf dem *select-from-where*-Block von SQL2 auf. OQL kann sowohl als eigenständige, interaktive – nicht berechnungsvollständige – Datenbanksprache als auch eingebettet in verschiedene Programmiersprachen benutzt werden. OQL wurde im ODMG-Standard definiert.

**ORB (Object Request Broker)**

In verteilten System wird die Kommunikation zwischen Klient und Server vom ORB durchgeführt. Er ist vergleichbar mit einer Telefonvermittlung, der das Anrufen anderer Teilnehmer und das Entgegennehmen von Anrufen realisiert.

**Paket (package)**

Ein Paket faßt Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene auszudrücken. Pakete können im Paketdiagramm dargestellt werden.

**Parametrisierte Klasse**

→generische Klasse

**Persistenz**

Persistenz ist die Fähigkeit eines Objekts, über die Ausführungszeit eines Programms hinaus zu leben, d.h. die Daten dieses Objekt bleiben auch nach Beendigung des Programms erhalten und stehen bei einem Neustart wieder zur Verfügung.

**Polymorphe Operation**

Eine polymorphe Operation ist eine Operation, die erst zur Ausführungszeit an ein bestimmtes Objekt gebunden wird. Man spricht vom späten Binden (*late binding*) bzw. vom dynamischen Binden.

**Polymorphismus (polymorphism)**

Ein Name kann Objekte verschiedener Klassen bezeichnen. Jedes Objekt, das durch diesen Namen bezeichnet wird, kann auf die gleiche Botschaft auf seine eigene Art und Weise reagieren. Polymorphismus und spätes Binden sind untrennbar verbunden.

**Prototyp**

Ein Prototyp dient dazu, bestimmte Aspekte vor der Realisierung des Softwaresystems zu überprüfen. Der Prototyp der Benutzungsoberfläche zeigt die vollständige Oberfläche des zukünftigen Systems, ohne daß bereits Funktionalität realisiert ist.

**Proxy-Muster (proxy pattern)**

Das Proxy-Muster ist ein objektbasiertes Strukturmuster. Es kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreter-Objekts.

**Qualifikationsangabe (qualifier)**

Die Qualifikationsangabe ist ein spezielles Attribut der Assoziation, dessen Wert ein oder mehrere Objekte auf der anderen Seite der Assoziation selektiert. Mit anderen Worten: Die Qualifikationsangabe zerlegt die Menge der Objekte am anderen Ende der Assoziation in Teilmengen. Der *qualifier* kann auch aus mehreren Attributen bestehen.

### Relation

→Tabelle

### Relationales Datenbanksystem (*relational database system*)

Ein relationales Datenbanksystem (RDBS) ist ein Datenbanksystem, dem ein relationales Datenmodell zugrunde liegt. Die Daten werden in Form von Tabellen gespeichert.

### Request

Mit einem *request* fordert der Klient ein Objekt auf dem Server zur Ausführung einer Operation auf.

### Rolle (*role name*)

Die Rolle beschreibt, welche Bedeutung ein Objekt in einer Assoziation wahrnimmt. Eine binäre Assoziation besitzt maximal zwei Rollen.

### Schablonenmethode-Muster (*template method pattern*)

Das Schablonenmethode-Muster ist ein objektbasiertes Verhaltensmuster. Es definiert den Rahmen eines Algorithmus in einer Operation und delegiert Teilschritte an Unterklassen.

### Schnittstelle (*interface*)

- 1 In der UML besteht eine Schnittstelle nur aus Operationen, die keine Implementierung besitzen. Sie ist äquivalent zu einer Klasse, die keine Attribute, Zustände oder Assoziationen und ausschließlich abstrakte Operationen besitzt.
- 2 Die Menge aller Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die Schnittstelle der Klasse bzw. des Objekts.
- 3 Die IDL-Schnittstelle spezifiziert die Signaturen von Operationen, die ein Klient aufrufen kann. Sie stellt die wichtigste Komponente einer IDL-Definition dar.
- 4 Eine Java-Schnittstelle kann aus Konstanten und abstrakten Operationen bestehen.
- 5 Der ODMG-Standard verwendet außer der Klasse das Konzept der Schnittstelle, die nur das Verhalten spezifiziert. Von einer Schnittstelle können – im Gegensatz zur Klasse – keine Objekte erzeugt werden können.

### Sequenzdiagramm (*sequence diagram*)

Ein Sequenzdiagramm besitzt zwei Dimensionen. Die Vertikale repräsentiert die Zeit und auf der Horizontalen werden die Objekte angetragen. In das Diagramm werden die Botschaften eingetragen, die zum Aktivieren der Operationen dienen.

### Sichtbarkeit (*visibility*)

Die Sichtbarkeit legt fest, ob auf Attribute und Operationen außerhalb ihrer Klasse zugegriffen werden kann. Auch für Assoziationen kann die Sichtbarkeit definiert werden. Die UML unterscheidet die folgenden Sichtbarkeiten:

- *public* = sichtbar für alle Klassen.
- *protected* = sichtbar innerhalb der Klasse und für alle ihre Unterklassen.
- *private* = sichtbar nur innerhalb der Klasse.

### Signatur (*signature*)

- 1 Die Signatur einer Operation besteht aus dem Namen der Operation, den Namen und Typen aller Parameter, und dem Ergebnistyp der Operation.
- 2 Die Signatur einer Operation definiert den Namen der Operation, die Namen und Typen aller Parameter, den Ergebnistyp und die Bezeichnungen aller Ausnahmebehandlungen (*exceptions*) im Fehlerfall.

### Singleton-Muster (*singleton pattern*)

Das *Singleton*-Muster ist ein objektbasiertes Erzeugungsmuster. Es stellt sicher, daß eine Klasse genau ein Objekt besitzt und ermöglicht einen globalen Zugriff auf dieses Objekt.

### Software-Ergonomie

Die Software-Ergonomie befaßt sich mit der menschengerechten Gestaltung von Softwaresystemen. Sie verfolgt das Ziel, die Software an die Eigenschaften und Bedürfnisse der Benutzer anzupassen.

### Spätes Binden (*late binding*)

Beim späten Binden wird erst zur Ausführungszeit bestimmt, welche polymorphe Operation auf ein Objekt angewendet wird. Man spricht auch von dynamischem Binden. Das Gegenstück zum späten Binden ist das frühe Binden, das zur Übersetzungszeit stattfindet.

### SQL (*Structured Query Language*)

SQL ist eine deklarative Programmiersprache, d.h. sie besitzt im Unterschied zu den klassischen Programmiersprachen keine Schleifen, keine Prozeduren, keine Rekursion und keine ausreichenden mathematischen Operationen. Sie dient der Definition und Manipulation relationaler Datenbanken. 1983 wurde von ANSI und ISO ein SQL-Standard definiert. Weiterentwicklungen führten zum derzeitigen Standard SQL2, der 1992 veröffentlicht wurde, und zu SQL3 (noch nicht verabschiedet).

### Statisches Modell

Das statische Modell realisiert außer den Basiskonzepten (Objekt, Klasse, Attribut) die statischen Konzepte (Assoziation, Vererbung, Paket). Es beschreibt die Klassen des Systems, die Assoziationen zwischen den Klassen und die Vererbungsstrukturen. Desweiteren enthält es die Daten des Systems (Attribute). Die Pakete dienen dazu, Teilsysteme zu bilden, um bei großen Systemen einen besseren Überblick zu ermöglichen.

### Steuerelement (*control*)

→Interaktionselement

### Strenge Drei-Schichten-Architektur

Bei einer strengen Drei-Schichten-Architektur kann die GUI-Schicht nur auf die Fachkonzeptschicht und letztere nur auf die Datenehaltungsschicht zugreifen.

### Strukturmuster (*structural pattern*)

Strukturmuster befassen sich damit, wie Klassen und Objekte zu größeren Strukturen zusammengesetzt werden.

### Systemanalyse

→Analyse

### Szenario (*scenario*)

Ein Szenario ist eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen auszuführen sind. Diese Schritte sollen das Hauptziel des Akteurs realisieren und ein entsprechendes Ergebnis liefern. Ein Geschäftsprozeß wird durch eine Kollektion von Szenarios dokumentiert. Szenarios werden mit Hilfe von Sequenz- und Kollaborationsdiagrammen dokumentiert.

### Szenario-basierter Makroprozeß

Der szenario-basierte Makroprozeß beginnt dem Erstellen von Geschäftsprozessen und Interaktionsdiagrammen und leitet daraus das Klassendiagramm ab.

### Tabelle (*table*)

Relationale Datenbanksysteme speichern Daten in Form von Tabellen (Relationen). Jede Zeile der Tabelle wird als Tupel bezeichnet. Alle Tupel einer Tabelle müssen gleich lang sein. Jedes Tupel muß durch einen eindeutigen Schlüssel identifizierbar sein. Der Schlüssel (auch als Primärschlüssel bezeichnet) kann aus einem oder mehreren Attributen bestehen. Beziehungen zwischen Tabellen werden mittels Fremdschlüsseln realisiert.

### Transition (*transition*)

Eine Transition verbindet einen Ausgangs- und einen Folgezustand. Sie kann nicht unterbrochen werden und wird stets durch ein Ereignis ausgelöst. Ausgangs- und Folgezustand können identisch sein.

### Typ (*type*)

- 1 Jedes Attribut ist von einem bestimmten Typ. Er kann ein Standardtyp (z.B. Int), ein Aufzählungstyp, eine elementare Klasse oder eine Liste (*list of <Typ>*) sein.
- 2 Der Typ wird auch im Sinne von Schnittstellen- oder Klassen-Spezifikation verwendet. Er legt fest, auf welche Botschaften die Objekte einer Klasse reagieren können, d.h. der Typ definiert die Schnittstelle der Objekte. Ein Typ wird implementiert durch ein oder mehrere Klassen.

**Überschreiben (overriding)**

Von Überschreiben bzw. Redefinition spricht man, wenn eine Unterklasse eine geerbte Operation der Oberklasse – unter dem gleichen Namen – neu implementiert. Beim Überschreiben müssen die Anzahl und Typen der Ein-/Ausgabeparameter gleichbleiben. Bei der Implementierung der überschriebenen Operation wird im allgemeinen die entsprechende Operation der Oberklasse aufgerufen.

**UML**

*Unified Modeling Language*, die von Booch, Rumbaugh und Jacobson bei der *Rational Software Corporation* entwickelt und 1997 von der *OMG (Object Management Group)* als Standard akzeptiert wurde.

**Unterklasse (sub class)**

Jede Klasse, die in einer Vererbungshierarchie Eigenschaften und Verhalten von anderen Klassen erbt, ist eine Unterklasse dieser Klassen. Mit anderen Worten: Eine Unterklasse besitzt immer Oberklassen.

**Vererbung (generalization, inheritance)**

Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und Assoziationen der Basisklasse. Operationen der Basisklasse dürfen redefiniert werden. Es entsteht eine Klassenhierarchie oder Vererbungsstruktur. Außer der Einfachvererbung, bei der Klassen eine Baumstruktur bilden, gibt es die Mehrfachvererbung (Netzstruktur).

**Verhalten (behavior)**

Unter dem Verhalten eines Objekts sind die beobachtbaren Effekte aller Operationen zu verstehen, die auf das Objekt angewendet werden können. Das Verhalten einer Klasse wird bestimmt durch die Operationsaufrufe (Botschaften), auf die diese Klasse bzw. deren Objekte reagieren.

**Verhaltensmuster (behavioral pattern)**

Verhaltensmuster befassen sich mit der Interaktion zwischen Objekten und Klassen. Sie beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind. Sie lenken die Aufmerksamkeit weg vom Kontrollfluß hin zu der Art und Weise, wie die Objekte interagieren.

**Verkapselung (encapsulation)**

Die Verkapselung sagt aus, daß zusammengehörende Attribute und Operationen in einer Einheit zusammengefügt sind.

**Virtuelle Funktion**

→polymorphe Operation

**Vorbedingung (precondition)**

Die Vorbedingung beschreibt, welche Bedingungen vor dem Ausführen einer Verarbeitung erfüllt sein müssen, damit die Verarbeitung definiert ausgeführt werden kann.

### Anhang 3 Gesamtglossar

#### **Werkzeug (tool)**

In der Softwareentwicklung versteht man unter einem Werkzeug ein Programm, das als Hilfsmittel zur Entwicklung von Software eingesetzt wird.

#### **Zustand (state)**

- 1** Ein Zustand eines Zustandsautomaten ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet. Ein Zustand besteht solange, bis ein Ereignis eintritt, das eine Transition auslöst.
- 2** Der Zustand eines Objekts wird bestimmt durch seine Attributwerte und seine Verbindungen (*links*) zu anderen Objekten, die zu einem bestimmten Zeitpunkt existieren.

#### **Zustandsautomat (finite state machine)**

Ein Zustandsautomat besteht aus Zuständen und Transitionen. Er hat einen Anfangszustand und kann einen Endzustand besitzen.

#### **Zustandsdiagramm (statechart diagram)**

Das Zustandsdiagramm ist eine grafische Repräsentation des Zustandsautomaten.

#### **Zustandsmuster (state pattern)**

Das Zustandsmuster ist ein Entwurfsmuster, mit dem Objekt-Lebenszyklen des OOA-Modells systematisch in ein OOD-Klassendiagramm umgesetzt werden können. Es ist insbesondere für die Realisierung komplexer Zustandsautomaten gedacht.

#### **Zwei-Schichten-Architektur (two-tier architecture)**

Bei einer Zwei-Schichten-Architektur sind die Benutzungsoberfläche und das Fachkonzept fest in einer Schicht verzahnt. Die zweite Schicht realisiert die Datenhaltung.

# Referenzierte und ergänzende Literatur

/ANSI 92/

ANSI X3H2, Database Language SQL, X3.135–1992, 1992

/ANSI/IEEE Std. 729–1983/

IEEE Standard Glossary of Software Engineering Terminology  
IEEE 1993

/Ambler 97/

Ambler S.W.

Mapping Objects to Relational Databases

An AmbySoft Inc. White Paper, October 1997

[www.AmbySoft.com/mappingObjects.pdf](http://www.AmbySoft.com/mappingObjects.pdf)



/Atkinson et al. 98/

Atkinson M., DeWitt D., Maier D., Bancilhon F., Dittrich K., Zdonik S.

The Object-oriented Database System Manifesto

Proc. First International Conference on Deductive and Object-Oriented Databases

Kyoto, Dezember 1989

/Balzert 96/

Balzert Helmut

Lehrbuch der Softwaretechnik – Software-Entwicklung

Spektrum Akademischer Verlag, Heidelberg 1996

/Balzert 96a/

Balzert Heide

Methoden der objektorientierten Systemanalyse, 2. Auflage

Spektrum Akademischer Verlag, Heidelberg 1996

/Balzert 98/

Balzert Helmut

Lehrbuch der Softwaretechnik – Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung

Spektrum Akademischer Verlag, Heidelberg 1998

/Balzert 99/

Balzert Helmut

Lehrbuch Grundlagen der Informatik

Spektrum Akademischer Verlag, Heidelberg 1999

/Ben-Natan 95/

Ben-Natan R.

CORBA


A Guide to the Common Object Request Broker Architecture

McGraw-Hill, New York, 1995



## Literatur

- /Berard 93/  
Berard E.  
Essays on Object-Oriented Software-Engineering, Volume I  
Prentice Hall, Englewood Cliffs, 1993
- /Bertino, Martino 93/  
Bertino A., Martino L.  
Object-Oriented Database Systems  
Concepts and Architectures  
Addison-Wesley, Wokingham, 1993
- /Blair et al. 91/  
Blair G., Gallager J., Hutschison D., Shepherd D.  
Object-Oriented Languages, Systems and Applications  
Halstead Press, John Wiley & Sons, New York, 1991
- /Booch 91/  
Booch G.  
Object-Oriented Design with Applications  
The Benjamin/Cummings Publishing Company, Redwood City,  
1991
- /Booch 93/  
Das Design der C++ Booch Components  
Rational, 1993
- /Booch 94/  
Booch G.  
Object-Oriented Analysis and Design with Applications  
Second Edition  
The Benjamin/Cummings Publishing Company, Redwood City,  
1994
- /Booch 94a/  
Booch G.  
The Evolution of the Booch Method  
Report on Object Analysis & Design, May-June 1994, pp. 2-5
- /Booch 94b/  
Booch G.  
Objektorientierte Analyse und Design: Mit praktischen Anwen-  
dungsbeispielen  
Addison-Wesley, Bonn 1994  
deutsche Übersetzung von /Booch 94/
- /Booch 96/  
Booch G.  
Object Solutions, Managing the Object-Oriented Project  
Addison-Wesley, Menlo Park, California, 1996
- /Booch 96a/  
Booch G.  
Best of Booch  
Sigs Books, New York, 1996

- /Booch, Rumbaugh 95/  
 Booch G., Rumbaugh J.  
 Unified Method, Version 0.8  
 Rational Software Corporation, Santa Clara, 1995  
 www.rational.com
-  /Booch et al. 98/  
 Booch G., Rumbaugh J., Jacobson I.  
 The Unified Modeling Language User Guide  
 Addison-Wesley, Reading, Massachusetts 1998
- /Brown, Whitenack 95/  
 Brown K., Whitenack B.G.  
 Crossing Chasms: A Pattern Language for Object-RDBMS  
 in /Vlissides et al. 96/
- /Burkhardt 97/  
 Burkhardt R.  
 UML – Unified Modeling Language  
 Objektorientierte Modellierung für die Praxis  
 Addison-Wesley, Bonn, 1997
- /Buschmann et al. 96/  
 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.  
 Pattern-oriented Software Architecture  
 A System of Patterns  
 John Wiley & Sons, Chichester 1996
- /Carmichael 94/  
 Carmichael A. (ed.)  
 Object Development Methods  
 Sigs Books, New York, 1994
- /Caroll 95/  
 Caroll J. (ed.)  
 Scenario-Based Design  
 Envisioning Work and Technology in System Development  
 John Wiley & Sons, New York, 1995
- /Cattel, Barry 97/  
 Catell R.G.G., Barry D.K. (Hrsg.)  
 The Object Database Standard: ODMG 2.0  
 Morgan Kaufmann Publishers, San Francisco, California, 1997
- /Coad, Yourdon 91/  
 Coad P., Yourdon E.  
 Object-Oriented Analysis  
 2. Auflage  
 Yourdon Press, Prentice Hall, Englewood Cliffs, 1991
- /Coad, Yourdon 91a/  
 Coad P., Yourdon E.  
 Object-Oriented Design  
 Yourdon Press, Prentice Hall, Englewood Cliffs, 1991

## Literatur

/Coad 92/

Coad P.

Object-Oriented Patterns

Communications of the ACM, September 1992, pp. 152–159

/Coad, Yourdon 94/

Coad P., Yourdon E.

OOA

Objektorientierte Analyse

Prentice Hall Verlag, München, 1994

deutsche Übersetzung von /Coad, Yourdon 91/

/Coad, Yourdon 94a/

Coad P., Yourdon E.

OOD

Objektorientiertes Design

Prentice Hall Verlag, München, 1994

deutsche Übersetzung von /Coad, Yourdon 91a/

/Coad 95/

Coad P. mit North D., Mayfield M.

Object Models, Strategies, Patterns, and Applications

Yourdon Press, Prentice Hall, Englewood Cliffs, 1995

/Cockburn 97/

Cockburn A.

Structuring Use Cases with Goals, 1997

[www.members.aol.com/acockburn/papers/usecases.htm](http://www.members.aol.com/acockburn/papers/usecases.htm)



/Coleman et al. 94/

Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F.,

Jeremes P.

Object-Oriented Development

The Fusion Method

Prentice Hall, Englewood Cliffs, 1994

/Carroll 95/

Carroll J.M. (Hrsg.)

Szenario-Based Design

Envisioning Work and Technology in System Development

John Wiley & Sons, New York 1995

/Coplien et al. 95/

Coplien James.O., Schmidt D.C. (ed.), Coplien Jim

Pattern Languages of Program Design

Addison-Wesley, Reading Massachusetts, 1995

/DeMarco 79/

DeMarco T.

Structured Analysis and System Specification

Yourdon Press, Prentice Hall, Englewood Cliffs, 1979

- /Derr 95/  
Derr K.  
Applying OMT  
A Practical Step-by-Step Guide to Using the Object Modeling  
Technique  
Sigs Books, New York, 1995
- /Dittrich, Geppert 95/  
Dittrich K.R., Geppert A.  
Objektorientierte Datenbanksysteme – Stand der Technik  
HMD, Heft 183, Mai 1995, pp. 8–23
- /D'Souza 94/  
D'Souza D.  
Working with OMT, Part 2  
Journal of Object Oriented Programming, February 1994, pp. 68–72
- /Eisenecker 95/  
Eisenecker U.W.  
Objekte versus Komponenten – Der Weg zur flinken Software  
IX 9/1995, pp. 164–169
- /Eisenecker 98/  
Eisenecker U.W.  
Korrespondenz mit U. Eisenecker, November 1998
- /Fagan 76/  
Fagan M.E.  
Design and code inspections to reduce error in program develop-  
ment  
IBM Systems Journal, No. 3, 1976, pp. 182–211
- /Fagan 86/  
Fagan M.E.  
Advances in Software Inspections  
IEEE Transactions on Software Engineering  
July 1986, pp. 744–751
- /Firesmith, Eykholt 95/  
Firesmith D., Eykholt E.  
Dictionary of Object Technology  
The Definitive Desk Reference  
Sigs Books, New York, 1995
- /Fowler 97/  
Fowler M.  
UML Distilled – Applying the Standard Object Modeling Language  
Addison Wesley, Reading, Massachusetts, 1997
- /Fowler 97a/  
Fowler M.  
Analysis Patterns – Reusable Object Models  
Addison Wesley, Menlo Park, California, 1997

## Literatur

/Gamma et al. 95/

Gamma E., Helm R., Johnson R., Vlissides J.  
Design Patterns  
Elements of Reusable Object-Oriented Software  
Addison-Wesley, Reading, Massachusetts 1995

/Gamma et al. 96/

Gamma E., Helm R., Johnson R., Vlissides J.  
Entwurfsmuster  
Elemente wiederverwendbarer objektorientierter Software  
Addison-Wesley, Bonn, 1996  
Übersetzung von /Gamma et al. 95/

/Geppert 97/

Geppert A.  
Objektorientierte Datenbanksysteme  
Ein Praktikum  
dpunkt-Verlag, Heidelberg, 1997

/Gilb, Graham 93/

Gilb. T., Graham D.  
Software Inspection  
Addison Wesley, Wokingham, England 1993

/GUI Guide 93/

The GUI Guide  
International Terminology for the Windows Interface  
Microsoft Press, Redmond, 1993

/Hansen 96/

Hansen H. R.  
Wirtschaftsinformatik I  
Lucius & Lucius, Stuttgart, 1996

/Harel 87/

Harel D.  
Statecharts: A Visual Formalism for Complex Systems  
Science of Computer Programming 8, 1987, pp. 231–274

/Harel 88/

Harel D.  
On Visual Formalism  
Communications of the ACM, May 1988, pp. 514–530

/Harmon, Watson 98/

Harmon P., Watson M.  
Understanding UML: The Developer's Guide  
with a Web-Based Application in Java  
Morgan Kaufmann Publishers, San Francisco, California, 1998

- /Henderson-Sellers 92/  
Henderson-Sellers B.  
A Book of Object-Oriented Knowledge  
Object-Oriented Analysis, Design and Implementation: A new  
approach to software engineering  
Prentice Hall, New York, 1992
- /Henderson-Sellers 96/  
Henderson-Sellers B.  
Object-Oriented Metrics  
Measures of Complexity  
Prentice Hall, Upper Saddle River, 1996
- /Heuer 97/  
Heuer A.  
Objektorientierte Datenbanken  
Addison-Wesley, Bonn, 1997
- /Hofmann 98/  
Hofmann F.  
Grafische Benutzungsoberflächen  
Generierung aus OOA-Modellen  
Spektrum Akademischer Verlag, Heidelberg, 1998
- /Horstmann 97/  
Horstmann C.  
Practical Object-Oriented Development in C++ and Java  
John Wiley & Sons, New York, 1997
- /Hruschka 98/  
Hruschka P.  
Ein pragmatisches Vorgehensmodell für die UML  
Objekt Spektrum, 2/98, pp. 34–54
- /IBM 97/  
IBM Object-Oriented Technology Center  
Developing Object-Oriented Software  
An Experience-Based Approach  
Prentice Hall, Upper Saddle River, New Jersey, 1997
- /ISO 9241-10: 1996/  
Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirm-  
geräten  
Teil 10: Grundsätze der Dialoggestaltung  
Deutsche Fassung EN ISO 9241-10: 1996, Beuth Verlag, Berlin  
1996
- /Jacobson 92/  
Jacobson I., Christerson M., Jonsson P., Övergaard G.  
Object-Oriented Software Engineering – A Use Case Driven  
Approach  
Addison Wesley, Wokingham, 1992

## Literatur

/Jacobson 94/

Jacobson I., Ericson M., Jacobson A.  
The Object Advantage  
Business Process Reengineering with Object Technology  
Addison Wesley, Wokingham, 1994

/Jacobson 95/

Jacobson I.  
The Use-Case Construct in Object-Oriented Software Engineering  
in /Carroll 95/

/Khoshafian 90/

Khoshafian S., Abnous R.  
Object Orientation  
Concepts, Languages, Databases, User Interfaces  
John Wiley & Sons, New York, 1990

/Klute 98/

Klute R.  
JDBC in der Praxis  
Datenbankanwendungen im Intranet und Internet  
Addison Wesley, Bonn, 1998

/Krasner, Pope 88/

Krasner G., Pope S.  
A Cookbook for Using the Model-View-Controller User Interface  
Paradigm in Smalltalk-80  
Journal of Object Oriented Programming (JOOP), August/September  
1988, pp. 26–49

/Kruchten 99/

Kruchten P.  
The Rational Unified Process  
An Introduction  
Addison-Wesley, Reading, Massachusetts, 1999

/Kruglinki 97/




Kruglinki D.  
Inside Visual C++ Version 5  
Microsoft Press Deutschland, 1997

/Kruschinski 99/

Kruschinski V.  
Layoutgestaltung grafischer Benutzungsoberflächen  
Sprektrum Akademischer Verlag, 1999

/Larman 98/

Larman C.  
Applying UML and Patterns  
An Introduction to Object-Oriented Analysis and Design  
Prentice Hall, Upper Saddle River, 1998

- /Lee, Tepfenhart 97/  
Lee R., Tepfenhart W.  
UML and C++  
A Practical Guide to Object-Oriented Development  
Prentice Hall, Upper Saddle River, 1997
- /Martin, Odell 95/  
Martin J., Odell J.  
Object Oriented Methods  
A Foundation  
Prentice Hall, Englewood Cliffs, 1995
- /Martin et al. 97/  
Martin R.C.(ed.), Riehle D. (ed.), Buschmann F. (ed.) Vlissides J.  
Pattern Languages of Program Design 3  
Addison-Wesley, Reading Massachusetts, 1997
- /McMenamin, Palmer 88/  
McMenamin S., Palmer J.  
Strukturierte Systemanalyse  
Coedition von Hanser und Prentice Hall, London, 1988
- /Meyer 97/  
Meyer B.  
Object-Oriented Software Construction  
Prentice Hall, 1997
- /MS 95/  
The Windows Interface Guidelines for Software Design  
Microsoft Corporation, Redmont, 1995
- /Odell 94/  
Odell J.  
Six different kinds of composition  
Journal of Object-Oriented Programming, January 1994, pp. 10–15
- /ODMG/  
 [www.odmg.org](http://www.odmg.org)
- /OMA 97/  
A Discussion of the Object Management Architecture  
January 1997  
 [www.omg.org](http://www.omg.org)
- /OMG/  
 [www.omg.org](http://www.omg.org)
- /Oestereich 97/  
Oestereich B.  
Objektorientierte Softwareentwicklung mit der Unified Modeling  
Language  
Oldenburg Verlag, München, 1997
- /Page-Jones 88/  
Page-Jones M.  
Practical Guide to Structured Systems Design  
Prentice Hall, Englewood Cliffs, 1988



## Literatur

/Poet 97/

Poet C++ SDK  
Programmer's Guide  
Version 5.0  
Poet Software, Hamburg, 1997

/Pree 95/

Pree W.  
Design Pattern for Object-Oriented Software Development  
Addison-Wesley, Wokingham, England, 1995

/Rechenberg, Pomberger 97/

Rechenberg P., Pomberger G. (Hrsg)  
Informatik-Handbuch  
Carl Hanser Verlag, München, 1997

/Redlich 96/

Redlich J.-P.  
Corba 2.0  
Praktische Einführung für C++ und Java  
Addison-Wesley, Bonn, 1996

/Rubin 92/

Rubin K., Goldberg A.  
Object Behavior Analysis  
Communications of the ACM, September 1992, pp. 48–62

/Rumbaugh et al. 91/

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenzen W.  
Object-Oriented Modeling and Design  
Prentice Hall, Englewood Cliffs, 1991

/Rumbaugh et al. 93/

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenzen W.  
Objektorientiertes Modellieren und Entwerfen  
Coedition von Hanser-Verlag und Prentice Hall, 1993  
deutsche Übersetzung von /Rumbaugh et al. 91/

/Schäfer 94/



Schäfer S.  
Objektorientierte Entwurfsmethoden  
Verfahren zum objektorientierten Softwareentwurf im Überblick  
Addison Wesley, Bonn 1994

/Schmidberger et al. 97/

Schmidberger R. (Hrsg.), Schippert R., Kölle V., Urban U., Riemert S.,  
Thüly G.  
Visual C++ 5 & MFC im praktischen Einsatz  
An International Thomson Publishing Company, Bonn 1997

/Shlaer, Mellor 88/

Shlaer S., Mellor S.  
Object-Oriented Systems Analysis  
Modeling the World in Data  
Yourdon Press, Prentice Hall, Englewood Cliffs, 1988

- /Shlaer, Mellor 92/  
Shlaer S., Mellor S.  
Object Lifecycles  
Modeling the World in States  
Yourdon Press, Prentice Hall, Englewood Cliffs, 1992
- /Sigfried 96/  
Sigfried S.  
Understanding Object-Oriented Software Engineering  
IEEE Press, New York, 1996
- /Sims 94/  
Sims O.  
Business Objects  
Delivering Cooperative Objects for Client-Server  
McGraw-Hill Book Company, London, 1994
- /Stein 94/  
Stein W.  
Objektorientierte Analysemethoden  
Vergleich, Bewertung, Auswahl  
BI Wissenschaftsverlag, Mannheim, 1994
- /Stroustrup 98/  
Stroustrup B.  
Die C++ Programmiersprache  
Addison-Wesley, Bonn, 1998
- /Texel, Williams 97/  
Texel P., Williams C.  
Uses Cases combined with Booch/OMT/UML  
Process and Products  
Prentice Hall, Upper Saddle River, 1997
- /UML 96/  
Booch G., Rumbaugh J., Jacobson J.  
The Unified Modeling Language for Object-oriented Development, Version 0.91  
Rational Software Corporation, Santa Clara 1996  
 [www.rational.com/uml](http://www.rational.com/uml)
- /UML 97/  
Unified Modeling Language 1.1  
UML Summary  
Notation Guide  
UML Semantics  
Object Constraint Language Specification  
Rational Software Corporation, Santa Clara, September 1997  
 [www.rational.com/uml](http://www.rational.com/uml)

## Literatur

/UML 97a/

Unified Modeling Language 1.0  
Notation Guide  
UML Semantics  
Object Constraint Language Specification  
Rational Software Corporation, Santa Clara, Januar 1997  
[www.rational.com/uml](http://www.rational.com/uml)

/UML 99/

OMG Unified Modeling Language Specification (draft), Version  
1.3, March 1999  
[www.rational.com/uml](http://www.rational.com/uml)

/Vetter 87/

Vetter M.  
Aufbau betrieblicher Informationssysteme mittels konzeptionel-  
ler Datenmodellierung  
Teubner, Stuttgart, 1987

/Vlissides et al. 96/

Vlissides J. (ed.), Coplien J.O. (ed.), Kerth N.L. (ed.)  
Pattern Languages of Program Design 2  
Addison-Wesley, Reading Massachusetts, 1996

/Vossen 94/

Vossen G.  
Datenmodelle, Datenbanksprachen und Datenbank-Mangement-  
Systeme  
Addison-Wesley, Bonn, 1994

/Wirfs-Brock 90/

Wirfs-Brock R., Wilkerson B., Wiener L.  
Designing Object-Oriented Software  
Prentice Hall, Englewood Cliffs, 1990

/Yourdon 89/

Yourdon E.  
Modern Structured Analysis  
Yourdon Press, Prentice Hall, Englewood Cliffs, 1989

/Yourdon, Argila 96/

Yourdon E., Argila C.  
Case Studies in Object-Oriented Analysis and Design  
Prentice Hall, Upper Saddle River, 1996

/Yourdon et al. 95/

Yourdon E., Whitehead K., Thomann J., Opper K., Nevermann P.  
Mainstream Objects: An Analysis and Design Approach for Business  
Prentice Hall, Upper Saddle River, 1995

/Züllinghoven 98/

Züllinghoven H.  
Das objektorientierte Konstruktionshandbuch  
nach dem Werkzeug & Material-Ansatz  
dpunkt.verlag, Heidelberg, 1998



# Index

## A

abgeleitete Assoziation 46, 150  
abgeleitetes Attribut 27, 160, 236, **533**  
abstrakte Klasse 52, 266, 379, **533**  
abstrakte Operation 239, **533**  
abstrakter Datentyp (ADT) 23, 345, **533**  
abstrakter Geschäftsprozeß 68  
Abstraktion 236, **533**  
*activity diagram*, siehe Aktivitätsdiagramm  
ad hoc-Polymorphismus 258  
ADT-Hierarchie 346  
Aggregation 46, 156, **534**  
Akteur 63, 128, **534**  
Aktion 79, 81, 180, **534**  
Aktionsmenü 202  
Aktivität 79, 180, **534**  
Aktivitätsdiagramm 68, 84, 276, **534**  
    Notation 85  
Analyse 8, **534**  
    Ziel 8  
Analysemuster 90, **534**  
Analyseprozeß **534**  
Anfangswert 25  
Anfangszustand 80  
Anfragesprache 335, siehe auch OQL  
Analyse im Großen 123  
anonymer Zustand 79  
ANSI-SQL-Standard, Typen 310  
Anwendungsfall 69  
Anwendungsfenster 199  
*application coordinator*, siehe Fassade  
*application objects* 357  
Arbeitstechnik  
    OOA-Modell 125  
    Zustandsautomat 178  
Assoziation 40, 147, 244, **535**  
    abgeleitete 46, 150  
    bidirektionale 40  
    binäre 40  
    Checkliste 152  
    Dialogstruktur 211–213  
    einfache 46, 92, 95–97  
    geordnete 43  
    höherwertige 50  
    in C++ 249  
    in Java 250  
    mehrere 150  
    Merkmale 248  
    Notation 40 f

objektorientiertes Datenbanksystem 400  
objekt-relationale Abbildung 316–319  
    ODL 333  
    Realisieren 427 f, 442, 454  
    Realisierung mittels Klassen 246  
    Realisierung mittels Zeigern 245  
    reflexive 40, 42  
    Sichtbarkeit 248  
    ternäre 50  
    Verfeinern 378  
Assoziationsname 42, 149  
assoziative Klasse 45, 151, **535**  
    objekt-relationale Abbildung 316  
asynchrone Kommunikation 359  
Attribut 25, 157, 235, **535**  
    abgeleitetes 27, 160, 236, **533**  
    Abstraktionsniveau 159  
    Checkliste 161 f  
    C++ OML 341  
    IDL 359  
    in C++ 236  
    in Java 237  
    Notation 26, 235  
    ODL 332  
    Sichtbarkeit 235  
    Verfeinern 378  
Attributname 26, 158, 235  
Attributspezifikation 29 f, **535**  
    Notation 30  
Attributtyp 27  
Aufgaben X  
Aufgabenangemessenheit 196  
Aufzählungstyp 28  
Auswahlliste 217

## B

balancierter Makroprozeß 121, **535**  
*balancing* 184  
*Basic Object Adapter* (BOA) 356  
Basisoperation 34, 183  
Baugruppe (Muster) 92  
Baum 230  
Bedienung  
    funktionsorientierte 198  
    objektorientierte 198  
Bedienungsarten 197  
Begriffe  
    blau X  
    halbfett X

*behavior*, siehe Verhalten  
Benutzungsoberfläche 273  
Beobachter-Muster 295, 373, 387, **535**  
Beschleunigung, Menüauswahl 204  
bidirektionale Assoziation 40  
binäre Assoziation 40  
blaue Schrift X  
BOA (*Basic Object Adapter*) 356  
Booch, G. 3  
Botschaft 69, 71, **536**  
*Broker* 407, 411  
*business process* 69

## C

C++ 3, 232, 241  
    Assoziation 249  
    Attribut 236  
    *mapping*, IDL 361  
    Objekt/Klasse 232  
    Operation 241  
    Paket 269  
    Polymorphismus 260  
    Szenario 273  
    Vererbung 266  
    Zustandsautomat 276  
C++ ODL (*Object Definition Language*) 339  
    Assoziation 340  
    Attribut 332  
    Operation 334  
C++ OML (*Object Manipulation Language*) 339, 341  
    Assoziation 342  
    Attribut 341  
    Objektname 341  
    Operation 341  
C++ OQL (*Object Query Language*) 339, 345  
*Cache* 408, 411  
*check box* 217  
Checkliste 124  
    »Einfache« Assoziation, Aggregation, Komposition 157  
Assoziation 152  
Attribute 161 f  
Geschäftsprozesse 133 f  
Kardinalitäten 154  
Klassen 146 f  
Operationen 185  
Pakete 135  
Szenario 176 f  
Vererbung 164  
Zustandsautomaten 182 f

## Index

- class extension*, siehe Klassenextension  
*client*, siehe Klient  
Codd, E. F. 308  
*coercion* 258  
*collaboration diagram*, siehe Kollaborationsdiagramm  
*combo box* 217  
*command button* 216  
*commit* 412  
*common facilities* 357  
*Common Object Request Broker Architecture* (CORBA) 356  
*constraint inheritance* 266  
*Container-Klasse* 230, **536**  
CORBA (*Common Object Request Broker Architecture*) 356, **536**  
CRC-Karte 50 f, **536**
- D**
- dangling references* 455  
DAO (*Data Access Objects*) 413  
*Data Definition Language* (DDL), siehe Datendefinitionssprache  
*Data Dictionary* (DD) 304, 309, 312  
*Data Manipulation Language* (DML) 310  
*Database Broker-Muster* 407, 409  
Datenbank (DB) 304  
Datenbankmanagementsystem (DBMS) 304  
Datenbanksystem (DBS) 304, **536**  
Architektur 304  
Eigenschaften 304  
Einsatz 370  
objektorientiertes (ODBS) 305, 326, 371, 400, **546**  
objekt-relationale 345, **547**  
relationales (RDBS) 305 f, 370, 405  
daten-basierter Makroprozeß 124, **536**  
Datendefinitionssprache (DDL) 309, **536**  
Datenhaltung 370, 375  
Datenhaltungsschicht 372, **537**  
Datenhaltungs-Zugriffsschicht 376  
Datenmanipulationssprache 310, **537**  
Datenmodell 305, **537**  
objektorientiertes 326  
relationales 308  
Datenübertragung, synchrone 359  
DDL (*Data Definition Language*) 309, **536**  
Defekt 187  
*define-Operator* 339  
*derived association*, siehe abgeleitete Assoziation  
*derived attribute*, siehe abgeleitetes Attribut  
Destruktor **537**  
Dialog 195, **537**  
modaler 195  
nicht-modaler 195  
Dialogfenster 201, 420  
Dialoggestaltung  
Alternativen 197  
Norm 196  
Dialogmodus 195, **537**  
Dialogstruktur  
Assoziation 211–213  
Einfachvererbung 214  
Klasse 210–212  
DII (*Dynamic Invocation Interface*) 355  
Diploma 467  
Klassendiagramm 470  
OOA-Modell 468  
Pflichtenheft 467  
Prototyp der Benutzungsoberfläche 471  
Schemadefinition 477–480  
Tabellenstruktur 475  
direkte Manipulation 198  
Diskriminator 54  
DML (*Data Manipulation Language*), siehe Datenmanipulationssprache  
Dokumentanalyse 142, 147, 157  
*domain interfaces* 357  
Drehfeld 216  
Drei-Schichten-Architektur 12, 372, **537**  
flexible 372, 399, **539**  
strenge 372, 397, 403, **552**  
*drop-down combo box* 218  
Dropdown-Kombinationsfeld 218  
*drop down list box* 218  
Dropdown-Listenfeld 218  
*drop-down-Menü* 203 f, 210  
DSI (*Dynamic Skeleton Interface*) 355  
*Dynamic Invocation Interface* (DII) 355  
*Dynamic Skeleton Interface* (DSI) 355  
dynamisches Binden, siehe spätes Binden  
dynamisches Modell 10, 14, **538**  
3 Schritte zum 124
- E**
- edit control* 215  
Eigenschaftsmenü 202  
einfache Anfragen, OOQL 336  
einfache Assoziation 46, 92, 95–97  
Einfachvererbung 54, 262, **538**  
Dialogstruktur 214  
objekt-relationale Abbildung 318  
Eingabefeld 215  
elementare Klasse 28, 382, **538**  
Embedded SQL 413  
*encapsulation*, siehe Verkapselung  
Endzustand 80, 179  
Entitäts-Integrität 307  
*Entity-Relationship-Modell* 26 f, 43  
*entry-Aktion* 79, 180  
Entwicklungsprozeß, evolutionärer 122  
Entwurf 11, **538**  
Entwurfsmuster 282, **538**  
Entwurfssprinzip 371  
Entwurfsziel 12, 293, 371, 375  
Ereignis **538**  
implizites 81  
Notation 81  
Zustandsautomat 81, 181  
Ereignisliste (Geschäftsprozeß) 129  
Erfassungsfenster 210, 470, **538**  
Realisieren 422, 435  
Realisierung durch Klasse 383–385  
Erstellung, OOA-Modell 11  
Erwartungskonformität 196  
Erzeugungsmuster 283, 286 f, **538**  
evolutionärer Entwicklungsprozeß 122  
Exemplar 21  
siehe auch Objekt  
Exemplartyp (Muster) 91  
*exit-Aktion* 79, 180  
*extends-Beziehung* 67, 131  
*extends-Vererbung* 330  
*extent*, siehe Klassenextension  
externe Operation 33  
externes Schema 312
- F**
- Fabrikmethode-Muster 286, **539**  
*facade*, siehe Fassade  
Fachkonzeptsschicht 372, 377, **539**  
Entwurf 377–382  
Fachkonzept-Zugriffsschicht 375  
Farbe 222  
Fassade 293, 389  
Fassaden-Muster 293, **539**  
Fehlertoleranz 197  
Fenstertitel 437  
Fensterotypen 199, **539**  
*final state*, siehe Endzustand

- finite state machine*, siehe Zustandsautomat  
 flache Datei 401, **539**  
*flat file*, siehe flache Datei  
 flexible Drei-Schichten-Architektur 372, 399, **539**  
*fork diagram* 173  
 formale Inspektion 185–188, **539**  
*Framework* 284–286, 297, 406, **539**  
*Framework*-Eigenschaften 407  
 Fremdschlüssel 307  
*frozen* 29  
 Führungstext 220  
 funktionsorientierte Bedienung 198  
 Funktionsweise, objekt-orientiertes Datenbanksystem 331
- G**
- Gamma, E. 282  
 Geheimnisprinzip 19, 26, 236, **540**  
 Generalisierung 51, 162, **540**, siehe auch Vererbung  
*generalization*, siehe Vererbung  
 generische Funktion 199  
 generische Klasse 228, 403, **540**  
 generische Operation 310  
 geordnete Assoziation 43  
 Geschäftsprozeß 63, 127, **540**  
 abstrakter 68  
 Kategorie 65  
 konkreter 68  
 Notation 64  
 Geschäftsprozeßdiagramm 66, 128, **540**  
 Arztregister 107  
 Friseursalonverwaltung 111  
 Materialwirtschaft 103  
 Seminarorganisation 115  
 Geschäftsprozesse  
 Anzahl 132  
 Arztregister 107  
 Checkliste 133 f  
 Friseursalonverwaltung 109  
 Konsistenz 132  
 Materialwirtschaft 101  
 Seminarorganisation 114  
 Sonderfälle 130  
 Standardfälle 128  
 vs. Funktion 131  
 Geschäftsprozeßschablone 64, 127, **540**  
 Gestaltung, harmonische 222  
 Gestaltungsregeln  
 Farbe 222  
 Gruppierung 221  
 Hervorhebung 222  
 Menü 206  
 Gestaltungsregelwerk 194, **540**
- Gliederungsschema, Pflichtenheft 464  
 Gruppe (Muster) 97  
 Gruppenhistorie (Muster) 97  
 Gruppierung 221  
*guard condition*, siehe Wächter  
 GUI 194, **540**  
 GUI-Bibliothek 382  
 GUI-Schicht 372, 382, **540**  
 GUI-System 194, 370, **541**
- H**
- harmonische Gestaltung 222  
 hcd-Datei 448, 453  
 Hervorhebung 221  
 Historie (Muster) 96  
 historische Entwicklung 3 f  
 höherwertige Assoziation 50
- I**
- Identität, siehe Objektidentität  
 IDL (*Interface Definition Language*) 332, 353, 357, **541**  
 Attribut 359  
 C++ *mapping* 361  
 Modul 360  
 Operation 358  
 Schnittstelle 353  
 Signatur 358  
*Skeleton* 354  
*Stub* 353  
 Syntax 358  
 Typen 359  
 Vererbung 359  
*Implementation Repository* 354  
 Implementierungshierarchie 265  
 implizites Ereignis 81  
 Import-Beziehung zwischen Paketen 268  
*inclusion inheritance* 266  
 Index 313  
 indirekte Kommunikation 373  
 Individualisierbarkeit 197  
 indizierte Organisation 402  
*information hiding*, siehe Geheimnisprinzip  
*initial state*, siehe Anfangszustand  
 inklusionsbasierter Polymorphismus 259  
 Inspektion, formale 185–188  
 Inspektionsprotokoll 188  
 Inspektions-sitzung 187  
 Inspektorenteam 186  
*instance* 21  
 Instanz **541**  
 integrierte Qualitätssicherung 122  
 Interaktionsdiagramm 71, **541**
- Interaktionselement 215, 382, 416–418, **541**  
*Interface Definition Language* (IDL) 332, 353, 357 f  
*Interface Repository* 354  
*interface*, siehe Schnittstelle  
 interne Operation 33  
 Internet XII
- J**
- Jacobson, I. 3, 62, 75  
 Java 3, 233  
 Assoziation 250  
 Attribut 237  
 Objekt/Klasse 233  
 Operation 243  
 Paket 269  
 Polymorphismus 261  
 Szenario 273  
 Vererbung 267  
 Zustandsautomat 276  
 JDBC (*Java Database Connectivity*) 414, **541**
- K**
- Kann-Assoziation 41, 153  
 Kardinalität 41, 153, **541**  
 Checkliste 154  
 Notation 41  
 Kaskadenmenü 202  
 Kategorien  
 von Assoziationen 148  
 von Klassen 144  
 key 29  
 Klappliste 218  
 Klasse 21, 23, 142, 228, 329, **541**  
 abstrakte 52, 266, 379, **533**  
 assoziative 45, 151, **535**  
 Checkliste 146 f  
 Dialogstruktur 210–212  
 elementare 28, 382  
 generische 228, 403, **540**  
 Kurzbeschreibung 24  
 Notation 22  
 objekt-relationale Abbildung 306, 314  
 oder komplexes Attribut 158  
 oder Typ 23  
 Stereotyp 228  
 Klassenattribut 27, 160, 236, **542**  
 objekt-relationale Abbildung 313  
 klassenbasiertes Muster 283  
 Klassenbibliothek 283 f, 404, **542**  
 Topologien 283  
 Klassendiagramm 22, **542**  
 Arztregister 105 f  
 Diploma 470  
 Friseursalonverwaltung 110

## Index

Materialwirtschaft 100  
Seminarorganisation 113  
Klassenextension 329, 336, 448, **542**  
  und Vererbung 329  
Klassenhierarchie 51  
Klassenkonzept 3  
Klassenname 22, 145, 228  
Klassenoperation 31f, **542**  
Klassenvariable, siehe Klassenattribut  
Klient 352, **542**  
Kollaborationsdiagramm 71, 75, 77, 271, **542**  
  Notation 272  
Kollektion 328  
  Lesen einer 344  
Kombinationsfeld 217  
Kommunikation, asynchrone 359  
komplexer Attributtyp, objektrelationale Abbildung 315  
komplexes Objekt 29, **542**  
Komposition 47, 91, 93 f, 155 f, **543**  
  Realisierung 247  
Kompositum-Muster 289, **543**  
  konkreter Geschäftsprozeß 68  
  Konstruktor 339, **543**  
  Konstruktoperation 31  
Kontrollkästchen 217  
Konzept **543**  
Konzepte 5  
  objektorientierte 6  
Koordinator (Muster) 94

## L

*late binding*, siehe spätes Binden  
Lebenszyklus 81f, 177  
Lernförderlichkeit 197  
*link*, siehe Objektverbindung  
*list box* 217  
*list view control* 218  
Liste (Muster) 90  
Listenelement 218  
Listenfeld 217  
Listenfenster 210, 474, **543**  
  Realisieren 424, 438  
  Realisierung durch Klasse 385  
Listenoperation 183  
Listen-Typ 315  
Literal 327, 332, **543**  
Literaltyp 328  
logisches Enthaltensein 156  
logisches Schema 308  
*look and feel* 194  
Lupe X

## M

*Mailing*-Liste XII  
Makroprozeß 121f, **543**  
  balancierter 121, **535**  
  daten-basierter 124, **536**  
  szenario-basierter 124  
*mandatory* 29  
Materialisierung 407  
  Objektstrukturen 409  
  Optimierung 408  
MDI-Anwendung 196  
MDI-Fenster 434, 437  
Mehrfachvererbung 261, **543**  
Mehr-Schichten-Architektur 375, **543**  
mehrzeiliges Textfeld 216  
Menge aller Objekte, siehe Klassenextension  
Menü 202, **544**  
  Erstellen 421, 434  
  Gestaltungsregeln 206  
Menüauswahl, Beschleunigung 204  
Menübalken 203, 210  
Merkmal 23, 25  
  Assoziation 248  
*message*, siehe Botschaft  
Metaklasse **544**  
Methode 5, **544**  
methodisch didaktische  
  Elemente IX  
methodische Vorgehensweise 6, **544**  
Methodologie **544**  
Mitteilungsfenster 201  
modaler Dialog 195  
*Model-View*-Architektur 373  
*Model/View/Controller* (MVC) 373 f  
Modul, IDL 360  
*multi object* 273  
*multi-line edit field* 216  
Muß-Assoziation 41, 153  
Muster 90, 155, **544**  
  Beschreibung 90  
  grundlegende Elemente 282  
  klassenbasiertes 283  
  objektbasiertes 283  
  vs. *Framework* 285  
MVC (*Model/View/Controller*) 373 f, **544**

## N

Nachbedingung 65, 239, **545**  
Nachricht, siehe Botschaft  
*natural join*, siehe natürlicher Verband  
natürlicher Verbund 312  
Navigation 244, **545**  
  Notation 244  
Navigieren, OQL 337  
nicht-modaler Dialog 195

Norm zur Dialoggestaltung 196  
Normalform 308  
Normalisierung 406  
Notation 6, **545**  
  Attribut 235  
  Kollaborationsdiagramm 272  
  Navigation 245  
  Operation 239  
  Sequenzdiagramm 270  
  Sichtbarkeit 235  
  Tabellenstruktur 313  
*notebook* 219  
Notiz 91  
Nullwerte 339

## O

OA (*Object Adapter*) 355  
Oberklasse 51, **545**  
*Object Adapter* (OA) 355  
*Object Database Management Group* (ODMG) 4, 326  
*Object Definition Language* (ODL) 327, 332, 358, **547**  
*object diagram*, siehe Objektdiagramm  
*object factory* 21  
*object identity*, siehe Objektidentität  
*object implementation*, siehe Objekt-Implementierung  
*Object Interchange Format* (OIF) 327  
*Object Management Architecture* (OMA) 356  
*Object Management Group* (OMG) 4, 326, 356  
*object model*, siehe ODMG-Objektmodell  
*Object Query Language* (OQL) 327, 335  
*object reference*, siehe Objektreferenz  
*object relational mapping*, siehe objektrelationale Abbildung  
*Object Request Broker* (ORB) 252  
*object services* 357  
Objekt 18, 228, 327, **545**  
  externes 21  
  internes 21  
  komplexes **542**  
  selektieren 451  
  speichern 449, 460  
objektbasiertes Muster 283  
Objektdefinitionsprache, siehe ODL  
Objektdiagramm 19, **545**  
  Notation 49  
Objektidentität (OID) 20, 327, 332, **545**  
  Realisierung 327

- Objekt-Implementierung 353, **546**
  - Objektlinie 71
  - Objekt/Klasse
    - in C++ 232
    - in Java 233
  - Objektname 20, 328
    - C++ OML 341
  - Objektoperation 31
  - objektorientierte Analyse (OOA) 8, **546**
  - objektorientierte Bedienung 198
  - objektorientierte Konzepte 6
  - objektorientierte Softwareentwicklung 2, **546**
  - objektorientierter Entwurf (OOD) 11, **546**
  - objektorientiertes Datenbanksystem (ODBS) 305, 326, 371, 400, **546**
    - Funktionsweise 331
  - objektorientiertes Datenmodell 326
  - Objektreferenz 341, 353 f, 356, **546**
  - objekt-relationale Abbildung 306, 406, **546**
    - Assoziation 316–319
    - assoziative Klasse 318
    - Einfachvererbung 318
    - Klasse 306, 314
    - Klassenattribut 316
    - komplexer Attributtyp 315
  - objekt-relationales Datenbanksystem 345, **547**
  - Objekttyp 329
  - Objektverbindung 18, 43
  - Objektverwaltung 24, **547**
  - ODBC (*Open Database Connectivity*) 413, **547**
  - ODL (*Object Definition Language*) 327, 332, 358, **547**
    - Assoziation 333
  - ODMG (*Object Database Management Group*) 4, 326, **547**
  - ODMG-konform 327
  - ODMG-Objektmodell 327, **547**
  - ODMG-Standard (*object database standard* ODMG) 4, 326, **547**
    - Typen 328
  - ODMG-zertifiziert 327
  - OID, siehe Objektidentität
  - OID-Attribut 314
    - Realisierung 314
  - OIF (*Object Interchange Format*) 327
  - OMA (*Object Management Architecture*) 356, **547**
  - OMG (*Object Management Group*) 4, 326, 356, **548**
  - OMG-Objektmodell 356, **548**
  - ondemand*-Materialisierung 408, 410
  - ondemand*-Referenz 457, 459
  - OOA, siehe objektorientierte Analyse
  - OOA-Modell 9, 377, **548**
    - Arbeitstechnik 125
    - Diploma 468
    - Erstellung 11
    - häufige Fehler 126 f
  - OOD, siehe objektorientierter Entwurf
  - OOD-Modell 13, 377, **548**
  - Operation 30, 183, 238, **548**
    - abstrakte 239, **533**
    - Beschreibung 34, 184
    - Checkliste 185
    - C++ OML 341
    - externe 33
    - generische 310
    - IDL 358
    - in C++ 241
    - in Java 243
    - interne 33
    - Notation 31, 239
    - ODL 334
    - OQL 338
    - polymorphe **549**
    - Qualitätskriterien 184
    - Sichtbarkeit 238
    - Vererbung 184
    - Verfeinern 378
    - Zustandsautomat 180
  - Operationsarten 32 f
  - Operationsaufruf 352
    - statischer 354
    - siehe auch Botschaft
  - Operationsname 33, 184, 238
  - option button* 216
  - Optionsfeld 216
  - OQL (*Object Query Language*) 327, 335, **548**
    - einfache Anfragen 336
    - Navigieren 337
    - Operation 338
    - Zugriff in Struktur 336
  - ORB (*Object Request Broker*) 252, **549**
  - ordered*-Restriktion 248
  - or*-Restriktion 44
  - overloading* 240, 258, 264
  - overriding* 264
- P**
- package*, siehe Paket
  - Paket 55, 134, 267, **549**
    - Checkliste 135
    - in C++ 269
    - in Java 269
    - Notation 55
    - Sichtbarkeit 267
    - Stereotyp 268
  - Paketname 135
  - Paketvarianten 268
  - parameterized class*, siehe generische Klasse
  - parametrischer Polymorphismus 258
  - parametrisierte Klasse, siehe generische Klasse
  - pattern*, siehe Muster
  - permanente Verbindung 76
  - Persistence Framework* (PFW) 406
    - Aufgaben 412
  - Persistenz 396, **549**
  - Pflichtenheft 9, 464
    - Diploma 467
    - Gliederungsschema 464
  - PFW (*Persistence Framework*) 406
  - physisches Enthaltensein 156
  - pointer*-Referenz 454
  - polling* 373
  - polymorphe Operation **549**
  - Polymorphismus 256, 320, 380, **549**
    - in C++ 260
    - in Java 261
  - pop-up*-Menü 203
  - postcondition*, siehe Nachbedingung
  - precondition*, siehe Vorbedingung
  - Primärdialog 195
  - Primärfenster 199
  - Projektion 311
  - property*, siehe Merkmal
  - property sheet* 219
  - proprietäre Schnittstelle 413
  - Prototyp 194, **549**
    - der Benutzungsoberfläche 9 f, 12
    - Diploma 471
  - Proxy-Muster 291, **549**
  - Proxy, virtuelles 292 f
  - push button* 216
- Q**
- qualifier*, siehe Qualifikationsangabe
  - Qualifikationsangabe 45, **549**
  - Qualitätssicherung, integrierte 122
- R**
- radio button* 216
  - Realisierung, Objektidentität 327
  - Redefinition 53, 264
  - Re-Engineering* 142
  - referentielle Integrität 307, 343
  - Referenzzähler (*link count*) 455
  - reflexive Assoziation 40, 42



## Index

Register 219  
Regler 218  
Relation 306  
  siehe auch Tabelle  
relationales Datenbanksystem  
  (RDBS) 305 f, 370, 405, **550**  
relationales Datenmodell 308  
*request* 352, **550**  
Restriktion 27, 44, 148, 263  
  zeitliche 97 f  
*rollback* 412 f  
Rolle 42, 149, **550**  
Rollen (Muster) 95  
Rollenname 42, 149  
Rumbaugh, J. 3

**S**

Schablone, Geschäftsprozesse  
  64, 127  
Schablonenmethode-Muster  
  297, 407, **550**  
Schaltfläche 216  
Schemadeklaration 448, 453  
  Diploma 477–480  
Schiebereglер 218  
Schlüssel 306  
Schlüsselattribut 160, 306  
Schnittstelle 231, 239, 330,  
  **550**  
SDI-Anwendung 196  
Sekundärdialog 195  
Sekundärfenster 199  
Selbstbeschreibungsfähigkeit  
  196  
*select*-Befehl 311  
Selektion 311  
*sequence diagram*, siehe  
  Sequenzdiagramm  
Sequenzdiagramm 71, 77, 269,  
  **550**  
  Bedingungen und Wiederholungen 73  
  Konsistenz 74  
  Notation 270  
Serialisierung 405  
Server-Klasse 353, 362  
*shared aggregation* 47  
Sicht 312  
Sichtbarkeit **550**  
  Assoziation 248  
  Attribut 235  
  Notation 235  
  Operation 238  
  Paket 267  
Signatur 238, 334, **551**  
  IDL 358  
*Singleton*-Muster 287, 385, **551**  
*slider* 218  
Smalltalk-80 3  
*smart pointer* 292, 341  
*smart reference* 292 f  
Softwareentwicklung, verteilte  
  361–365

Software-Ergonomie 194, **551**  
*sorted*-Restriktion 248  
*spaghetti inheritance* 379  
spätes Binden 256, 338, **551**  
*specialization inheritance* 266  
Spezialisierung 51, 162  
Spezifikationshierarchie 265  
*spin box* 216  
*spin button* 216  
SQL (*Structured Query Language*) 309, **551**  
*stair diagram* 174  
Standardtyp 28  
*state*, siehe Zustand 18  
*state-chart diagram*, siehe  
  Zustandsdiagramm  
*state pattern*, siehe Zustandsmuster  
*static text* 220  
statischer Operationsaufruf  
  352  
statisches Modell 9, 14, 142,  
  **551**  
  6 Schritte zum 123  
Stereotyp 22, 33  
  Klasse 228  
  Paket 269  
*stereotype*, siehe Stereotyp  
Steuerbarkeit 196  
Steuerelement, siehe  
  Interaktionselement  
strenge Drei-Schichten-Architektur  
  372, 397, 403, **552**  
*strong ownership* 47, 155  
*Structured Query Language*  
  (SQL) 309  
Struktur 328  
Strukturansicht 220  
strukturierte Entwicklung 2  
Strukturmuster 283, 289, 291,  
  293, **552**  
Struktur-Typ 315, 328  
Stückliste (Muster) 93  
*style guide* 194  
*sub class*, siehe Unterklasse  
*subset*-Restriktion 44  
*substitution inheritance* 265  
*subtyping*-Vererbung 331  
*super class*, siehe Oberklasse  
Surrogat 291, 328  
synchrone Datenübertragung  
  359  
Systemanalyse, siehe Analyse  
Szenario 70, 170, 269, **552**  
  Checkliste 176 f  
  Dokumentation 170  
  in C++ 273  
  in Java 273  
  Kommunikation der Objekte  
  171  
  Konsistenz 175  
  Struktur 173  
szenario-basierter Makroprozeß  
  124, **552**

## T

*tab control* 219  
Tabelle 306, **552**  
Tabellenhierarchie 346  
Tabellenstruktur  
  Diploma 475  
  Notation 313  
*template*, siehe generische  
  Klasse  
temporäre Verbindung 76  
ternäre Assoziation 50  
*text box* 215  
Textfeld 215  
  mehrzeiliges 216  
*three-tier architecture*, siehe  
  Drei-Schichten-Architektur  
*top-down*-Vorgehensweise 142  
Transaktion 344, 410  
Transaktionszustände 410, 412  
Transition 80, 181, **552**  
*tree view control* 220  
Tupel 306  
*two-tier architecture*, siehe  
  Zwei-Schichten-Architektur  
Typ 23, **552**  
  ANSI-SQL-Standard 310  
  eines Attributs 27  
  IDL 359  
  ODMG-Standard 328  
Typkonvertierung 384, 396

## U

Überladen 258  
Überschreiben 53, 264, **553**  
UML (*Unified Modeling Language*) 3, **553**  
universeller Polymorphismus  
  258  
Unterfenster 200  
Unterklasse 51, **553**  
Unterzustand 83  
*use case* 62  
  in einem Informationssystem 62  
  in einem Unternehmen 62  
*use case diagram*, siehe  
  Geschäftsprozeßdiagramm  
*uses*-Beziehung 67, 131

## V

Vererbung 51, 162, 261, 330,  
  **553**  
  Checkliste 164  
  »gute« 163  
  IDL 359  
  in C++ 266  
  in Java 267  
  Notation 52  
  Realisieren 452  
  Verfeinern 377  
  Vor- und Nachteile 54

Vererbungsstruktur 51  
 Verhalten 18, **553**  
 Verhaltensmuster 283, 295,  
 297, **553**  
 Verkapselung 236, **553**  
 verteilte Softwareentwicklung  
 361–365  
 Verwaltungsoperation 34, 183  
 externe 34, 183  
*view*, siehe Sicht  
*Virtual Proxy*-Muster 408 f  
 virtuelle Funktion, siehe  
 polymorphe Operation  
 virtuelle Linien 225  
 virtuelles Proxy 292 f  
*visibility*, siehe Sichtbarkeit  
 Vorbedingung 65, 239, **553**

**W**

Wächter 81, 85  
 wechselnde Rollen (Muster) 95  
 Werkzeug **554**  
*whole-part*-Beziehung 47  
 Wiederverwendung 381  
*Windows style guide* 204

**Z**

zeitliche Restriktion 97 f  
 Zugriff in Struktur, OQL 336  
 Zusammenhänge X  
 Zustand 18, 79, 178, **554**  
 Verfeinerung 83  
 zusammengesetzter 83

Zustandsautomat 78, 177, 273,  
**554**  
 Arbeitstechnik 178  
 Checkliste 182 f  
 einfache Realisierung 274  
 Ereignis 81, 181  
 in C++ 276  
 in Java 276  
 Konsistenz 82, 181  
 Operation 178  
 Zustandsdiagramm 78, **554**  
 Zustandsmuster 274, **554**  
 Zustandsname 79, 181  
 Zustandsübergang 80  
 Zwei-Schichten-Architektur  
 371, **554**

# ***Das Standardwerk der Software-Technik!***

Umfassend in der Themenauswahl und -behandlung,  
revolutionär im Layout.

## **Helmut Balzert Lehrbuch der Software-Technik**

Band 1: Software-Entwicklung  
1025 Seiten, geb., inkl. CD-ROM  
DM 148,- / öS 1081,- / sFr 134,-

Band 2: Software-Management, Software-Qualitäts-  
sicherung, Unternehmensmodellierung  
792 Seiten, geb., inkl. CD-ROM  
DM 128,- / öS 935,- sFr 116,-

Band 1 + 2 im Paket  
DM 198,- / öS 1446,- / sFr 179,-

Arbeiten Sie mit allen Mitteln!

- Klassisches Buch
- Elektronisches Buch
- Multimediales Computer Based Training

»Dieses einzigartige Lehrbuch ist nicht nur Dozenten  
und Studenten der Informatik zu empfehlen. Als Nach-  
schlagewerk und zur Aktualisierung der Kenntnisse  
ist es für jeden professionellen Software-Entwickler eine  
Bereicherung.«

c't Magazin für Computertechnik

# **Vortragspräsentationen zu allen Lehreinheiten!**

Zu den Büchern aus der Reihe »Lehrbücher der Informatik« enthalten die separat erhältlichen CD-ROMs Vortragspräsentationen (in Farbe und Schwarz/Weiß) für den Einsatz in Vorlesungen, Schulungen und Seminaren.

Die PowerPoint-Präsentationen können auf Folien und Papier ausgedruckt werden. Die Käufer erhalten das Recht, die Präsentationen in eigene Vorträge einzubinden.

Heide Balzert

## **Präsentationen zur Objektmodellierung**

ISBN 3-8274-0545-9

Helmut Balzert

## **Präsentationen zur Software-Technik 1**

ISBN 3-8274-0183-6

Helmut Balzert

## **Präsentationen zur Software-Technik 2**

ISBN 3-8274-0310-3

Helmut Balzert

## **Präsentationen zu Grundlagen der Informatik**

ISBN 3-8274-0550-5

»Ich wünsche den Folien eine große Verbreitung und kann sie jedem Referenten, der Lehrinhalte aus dem Bereich Software-Technik vermitteln möchte, nur wärmstens empfehlen.«

Prof. Dr. Harald Reiterer, Universität Konstanz

Die CD-ROMs enthalten jeweils über 2500 PowerPoint 97-Folien in Farbe und Schwarz/Weiß, über 1000 mehrfarbige Graphiken und Tabellen sowie Animationen und Hyperlinks.

Systemvoraussetzungen:

Windows 95/98/NT; PowerPoint 97 (Viewer ist auf den CD-ROMs vorhanden), VGA-Grafikkarte mit 256 Farben; CD-ROM-Laufwerk.

Preis je CD-ROM DM 598,- / öS 4784,- / sFr 538,-

Mehr Informationen im WWW unter <http://www.spektrum-verlag.com>